Karlsruhe Institute of Technology

Parallel SAT-Solving for Product Configuration

Master Thesis of

Nils Merlin Ullmann

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	Prof. Dr. rer. nat. Carsten Sinz
	Prof. Dr. rer. nat. Peter Sanders
Advisors:	Msc. Marko Kleine Büning
	Dr. Tomáš Balyo

Time Period: 15th October 2019 – 14th April 2020

Statement of Authorship

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, April 14, 2020

Abstract

This thesis presents parallel algorithms to solve SAT problems in the domain of product configuration. During an interactive configuration process, a user selects components step-by-step to find a suitable configuration that fulfills a set of constraints. A configuration system can be used to guide the user through the process by validating the selections and providing feedback. Each validation of a user selection is formulated as a SAT problem. Furthermore, an optimization problem is identified to find solutions with the minimum amount of changes compared to the previous configuration. The necessity of reproducible solutions, despite using parallel algorithms, is considered and concepts to provide deterministic results are presented. Different parallel algorithms are proposed and compared. Experiments show that reasonable speedups are achieved by using multiple threads over the sequential counterpart.

Deutsche Zusammenfassung

Diese Arbeit präsentiert verschiedene parallele Algorithmen zur Lösung von SAT Problemen im Bereich der Produktkonfiguration. Während des interaktiven Konfigurationsprozesses wählt ein Nutzer schrittweise die Bestandteile eines Produkts aus, um eine Konfiguration zu erhalten die sowohl den Kundenwünschen entspricht als auch konsistent und produzierbar ist. Ein Produktkonfigurator kann verwendet werden, um den Nutzer durch den Konfigurationsprozess zu leiten, in dem die Kundenwünsche geprüft werden und bei Verletzungen der Beschränkungen Rückmeldung gegeben wird. Das Prüfen der einzelnen Kundenwünsche kann als SAT Problem formuliert werden. Zusätzlich wird ein Optimierungsproblem identifiziert, mit dem Ziel Lösungen zu finden, die minimale Änderungen an der vorherigen Konfiguration vornehmen. Die Anforderung reproduzierbare Konfigurationen unter Verwendung von parallelen Algorithmen zu erhalten wird untersucht und Konzepte werden vorgestellt, um deterministische Lösungen zu erhalten. Verschiedene parallele Algorithmen werden vorgestellt und miteinander verglichen. Erste Experimente zeigen, dass bei Nutzung von mehreren Rechenkernen, gute Beschleunigungen im Vergleich zum sequenziellen Algorithmus erreicht werden.

Contents

1. Introduction

1.1 Motivation

In recent years, customers demand for more personalized and customized products increases in industries that traditionally have used standardized products to reap the benefits of mass production. Companies try to adapt by offering personalized, configurable goods to meet the customers' demand which led to a paradigm called *Mass Customization*. The paradigm's objective is "to deliver products and services that best meet individual customers' needs with near mass production efficiency" ([TJM96]). This development leads to increasingly complex *configuration models* to describe product variants that are manufacturable and desirable to be offered to customers. Furthermore, configuration is applicable to various domains, such as simple product options for clothing or more complex ones like commercial trucks and airplanes. A key factor to enable this kind of mass customization are so called configuration systems ([FHBT14]). A configuration system (configurator) supports the process of configuring products that fulfill the user requirements as well as all constraints imposed by manufacturing limitations and manufacturer interests. In this work, the focus is on interactive configuration. Every time a user makes a new choice, the configurator provides feedback about the validity of the last step and other possible choices. For instance, choices that are no longer valid due to previous changes are grayed out.

Configuration systems offer great benefits for the sales process and customers, while technical challenges rise to enable configuration models with increasingly large knowledge bases. One method to express constraints of a configuration model is to use propositional logic ([SKK03]). Every valid configuration has to fulfill a set of propositional formulae. The configuration system's task is to find such an assignment for a given set of user requirements. Janota shows in [Jan08] that this problem can be expressed as a *Boolean Satisfiability Problem* (SAT) and consequently be solved by a SAT-solver. Essentially, the configuration model together with the user requirements are transformed into conjunctive normal form (CNF). If the CNF is satisfiable, then there exists at least one valid configuration for the underlying configuration model and the user's needs. The SAT problem was the first problem proven to be NP-complete ([Coo71]) which can lead to long calculation times. This is in stark contrast with the requirements of an interactive configuration system used in the sales process, in which a quick response time is expected to validate the latest user requirements against the configuration model. Nevertheless, research in SAT brought a lot of advances in last decades to improve the commonly used DPLL-algorithm.

A more recent trend and field of research is to parallelize the tree search performed by the DPLL algorithm. This development aims to utilize the rapid advances in multi-core computers and clusters that followed the diminishing returns of per core performance. Thus, increasing the number of processor cores is more common. Parallel SAT-solvers have been mainly studied on very hard SAT problems. Configuration systems tend to have different requirements that exceed the common SAT problem. Generally, the created problem instances are smaller and less complex, due to the a step-by-step configuration process. However, new problems are introduced. Firstly, in case that the customer's latest selection in the interactive process combined with the current configuration are unsatisfiable, the configuration system should return an alternative solution that minimizes the number of changes with regard to the current configuration. This problem can be modeled as an optimization problem, for example as a *Maximum Satisfiability Problem* (MaxSAT) or *Minimum-Cost Satisfiability Problem* (MinCostSAT). The underlying cost function evaluates the changes of every assignment compared to the current configuration. Furthermore, considering an optimization problem, more than one optimal solution can exist. For example, a user requirement can be fulfilled by several optimal configurations with an equal amount of changes. In this case, the user should have the choice which configuration is the subjectively optimal one. That means, it is not sufficient to solely return the first optimal solution. Instead, all other alternative configurations with minimal costs are demanded. Lastly, a configuration system requires fully deterministic behavior. In this context, determinism describes the reproducibility of a configuration given a user change. Therefore, the entire configuration process is reproducible.

1.2 Goals of this Thesis

In this thesis, the goal is to conceptualize and develop multiple parallel algorithms that are capable of solving SAT and MinCostSAT instances that are prevalent in product configuration. The developed algorithms are evaluated using the *CAS Configurator Merlin* ([CAS20]), which is specialized on solving problems from interactive configuration. Based on the stated goal, the following concrete objectives have been identified.

1. *Scalability*: developed algorithms should scale well with increasing numbers of processing units. Scaling is measured by the speedup through additional processor cores. Moreover, the resource consumption is considered to evaluate the applicability for industry cases. Focus is set on applications running on shared-memory systems.
2. *Completeness*: computing all optimal configurations that the sequential version returns, given that multiple optimal solutions exist. The requested maximal number is typically defined between two to ten alternative solutions.
3. *Determinism*: achieving reproducible results for the user. That means, during the interactive configuration process, every repetition of a user change has to result in the same n assignments. Thus, repeating two equal configuration processes lead to the same alternative configurations.

1.3 Contributions

The main contribution of this work is the development of parallel algorithms for problem instances created during interactive configuration processes. These instances are described by calculating the optimal valid solutions for a given user change, start-configuration, and Boolean formula. Several parallel algorithms are presented that fulfill the completeness and optimality criteria for assignments required by configuration systems. Experimental results show that the presented approaches can achieve significant improvements in response time

by using multiple processor cores. Furthermore the approaches fulfill the completeness, optimality, and determinism demand. Especially the latter has not been widely researched in this domain, because many applications do not rely on reproducible results.

1.4 Related Work

SAT and Product Configuation

The development of the DPLL algorithm in 1962 by M. Davis, H. Putnam, G. Logemann and D.W. Loveland lays the foundation for most modern SAT solvers ([DLL62]). Due to this effective method, different domains try to model their problems as SAT to utilize the DPLL algorithm. Examples are formal verification ([BCRZ99]), planning problems ([EMW97]), and scheduling problems ([GHM+12]). In the following decades, many improvements have been proposed. Important ones are the development of sophisticated branching heuristics (for example [LGZ+15]), clause learning ([MMZ+01], and non-chronological backtracking. Due to its broad applicability, SAT solving is an active area of academic research and software development. Furthermore, it is propelled by yearly SAT competitions with various competition tracks (for instance [BHJ17]).
Nevertheless, modeling SAT problems has clear limitations as well. By expecting "satisfiable" or "unsatisfiable" as a result, SAT is a decision problem that tries to find a valid assignment. Besides being valid, the assignment does not consider any quantitative evaluation, thus it cannot be used for optimization problems ([L+04]). As alternatives, the SAT extensions MaxSAT and MinCostSAT were introduced, with various implementations such as [BF98], [LM09], [L+04], and [FM06].

SAT and the two extensions have been applied to the field of product configuration as well. The authors of [SKK01] and [SKK03] present a method to check the general consistency of a product data base, by converting it into formulae of propositional logic. Their publications show the usage of a SAT solver in a real product configuration system for the automotive industry. A broader discussion of the applicability of SAT solvers for configuration systems is given in [Jan08], in which scalability is emphasized as a potential benefit. Early research on concepts focusing on the configuration process has been presented by Sabin and Weigel in [SW98]. A good overview of the *interactive configuration* process, in which a user specifies his requirements step-by-step with validation and feedback in between, is given in [JBGMS10]. Different solving techniques for this process have been proposed. Batory and Freuder et al. present in [Bat05] and [FLW01] algorithms for a lazy approach. In this context lazy means that no precompilation is required, because all computations are performed during the configuration process. Non-lazy approaches mainly use binary-decision-diagrams (BDD) instead of SAT solvers, examples are [HSJ+04] and [AHP10]. Furthermore, Janota discusses many optimization techniques and algorithms to model the interactive configuration process lazily with the help of SAT ([Jan10]. Additionally, he elaborates methods to improve the transparency of a configuration system. This is achieved by providing algorithms to generate comprehensible explanations using resolution trees and completing partial configurations.

Parallel Search

In SAT, the number of existing parallel solvers has increased significantly in the last years due to the growing interest in developing algorithms that utilize multi-core architectures. Currently, most parallel SAT solvers are based on two main approaches: *divide-and-conquer* (also called search space splitting) and *parallel portfolios*.

Early parallel SAT solvers are based on the divide-and-conquer approach. This method divides the search space into smaller subproblems. One of the first solvers to implement

this idea is PSATO ([ZBH96]), a solver designed for networks of workstations. It is based on the sequential solver SATO and divides the search space by using *guiding-paths*, which describe a path from the root to a specific node within the search tree. Organization is managed according to the "master-slave model", in which the master partitions the problem and load balances the subproblems to be processed by the workers. In [JLU05], the PSatz solver (based on Satz) is described which utilizes work-stealing as a dynamic form of work distribution. In contrast to PSATO and PSatz, the parallel solver PaSAT presented in [SBK01] introduces a form of clause sharing (called lemma exchange) for conflict derived clauses. The mentioned solvers run on local networks. GridSAT ([CW03]), a parallel SAT solver based on the sequential solver zChaff, is designed to be used on a distributed network. Similar to earlier work, it employs a "master-client" model and partitions the search space using guiding paths. Additionally, it implements a method to exchange learned clauses, up to a predefined number of literals, across all clients on the grid. In contrast to earlier distributed systems, the parallel SAT solver MiraXT is designed for shared-memory multiprocessor systems ([LSB07]). In their work, all threads use a shared-memory clause database and use a locking mechanism to update the clauses. Furthermore, PaMiraXT uses a master-client model for workstation clusters in which the clients are MiraXT solvers ([SLB10]).
With the goal of improving the scalability of parallel SAT solvers, the paradigm *cube-and-conquer* has been introduced. It is a two-phase approach that partitions the original problem into many subproblems (cubes) which are subsequently solved in parallel. At first, the partitioning is performed using lookahead solvers (e.g. [Hv09]) that try to reduce the complexity of the remaining Boolean formulae. These cubes are solved by a CDCL solver. Hence, the first phase utilizes global heuristics that are good at reducing the problem size and the second phase employs local heuristics that are designed to find an assignment quickly. An early example of a cube-and-conquer SAT solver is [HKWB11]. A cube-and-conquer solver designed for grids is presented in [BKB$^+$13].

Parallel portfolios, popularized by Hamadi et al. with the solver ManySAT ([HJS10]), differ by starting several SAT solvers with different configurations in parallel. The solvers compete to find a solution to the input problem and terminate as soon as one has been found. They showed that portfolio-based solvers can perform better on industrial cases than divide-and-conquer approaches, by reducing the sensitivity to parameter tuning. The used configuration settings are manually chosen to achieve complementary strategies. An automated approach for selecting parameters is shown in [XHLB10]. A more recent portfolio solver is Plingeling ([Bie13]). Many early portfolio solvers run on single multi-processor computers. The scalability of portfolio-based solvers is analyzed in HordeSat ([BSS15]). The distributed solver is intended to be used on clusters with thousands of processors, with the objective to solve very hard SAT instances. Their results show good speedups for up to 2048 processor cores.

This thesis is focused on the combination of SAT problems and optimization problems. Hence, some publications about parallel best-first search algorithms are summarized. Considering parallel best-first algorithms, many have been developed for A* search. Approaches for parallel A* can be separated into two major categories, depending on the management of the OPEN list.
The first approach is centralized parallel A*. In [IS86], Irani and Shih introduced a parallel algorithm that works on a shared central OPEN list. Additionally, they adapted the termination criteria to ensure optimality while creating little search overhead. A more recent example is [PLK14]. Their work is concentrated on problems with slow node expansion. Furthermore, they added a method to avoid re-expanding nodes during parallel A* search. To remove potential bottlenecks on the shared OPEN list, algorithms that use the so called decentralized approach have been developed. The algorithm PRA* (Parallel Retracting

A*) assigns an OPEN list to each processor ([EHMN95]). Every generated node is mapped to a processor using a hash function. Thus, the load balancing depends on the used hash function. The passing of nodes to other processors is done by locking the OPEN lists. Following work mainly concentrated on developing sophisticated hash functions to reduce sources of overhead, for instance in [KKW11] and [JF16]. Many approaches use domain specific hashing to exploit the underlying problem structure. For example, [JF16] designed a hash function that uses the structure of the problem instance, which are based on different sized puzzles.

Deterministic Search

Parallel SAT solvers provide a speed-up in many cases and improve the current state-of-the-art. However, most parallel SAT solvers introduce non-deterministic behavior. Consequently, they cannot be used in applications that rely on reproducible results. This problem has been acknowledged by Hamadi et al. in [HJPS11]. They proposed the first deterministic parallel SAT solver, which is based on the portfolio-based ManySAT solver. To ensure reproducibility, the solver only exchanges information (e.g. learned clauses) at specific points. These synchronization points occur after a number of conflicts during the search in each thread. Moreover, Hamadi et al. implemented static as well as dynamic strategies to define synchronization points. Their results show that the deterministic solver can achieve competitive results to its non-deterministic counterpart. In [MML12], the concept of synchronization points is applied to develop the first deterministic parallel MaxSAT solver. Similarly to the SAT solver by [HJPS11], they present static and dynamic synchronization strategies to reduce overhead.

1.5 Outline

In Chapter 2, all necessary definitions and concepts required for this thesis are presented. These are primarily the introduction into propositional logic and the well-known SAT problem as well as extensions of it, such as MinCostSAT. Furthermore, different parallel algorithms for SAT are introduced. Finally, brief descriptions of the domain product configuration as well as the usage of a configuration system are given. Subsequently, Chapter 3 presents a detailed problem definition for this thesis, to motivate the usage of parallel algorithms for the interactive configuration process. Chapter 4 describes the developed concepts for parallel algorithms to solve problem instances for the defined configuration task. Different approaches are discussed, compared, and analyzed. Additionally, strategies to ensure deterministic results for the user are presented, a key requirement for configuration systems. The different algorithms are evaluated in Chapter 5, using theoretical problems like Random 3-SAT and real industry cases. The focus is on computation time and memory efficiency for problems with varying complexity. Finally, Chapter 6 provides a summary of the presented work as well as an outlook for potential future work.

2. Preliminaries

This chapter describes the necessary theoretical background for this work. Firstly, fundamental elements of propositional logic are introduced. Secondly, the satisfiability problem (SAT) is presented, as well as a common algorithm to solve it, the DPLL algorithm. Thirdly, the SAT related problems MaxSAT and MinCostSAT are shown. Afterwards, several approaches of parallel algorithms for SAT-Solvers are presented. Finally, the domain product configuration and the interactive configuration process are introduced.

2.1 Propositional Logic

The basis of propositional logic was introduced by George Boole in his book "An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities" [Boo54].

This section about propositional logic is primarily based on the second chapter of [BA12].

2.1.1 Syntax and Semantics

In propositional logic, truth values are assigned to statements called *atomic propositions*. Therefore they can either be assigned *true* (t) or *false* (f). These atomic propositions (variables) can be used to construct more complex *propositional formulae* using *logical operators* (Boolean operators). The logical operators are the following:

- Conjunction: $\wedge$
- Disjunction: $\vee$
- Negation: $\neg$
- Implication: $\Rightarrow$
- Equivalence: $\Leftrightarrow$

Interpretation of a Formula

The truth value of a propositional formula F can be derived from the truth values of the atomic propositions that are part of F.

Definition 2.1 (Assignment)**.** *Let F be a propositional formula and VAR_F be the set of variables appearing in F. An assignment is a function $\beta : VAR_F \to \{t, f\}$ assigning a truth value to the variables of VAR_F. The assignment is complete for F if the function β is defined for all variables in VAR_F. Otherwise it is a partial assignment for F.*

Truth Tables

Truth tables can be used to provide the semantics of a propositional formula by giving a truth value for every possible assignment. The idea was first shown in [Pos21]. A truth table with n atomic propositions and one propositional formula consists of $n+1$ columns and 2^n rows (each atomic proposition can have the value t or f). Thus, the semantics of complex formulae using the above mentioned logical operators can be displayed. With p and q being two atomic propositions, Table 2.1 shows all possible interpretations of different formulae.

p	q	p ∧ q	p ∨ q	¬ p	p ⇒ q	p ⇔ q
t	t	t	t	f	t	t
t	f	f	t	f	f	f
f	t	f	t	t	t	f
f	f	f	f	t	t	t

Table 2.1: A truth table of different formulae for the five introduced logical operators in Section 2.1.1. The two atomic proposition p and q result in 2^2 interpretations (rows).

2.2 The Satisfiability Problem (SAT)

Propositional formulae can be arbitrarily complex by combining atomic propositions using logical connectives. As such, they can be classified depending on whether it is possible to find an assignment for which the propositional formula evaluates to *true*.

Definition 2.2. *Let F be a propositional formula, F is called ...*

- *satisfiable* if an assignment $\beta : VAR_F \to \{t, f\}$ exists that lets F evaluate to *true*. A satisfying assignment is called a *model* for F.
- *unsatisfiable* if formula F evaluates to *false* for all assignments $\beta : VAR_F \to \{t, f\}$.
- *valid* if every possible assignment $\beta : VAR_F \to \{t, f\}$ is a model of F. A valid propositional formula is called *tautology*.
- *falsifiable* if it is not valid. Hence an assignment exists, for which F evaluates to *false*.

The Boolean Satisfiability Problem (SAT) describes the problem that determines the satisfiability of a propositional formula (Boolean formula). In detail, it assesses whether an assignment exists that lets the Boolean formula evaluate to *true*. A concrete SAT problem is also called *SAT instance*.

Example 1. A satisfiable SAT instance is given by the propositional formula:

$$F = (p \vee q) \wedge (\neg p \vee q)$$

F is satisfiable with $\beta(p) = f$ and $\beta(q) = t$, leading to the variable assignment $B = \{\neg p, q\}$

Example 2. A unsatisfiable SAT instance is given by the propositional formula:

$$F = (p \vee q) \wedge (\neg p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee \neg q)$$

All four possible assignments evaluate to *false*.

Significantly more complex SAT instances are formulated in various domains, for instance formal verification ([BCRZ99]), planning problems ([EMW97]), scheduling problems but also within product configuration ([Jan08]). In the following sections, formulation and solving techniques for SAT problems are elaborated.

2.2.1 Normal Forms

In SAT-problems, Boolean formulae are expressed with *literals*. A literal is a Boolean variable (positive literal) or a negation of a Boolean variable (negative literal). A *clause* is defined as a disjunction ($\vee$) of literals:

$$c = l_1 \vee l_2 \ ... \vee l_n \tag{2.1}$$

A positive literal is satisfied if it is assigned with the truth value *true*, a negative literal if it is assigned with the truth value *false*. A positive literal is unsatisfied if it is assigned with the truth value *false*, a negative literal if it is assigned with the truth value *true*. Otherwise, if a literal has no truth value assigned, it is defined as *unassigned*.
The Conjunctive Normal Form (CNF) represents a Boolean formula as a conjunction ($\wedge$) of clauses:

$$F_{cnf} = c_1 \wedge c_2 \wedge c_3 \tag{2.2}$$

A clause is satisfied, if at least one of its literals is satisfied. A clause is unsatisfied, if all of its literals are unsatisfied. Subsequently, the Boolean formula F_{cnf} in CNF is satisfied, if all its clauses are satisfied. In case a clause consists of only one unassigned literal, it is called a *unit clause*. Additionally, every formula in propositional logic can be transformed into CNF ([BA12, p.75]).
In contrast to CNF, the Disjunctive Normal Form (DNF) describes a Boolean formula as a disjunction of conjunctions of literals (terms):

$$F_{dnf} = t_1 \vee t_2 \vee t_3 \tag{2.3}$$

2.2.2 DPLL Algorithm

A naive approach to determine the satisfiability of a Boolean formula F is to construct a truth table. Each row represents a possible assignment, thus 2^n rows exist with n being the number of variables in F. This leads to an exponential number of calculations. Furthermore, SAT was the first problem proven to be NP-complete [Coo71]. Hence no algorithm exists that is capable of solving the problem in polynomial time, assuming that $P \neq NP$.
Most current state-of-the-art SAT solvers use the DPLL algorithm, introduced by Martin Davis, Hilary Putnam, George Logemann and Donald Loveland in [DLL62]. The algorithm is a depth-first search based algorithm with several enhancements. Given a Boolean formula F_{cnf} in CNF, partial assignments are formed and successively extended by adding literals during the search. β describes the current partial assignment and F_{cnf} the remaining set of clauses. Thus, at the beginning β is an empty set and F_{cnf} represents the initial Boolean formula. At every step of the algorithm, a partial assignment β is extended by a literal. Consequently, F_{cnf} is simplified to a problem that does not contain the literal used for extension.

A central part of the DPLL algorithm is *unit propagation*. If F_{cnf} contains a unit clause with the literal l, then the unassigned literal l must be assigned *true* and is added to β. Additionally, l is propagated. Every clause in F_{cnf} containing l is satisfied and thus removed from the Boolean formula. Every clause in F_{cnf} that contains $\neg l$ is simplified by

removing $\neg l$ from the clause. This reduction can create new unit clauses with unassigned literals which are subsequently propagated and added to β. This cascade stops when no unit clause is present (cf. [NOT06]).

Afterwards, one of the following states is reached:

1. The resulting set of clauses in F_{cnf} is empty $\emptyset$. Therefore, a satisfying assignment β has been found. All unassigned propositional variables can be assigned arbitrarily.
2. The resulting set of clauses in F_{cnf} contains empty clauses (a clause in which all literals have been falsified). These clauses cannot evaluate to *true*, thus the entire Boolean formula cannot be satisfied. The partial assignment leads to a contradiction.
3. To continue, the algorithm requires a new unit clause. This unit clause is formed by using a decision literal l_d. A propositional variable from F_{cnf} and a respective truth value is chosen, which is represented by the decision literal l_d. Afterwards, two recursive calls with different parameters are performed. One call uses the parameters $(\beta, F_{cnf} \cup \{l_d\})$, the second one $(\beta, F_{cnf} \cup \neg\{l_d\})$. This step can be interpreted as a branching point within the search tree.

The Pseudocode of the DPLL algorithm is presented in Algorithm 2.1 and Procedure 2.2.

Algorithm 2.1: DPLL Algorithm

Input: partial assignment β, Boolean Formula F_{cnf}

```
1  if ∅ ∈ F_cnf then
2      return UNSAT
3  else if F_cnf = ∅ then
4      return β
5  β, F_cnf := unitPropagation(β, F_cnf)
6  l_d := select unassigned variable in F_cnf
7  if DPLL(β, F_cnf ∪ {l_d}) ≠ UNSAT then
8      return DPLL(β, F_cnf ∪ {l_d})
9  else if DPLL(β, F_cnf ∪ {¬l_d}) ≠ UNSAT then
10     return DPLL(β, F_cnf ∪ {¬l_d})
11 else
12     return UNSAT
```

Procedure 2.2: UnitPropagation

Input: partial assignment β, Boolean Formula F_{cnf}

```
1  while unit clause {l} exists in F_cnf do
2      β := β ∪ l
3      forall clause c ∈ F_cnf do
4          if l ∈ c then
5              remove c from F_cnf
6          else if ¬l ∈ c then
7              remove l from c
8  return β, F_cnf
```

Since the DPLL is essentially a depth-first search of partial assignments, the algorithm is *sound* and *complete*. That means, it always returns the correct answer for all Boolean

formulae. The described enhancements only cause certain paths of the search tree to be pruned by detecting not satisfying assignments ([NOT06]). The worst-case performance compared to the naive approach of truth tables remains $O(2^n)$ with n being the number of variables, due to exponential growth of nodes in the binary search tree of depth n. Nevertheless, the average performance is often much better due to techniques such as unit propagation. Thus, the maximal tree depth is usually smaller than n.

2.2.3 Conflict Driven Clause Learning Solvers

One major improvement of DPLL-based SAT solvers are conflict driven clause learning (CDCL) solvers. Every time the DPLL algorithm encounters an empty clause, the conflict is analyzed to derive a clause that can be added to the Boolean formula. The goal is to learn from a contradiction to avoid repeating it. Hence, the incorporated clauses are called *learned clauses*. Additionally, non-chronological backtracking can be used. After a conflict, an appropriate level to backtrack is derived using the learned clause instead of backtracking only one level.
The idea of clause learning was first introduced by the GRASP solver in [MSS99]. Subsequently, many solvers incorporated the concept and improved it, for instance by the Chaff solver in [MMZ$^+$01]. A good description of CDCL solvers is given in [MSLM09], thus the following provides only a brief overview of the general concepts.

Due to the recursive nature of the DPLL algorithm, each truth assignment can be associated with a *decision level*. Decision level 0 is given by assignments deduced from the input formula, without selecting a branching variable. Decision level n describes the assignment implied by the decision in the n-th recursive call of DPLL. Considering a search tree representation, the n-th decision describes a node on the n-th level. A variable x_i is said to be *implied*, if it is assigned a truth value by unit propagation after a clause became unit. This unit clause is called the *antecedent* of the assignment ([MSLM09]). Thus, decision and unassigned variables do not have an antecedent.
The relation between assigned variables and their antecedents form a directed acyclic graph $I = (V_I, E_I)$, in [SS96] referred to as an *implication graph*. An example is given in the following.

Example 3. Implication graph with a conflict based on [MSLM09]. Given is the Boolean formula with six clauses and eight variables:

$$\begin{aligned} F_{cnf} &= c_1 \wedge c_2 \wedge c_3 \wedge c_4 \wedge c_5 \wedge c_6 \\ &= (x_1 \vee x_7 \vee \neg x_2) \wedge (x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge \\ &\quad (\neg x_4 \vee \neg x_5) \wedge (x_8 \vee \neg x_4 \vee \neg x_6) \wedge (x_5 \vee x_6) \end{aligned} \tag{2.4}$$

Furthermore, current decision assignments are given with their respective decision level: $x_8 = f(2)$, $x_7 = f(3)$, and $x_1 = f(5)$. These assignments yield a conflict, because clause $(x_5 \vee x_6)$ becomes unsatisfied after unit propagation. The corresponding implication graph is shown in Figure 2.1.

The shown implication graph can be used to deduce different learned clauses to prevent the conflict from appearing again ([MSLM09]). Considering Example 3 and Figure 2.1, different cuts can be performed, each resulting in a learned clause. For every cut, all decision variables are on one side of the cut. The goal is to reduce the size of learned clauses. One concept to find short learned clauses is the usage of *Unique Implication Points* (UIPs), shown in [MSS99]. A UIP is a node of the implication graph, such that each path from the decision variable at the current decision level to the conflict contains that node.

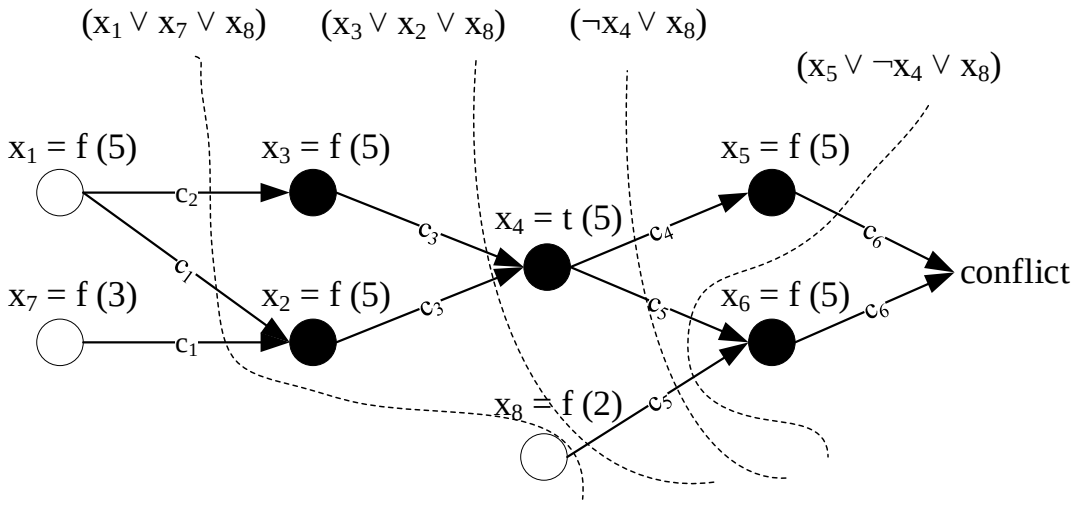

Figure 2.1: An implication graph based on [MSLM09] for Example 3. Numbers in brackets denote the decision levels. White nodes indicate decision literals. Due to the decision assignments, the clause $(x_5 \vee x_6)$ becomes unsatisfied, resulting in a conflict. Four different cuts can be performed to deduce different learned clauses.

One learning strategy is to make the cut at the *first UIP*. In Figure 2.1, the first UIP is x_4 with the corresponding cut to its right. Therefore, the learned clause $(\neg x_4 \vee x_8)$ can be derived. This clause is the minimal-length clause that can be deduced from the conflict. The authors of [ZMMM01] give in-depth descriptions and algorithms to find UIPs.

2.3 Extensions to SAT: MaxSAT, MinCostSAT

SAT only searches for an assignment that satisfies the Boolean formula. Optimality with regard to the assignment is not considered. Two approaches that introduce an optimization function are the Maximum Satisfiablity Problem (MaxSAT) and the Minimum-Cost Satisfiability Problem (MinCostSAT). Both problems extend SAT by incorporating a cost function that evaluates assignments. Thus, they are optimization versions (maximization/minimization problem) of SAT.

The better known MaxSAT problem consists in finding an assignment that maximizes the number of satisfied clauses of a Boolean formula F_{cnf} ([LM09]).Therefore, MaxSAT can be used for unsatisfiable formulae. Every clause in F_{cnf} has a weight. A solution is evaluated by the sum of weights of all satisfied clauses. In pure MaxSAT, all weights are equal. This is adapted in *weighted* MaxSAT through the usage of specific weights for individual clauses.

Definition 2.3 (MaxSAT)**.** *Let $w_1 \dots w_n \in \mathbb{N}_{\geq 1}$ be weights for the clauses $c_1 \dots c_n$. The evaluation of an assignment for the weighted MaxSAT problem is given by:*

$$\sum_{c_i \in F_{cnf}} \begin{cases} \text{c is true for assignment} \rightarrow w_{c_i} \\ else \qquad\qquad\qquad\qquad\quad \rightarrow 0 \end{cases} \tag{2.5}$$

MinCostSAT on the other hand assigns non-negative costs to each variable to quantify an assignment. It is formalized in Definition 2.4, adapted from [L$^+$04].

Definition 2.4 (MinCostSAT)**.** *A MinCostSAT instance is given by a Boolean formula F_{cnf} with a set of n variables VAR_F, each variable $x_i \in \{0, 1\}$ having a non-negative cost*

c_i for $i \in [1, n]$. The problem is to find a variable assignment that satisfies F and minimizes the costs:

$$\sum_{i=1}^{n} c_i x_i \tag{2.6}$$

Consequently, only variables that are assigned with the value *true* ($x_i = 1$) increase the cost of an assignment. The authors of [L+04] show that the transformation between MaxSAT and MinCostSAT problems can be performed by adding variables and clauses to the respective formulae. Since the problems can be transformed, only MinCostSAT is considered in the following. The focus in this thesis is on instances from the domain of product configuration which can be best described by using MinCostSAT.

DPLL for MinCostSAT

The introduced DPLL algorithm in Section 2.2.2 can be extended to solve the optimization problem present in MinCostSAT instances. Costs for every variable truth assignment are accumulated, i.e. during unit propagation. At every branching point a decision variable is chosen, partial assignments are calculated, and the assignment with the lower costs is used. The extended DPLL algorithm is shown in Algorithm 2.3 and Procedure 2.4. It is a recursive branch-and-bound algorithm, adapted from [FM06] and [L+04].

Algorithm 2.3: DPLL Algorithm for MinCostSAT

Input: partial assignment β, Boolean Formula F_{cnf}, current-costs *costs*, upper bound UB

```
1  β, F_cnf, costs := UnitPropagationWithCosts(β, F_cnf, costs)
2  if ∅ ∈ F_cnf | costs ⩾ UB then
3      return
4  else if F_cnf = ∅ & costs < UB then
5      UB := costs
6      return β, costs
7  l_d := select unassigned variable in F_cnf
8  β, costsPos := DPLL(β, F_cnf ∪ {l_d}, UB)
9  β, costsNeg := DPLL(β, F_cnf ∪ {¬l_d}, UB)
10 if costsPos < costsNeg then
11     return β, costsPos
12 else
13     return β, costsNeg
```

2.4 Parallel SAT-Solvers

The development of parallel SAT algorithms has started in the 1990s (e.g. [ZBH96]). This section provides an overview of different parallel computing approaches to solve SAT by utilizing increasingly powerful multi-core computers. Since the improvement of individual processor core performance has slowed down in recent years, the importance of parallel algorithms increased ([Wal16]). All presented approaches are based on the DPLL algorithm discussed in Section 2.2.2. Furthermore, the focus of this work is on multi-core computers with shared-memory architectures, thus possibilities for computer clusters are not discussed in detail.

Procedure 2.4: UnitPropagationWithCosts

Input: partial assignment β, Boolean Formula F_{cnf}, current-costs *costs*

```
1  while unit clause {l} exists in F_cnf do
2      β := β ∪ l
3      if l is not negated then
4          costs += c_l
5      forall clause c ∈ F_cnf do
6          if l ∈ c then
7              remove c from F_cnf
8          else if ¬ l ∈ c then
9              remove l from c
10 return β, F_cnf, costs
```

Parallel Portfolios

A simple approach to parallelize SAT are *parallel portfolios*. The general idea is to run a set of different SAT solvers in parallel on a SAT instance. As soon as one solver finds a solution, all other solvers are terminated. Thus, for each SAT instance, the best solver of all available ones is used to find a solution. Roussel describes this simple idea in [Rou12] for the solver PPfolio, which was used in the 2011 SAT Competition. PPfolio simply combined the best solvers from the previous SAT competition.

Besides Roussel's approach, other portfolio compositions are possible. For instance, a portfolio can be created by using one base SAT solver which is run in parallel using varying configuration settings. The SAT Solver ManySAT was first to use this portfolio strategy ([HJS10]). In their work, the authors state that the solver exploits the sensitivity of modern SAT solvers to configuration settings and parameters. This is achieved by running multiple instances with carefully picked orthogonal settings (cf. [HJS10]). Varied parameters are used for branching heuristics that select decision variables in the DPLL algorithm, polarity of decision variables, different restart policies, clause deletion strategies and clause learning (detailed in [HJS10], [GJLS14], [SKI14]). The goal of this *diversification* is to perform well on various SAT instances by minimizing the overlap and thus redundancy of different solvers in the search space. Many solvers, like ManySAT, use manually crafted sets of parameters. Additionally, attempts to automate this diversification process have been studied, for example in [XHLB10].

Due to the success of CDCL solvers, the exchange of learned clauses for parallel portfolio has been studied as well. The exchange introduces a form of cooperation between the different solvers, with the objective to further reduce the amount of redundant work. If one instance learns a clause from a conflict, it is distributed to all other instances to prevent them from repeating the same conflict. Questions regarding clause sharing for portfolios are: which clauses should be exchanged, how many of them and at what time. Static strategies use conditions that clauses have to satisfy to be shared, for instance the number of literals. The solver ManySAT exchanges clauses with up to 8 literals, derived by experiments with different lengths. The exchange is performed after every decision, except for learned clauses of length 1 (unit clauses) which are shared after every restart ([HJS10]). The authors of [HJPS11] state that the size of learned clauses tends to increase during the run, hence a static policy cannot be optimal. As a solution, they present a dynamic strategy that adapts the conditions used for learned clauses throughout the search process ([HJPS11]).

Divide-and-Conquer Approaches

The divide-and-conquer approach splits the search space into several disjunct subproblems. Within each subproblem, a satisfying assignment is searched which can be performed in parallel. The main challenge is to find approximately equal sized subproblems to balance the work among multiple processors. In contrast to the pure portfolio-based approach which tries to partition the search space implicitly, divide-and-conquer aims to explicitly split the search space to prevent redundant work.

The process of splitting the search space into subproblems (tasks) is called *problem decomposition* (cf. [ZBH96]). Generating subproblems pursues several goals. Firstly, the processors' idle times should be minimized. Secondly, the communication overhead between processors should be low. Thirdly, the search overhead (excess computation) should be minimized. In many cases, these goals are conflicting and are traded for one another. The subproblems complexity for DPLL vary strongly and the use of sophisticated heuristics in modern SAT solvers increases the variance across subproblems. Therefore, a *static* way of decomposing the search space (i.e. at the start of the algorithm) is disfavored against a *dynamic* decomposition which generates new subproblems during the calculation for idling processors. The foundation for approaches using dynamic problem decomposition is shown in the solvers PaSAT ([SBK01]) as well as GridSAT ([CW03]). Both are based on the work of [ZBH96], which introduces the idea of *guiding paths*. A guiding path describes a path from the root node to the current node (partial assignment). Furthermore, the path holds information about decision literals and their implication on that path.
Considering the search tree of a DPLL algorithm, subproblems can be described as branches/guiding paths that are distributed over multiple processors. The granularity of a subspace can be dynamically adjusted by splitting of a branch as a new task. Each path can be described by the partial assignment of the current node. To split off a branch, a decision literal on the path is negated. The frequency as well as the decision level at which a subproblem is split is used to adjust the amount of work sharing. Shorter guiding paths correlate to larger portions of the search space ([SLB10]). Thus, at the beginning, only one task is available, represented by the root node of the search tree. Subsequently, additional tasks can be split off and assigned to different processors. The parallel computation finishes, when all tasks have been processed or a solution has been found. This dynamic problem decomposition can achieve good load balancing, however communication and search overhead is introduced.

The described dynamic problem decomposition generates many subproblems. These can be collected in a *task pool* which can be managed in different ways, mainly centralized or decentralized.
A centralized task pool, for example used in [CW03] and [SLB10], is a global data structure (e.g. a queue) holding all subproblems, which can be accessed by all processors. The data structure can be hold and managed by a master processor, while several worker processors remove tasks from it when they are idling. Thus, the master processor holds the responsibility for load balancing, search space splitting and search termination detection (cf. [CW03], [SLB10])
In a decentralized model, each processor maintains its own local task pool. A more recent example is the ySAT solver presented in [FDH05]. In case a processor's task pool is empty, it can request a task from another processor, for instance randomly. Without a master processor, tasks such as problem decomposition and search termination detection have to be managed autonomously, which requires special algorithms. A major benefit of the more complex decentralized model is the scalability with an increasing number of processors, because no global data structure is used to hold all subproblems. This eliminates a potential bottleneck.

Similar to the portfolio-based approach, clause sharing has been studied for divide-and-conquer strategies. The authors of GridSAT ([CW03]) and PaSAT ([SBK01]) state that clause sharing can easily introduce significant communication overhead. Thus, they limit the exchange to clause lengths of three and ten, respectively. To perform the clause exchange, different models exist, such as message passing used in GridSAT or a dedicated clause database used in PaSAT. Since clause learning is not a focus of this work, refer to [SLB10] for a good overview for clause sharing models.

Cube-and-Conquer Approaches

Cube-and-Conquer is a two-phase approach to solve SAT problems. Firstly, a SAT problem is partitioned into many subproblems (cubes). Afterwards, the cubes are solved by CDCL solvers, which can be performed in parallel.

In the first phase, a lookahead solver is used to generate many subformulae from a Boolean formula. Each subformula represents a node in a search tree which can therefore be seen as a partial assignment. The use of a lookahead solver has been proposed by Hyvärinen et al. in [HJN10]. The main point are the *lookaheads*. Basically, a lookahead uses a variable which is propagated once for each truth value for a given Boolean formula F_{cnf}. Afterwards, the difference between the initial formula F_{cnf} and the reduced formulae are measured. The goal is to find a variable that leads to large reduction of the formula F_{cnf}. Lookahead solvers are considered to be good at choosing decision literals at the higher levels in a search tree, by using more global and expensive heuristics ([HJN10]). The leaf nodes of the search tree constructed by the lookahead solver can either be conflicts or form cuttoff leafs that are used in the second phase as cubes for the CDCL solver. Thus, a cube describes a path from root node to a leaf node. The decision literals on the path can then be used as unit clauses by the CDCL solver. The decision, whether a node is a cutoff leaf, is performed by a *cutoff heuristic*. For instance, methods use a depth parameter $D \geqslant 0$ or a number of assigned variables at which a branch is cut off and used for phase two ([HKWB11]). However, Heule et al. show that these can be combined as well, for example by using a product of the two metrics.

In the second phase, all cubes are processed. Each cube is processed by a CDCL solver, by solving the reduced formula. Therefore, each cube can be seen as a task. All tasks can be easily computed in parallel, by distributing the cubes over multiple processors. The search is finished, when one processor found a solution or all cubes have been computed.

Comparison of Parallel SAT Algorithms

Despite the similarities of the three presented strategies, distinction can be drawn when comparing the search tree. Parallel portfolio, divide-and-conquer, and cube-and-conquer approaches use different methods to traverse the search tree in parallel. Figure 2.2 visualizes for every paradigm how two threads process nodes in the search space.
Firstly, the parallel portfolio duplicates the Boolean formula and each solver is limited to its own search space. Communication between threads is reduced to exchanging learned clauses. This is visualized in Figure 2.2 (1) by coloring each node with both colors. This shows that when the entire search tree needs to be calculated, every solver expands all nodes. Secondly, divide-and-conquer approaches partition the work space, for instance by using guiding paths. Thus, portions of the search space are split off and processed by another core. In Figure 2.2 (2), the search tree is shared by two threads reflected by the two node colors. Thirdly, Figure 2.2 (3) shows the two-phase approach of cube-and-conquer. The white nodes are expanded in the first phase which performed sequentially using a lookahead solver. Afterwards, the cubes (leaf nodes of the subtree containing all white colored nodes) are distributed to the two cores running in parallel. Each core processes its assigned cube.

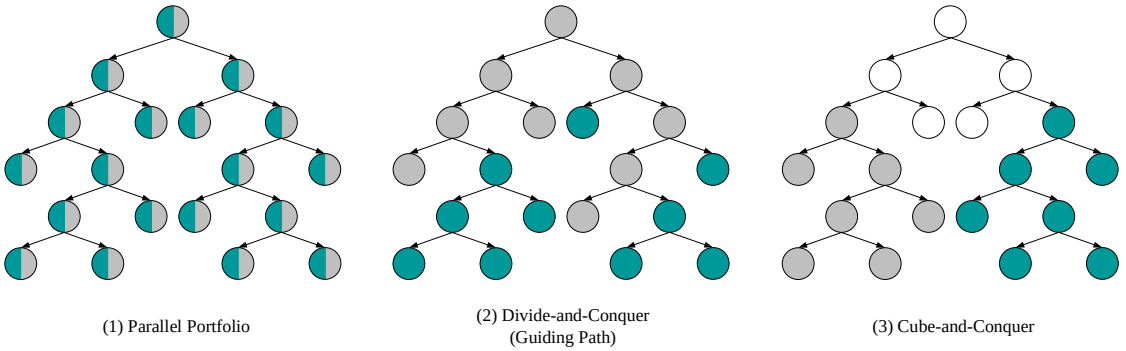

Figure 2.2: Comparison of complete search trees spanned by different parallel SAT algorithms. The colors gray and cyan indicate different cores run in parallel. White nodes express states that have been expanded in sequential manner, for instance in the first phase of the cube-and-conquer paradigm.

Nondeterminism

The introduction of parallel computation can lead to non-deterministic behavior, causing non-reproducible solutions. Determinism is critical for applications that involve user interaction, such as the domain of product configuration. In [HJS11] Hamadi et al. identify clause learning and the learned clauses' exchange as the main source of non-determinism for parallel SAT. They designed a fully deterministic parallel portfolio solver, by utilizing synchronization points at which a processor waits for all other processors to reach the same point. They use the synchronization points to exchange learned clauses as well as search determination detection. To reduce the amount of time a processor spends waiting, a synchronization barrier is not used after every conflict. Static strategies use synchronization points after a fixed number of conflicts (or other metrics like propagations and decision literals). A dynamic strategy calculates a processor specific number of conflicts, after which it has to wait at a barrier. For this calculation, the number of unit propagations performed by a processor is used as a metric of relative speed. Both methods are described and evaluated in [HJS11].

2.5 Product Configuration

This Section shows how elements of propositional logic can be applied to the domain of product configuration. The first Section 2.5.1 introduces the term product configuration and how product knowledge can be represented. Afterwards, interactive product configuration from the user's point of view is described in Section 2.5.2.

2.5.1 Knowledge Representation

Product configurators can be used to customize highly complex and varied products, which benefit the customer as well as the producer. Major benefits are the reduction of configuration errors, shortened delivery times for products, increased productivity of sales personnel and the overall accelerated sales process ([FHBT14]).
Sabin and Weigel define the term *configuration process* in [SW98]. Accordingly, the configuration process is a "design activity where the artifact being configured is assembled from instances of a fixed set of well defined component types which can be composed conforming to a set of constraints.". Essentially, a product can be assembled from a predefined set of component types, given that all constraints are fulfilled. This approach to product configuration is called *knowledge-based*, due to the reliance on domain knowledge

and problem-solving knowledge ([FHBT14]). Various concepts have been proposed to represent configuration knowledge. An early representation form is called *rule-based*. This approach uses a set of IF-THEN expressions (rules) that form the knowledge base ([FW94]). Several downsides of this concept have been pointed out, mainly the problematic maintenance of such knowledge-bases (e.g. [FW94]). In the following, another approach is introduced, the constraint-based knowledge representation.

Constraint-Based Knowledge Representation

The general idea of constraint-based models is to define a set of constraints that enclose all correct solutions. Applied to product configuration, the configuration process solves a *configuration task* which can be described as a constraint satisfaction problem. The objective is to find a configuration that fulfills the artifact's constraints as well as the requirements of the user. Therefore, the domain knowledge is defined, in which the object types (i.e. *components* and *attributes* that specify them) and their relations are described. In [SW98], different relations are given, such as classification (is-a), aggregation (part-of) and cardinality constraints. In [FHBT14], component types and constraints are referred to as the *configuration model*. According to Felfernig et al., a configuration model is necessary, because storing all possible configurations would lead to time-consuming searches and high memory requirements. Finally, the *configuration task* can be described with the configuration model and the user requirements which form the input for a configuration system ([FHBT14]).
A formal definition of a configuration task is shown in Definition 2.5, adapted from the work of Falkner, Felfernig and Haag in [FFH11].

Definition 2.5 (Configuration Task)**.** *A configuration task is described by the triplet* (V, D, C)*, where* $V = \{v_1, v_2 \ldots, v_n\}$ *is a finite set of domain (feature) variables and* $D = \{dom(v_1), dom(v_2), \ldots, dom(v_n)\}$ *represent the set of corresponding variable domains. Furthermore,* $C = P_{KB} \cup C_R$ *represent constraints, with* P_{KB} *being the product knowledge base (configuration model) and* C_R *a set of user requirements.*

The *solution* to a configuration task is called *configuration*. Falkner, Felfernig and Haag define a configuration as an instantiation (assignment) $I = \{v_1 = i_1, v_2 = i_2, \ldots, v_n = i_n\}$, with each i_j being one of the elements of $dom(v_j)$. Furthermore, they state the importance of *valid* configurations for the user. A configuration is called valid, if it is *complete* (every variable is assigned with a value) and *consistent* with all constraints ([FFH11]). Example 4 shows a simplified but illustrative configuration task.

Example 4. This configuration task describes a simplified Truck configuration, consisting of four feature variables (Axles, Transmission, Fuel type, and Color). Furthermore, a product knowledge base is given that limits the possible combinations of certain attributes.

$$
\begin{aligned}
V = \{&v_1 = Axles,\ v_2 = Transmission,\ v_3 = Fuel\ type,\ v_4 = Color\} \\
D = \{&dom(Axles) = \{2, 3, 4\} \\
&dom(Transmission) = \{5\text{-}Gear, 6\text{-}Gear, 7\text{-}Gear\}, \\
&dom(Fuel\ type) = \{Diesel, Gasoline\} \\
&dom(Color) = \{Blue, Red, White\}\} \\
P_{KB} = \{&c_1 : Axles = 2 \Rightarrow Transmission = 5\text{-}Gear, \\
&c_2 : Transmission = 7\text{-}Gear \Rightarrow Fuel\ type = Diesel, \\
&c_3 : Transmission = 5\text{-}Gear \Rightarrow Fuel\ type = Gasoline\} \\
C_R = \{&c_4 : Axles = 4,\ c_5 : Transmission = 7\text{-}Gear,\ c_6 : Color : White\}
\end{aligned}
$$

One possible valid configuration which would be calculated by a configuration system is:

$$I = \{Axles = 4,\ Transmission = 7\text{-}Gear,\ Fuel\ type = Diesel,\ Color = White\}$$

To find a solution for a given configuration task, the configuration system can use different solution search techniques. Janota discusses in [Jan08] whether SAT solvers can be used effectively in a configuration system (*configurator*). Janota shows that every configuration task can be translated into a Boolean formula. Consequently, by solving the Boolean formula in CNF by using a SAT solver, the initial configuration task is solved as well. Thus, the usage of SAT solving techniques in Product Configuration is discussed in Section 3.1. Regarding the terminology, in this work *variable* and *feature* are used interchangeably and describe an atomic characteristic of a product. The term *attribute* is used for a specification of a variable. For instance, in Example 4, "Transmission" is a variable with three attributes: "5-Gear", "6-Gear", and "7-Gear".

2.5.2 Product Configuration from the Users' View

Establishing a configuration model is central to product configuration. Usually, domain experts define variables, domains, and constraints that are necessary to model the product knowledge base. Many configurators support graphical user interfaces to maintain the configuration model, hence domain experts represent one of the user groups. An example for graphical modeling in shown in Figure 2.3. The instance being configured is a simplified Truck with a set of variables and domains. In this example, the product consists of four feature variables: Transmission, Axles, Fuel Type, and Color. The respective domains are shown below the feature variable, for instance the variable Axles has three attributes: 2, 3, and 4.

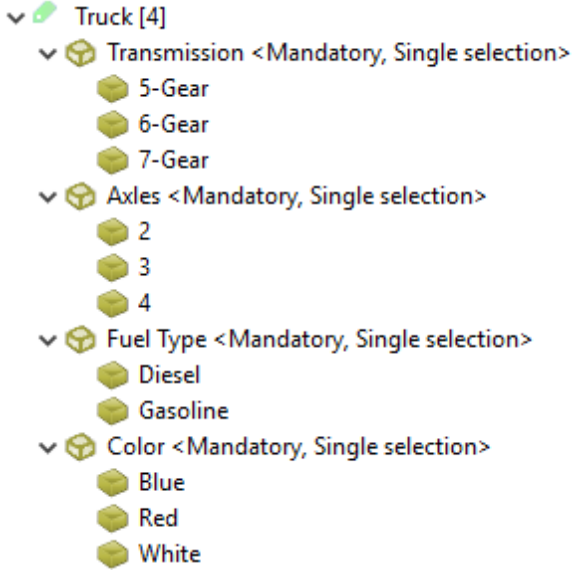

Figure 2.3: Example of modeling feature variable and domains of a simplified Truck using *CAS Configurator Merlin* ([CAS20]). The Truck consists of four feature variables: Transmission, Axles, Fuel Type, and Color. Each variable has a domain with possible values, called attributes. Additionally, constraints within a variable can be defined, for instance only one Transmission can ("Single Selection") and must ("Mandatory") be part of a complete configuration.

Generally, attributes cannot be combined arbitrarily. In Figure 2.3, only one transmission at a time can be part of a truck and the number of Axles is confined to a specific number.

This is indicated by the keyword ”Single Selection”. Furthermore, for each variable one attribute has to be part of the configuration, indicated by the keyword ”Mandatory”. The combination of keywords ”Mandatory” and ”Single Selection” result in the cardinality 1..1, exactly one attribute is required. Alternatively, variables can also be ”Optional” and ”Multiple Selection” (0..*). Usually, additional constraints across variables are added as well. In Example 4, the Transmission ”7-Gear” can only be combined with the Fuel Type ”Diesel”. A graphical representation of such a constraint is shown in Figure 2.4. Finally, the modeled configuration model has to be transformed into propositional logic and ultimately into conjunctive normal form, forming the underlying Boolean formula F_{cnf}. The formula contains the set of clauses that must be fulfilled for every configured product variant.

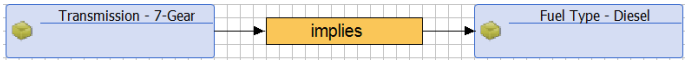

Figure 2.4: Example of a graphical constraint between two attributes of different feature variables using *CAS Configurator Merlin* ([CAS20]).

The interactive configuration process itself is usually performed on a secondary user-friendly system targeting sales personnel or end users, forming the second major user group. They use the configurator to generate a valid configuration that fulfills their requirements and all constraints of the product knowledge base. During the process, attributes are *selected* or *deselected* step-by-step, meaning that an attribute must or must not be part of the final configuration. Validity is ensured by the configurator and feedback is returned to the user, signaling changes and related information. Janota calls this type of feedback ”*explanation*” ([Jan10]). For instance, he argues that an explanation should be shown when specific attributes cannot be selected, because prior user choices led to an automatic deselection of these attributes.
All presented examples used exclusively Boolean variables. However, the configuration process does not have to be restricted to Boolean variables. Many configurators support numeric variables, string variables etc. that the user can change. Nonetheless, this thesis is focused on the configuration of Boolean variables which can be edited by the user.

3. Problem definition

The objective of this chapter is to define the problem for which parallel algorithms are presented in Chapter 4. At first, the interactive configuration process is explained. Secondly, a custom cost function is introduced to evaluate the quality of assignments generated in the process. Additionally, several examples are presented to motivate the problem of finding optimal solutions for configuration steps.

3.1 SAT and Interactive Product Configuration

The introduced term *configuration* in Section 2.5.1 can be interpreted as an assignment for an underlying Boolean formula F_{cnf}. During the interactive configuration process, a user selects or deselects attributes step-by-step to add them or remove them from a configuration. The attributes are translated to corresponding Boolean variables of F_{cnf}, thus every attribute is also an atomic proposition. A selection expresses that the attribute must be part of the current configuration. Any selection or deselection can be reverted throughout the configuration process. Contrary to Falkner, Felfernig and Haag in [FFH11], a configuration is considered valid if it is consistent with all constraints (C), but not every variable (V) needs to be assigned. The user may start with an empty configuration which is progressing through user selections and deselections until is it complete. With respect to the underlying Boolean formula, an empty configuration contains every literal as a negative one. In the following, a user selection or deselection is also called *user wish*.

During the configuration process, each user wish δ causes a *configuration step*. The step calculates a new valid configuration (solution) s using an existing assignment β (called *start-configuration*), by applying the user wish. A start-configuration describes the valid preexisting assignment for a configuration step. The user wish represents the desired change that should be applied to the existing configuration. Therefore, a user wish δ can be described as a set of literals. A positive literal $l \in \delta$ shall be added, a negative literal $\neg l \in \delta$ shall be removed from the current configuration β. Accordingly, a configuration task is defined in the following, adapted from [FFH11].

Definition 3.1. *The interactive configuration task is described by the set:*

$$(A_p, C_p, \beta, \delta)$$

Where A_p describes the set of attributes for a product p given its feature variables. The set of constraints for a specific product is given by C_p. The dynamic components are the

start-configuration β and the user wish δ. A solution s is valid if all constraints C_p are fulfilled and δ is part of the solution ($\delta \subseteq s$). A valid solution (assignment) s is called configuration.

Evaluating Assignments: Cost Function

For every configuration step, the configurator ensures validity of the resulting configuration to prevent invalid user selections. Depending on the constraints, certain user wishes violate the configuration model which are resolved by the configurator through automatically selecting or deselecting attributes. To quantify the result of a configuration step, a cost function is introduced. The costs of the changes resulting from the user wish are calculated by analyzing the difference between start-configuration β and the resulting configuration s, shown in Equation 3.1.

$$deltaCost(\beta, s) = \sum_{l \in s} \begin{cases} l \in \beta \rightarrow 0 \\ l \notin \beta \rightarrow c(l) \in \mathbb{N}_{\geq 0} \end{cases} \tag{3.1}$$

The cost function $c(l)$ must be non-negative but can be domain specific. For instance literal changes from positive to negative can be more expensive to prefer keeping literals that the user already selected in the configuration process. A change from a positive literal to a negative one expresses a deselection of an attribute for the user.

Configuration Step: From SAT to MinCostSAT

Each configuration step has two concrete requirements to fulfill.

1. Every configuration step has to apply the user wish δ to the start-configuration β, expressed as $\delta \subseteq s$.
2. The resulting configuration s has to be optimal. Hence, there is no other solution s' that results in lower costs (by applying the function $deltaCosts(\beta, s')$) and introduces the user wish δ to the start-configuration β.

The second requirement extends the SAT problem by incorporating an optimization problem of finding the minimal-cost configuration for each step.
To ensure that a user wish δ is applied, it is added as a unit clause to the Boolean formula F_{cnf}. This enforces that an assignment resulting from a configuration step can only be valid if it contains the requested user wish. This relates to the model in [Jan08], in which Janota extends F_{cnf} with user changes to $F'_{cnf} = F \wedge \delta$.
The problem of calculating a configuration step is comparable to the *MinCostSAT* (2.4) problem. The main difference is that positive variables do not inherently increase the costs. Costs are only accumulated by changes to the start-configuration, independent of the variable's truth value. The problem definition can be extended by incorporating the cost function as follows:

Definition 3.2. *The minimal-cost interactive configuration task is described by the set:*

$$(A_p, C_p, \beta, \delta, c(l))$$

Where A_p describes the set of attributes for a product p given its feature variables. The set of constraints for a specific product is given by C_p. The dynamic components are the start-configuration β and the user wish δ. Furthermore, a non-negative cost function $c(l)$ defines the cost for literal $l \in A_p$, when attribute l changes with respect to β.
A solution s is valid if all constraints C_p are fulfilled and δ is part of the solution ($\delta \subseteq s$). An optimal solution s is the minimal-cost assignment with respect to β ($\sum_{l \in s} c(l)$, $\forall l \notin \beta$). A valid optimal solution s of this task is called configuration.

Two cost functions are applied in the following examples, which use the function $deltaCost(\beta, s)$ from Equation 3.1:

1. $c_{diff}(l) = 1$

2. $c_{keep}(l) = 1$ and $c_{keep}(\neg l) = 10$

Example 5. Given are a start-configuration β and a user wish δ. Goal of a configuration step is to find all optimal configurations, quantified by the cost function $c(l)$.

Attributes A_p:

$$\{a, b, c, d, e\}$$

Constraints C_p:

$$(a\&b) \rightarrow c$$
$$c \rightarrow d$$
$$d \rightarrow e$$

In CNF:

$$F_{cnf} = (\neg a \vee \neg b \vee c) \wedge (\neg c \vee d) \wedge (\neg d \vee e)$$

Cost function:

$$c(l) = c_{diff}(l)$$

Start-configuration:

$$\beta = \{a, \neg b, \neg c, \neg d, \neg e\}$$

User wish:

$$\delta = \{b\}$$

Valid configurations:

$$s_1 = \{a, b, c, d, e\}$$
$$s_2 = \{\neg a, b, \neg c, \neg d, \neg e\}$$

Costs of valid configurations:

$$deltaCosts(\beta, s_1) = 4$$
$$deltaCosts(\beta, s_2) = 2$$

Therefore, the configuration s_2 is the optimal solution using the cost function c_{diff}.

In Example 5, the simple cost function c_{diff} is used that evaluates every change to the start-configuration with the cost of 1. This cost function is simplified to explain the general idea.
In practical applications the cost functions can be more complex and domain dependent. For example considering a configuration process, it can be beneficial to deploy a more complex cost function that weighs changes from positive to negative literals higher. Example 6 adapts the earlier scenario by using a different cost function (c_{keep}) that penalized changes from positive to negative literals by a factor of 10. Thus, attributes set through previous configuration steps are more valuable to keep than changing negative literals to positive ones (from the user's perspective: adding additional attributes to the current assignment). This illustration shows that the cost function has a strong impact on the resulting solutions for a configuration step.

Example 6. Given the scenario from Example 5, the cost function has been changed.

$$c(l) = c_{keep}(l)$$
$$\beta = \{a, \neg b, \neg c, \neg d, \neg e\}$$
$$\delta = \{b\}$$

Valid configurations:

$$s_1 = \{a, b, c, d, e\}$$
$$s_2 = \{\neg a, b, \neg c, \neg d, \neg e\}$$

Costs of valid configurations:

$$deltaCosts(\beta, s_1) = 4$$
$$deltaCosts(\beta, s_2) = 11$$

Owing to the changed cost function $c_{keep}(l)$, the configuration s_1 is the optimal and thus favorable solution.

Preserving User Selections: Pinned Attributes

Example 6 penalizes literal changes from positive to negative by a factor of 10 (c_{keep}). However, the factor is not only applied to user selections, but also to literals that have been assigned *true* as a consequence of that selection. For the user, these literals may not reflect equally important attributes. To further differentiate prior user wishes and other positive literals, two concepts are discussed in the following.

On the one hand, the cost function can be adapted to reflect the importance of user selections. For instance, each change of a literal that had been assigned by a user wish, introduces costs of 20. Therefore, user selection are only reverted (by changing the respective literal's polarity) if otherwise many changes would be required. Advantage of this approach is that a conflict between an earlier user wish and the current user selection can be resolved by reverting the earlier user selection. Major disadvantage is the complexity for the user to understand why an earlier change has been reverted. The user may not know whether a conflict exists or the number of required changes would lead to higher costs.

On the other hand, the concept of *pinned attributes* can be introduced. Every user selection δ is pinned (alternatively, the user could select whether it should be pinned or not) which enforces that the literals added remain in the configuration during subsequent configuration steps. To remember all pinned selections, the set P is maintained, containing all user wishes as a set of literals. For any subsequent configuration step, all literals of P are added as unit clauses, comparable to the user wish itself. In case the next user wish conflicts with a pinned attribute, an explanation can be shown to the user. Alternatively, the configuration step can be repeated without the conflicting pinned attribute in P, followed by signaling a required change in the pinned attributes.
Compared to the first approach, the transparency for the user is increased, because user selections are not changed unless it is necessary. Disadvantage is that enforcing pinned attributes can lead to costly configuration steps, due to more changes to the start-configuration β. Example 7 shows the impact of pinned attributes on the earlier configuration scenario.

Example 7. Given the scenario from Example 5, the cost function has been changed. Furthermore, the set P contains all literals that reflect previous user selections. Pinned attributes must be part of the solution, unless a conflict between user wish and literals of P exist.

$$\begin{aligned} c(l) &= c_{keep}(l) \\ \beta &= \{a, \neg b, \neg c, \neg d, \neg e\} \\ P &= \{a\} \\ \delta &= \{b\} \end{aligned}$$

Valid configurations:

$$s_1 = \{a, b, c, d, e\}$$

Costs of the valid configurations:

$$deltaCosts(\beta, s_1) = 4$$

The addition of P invalidates the second configuration $s_2 = \{\neg a, b, \neg c, \neg d, \neg e\}$ in Example 6, due to the negation of a. The only remaining valid solution is s_1.

MinCostConf

All previous concepts combined describe the underlying problem for this thesis, called *MinCostConf*. Being specialized on the interactive configuration process, an additional requirement is added. The user can define a limit r on how many solutions should be returned at the most, given that multiple optimal valid configurations exist. All found solutions are represented by the set S. Furthermore, this set of solution must be the same for repetitive configuration steps. Hence, deterministic behavior is required.

A formal definition for MinCostConf is given in Definition 3.3. Essentially, the task is to find up to r minimal-cost configurations for a configuration step that is triggered by a user wish. Optimality is quantified by using a cost function that evaluates the changes made to the current assignment (β) by solution s.

Definition 3.3 (MinCostConf)**.** *The minimal-cost interactive configuration task (MinCostConf) is described by the set:*

$$(A_p, C_p, \beta, P, \delta, c(l), r)$$

Where A_p describes the set of attributes for a product p given its feature variables. The set of constraints for a specific product is given by C_p. The dynamic components are the start-configuration β, the user wish δ, and pinned attributes $P \subseteq A_p$. Furthermore, a non-negative cost function $c(l)$ defines the cost for attribute $l \in A_p$, when l changes with respect to β. The maximum amount of returned solutions is limited to r.
A solution s is valid if all constraints C_p are fulfilled and δ and P are part of the solution ($\delta \cup P \subseteq s$). An optimal solution s is a minimal-cost assignment with respect to β ($\sum_{l \in s} c(l)$, $\forall l \notin \beta$). A valid optimal solution s of this problem is called configuration, the set of found solutions is given by S.
The task is to find all configurations. In case more than r configurations exist, the cardinality of S is limited to r. Repeatedly solving the same task must return the equivalent set S.

Regarding the complexity, MinCostSAT belongs to the class of NP-hard problems (e.g. [L$^+$04]). Interpreting MinCostConf as a decision problem, SAT can be reduced to MinCostConf. Considering the following CNF:

$$(a \vee b \vee c) \wedge (d \vee e)$$

Modeling constraints such that the following CNF results:

$$(\neg p \vee a \vee b \vee c) \wedge (\neg p \vee d \vee e)$$

Starting a configuration process with a start-configuration of $\{\neg p, \neg a, \neg b, \neg c, \neg d, \neg e\}$ and user wish $\{p\}$, the SAT problem is solved. Thus, MinCostConf is also NP-hard.
Assuming $P \neq NP$, there are no algorithms to solve the problem in polynomial time. In the next chapter, concepts for algorithms are presented that parallelize the search for this task using SAT and MinCostSAT techniques.

4. Concept: Parallel Algorithms for Product Configuration

This chapter describes the developed concept of a parallel SAT-solver that can be used effectively in a configurator. This work focuses on processing configuration steps (term introduced in 3.1) that a user requests during the configuration process.
This chapter is structured as follows. At first, in Section 4.1 an overview is given about the baseline sequential algorithm, A* Search, and how it is applied to DPLL. Secondly, three different parallel solver approaches are presented. The first strategy is a parallel version of the introduced A* search, shown in Section 4.2. Afterwards, a Cube-and-Conquer algorithm is discussed (Section 4.3) as well as parallel portfolios (Section 4.4). Finally, concepts to ensure reproducibility of the interactive configuration process for the parallel algorithms are displayed in Section 4.5. An overview of the presented algorithms is also given in Figure 4.1.

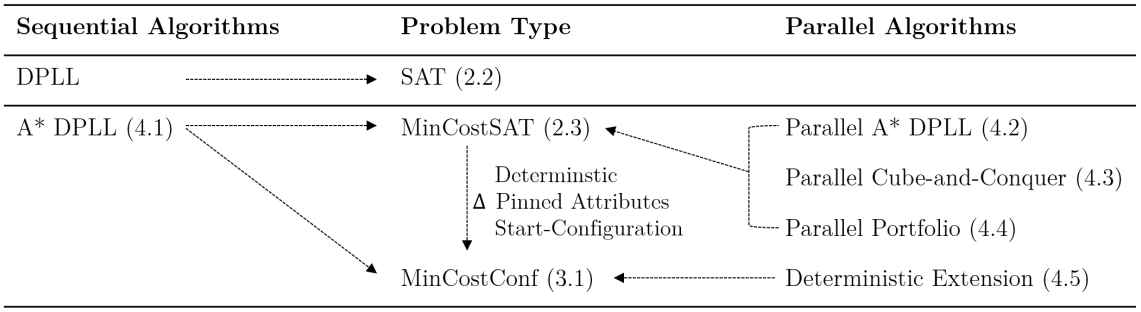

Figure 4.1: Overview of algorithms for different problem types. The lower half shows algorithms presented in this chapter.

4.1 Baseline: Sequential A* Search for MinCostConf

Most state-of-the-art SAT-solvers are based on the DPLL algorithm (see Section 2.2.2). The DPLL algorithm terminates as soon as a valid assignment is found or unsatisfiability is proven. In product configuration, a MinCostSAT problem can be identified to ensure that the optimal assignment is returned for every configuration step. The optimality of an assignment is measured by a cost function. This problem has been extended to MinCostConf in Chapter 3.

Various tree-search strategies and algorithms can be used to guarantee that the optimal assignment is found. One tree traversal strategy is called *best-first search* (BFS). At every decision point, the node with the best heuristic evaluation is explored first. A* search fulfills the optimality criteria and is used in many applications, predominantly in travel-routing systems. The algorithm has been published by Hart, Nilsson and Raphael in [HNR68], as an extension of Dijkstra's algorithm ([Dij59]).
The A* algorithm formulates its problem as a weighted directed graph and aims at finding the minimal cost path from a source node to a goal node. In this process, the algorithm constructs a search tree and always expands the most promising node. Furthermore, the A* algorithm holds an OPEN list of nodes that have not been expanded yet. The list is ordered by the costs accumulated from the root node to the current node and an heuristic estimation of the cost to reach a goal node. In MinCostConf, the path costs are given by the costs introduced through propagated literals. A second list, called CLOSED list contains all nodes that have been expanded. To decide which path to expand, Equation 4.1 is minimized over all nodes, where $g(n)$ is the path's cost from the source node to n, which is the next node on the path. Additionally, $h(n)$ estimates the cost of extending the path from node n to the goal node.

$$f(n) = g(n) + h(n) \tag{4.1}$$

The heuristic function $h(n)$ differentiates the A* algorithm from Dijkstra's algorithm. Often, the heuristic function is admissible, as such it does not overestimate the costs from node n to the goal node. In case h is an admissible function, then the A* search using this function is optimal ([HNR68]). Consequently, utilizing this property allows to terminate the search as soon as a solution has been found, because the estimated costs of all other nodes are larger than the actual cost of the found solution. Consequently, the algorithm belongs to the best-first search algorithms, because it always extends the most promising path. Additionally, A* is regarded as an efficient algorithm in terms of number of expanded nodes, for example compared to depth-first search strategies (cf. [L$^+$04, p.67] for comparisons on MinCostSAT instances).

A* Search in the DPLL Algorithm

The A* algorithm can be applied to the MinCostConf problem, which is an optimization problem. The DPLL algorithm constructs a search tree which can be interpreted as a weighted directed graph, starting from a source node. Every node in the graph represents an assignment (partial or complete), also called "state". Every edge serves as a unit propagation of a decision literal. The edges' weights are given by the costs of the propagated literals, thus the weights are non-negative. This leads to a search tree in which the costs can only increase or remain unchanged with increasing tree depth. Finally, the goal node is the lowest cost valid assignment. In the domain of product configuration, multiple valid optimal solutions may exist, that represent potential alternatives for the user to choose from. Thus, several goal nodes may exist that have to be reached.
To decide which node to expand, every node in the tree is evaluated by the function $f(n) = g(n) + h(n)$, equivalent to the A* search algorithm. The function $g(n)$ shown in Equation 4.2 sums the costs of already assigned literals. The set L_n describes the literals that are assigned on the path from source node n_1 to node n_i.
As a lower bound for the costs from n to the goal node, the heuristic function $h(n)$ (shown in Equation 4.3) is used. It accumulates the costs of all remaining unit clauses in F, that will result from the chosen literal, because their truth value is already defined but they have not been propagated yet. U_{n_i} is the set of unit clauses remaining in the node n_i.

$$g(n_i) = costs(L_{n_i}) \tag{4.2}$$

$$h(n_i) = costs(U_{n_i}) \tag{4.3}$$

Cost function for Product Configuration

During the A* search, node evaluations are mainly based on the costs of propagated literals. The general concept of quantifying costs for a configuration change is expressed in Equation 3.1 in Section 2.5. Essentially, every change with regards to the start-configuration introduces costs. The start-configuration always describes the configuration prior to the current user wish. Having selected an attribute, the user wants as few changes to the prior configuration as possible, thus a cost function is utilized. To prioritize user selections as well as keeping the cost function comprehensible, the schema shown in Procedure 4.1 is used. Hence, a literal change that reflects a removal of a literal (build-out) is ten times more expensive than an additional literal (build-in). Exceptions are so called "costless" variables which do not invoke any additional costs. These are domain-specific, for instance helper variables to enable specific feature variables to be editable on the user interface. For every node in the search tree, the costs $g(n)$ can be derived by evaluating all propagated literals on the path from the source node to the respective node.

Procedure 4.1: literalCost

Input: literal l, start-configuration β

```
1 BUILD_OUT_COSTS := 10
2 costs := 0
3 if l ∉ β & l is not costless then
4     if l is negated then
5         costs := BUILD_OUT_COSTS
6     else
7         costs := 1
8 return costs
```

An example of the A* algorithm search tree is given in Figure 4.2. The problem is displayed as a weighted directed graph, in which the weights are derived from the cost function shown in Procedure 4.1. The example shows the entire search tree, the A* search algorithm would not expand all nodes, for instance node n_3 would not be further expanded to reach nodes n_6 and n_7, since node n_2 and all subsequent nodes are cheaper and lead to a valid assignment (n_4).

In subsequent sections, two general approaches are compared. The first class of algorithms use the space-splitting approach, also known as divide-and-conquer which is presented in Section 4.2 and 4.3. Secondly, an approach based on a parallel portfolio is used, described in Section 4.4.

4.2 Parallel Divide-and-Conquer

One popular approach for parallel search is to use the divide-and-conquer paradigm. During the search, the problem is recursively broken down into subproblems. These simplified problems can be distributed to different processing units to ensure that no search space overlapping between cores occur. By finding the solutions for the subproblems, the overarching problem is solved. Two versions of divide-and-conquer are presented, the first one is based on a centralized task pool and the second approach decentralizes this task pool.

4.2.1 Centralized Parallel A* Search

An intuitive solution is to extend the sequential A* algorithm to perform it in a multithreaded environment. In parallel versions of the centralized A* approach, the single

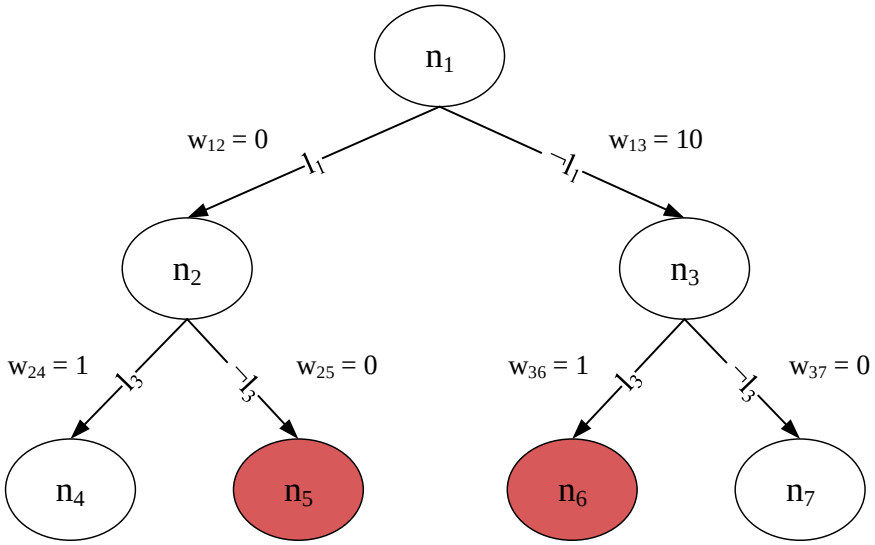

Figure 4.2: A* search tree of the DPLL algorithm for the formula $F = (\neg l_1 \vee \neg l_2 \vee l_3) \wedge (l_1 \vee \neg l_3)$, start-configuration $\beta = \{l_1, \neg l_2, \neg l_3\}$ and user wish $\delta = \{l_2\}$. Nodes colored in white display expanded nodes, red nodes indicate conflicts. The optimal valid assignment is node n_4 with $c_4 = 1$. Costs are determined by adding all weights along the path from root node n_1 to n_4.

OPEN list is kept (e.g. in [IS86] and [PLK14]). Therefore, k threads work concurrently on a shared OPEN list. Each thread retrieves nodes from that data structure in order of their f-values, expands them, and inserts successors of explored nodes (states) into the OPEN list.
The authors of [IS86] state that re-expansions of nodes (contrary to the sequential algorithm) are possible, because a state may not have the optimal g-value when taken from the list and being expanded. For instance, a node that was taken second may have been updated in the sequential algorithm by processing the first node. This issue of re-expansions is not applicable to the DPLL algorithm, since nodes in the search tree are only reached by one specific path. This property leads to two improvements. Firstly the g-value of a node cannot be updated by processing another node first. Secondly, duplicate detection of states is not required, because only one path leads to every state, thus two threads cannot arrive at the same state. Therefore, when using parallel A* search for the DPLL algorithm, the CLOSED list is not required.
Nevertheless, the concurrent work can lead to search overhead by expanding suboptimal nodes that would not have been expanded by a sequential version of A*. Besides search overhead, expanding nodes in parallel easily leads to nondeterministic behavior. This critical problem as well as solutions are discussed thoroughly in Section 4.5.

In a multithreaded environment, a coordination mechanism for all threads is required. An overview of the used thread management is shown in Figure 4.3. It follows the *Master-Worker* paradigm, with a single master thread and a set of worker threads. The latter can scale from 1 to k threads. The master maintains a overview of the procedure, while the workers process tasks in parallel. The master initializes the MinCostConf instance resulting in the root node and dispatches the k threads to start processing. Subsequently the workers remove nodes from the OPEN list, which is implemented as a priority queue. The highest priority is given to the node having the lowest cost estimation (f-value). Each task consists of the unit propagation of a decision literal (except for the root node) and resulting unit clauses. Communication between threads is performed by using shared data, e.g. for thread state information and all currently known optimal solutions. Access to the

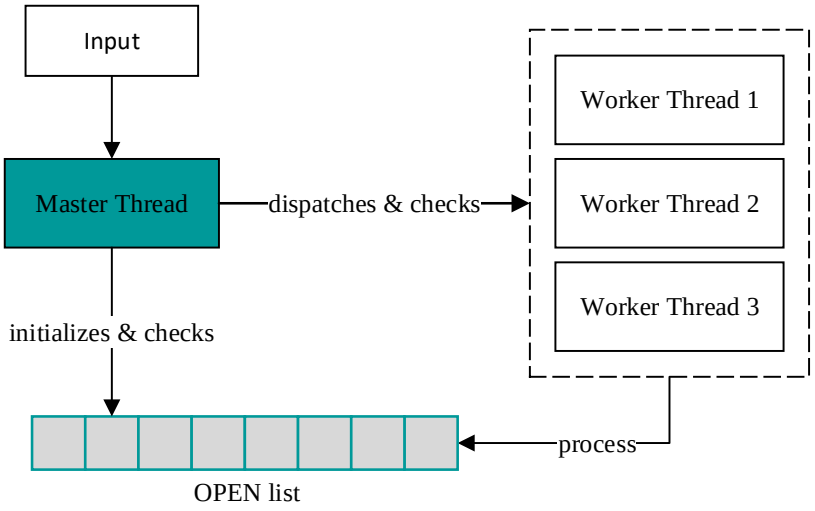

Figure 4.3: An explanatory thread overview of the centralized parallel A* approach using a single master thread and three worker threads (Master-Worker paradigm). All threads are accessing one central OPEN list, using a shared-memory architecture.

shared data is required for workers as well as for the master. Workers need to know the current optimal solutions to decide whether their task (node) needs to be processed or can be skipped.
While all workers are processing nodes, the master checks whether the OPEN list is empty, because an empty list indicates that the search is finished. Additionally, the master checks the state of all worker threads by accessing shared data, to decide whether the search can be terminated or not, for instance in case the optimal solution has been found already. A detailed description of the responsibilities is given in the following.

Master Thread

The master thread's responsibility is to prepare all required data for the DPLL algorithm, to initialize all worker threads, and to continuously check the search termination criteria to minimize the computation time. Finally the master thread returns all found optimal valid configurations.

In Algorithm 4.2 the logic is presented. Initially, the shared variables *searchFinished* and *minCost* are initialized. The variable *minCost* defines the minimum cost of currently known solutions (upper bound). Furthermore, all literals of the user wish δ are added as unit clauses to the Boolean formula F. To manage the OPEN list used in A* search, a priority queue Q is initialized and the root node is inserted with the f-value (cost estimation) of 0.
Secondly, all k worker threads are initialized and started to process the nodes on Q (line 6-8). At first, each *thread-waiting-sate* is set to *false*. Array TWS holds information of the waiting states of all worker threads, whether a thread is currently processing a node or awaiting a task. This information is shared with the master thread and is continuously updated by each worker thread to signal its internal state. Moreover, the shared array *running-thread-costs* RTC holds information about the cost of the node that each worker thread is currently processing. The costs of a node consists of the cost of current literal changes with respect to the start-configuration (g-value) as well as the estimation of the

Algorithm 4.2: Parallel centralized A* - Master Thread

Input: Start-configuration β, user wish δ, thread count k, requested solutions r, source node n
Data: Clauses F, Priority queue Q, array of thread waiting states TWS, array of costs for each thread RTC, integer minCost, Boolean searchFinished
Output: Set of configurations S

```
1  searchFinished := false
2  minCost := ∞
3  forall literal ∈ δ do
4      add literal as unit clause to F
5  Q.INSERT(n, 0)
6  for i := 0; i < k; i++ do
7      t_i.PROCESS(F, β, Q, TWS, RTC, S, minCost, searchFinished, i)
8      TWS[i] := false
9  while searchFinished = false do
10     Lock TWS
11     for i := 0; i < k; i++ do
12         if TWS[i] = false then
13             break
14         allThreadsWaiting := true
15     Unlock TWS
16     Lock Q, RTC
17     if Q is empty & allThreadsWaiting = true then
18         searchFinished:= true
19     else
20         headHigherCosts := Q.PEEK().COSTS() > minCost
21         headEqualCosts := Q.PEEK().COSTS() ⩾ minCost
22         threadsHigherCosts := min value in RTC > minCost
23         threadsEqualCosts := min value in RTC ⩾ minCost
24         if |S| >= r & headEqualCosts = true & threadsEqualCosts = true then
25             searchFinished:= true
26         else if headHigherCosts = true & threadsHigherCosts = true then
27             searchFinished:= true
28     Unlock Q, RTC
29 return S
```

admissible heuristic function h.
Finally, the master thread's main responsibility is to periodically test all termination criteria, to abort the search as early as possible with the goal of reducing computation time. Two distinct termination conditions are used. On the one hand, the search can be stopped in case all nodes are processed. Thus, the priority queue Q is empty and all k threads are waiting for new nodes to expand which is reflected by all waiting states being set to *true* in the array TWS. On the other hand, the search can be aborted, if all nodes that have not been explored yet are more expensive, i.e. their f-value is higher than the cost of the known optimal solutions. Additionally, in case the requested amount of optimal solutions (r) has been found, the configurations can be returned early given that all other nodes are of equal or greater cost. Both cost criteria are tested by analyzing the head of the priority

queue Q and checking the running-thread-costs via the array RTC (line 20-27). Both, the master thread and worker threads utilize locking mechanism to access shared variables, such as the priority queue Q, TWS etc. Strategies concerning the locking behavior are discussed in detail in a following paragraph.

Worker Thread

A worker thread is responsible for processing nodes within the search tree span by the DPLL algorithm. Pseudocode in Algorithm 4.3 shows the processing logic of nodes for a worker thread. All k worker threads are initialized by the master thread with access to shared variables and is assigned an unique thread identifier t. The shared variable encompass all clauses F, the start-configuration β, priority queue Q with all nodes, data structures for the termination criteria TWS and RTC, the result set of all configuration S and the currently known minimum cost value of all solutions $minCost$.
Being initialized, each thread starts to expand nodes in parallel until the search is finished, reflected by the Boolean variable $searchFinished$. In the course of that, a worker thread removes the minimum cost node of the priority queue Q. If Q is empty, the worker thread changes its waiting state to *true* (line 6) and skips the current iteration of the loop (line 8). Otherwise, the thread registers the node's current costs (f-value) in the array RTC and updates its waiting state to *false* (line 10-12). In this phase, only one thread at a time has access to the priority queue Q, which is ensured by locking mechanisms.
After successfully obtaining a node, the procedure $GetSuccessors$ is executed which performs unit propagation to generate up to two child nodes. All successors of N are analyzed in line 14 to 26. In case a node represents a valid configuration that is at most as expensive as a currently known solution, it is added to the set of shared configurations (solutions) S. Furthermore, if the node's costs are lower than all present solutions, already found ones are removed before inserting the current node and the shared variable $minCost$ is updated to reflect the new optimum. Thus, S always contains the solutions with currently known lowest cost. Alternatively, a node n' can represent a conflict which is analyzed, for instance to derive a learned clause. Otherwise, two nodes are created by adding a decision literal which represents a branching point in the search tree. The two nodes are inserted into Q with their current f-value as their priority. Afterwards, the worker thread t resets its running-thread-costs in the shared array RTC to signal that the loop has been finished.

The generation of successor nodes is detailed in Procedure 4.4 and 4.5. During processing, unit propagation of the decision literal (unless it is the root node, because it does not have a decision literal) as well as all literals of resulting unit clauses are performed. During the propagation, three different results can be reached:

1. F contains an empty clause, the node represents a contradiction and can be used to analyze the conflict.

2. The set of clauses F is empty, a valid configuration has been found. The set of changed literals with regards to the start-configuration describe the delta and can be used to derive a new configuration/solution.

3. All unit clauses have been processed but the node does not represent a valid configuration. A new decision literal has to be chosen by using a branching heuristic. Afterwards, two new nodes are created, one with the positive and one with the negative decision literal.

During each unit propagation, the costs of the propagated literal are added to the variable *costs*, by analyzing the start-configuration to determine whether the literal has changed its truth value with respect to it. Additionally, propagated literals are kept in the set

Algorithm 4.3: Parallel centralized A* - Thread Process

Input: Clauses F, start-configuration β, Priority queue Q, array of thread waiting states TWS, array of costs for each thread RTC, set of optimal valid configurations S, integer minCost, Boolean searchFinished, thread identifier t

```
1  while searchFinished = false do
2      Lock Q, TWS, RTC
3      n := Q.poll()
4      Unlock Q
5      if n = null then
6          TWS[t] := true
7          Unlock TWS, RTC
8          Continue
9      else
10         TWS[t] := false
11         RTC[t] := n.costs()
12         Unlock TWS, RTC
13     N := n.getSucessors(F, β)
14     forall n' ∈ N do
15         nodeCost := n'.costs()
16         if n' is valid configuration & nodeCost ⩽ minCost then
17             Lock S
18             if nodeCost < minCost then
19                 minCost:= nodeCost
20                 S := ∅
21             S.insert(n')
22             Unlock S
23         else if n' is a contradiction then
24             n'.analyzeConflict()
25         else if nodeCost ⩽ minCost then
26             Lock Q
27             Q.insert(n', nodeCost)
28             Unlock Q
29     RTC[t] := 0
```

propagatedLiterals to derive the resulting solutions, by comparing them to the start-configuration. The cost function for a literal (line 5) is shown in Procedure 4.1.

Synchronization Mechanisms and Data Structures

In parallel search with multiple threads, data has to be shared across processing units. Accessing and operating on shared variables can easily lead to consistency errors and interference problems.

Memory consistency errors occur whenever two or more threads do not share a common view of a resource. In case one thread updates a variable and this update is not propagated to all other threads, the view on that data becomes inconsistent. Consequently, if a thread reads the non-updated data, a memory consistency error arises. Therefore, each thread has to ensure that accessed data is up-to-date, for example by reading from main memory

Procedure 4.4: GetSuccessors

Input: Clauses F, start-configuration β
Data: decision literal $l_{decision}$, set of propagated literals propagatedLiterals, aggregated costs of node costs
Output: list of successors N

```
1  N := ∅
2  F := F ∪ {l_decision}
3  F, propagatedLiterals, costs:= UnitPropagation(F, propagatedLiterals, costs)
4  if ∅ ∈ F then
5      return contradiction
6  else if F = ∅ then
7      return configuration with propagatedLiterals
8  l_decision := choose unassigned literal l in F
9  n_1 := new state with l_decision, propagatedLiterals, costs
10 n_2 := new state with ¬l_decision, propagatedLiterals, costs
11 N.add(n_1, n_2)
12 return N
```

Procedure 4.5: UnitPropagation

Input: Boolean Formula F, propagatedLiterals, costs

```
1  while F contains unit clause {l} do
2      propagatedLiterals.insert(l)
3      costs + = literalCost(l, S)
4      forall clause c ∈ F do
5          if l ∈ c then
6              remove c from F
7          else if ¬ l ∈ c then
8              remove l from c
9  return F, propagatedLiterals, costs
```

instead of temporary caches.
Interference problems can be, for example, described by *race conditions* between two threads. They occur if threads simultaneously try to read and write a shared variable. In most programming languages, operations like "increment" are not atomic. An atomic operation performs a task without the possibility of interference from other operations. One strategy to perform atomic operation is called *compare-and-swap* (CAS). This operation compares the current value with an expected value and only updates the current value if it matches the expected value. This is a form of *optimistic locking*. Thus, in case two threads try to update a value simultaneously, only one will succeed. The failing thread has to handle the scenario and decide whether to try again or continue its work.

An alternative to optimistic locking in form of atomic operations is *pessimistic locking*. It is performed by using synchronization mechanisms which ensure that only one thread enters a critical code section or accesses a resource at a time ([Jon07]). *Locks* are one mechanism which enforce mutual exclusion to prevent race conditions. Utilizing this mechanism, a thread that tries to access specific data has to first acquire a lock. After a thread has finished its operation, the lock is released to let another thread acquire it. In case a thread

is not allowed to acquire the lock, different waiting techniques can be applied. A thread can be blocked, using a so called spinlock, until the lock is released. Alternatively it can be re-scheduled by the operating system. The former is efficient whenever threads are only blocked for short periods of time. Using locks introduces a source of overhead, due to memory allocation, time that threads spent waiting, acquiring locks, and releasing locks. Regarding this synchronization overhead, it is important to consider the granularity of a lock. The granularity describes the amount of data covered by the lock ([GLP75]).
Coarse-grained locking uses relatively few locks (in an extreme case only one lock to protect all shared data). This leads to higher lock contention (threads waiting to access the data) but less lock overhead (fewer acquire and release operations). With increasing thread count, lock contention becomes more severe. *Fine-grained locking* utilizes multiple locks, in which each lock protects a smaller section of data. This increases the lock overhead but decreases lock contention. Consequently, the granularity has to be balanced to limit the overall added overhead ([GLP75]).

The centralized parallel A* approach, displayed in Algorithm 4.2 and 4.3, uses two shared primitive variables, the Boolean $searchFinished$ and the integer $minCost$. Especially the latter is updated several times during a configuration step, because many suboptimal solutions may be found during the parallel search process. Thus, multiple threads try to retrieve and update the value of this variable simultaneously. To avoid memory inconsistency, all updates to these variables use optimistic locking in form of atomic CAS instructions.
Moreover, several complex data structures are utilized. The most central data structure is the priority queue Q, representing the OPEN list by maintaining all nodes that may be expanded during the tree search. It is based on a binary heap and uses locks as a synchronization mechanism to allow parallel work of k threads on a single queue. An increasing number of threads (k) can result in high lock contention on Q. However, the assumption is that the number of operations on Q is relatively low in the domain of product configuration, due to solving smaller problem instances. Moreover, the node expansion is expensive, because it involves performing compute-intensive unit propagation which further limits the lock contention. For performing search termination detection, the two arrays thread-waiting states (TWS) and running-thread-costs (RTC) are used. Both enforce locking mechanisms to restrict concurrent thread access, for instance of the master thread and worker threads. While a worker thread retrieves a node from Q, the master thread has to wait until the worker thread has updated its waiting state as well as running costs (Algorithm 4.3, line 2-12). Otherwise the queue might be empty by removing the last node and the thread waiting state may still be set to *true*, due to a preceding failed removal attempt (Algorithm 4.3, line 6-8). This leads to the master thread terminating the search early (Algorithm 4.2, line 17-18). All mentioned locking mechanism are fine-grained, because each lock protects a small section of data, in particular single data structures like Q, TWS, and RTC.
Lastly, the set S holds all found valid optimal configurations. The number of requested and found solutions is usually rather small ($\leqslant 10$). Accordingly, the load for that data structure is low which allows for synchronized write mutation operations without causing a bottleneck. Reading operations are performed without synchronization, hence multiple threads can read S concurrently. The described locking techniques are also applied in all other approaches discussed in the following sections.

4.2.2 Decentralized Parallel A* Search

The centralized parallel A* search uses a single OPEN list for all k threads. Concurrent access to this list can become a bottleneck, especially for larger thread pools. For problem domains that generate and process many nodes, a centralized parallel A* search algorithm can perform worse than its sequential version [BLRZ10]. To assess and evaluate the

centralized approach in the domain of product configuration, a second algorithm based on a *decentralized* parallel A* search is presented. The objective is to addresses the main bottleneck, by reducing the lock contention on the OPEN list which is the most central data structure.

In a decentralized parallel A* search, each thread maintains its own local OPEN list to perform the search. Initially, the root node is assembled and processed by a thread. Subsequently generated nodes are distributed among all threads to achieve good load-balancing with low idle time. Especially the load balancing strategy is a deciding factor for computation time, thus many different ideas have been developed.
Kumar, Ramesh, and, Rao were among the first to utilize a "distributed strategy" in [KRR88]. Their approach generates an initial amount of nodes and distributes them to all OPEN lists. The different threads expand the nodes in parallel. To avoid threads working on suboptimal parts of the search tree, since the thread is limited to its own OPEN list, they introduced a communication strategy to share nodes. The goal is to have all threads working on promising sections of the search space. The communication to distribute newly spawned nodes is performed by choosing another thread *randomly* and inserting the node to the target's local OPEN list.
Another frequently used strategy is called *hash-based work distribution*. This technique uses a hash function to determine the thread which has to process a new node. The objective is to achieve efficient load-balancing and duplicate detection, because each node is assigned to exactly one thread. The idea was introduced by Evett et al. in [EHMN95]. Further work on efficient hash functions, which are predominantly domain specific, led to the "Abstract Zobrist hash function" presented in [JF16]. For each node, representing a state in the search space, an abstract hash is calculated by ignoring some features. Using an abstract hash function, a search space is split up into partitions which can be assigned to unique threads. This strategy is particularly useful in distributed memory architectures that introduce high communication overhead.

To apply the decentralized parallel A* search to DPLL, each of the k threads maintains a local priority queue as an OPEN list. Moreover, a distribution strategy is required to attain good load balancing. Besides additional priority queues, the overall coordination mechanism is similar to the presented centralized parallel A* search. Therefore, a master thread and k worker threads are used. In the following, the responsibilities of the different threads are explained. Being a variation of parallel A* search, only key differences to the centralized approach are elaborated.

Master Thread

The master thread's function is the search initialization and termination detection. Being an extension of the centralized A* search algorithm, only a few changes have been made for the master thread's logic, shown in Algorithm 4.6. Due to the distributed strategy using k OPEN lists, Q represents an array of priority queues, one for each thread. For initialization, the root node is inserted into the queue of the first thread (line 5). Successive nodes are distributed by the worker threads.
Having multiple OPEN lists changes the search termination detection. First criteria tests whether all queues in Q are empty and all threads are on a waiting state (line 17). Second criteria tests whether all remaining nodes in all OPEN lists of Q and RTC are more expensive or at least equal-cost and r solutions have been found (line 20-27). Thus, the search termination detection is more complex, to accommodate the increased number of priority queues.

Algorithm 4.6: Parallel decentralized A* - Master Thread

Input: Start-configuration β, user wish δ, thread count k, requested solutions r, source node n
Data: Clauses F, array of Priority queues Q, array of thread waiting states TWS, array of costs for each thread RTC, integer minCost
Output: Set of optimal valid configurations S

```
1  searchFinished := false
2  minCost := ∞
3  forall literal ∈ δ do
4      add literal as unit clause to F
5  Q[0].INSERT(n, 0)
6  for i := 0; i < k; i++ do
7      t_i.PROCESS(F, β, Q, TWS, RTC, S, minCost, i)
8      TWS[i] := false
9  while searchFinished = false do
10     Lock TWS
11     for i := 0; i < k; i++ do
12         if TWS[i] = false then
13             break
14         allThreadsWaiting := true
15     Unlock TWS
16     Lock Q, RTC
17     if all queues in Q are empty & allThreadsWaiting = true then
18         searchFinished:= true
19     else
20         headHigherCosts := Q[i].PEEK().COSTS() > minCost ∀ i ... k
21         headEqualCosts := Q[i].PEEK().COSTS() ⩾ minCost ∀ i ... k
22         threadsHigherCosts := min value in RTC > minCost
23         threadsEqualCosts := min value in RTC ⩾ minCost
24         if |S| >= r & headEqualCosts = true & threadsEqualCosts = true then
25             searchFinished:= true
26         else if headHigherCosts = true & threadsHigherCosts = true then
27             searchFinished:= true
28     Unlock Q, RTC
29 return S
```

Worker Thread

In a decentralized approach, displayed in 4.7, the worker thread only processes nodes from its local task pool to reduce lock contention (line 3). Processing a node and in particular the function *GetSuccessors* follows the same logic as shown in Algorithm 4.3 and Procedure 4.4.
One major difference between the centralized and decentralized strategy is the handling of successor nodes. Centralized parallel A* search inserts all successor node into a shared OPEN list. During the decentralized search, every successor node is assigned to one specific thread, determined by the function *DetermineThreadToShare* (line 24). Different distribution techniques are viable, such as random, round-robin, and hash-based task

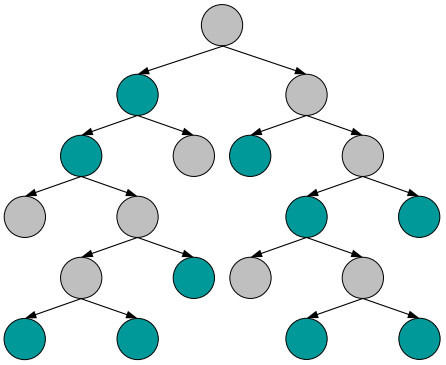

Figure 4.4: Search tree span by using parallel A* search. The colors gray and cyan indicate different threads run in parallel.

sharing. In round-robin distribution, each thread uses a specific order to share tasks, for instance by repeatedly iterating the list of threads from top to bottom. Random and round-robin distribution are very simple mechanisms to achieve efficient load-balancing by distributing tasks uniformly among all threads. Hash-based distribution has been used lately across various domains, due to lowered communication overhead and its ability to perform duplicate detection by assigning each node to a unique thread. The latter is not applicable in the DPLL algorithm, since nodes can only be reached by a unique path, making duplicate detection obsolete.
These distribution mechanisms differ from the *work stealing* strategy, because nodes are distributed after their generation. In contrast, work stealing lets threads steal nodes from other threads, usually performed when their task pool is empty. This would not fulfill the goal of having all threads working on good parts of the search space, because each thread would only request new nodes when the local OPEN list is empty. Sharing newly generated nodes leads to more threads working on good parts of the search tree, since newly generated nodes are always children of previously locally optimal nodes. This property is ensured by using priority queues for the OPEN list, in which highest priority states have the lowest cost estimations.

Comparison of Centralized and Decentralized Parallel A* Search

Considering the search tree span by the parallel A* algorithm, both approaches are similar. Figure 4.4 shows a search tree span by using two different threads, comparable to 2.2 where different parallel SAT algorithms are visualized. The figure shows that the search tree is similar to the presented divide-and-conquer paradigm using guiding paths. A difference is noticeable in the granularity, parallel A* shares single nodes while guiding paths split off longer paths.

Comparing properties of the decentralized and centralized parallel A*, the main advantage of the former is the reduced lock contention on the shared OPEN list. With increasing number of threads, the contention on the central data structure surges (synchronization overhead). Hence, the decentralized parallel A* search has potential to scale better with more processing units.
However, distributing nodes across multiple OPEN lists brings disadvantages as well. Firstly, communication overhead is increased, because nodes are exchanged across threads to reduce idle time and the number of suboptimal nodes expanded. Even though this thesis primarily considers shared-memory environments, effort is introduced by moving nodes between OPEN lists and consequently threads. Secondly, search overhead is increased compared to the centralized approach. In general, it can be measured by comparing the

Algorithm 4.7: Parallel decentralized A* - Thread Process

Input: Clauses F, start-configuration β, array of Priority queues Q, array of thread waiting states TWS, array of costs for each thread RTC, set of optimal valid configurations S, integer minCost, threadId t

```
1  while searchFinished = false do
2      Lock Q[t], TWS, RTC
3      n := Q[t].poll()
4      Unlock Q[t]
5      if n = null then
6          TWS[t] := true
7          Unlock TWS, RTC
8          Continue
9      else
10         TWS[t] := false
11         RTC[t] := n.costs()
12         Unlock TWS, RTC
13     N := n.getSucessors(F, β)
14     forall n' ∈ N do
15         nodeCost := n'.costs()
16         if n' is valid configuration & nodeCost ⩽ minCost then
17             Lock S
18             if nodeCost < minCost then
19                 minCost:= nodeCost
20                 S := ∅
21             S.insert(n')
22             Unlock S
23         else if n' is a contradiction then
24             n'.analyzeConflict()
25         else if nodeCost ⩽ minCost then
26             i := determineThreadToShare()
27             Lock Q[i]
28             Q[i].insert(n', nodeCost)
29             Unlock Q[i]
30     RTC[t] := 0
```

expanded nodes in parallel to the number of its sequential implementation. Reason for this growth is that threads potentially work on suboptimal parts of the search tree while waiting for more promising nodes from other threads. In a centralized A* search, all threads work on one data structure, thus promising nodes are expanded first and threads often work on similar sections of the search tree.

In many cases, the mentioned overheads are interdependent. Reducing one overhead usually increases another one. Therefore, advantageous and disadvantageous properties of the centralized and decentralized parallel A* must be compared for different problems, in this case SAT instances. The harder the SAT instance is, for instance measured by the size of the spanned search tree, the larger becomes the synchronization overhead of the centralized A* approach. Theoretical problems like random SAT tend to be more complex while MinCostConf instances generated by an interactive configuration process are simpler.

An extensive analysis and benchmark of the two concepts is shown in Chapter 5, using different theoretical problems as well as industry cases.

4.3 Parallel Cube-and-Conquer

An introduction into Cube-and-Conquer (C&C) algorithms is presented in Section 2.4. Summarized, it is a two-step approach to solve the SAT instances. First, the problem is divided into subproblems (cubes) which are subsequently solved in parallel. Regarding thread management, the Master-Worker paradigm is used, comparable to previous parallel A* search algorithms. The master thread is entirely responsible for the first phase, thus generating the desired amount of cubes. Afterwards, the master distributes the cubes among all k available worker threads which process them in parallel. A detailed description of C&C using master and worker threads is given in the following paragraphs.

Master Thread

The pseudocode in Algorithm 4.8 shows how the master process works. At first, the user wish δ is applied to the Boolean formula F containing all clauses, to ensure that the solutions incorporate the user requirements (line 3-4). Then priority queue Q is initialized with the root node and cubes are generated by the procedure *CreateCubes* (line 5-6). In case that the problem has been solved during the sequential generation of cubes, all optimal valid configurations S are returned (line 7-8). Otherwise, the set of subproblems are distributed uniformly among all k threads, by preparing a list of cubes for each thread (loop 9-11). Consequently, all worker threads are initialized and start to process the work units (loop 12-14). While the worker threads are busy, the master thread is idling until all workers finished their assigned cubes. Search termination detection to ensure optimality is not performed by the master, but solely by the workers through assessments whether a cube can be skipped or aborted. This approach is in stark contrast with the parallel A* search algorithms presented in Section 4.2.1. Nevertheless, other search termination criteria can be added, for instance limiting the amount of time after which the search is aborted (timeout). Eventually, the found configurations S are returned.

A crucial factor for effectively partitioning the search space is the *cutoff heuristic*, responsible for determining the optimal amount of generated cubes. Various strategies are explained in [HKWB11], all sharing the objective to have equally complex subproblems that combined can be solved at least as quickly as the original problem. For this work, a rather simple cutoff heuristic is incorporated. Due to using a best-first search to generate cubes, a priority queue Q holds all current nodes that may be expanded. The cutoff b describes the maximum number of nodes in Q. After exceeding b, the first phase is stopped and nodes in Q represent the cubes. Therefore, it is dependent on the amount of branches performed and nodes refuted. For example, with $b = 100$, the first phase generates 100 cubes that are consequently distributed in the second phase. In case the set of optimal solutions is found during the first phase, the second phase is not performed.

CreateCubes, shown in Procedure 4.9, sequentially generates a set of subproblems. Each subproblem represents a node (cutoff leaf) in the search tree. Therefore, a cube is a partial assignment with a set of literals assumed to be true. All cubes combined cover all subproblems of the Boolean formula. The generation of cubes is also performed in best-first search, comparable to previous approaches.
During the calculation, the priority queue's size is periodically checked to create at most b nodes (line 3). In each loop, a node is removed from Q and processed by calling the procedure *getSuccessors* shown in Procedure 4.4. Afterwards, every child node is analyzed, whether it is a valid solution, a conflict or a branching point in the tree (line 6-14). Since the cube generation is performed sequentially in best-first search, any found configuration

cannot be suboptimal and is added to S. Thus, a new solution cannot be of lower cost than any previous one and S can only be expanded.
This property is also important for the search termination detection on line 15-19, showing three criteria. Firstly, the search is aborted if the queue is empty. Secondly, if r solutions have been found, the search can be stopped because these configurations are optimal due to the sequential and deterministic processing. Thirdly, the head of priority queue Q is more expensive than the currently known minimum cost solution, hence all contained nodes are suboptimal.

Algorithm 4.8: Parallel Cube & Conquer - Master Thread

Input: Start-configuration β, user wish δ, thread count k, requested solutions r, source node n, generation cutoff b
Data: Clauses F, Priority queue Q, array of lists cubesPerThread, integer minCost
Output: Set of optimal valid configurations S

```
1  searchFinished := false
2  minCost := ∞
3  forall literal ∈ δ do
4      add literal as unit clause to F
5  Q.INSERT(n, 0)
6  searchFinished, S, Q:= CREATECUBES(F, β, Q, r, b, S)
   // solution may already be found during sequential search
7  if searchFinished then
8      return S
   // distribute cubes
9  for i := 0; i < length of queue Q; i++ do
10     threadId := i % k
11     cubesPerThread [threadId].ADD(Q.POLL())
   // start parallel processing with k threads
12 for i := 0; i < k; i++ do
13     cubes_i := cubesPerThread [i]
14     t_i.PROCESS(F, β, cubes_i, S, minCost, i)
15 wait for all threads to finish
16 return S
```

Worker Thread

The pseudocode for processing predefined cubes is shown in Algorithm 4.10 which is executed by multiple threads in parallel. Each worker thread is assigned a list of subproblems, "*cubes*", that have to be processed (loop 3). For each cube, a local priority queue Q is initialized with the cube, a partial assignment, as the root node (line 2-3). Subsequently, a best-first search is executed for the current cube (line 4-19), constructing a sub-tree of the original search space. Thus, nodes are removed from the priority queue iteratively and successor nodes are generated by the procedure *getSuccessors* (shown in Procedure 4.4). Essentially, unit propagation and the choice process for the next decision literal $l_{decision}$ is performed, resulting in up to two child nodes, one for each truth value of $l_{decision}$. The handling of a successor node, shown on lines 7-17, corresponds to the previous parallel A* search algorithms. In essence, covering the three possible outcomes: a valid currently optimal configuration, a conflicting assignment or a branching point. Furthermore, to reduce search overhead, cube termination detection is performed (line 18-19). On condition that the

Procedure 4.9: CreateCubes

Input: Clauses F, start-configuration β, Priority queue Q, requested solutions r, cutoff b, set of optimal valid configurations S

Output: set of optimal valid configurations S, priority queue Q

```
1  minCost := ∞
2  searchFinished:= false
3  while Q.SIZE() <= b do
4      n := Q.POLL()
5      N := n.GETSUCESSORS(F, β)
6      forall n' ∈ N do
7          nodeCost := n'.COSTS()
8          if n' is valid configuration & nodeCost ⩽ minCost then
9              minCost := nodeCost
10             S.INSERT(n')
11         else if n' is a contradiction then
12             n'.ANALYZECONFLICT()
13         else if nodeCost ⩽ minCost then
14             Q.INSERT(n', nodeCost)
15     if Q is empty | |C| >= r | Q.PEEK().COSTS() > minCost then
16         searchFinished:= true
17         break
18 return searchFinished, S, Q
```

head of priority queue Q has accumulated costs that exceed the costs of any found solution, given by *minCost*, the cube can be aborted. The reason being that the cube cannot lead to a valid optimal solution.

Comparison of Cube-and-Conquer and Parallel A* Search

In the following, the different approaches of cube-and-conquer and parallel A* Search are compared. Considering cube-and-conquer, it is a two-phase method that is suited for hard SAT instances. For such CNF formulas, cube-and-conquer approaches can generate between thousand and a million cubes, evaluated extensively in [HKWB11] and [BKB+13]. The first phase is executed sequentially, followed by parallel processing of subproblems. Distributing cubes in phase two among all worker threads results in a relatively clear search space partitioning and thus little communication overhead. Worker threads only check the currently known minimum cost solution to decide whether a cube can be aborted. Using local information also reduces the synchronization overhead, especially by not using a global OPEN list. Overall, these advantages lead to a simpler approach with less complexity.
However, the used method partitions the original search tree into sub-trees. These cubes are processed iteratively, which potentially leads to larger search overhead. The reduced communication and iterative processing of cubes can increase the time threads spend on expanding suboptimal parts of the overall search space. This disadvantage is enhanced in SAT instances that are not very hard, for instance a configuration step within a configuration system. Due to the underlying optimization problem (MinCostConf) and lower instance hardness, the search trees usually have a small height and width. Two reasons are that many paths can be pruned and only few conflicts are encountered. Having hard SAT instances, search trees tend to be of greater height and width, reducing potential search overhead.

Algorithm 4.10: Parallel Cubes & Conquer - Thread Process

Input: Clauses F, start-configuration β, linked list of nodes cubes, set of optimal valid configurations S, integer minCost, threadId t

Data: local priority queue Q of nodes

```
1  while cubes is not empty do
2      Q:= initialize queue
3      Q.INSERT(cubes.REMOVEFIRST())
       // process single cube
4      while Q is not empty do
5          n := Q.POLL()
6          N := n.GETSUCESSORS(F, β)
7          forall n' ∈ N do
8              nodeCost := n'.COSTS()
9              if n' is valid configuration & nodeCost ⩽ minCost then
10                 Lock S
11                 if nodeCost < minCost then
12                     minCost:= nodeCost
13                     S := ∅
14                 S.INSERT(n')
15                 Unlock S
16             else if n' is a contradiction then
17                 n'.ANALYZECONFLICT()
18             else if nodeCost ⩽ minCost then
19                 Q.INSERT(n', nodeCost)
           // check for early cube termination
20         if Q.PEEK().COSTS() > minCost then
21             break
```

In contrast to this, the parallel A* Search uses more complex search space splitting strategies, by extensively sharing data. On the one hand, regarding communication and synchronization overhead, Cube-and-conquer introduces the smallest efforts. On the other hand, this benefit leads to the drawback of higher search overhead, especially compared to the centralized A* Search approach. These properties affect the performance depending on the considered CNF formula's complexity.
Summarized, cube-and-conquer potentially performs well on hard SAT instances, for example for Random SAT and Random 3-SAT benchmarks. Regarding industry cases, the performance is dependent on the size of the constructed search tree. Full evaluation, analysis and comparison of the different methods is presented in Chapter 5.

4.4 Parallel Portfolio

An introduction into portfolio solvers is presented in Section 2.4. The main motivation to apply a portfolio solver is the strong performance of this approach in various SAT competitions. One reason for the success of portfolio is given in [HJS10]. The authors state that the portfolio solvers exploit the sensitivity of modern SAT solvers to configuration settings and parameters. Therefore, in the following, two portfolio based solvers are proposed.

4.4.1 Parallel Portfolio Solver

The first portfolio approach combines several instances of a sequential solver (base solver). Each instance has its own set of configuration settings, consisting of several parameters. In the following, important parameters are described and different sets thereof to use in portfolios is shown.

Branching Heuristic

The branching heuristic is responsible to choose an unassigned variable during DPLL, if no unit clauses are available. In cases with many unassigned variables, the function has to choose the most promising variable and corresponding assignment (literal) to let the algorithm find a solution quickly. Many different branching heuristics exist in literature, often focused on modern CDCL solvers. These solvers are conflict driven, thus heuristics focus on literals that often occur in contradictions. A good overview of modern heuristics for CDCL solvers is given in [LGZ+15].
In the domain of product configuration, SAT instances are significantly simpler regarding runtime and occurring conflicts. Additionally, *helper variables* are used to encode use-case specific and complex expressions, for instance to activate feature variables for the user on the interface. For that reason, different score-based branching heuristics are used which are comparable to the One-Sided (OS) and Two-Sided (TS) Jeroslow-Wang Rules (JW) by Jeroslow and Wang in [JW90].

Furthermore, a second component is added as a parameter which influences the priority of choosing variables that have a positive cost (i.e. are not special variables with zero cost). Due to the underlying optimization problem, it can be beneficial to preferably branch over variables with positive cost to limit the search tree growth. Especially the mentioned helper literals are often costless and thus are of lower priority.

Priority Criteria of OPEN list

The used base solver is a sequential A* search algorithm that uses a priority queue Q to store all nodes that may be expanded. To ensure optimality, the first priority criteria of Q is the cost evaluation $f(n) = g(n) + h(n)$, with n being a node. However, in case multiple nodes in Q are of equal cost, additional priority criteria are necessary. These criteria have an impact on the search behavior, defining which parts of the search space are expanded first. Hence, the instances in the portfolio use different strategies to prioritize nodes within Q.
As secondary priority criteria, two different ones are used. The first ones evaluates the current depth of each node within the search tree. This criteria can prefer nodes that are either have a small depth or high depth. The former encourages exploration within the search space, the latter assumes that solutions tend to be further down in the search tree. The second criteria assesses the number of unresolved clauses in node n. Again, the assumption is that a node with few unresolved clauses is closer to a valid assignment than a node with many unresolved clauses, considering that both nodes have equal costs.

Clause Learning

Clause learning is a prominent improvement used in many modern SAT solvers. A short overview is given in Section 2.2.3. The goal is to prune paths within the search tree that lead into already encountered conflicts. The reduced complexity of common SAT instances within product configuration reduce the effectiveness of clause learning compared to instances such as Random SAT. Additionally, most configuration steps are satisfiable, only few require to prove unsatisfiability. Lastly, the employed base solver is based on the A* search algorithm, which expands the search tree in best-first manner. This reduces

the effectiveness of non-chronological backtracking. Thus, the main benefit is utilizing the learned clause.
Conclusively, clause learning is not always beneficial in this environment, due the introduced overhead. Regarding the parallel portfolio, some instances utilize clause learning and others do not.

Portfolio Configuration

With various defined parameters, a portfolio can be constructed by combining solvers with different configuration sets. The diversification's purpose is to have a portfolio with solvers that have little search space overlap to reduce search overhead and the solver's sensitivity to parameters. A summary of preconfigured base solvers to use in a portfolio is shown in Table 4.1. The set of settings is handcrafted, due to strong impact and domain-specific knowledge. A similar approach is used for solvers such as ManySAT in [HJS10].

Strategies	Instance 1	Instance 2	Instance 3	Instance 4
Branching Heuristic	OS-JW	OS-JW	TS-JW	TS-JW
Prefer Cost Literals	true	false	true	false
Secondary Priority Criteria	Max Tree Depth	Minimal Unresolved clauses	Max Tree Depth	Minimal Unresolved clauses
Clause Learning	false	true	false	true

Table 4.1: Different strategies used for a parallel portfolio solver. Parameters are focused on the branching heuristic, ordering of the OPEN list and clause learning. The abbreviation OS-JW and TS-JW stand for "One-sided Jeroslow-Wang" and "Two-sided Jeroslow-Wang", respectively. Objective is to configure the different instances with settings that reduce the amount of search space overlap.

Comparison between Parallel Portfolios and Divide-and-Conquer

Comparing parallel portfolios to divide-and-conquer approaches, several distinctions can be drawn. A major advantage is the robustness of parallel portfolios against the impact of configuration parameters that SAT solvers generally face. By using multiple complementary instances of a base solver with varying parameter sets, peaks that result from suboptimal parameters are reduced. Having many instances increases the likelihood of utilizing good settings. The presented divide-and-conquer approaches use the same parameters across all processing cores, thus being more sensitive. Furthermore, parallel portfolios are less complex than divide-and-conquer strategies. Not working on larger central data structures reduces the required synchronization and communication efforts.

However, considerable disadvantages exist, especially for industrial use cases. The main problem is that the parallel portfolio does not partition the search space into disjunct portions, but rather duplicates the Boolean formula for each instance. As a result, the memory consumption increases approximately proportionally to the number of cores. For industry cases and production systems, resources are limited. This duplication also results in a lot of search overhead, since multiple instances work on the same portion of the search space, looking for the optimal assignment. Divide-and-conquer fosters more cooperation between the cores, lessening the search overhead. Lastly, the shown portfolio configuration is chosen manually. Potential scalability is limited, which requires an automated approach to find good parameter sets, also pointed out in [HJS10]. An in-depth performance comparison is shown in Chapter 5.

4.4.2 Parallel Portfolio A*

An alternative strategy is to adapt the parallel A* search algorithm presented in 4.2. The main motivation is to avoid the Boolean formula duplication that is performed by the

previously shown parallel portfolio. Instead, the search space is divided but multiple threads work cooperatively on the same instance. To adopt the benefit of being less sensitive to parameter tuning, each thread uses different settings but work on the same formula. Regarding parameters, changing the branching heuristic and related settings such as the "Prefer Cost Literals" can be changed for each processing unit (for comparison, see Table 4.1). Hence, when a thread processes a node and has to choose the next decision literal for that path, a thread specific heuristic is utilized. Usually, these parameters have a strong impact on the search behavior with respect to the order of path expansions.
This strategy can be applied to both, centralized and decentralized parallel A* search. Despite using varying parameters, no other changes are required for these algorithms.

4.5 Deterministic Search

Sequential SAT solvers can deliver reproducible results, as long as the core functions such as unit propagation, usage of heuristics (e.g. branching decision) etc. are deterministic. In contrast, most current parallel SAT solvers are not capable of producing stable results, due to their architectures relying on weak synchronization ([HJPS11]).
Nevertheless, reproducibility of configuration processes is a key requirement for a configuration system, because the result is not only "SAT" or "UNSAT" but an optimal assignment, for which even minor differences can be discovered by the user. Therefore, during a configuration process, a user expects deterministic behavior in cases where entire configuration processes or only steps are performed repetitively. This section presents problems introduced by parallel SAT algorithms, their influence on the optimality of configurations as well as measures to ensure deterministic behavior for the user.

Given a configuration process, modeled by a chain of configuration steps, every execution of this chain should return the exact same assignments for each step. Elementary scenarios are undo and redo functionalities which have to result in reproducible outcomes. Offering non-deterministic configurations to the user may lead to confusion and frustration. Although the presented algorithms always return a set of up to r configurations, it is not guaranteed that any two executions of the parallel algorithms return the exact same solutions, i.e. the exact same set of variable assignments. For example, multiple valid configurations exist but the user only requests one. If non-deterministic behavior is introduced, e.g. by parallel algorithms, different valid assignments are returned when repeating the configuration step numerous times.

Considering a configuration step, the number of optimal (lowest cost) solutions can vary. Therefore, the user can decide how many optimal solutions should be returned (in case more optimal solutions exist). Generally, if several solutions are returned, they are presented as alternatives to the user to choose from. Hence, useful values for r are usually one to five to not overwhelm the user with alternatives.
To ensure that the computed set of solutions with size m ($m \leqslant r$) always contains the same ones, i.e. deterministic behavior for the user, the configurations need an order among them. For instance, the user requests up to three solutions for a configuration step, but five optimal ones exist, the configuration system has to return the same three solutions consistently. The following multiple criteria are used to order configurations as the foundation to achieve reproducible configuration steps:

1. Cost of the configuration using the delta-cost function shown in 3.1
2. Number of decision literals (tree-depth, ascending)
3. Number of literals contained in the solution (ascending)
4. Hash of literals contained in the solution

The first deciding factor is the solution's cost. Between two configurations with unequal costs, the one with lower costs is prioritized, because the changes made to the start-configuration are evaluated as smaller. Thus, additional sorting criteria are only used between equal-cost solutions. All criteria are sorted from lowest to highest (ascending). An important factor for all criteria is the stability across several executions, which is fulfilled for the enumerated ones.

During parallel search in the search tree, solutions can be found in varying order across several executions. An execution describes the computation of a specific configuration step. Main reason for this nondeterminism is that parallel working threads expand nodes in inconsistent time, which leads to a potentially different order of node visits for each execution. Due to the unstable expansions, solution can be found in varying order. To accommodate this instability, the termination criteria from 4.2 is adapted to ensure that previously returned solutions are not skipped through early termination in another execution of the same configuration step. Therefore, two alterations are presented in the following, both aiming at ensuring deterministic behavior as well as minimizing the amount of search overhead.

4.5.1 Tree-depth Depending Termination

In previously shown algorithms, two search termination criteria are used. Firstly, the search is aborted in case all other nodes are more expensive than an already found configuration. Secondly, if the requested amount of solutions has been found (all of equal cost) and all other nodes are at least equal-cost, the search can be terminated.
Especially the second criteria is unstable due to the varying sequence of finding solutions. Using this termination strategy, always the first m reached solution states are collected which prevents reproducibility. A naive strategy is to not use this secondary early-termination criteria, but it is important to improve the performance by avoiding unnecessary node expansions (search overhead).

The first approach exploits the number of decision variables in any obtained solution. As soon as the requested amount of alternative configurations r has been found, the maximum tree-depth (number of decision variables) j is extracted from all found solutions. This indicates the solution that is ordered last among all solutions, thus it can be used to prune unexplored nodes. Consequently, only paths within the search tree that either have lower costs than already known valid assignments or are equal-cost and have a tree-depth less or equal to j are expanded. For subsequently found equal-cost solutions, the maximum j may be updated to reflect the new upper bound, in case it has fewer decision variables. This procedure incrementally reduces the depth of j and consequently the number of paths that may be expanded. As soon as no viable nodes can be processed, because all are more expensive or equal-cost and at deeper levels, the search can be terminated. This strategy ensures that always the same m solutions are returned, because all paths are expanded that contain the m optimal solutions with the smallest number of decision literals.

Figure 4.5 shows an explanatory abstract DPLL search tree for a configuration step. In this example, the user requests two alternative solutions ($r = 2$), but four valid optimal solutions exist (green nodes). To ensure deterministic behavior for repeating executions, the same set of solutions has to be returned consistently. Using the multiple-criteria sorting order shown in Section 4.5, node n_6 and n_{11} have to be returned as the optimal configurations, because they have the lowest tree-depths (with two and three decision literals, indicated by the edges) of all optimal solutions.
During parallel search, the best-first-search does not guarantee to find the optimal assignments in correct order. With $r = 2$, three explanatory scenarios are discussed to explain the procedure:

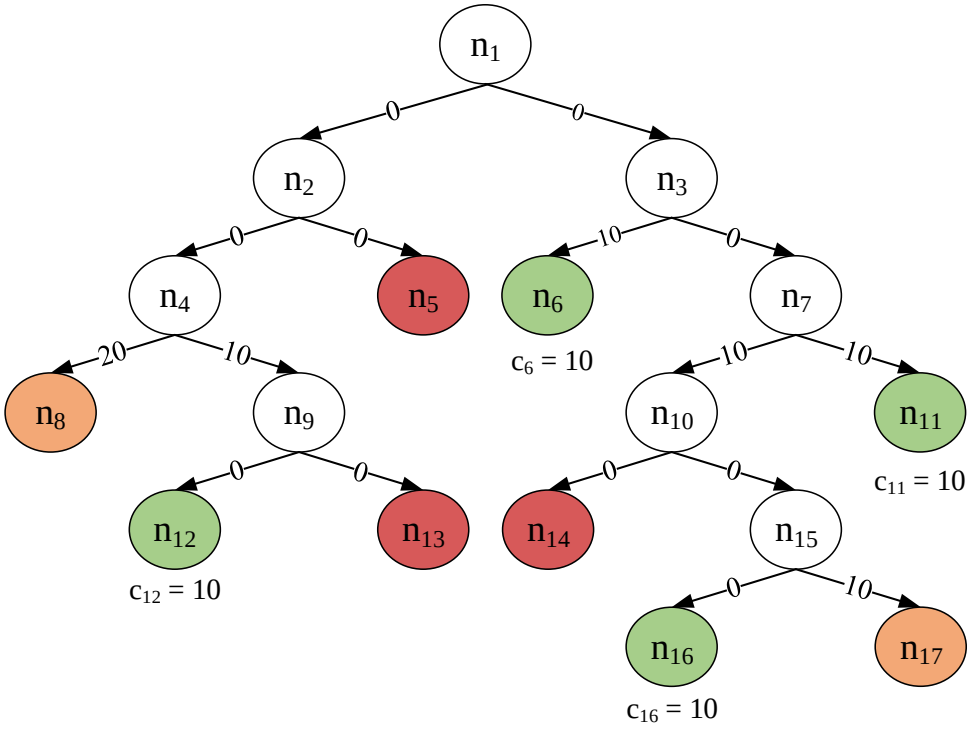

Figure 4.5: A DPLL search tree for a configuration step. White nodes indicate explored states, red nodes display conflicts, green nodes are valid solutions and orange nodes indicate states that are not further expanded due to their costs. The weighted edges indicate added literal costs through unit propagation.
Given a user wish and a requested amount of solutions $r = 2$, the objective of the search termination algorithm is to ensure that always the same two configurations of the four possible ones are returned. The correct nodes to return in this example are nodes n_6 and n_{11} due to lowest tree-depths (two and three decision literals, respectively) among the four equal-cost solutions. Depending on the order in which optimal assignments are found, specific paths that are equal-cost to the current optimum can be pruned, when r solutions have been found. The requirement for pruning is that the current node of the path has a higher depth than all currently hold solutions. Thus, in case that nodes n_6 and n_{11} are expanded first, which result in two solutions, nodes that have equivalent costs ($= 10$) and are on tree level four or larger can be pruned.

1. Nodes n_6 and n_{11} are expanded first
2. Nodes n_6 and n_{12} are expanded first
3. Nodes n_{12} and n_{16} are expanded first

The first scenario is the best-case behavior, in which the desired solutions are found first through expansion of nodes n_6 and n_{11}. Afterwards, the maximum tree-depth j is set to 3 and paths are only further explored up to a tree-depth of 3, as long as they are less or equal in cost. In this example, nodes in the range of n_{12}-n_{16} are not expanded, if they have not been processed already. Node n_8 is not expanded as soon as a solution is found, because the costs ($c_8 = 20$) succeed the currently known minimum costs ($c_6 = c_{11} = 10$). In the second scenario, j is set to 4, due to the valid configuration of node n_{12}. The maximum depth has to be used, otherwise solution n_{11} would be missed through pruning. Paths with less or equal to four decision literals have to be extended before the search can

be terminated. Thus, n_{11} has to be expanded which leads to a new solution. This solution is favored against n_{12} owing to requiring fewer decision literals. Therefore, n_{11} replaces the solution n_{12}, j is set to 3 and paths are only extended up to a length of three decision literals as long as the cost is equivalent to the currently known solutions. Paths with lower costs are always extended.
Finally, case 3 displays the worst-case, in which all solutions are explored. Firstly, nodes n_{12} and n_{16} are processed which sets $j = 5$. All nodes that are cheaper or equal-cost and have at most five decision literals are expanded. Subsequently, node n_6 or n_{11} are found which will update j accordingly. Independent of the order of n_6 and n_{11}, $j = 4$ is set due to the solution n_{12} which replaces n_{16}. This ensures that the two preferred solutions are returned.

This example shows that for any case with several optimal solutions, the same configurations can be returned for repetitive executions using parallel search. The requirement of reproducibility is fulfilled. Furthermore, this strategy enables pruning techniques, by updating the maximum tree-depth j, speeding up the search in cases in which multiple optimal solutions are prevalent. Strong benefit of this technique is its applicability for all presented parallel algorithms, i.e. parallel A* search, cube and conquer, and parallel portfolios.

Advanced Tree-depth Depending Termination

This strategy can further be improved by not only utilizing the tree-depth (vertical), but also the position within one level of the search tree (horizontal). For instance, all z optimal solutions share the same tree-depth, but only at most r alternative configurations are requested. Using the presented *tree-depth depending termination*, all z solutions have to be computed, since the sequence of finding them is unstable using multiple threads. The larger z is, the higher is the search overhead (in worst case $z - r$ avoidable nodes are expanded). To circumvent this, the multiple-criteria order of solutions shown in 4.5 is adapted by adding the number of left edges in the path from root to the node n. The number of left edges in a path is abbreviated with "left-branches" in the following.
After having found r solutions, the minimum tree-depth as well as the maximum number of left-branches on that level are shared with all threads to prune paths. The value can be updated after having found another solution on a smaller or equal level. In the former case, the value is always updated because the new solution has a smaller tree-depth. In the latter case, the value is only updated if the number of left-branches is larger than the minimum of all currently hold solutions on that level. This strategy reduces the number of expanded nodes, when several solutions are on the same level.

Implementation of Tree-depth Depending Search Termination

Implementation wise, two methods to facilitate the adapted search termination detection are described. First, the priority queue Q, holding all nodes that can be expanded, uses a sorting criteria. This criteria can be extended to use the cost as well as the depth and number of left-branches. The search can be terminated if the head of the queue is more expensive than known solutions. Furthermore, given that r solutions have been found already, the procedure can be aborted if the head of Q is equal-cost and ranked behind the least preferred known solution, using the adapted multiple-criteria order.
This approach is valid, because priority queue and solutions share the same sorting criteria. However, the approach has one major drawback. It changes the order of nodes within the priority queue Q, which strongly influences the parts of the search tree that are explored first. Using the described criteria, nodes that are further up in the search tree are expanded first, which translates to nodes with fewer decision literals. Under the assumption that solutions are found within less node expansions at lower levels (more decision literals

added), due to a simplified Boolean formula, the performance can be influenced negatively. Tests show that the changed sorting criteria leads to more expansions by exploring larger portions of the tree, resulting in increased overall computation time. Thus, the priority queue's sorting criteria should not be changed.
Alternatively, the termination criteria is simplified to only abort if the next node on the queue has a higher cost evaluation than any found solution. Additionally after having found r solutions and the search not being terminated, every node that is taken from the queue is checked whether the path can be pruned to stop further expansions. To prune a path, the current node has to have equal-cost to the currently known minimum as well as a greater or equal path length compared with the shortest path of any currently optimal found solution. In case the path length is equal, it is additionally required to have less left-branches than any known solution on the minimum depth level.

4.5.2 Limited Node Expansion Termination

For the presented cube-and-conquer approach (4.3), a simpler deterministic search termination detection can be used as well. The distribution of cubes after the first phase consistently assigns a set of work to each thread. Each worker thread processes the cubes in the same order without exchanging them, hence the initial search space partitioning is consistent for repeating configuration steps.
This property can be exploited to terminate cubes early, by limiting the amount of nodes each thread is processing after r solutions have been found. For this, the sorting criteria shown in Section 4.5 is changed. Having two equal-cost solutions, the one that originated from an earlier cube is preferred (each cube has a position within its thread, indicating the cube processing order):

1. Cost of the configuration using the delta-cost function shown in 3.1
2. Cube position within a thread (ascending)
3. Number of literals contained in the solution (ascending)
4. Hash of literals contained in the solution

Each cube can be terminated in case it has higher costs than any found assignment (Algorithm 4.10, line 18). Now, a cube is also aborted given two conditions: Firstly, r valid assignments have been found. Secondly, the head of the priority queue used for the current cube is equal-cost and the cube's position (indicating its processing order within the specific thread) is higher than of all found assignments. For example, with $r = 2$, the respective solutions are found by processing cube 2 and 4. Consequently, threads can terminate cubes earlier starting with the fifth cube, whenever the most promising node is equal-cost to currently known solutions. However, this approach is not applicable to parallel A* search, because threads are sharing nodes which lead to unstable node processing order.

5. Experiments

This chapter presents an evaluation and comparison of the developed algorithms in Chapter 4. As a baseline, the existing *CAS Configurator Merlin* ([CAS20]) is used. The presented parallel algorithms have been implemented into the existing application to compare them to the sequential algorithm included in the current version.
At first, the test setup is explained including test data and test environment. Afterwards, the experimental results are shown with focus on wall clock time (elapsed real time), scalability and resource consumption. Lastly, a discussion of the experimental results is presented.

5.1 Test Setup

Test Algorithms

The experiments include all parallel algorithms presented in Chapter 4 as well as the sequential algorithm as baseline. A short summary is given:

1. *Centralized Parallel A* Search* (CA*) (4.2.1): Extension of the sequential A* search using multiple threads on a single OPEN list.
2. *Decentralized Parallel A* Search* (DA*) (4.2.2): Extending the A* search by assigning an OPEN list to each thread. Nodes are exchanged among the workers.
3. *Parallel Cube-and-Conquer* (C&C) (4.3): Two-phase approach. Firstly, the problem is decomposed into subproblems. Afterwards, the cubes are processed in parallel.
4. *Parallel Portfolio* (PP) (4.4.1): Combining multiple sequential base solvers with different configurations.
5. *Parallel Portfolio A** (PPA*) (4.4.2): Extending parallel A* search by using different settings for each worker thread.

Test Data

The data used for the experiments consists of various real industry cases as well as random SAT and random 3-SAT instances.
Concerning the industry cases, different *rule sets* are used, whereby each rule set reflects one configuration model. Furthermore, the algorithms are developed for interactive configuration processes. Thus, a set of configuration processes is available to perform measurements.

Each configuration process consists of a chain of configuration steps, each step is triggered by a user change. Random SAT and random 3-SAT are also used to have a comparison on well known problem instances used in literature. For random 3-SAT, the instances "Random-3-SAT Instances with Controlled Backbone Size" by Josh Singer are used, which are obtained from SATLIB [1]. The random SAT instances are part of the "JNH" benchmark by John Hooker, also taken from SATLIB.

Table 5.1 shows a comparison of the different rule sets. It contains key properties of the used rule sets, including the number of clauses, variables, ratio of clauses to variables, and average clause length. For the industry cases, an empirical domain complexity is stated. A major difference between industry cases and random SAT instances are the number of variables and clauses. The rule sets derived from configuration models show a high number of clauses and variables, because they represent complex product knowledge bases. Furthermore, additional clauses and variables are required to control product configuration specific logic, for instance helper variables to activate specific features to be editable for the user. Considering the average clause length, the configuration models and random tests have comparable numbers. In most cases, the random SAT instances have a significantly larger clause to variable ratios. The assumption is that instances with higher ratios are harder to solve, because more clauses have to be fulfilled. Additionally, fewer satisfying assignments exist. This is not entirely applicable to the MinCostConf task, due to the following reasons:

- An optimization problem is solved. Less clauses imply that more valid assignments exist that could be optimal.
- By using a preexisting *valid* start-configuration and a user wish, only small changes are usually required to find a valid assignment.

Rule Set	Domain Complexity	# Clauses	# Variables	Ratio	Avg. clause length
RS1	High	17157	8483	2.02	3.98
RS2	Medium	4803	8318	0.58	3.34
RS3	Low	8023	1722	4.66	4.70
R3SAT	-	449	100	4.49	3
RSAT	-	850	100	8.50	5.8

Table 5.1: Comparison of different rule sets (Boolean formulae). RS1 to RS3 show formulae of industry cases. R3SAT are random 3-SAT instances and RSAT stands for random SAT instances. Ratio describes the clause to variable ratio of the resulting Boolean formulae.

Hardware and Software

For this work, all algorithms are implemented into the existing configuration system. This is based on Java 11[2] and Java Enterprise Edition (JEE) 7 [3]. All benchmarks are run on the application server WildFly 15[4]. The application is run on the following system:

Computer: Intel Core i7-7820HQ, 16 GB RAM, Windows 10 (64-Bit)

[1] Aailable from https://www.cs.ubc.ca/ hoos/SATLIB/benchm.html
[2] Available from https://www.oracle.com/java/technologies/javase-downloads.html
[3] Available from https://docs.oracle.com/javaee/7/index.html
[4] Available from https://wildfly.org/

Test Procedure

The entire system is tested using automated simulation of configuration processes. Each test for a rule set consists of n configuration steps. Every configuration step is repeated three times to have more accurate mean wall clock times. Therefore, for each rule set $3n$ data points are available. The usage of simulated configuration steps yields a mixture of small and larger configuration changes with varying response times. Considering randomly generated tests, 23 random SAT and 57 random 3-SAT instances are used (hence 80 data points). For each tested algorithm configuration, a warm-up phase is executed for the Java Virtual Machine (JVM). Despite measuring different metrics, the usage of automated tests also ensures the correctness of all implemented algorithms.

5.2 Experimental Results

The experimental results are split into three paragraphs. First the calculation times are compared, secondly the scalability using different number of threads is analyzed, and lastly the search overhead is shown.

5.2.1 Calculation Time

The results of measuring the wall clock time for different rule sets are shown in Figure 5.1, 5.2, 5.3, 5.4, and 5.5. Each figure compares the parallel algorithms using 4 threads to the sequential solver (baseline). The y-axis shows the wall clock time required for each configuration step, using a logarithmic scale. The x-axis displays the problem instances sorted in ascending order with respect to the time required to solve them.

Considering the results of the real industry cases (RS1, RS2, RS3), several observations can be made. In all three cases, the two parallel A* search approaches perform best on demanding configuration steps. Of the two parallel A* strategies, the centralized one using a single OPEN list performs better across the various rule sets. The cube-and-conquer (C&C) approach performs worse than parallel A* search but better than the parallel portfolio (PP). Comparing the C&C results, it performs best on RS1 which is the most complex one. On the ”medium’ complexity rule set RS2, C&C does not outperform the sequential solver consistently. The presented PP solver performs worst on all three cases. For most problem instances, it performs worse than the sequential solver, only on some problem instances in RS1 it outperforms the baseline.
The graphs also point out that the parallel A* algorithms add overhead, especially on the low complexity problem instances measured by required response time. For example, on RS2 the break-even point is around 20-25 milliseconds (ms). Up to this point, the parallel A* algorithms perform worse than the baseline. The added overhead is larger for the decentralized approach than for the centralized one. The results are similar for RS1 and RS3. C&C and PP do not introduce this overhead on simple instances, due to using a sequential solver in the first phase (C&C) or as a base solver (PP).

Following the break-even point, the speedups of the parallel algorithms over the sequential solver are increasing. Parallel centralized A* search (CA*), the best performing algorithm, reaches consistent speedups of 2 to 3 for hard configuration tasks.
For RS1 shown in Figure 5.1, the maximal speedup is around 2.8 to 3.2, achieved for the hardest problem instances in which the sequential solver requires 4000-8000 ms. In the range of 100-3000 ms (baseline) the speedup is between 2 and 2.5. The speedup using the decentralized A* search ranges from 2 to 2.65 over the range of 600-8000 ms. The results of C&C show a maximum speedup of 1.89, however the speedup for large portions of the problem instances is between 1.2 and 1.5. Lastly, the portfolio solver has a maximum speedup of 1.32. The values are between 1.1 and 1.3 in the range up to 80 ms baseline

time. Subsequently, the speedup is decreasing quickly and it is performing worse than the sequential solver for many problems.

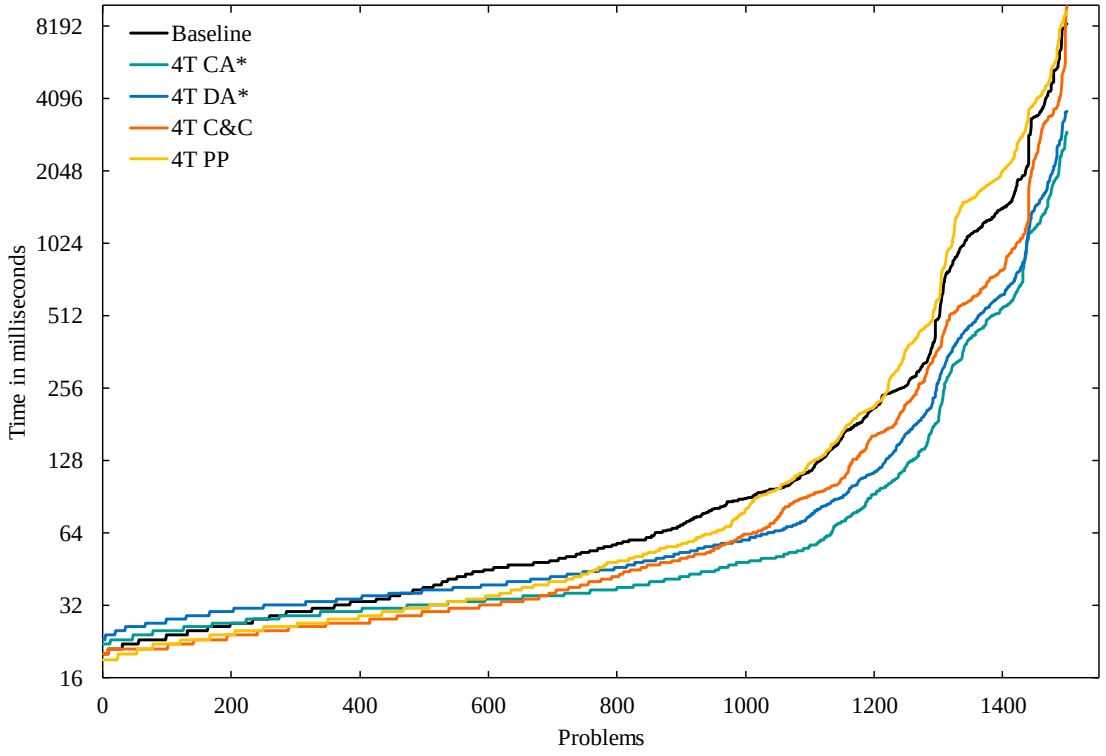

Figure 5.1: Per task time comparison of parallel algorithms using 4 threads on the high complexity rule set RS1.

The speedup for the ”medium” result set RS2 (Figure 5.2) is lower across all parallel algorithms. CA* search reaches a reduction factor of up to 2.6. For a wide range, from 80-1000 ms the speedup is between 2 and 2.5. The other approaches perform significantly worse. DA* search peaks at a speedup of 2.17. In the range from 80-1000 ms is varies between 1.2 and 1.8. C&C does not achieve consistent speedups, only smaller portions of tasks have a speedup of up to 1.25. Again, the portfolio solver does not outperform the sequential algorithm.

Considering the rule set with the lowest complexity, RS3 in Figure 5.3, the results are two-sided. On the one hand, the parallel algorithms perform worse than the sequential solver for the short running configuration steps (<30 ms). Especially parallel A* approaches add a noticeable amount of overhead. On the other hand, for configuration tasks that require more than 100 ms on the baseline, the two parallel A* variants outperform all other solvers. Consequently, the average per task speedups are lower. CA* achieves a peak speedup of 2.57 but for many tasks performs worse than the baseline. The DA* strategy peaks at a speedup of 2.15 but has even more overhead than its centralized version on easier instances. C&C and the PP solver achieve a speedup of up to 1.64 and 1.53 respectively. However, their overhead is much smaller overall. Summarized, it shows that a considerable improvement is achieved over a large variety of configuration steps. Additionally, a detailed comparison is shown in Table 5.2. The table also displays overall speedup and the average per task speedup for tasks that require more than 100 ms, which is a threshold after which users might experience a slow down in their configuration process.

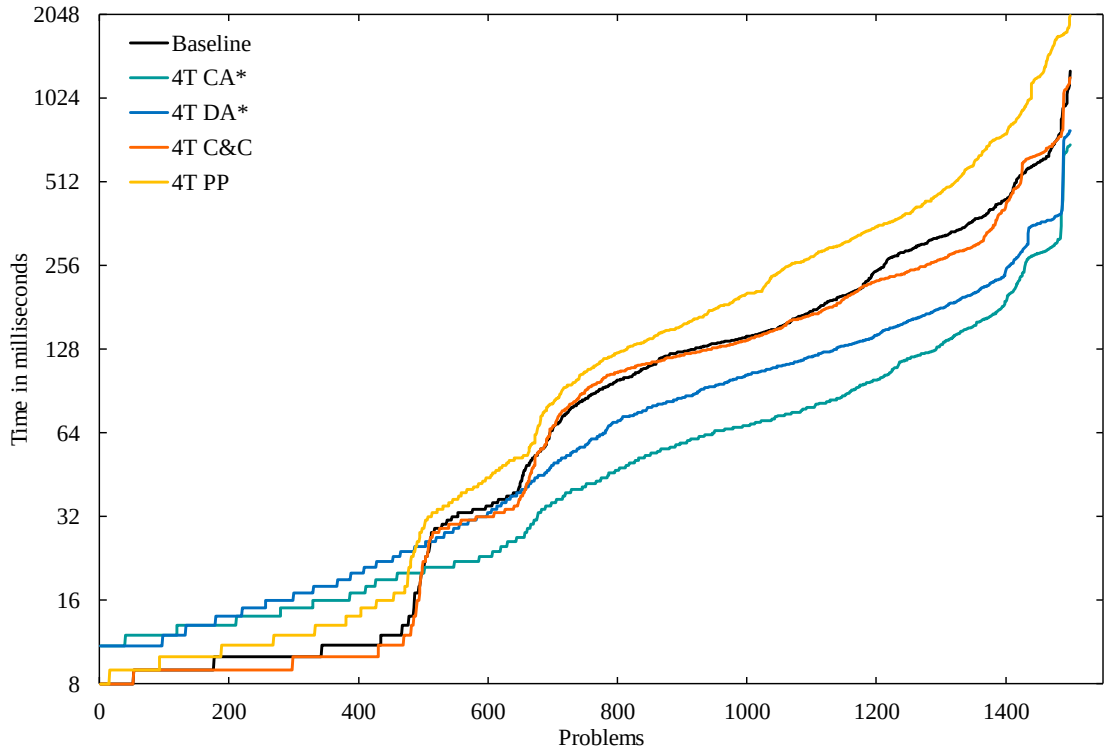

Figure 5.2: Per task time comparison of parallel algorithms using 4 threads on the medium complexity rule set RS2.

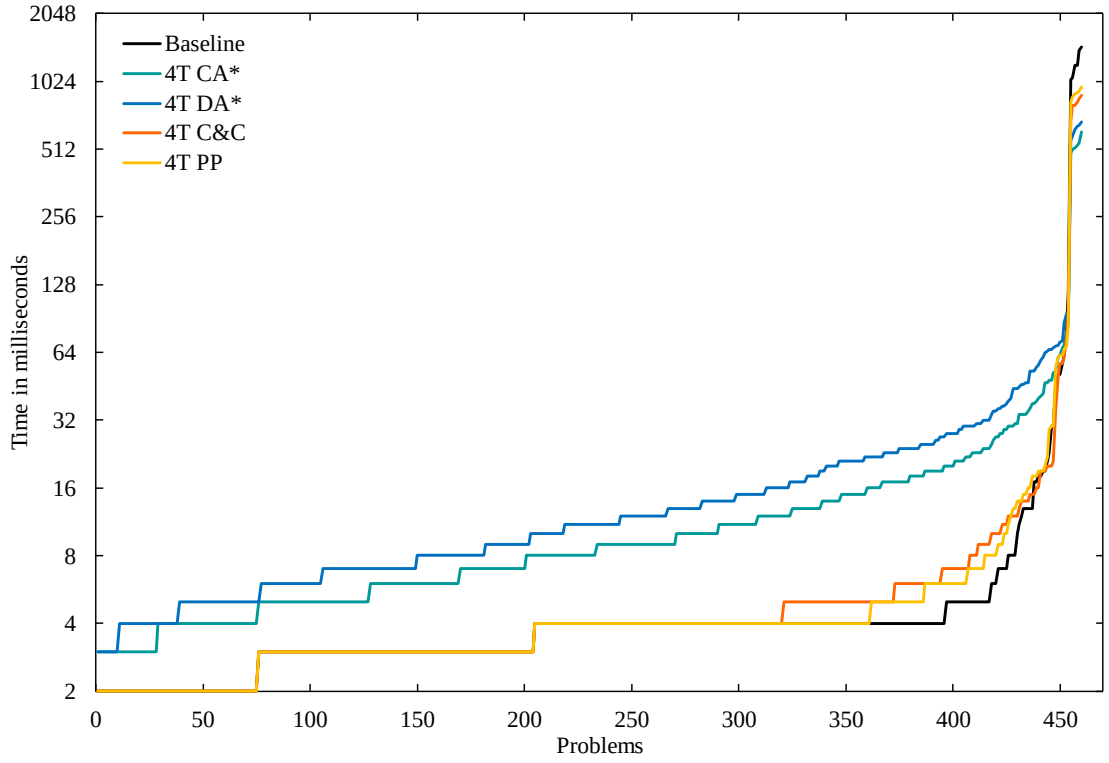

Figure 5.3: Per task time comparison of parallel algorithms using 4 threads on the low complexity rule set RS3.

Rule Set	Speedup	CA*	DA*	C&C	PP
RS1	Max	3.20	2.66	1.89	1.32
	Overall	2.50	2.08	1.44	0.83
	Avg. per task	1.63	1.35	1.32	1.07
	Avg. per task >100 ms	2.44	1.97	1.46	0.84
RS2	Max	2.61	2.17	1.25	1.00
	Overall	2.06	1.55	1.05	0.63
	Avg. per task	1.61	1.20	1.05	0.76
	Avg. per task >100 ms	2.31	1.70	1.09	0.63
RS3	Max	2.57	2.15	1.64	1.53
	Overall	1.13	0.90	1.33	1.24
	Avg. per task	0.47	0.37	0.94	0.96
	Avg. per task >100 ms	2.16	1.87	1.48	1.37
R3SAT/RSAT	Max	2.76	3.23	3.27	1.24
	Overall	1.72	2.06	2.46	0.97
	Avg. per task	1.78	2.21	2.51	1.00
	Avg. per task >100 ms	1.78	2.22	2.54	1.00

Table 5.2: Comparison of achieved speedups for parallel algorithms running 4 threads using different rule sets. Overall speedup compares the cumulative time over all configuration steps. "Avg. per task >100 ms" only considers configurations steps that require more than 100 ms using the sequential solver (baseline).

Comparing parallel algorithms using random SAT instances, a contrasting ranking is shown in Figure 5.4 and 5.5. On both problem types, the C&C performs well. The DA* search performs comparably to C&C and outperforms its centralized version. Similarly to the industry cases, the worst performing parallel algorithm is the portfolio solver.
Due to the relatively long baseline computation time, compared to configuration tasks, the parallel algorithms do not show a significant overhead on the simpler random SAT instances. Regarding the achieved speedups, detailed in Table 5.2, the C&C peaks at 3.27 with an per instance average of 2.51. DA* ranks second with max speedup of 3.23 and averaging 2.22 per task. CA* is considerably slower with a maximum of 2.76 and average of 1.78. No consistent improvement is gained by the portfolio solver with an average speedup of 1.

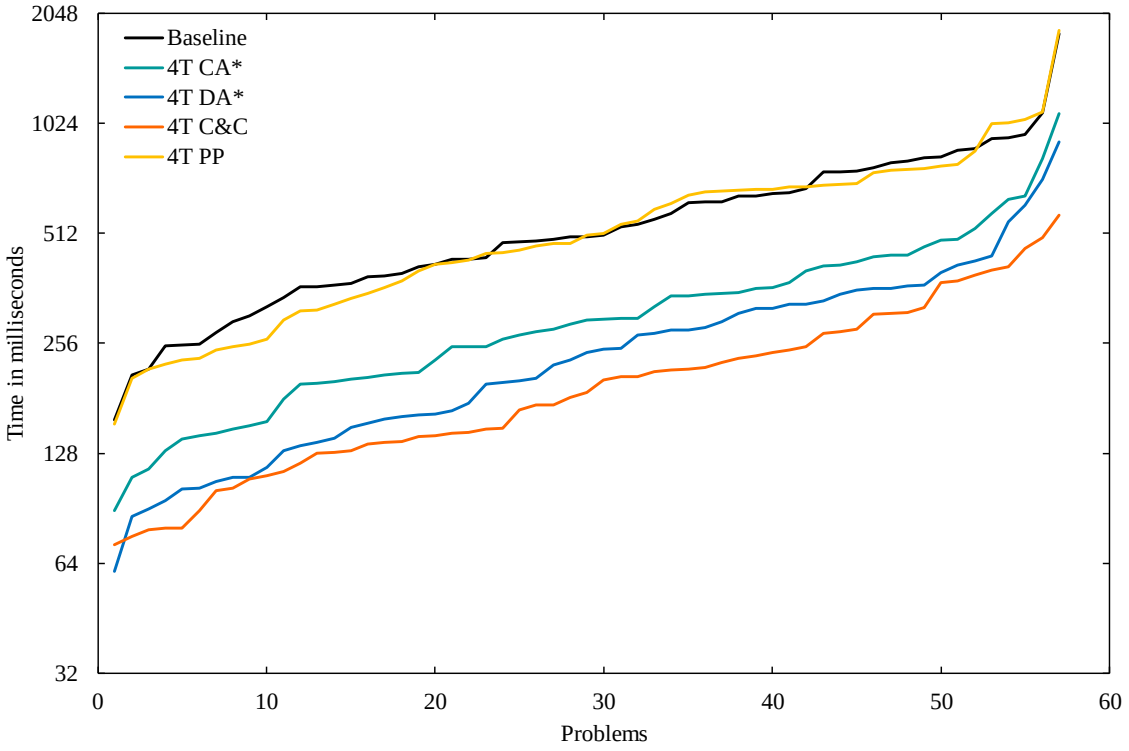

Figure 5.4: Per task time comparison of parallel algorithms using 4 threads on R3SAT.

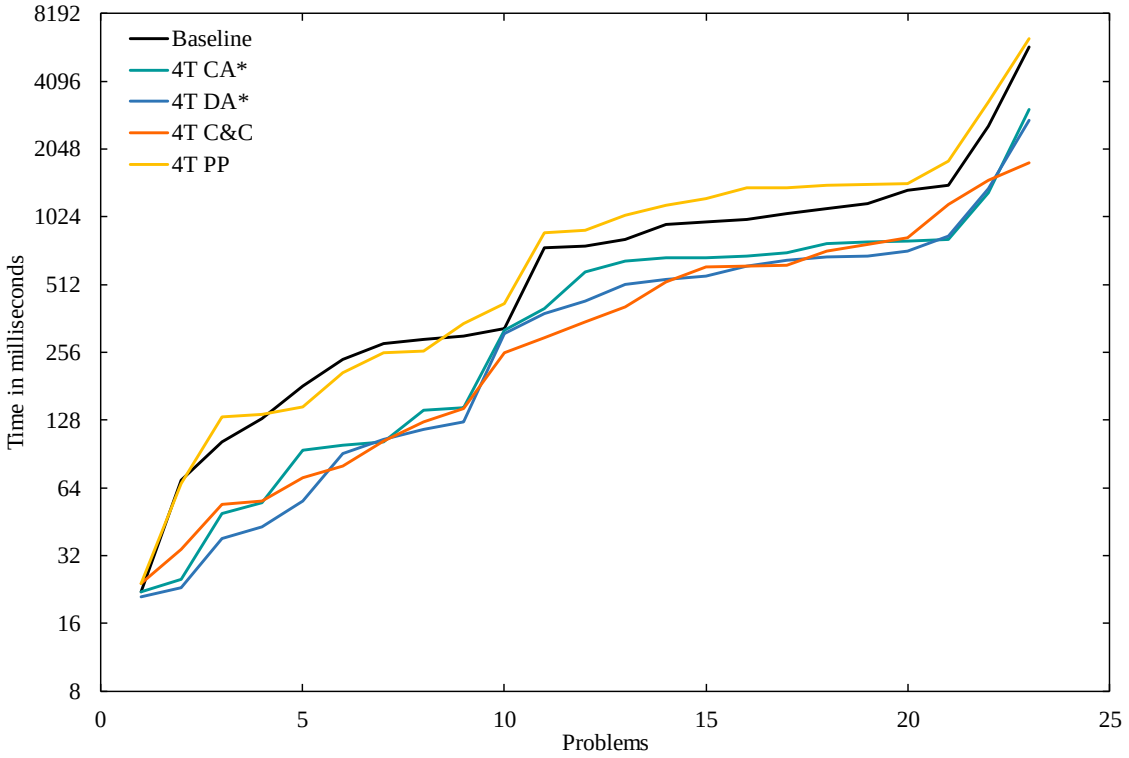

Figure 5.5: Per task time comparison of parallel algorithms using 4 threads on RSAT.

Lastly, due to the mediocre results of the presented parallel portfolio, the results of the proposed Parallel Portfolio A* (PPA*) as an adaptation of CA* are compared as well. Table 5.3 shows all metrics for the different algorithms. ”CA*” and ”CA* Generalized”

differ in the used settings. For all prior tests, each rule set provided a set of parameters used for the different algorithms that are supposed to fit the problem instances well. Using a portfolio approach, varying settings are used for the different threads. Thus, the manual process can be avoided. To have a comparable result from the regular CA* search, another test was performed without using the rule set specific settings but rather using one setting across all tests (CA* Generalized).
The results indicate the PPA* performs significantly better than the regular PP algorithm. This can be explained by the fact that the A* portfolio is an adaption of CA* search. Furthermore, comparing the results to CA* and CA* generalized, the portfolio performs very similar to the CA* search with constant parameters, depending on the rule set. The benefit of the PPA* is the fact that no parameter preselection is required.

Rule Set	Speedup	CA*	CA* Generalized	PP	PPA*
RS1	Max	3.20	3.20	1.32	2.95
	Overall	2.50	2.50	0.83	2.40
	Avg. per task	1.63	1.63	1.07	1.59
	Avg. per task >100 ms	2.44	2.44	0.84	2.43
RS2	Max	2.61	2.40	1.00	2.44
	Overall	2.06	1.81	0.63	1.94
	Avg. per task	1.61	1.33	0.76	1.46
	Avg. per task >100 ms	2.31	2.12	0.63	2.16
RS3	Max	2.57	2.93	1.53	3.04
	Overall	1.13	0.97	1.24	1.00
	Avg. per task	0.47	0.36	0.96	0.37
	Avg. per task >100 ms	2.16	2.51	1.37	2.64
R3SAT/RSAT	Max	2.76	2.76	1.24	2.77
	Overall	1.72	1.72	0.97	1.72
	Avg. per task	1.78	1.78	1.00	1.78
	Avg. per task >100 ms	1.78	1.78	1.00	1.78

Table 5.3: Comparison of portfolio based solvers and the CA* search. "Generalized" describes the fact that one parameter set was used for all tests. All solvers utilized 4 threads.

Impact of Determinism

All before presented results include the deterministic versions of the algorithms. To determine the approximate impact of the stricter search termination criteria, the best performing algorithm is considered, centralized parallel A* search. For the evaluation, the most complex benchmark RS1 is used which contains a wide variety of configuration tasks. The impact is measured by the overall speedup of the non-deterministic version over the deterministic version of CA* search. The results are shown in Table 5.4. Summarized, the added overhead is approximately between 2-3% measured on RS1. The overall performance cost is 2.4%, if only long runnings tasks (>100 ms) are considered the cost is measured with 3.8% in this test.

Algorithm	Overall Runtime	Speedup Overall	Runtime >100 ms	Speedup >100 ms
Deterministic CA*	226.1 s	1	190.5 s	1
Non-Deterministic CA*	220.8 s	1.024	183.5 s	1.038

Table 5.4: Comparison of deterministic and non-deterministic CA* search using RS1. Each solver utilized 4 threads. Deterministic CA* is the reference, thus the speedup is 1. "Runtime/Speedup >100 ms" only account configuration steps that require more than 100 ms on the reference solver.

5.2.2 Scalability

One goal of this thesis is to conceptualize approaches that scale well with increasing numbers of processing units. In Table 5.5 the speedup of the different parallel algorithms is displayed using varying numbers of threads.
Most noticeably is the overall scalability with rule set complexity. Largest improvements are attained on RS1 with an overall speedup of 2.5 for CA*. On the easier problem domains RS2 and RS3, the improvements decrease to 2.06 and 1.13. Especially the latter is explained by the numerous short running configuration tasks in RS3 that lead to overhead using CA* search. Results of DA* search show very similar effects, scaling well on complex rule sets but suffering from overhead on simpler ones like RS3. On RS3, the search algorithms performs better with two threads (0.91) than with four (0.90) due to the added synchronization and communication overhead. C&C's results do not show a clear pattern, it scales best on RS1 and RS3, less so on the "medium" rule set RS2. However, the addition of more threads shows larger diminishing returns. On RS1 the difference between three and four threads is only a speedup of 0.06 compared to the baseline. Furthermore, the table shows that the portfolio solver scales negatively in many cases with the number of processing units. An exception is the less complex rule set RS3. Hence, the scaling issues of the PP solver is related to the complexity of the problem instance. A in-depth discussion is given in Section 5.3.

Rule Set		CA*	DA*	C&C	PP
RS1	2 Threads	1.63	1.58	1.22	0.91
	3 Threads	2.07	1.98	1.37	0.88
	4 Threads	2.50	2.08	1.44	0.83
RS2	2 Threads	1.58	1.40	0.94	0.86
	3 Threads	2.03	1.43	1.01	0.72
	4 Threads	2.06	1.55	1.05	0.63
RS3	2 Threads	0.89	0.91	1.12	1.07
	3 Threads	1.01	0.90	1.27	1.15
	4 Threads	1.13	0.90	1.33	1.24
R3SAT/RSAT	2 Threads	1.40	1.49	1.91	1.02
	3 Threads	1.55	1.86	2.26	0.99
	4 Threads	1.72	2.06	2.46	0.97

Table 5.5: Comparison of overall speedup for parallel algorithms using varying numbers of threads.

5.2.3 Search Overhead

To compare the efficiency of the developed parallel algorithms, search overhead (SO) is measured. The authors of [JF16] define it as the ratio of expanded nodes in parallel to the number of expanded nodes during sequential search.

$$SO = \frac{|expanded\ nodes\ in\ parallel|}{|expanded\ nodes\ in\ sequential\ search|} - 1 \tag{5.1}$$

An overview of the comparison is shown in Table 5.6. Two metrics are collected for each rule set. "Overall SO" describes the search overhead of the cumulated node expansions over all configuration steps compared to the sequential solver (baseline). "Avg. SO per task" calculates the average search overhead per configuration step.
Considering overall search overhead, CA* expands between 20-45% more nodes, depending on the rule set. On more complex problem instances the overhead is on the lower end. DA* search increases the overhead to 39-104%, again strongly dependent on the rule set. C&C shows a large variance due to having a SO of 7% on RS1 but 51% on RS2. The portfolio solver shows an increase between 140% and 232%. Each solver expands its own search tree, hence the overhead becomes significantly larger. Regarding the average SO per click, the values change a bit. Both parallel A* algorithms show significantly higher average search overheads with respect to all configuration steps (e.g. CA* with 210% on RS1), while C&C achieves lower values.
The reason for this discrepancy is visualized in Figure 5.6, taking RS1 as an example. Most noticeable is the search overhead of parallel A* search on configuration tasks that require very few node expansions on the sequential solver. Parallel C&C does solve those instances in the first phase which is performed sequentially, thus search overhead is non-existent. This changes for harder configuration steps in which C&C has a larger overhead compared to the parallel A* approaches. Thus, the relative average per task overhead stays in contrast to the overhead based on all tasks. The amount of expanded nodes is related to the wall clock time, therefore the problem instances with many expansions are of higher interest for the overall performance improvement.
The tested random SAT instances show smaller differences between the parallel algorithms, except for the portfolio solver due to its search tree duplication. Overall, C&C shows a slightly higher SO on the randomized SAT instances compared to parallel A* search.

Rule Set	Search Overhead	CA*	DA*	C&C	PP
RS1	Overall	0.20	0.39	0.07	2.07
	Avg. per task	2.10	2.98	0.00	2.07
RS2	Overall	0.21	0.52	0.51	2.32
	Avg. per task	1.07	1.98	0.33	2.33
RS3	Overall	0.45	1.04	0.22	1.40
	Avg. per task	1.56	3.19	0.00	2.14
R3SAT/RSAT	Overall	0.00	0.01	0.02	2.17
	Avg. per task	0.00	0.01	0.01	2.10

Table 5.6: Comparison of search overhead (SO) for parallel algorithms using different rule sets and 4 threads.

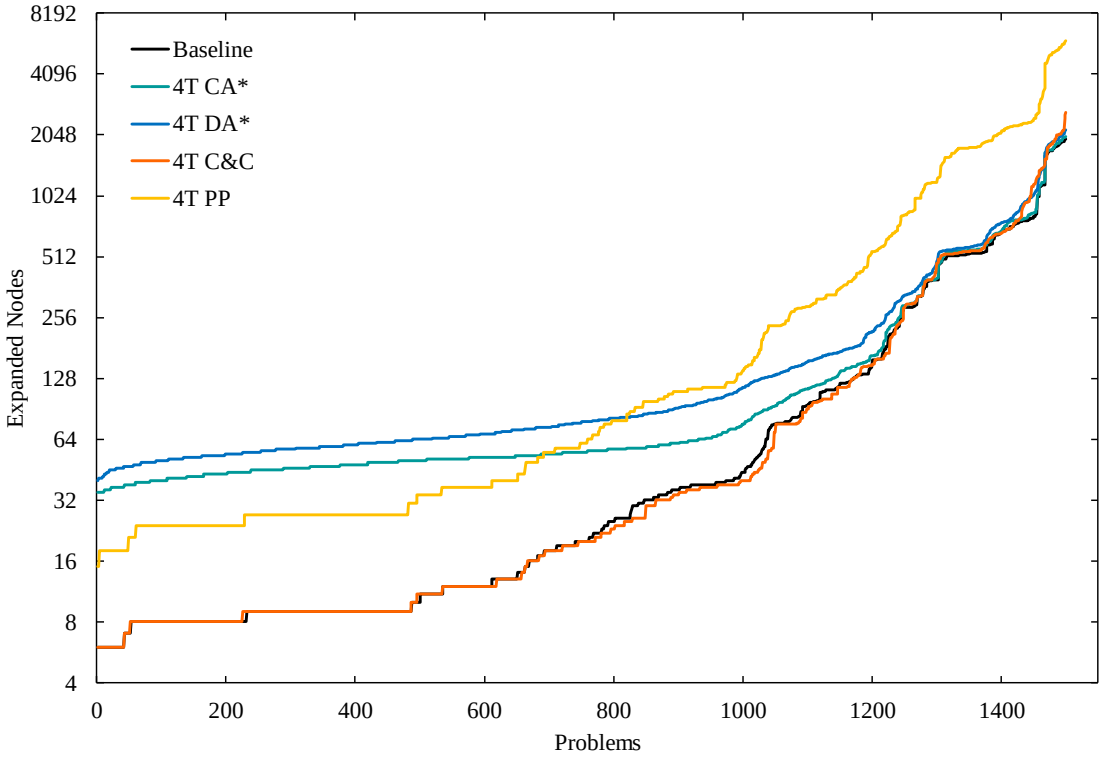

Figure 5.6: Comparison of the number of expanded nodes (logarithmic scale) for different parallel algorithms using 4 threads on RS1.

5.3 Discussion

This section provides detailed explanations for the key points of the previously presented results.

A general observation is the increasing performance improvement on more complex rule sets. A main reason for this is the share of short running configuration tasks on less complex formulae (e.g. RS3). Having instances that can be solved very fast (in less than 50 ms), it is difficult to achieve significant improvements through additional processing units. The added overhead due to initialization and synchronization outweighs the benefits of solving the instance in parallel. Additionally, the requirement of deterministic results lead to stricter search termination detection mechanisms. These are not used during sequential search, because the order of found solutions is stable. The impact is clearly shown in Figure 5.3, where a large portion of configuration steps require less than 20 ms and the parallel algorithms perform worse than the baseline. A good visualization is also shown in Figure 5.7 and 5.8 comparing search trees from a simple configuration step using sequential and parallel A* search. The figures display search overhead added by introducing parallel search.

On the other hand, Figure 5.1 displays a complex rule set, in which most configuration tasks require more than 50 ms and many of them more than 100 ms. Therefore, the majority of problem instances are solved faster by using parallel A* search. This also explains why C&C outperforms parallel A* search on RS3, because a sequential algorithm is used during the first phase, consequently adding no search overhead. On rule sets R2 and R3, the performance is mainly dependent on the search overhead for complex configuration tasks. For steps with long run times, the CA* search adds less than 5% of search overhead, DA* search approximately 10% and C&C 25% (see Figure 5.6). CA* search is most efficient at exploring nodes that belong to promising parts of the search tree. C&C and to a lesser

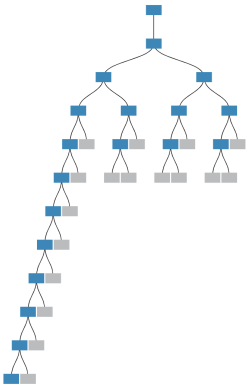

Figure 5.7: Example of a constructed search tree during a simple configuration step using the sequential solver. Gray nodes have not been expanded.

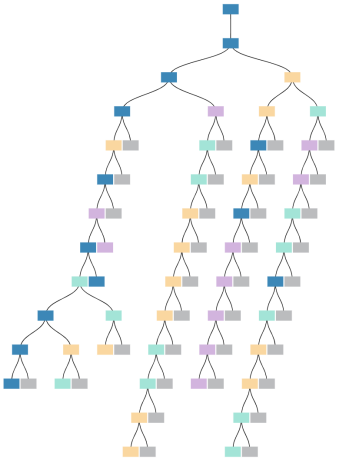

Figure 5.8: Example of a constructed search tree during a simple configuration step using parallel A* search. Each color represents a thread. Gray nodes have not been expanded.

extend DA* search expand parts of the search tree that are suboptimal, owing to reduced communication between threads. This can also be seen on Figure 5.9 and 5.10, comparing the constructed search trees of the sequential solver and parallel C&C. The figures show that C&C does span a larger tree for complex configuration tasks which severely prolongs computation time. Overall, the tests conclude that the single OPEN list of CA* search is not a severe bottle neck for the given configuration steps. The number of nodes maintained in one queue is too small in comparison to the added time required for exploring additional suboptimal nodes, like in DA* search and C&C.

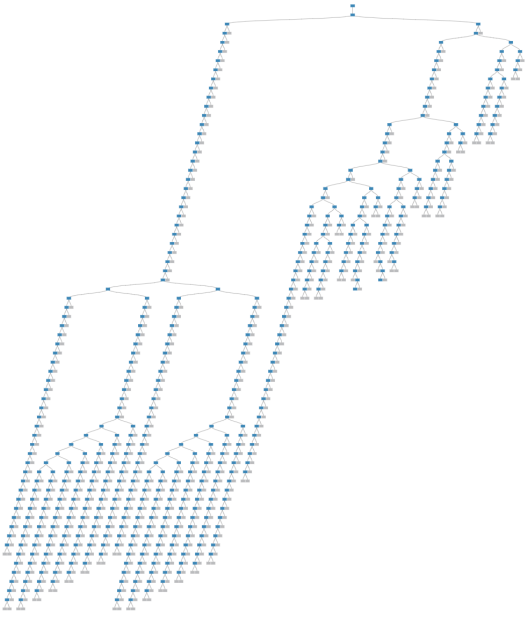

Figure 5.9: Example of a constructed search tree during a configuration step (RS1) using the sequential solver. Gray nodes have not been expanded.

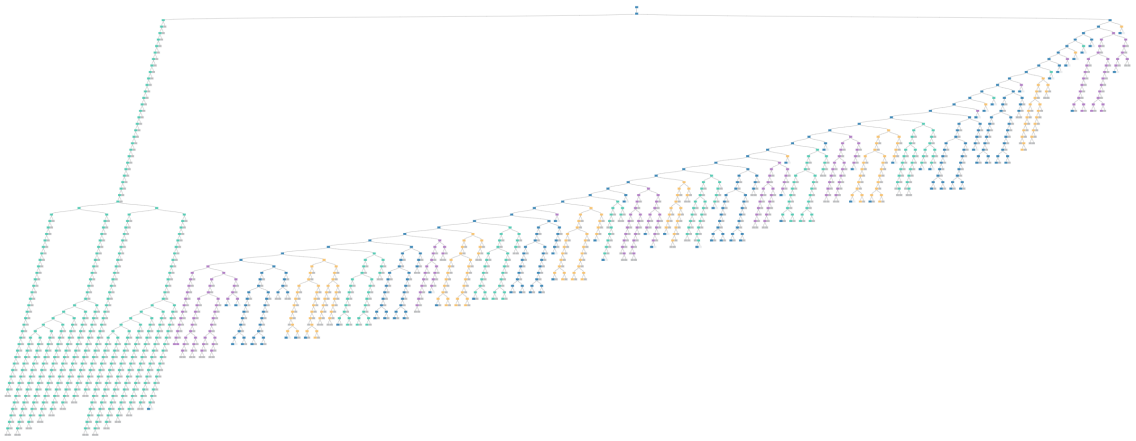

Figure 5.10: Example of a constructed search tree during a configuration step (RS1) using the parallel C&C solver. Each color represents a thread. Gray nodes have not been expanded.

In addition to the severe search overhead of parallel C&C, the implementation also suffers from its two phased approach. The first phase is always performed in sequence. Consequently, the scalability effects only benefit the second phase which is performed in parallel. Thus, even on rule sets that suit this approach, the scalability is limited. It also emphasizes the importance of finding a good cutoff heuristic that determines appropriate numbers of cubes that reduce the complexity sufficiently to start the parallel phase. Both problems lead to lower speedups through additional threads compared to parallel A* search, shown in Table 5.5.

Regarding the parallel portfolio, the solver performs worst on many industry cases. Moreover, it partly scales negatively with increasing thread pools. For each sequential base solver, the Boolean formula is duplicated. Performing k separate A* searches, with k being the number of threads, escalates the memory consumption due to the search tree construction. In most tests, the increased load on the system outweighs the benefits of having separate solver configurations that reduce the sensitivity to parameter tuning. Thus, the alternative portfolio approach PPA*, which applies different sets of parameters to parallel A* search, achieves more consistent and better results. Moreover, it does not rely on manually selecting fitting settings.

6. Conclusion and Future Work

This chapter summarizes the results of this work. Furthermore, an outlook is given for potential future work in the domain of parallel algorithms for product configuration.

6.1 Conclusion

The scope of this thesis was to conceptualize and develop parallel algorithms that can be applied to the interactive configuration process seen in modern product configuration systems. The goal was to find search algorithms that can exploit the capabilities of common multi-core processors while maintaining their completeness and determinism with respect to found solutions. To define the problem occurring in interactive configuration, the MinCostConf problem was introduced which extends the SAT and MinCostSAT problems and belongs to the class of NP-hard problems. It describes the task of finding minimal-cost solutions given a start-configuration, an user wish, and pinned attributes. Additionally, different custom cost functions were shown to model distinctive behaviors that evaluate configuration changes with respect to the start-configuration.

Subsequently, various parallel algorithms were presented that aim to solve the MinCostConf problem. As a baseline, the existing sequential A* search algorithm was introduced with a custom cost function that prefers to keep prior user selections. Three major strategies for parallel search were implemented.
Firstly, two versions of parallel A* search were conceptualized. Centralized parallel A* search extends the existing sequential baseline algorithm by sharing a single OPEN list across multiple threads. Moreover, new search termination detection mechanisms were required as well as efficient locking mechanisms to reduce synchronization overhead. The shown decentralized parallel A* search adapts this approach by assigning an OPEN list to each thread and using inter-thread communication for load balancing. Secondly, a parallel cube-and-conquer algorithm was presented. It is a two-phase approach that simplifies the problem sequentially into cubes followed by parallel processing of these subproblems. Lastly, a parallel portfolio approach was proposed that starts several base solvers in parallel, each supplied with a unique set of parameters to reduce the sensitivity to such settings. To avoid search space duplication, an alternative was shown which applies portfolio concepts to parallel A* search. The introduction of parallel algorithms for MinCostConf added nondeterminism with regard to the order of found solutions. This cannot be present in a production environment, such as a configuration system, in which users expect reproducibility. This has been addressed by designing robust search termination detection

strategies which ensure that the search is only terminated when the expected configurations are found while maintaining pruning techniques.

Finally, experiments were performed to compare the implemented algorithms using real industry cases as well as random SAT instances. Regarding the industry cases, three different rule sets with varying complexity were analyzed. Besides the completeness, ensured by using automated integration tests, additional metrics comprising speedup, break-even point, and search overhead were considered. The results of the existing sequential solver and all parallel algorithms were compared. Depending on the complexity, the parallel algorithms showed different strengths and weaknesses. Nevertheless, the centralized parallel A* search produced the most convincing results with respect to required time to solve the configuration tasks. The achieved speedup varied depending on the rule set, but for critical configuration tasks with longer response times, a consistent speedup between 2 and 3 was attained utilizing 4 worker threads. This resembles a convincing speedup. Furthermore, good scalability with the number of threads as well as a relatively small search overhead showed the algorithm's applicability to a wide variety of MinCostConf tasks. Lastly, the experiments displayed that deterministic behavior is achieved with a reasonable amount of effort.

6.2 Future Work

There are several aspects in this thesis that can be extended and further improved upon. Firstly, the evaluation was performed on a limited selection of rule sets using up to four threads. Thus, the presented algorithms can be optimized to utilize a larger number of processing units, although diminishing returns are expected. Secondly, the presented algorithms are only a subset of possible approaches. Other algorithms that have been used in the literature can be adopted and changed to fit the presented problem. For instance, to limit the memory footprint the iterative deepening A* (IDA*) algorithm can be adapted ([Kor85]). This can also improve the presented parallel portfolio approach which is limited by its resource consumption.
Furthermore, the presented MinCostConf problem defines solutions as minimal-cost configurations. In some domains with very complex configuration models, this criteria may be loosened and only good but suboptimal solutions are requested. This could be performed for example with a parallel and deterministic version of beam search.
Lastly, parallel SAT related algorithms may also be used in other areas of interactive product configuration. Besides a valid configuration, additional information can be calculated, for example the attributes that are not possible to select without changing the pinned attributes. Therefore, these attributes may be grayed out for the user. To accelerate this calculation, a parallel algorithm can be applied. Another area of interest is multi-product configuration. Given several loosely coupled products, a user wish in one product can cause changes in other dependent ones, possibly causing a chain reaction. The calculation of this impact can be performed in parallel, by analyzing the impact of the user with the help of a dependency graph and performing independent sub-configurations in parallel.

Bibliography

[AHP10] Henrik Reif Andersen, Tarik Hadzic, and David Pisinger. Interactive cost configuration over decision diagrams. *Journal of Artificial Intelligence Research*, 37:99–139, 2010.

[BA12] Mordechai Ben-Ari. *Mathematical logic for computer science*. Springer Science & Business Media, 2012.

[Bat05] Don Batory. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*, pages 7–20, 2005.

[BCRZ99] Armin Biere, Edmund Clarke, Richard Raimi, and Yunshan Zhu. Verifying safety properties of a powerpc- microprocessor using symbolic model checking without bdds. In *International Conference on Computer Aided Verification*, pages 60–71, 1999.

[BF98] Brian Borchers and Judith Furman. A two-phase exact algorithm for max-sat and weighted max-sat problems. *Journal of Combinatorial Optimization*, 2(4):299–306, 1998.

[BHJ17] Tomás Balyo, Marijn J. H. Heule, and Matti Jarvisalo. Sat competition 2016: Recent developments. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[Bie13] Armin Biere. Lingeling, plingeling and treengeling entering the sat competition 2013. *Proceedings of SAT competition*, 2013:1, 2013.

[BKB+13] Csaba Biro, Gergely Kovasznai, Armin Biere, Gábor Kusper, and Gábor Geda. Cube-and-conquer approach for sat solving on grids. In *Annales Mathematicae et Informaticae*, pages 9–21, 2013.

[BLRZ10] Ethan Burns, Sofia Lemons, Wheeler Ruml, and Rong Zhou. Best-first heuristic search for multicore machines. *Journal of Artificial Intelligence Research*, 39:689–743, 2010.

[Boo54] George Boole. *An investigation of the laws of thought: On which are founded the mathematical theories of logic and probabilities*. Dover Publications, 1854.

[BSS15] Tomáš Balyo, Peter Sanders, and Carsten Sinz. Hordesat: A massively parallel portfolio sat solver. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 156–172, 2015.

[CAS20] CAS Software AG. Cas configurator merlin, 2020.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[CW03] Wahid Chrabakh and Richard Wolski. Gridsat: A chaff-based distributed sat solver for the grid. In *SC'03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 37, 2003.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[EHMN95] Matthew Evett, James Hendler, Ambuj Mahanti, and Dana Nau. Pra*: Massively parallel heuristic search. *Journal of Parallel and Distributed Computing*, 25(2):133–143, 1995.

[EMW97] Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld. Automatic sat-compilation of planning problems. In *IJCAI*, volume 97, pages 1169–1176, 1997.

[FDH05] Yulik Feldman, Nachum Dershowitz, and Ziyad Hanna. Parallel multithreaded satisfiability solver: Design and implementation. *Electronic Notes in Theoretical Computer Science*, 128(3):75–90, 2005.

[FFH11] Andreas Falkner, Alexander Felfernig, and Albert Haag. Recommendation technologies for configurable products. *Ai Magazine*, 32(3):99–108, 2011.

[FHBT14] Alexander Felfernig, Lothar Hotz, Claire Bagley, and Juha Tiihonen. *Knowledge-based configuration: From research to business cases*. Newnes, 2014.

[FLW01] Eugene C. Freuder, Chavalit Likitvivatanavong, and Richard J. Wallace. Explanation and implication for configuration problems. In *IJCAI 2001 workshop on configuration*, pages 31–37, 2001.

[FM06] Zhaohui Fu and Sharad Malik. Solving the minimum-cost satisfiability problem using sat based branch-and-bound search. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 852–859, 2006.

[FW94] Boi Faltings and Rainer Weigel. Constraint-based knowledge representation for configuration systems. In *Technical Report TR-94/59*. Citeseer, 1994.

[GHM+12] Peter Großmann, Steffen Hölldobler, Norbert Manthey, Karl Nachtigall, Jens Opitz, and Peter Steinke. Solving periodic event scheduling problems with sat. In *International conference on industrial, engineering and other applications of applied intelligent systems*, pages 166–175, 2012.

[GJLS14] Long Guo, Said Jabbour, Jerry Lonlac, and Lakhdar Sais. Diversification by clauses deletion strategies in portfolio parallel sat solving. In *2014 IEEE 26th International Conference on Tools with Artificial Intelligence*, pages 701–708, 2014.

[GLP75] Jim N. Gray, Raymond A. Lorie, and Gianfranco R. Putzolu. Granularity of locks in a shared data base. In *Proceedings of the 1st International Conference on Very Large Data Bases*, pages 428–451, 1975.

[HJN10] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Partitioning sat instances for distributed solving. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 372–386, 2010.

[HJPS11] Youssef Hamadi, Said Jabbour, Cédric Piette, and Lakhdar Sais. Deterministic parallel dpll. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):127–132, 2011.

[HJS10] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat: A parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, 2010.

[HJS11] Youssef Hamadi, Said Jabbour, and Jabbour Sais. Control-based clause sharing in parallel sat solving. In *Autonomous Search*, pages 245–267. Springer, 2011.

[HKWB11] Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding cdcl sat solvers by lookaheads. In *Haifa Verification Conference*, pages 50–65, 2011.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[HSJ+04] Tarik Hadzic, Sathiamoorthy Subbarayan, Rune M. Jensen, Henrik R. Andersen, Jesper Møller, and Henrik Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. *small*, 10(1):3, 2004.

[Hv09] Marijn Heule and Hans van Maaren. Look-ahead based sat solvers. *Handbook of satisfiability*, 185:155–184, 2009.

[IS86] KEKIB IRANI and YI-FON SHIH. Parallel a* and ao* algorithms: An optimality criterion and performance evaluation. In *1986 International Conference on Parallel Processing, University Park, PA*, pages 274–277, 1986.

[Jan08] Mikolas Janota. Do sat solvers make good configurators? In *SPLC (2)*, pages 191–195, 2008.

[Jan10] Mikoláš Janota. *SAT solving in interactive configuration*. PhD thesis, Citeseer, 2010.

[JBGMS10] Mikoláš Janota, Goetz Botterweck, Radu Grigore, and Joao Marques-Silva. How to complete an interactive configuration process? In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 528–539, 2010.

[JF16] Yuu Jinnai and Alex Fukunaga. Abstract zobrist hashing: An efficient work distribution method for parallel best-first search. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[JLU05] Bernard Jurkowiak, Chu Min Li, and Gil Utard. A parallelization scheme based on work stealing for a class of sat solvers. *Journal of Automated Reasoning*, 34(1):73–101, 2005.

[Jon07] Simon Peyton Jones. Beautiful concurrency. *Beautiful Code: Leading Programmers Explain How They Think*, pages 385–406, 2007.

[JW90] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence*, 1(1-4):167–187, 1990.

[KKW11] Yoshikazu Kobayashi, Akihiro Kishimoto, and Osamu Watanabe. Evaluations of hash distributed a* in optimal sequence alignment. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[Kor85] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[KRR88] Vipin Kumar, K. Ramesh, and V. Nageshwara Rao. Parallel best-first search of state-space graphs: A summary of results. In *AAAI*, volume 88, pages 122–127, 1988.

[L+04] Xiao Yu Li et al. Optimization algorithms for the minimum-cost satisability problem. 2004.

[LGZ+15] Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. Understanding vsids branching heuristics in conflict-driven clause-learning sat solvers. In *Haifa Verification Conference*, pages 225–241, 2015.

[LM09] Chu Min Li and Felip Manya. Maxsat, hard and soft constraints. *Handbook of satisfiability*, 185:613–631, 2009.

[LSB07] Matthew Lewis, Tobias Schubert, and Bernd Becker. Multithreaded sat solving. In *Asia and South Pacific Design Automation Conference, 2007*, pages 926–931, Piscataway, NJ, 2007. IEEE Operations Center.

[MML12] Ruben Martins, Vasco Manquinho, and Inês Lynce. Clause sharing in deterministic parallel maximum satisfiability. In *RCRA International Workshop on Experimental Evaluation of Algorithms for solving problems with combinatorial explosion*, 2012.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.

[MSLM09] Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In *Handbook of satisfiability*, pages 131–153. IOS Press, 2009.

[MSS99] Joao P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

[PLK14] Mike Phillips, Maxim Likhachev, and Sven Koenig. Pa* se: Parallel a* for slow expansions. In *Twenty-Fourth International Conference on Automated Planning and Scheduling*, 2014.

[Pos21] Emil L. Post. Introduction to a general theory of elementary propositions. *American Journal of Mathematics*, 43(3):163, 1921.

[Rou12] Olivier Roussel. Description of ppfolio (2011). *Proc. SAT Challenge*, page 46, 2012.

[SBK01] Carsten Sinz, Wolfgang Blochinger, and Wolfgang Küchlin. Pasat—parallel sat-checking with lemma exchange: Implementation and applications. *Electronic Notes in Discrete Mathematics*, 9:205–216, 2001.

[SKI14] Tomohiro Sonobe, Shuya Kondoh, and Mary Inaba. Community branching for parallel portfolio sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 188–196, 2014.

[SKK01] Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. Detection of inconsistencies in complex product configuration data using extended propositional sat-checking. In *FLAIRS conference*, pages 645–649, 2001.

[SKK03] Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. Formal methods for the validation of automotive product configuration data. *Ai Edam*, 17(1):75–97, 2003.

[SLB10] Tobias Schubert, Matthew Lewis, and Bernd Becker. Pamiraxt: Parallel sat solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):203–222, 2010.

[SS96] J. MarquesP Silva and Karem A. Sakallah. Conflict analysis in search algorithms for satisfiability. In *Proceedings Eighth IEEE International Conference on Tools with Artificial Intelligence*, pages 467–469, 1996.

[SW98] Daniel Sabin and Rainer Weigel. Product configuration frameworks-a survey. *IEEE Intelligent Systems and their applications*, 13(4):42–49, 1998.

[TJM96] Mitchell M. Tseng, Jianxin Jiao, and M. Eugene Merchant. Design for mass customization. *CIRP annals*, 45(1):153–156, 1996.

[Wal16] M. Mitchell Waldrop. The chips are down for moore's law. *Nature News*, 530(7589):144, 2016.

[XHLB10] Lin Xu, Holger Hoos, and Kevin Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[ZBH96] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. Psato: A distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4-6):543–560, 1996.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No. 01CH37281)*, pages 279–285, 2001.