ÉCOLE NATIONALE
SUPÉRIEURE
D'INFORMATIQUE
ET DE MATHÉMATIQUES
APPLIQUÉES GRENOBLE

KARLSRUHE INSTITUT
FÜR TECHNOLOGIE

MASTER THESIS

Hierarchical Task Network Planning Using SAT Techniques

Author:
Dominik Pascal SCHREIBER

Supervisors:
Dr. Damien PELLIER,
Dr. Humbert FIORINO

For the degree of
Master of Science in Informatics at Grenoble
Master Informatique, Université Grenoble Alpes
Specialization Graphics, Vision and Robotics
In a double degree with
Informatik M.Sc., Karlsruhe Institut für Technologie

Performed at
MAGMA, Laboratoire d'Informatique de Grenoble

Defended before a jury composed of
James Crowley, President
Damien Pellier
Humbert Fiorino
Marc Métivier
Dominique Vaufreydaz

June 25, 2018

Declaration of Authorship

I, Dominik Pascal SCHREIBER, declare that this thesis titled "Hierarchical Task Network Planning Using SAT Techniques" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

ENSIMAG, KIT

Abstract

Hierarchical Task Network Planning Using SAT Techniques

by Dominik Pascal SCHREIBER

Automated planning is useful for a wide range of general decision-making processes in the area of Artificial Intelligence. The solving approach of encoding a planning problem into propositional logic and finding a solution with a SAT solver is a well-established method. Likewise, a planning model called Hierarchical Task Networks (HTN) which enhances planning problems with expert knowledge can help to find structured plans more easily and efficiently. This work focuses on combining these techniques by using SAT solving techniques to resolve HTN planning problems.

Initially, a previous approach to encode HTN problems in SAT is analyzed, and various shortcomings are identified from today's perspective. Then, three original encodings are proposed: *Grammar-Constrained Tasks* (GCT) which is inspired by one of the previous encodings and is the first to feature modern HTN domains and recursive task relationships; *Stack Machine Simulation* (SMS) which is designed for incremental SAT solving and works reliably on all special cases; and *Tree-like Reduction Exploration* (T-REX) which leads to a particularly efficient solving process due to its short amount of needed iterations and various introduced optimizations. All encodings are implemented to exploit existing HTN grounding routines, and the T-REX approach features a novel abstract formula notation and an efficient Interpreter application tailored to the encoding. In addition, an Anytime plan length optimization within T-REX is proposed.

Experiments show that SMS dominates GCT and that T-REX dominates SMS. The proposed T-REX planning framework outperforms a state-of-the-art classical SAT planner on various domains. Regarding run times, T-REX cannot compete with state-of-the-art HTN planners yet, but is still an appealing planning approach due to its robustness, its plan length optimization, and its proving abilities.

By the design, implementation and evaluation of T-REX, the work at hand demonstrates that HTN planning via SAT solving is a viable option and worthy of the attention of future research.

Résumé

Planification *Hierarchical Task Network* à l'aide des techniques SAT

La planification automatisée est utile pour un large éventail de processus décisionnels généraux dans le domaine de l'intelligence artificielle. L'approche de résolution consistant à coder un problème de planification en logique propositionnelle et à trouver une solution à l'aide d'un solveur SAT est une méthode bien établie. Parallèlement, un modèle de planification appelé Hierarchical Task Networks (HTN) a été developpé pour améliorer la résolution des problèmes de planification grâce à l'ajout de connaissances particulières. Le travail de ce memoire se concentre sur la combinaison de ces techniques en utilisant les techniques de résolution SAT pour résoudre les problèmes de planification HTN.

Dans un premier temps, nous analysons une approche précédente des encodages des problèmes HTN dans SAT. Diverses lacunes sont identifiées du point de vue d'aujourd'hui. Ensuite, trois encodages originaux sont proposés: *Grammar-Constrained Tasks* (GCT), qui s'inspire de l'un des encodages précédents et est le premier à proposer des domaines HTN modernes et des relations de tâches récursives; *Stack Machine Simulation* (SMS), qui est conçu pour la résolution SAT incrémentale et fonctionne de manière fiable; et *Tree-like Reduction Exploration* (T-REX), qui propose un processus de résolution particulièrement efficace en raison du nombre réduit d'itérations nécessaires et des diverses optimisations introduites. Tous les encodages sont implémentés pour exploiter des problèmes HTN complètement instantiés, et l'approche T-REX comporte une nouvelle formule de notation abstraite et une application d'interprétation efficace adaptée à l'encodage. De plus, une optimisation *anytime* de la longeur du plan dans T-REX est proposée.

Les expériences montrent que SMS domine GCT et que T-REX domine SMS. Le cadre de planification T-REX proposé surpasse un planificateur SAT classique de pointe sur divers domaines. En ce qui concerne les durées d'exécution, T-REX ne peut pas encore rivaliser avec les planificateurs HTN les plus récents, mais demeure une approche de planification attrayante en raison de sa robustesse, de l'optimisation de la longueur du plan et de ses capacités à faire ses preuves.

Par la conception, la mise en œuvre et l'évaluation de T-REX, le travail en cours démontre que la planification HTN via SAT solving est une option viable et digne de l'attention de recherche future.

Kurzfassung

Hierarchical Task Network Planung unter Verwendung von SAT-Techniken

Automatisiertes Planen hat sich für eine Vielzahl von allgemeinen Entscheidungsprozessen im Bereich der Künstlichen Intelligenz bewährt. Eine etablierte Methode ist dabei, ein Planungsproblem in Aussagenlogik zu kodieren und eine Lösung mit einem SAT-Solver zu finden. Ebenso kann ein Planungsmodell namens Hierarchical Task Networks (HTN), welches Planungsprobleme mit Expertenwissen erweitert, dabei helfen, einfacher und effizienter strukturierte Pläne zu finden. Die vorliegende Arbeit konzentriert sich auf die Kombination dieser Techniken, namentlich auf den Einsatz von SAT-Techniken zur Lösung von HTN-Planungsproblemen.

Zunächst wird ein ehemaliger Ansatz zur Kodierung von HTN-Problemen in SAT analysiert, und verschiedene Mängel aus heutiger Sicht werden identifiziert. Daraufhin werden drei neuartige Kodierungen vorgeschlagen: *Grammar-Constrained Tasks* (GCT), die von einer der ehemaligen Kodierungen inspiriert ist und erstmalig moderne HTN-Domänen und rekursive Subtask-Beziehungen unterstützt; *Stack Machine Simulation* (SMS), die für inkrementelles SAT-Solving ausgelegt ist und in allen Sonderfällen zuverlässig funktioniert; und *Tree-like Reduction Exploration* (T-REX), die aufgrund der geringen Anzahl nötiger Iterationen und diversen Optimierungen zu einem besonders effizienten Lösungsprozess führt. Alle Kodierungen wurden unter Verwendung bestehender HTN-Grounding-Routinen implementiert, und der T-REX-Ansatz verwendet eine neuartige abstrakte Formel-Notation und eine auf die Kodierung zugeschnittene Interpreter-Anwendung. Zusätzlich wird eine Anytime-Planlängenoptimierung innerhalb von T-REX präsentiert.

Experimente zeigen, dass GCT von SMS dominiert wird und SMS von T-REX dominiert wird. Das entwickelte T-REX-Framework unterbietet die Laufzeiten klassischer SAT-Planer auf verschiedenen Domänen. T-REX kann noch nicht die Laufzeiten eines modernen HTN-Planers erreichen, ist aber aufgrund seiner Robustheit, seiner Planlängenoptimierung und seiner Beweis-Fähigkeiten dennoch ein attraktiver Planungsansatz.

Durch die Konzeption, Implementierung und Evaluation von T-REX zeigt die vorliegende Arbeit, dass HTN-Planung per SAT-Solving eine praktikable Option ist und die Aufmerksamkeit zukünftiger Forschung verdient.

Acknowledgements

I would like to thank my project advisors, namely Damien Pellier and Humbert Fiorino, for their continuous and encouraging support.
I would also like to thank Tomáš Balyo for externally supervising the project on the part of the KIT and specifically for providing helpful tips, IPASIR bindings to various SAT solvers and the encoding notation DimSpec, all of which I thankfully used and built upon.
Additionally, I would like to thank Abdeldjalil Ramoul for very helpful and insightful discussions and for his support regarding the preprocessing code.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the following, an introduction to the topic at hand is provided, and a general overview of the structure and content of this report is given.

1.1 Motivation and Background

The topic of automated planning has been of great research interest for a long time, and by today's technological advancements it has become more relevant than ever. The objective of a machine being able to autonomously find a viable sequence of doable tasks in order to fulfill some given goal is very generic and universal. For this reason, a broad range of application domains are interested in automated planning, such as spacecraft control (Fukunaga et al., 1997), autonomous robotics (Raja and Pugazhenthi, 2012), logistics (García et al., 2013), and many more areas making use of artificial intelligence and general decision-making processes.

For instance, this great diversity of applications can be seen in the various benchmark domains which are commonly used in automated planning. One set of problems describes a set of rovers which have to gather and report various soil, rock, and image data which is spread across a virtual map of waypoints. Another problem set models a robotic barman which needs to mix and serve a set of cocktails to its customers. Other benckmark domains include stacking blocks in some correct manner, finding flight routes between a set of cities, or planning the logistics of a set of transporters.

As automated planning is a challenging task which often becomes limited by the available computational resources, research focuses on rendering the planning process as efficient as possible. For this reason, a number of techniques, both for modeling planning problems and for solving the resulting problem statements, have been successfully introduced in the past decades.

An important planning model which may greatly improve the planning process is called *Hierarchical Task Network* (HTN) planning. The central idea in HTN planning is to make use of expert knowledge about the problem domain at hand, provided once by human interactors. This expert knowledge, given in the form of a hierarchy of possible *task reductions*, is then used by an automated planner to find a concrete plan for an abstract objective. As HTN planning domains provide much more structure about the problem than any classical formulation which only contains primitive actions, the space of possible solutions to search is usually significantly smaller, rendering the planning process faster and less resource-demanding.

On the side of solving conventionally modeled planning problems, logical methods have been introduced to automated planning with great success. The paradigm of *SAT solving* is based on the principle of rephrasing a problem in propositional logic, and then using well-developed methods to solve this logical formula. In practice, this is performed by encoding a planning problem into a set of boolean variables constrained by only using the fundamental logical operators $\wedge, \vee, \neg$ in some appropriate manner. As soon as this is achieved, one of the highly efficient SAT solvers which have emerged during the last decades (e.g. Eén and Sörensson, 2003; Audemard and Simon, 2009; Biere, 2013) can be used to find a satisfying assignment for the encoded formula. A decoding process can then transform the solution back to the original problem, yielding a valid plan.

SAT solving is an appealing approach to automated planning for a number of reasons. All of the "heavy work" of the planning process is performed by an external algorithm which is implemented efficiently, and instead of having the need to develop domain-specific heuristics, a SAT solver uses its own, universal heuristics based on various properties of the formula. Consequently, any improvement done in the field of efficient SAT solving may also directly improve a SAT-based planning process.

Interestingly, while both the HTN modeling of planning problems and the SAT solving approach have been widely used, few scientific research has been done on combining these techniques, namely on solving HTN planning problems via SAT solving. The last published effort to find a propositional logic encoding for HTN planning domains (Mali and Kambhampati, 1998) features classical SAT encoding techniques which have since then become outdated in the domain of classical planning, as more efficient alternatives have gained popularity. However, by the introduction of efficient grounding procedures of HTN domains (Ramoul et al., 2017), a viable foundation has been created for revisiting the topic of HTN-to-SAT encodings.

To exploit the merits of SAT-based planning also for HTN domains, the work at hand focuses on finding efficient encodings of HTN planning problems in propositional logic, with their practical applicability in mind.

1.2 Methodology and Results

In the following, the general methodology of the work at hand is explained, and the central results are provided in a compact manner.

As a point of departure, the first and only published HTN-to-SAT encodings (Mali and Kambhampati, 1998) have been analyzed. From this point, multiple iterations of the following scientific procedure have been performed: In each iteration, the central shortcomings of the previous encoding have been identified, and some promising idea mending these shortcomings has been applied to design a new encoding. This encoding has then been implemented and evaluated in an explorative manner in order to quickly assess the encoding's potential. When an encoding has been fully engineered, theoretically analyzed, and proven to work correctly and efficiently, it concluded an iteration.

In total, three such cycles have been gone through, resulting in the three respective contributions presented in Chapters 4, 5, and 6. Each of these contributions dominates its predecessor. The encoding Grammar-Constrained Clauses (GCT) is the first to introduce modern HTN preprocessing to SAT-based planning and the first encoding to feature recursive task relationships (Chapter 4). The next encoding Stack Machine Simulation (SMS) is specifically designed for incremental SAT solving, leading to a more efficient solving process and a more compact encoding representation (Chapter 5). The final contribution, named Tree-like Reduction Exploration (T-REX), is a carefully engineered enhancement of the idea of SMS and an overall satisfying solution to the stated problem (Chapter 6).

After this design stage, thorough and reliable experimental evaluations of the developed approaches have been conducted using problem benchmarks from the International Planning Competition (IPC)[1]. The encodings are experimentally compared to one another, confirming the superiority of T-REX over its predecessors. T-REX is then evaluated in-depth concerning its choice of encoding parameters by using an automatic parameter tuning framework. The performances of T-REX are also compared to state-of-the-art classical SAT planning in order to validate the usage of the proposed HTN-SAT technique over just making use of classical SAT planning.

The evaluations yielded that the T-REX approach may well compete with state-of-the-art classical SAT planning, but it does not achieve run times on par with state-of-the-art HTN planning yet. Still, T-REX is an appealing planning approach because of its robustness, its proving abilities and an integrated plan optimization technique. Overall, the results are highly encouraging for further research into this direction.

1.3 Document summary

In the following, a compact summary of each included chapter is provided.

Chapter 1 introduces the general topic and provides information on the structure of the work and the report.

In Chapter 2, the thematical background to the work is presented and important related work is discussed. In particular, a brief overview of classical planning and HTN planning as well as on the foundations of modern SAT solving and particularly incremental SAT solving is provided.

Chapter 3 states the formal model and problem statement which is used throughout the report. Concise formal definitions are provided in order to clearly refer to the according structures of Hierarchical Task Networks.

In Chapter 4, a previous contribution by Mali and Kambhampati, 1998 is discussed. The encodings proposed there have a polynomial complexity of variables and clauses, and they do not support recursive reductions, which proves to be a significant restriction. A new encoding named Grammar-Constrained Clauses (GCT) based on the most applicable of the previous encodings is then proposed, mending the shortcomings of the previous encodings and thus enabling the use of efficient existing HTN grounding procedures (Ramoul et al., 2017). The GCT encoding has a lower complexity than the one stated in the original publication, but still is an asymptotically quite complex encoding and neglects some subtle special cases. The conclusion of this first contribution is that some significantly different and more lightweight way of encoding the HTN constraints must be found in order to render HTN planning via SAT techniques viable.

Building on the previous chapter, an incremental encoding approach is proposed, analyzed and discussed in Chapter 5. The Stack Machine Simulation (SMS) encoding is based on the idea of maintaining the entire "active hierarchy" at each computational step within a stack encoded in propositional logic. SMS uses the DimSpec notation (Gocht and Balyo, 2017) for efficient incremental SAT solving, is more lightweight than the GCT encoding, and works reliably on all considered special cases. However, its encoding framework renders the approach inflexible towards optimizations, and its completeness is limited by a parameter (the maximal stack

[1]See `http://icaps-conference.org/index.php/Main/Competitions`.

size which is being encoded). Furthermore, the computation induced by the SMS encoding takes a high amount of computational steps, in particular more than any classical planning encoding. Overall, the SMS encoding is a significant improvement over GCT in every aspect, but in total still leaves to be desired regarding its efficiency and practical usability.

Chapter 6 presents the most efficient and viable option beneath the proposed encodings: The Tree-like Reduction Exploration (T-REX) encoding can be incrementally extended along the *depth* of the considered hierarchy instead of the length of the final plan, and is able to reduce *all* occurring task reductions in a simultaneous manner at each computational step. These properties lead to a fast solving process with very few incremental iterations needed. T-REX is realized by an abstract encoding notation tailored to the problem and interpreted by a separate application which can instantiate the required clauses just as needed. Additionally, the encoding has been optimized by a number of techniques which effectively reduce the amount of required variables and clauses, such as a sparse encoding of variables representing facts and decompositions. T-REX also features an optional post-processing stage wherein an initially found plan may be efficiently optimized regarding its length. A logical structure counting the plan length is introduced which enables restricting certain plan lengths by iteratively setting assumptions and executing the SAT solver.

Some relevant details about the implementation of the proposed methods are given in Chapter 7, in particular about the two implemented applications HTN-SAT (to encode HTN planning problems in SAT and perform planning using a SAT solver) and the T-REX Interpreter (to interpret a T-REX encoding file and solve it by directly communicating with a SAT solver). The implementation of the conducted experiments is described as well.

In Chapter 8, thorough evaluations of the developed methods are presented. First, the T-REX approach is tuned using the popular automatic tuning framework ParamILS (Hutter et al., 2009), leading to a certain encoding variant which generally performs well on the considered problems. Afterwards, all proposed encoding approaches are experimentally evaluated and compared to one another, and the best found configuration of T-REX is compared to the state-of-the-art classical SAT planner Madagascar (Rintanen, 2014) which does not make use of any hierarchical information.

It is seen that SMS clearly outperforms GCT, and that T-REX significantly dominates SMS. Furthermore, T-REX outperforms Madagascar on some of the problem domains, validating the use of T-REX over classical SAT planning approaches. The results are discussed while taking into account the most relevant properties of individual domains. As a consequence, some domain properties which are well-suited and some which are ill-suited for the T-REX approach are identified. Comparing the T-REX results to the state-of-the-art conventional HTN planner GTOHP (Ramoul et al., 2017), it is clear that GTOHP outperforms T-REX regarding run times, but T-REX can still be a viable option due to its plan optimization, its robustness towards domains missing preconditions, and its proving abilities. A qualitative demonstration and exemplary discussion of the plan length optimization of T-REX concludes the evaluations which overall successfully demonstrate the viability of the developed methods.

Chapter 9 concludes the report by providing a closing summary and an outlook. Possible future work mentioned there includes various extensions of the problem model, specialized preprocessing procedures for T-REX, and further investigations of alternative SAT solving strategies which may be well-suited to solve the developed encodings.

Chapter 2

Background and Related Work

In the following, some foundations of automated planning and SAT solving in the context of this work are provided, and related work is discussed.

2.1 Automated Planning

Automated planning is a research branch of Artificial Intelligence. It aims to enable the autonomous reasoning over some model of the world in order to pursue some specific goal (Ghallab, Nau, and Traverso, 2004). In its most classical form, which has been in particular coined by Fikes and Nilsson, 1971 as STRIPS planning, the world and its changes are modeled as a sequence of *states* which may be manipulated by the execution of atomic *actions*. A sequence of actions which successively transforms the world from some particular *initial state* to some *goal state* is called a *plan*. Finding such a plan from a specification of the world states and the possible actions is the fundamental problem of automated planning.

Domain-independent planning is generally a very complex problem, as the default STRIPS model is already PSPACE-complete (Bylander, 1994). Conventional methods for classical automated planning are based on exploring a graph structure which represents the set of possible states and the actions which correspond to transitions between states. The GRAPHPLAN approach by (Blum and Furst, 1997) introduced an important technique with the new model of a *planning graph*, where both facts and actions are nodes in alternating layers of the graph. Another technique has been successfully presented in the FF (Fast-Forward) planning system (Hoffmann and Nebel, 2001), where heuristics are used to intelligently traverse the state-space, in combination with a relaxed GRAPHPLAN technique. The approach of solving planning problems with SAT techniques is further discussed in Chapter 2.2.2.

2.1.1 HTN Planning

A common subtype of automated planning is called Hierarchical Task Network (HTN) planning. It is based on the idea to extend classical planning domains such that certain domain-specific expert knowledge is provided to the planner in order to facilitate the planning process.

Today's modeling of HTN problems features a set of *tasks* which are successively decomposed into *actions* by well-defined *reduction* rules. For instance, consider some simple example of a robot that should plan how to move a ball from A to B. Instead of just providing the functioning of primitive actions like `grasp`, `move`, or `drop` and letting a planner search for "arbitrary" action sequences leading to a final goal, the HTN model may add information of a task `transport_ball` which *contains* a sequence of the mentioned primitive actions. A planner can use this information to only search for such valid task reductions, rendering the search space much smaller. A complete model of HTN planning including further examples is provided in Chapter 3.

The idea to enrich a planning problem with additional knowledge of how certain tasks are realized ranges back to (Sacerdoti, 1975), where a structure called *procedural net* has been proposed. Since then, automated planning using Hierarchical Task Networks advanced with a range of proposed planning systems such as Nonline (Tate, 1976), O-Plan (Currie and Tate, 1991) or SIPE (Wilkins, 1984), with UMCP (Erol, Hendler, and Nau, 1994) being the first proposed solving procedure which has been proven to be sound and complete. All of these algorithms have the common point of operating in a state-less manner; they do not maintain a set of facts at each step of the planning, but instead they search the general space of possible plans in a unified manner.

The planner SHOP (Nau et al., 1999) and its enhancement SHOP2 (Nau et al., 2003) are among the most popular HTN planners (Nau et al., 2005) and have been used as a foundation for the most recent HTN planner GTOHP (Ramoul et al., 2017). In contrast to previous approaches, SHOP and its enhancements are state-based and primitive actions are visited exactly in the order which they will have in the final plan. This leads to an easy identification of the applicability of decompositions at some given point of the planning process.

Today, HTN planning is used in practice in various application domains such as web service compositions (Sirin et al., 2004), robot planning (Weser, Off, and Zhang, 2010), and drone coordination (Bevacqua et al., 2015).

2.2 SAT Solving

The propositional logic satisfiability problem, SAT in short, is one of the most fundamental NP-complete problems in computer science (Cook, 1971). The problem statement is as follows: Given a propositional logic formula F, decide whether there is a satisfying assignment $\mathcal{A}$ of boolean values to each of the variables occurring in F such that $\mathcal{A}$ is a *model* for F, or in short $\mathcal{A} \models F$. In other words, the problem is to decide whether a given formula is *satisfiable* and also to provide a satisfying variable assignment if it exists.

Despite its theoretical origin, the resolution of SAT problems is of great practical interest. This is due to the fact that a rich variety of problems can be modeled (or *encoded*) in propositional logic and that an assignment for the created formula directly leads to a solution for the problem at hand by decoding the variables back to their original meaning.

For instance, some applications of SAT solving include FPGA routing (Wood and Rutenbar, 1998), software verification (Prasad, Biere, and Gupta, 2005), scheduling problems (Großmann et al., 2012), and automated planning (Kautz and Selman, 1992). Because of the great interest in resolving SAT problems, very efficient SAT solvers – applications which solve the SAT problem for a formula given as an input – have been developed (Moskewicz et al., 2001; Eén and Sörensson, 2003; Audemard and Simon, 2009; Biere, 2013).

Modern SAT solving mainly focuses on the method of *Conflict-Driven Clause Learning* (CDCL) (Silva and Sakallah, 1997, Bayardo Jr and Schrag, 1997) which in itself is a sophisticated enhancement of the previous DPLL method (Davis, Logemann, and Loveland, 1962). The core principle is to intelligently choose a variable to assign some value to, and to propagate this value through the clauses until either a conflict or a non-conflicting assignment to all variables is found. Learned conflicts are remembered for the further examination of the clause.

In practice, most SAT solvers expect a formula in *Conjunctive Normal Form* (CNF), i.e. a formula F of the following form:

$$F = \bigwedge_{i=1}^{c} C_i = \bigwedge_{i=1}^{c} \bigvee_{j=1}^{l_i} L_{ij}$$

Such a formula contains c *clauses*. Each clause C_i contains some number l_i of *literals*. Each literal L_{ij} is either some atomic variable A or its negated counterpart $\neg A$. Intuitively, an assignment satisfies such a formula F if and only if every clause contains at least one literal which is `true` under the assignment.

The DIMACS file specification ("Satisfiability: Suggested Format" 1993) provides a standardized way to denote a SAT encoding in such a way. Each variable is represented by a positive integer v; positive literals are represented by this number v while negated literals are represented by $-v$. Each line of a DIMACS file specifies one clause in the form of a space-separated sequence of such literals. Using this notation, applications can easily output the created formula as a file and then call a SAT solver on this file as an input.

2.2.1 Incremental SAT Solving

A common subtype of SAT solving is called *incremental SAT solving*. This paradigm can be exploited for various kinds of problems which are incremental in nature, such as general SAT-based encodings of planning problems.

An application solving an incremental problem with conventional SAT solving repeatedly re-encodes the entire problem and uses a new SAT solver process at each iteration for isolated solving attempts. In contrast, incremental SAT solving enables the application to re-use one single SAT solver instance over the entire computation and to only *extend* the previously computed encoding with new clauses instead of completely replacing it. In addition, some particular parts of the formula are considered only for a single solving attempt and dropped afterwards. These clauses are *unit* (i.e. they are comprised of single literals) and are called *assumptions*.

A central advantage of incremental SAT solving is that solvers can remember properties (in particular, conflicts) of the formula learned from past solving attempts and thus find a result more quickly (Nabeshima et al., 2006). In addition, the encoding of the problem can become more compact, specifying only the "blueprints of clauses" that will need to be added.

2.2.2 Planning via SAT

As automated planning is a problem of logical nature, SAT solving has been successfully applied to this domain since its initial proposal by (Kautz and Selman, 1992; Kautz and Selman, 1996). In classical STRIPS-style planning, each fact and each action at each step is commonly represented by an atomic variable. The logic of how actions may be executed and how they transform the world state is encoded with general clauses, while the initial state and the goal state are added to the formula in the form of *unit clauses* (i.e. clauses that contain only a single literal).

As the number of actions needed to comprise a valid plan of some given planning problem is generally unknown in advance, the planning problem is usually re-encoded iteratively for a growing amount of considered steps. As a consequence, the solving procedure only terminates when there is a plan. On problems for which no plan exists, common SAT planners will never identify this general unsatisfiability, but just indefinitely increase the number of considered steps.

The central advancement in SAT planning during the last years has been a more efficient action encoding paradigm based on the execution of multiple actions in parallel as long as some valid ordering on the actions exists (Rintanen, Heljanko, and Niemelä, 2004; Rintanen, Heljanko, and Niemelä, 2006). This method, also called $\exists$-step semantics, reduces both the size of the resulting encodings and the necessary amount of computational steps to reach a plan. One of the most recent, unified SAT planning approaches implementing these techniques is the planner Madagascar (Rintanen, 2014) which uses a non-incremental SAT Solver specifically written for this purpose. However, incremental SAT solving has been shown to sometimes speed up the planning procedure of this SAT planner (Gocht and Balyo, 2017).

In order to make SAT planning problems eligible for incremental SAT solving, Gocht and Balyo, 2017 have introduced a formula notation style named DimSpec[1]. In contrast to usual CNF notations in the DIMACS format, a DimSpec file specifies four different blocks of clauses, which are interpreted by an incremental SAT solving application as follows:

Initial state specification: These clauses will be added only once at the start of the computation.

Universal state specification: These clauses specify the “isolated” constraints of the problem at any given makespan, and are thus instantiated at each new iteration.

Goal state specification: These *unit clauses* are instantiated at the current makespan as assumptions before each solving attempt.

Transitional specification: These clauses specify all constraints between variables of the previous makespan with variables of the current makespan; they are instantiated at each iteration (except for the first).

In the context of classical planning, the initial state contains the facts which hold in the beginning, the goal state contains the goal facts, and the universal and transitional specification together model the workings of action preconditions, effects, and fact transitions. The DimSpec notation only supports constraints over variables in neighbored makespans, and the amount of clauses added at each makespan is constant. These restrictions will become relevant for the incremental encodings proposed in Chapter 5 and 6.

2.3 SAT Encodings for HTN Planning

An encoding approach of HTN planning problems in propositional logic has been proposed by (Mali and Kambhampati, 1998). In addition, a few practical enhancements to this approach have been suggested (Mali, 1999; Mali, 2000), such as the introduction of various heuristics and preprocessing procedures which were able to significantly reduce the size of the resulting formulae.

To my best knowledge, these publications have been the only published approaches regarding the encoding of HTN planning problems in propositional logic so far. The work at hand is thus the first to introduce SAT solving to modern HTN planning.

[1]See also `https://github.com/StephanGocht/Incplan`

Chapter 3

Model and Problem Statement

3.1 System model

The following definitions are based on the model from (Ghallab, Nau, and Traverso, 2004) as applied by (Ramoul et al., 2017), with some additions and modifications. Throughout the formal definitions, practical examples from the *Rover* domain will be provided.

At first, the syntactical composition of some fundamental objects are defined.

Definition 1 (Constant)**.** A **constant** is an atomic syntactic expression c.

Definition 2 (Typing system for constants)**.** A **type** is some subset of the set of all constants. A constant belongs to at least one type, and it can also have multiple types.

Common constants of *Rover* problems include expressions like *rover1* or *camera3*; they are referring to particular objects of the problem. Types like *Rover* and *Camera* are defined to bundle all objects of the corresponding kind. Types in PDDL domains can be defined in an polymorphic manner: for instance, *Rover* and *Lander* objects could have a common higher type *Machine*.

The basic embedding of such typed constants in expressions are defined next:

Definition 3 (Parameter)**.** A **parameter** is a placeholder variable for any constant of one or multiple particular types.

Definition 4 (Signature)**.** A **signature** of some object is a syntactic expression of the form $t(u_1, \ldots, u_k)$, where t is the *name* of the object and $u_1, \ldots, u_k$ are parameters, also called the *explicit parameters* of the object. Two signatures $t_1(x_1, \ldots, x_k)$ and $t_2(y_1, \ldots, y_k)$ are **matching** iff $t_1 = t_2$ and the types of all corresponding pairs of parameters are equal.

Definition 5 (Grounding, Instance)**.** The **grounding** of an object is the process of replacing each of the object's parameters with a valid constant. The resulting objects are called **instances** of the object.

For instance, *can_traverse(r, wa, wb)* is a signature where r is a parameter of type *Rover* and *wa* and *wb* are parameters of type *Waypoint*. A valid instance could be *can_traverse(rover1, waypoint7, waypoint4)*.

Next, we introduce predicates, facts, and world states:

Definition 6 (Predicate, Fact)**.** A **predicate** is an element of a special subset S of signatures united with the set of expressions $\overline{S} := \{\bar{s} \mid s \in S\}$, whereas $\bar{s} := \mathit{not}(s)$. Instances of predicates are called **facts**.

Definition 7 (World state)**.** A **world state**, or **state** in short, is a consistent set of facts, i.e. it contains either p or $not(p)$ for any fact p. A fact p **holds** in a state s iff $p \in s$.

In the following, when denoting a union $s_1 \cup' s_2$ on sets of facts, then the facts in s_2 replace any complementary facts from s_1 which would violate the previous condition of world states. Formally, we introduce the following adjusted operation: $s_1 \cup' s_2 := s_1 \cup s_2 \setminus \{p : \overline{p} \in s_2\}$

Continuing the previous example, *can_traverse(r, wa, wb)* is a predicate in the *Rover* domain, and *can_traverse(rover1, waypoint7, waypoint4)* can be one of the facts obtained by grounding the predicate.

In the next step, we begin to model how world states can be transformed.

Definition 8 (Operator)**.** An **operator** is a 3-tuple $o = (sig(o), pre(o), eff(o))$, where $sig(o)$ is the operator's signature, $pre(o)$ is a set of predicates, the *preconditions*, and analogously $eff(o)$ is the set of *effects*. $pre(o)$ and $eff(o)$ may only contain parameters which occur in $sig(o)$.

By the restriction on the occurrence of parameters, it is guaranteed that each operator may only have one single instance for each instance of its signature (because the signature already defines the entire "body" of the operator). The resulting instances are called actions:

Definition 9 (Action)**.** An **action** a is an instance of an operator o. The corresponding preconditions $pre(a)$ and effects $eff(a)$ are then sets of facts, and if $pre(a) \subseteq s$ for some state s, then a is **applicable** in s, and the **application** of a is defined as the resulting state $s' = \gamma(s, a) := s \cup' eff(a)$. On a sequence of actions, the application is recursively defined as $\gamma(s, \langle\rangle) = s$ and $\gamma(s, \langle a_1, \dots, a_k \rangle) = \gamma(\gamma(s, a_1), \langle a_2, \dots, a_k \rangle)$.

One of the operators of the Rover domain is *drop(rover, store)*. Its preconditions are $\{$*store_of(store, rover), full(store)*$\}$ and its effects are $\{$*not(full(store)), empty(store)*$\}$. Intuitively, this operator describes how a given rover may empty its store; if the given store belongs to the given rover and if the former is filled, then it can be ejected. Actions of this operator may look like *drop(rover1, store1)*.

The first hierarchical notion of the problem definition is now given in the form of methods:

Definition 10 (Method)**.** A **method** is a 3-tuple $m = (sig(m), pre(m), subtasks(m))$, where $sig(m)$ is the method's signature, $pre(m)$ is the set of preconditions that are required for the application of the method, and $subtasks(m) = \langle s_1, s_2, \dots, s_k \rangle$ is an ordered sequence of signatures which each match either the signature of another method or of an operator.

In general, a method may contain itself recursively as one of its subtasks. Also note that the preconditions and subtasks of a method may contain parameters which do not occur in $sig(m)$, in which case we call these *implicit* parameters. Note that the grounding of a method also involves substituting all of its implicit parameters by constants, so a single method can have multiple instances.

One of the methods of the Rover domain has the signature *send_soil_data(rover, from)*. Its preconditions are $\{$*at_lander(lander, w2) , visible(w1, w2)*$\}$. At this point, it can already be seen that this method contains implicit parameters, namely the waypoints *w1*, *w2* and the *lander* object. The subtasks of the method are $\langle$*do_navigate(rover, w1) , communicate_soil_data(rover, lander, from, w1, w2)*$\rangle$, where the first subtask is the signature of another method and the second one is the signature of an operator. Intuitively, the method models how a rover may fulfill the task of communicating certain data to a lander: It first needs to navigate to some waypoint from where the lander is visible (which might be a complex task itself) and then actually communicate the data to the lander.

In analogy to actions being instances of operators, we define reductions as instances of methods:

Definition 11 (Reduction). A **reduction** r is an instance of a method m. $pre(r)$ is a set of facts, and if $pre(r) \subseteq s$ for some state s, then r is **applicable** in s.

A possible reduction of the previous method could be *send_soil_data(rover1, waypoint3)* with the preconditions $\{$*at_lander(lander1, waypoint7), visible(waypoint1, waypoint7)*$\}$ and the subtasks $\langle$*do_navigate(rover, waypoint1), communicate_soil_data(rover, lander, waypoint3, waypoint1, waypoint7)*$\rangle$.

Next, we unite actions and reductions under the important notion of tasks:

Definition 12 (Task). A **task** t is an instance of a signature σ which matches the signature of either an operator or a method.
If σ matches the signature of an operator, then t is called *primitive* and it *references* the operator's action; if σ matches the signature of a method, t is called *non-primitive* and it *references* a set of reductions.

For instance, *send_soil_data(rover1, waypoint3)* is a task, and as the general signature references a method, it is non-primitive. Tasks are expressions defining *what* needs to be done; they do not directly specify *how* to do it, but they reference actions or reductions which provide such information.

All necessary structures are now assembled to define the structure of the problem at hand:

Definition 13 (HTN domain). An **HTN domain** is a 3-tuple $\mathcal{D} = (P, O, M)$ of predicates P, operators O and methods M. The set of tasks T of $\mathcal{D}$ is implied by its operators and methods.

Definition 14 (HTN problem). An **HTN problem** is a 5-tuple $\mathcal{P} = (\mathcal{D}, C, s_0, g, T_0)$ where $\mathcal{D}$ is the HTN domain the problem belongs to, C is a set of constants, s_0 is a state, the *initial state*, g is the set of *goal facts*, and T_0 is an ordered sequence of tasks, called the *initial task network*.

3.2 Problem statement

In order to define the full problem statement and the notion of a correct plan for such problems, the actual semantics of Hierarchical Task Networks need to be defined. Intuitively, with each applied decomposition inside some sequence of tasks, some non-primitive task is replaced by a new sequence of tasks, its subtasks. Doing such decompositions in an exhaustive manner should lead to a sequence of actions whose preconditions and effects are consistent and successively transform the initial state to some goal state.

Definition 15 (Plan). A **plan** π of a HTN problem $\mathcal{P} = (\mathcal{D}, C, s_0, g, T_0)$ is inductively defined as follows:

- If $T_0 = \langle\rangle$ and $g \subseteq s_0$, then the empty action sequence $\pi := \langle\rangle$ is a plan for $\mathcal{P}$.

Otherwise, let $T_0 := \langle t_1, \ldots, t_k \rangle$ for some $k \geq 1$.

- If t_1 is a primitive task referencing an action a which is applicable in s_0, and π' is a plan for the problem $\mathcal{P}' := (\mathcal{D}, C, \gamma(s_0, a), g, \langle t_2, \ldots, t_k \rangle)$, then $\pi := \langle a \rangle \cup \pi'$ is a plan for $\mathcal{P}$.
- If t_1 is a non-primitive task referencing a reduction r which is applicable in s_0, and π' is a plan for the problem $\mathcal{P}' := (\mathcal{D}, C, s_0, g, \langle subtasks(r), t_2, \ldots, t_k \rangle)$, then $\pi := \pi'$ is a plan for $\mathcal{P}$.

Definition 16 (Solvability)**.** An HTN problem $\mathcal{P}$ is **solvable** if and only if a plan π for $\mathcal{P}$ exists.

Definition 17 (HTN problem statement)**.**
Given a solvable HTN problem $\mathcal{P} = (\mathcal{D}, C, s_0, g, T_0)$, find a plan π for $\mathcal{P}$.

We conclude the model by providing some measures of a plan π, namely its length and its depth. To define a plan's depth, we introduce the notion of a task hierarchy:

Definition 18 (Task hierarchy)**.** Consider an HTN problem $\mathcal{P} = (\mathcal{D}, C, s_0, g, T_0)$. Its **task hierarchy** H is defined as a forest with the following properties:

- Each root node represents one of the tasks in T_0.
- Each node representing a primitive task is a leaf.
- Each node representing a non-primitive task is an inner node, and it has one child node for each of its possible subtasks.

Definition 19 (Length and depth of a plan)**.** Consider a plan π of an HTN problem $\mathcal{P}$ with its task hierarchy H. The **length** of π is defined as $|\pi|$ (the amount of actions in the plan). The **depth** of π is defined as the maximal depth of any task node which must be reached in H during its computation.

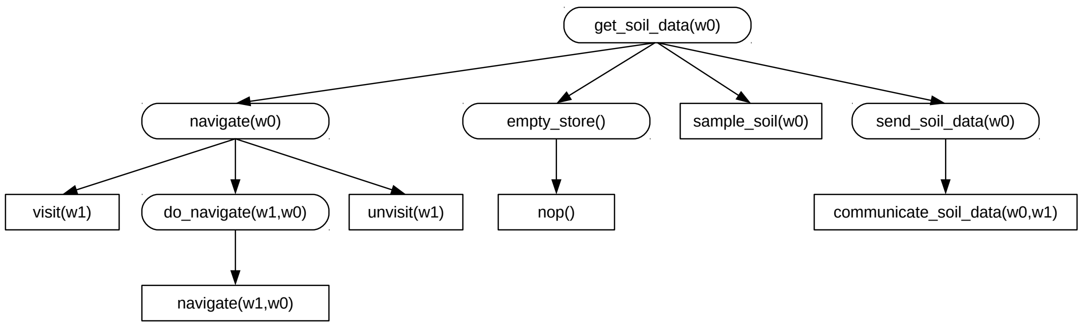

FIGURE 3.1: Exemplary hierarchy of a task from the *Rover* domain

As a complete example, Figure 3.1 illustrates a possible hierarchy of a single initial task *get_soil_data(w0)*. Non-primitive tasks are represented by boxes with round corners while primitive tasks are represented by rectangular boxes. Some task signatures have been shortened for the sake of brevity. Under the chosen reductions, the initial task is decomposed into four subtasks, among them one primitive and three non-primitive tasks. The non-primitive tasks are then reduced as well, until only primitive tasks remain. A plan for the initial task *get_soil_data(w0)* can be retrieved from the hierarchy by reading all primitive tasks from left to right, resulting in the following plan: *visit(w1); navigate(w1,w0); unvisit(w1); nop(); sample_soil(w0); communicate_soil_data(w0,w1)*.

The found plan has a length of 6 and a depth of 3 (with the action *navigate(w1,w0)* requiring the maximum depth).

The action *nop()* is a special action occurring in various domains; it represents "no operation", and as such it does not have any preconditions nor effects. Such an action can be convenient to model the scenario that some task's objective is already fulfilled, so it requires no actual actions to be executed. In this example, the store of the rover is already empty, so the task to empty its store is reduced to such a *nop* action. From the viewpoint of the application, such actions should not contribute to the plan length; consequently, the plan has a "true length" of 5 after eliminating the unneeded action.

3.3 Restrictions of the model

The problem statement which has just been introduced allows to model complex planning problems. Yet, some restrictions have consciously been done compared to the definition of Hierarchical Task Networks in (Ghallab, Nau, and Traverso, 2004), as explained in the following.

Definition 17 and its underlying definitions imply a total order on the initial task network as well as on the subtasks of each non-primitive task. In general, Hierarchical Task Networks as defined by (Ghallab, Nau, and Traverso, 2004) can be partially ordered; they may contain a set of ordering constraints between arbitrary subtasks. This restriction is in accordance with the previous work on HTN domains done by (Ramoul et al., 2017). In practice, many of the used HTN problem domains implicitly assume a total order on the subtasks they define, while others may work with different subtask orderings as well (however, they do not *rely* on such re-orderings).

Additionally, to comply with (Ramoul et al., 2017), the model has been restricted from using general causal constraints between subtasks to merely validating the applicability of reductions with preconditions. However, all of the encodings proposed in the following chapters may be extended to support these additional constraints.

Chapter 4

1st Contribution: GCT Encoding

In the following, an encoding of HTN planning problems modeled in PDDL into propositional logic is proposed, based on the *Linear Bottom-up forward* (LBF) encoding of HTN problems (Mali and Kambhampati, 1998). First, existing encodings and their issues from today's point of view are discussed. Then, a new encoding named *Grammar-Constrained Tasks* (GCT) encoding is proposed which builds upon LBF, but takes into account the issues identified before. Finally, some remarks about GCT's complexity and its general viability are made.

4.1 Discussion of previous encodings

Mali and Kambhampati, 1998 have proposed a number of encodings of HTN problems in propositional logic. In order to apply these encodings to HTN domains as described in PDDL, various issues had to be considered. Also note that no complete, formal specification of the clauses of the encodings by (Mali and Kambhampati, 1998) has been available; as a result, the encodings were not fully reproducible and had to be reconstructed.

A central restriction of the encodings by Mali and Kambhampati, 1998 concerning their expressiveness is that each task is assumed to have some fixed maximum amount of primitive steps when fully reduced. Under this assumption, it is easily possible to "allocate" a fixed amount of variables to each initial task in the plan, knowing exactly the point where a task, and ultimately the entire plan, will certainly have finished. By contrast, many common HTN problem domains can theoretically expand indefinitely due to recursive method definitions. As such, it is difficult or even impossible to decide in advance how many primitive steps a given task will take depending on which reductions are chosen and which facts hold before.

As another consequence of the non-recursive modelling of tasks, it has not been necessary to consider the scenario where some task has to be encoded multiple times at the same time step. But when allowing recursive method definitions, such a scenario may occur and has to be considered for the encoding.

Furthermore, the modeling of constraints in the previous encodings differs from the PDDL-compliant model provided in Chapter 3: method definitions feature a partially ordered *set* of subtasks, whereas we just consider a total order on the subtask *sequence*. Similarly, *before/after/between* constraints of subtasks are originally given pairwise and may specify *where* the needed precondition of a subtask is coming from, whereas the modeling used here describes these constraints in a flat manner as simple sets of facts that must hold before or after certain subtasks, no matter their origin.

The encoding by Mali and Kambhampati, 1998 which is the most applicable to today's state of the art in classical planning and which can be applied most directly to the model introduced in Chapter 3 is the *Linear Bottom-up Forward* (LBF) encoding. For this reason, this encoding has been examined primarily. The other two encodings are not state-based and they rely on causal relations and precedences between tasks and actions; the restrictions stated above are much harder to mend for these encoding types than for LBF. As the tasks cannot be assumed to be unique any more, some careful "variable indexing" needs to be done in order to define consistent constraints between logical variables. Such an indexing is already implied in the LBF encoding by the explicit enumeration of steps.

All of the encodings by Mali and Kambhampati, 1998 feature a composition of classical planning and HTN planning, taking into account a number of additional primitive (task-less) steps at the end of the computation. This has been omitted in the new encoding as the problem definition at hand exclusively features an initial task network. However, any HTN domain/problem definition can easily be extended to feature a number of initial tasks which may be reduced to any primitive action, thus achieving the same effect.

Lastly, it is worth noting that various improvements to the discussed encodings have been proposed (Mali, 1999; Mali, 2000), in particular by introducing preprocessing procedures and heuristics which significantly reduced the size of the resulting formulae. These enhancements are not further discussed in the work at hand for various reasons. As the modeling of the problem domains is quite different, the proposed improvements are only applicable to a very limited extent. In addition, today's preprocessing procedures are highly developed (Ramoul et al., 2017) and can likewise significantly reduce the problem complexity by efficient grounding algorithms. The encodings proposed here thus already exploit considerable simplifications of the problem at hand.

4.2 GCT Encoding

Applying the LBF encoding idea to the modeling used throughout this report, the *Grammar-Constrained Tasks* (GCT) encoding is proposed. As (Mali and Kambhampati, 1998) already noted, their general encoding approach is equivalent to supplementing a classical planning encoding with HTN-specific constraints which essentially enforce a valid grammar over the sequence of actions. The GCT encoding realizes such grammatical constraints in the form of defining start and end points of each reduction of some nonprimitive task, making heavy use of further variables in addition to the encoding of facts and actions.

In the following, the clauses of GCT are provided and explained, and their relations to the previous LBF clauses are mentioned where applicable. Atomic variables are written in a `typeface` style, and the mathematical notation complies with the model presented in Chapter 3. In Addition, s denotes some specific step, and n is the total amount of encoded steps. A denotes the set of all actions, and $R(t)$ denotes the set of possible reductions of a non-primitive task t. For the sake of simplicity, primitive tasks are also directly referred to as their corresponding action in some variable definitions.

4.2.1 Classical Planning clauses

The following clauses are a well-established way of encoding STRIPS-style planning, similar to the clauses presented in (Ghallab, Nau, and Traverso, 2004).

All facts from the specified initial state hold at the beginning:

$$\bigwedge_{p \in s_0} \texttt{holds}(p, 0) \wedge \bigwedge_{p \notin s_0} \neg\texttt{holds}(p, 0) \tag{4.1}$$

In the end, all facts from the goal state have to hold:

$$\bigwedge_{p \in g} \texttt{holds}(p, n) \tag{4.2}$$

The preconditions of an action a must hold for it to be executed at some step s, and the execution of a implies its effects at the step $s + 1$:

$$\texttt{execute}(a, s) \implies \bigwedge_{p \in pre(a)} \texttt{holds}(p, s) \wedge \bigwedge_{\overline{p} \in pre(a)} \neg\texttt{holds}(p, s) \tag{4.3}$$

$$\texttt{execute}(a, s) \implies \bigwedge_{p \in eff(a)} \texttt{holds}(p, s+1) \wedge \bigwedge_{\overline{p} \in eff(a)} \neg\texttt{holds}(p, s+1) \tag{4.4}$$

Exactly one action is executed per step:

$$\bigvee_{a \in A} \texttt{execute}(a, s) \tag{4.5}$$

$$\bigwedge_{a \neq a'} \neg\texttt{execute}(a, s) \vee \neg\texttt{execute}(a', s) \tag{4.6}$$

Rule 4.6 may be replaced by any valid set of clauses that ensures that no more than one of the given atoms can be `true`. In the implementation, a binary-style encoding has been used which requires $\mathcal{O}(n)$ helper variables and $\mathcal{O}(n \log n)$ clauses if the given set of atoms is of size n. More information on the used At-Most-One encodings is provided in Appendix A.

Facts may only change between two neighbored steps if an action is executed which has this change as an effect:

$$(\neg\texttt{holds}(p, s) \wedge \texttt{holds}(p, s+1)) \implies \bigvee_{p \in eff(a)} \texttt{execute}(a, s) \tag{4.7}$$

$$(\texttt{holds}(p, s) \wedge \neg\texttt{holds}(p, s+1)) \implies \bigvee_{\overline{p} \in eff(a)} \texttt{execute}(a, s)$$

4.2.2 Initial task network clauses

Let $T_0 = \langle t_0, \ldots, t_k \rangle$ the initial task network. The following clauses ensure the ordering and sequential execution of initial tasks. The first task begins at the first step; when an initial task ends, then the next initial task begins at the next step; the last task ends at the final step.

Corresponding "frame conditions" of the initial task network have not been explicitly provided in the LBF encoding, but are rather inferred from the textual description.

$$\texttt{taskStarts}(t_0, 0) \tag{4.8}$$

$$\bigwedge_{0 \leq i < k} \texttt{taskEnds}(t_i, s) \iff \texttt{taskStarts}(t_{i+1}, s+1) \tag{4.9}$$

$$\texttt{taskEnds}(t_k, n-1) \tag{4.10}$$

Clauses 4.11 essentially provide a valid grammar for the start and end points of tasks. Any start point of a task must precede a corresponding end point and vice versa. Helper variables and constraints between these, as defined here and in the following, have not been explicitly specified in the LBF encoding description.

$$\begin{aligned} \texttt{taskStarts}(t,s) &\implies \bigvee_{s' \geq s} \texttt{taskEnds}(t,s') \\ \texttt{taskEnds}(t,s) &\implies \bigvee_{s' \leq s} \texttt{taskStarts}(t,s') \end{aligned} \tag{4.11}$$

4.2.3 Task reduction clauses

The following clauses are added only for non-primitive tasks t. They define helper variables in order to uniquely refer to some particular task reduction beginning or ending at some computational step.

$$\begin{aligned} \texttt{taskStarts}(t,s) &\implies \bigvee_{r \in R(t)} \texttt{reductionStarts}(t,r,s) \\ \texttt{taskEnds}(t,s) &\implies \bigvee_{r \in R(t)} \texttt{reductionEnds}(t,r,s) \end{aligned} \tag{4.12}$$

To provide a meaning to the newly defined variables, it is enforced that the first subtask begins whenever the reduction begins, and the last subtask ends whenever the reduction ends.

$$\begin{aligned} \texttt{reductionStarts}(t,\langle t',\dots\rangle,s) &\implies \texttt{taskStarts}(t',s) \\ \texttt{reductionStarts}(t,\langle a,\dots\rangle,s) &\implies \texttt{execute}(a,s) \\ \texttt{reductionEnds}(t,\langle \dots,t'\rangle,s) &\implies \texttt{taskEnds}(t',s) \\ \texttt{reductionEnds}(t,\langle \dots,a\rangle,s) &\implies \texttt{execute}(a,s) \end{aligned} \tag{4.13}$$

All subtasks (both primitive (4.14) and non-primitive (4.15)) of any occurring non-primitive task t need to be completed within the execution of t.
These clauses correspond to the constraints (i) and (ii) of the LBF encoding, in a more explicitly provided step interval.

$$\begin{aligned} \texttt{reductionStarts}(t,\langle \dots,a,\dots\rangle,s) &\implies \bigvee_{s' \geq s} \texttt{execute}(a,s') \\ \texttt{reductionEnds}(t,\langle \dots,a,\dots\rangle,s) &\implies \bigvee_{s' \leq s} \texttt{execute}(a,s') \end{aligned} \tag{4.14}$$

$$\begin{aligned} \texttt{reductionStarts}(t,\langle \dots,t',\dots\rangle,s) &\implies \bigvee_{s' \geq s} \texttt{taskStarts}(t',s') \\ \texttt{reductionEnds}(t,\langle \dots,t',\dots\rangle,s) &\implies \bigvee_{s' \leq s} \texttt{taskEnds}(t',s') \end{aligned} \tag{4.15}$$

The preconditions of any applied reduction need to hold.

The following clauses, together with the classical planning clauses 4.3 and 4.4, effectively replace the arbitrary causal constraints between actions and tasks as specified in (v)-(ix), (xi), (xiii), (xiv) of the LBF encoding.

$$\texttt{reductionStarts}(t,r,s) \implies \bigwedge_{p \in pre(r)} \texttt{holds}(p,s) \wedge \bigwedge_{\overline{p} \in pre(r)} \neg\texttt{holds}(p,s) \tag{4.16}$$

Again, an additional set of variables is introduced. These denote that a task $\hat{t}$ (primitive or non-primitive) at step s' is part of some reduction of task t at step s. They are required in order to reference the subtask relationship between two tasks in an unambiguous manner in the following subtask ordering constraints.

$$\begin{aligned}
\texttt{reductionStarts}(t, \langle \dots, \hat{t}, \dots \rangle, s) &\implies \bigvee_{s' \geq s} \texttt{subtaskStarts}(t,s,\hat{t},s') && (4.17)\\
\texttt{subtaskStarts}(t,s,t',s') &\implies \texttt{taskStarts}(t',s')\\
\texttt{subtaskStarts}(t,s,a,s') &\implies \texttt{execute}(a,s')\\
\texttt{reductionEnds}(t, \langle \dots, \hat{t}, \dots \rangle, s) &\implies \bigvee_{s' \leq s} \texttt{subtaskEnds}(t,s,\hat{t},s') && (4.18)\\
\texttt{subtaskEnds}(t,s,t',s') &\implies \texttt{taskEnds}(t',s')\\
\texttt{subtaskEnds}(t,s,a,s') &\implies \texttt{execute}(a,s')
\end{aligned}$$

Now, the following clauses can (to a certain extent) enforce that the subtasks of any given task are totally ordered, in contrast to the arbitrary ordering constraints provided in (iii), (iv), (x), (xii) of the LBF encoding.

Assume that $r = \langle t'_1, \dots, t'_k \rangle$ and $1 \leq i < k$.

$$\begin{aligned}
&\texttt{reductionStarts}(t,r,s) \wedge \texttt{subtaskEnds}(t,s,t'_i,s')\\
&\qquad\qquad \implies \texttt{subtaskStarts}(t,s,t'_{i+1},s'+1) && (4.19)\\
&\texttt{reductionStarts}(t,r,s) \wedge \texttt{subtaskStarts}(t,s,t'_{i+1},s'+1)\\
&\qquad\qquad \implies \texttt{subtaskEnds}(t,s,t'_i,s') && (4.20)
\end{aligned}$$

At this point, if one would just use variables $\texttt{taskStarts}(t',s')$ instead of the more explicit $\texttt{subtaskStarts}(t,s,t',s')$, then the clauses may lead to unresolvable conflicts. This is because some task t' may then in fact not belong to the reduction of task t at step s, but still impose restrictions on some actual subtask of t. This problem is avoided by having introduced explicit variables for the subtask relationship between tasks.

4.3 Analysis and comments

In the following, a brief analysis on the complexity of the GCT encoding is provided. Hereby, T, F, and A correspond to the respective amount of tasks, facts, and actions. $r := \max \{ |R(t)| \mid t \in T \}$ is the maximal amount of reductions per task and $e := \max \{ |subtasks(r)| \mid r \in R \}$ is the maximal amount of subtasks per task reduction. In all complexity analyses provided in this document, the maximum amount of preconditions and effects per action and reduction is assumed to be some small constant and thus negligible for asymptotic measures.

The GCT encoding features $\mathcal{O}(S \cdot T)$ variables for the start and end of tasks, $\mathcal{O}(S \cdot T \cdot r)$ variables for the start and end of task reductions, and $\mathcal{O}(S^2 \cdot T^2)$ variables for modeling the subtask relationship between tasks. In addition, $\mathcal{O}(S \cdot F)$ variables for the encoding of facts, $\mathcal{O}(S \cdot A)$ variables for the execution of actions and $\mathcal{O}(S \cdot A \log A)$ helper variables for *At-Most-One* constraints are needed. This leads to a total variable complexity of $\mathcal{O}\big(S^2T^2 + S(Tr + A\log A + F)\big)$, which in practice is clearly dominated by the term S^2T^2.

Similarly, the clause complexity of the GCT encoding is asymptotically dominated by rules 4.19, 4.20 which cause a total of $\mathcal{O}(S^2 \cdot T^2 \cdot r \cdot e)$ clauses. The clause sizes can be linear either in the amount of tasks, of reductions per task, or of steps.

In comparison, the original LBF encoding by (Mali and Kambhampati, 1998) featured a variable complexity of $\mathcal{O}(S^2 \cdot TA \cdot r^2 \cdot e)$ and a clause complexity of $\mathcal{O}(S^3 \cdot TA \cdot r^2 \cdot e)$. The T^2 factor in GCT's complexity is fundamentally caused by the admission of recursive task definitions, while the r^2 factor in the complexity of LBF may originate from the arbitrary constraints that can be specified between reductions. However, the LBF encoding has a clause complexity which is cubic, not quadratic, in the number of encoded steps.

While the GCT encoding specifies a considerable number of constraints on the execution of primitive actions, not all HTN constraints are necessarily fully obeyed. Slight interleavings of tasks may occur when recursive tasks are contained in the problem; for instance, the subtask specifications (4.14, 4.15) allow for a subtask to start during the reduction of its parent task, but to end at some arbitrary later point, provided that the same task re-occurs at a deeper level of the hierarchy and ends during the parent task's reduction. Also note that 4.9 causes the encoding to be restricted to problems where each of the initial tasks is unique, i.e. it does not re-occur as another initial task or as some subtask.

In general, a less ambiguous way of referencing the subtasks of some specific task is needed in order to make the encoding fully compliant to the problem statement in all special cases.

To conclude, the GCT encoding is an extension of a naïve primitive action-based encoding, introducing a polynomial amount of additional clauses and variables in order to satisfy the HTN constraints. In practice, this additional encoding layer is a high price to pay, considering that HTN constraints are supposed to *simplify* the task of finding a plan, instead of rendering the problem harder than before. As a result, today's state-of-the-art encodings of classical planning (*not* making use of any HTN information) are presumably significantly faster than the HTN-based encoding at hand (what will be confirmed in Chapter 8).

In addition, the encoding as proposed has a "one-time use" – it is encoded for one specific maximum plan length, and if a SAT solver finds the encoding to be unsatisfiable, then an entire new encoding considering a higher amount of steps has to be created, and the SAT solver has to restart the entire solving process.

In the following chapters, new encodings improving on these shortcomings will be proposed.

Chapter 5

2nd Contribution: SMS Encoding

The GCT encoding proposed in the last chapters has various shortcomings restricting its practical use. Most significantly, it has a polynomial complexity of variables and clauses, rendering it inefficient both for the encoding process and for the solving stage, and a complete re-encoding has to be performed for each additional computational step considered.

To improve on the encoding, an initial idea has been to adjust the encoding in order to make it eligible for incremental SAT solving. This way, an abstract encoding is created only once and is then handed to the solver without any given limit on the maximal number of computational steps to consider. The solver can instantiate the clauses as needed and will remember conflicts learned from previous solving steps, significantly reducing the necessary run time.

However, to achieve an incremental formulation of GCT, a number of challenges need to be considered. Specifically aiming at a DimSpec-compliant encoding (see Chapter 2.2.2), it is necessary to reformulate all the clauses such that each clause only contains variables from the computational steps i and $i+1$, for some clause-specific i, and that for each time step, all of the encoded clauses follow a single, general "construction scheme". In the following, such an encoding is presented, eventually turning out to differ significantly from the initial GCT encoding.

5.1 Abstract description

The encoding described in the following has been named *Stack Machine Simulation* (SMS) encoding. Essentially, the encoding simulates a stack of tasks which is transformed between computational steps, always checking the task on top of the stack and either `push`ing the subtasks of one of its reductions if it is non-primitive, or executing the corresponding action if it is primitive. This central idea is illustrated in Fig. 5.1, using the previous example from Fig. 3.1. Such transitions are performed repeatedly until the stack is empty. Using a stack memory, it is possible to transfer the entire currently considered task hierarchy from one step to the next, eliminating the necessity of clauses which link two computational steps that are further apart.

With this general procedure, the initial tasks and their subtasks will be sequentially processed and broken down into subtasks until all tasks are removed and only *bottom* remains.

5.2 Realization

The following clauses specify the default variant of the SMS encoding. For these definitions, σ denotes the stack size, a parameter of the encoding. The encoding is given in a DimSpec-compliant format, separated into four specifications whose clauses

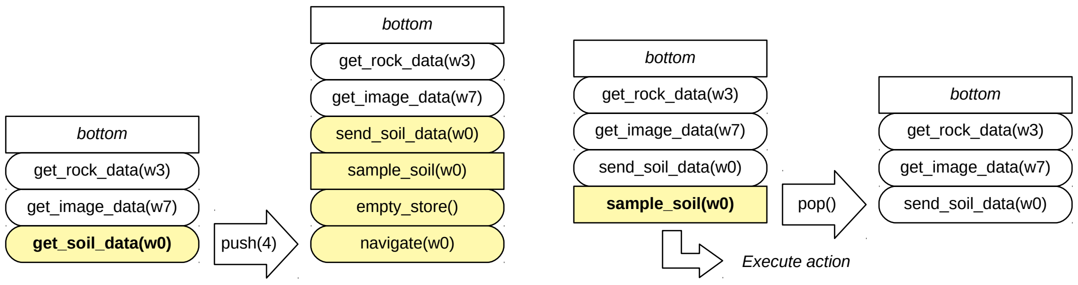

FIGURE 5.1: Illustration of the two central transitions between computational steps in the SMS encoding; pushing the subtasks of a non-primitive task (left) and executing an action (right)

will be instantiated accordingly during the solving attempt (see Chapter 2.2.2). In the transitional clauses, an `atom` at the following step is denoted as `atom`′.

5.2.1 Initial state clauses

All facts specified in the initial state must hold:

$$\bigwedge_{p \in s_0} \texttt{holds}(p) \wedge \bigwedge_{\overline{p} \in s_0} \neg\texttt{holds}(p) \tag{5.1}$$

The initial stack contains the initial tasks and a *bottom* symbol afterwards. Again, let $T_0 = \langle t_0, \ldots, t_{k-1} \rangle$ the initial task network:

$$\bigwedge_{0 \le i < k} \texttt{stackAt}(i, t_i) \tag{5.2}$$

$$\texttt{stackAt}(k, \mathit{bottom}) \tag{5.3}$$

5.2.2 Goal state clauses

The stack must be empty in the end:

$$\texttt{stackAt}(0, \mathit{bottom}) \tag{5.4}$$

All facts from the goal state must hold:

$$\bigwedge_{p \in g} \texttt{holds}(p) \tag{5.5}$$

5.2.3 Universal clauses

The execution of an action implies its preconditions to hold:

$$\texttt{execute}(a) \implies \bigwedge_{p \in pre(a)} \texttt{holds}(p) \wedge \bigwedge_{\neg p \in pre(a)} \neg\texttt{holds}(p) \tag{5.6}$$

At each step, all the $\texttt{push}(k)$ and $\texttt{pop}()$ operations are mutually exclusive.

$$\mathit{AtMostOne}\left\{\{\texttt{push}(k) \mid 0 \le k < \mathit{maxPushes}\} \cup \{\texttt{pop}()\}\right\} \tag{5.7}$$

If no pop() operation is done, then no action is executed, enforced by a "virtual" action which does not have any preconditions or effects.

$$\neg\texttt{pop}() \implies \texttt{execute}(\mathit{noAction}) \tag{5.8}$$

If a primitive task corresponding to some action a is on top of the stack, then a is executed (and only this action); additionally, a pop operation is done.

$$\begin{aligned} \texttt{stackAt}(0, a) &\implies \texttt{execute}(a) \land \texttt{pop}() \\ \texttt{execute}(a) &\implies \texttt{stackAt}(0, a) \end{aligned} \tag{5.9}$$

If a non-primitive task t is on top of the stack, then one of its possible reductions must be applied.

$$\texttt{stackAt}(0, t) \implies \bigvee_{r \in D(t)} \texttt{reduction}(r) \tag{5.10}$$

If a non-primitive task on top of the stack is reduced by some specific reduction $r = \langle t_1, \dots, t_k \rangle$, then a push by the amount of subtasks in the reduction is performed, and all of its preconditions must hold.

$$\begin{aligned} \texttt{stackAt}(0, t) \land \texttt{reduction}(r) \implies \Big(&\texttt{push}(k-1) \\ &\land \bigwedge_{p \in pre(r)} \texttt{holds}(p) \land \bigwedge_{\overline{p} \in pre(r)} \neg\texttt{holds}(p)\Big) \end{aligned} \tag{5.11}$$

5.2.4 Transitional clauses

The execution of an action implies its effects to hold in the next step.

$$\texttt{execute}(a) \implies \bigwedge_{p \in \mathit{eff}(a)} \texttt{holds}'(p) \land \bigwedge_{\overline{p} \in \mathit{eff}(a)} \neg\texttt{holds}'(p) \tag{5.12}$$

Frame axioms: Facts only change if a supporting action is executed.

$$\begin{aligned} (\neg\texttt{holds}(p) \land \texttt{holds}'(p)) &\implies \bigvee_{p \in \mathit{eff}(a)} \texttt{execute}(a) \\ (\texttt{holds}(p) \land \neg\texttt{holds}'(p)) &\implies \bigvee_{\bar{p} \in \mathit{eff}(a)} \texttt{execute}(a) \end{aligned} \tag{5.13}$$

Stack movement: The stack content moves accordingly to the performed operation at a given step.

$$\begin{aligned} \bigwedge_{0 < s < \sigma - k} \texttt{push}(k) \land \texttt{stackAt}(s, t) &\implies \texttt{stackAt}'(s + k, t) \\ \bigwedge_{s > 0} \texttt{pop}() \land \texttt{stackAt}(s, t) &\implies \texttt{stackAt}'(s - 1, t) \end{aligned} \tag{5.14}$$

A non-primitive task and a chosen reduction together define the corresponding subtasks as the new stack content at the positions which are freed by the occurring push.

$$\texttt{stackAt}(0, t) \land \texttt{reduction}(\langle t'_0, \dots, t'_{k-1} \rangle) \implies \bigwedge_{i=0}^{k-1} \texttt{stackAt}'(i, t'_i) \tag{5.15}$$

5.2.5 Variants

The most straight-forward variant of the SMS encoding, named `SMS-ut` (unary tasks), conventionally encodes tasks as the content of the stack, as just described.

Another variant `SMS-ur` (unary reductions) does not encode tasks, but instead reductions as the content of the stack. This leads to transitional clauses of a differing kind, and the set of variables deciding the chosen reduction are no more necessary.

In a third variant `SMS-bt` (binary tasks), the stack content is encoded not with one variable for each possible task at each position, but instead with one variable for each binary digit of a number which represents the task at that position. This reduces the total variable requirement for the stack content at each cell from $\mathcal{O}(T)$ to $\mathcal{O}(\log T)$ per computational step, but complicates some transitional clauses.

5.3 Analysis

In the following section, the termination and the correctness of a SAT solving procedure using the SMS Encoding are argumented. Proofs are given in the form of sketches, meant to provide an intuition on why the general design of the encoding satisfies the corresponding properties.

5.3.1 Termination and correctness

Theorem 1 (Termination and correctness of the SMS encoding)**.** Consider a solvable HTN problem $\mathcal{P} = (\mathcal{D}, C, s_0, g, T_0)$. Using the task-based, unary-style SMS encoding with a sufficiently large stack size, an incremental SAT solving procedure terminates after at most N computational steps, and a valid plan can be extracted from the found satisfying assignment. Hereby, N is equal to the length of the found plan plus the amount of applied reductions during the calculation.

Sketch of proof. As $\mathcal{P}$ is solvable, some plan π of length $n \geq 0$ exists. The idea is to show that each computational step of the SMS encoding essentially performs one inductive step as provided in Definition 15.

- If $n = 0$, then the initial state and goal state specifications of SMS enforce that all initial tasks are on the stack, that the stack is empty, and that both the initial state and the goal state hold. This formula is satisfiable if and only if the sequence of initial tasks is empty and if $g \subseteq s_0$. So the encoding leads to some satisfying assignment if and only if $\pi := \langle\rangle$ is a valid plan for $\mathcal{P}$.

- If $n > 0$, then the plan π is defined inductively by one "atomic" reduction step and a subsequent plan for the remaining problem. The following two cases are possible:

  If the top of the stack contains some primitive task t, then (by definition of the transitional clauses) one computational step will execute the action (thus enforcing its preconditions to hold at this step and its effects to hold at the next step) and remove the task from the stack, before proceeding to perform the remaining calculation.

  If instead some non-primitive task t is on top of the stack, then some reduction is chosen, its preconditions are enforced to hold, and all corresponding subtasks are pushed onto the stack, removing t itself.

> In both cases, the behaviour of the simulated stack machine is identical to the inductive definition of a plan. Consequently, the encoding leads to some satisfying assignment if and only if the computation of a valid plan has finished.

To conclude, the simulated stack machine executes a sequence of actions which is conform to the definition of a plan.

After a successful computation, this plan can then be extracted in the form of the satisfying assignment which has been found for the action execution variables.

The upper limit on the amount of needed steps directly follows: At each step, either a reduction is applied or an action is executed (and appended to the plan), leading to a total amount of steps equal to the number of applied reductions plus the plan length. □

5.3.2 Complexity

In the following, the asymptotic complexity of clauses and variables of the encoding is discussed. The task-based, unary-style SMS encoding variant is chosen for this purpose. Assume that after n computational steps, a plan π of length $|\pi| \leq n$ is found.

The amount of variables is dominated by the encoding of the stack itself: If a stack of size σ has been encoded, then $\mathcal{O}(n \cdot \sigma \cdot T)$ variables are used to encode the stack. Additionally, the action executions contribute $\mathcal{O}(n \cdot A)$ variables and the state encoding makes up for $\mathcal{O}(n \cdot F)$ variables. Last but not least, $\mathcal{O}(n \cdot r)$ variables representing the chosen reduction of the current task are needed for $r := \max\{|R(t)| \mid t \in T\}$, and for AMO constraints over actions, $\mathcal{O}(n \cdot \log A)$ helper variables are used. This leads to a total of $\mathcal{O}\big(n \cdot (\sigma T + A + F + r)\big)$ variables. Note that with some $\sigma \in \mathcal{O}(n)$, this measure implies a quadratic term $n^2 T$.

Regarding the amount of clauses, the classical planning clauses include $\mathcal{O}(n \cdot A)$ clauses for preconditions and effects of actions, and $\mathcal{O}(n \cdot F)$ clauses for frame axioms. To uniquely specify the content of the stack at each computational step, $\mathcal{O}(n \cdot \sigma \cdot T \cdot e)$ transitional clauses (for each `push(k)`, $0 < k \leq e$, and for `pop()`) are needed, where $e := \max\{|subtasks(r)| \mid r \in R\}$. $\mathcal{O}(n \cdot A)$ clauses link the stack operations with the execution of actions. As e is usually a small constant, a pairwise AMO constraint over them is added according to rule 5.7 by introducing $\mathcal{O}(n \cdot e^2)$ clauses. For the actual transitions, each task which can be on top of the stack causes $\mathcal{O}(r)$ clauses; consequently, $\mathcal{O}(n \cdot T \cdot r)$ clauses are needed. In total, $\mathcal{O}\big(n \cdot (A + F + \sigma Te + e^2 + Tr)\big)$ clauses are added. Again assuming $\sigma \in \mathcal{O}(n)$, the dominating factor becomes $n^2 Te$.

5.4 Discussion

In the following, some advantages and disadvantages of the proposed encoding are discussed.

The SMS encoding explicitly stores the entire hierarchy in the step-dependent stack content. For this reason, recursive subtask relations or some task occuring multiple times in a successive manner do not pose a problem. The resulting calculation is perfectly stable with respect to such particular properties of a problem.

By design, the encoding is DimSpec-compliant. As a consequence, the encoding can benefit from generic incremental SAT solving procedures where the formulae are automatically instantiated for each additional step and no full re-encoding is needed for each further step considered.

The completeness of the SMS calculation depends on the maximum considered stack size. However, as the DimSpec format inserts the same amount of variables and clauses at each additional step, this parameter has to be known a priori in order to create the encoding. A safe upper-bound on the stack size is the maximum number of computational steps being considered: Every pushed task will need to be removed later such that the stack is empty in the end, so there is no valid calculation with n computational steps and a maximal stack size higher than n.

In general, the maximal amount of computational steps is desired to be left unbounded; a solver should continue until a solution is found or until the computational resources are exhausted. As a consequence, providing the maximum stack size as a hyper-parameter to the encoding is somewhat unsatisfactory and limits the computation's completeness. Alternatively, the DimSpec format must either be replaced or generalized to some specification which allows the addition of variables and clauses in an adaptive manner.

Another issue arises from the necessary amount of computational steps as proven in Theorem 1. Naïve classical SAT planning procedures take exactly n steps to find a plan of length n, whereas SMS takes $n + n'$ steps where n' is the amount of applied reductions. In fact, n' can be of significant size. For instance, consider that the hierarchy forms a balanced binary tree (where, as described in Definition 18, each leaf is an action of the plan and each inner node is a reduction to apply). Assuming n to be a power of two for the sake of simplicity, this leads to $n' = n - 1$ reductions, implying that solving the SMS encoding takes $2n - 1$ steps.

There are even worse cases: If some tasks are reduced to a *single* subtask, then the amount of inner nodes in the tree is not bound by the amount of leaves any more. The worst case in such cases is only limited by the amount of applicable reductions that can be chained to a long sequence of subtask relations without forming a loop in state-space.

To sum up this issue, the amount of needed computational steps is quite large and may render the explorable space of possible variable assignments very big.

The SMS encoding presented in this chapter demonstrates that HTN planning with incremental SAT solving is possible in a well-structured and comparably lightweight manner. The central shortcomings of the SMS encoding, namely its inflexible, parameterized stack encoding and the large amount of necessary computational steps, are both dealt with in the following chapter by proposing the T-REX encoding.

Chapter 6

3rd Contribution: T-REX Encoding and Instantiation

The SMS encoding presented in the last chapter has been found to work reliably. Yet, it requires a problem-dependent parameter and it does not handle bigger problems well, one of the reasons being the large number of computational steps needed to solve the problem. In the following, a new encoding approach is presented which entirely avoids these restrictions.

6.1 Abstract description

As a point of departure, consider a sequence of reductions and actions just as they occur as the stack content of the SMS encoding at any computational step. In the SMS encoding, only the *first* element of this sequence is removed from the sequence and processed in an isolated manner, leading to a long linear chain of small changes to the stack's content.

The main idea of the following encoding is to replace the stack structure with a general array of elements, and to process *all* of the stored elements in one single computational step. As illustrated in Fig. 6.1, all non-primitive tasks which are present at some computational step are reduced in a simultaneous manner while preserving the order of all tasks in the array and while also enforcing the consistency of facts with respect to the occurring preconditions and effects. This implies that it does not suffice any more to maintain a single set of facts for each computational step; instead, all respectively relevant facts need to be encoded *at each position* of the array, at each step, and their truth values need to be propagated between successive layers. The encoding now effectively grows along the *depth* of the hierarchy until all elements in the array are primitive actions, implying a finished plan.

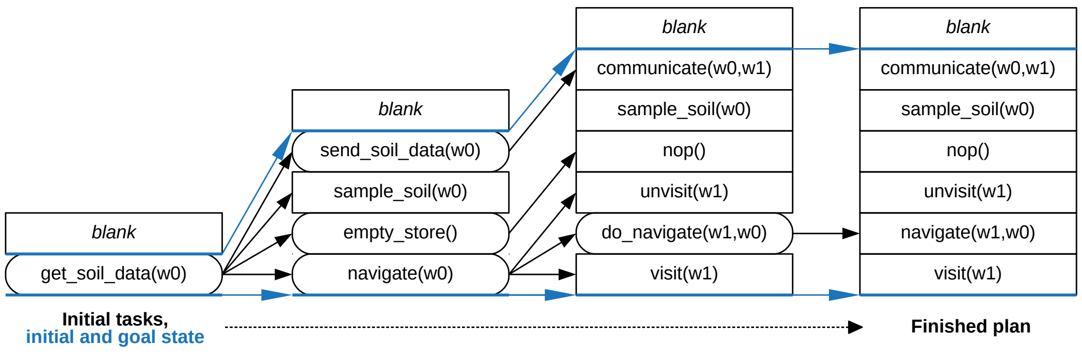

FIGURE 6.1: Illustration of the transitions between computational steps in the T-REX encoding

The described approach has been called **T-REX**, short for **T**ree-like **R**eduction **Ex**ploration, as the encoding results in a tree structure of reductions which is being explored in a breadth-first manner, up to an iteratively growing depth. Using this technique, a planner can find a plan after a number of computational steps corresponding to the minimal depth of the hierarchy for which a plan exists. Just like the arms of the prehistoric *Tyrannosaurus Rex* in relation to its total size, this depth to explore is very short for most of the considered problems. This property leads to a fast solving process, just like a T-REX was allegedly able to run very fast[1].

6.2 Clauses

In the following, the clauses of the default T-REX encoding are provided. Note that the encoding does not directly feature non-primitive tasks, but their possible reductions instead. This is because the amount of subtasks of a given reduction is always fixed whereas the number of children of a given non-primitive task may vary. The propagation of elements can be implemented much more easily in the former case.

6.2.1 Initial state clauses

The initial facts hold at position 0:

$$\bigwedge_{p \in s_0} \texttt{holds}(p, 0) \land \bigwedge_{p \notin s_0} \neg\texttt{holds}(p, 0) \tag{6.1}$$

At each position i at the initial layer, any reduction of the i-th initial task t_i is possible.

$$\bigwedge_{i=0}^{k-1} \bigvee_{r \in R(t_i)} \texttt{element}(r, i) \tag{6.2}$$

The last position k at the initial layer contains a *blank* element:

$$\texttt{element}(\mathit{blank}, k) \tag{6.3}$$

At the last position k at the initial layer, all goal facts hold:

$$\bigwedge_{p \in g} \texttt{holds}(p, k) \tag{6.4}$$

6.2.2 Universal clauses

The presence of an action at some position i implies its preconditions at position i and its effects at position $i+1$:

$$\begin{aligned} \texttt{element}(a, i) &\implies \bigwedge_{p \in \mathit{pre}(a)} \texttt{holds}(p, i) \land \bigwedge_{\overline{p} \in \mathit{pre}(a)} \neg\texttt{holds}(p, i) \\ \texttt{element}(a, i) &\implies \bigwedge_{p \in \mathit{eff}(a)} \texttt{holds}(p, i+1) \land \bigwedge_{\overline{p} \in \mathit{eff}(a)} \neg\texttt{holds}(p, i+1) \end{aligned} \tag{6.5}$$

[1]The actual running speed of the *Tyrannosaurus Rex* is a somewhat controversial topic in research, with more recent publications suggesting quite moderate peak values of around 8 ms^{-1} (Sellers and Manning, 2007).

A reduction at some position i implies its preconditions at that position:

$$\texttt{element}(r,i) \implies \bigwedge_{p \in pre(r)} \texttt{holds}(p,i) \wedge \bigwedge_{\bar{p} \in pre(r)} \neg\texttt{holds}(p,i) \tag{6.6}$$

Each action is primitive, and each task reduction is non-primitive (effectively eliminating the possibility of an action and a task reduction to co-occur):

$$\begin{aligned} \texttt{element}(a,i) &\implies \texttt{primitive}(i) \\ \texttt{element}(r,i) &\implies \neg\texttt{primitive}(i) \end{aligned} \tag{6.7}$$

If a fact changes, then either this position does not contain any action yet or it contains any action which supports this fact change.

$$\begin{aligned} \neg\texttt{holds}(p,i) \wedge \texttt{holds}(p,i+1) &\implies \neg\texttt{primitive}(i) \vee \bigvee_{p \in eff(a)} \texttt{element}(a,i) \\ \texttt{holds}(p,i) \wedge \neg\texttt{holds}(p,i+1) &\implies \neg\texttt{primitive}(i) \vee \bigvee_{\bar{p} \in eff(a)} \texttt{element}(a,i) \end{aligned} \tag{6.8}$$

At each position, all possibly occurring actions are mutually exclusive. (Note that this also includes the *blank* action variable.)

$$\mathit{AtMostOne}\left\{ \texttt{element}(a,i) \mid a \in A \right\} \tag{6.9}$$

6.2.3 Goal state clauses

For the full formula to be satisfiable, all positions of the last (i.e. the current) hierarchical layer l must be primitive.

$$\bigwedge_{0 \le i < k_l} \texttt{primitive}(i) \tag{6.10}$$

6.2.4 Transitional clauses

In the following, if a variable `atom` is instantiated at some layer $l-1$, then the variable $\texttt{atom}'$ corresponds to that variable at layer l. In addition, for any position i at some layer $l-1$, the *successor position* i' at layer l is the new position at that layer to which the elements are shifted depending on the applied reductions at previous positions.

A fact p holds at some position i if and only if it also holds at the successor position i' at the next hierarchical layer.

$$\texttt{holds}(p,i) \iff \texttt{holds}'(p,i') \tag{6.11}$$

If an action occurs at some position i, then it will also occur at the successor position i' at the next hierarchical layer.

$$\texttt{element}(a,i) \implies \texttt{element}'(a,i') \tag{6.12}$$

In the following, the expression $e(i)$ is equal to the maximum length of an expansion that any of the potential elements at position i at layer $l-1$ may induce. It is evaluated while instantiating the clauses.

If an action occurs at some position i, then all further child positions $i' + j$ at the next layer will contain a *blank* element. For $0 < j < e(i)$, we have:

$$\texttt{element}(a, i) \implies \texttt{element}'(blank, i' + j) \tag{6.13}$$

If a task reduction occurs at some position i, then it creates any of its possible subtasks at the next layer. Let $subtasks(r) = \langle t_0, \ldots, t_{k-1} \rangle$:

$$\texttt{element}(r, i) \implies \bigwedge_{c=0}^{k-1} \bigvee_{r' \in R(t_c)} \texttt{element}'(r', i' + c) \tag{6.14}$$

Any positions at the next layer which would stay undefined by the task's expansion are filled with *blank*symbols. For $k \leq j < e(i)$, we have:

$$\texttt{element}(r, i) \implies \texttt{element}'(blank, i' + j) \tag{6.15}$$

6.3 Realization

The following section describes in more detail how the approach has been realized in practice and which optimizations have been done in order to allow for an efficient encoding and solving process.

6.3.1 Sparse Element Encoding

Encoding each of the problem's reductions and actions at each position at each considered depth leads to a high number of variables (comparable to the stack of SMS) and clauses. Instead, only the elements which are actually possible at the given position and depth will be encoded. The occurrence of an element is decided during the instantiation process as follows:

At the initial layer 0, exactly the possible initial task reductions are possible at their corresponding position, followed by a single *blank* action. Inductively, given the possible elements at some position p at layer l which gets mapped to the positions $\{p', \ldots, p' + k\}$ at layer $l + 1$, the successors of these possible elements will be possible at layer $l + 1$, beginning at position p'. Finally, if an element has no successors at one or multiple child positions regarding the previous rules, then a *blank* element will be possible there.

Note that in this definition, the successors of actions are defined as the element itself at offset zero, and no further elements.

Using this reduced set of possible elements, no variables encoding the respective impossible elements are needed and the domain of variables as well as the amount of clauses can be reduced significantly. The index of an array's last position where some element may occur corresponds to the total array size to encode.

6.3.2 Efficient Encoding and Instantiation

When using the SMS encoding, the maximal stack size has to be provided as a hyperparameter, because for each additional makespan to consider, exactly the same set of clauses has to be instantiated. For the T-REX approach, a more specialized encoding strategy has been developed to ensure an efficient and complete solving process without any required parameters. Essentially, an encoding similar to the DimSpec

```
Data: T-REX abstract encoding file
Result: Satisfying variable assignment, if a solution exists; $\perp$, else
Process abstract encoding;
Build data structures of element successors and possible fact changes;
nextLayer $\leftarrow$ CalculateFirstLayer();
foreach $p \leftarrow 0$ to nextLayer.ArraySize() $-1$ do
    Add universal clauses at (nextLayer, position $p$);
end
result $\leftarrow$ UNSAT;
while result $\neq$ SAT do
    prevLayer $\leftarrow$ nextLayer;
    nextLayer $\leftarrow$ prevLayer.CalculateNextLayer();
    foreach $p \leftarrow 0$ to prevLayer.ArraySize() $-1$ do
        Add transitional clauses from (prevLayer, position $p$)
          to (nextLayer, position prevLayer.NextPosition($p$));
    end
    foreach $p \leftarrow 0$ to nextLayer.ArraySize() $-1$ do
        Add universal clauses at (nextLayer, position $p$);
        Assert goal literal at (nextLayer, position $p$);
    end
    result $\leftarrow$ SatSolve();
end
Output satisfying variable assignment.
```

Algorithm 1: T-REX Instantiation procedure

format is produced, but the specified clauses are instantiated along *two* dimensions, namely the hierarchical depth *and* the cells of the array at that depth.

Algorithm 1 provides an abstract view on the instantiation procedure. After processing the abstract encoding file, all general data structures which are necessary are computed, such as the possible successors (or children) of each reduction and the facts that may be potentially changed by each reduction. Thereupon, an initial layer of the hierarchy is computed from the initial state specification of the encoding. A layer object hereby encapsulates various data structures determining which elements and facts may occur at which positions, which positions are preceding which positions at the next layer, and which elements get mapped to which boolean variables in the resulting formula. A `while` loop is then executed as long as the problem is unsolved, always calculating the succeeding layer of the layer before, adding all needed clauses, and calling the SAT Solver.

More details about the developed T-REX Interpreter which performs the just-in-time creation of such clauses and the communication with a SAT solver are provided in Chapter 7.2 (Implementation).

6.3.3 Encoding optimizations

In the following, some considered optimizations of the implemented encoding are explained.

Sparse Fact Encoding

Just like the sparse encoding strategy for reductions and actions, the amount of encoded facts can be reduced as well. Using information of which facts might be changed by which reductions, the following strategy can be employed:

- At each depth, all facts are encoded at the first and at the last position.
- At each depth, a fact p is encoded at position i if an action with p as an effect may occur at position $i-1$, or if a reduction possibly leading to a change of p may occur at position $i-1$.

Frame axioms (Rule 6.8) for a fact p are adjusted to not necessarily link neighbored positions, but only the positions which contain a variable for p. For preconditions of actions and reductions, the following rule is used: When a fact is not encoded at some position i, then the fact variable at the highest possible position $j < i$ is used instead. This is valid because the fact cannot change inbetween the interval $(j, i]$ by definition of the sparse fact encoding rule. Essentially, constraints involving facts will "jump" over positions where the fact is known to remain unchanged.

Variable reusage

When a fact is encoded at position i at layer $l-1$ and also encoded at the successor position i' at layer l, then the same boolean variable can be used for $\texttt{holds}(p, i)$ and $\texttt{holds}'(p, i')$; the fact does not need to be explicitly re-encoded as a new variable. This eliminates the need for fact propagation clauses (Rule 6.11).

The same technique may be applied to actions: If an action occurs at some spot, then it will be propagated without ever vanishing, so the same action variable may be reused for all succeeding layers. However, there is a subtle issue with this representation: An action variable might in fact need to be re-encoded in the case that the same action has already been possible at a predecessor position at some previous layer, because the different sets of frame axioms might be conflicting otherwise.

At-Most-One constraints

When using one boolean variable for each possible element, multiple elements may be present at the same position and depth. To enforce a correct realization of the problem's hierarchical structure, At-Most-One constraints over actions are added. No At-Most-One constraints over reductions are needed because any co-occurring set of reductions at the same spot will inevitably lead to a conflict if their expansions differ regarding their final primitive actions.

To further optimize At-Most-One constraints, a threshold parameter has been introduced which decides on the kind of encoding to be done (see Appendix A) depending on the specific amount of actions at some position. Additional variations and modifications considered in the evaluation include encoding At-Most-One constraints over *all* elements (instead of only constraining actions) or encoding only a single, successively accumulating set of At-Most-One constraints for each propagating position of the hierarchy when action variables are being reused.

Successor clauses

As the successors of composite elements (Rules 6.14, 6.15) and the propagation of primitive elements (Rules 6.12, 6.13) have been specified, the *sufficient* conditions for elements at the next position are fully defined. It may also help the solving process to add the *necessary* conditions for an element to be at a certain position at the next layer. Any action at the next layer either has already been at the corresponding parent position at the previous layer before, or it has been created by expanding an appropriate reduction at the parent position. Likewise, any reduction at the next layer must have been created by some appropriate reduction before.

6.4 Plan optimization

The T-REX approach as presented finds a solution to the planning problem at hand at the most shallow possible hierarchical depth. However, it does not guarantee any upper bound on the plan length besides the maximum array size at the final layer. As neither the encoding nor the SAT solver differentiate between solutions of varying plan length, a short plan is not inherently more probable than a longer plan. To overcome this drawback, the paradigm of incremental SAT solving can be further exploited to reduce the plan length after identifying an initial plan.

The general procedure of the plan optimization is as follows: When the SAT solver reports satisfiability for the first time, a plan is found. Afterwards, the goal literals which have previously only been assumed (for a single solving attempt) are added permanently, for *all* following solving attempts. Then, new clauses are added:

The variable $\texttt{planLengthGeq}(k, i)$ ("the plan length at position i is greater or equals k") represents that the amount of all "proper" actions up to position i at the final hierarchical layer make up a partial plan of length k or longer. In particular, *blank*and `nop` actions (without any preconditions and effects) are not counted.

The plan length is at least zero at position zero, and the minimal plan length will never decrease, but at least stay the same when advancing to the next position at the final layer:

$$\texttt{planLengthGeq}(0,0) \tag{6.16}$$

$$\texttt{planLengthGeq}(k,i) \implies \texttt{planLengthGeq}(k,i+1) \tag{6.17}$$

If a proper action is at position i at the final layer, then the minimal plan length will increase by one at position $i+1$:

$$\begin{aligned}\texttt{planLengthGeq}(k,i) \wedge \neg\texttt{element}(\texttt{nop},i) \wedge \neg\texttt{element}(\mathit{blank},i)& \\ \implies \texttt{planLengthGeq}(k+1,i+1)&\end{aligned} \tag{6.18}$$

Above clauses together ensure the following: If some found plan has a length of k^* on an array of size A, the variable $\texttt{planLengthGeq}(k^*, A)$ will be `true`. Consequently, to restrict the plan to be shorter than k^*, the following literal is assumed:

$$\neg\texttt{planLengthGeq}(k^*, A) \tag{6.19}$$

A plan of length smaller than k^* can now be searched by assuming such a literal and performing another solving attempt. This technique is combined with a general search method (e.g., a bisection search or a linear search) in order to iteratively improve the lower and upper bounds on possible plan lengths at the final layer of the considered hierarchy, until the computation is canceled or until the plan is fully optimized, i.e. the lower and upper bounds on the plan length collide.

Note that the *globally optimal* solution to the problem is not necessarily found this way. There may still be a plan of shorter length at some deeper layer because additional `nop` and *blank* actions may arise at that point, decreasing the effective plan length despite a larger array size. When accounting for these special actions, the reduction expansions are comparable to shortening production rules of a formal grammar, leading to the assumption that it is indeed a very hard problem to find the globally shortest plan under this HTN model.

The described procedure is an Anytime algorithm which may be interrupted after an arbitrary amount of time, reporting the shortest plan found as the final result.

The complexity of variables and clauses is quadratic in the size of the final hierarchical layer. An experimental demonstration of the proposed plan optimization featuring different search procedures is provided in Chapter 8.5 (Evaluation).

6.5 Analysis

In the following, some theoretical properties of the T-REX approach are discussed.

6.5.1 Termination and correctness

Theorem 2 (Termination and correctness of the T-REX encoding)**.** Consider a solvable HTN problem $\mathcal{P} = (\mathcal{D}, C, s_0, g, T_0)$. Using the T-REX encoding, an incremental SAT solving procedure terminates after at most δ computational steps, and a valid plan can be extracted from the found satisfying assignment. Hereby, δ is equal to the minimal depth of any existing plan for $\mathcal{P}$.

Idea of proof. It can be shown inductively that a T-REX solving procedure perfectly simulates a breadth-first search of the problem's task hierarchy (Def. 18). The remaining argument is simple: If a plan is found within T-REX, then it is a valid part of the problem's hierarchy, implying that the plan is in fact a solution to the problem at hand. If δ is the minimum depth of which a plan to the problem exists, then the procedure will find a plan after δ computational steps. $\square$

6.5.2 Complexity of clauses and variables

To assess the complexity of the T-REX encoding, asymptotic measures over the number of variables and clauses are provided.

Consider a problem for which after k hierarchical layers, each layer i with an array size of s_i, a plan π of length $|\pi| \leq s_k$ is found. Using the sparse fact encoding, each fact has then been encoded as a variable at most s_k times. Each action $a \in A$ is encoded at most $C := \sum_{i=1}^{k} s_i$ times (once for each position at each layer). Similarly, each task reduction $r \in R$ has been encoded at most C times. Adding at most $C \cdot \log(A)$ helper variables for At-Most-One constraints and exactly C primitivity variables leads to a variable complexity of $\mathcal{O}\big(s_k \cdot F + C(R + A)\big)$.

Regarding the needed clauses for finding the mentioned plan, $\mathcal{O}(\sum_{i=1}^{k-1} s_i \cdot (R + A) \cdot e)$ clauses for reductions and the introduction of new *blank* symbols are needed, for $e := \max\big\{|subtasks(d)| \;\big|\; d \in R\big\}$. Preconditions and effects add up to $\mathcal{O}(C(R + A))$ clauses. The definition of the primitiveness variables is included by this complexity measure as well. $\mathcal{O}(C \cdot F)$ clauses for frame axioms and at most $C \cdot A \log(A)$ clauses for binary-style At-Most-One constraints are needed. This leads to a total of $\mathcal{O}\big(\sum_{i=1}^{k-1} s_i \cdot (R + A) \cdot e + C(R + A\log(A) + F)\big)$ clauses.

Note that these complexity measures assume that at each position and at each layer, all actions and all methods occur, which is highly unlikely in practice. Nonetheless, there is no obviously quadratic term in the complexity of clauses and variables, in contrast to the previously presented SMS encoding. Intuitively, this is because instead of encoding a full "matrix" of $k \cdot s_k$ positions, only the actually needed tree of size $\sum_{i=1}^{k} s_i$ is encoded. For instance, consider a problem with 2 initial tasks where each position will always expand to a sequence of 2 new positions. With a hierarchy of depth k where eventually some plan of length $l = 2^k$ is found, a more naïve encoding could yield $k \cdot l$ positions to encode, whereas the tree-like encoding only takes $\sum_{i=1}^{k} 2^i < 2 \cdot l$ positions.

Chapter 7

Implementation

In the following, technical details about the realization of the presented encoding and solving techniques are provided. All software developed over the course of this work will be made available on `https://gitlab.com/domschrei/htn-sat`.

7.1 HTN-SAT

The Java application **HTN-SAT** has been developed in order to encode HTN planning problems in propositional logic, to launch an external SAT solving process on the encoding, and to decode the result and output it as a human-readable sequence of primitive actions.

7.1.1 Abstract description

HTN-SAT uses the framework PDDL4j with an extension developed by Ramoul et al., 2017 which includes an effective grounding procedure as a part of the preprocessing. After receiving a completely grounded HTN planning problem from the preprocessing, HTN-SAT continues to create further simplified data structures which are needed for the various featured HTN-to-SAT encodings. Subsequently, a SAT encoding of the desired type, specified by a command-line argument, is produced. The resulting set of clauses is then output into a file (whose structure depends on the chosen encoding type) and an according solver executable is launched on this file. Such solver applications include compiled standalone SAT solvers (for the non-incremental GCT encoding), the Incplan application by Gocht and Balyo, 2017 linked with an incremental SAT solver (for the SMS encodings), and the T-REX Interpreter application (for the T-REX encoding). Being notified at the end of the computation, HTN-SAT continues to parse the found assignment and decodes the assigned variables back into the original problem domain. Finally, the found plan is output to a solution file.

In the case of non-incremental solving (which is only the case for the GCT encoding), the process of encoding, writing and SAT Solver execution is repeated until the solver has found a solution or the computation is canceled.

7.2 T-REX Interpreter

The **T-REX Interpreter** is an application written in C++ which expects an abstract T-REX encoding file as an input, performs an incremental instantiation and solving process by directly communicating with a SAT solver library, and outputs the found satisfying assignment. Optionally, the application can subsequently launch a plan length optimization procedure which is interruptible at any time.

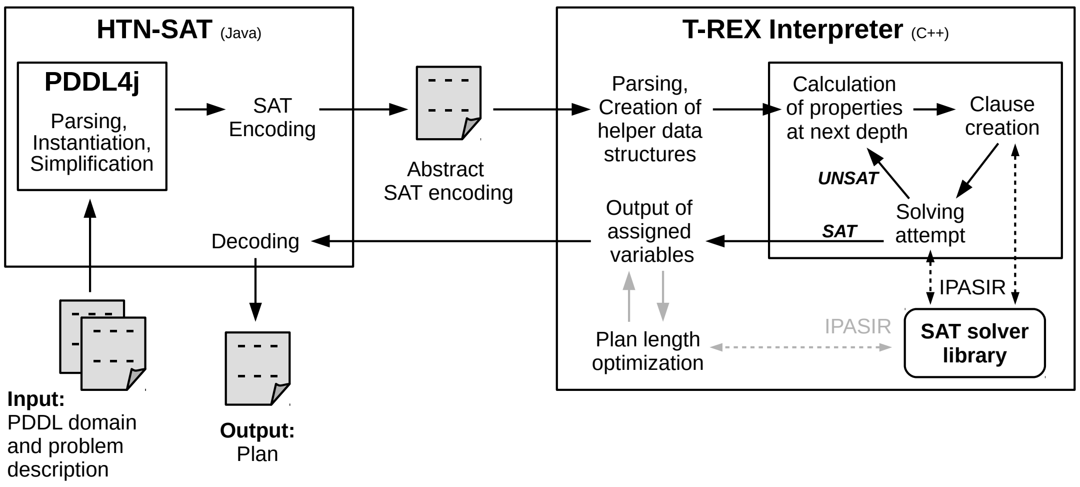
HTN-SAT (Java)
PDDL4j
Parsing, Instantiation, Simplification
SAT Encoding
Decoding
Abstract SAT encoding
Input: PDDL domain and problem description
Output: Plan
T-REX Interpreter (C++)
Parsing, Creation of helper data structures
Calculation of properties at next depth
Clause creation
UNSAT
SAT
Solving attempt
Output of assigned variables
IPASIR
Plan length optimization
SAT solver library

FIGURE 7.1: The general pipeline of solving a planning problem with the T-REX approach

7.2.1 Abstract description

Figure 7.1 illustrates the abstract pipeline of a full execution of the T-REX Interpreter within the scope of HTN-SAT. Initially, the T-REX encoding is read and converted into appropriate data structures. It is worth noting that the parsed abstract clauses are not stored in the form of pure clauses, but rather in specialized data structures which directly hold information on the properties of the encoded elements and facts.

With these data structures, an initial layer of the hierarchy is computed, only containing the initial tasks and facts. All necessary clauses describing this layer are then instantiated and transferred to a SAT solver over the IPASIR (see Chapter 7.2.3). When the SAT solver found the formula to be unsatisfiable, the subsequent hierarchical layer is computed by applying the transitional properties of the problem to the data of the previous layer. This process of computing a hierarchical layer, instantiating the necessary clauses, and executing a SAT Solver is repeated until success (or until the computation is cancelled).

Optionally, the computation continues with a plan length optimization, repeatedly assuming literals and calling the SAT solver until the optimal plan on the final layer is found or until the computation is canceled. In the latter case, the decoding and the output of the best found plan is still performed before terminating.

7.2.2 Encoding format

The abstract encoding notation used for the T-REX approach is an extension of DimSpec (Gocht and Balyo, 2017) implying an instantiation of the included clauses along *two* directions: along the makespan, which is the hierarchical depth, and along the positions of each layer. Like DimSpec, it contains four blocks `i`, `u`, `g`, and `t` with an identical meaning. Additionally, the following *literal annotations* are introduced:

The annotated literal `lit@`i denotes that the literal `lit` holds at position i at the depth which is currently instantiated. This is used for the initial state.

`lit@A` denotes that `lit` holds at *all* positions at the current depth. This is used for the goal assumption of all positions being primitive.

`lit`$+k$ denotes that `lit` holds at position $i + k$, if i is the instantiated position in the context of the literal. This is mainly used for the specification of the reduction expansions.

The variable domains have been fixed in order to allow for an uncomplicated and efficient implementation: variables $\{1, \ldots, A\}$ represent the actions, variables $\{A + 1, \ldots, A + R\}$ represent the reductions, variable $A + R + 1$ denotes the `isPrimitive` predicate, and variables $\{A + R + 2, \ldots, A + R + F + 1\}$ represent the facts of the problem. Information on the quantities A, R, F and on any improper *nop* actions are provided by header lines at the encoding's beginning.

Only the clauses which actually define the problem need to be provided as a part of the encoding; other clauses such as frame axioms, At-Most-One constraints or the propagation of facts and actions along the layers can be inferred and will be added to the incremental formula automatically without the need to explicitly specify them.

7.2.3 Dependencies

The T-REX Interpreter uses the IPASIR (Balyo et al., 2016) which facilitates the communication with a SAT solver. On a technical level, the C++ application is linked with a pre-compiled library of the desired SAT solver. Such a library provides a slim interface of C methods such as `ipasir_add` (to add a literal to a clause), `ipasir_assume` (to assume some literal) or `ipasir_solve` (to initiate a solving attempt). As the interface itself is agnostic to the used solver, any solver for which IPASIR bindings exist can be linked with the T-REX Interpreter. Therefore, various popular SAT solvers such as MiniSAT (Eén and Sörensson, 2003), PicoSAT (Biere, 2008), Glucose (Audemard and Simon, 2009) and Lingeling (Biere, 2013) have all been successfully linked and used with the T-REX Interpreter with minimal effort. Changing the SAT solver used by T-REX just requires re-building the application with the environment variable `IPASIRSOLVER` set accordingly.

For efficiently storing and querying the possible elements at every position at each layer of the hierarchy, the `dynamic_bitset` implementation of the `boost` libraries has been used. Additional `boost` modules have been used for the parsing of command-line arguments.

7.3 Experiments

In the following, the general approach of performing the performance evaluations is described. For reproducibility, the implemented evaluation methods will be made available on `https://gitlab.com/domschrei/htn-sat`.

7.3.1 Methodology

All evaluations have been performed by first creating a textfile which contains all configurations to evaluate, one for each line. Each configuration contains the problem instance (domain and problem file), the executable to solve the problem (such as HTN-SAT or Madagascar), and additional program arguments. When the configuration file is created, a `bash` script randomizes the ordering of the lines. This leads to a sequence of configurations to be tested which is properly randomized in order to reduce any skews in the results caused by external circumstances.

The configuration file is read by a script which then executes one configuration one after the other, using GNU parallel (Tange, 2011). A script named `timeout`[1] cancels any computation which does not terminate after the specified amount of time or which takes more memory than allowed. In every case, all relevant information

[1] `https://github.com/pshved/timeout`

about the program execution is logged into separate directories, also including meta information like memory usage, CPU temperature and MD5 checksums of used binaries.

After the entire configuration file has been processed, various relevant results such as the total run times of the different configurations can be assembled by the usage of additional scripts. The data has then been visualized by Python scripts using `Pyplot` and aggregated into illustrative tables.

7.3.2 Validation

In order to guarantee that all results used for the evaluation are based on correct calculations, the VAL validation tool (Howey, Long, and Fox, 2004) has been used. In an automatic manner, all found solutions of an evaluation can be examined by VAL and any errors in the plan will be reported. As VAL is restricted to the validation of classical planning problems, the plan is only validated in the sense that the set of facts is consistent at each step and correctly transforms the initial state to some goal state. The definition of a plan as given in Def. 15 is stronger and also incorporates abiding to the HTN-specific constraints of the problem.

7.3.3 Hardware and software set-up

The T-REX parameter tuning and the comparison of T-REX with Madagascar have been done on a server with 24 cores of Intel Xeon CPU E5-2630 clocked at 2.30 GHz and with 264 GB of RAM, running Ubuntu 14.04 with the Linux kernel 3.13.0-142-generic.

All other evaluations have been conducted on a notebook with an Intel i7-7500U dual-core CPU clocked at 2.70 GHz and with 16 GB of RAM, running GNU/Linux Debian 9 with the Linux kernel 4.16.0-1-amd64.

All experiments have been done in a head-less, non-graphical terminal session. The server has been accessed via SSH.

For all experiments, Glucose (Audemard and Simon, 2009) has been used as the primary SAT solver, as it generally performed well on preliminary tests compared to the other solvers.

7.3.4 T-REX parameter tuning with ParamILS

For the parameter tuning of T-REX, the default configuration of ParamILS v2.3.5 has been used (Hutter et al., 2009), namely the *FocusedILS* approach with default parameters. No general computational bounds were provided, and the cutoff time per instance was set to 3 minutes. The total run time average has been chosen as the quality metric for individual configurations.

Chapter 8

Evaluation

In the following, the original encoding approaches proposed in the previous chapters are experimentally evaluated and compared to other state-of-the-art methods.

8.1 Parameter tuning of T-REX

Various encoding optimizations have been developed for the T-REX encoding. In order to validate these and to find a configuration which performs well on a broad range of common problem domains, an automated parameter tuning has been conducted. The used tuning framework ParamILS (Hutter et al., 2009) has previously successfully been used by a wide range of applications, in particular for SAT solving (Hutter et al., 2007) and automated planning (Alhossaini and Beck, 2012).

8.1.1 Considered parameters

The following set of parameters for the T-REX encoding has been considered:

`init_layer_amo` (a): At the initial layer, add At-Most-One constraints over each set of possible task reductions.

`full_amo` (A): At all non-initial layers, add At-Most-One constraints over *all* occurring elements, also over reductions.

`binary_amo_threshold` (b): Sets the minimal amount of elements for At-Most-One constraints in order to use a binary At-Most-One encoding instead of the pairwise variant (see Appendix A).

`cumulative_amo` (c): If action variables are reused, also reuse the set of At-Most-One helper variables from the previous depth and position, leading to a single cumulative set of At-Most-One constraints for each "final" position.

`full_prim_elem_encoding` (e): Do not reuse action variables.

`full_fact_encoding` (f): Do not use the strategy of sparse fact encoding and variable reusage.

`encode_predecessors` (p): Encode the possible origins of each element regarding its previous layer.

As an initial configuration of T-REX, the encoding `T-REX-b4` has been used. The value 4 for the threshold `b` has been chosen as it strongly enforces compact, non-quadratic constraints, leading to good run times in previous explorative experiments.

The parameter tuning has been conducted using 139 problems from seven problem domains which originate from the International Planning Competition (IPC).

8.1.2 Results

After running ParamILS for 100 000 seconds (around 27 $\frac{3}{4}$ hours), the configuration `T-REX-p-b8192` has been found as a stable candidate over multiple iterations for the best configuration regarding the average run times. This encoding configuration has been evaluated on all considered problem instances and lead to an average run time of 34.05 seconds, which is only slightly better than the next best found configuration `T-REX-pa-b512` with an average run time of 34.65 over all considered instances.

The high threshold values which have been found imply that the fast propagation properties and the absence of any additional variables in the pairwise At-Most-One style outweigh the disadvantage of requiring a quadratic amount of clauses. Note that 8192 has been the highest value of `b` that has been considered during the tuning process. An even higher value, or a complete omission of binary-encoded At-Most-One constraints, might have further influence on the run times.

The optimizations regarding the sparse encoding of elements and facts have been confirmed to positively contribute to the approach's performance. Likewise, the addition of redundant clauses specifying the possible predecessors of each element (`p`) has been found to consistently help the solver to find an answer more quickly. However, note that the results of the tuning cannot be used to argument with certainty that this configuration is indeed the globally optimal configuration on the considered instances.

The resulting speedup of the found configuration over the initially considered "naïve" version of T-REX is shown during the following evaluations.

8.2 Benchmarks of GCT, SMS, and T-REX

In the following, the different encodings proposed over the course of this work are experimentally compared to one another. Namely, the competitors are the GCT encoding (Chapter 4), three variants of the SMS encoding (Chapter 5), and both the most naïve (-ef-b4) and the tuned (-p-b8192) configurations of the T-REX approach (Chapter 6) without any plan optimization. The three featured SMS variants are described in Chapter 5.2.5.

The evaluation has been focused on a selection of problem domains which include some very simple instances but become considerably harder when proceeding to the more complex instances. The domains Blocksworld and Elevator lead to a modest amount of tasks and actions to be encoded while Rover and Depots are more voluminous domains in that regard. 80 problem instances from these four domains have been considered. Runs have been cut off as soon as they reached a total execution time of five minutes or an effective RAM usage of 12 GB.

8.2.1 Results

A compact overview over the results is given in Fig. 8.1, where the amount of solved instances of an encoding is displayed with respect to the admitted time limit per instance. GCT, as the overall weakest encoding, resolved 12 out of 80 instances within a time limit of 5 minutes per instance. The SMS variants performed considerably better and are quite similar concerning their overall performance; the reduction-based variant SMS-ur solved 23 instances whereas the other two variants both solved 27 instances. Finally, both T-REX configurations significantly outperform the previous encodings, whereas T-REX-ef-b4 solved 70 and T-REX-p-b8192 solved 74 out of 80 instances, all within very reasonable execution times.

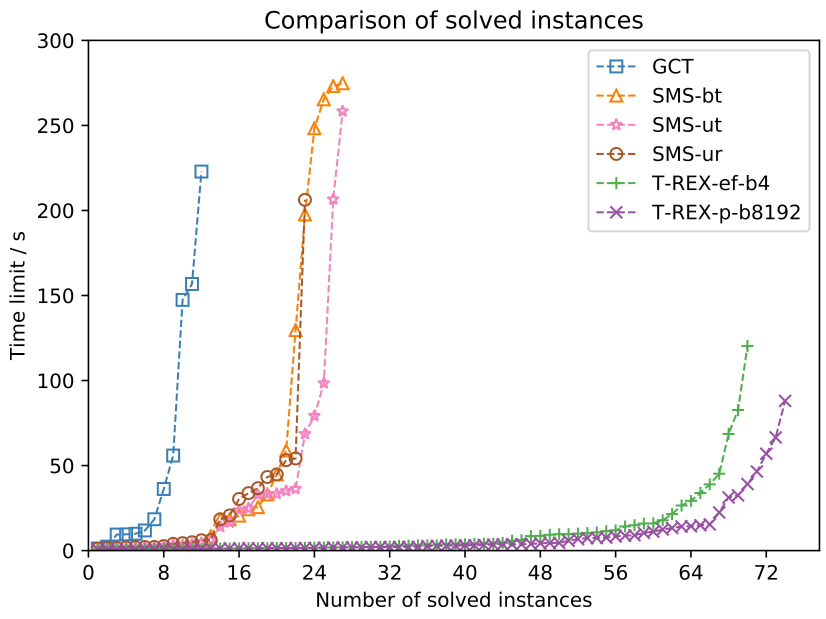

FIGURE 8.1: Comparison of the proposed encodings, with instances from the domains Blocksworld, Depots, Elevator, and Rover

Domain	GCT	SMS-bt	SMS-ur	SMS-ut	T-REX-ef-b4	T-REX-p-b8192
Blocksworld	0.10	2.61	1.80	2.20	17.52	**19.73**
Depots	0.03	1.24	1.07	0.77	18.92	**19.48**
Elevator	1.02	2.62	1.28	2.19	7.43	**13.94**
Rover	0.24	2.21	1.59	1.60	15.20	**19.89**
Total	1.38	8.68	5.73	6.76	59.07	**73.04**

TABLE 8.1: Total run time scores of the proposed encodings

Table 8.1 provides a more detailed insight on the encodings' performances. For each considered instance, a score value of 1 is attributed to the fastest competitor with a run time t^*, and each other competitor with a run time t is attributed a score value of t^*/t, or zero if no solution has been found. These score values are summed up for each domain and competitor, leading to the displayed results.

When comparing the scores of the SMS encodings, it is remarkable that SMS-bt is the best variant regarding the overall score, although SMS-ut seems to perform better judging from Fig. 8.1. This is because the scores values weight each instance equally, and SMS-bt outperformed SMS-ut over most of the easier instances; however, for more complex problems, SMS-ut performed significantly better. SMS-ur performed slightly worse than the other SMS variants on Blocksworld and Elevator, but was competitive in the more complex domains.

The tuned T-REX variant leads to an improvement of 23.6% of the total score over the naïve variant on the four domains. In particular, large problem instances are solved significantly better with the tuned variant, as it found solutions to multiple problems which have not been solved by the naïve variant.

Table 8.2 shows the scores for the found plan lengths of each competitor on each domain. By design of the encoding, GCT always found the shortest plan beneath all competitors *if* it found any solution at all. The plan length differences between the SMS variants are small, and the worse scores for SMS-ur are due to the smaller number of instances it solved. Interestingly, at an average plan length of around 23

Domain	GCT	SMS-bt	SMS-ur	SMS-ut	T-REX-ef-b4	T-REX-p-b8192
Blocksworld	3.00	11.00	10.00	11.00	**19.32**	19.09
Depots	1.00	5.61	4.68	6.88	19.02	**19.47**
Elevator	5.00	6.00	4.00	5.00	11.80	**13.88**
Rover	3.00	3.50	3.50	3.47	16.29	**18.78**
Total	12.00	26.11	22.18	26.35	66.42	**71.23**

TABLE 8.2: Total plan length scores of the proposed encodings

actions over all SMS encodings, the incremental SAT solving procedure took an average of 62 iterations to find these plans. This demonstrates the disadvantage of SMS requiring significantly more iterations than the found plan length. In comparison, the T-REX approach only required an average of 4.8 iterations and never more than 11 iterations, confirming that the minimal plan depth is indeed very small for most of the considered problems and that this benefits the T-REX procedure. However, T-REX may sometimes find a plan which is significantly longer than the minimal plan under the given constraints. This drawback is the reason why an additional plan optimization stage has been developed for T-REX.

Full visualizations of the domain-dependent execution times and plan lengths of each encoding are provided in Fig. B.1 and B.2 in the Appendix.

8.3 Benchmarks of T-REX vs. Madagascar

In the following, the tuned T-REX variant is experimentally compared to the state-of-the-art SAT planner Madagascar (Rintanen, 2014) operating on classical planning, so without any HTN information. The hypothesis to support is that the proposed approach is sometimes able to outperform state-of-the-art classical SAT planning, generally validating the use of HTN-based planning over classical planning when employing SAT techniques.

In total, 180 instances from nine IPC domains have been prepared both with and without HTN-related information. Again, a cutoff time of five minutes and a maximum RAM usage of 12 GB have been specified.

The competitors are `T-REX-p-b8192`, the three default variants of Madagascar (`M`, `Mp`, and `MpC`) with differing heuristics and makespan scheduling, and additionally a modification of Madagascar (`MpC_Incplan`) where the internal SAT solver has been replaced by Incplan, a generic incremental SAT solving procedure making use of the DimSpec format (Balyo et al., 2016), which has been shown to sometimes outperform the default Madagascar versions. Each of the default Madagascar versions been used with their default parametrization, and the Incplan modification has been executed with the parametrization used in the according publication (`--icaps2017` flag).

8.3.1 Results

Figure 8.2 visualizes the run time dependent amount of solved instances for each of the competitors. It can be seen that T-REX was able to solve a number of instances comparable to the Madagascar configurations under the given time and memory constraints. T-REX solved a total of 149 instances each in under two minutes while MpC was able to solve 145 instances in total.

Table 8.4 shows the run time scores for each of the competitors, weighting each instance equally. T-REX achieves a significantly lower total score, which is partly due to the overhead of comparably heavyweight HTN preprocessing routines and of the

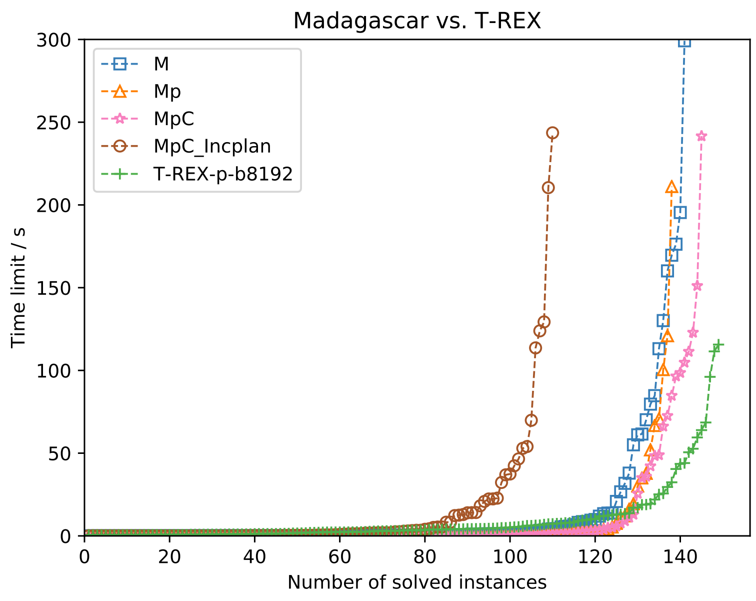

FIGURE 8.2: Comparison of T-REX with various Madagascar configurations, using 180 instances from nine domains

Domain	M	Mp	MpC	MpC_I	T-REX
Barman	0	0	4	0	**20**
Blocksworld	10	11	15	13	**20**
Childsnack	19	9	8	16	**20**
Depots	**20**	**20**	**20**	**20**	**20**
Elevator	**20**	**20**	**20**	13	14
Gripper	**20**	**20**	**20**	6	6
Rover	**20**	**20**	**20**	18	**20**
Satellite	15	**18**	**18**	8	14
Zenotravel	17	**20**	**20**	16	15
Total	141	138	145	110	**149**

TABLE 8.3: Amount of solved instances per competitor and domain

Domain	M	Mp	MpC	MpC_I	T-REX
Barman	0.00	0.00	0.17	0.00	**20.00**
Blocksworld	5.11	5.00	9.82	4.76	**14.24**
Childsnack	8.44	7.75	6.99	5.58	**10.62**
Depots	13.10	18.93	**19.20**	14.53	1.66
Elevator	15.94	**19.61**	17.78	4.46	0.86
Gripper	8.88	**19.42**	18.84	1.75	0.30
Rover	18.35	18.98	**19.55**	15.59	5.16
Satellite	9.49	**17.83**	16.89	3.18	2.83
Zenotravel	12.19	**19.83**	19.11	11.18	0.75
Total	91.50	127.35	**128.34**	61.03	56.42

TABLE 8.4: Total run time scores on each of the nine domains

Domain	M	Mp	MpC	T-REX
Barman	0.00	0.00	2.29	**20.00**
Blocksworld	9.76	7.32	8.86	**19.27**
Childsnack	**16.70**	8.84	7.88	15.75
Depots	12.22	16.32	16.38	**19.79**
Elevator	18.04	**18.99**	18.81	13.91
Gripper	18.04	**20.00**	**20.00**	6.00
Rover	6.03	19.79	**19.99**	8.89
Satellite	9.99	17.32	**17.80**	10.41
Zenotravel	11.75	18.61	**18.60**	6.46
Total	102.53	127.19	**130.60**	120.47

TABLE 8.5: Total plan length scores on each of the nine domains

T-REX pipeline leading to much higher runtimes for easy problems if compared to the straight-forward preprocessing and solving routines of Madagascar.

In order to explain the large differences between the considered domains, three different groups among them can be identified as follows:

- **Few action parallelization and/or easy hierarchy models.** There are some instances which are problematic for Madagascar, namely problems which are hardly parallelizable regarding ∃-step semantics. By design, the domains Barman and Blocksworld do not allow any parallel actions, stripping Madagascar of its strongest advantage over T-REX. Using the hierarchical information, T-REX easily solved each Barman instance (from the benchmarks used by Ramoul et al., 2017) while the Madagascar configurations only solved few instances. The domain Childsnack is modeled with an extremely simple hierarchy, taking a fixed makespan of 1 for T-REX to solve each instance. Madagascar does not have this information, leading to a very large search space.

- **Problematic hierarchy models.** T-REX performed comparably poor at the instances of Elevator, Gripper, and Zenotravel, which have one evident common point: some HTN methods are modeled resembling a loop without an explicit break condition. For instance, the Elevator domain contains a method `check_floor` which may be reduced either to some sequence of tasks followed by a recursive call of `check_floor` itself, or to a single `nop`. The planner itself has to decide when to exit this loop. T-REX has difficulties handling this kind of hierarchical structure: while the procedure quickly finds a satisfying assignment at the final makespan, it takes large amounts of time to prove the unsatisfiability of non-final makespans.

- **Hard domains with reasonable hierarchy models.** The remaining domains, namely Depots, Rover, and Satellite, have well-designed HTN models; however, the instances have a comparably high complexity (regarding the amount of tasks and actions after grounding). The Madagascar variants outperform T-REX on these domains, possibly due to the latter's computational overhead and due to the domains being well parallelizable by ∃-step semantics.

It can be concluded that, in comparison with previous SAT planning approaches, T-REX is especially strong on single-agent planning domains where the problem semantics does not allow for the parallel execution of actions and where no unfavorable loop-like structures are featured in the HTN model.

Regarding the produced plan lengths (Table 8.5), T-REX proved to create overall competitive plans. Evidently, the plan lengths highly depend on the domain as well: For instance, the HTN model for Childsnack enforces plans that are quite long compared to the minimum possible plan when not considering the HTN constraints. In contrast, the HTN model for Gripper always leads to the shortest possible plan (but T-REX only solved few instances from this domain). T-REX found comparably long plans in the Rover, Satellite and Zenotravel domains whereas it found the generally shortest plans in the Depots domain and plans of average length on the remaining domains. As a conclusion, T-REX is competitive regarding the produced plan lengths as well, even moreso if the plan optimization technique is employed (see Chapter 8.5).

Note that, for the considered set of problem instances and with Glucose as a SAT solver backend, the Incplan-patched variant of Madagascar with default parameters was not able to outperform the original variants.

More detailed visualizations of the execution times and plan lengths are provided in Fig. B.3 and B.4 in the Appendix.

8.4 Comparing T-REX to conventional HTN planning

In the following, T-REX is compared to GTOHP (Ramoul et al., 2017), a state-of-the-art HTN planner which operates on the same problem inputs and uses the same preprocessing routines.

Regarding run times, T-REX is currently not able to outperform GTOHP. On identical problem instances (instances 01-20 each from Barman, Childsnack, Rover, and Satellite), GTOHP solved each instance in well under a minute and almost all instances in the matter of a few seconds. In contrast, T-REX takes multiple minutes for some more difficult problems on comparable hardware. One of the reasons for this performance difference is the high computational overhead of the implemented SAT-based approach. Specifically, the problem has to be transformed into efficient data structures, encoded into SAT, written into a formula, solved by the interpreter application, and decoded back to a result, whereas GTOHP directly operates on the results of the initial preprocessing. In addition, the exploration of the solution space is done in a much more guided manner by GTOHP, greedily applying reductions in a depth-first manner. T-REX still needs to enumerate and process all possible task reductions that may occur at some depth of the hierarchy.

The advantages of T-REX over GTOHP include the plan length optimization of T-REX, leading to shorter plans the more resources are invested. T-REX may also be used to prove certain properties of an HTN problem, such as the minimal hierarchical depth for which there exists any solution, or tight plan length bounds for a certain layer of the problem's hierarchy.

Additionally, T-REX is a robust approach which operates reasonably well on domains which are missing method preconditions. As shown in Fig. 8.3, the tuned T-REX variant has been tested on the Rover domain both with the usual HTN model and with all method preconditions stripped from the model. As can be seen, T-REX finds plans surprisingly well without any method preconditions, only being slightly slower than on the original domain. In contrast, GTOHP greedily chooses some applicable method at each point of the computation, enters an infinite loop in this case and does not find any solutions on the modified domain.

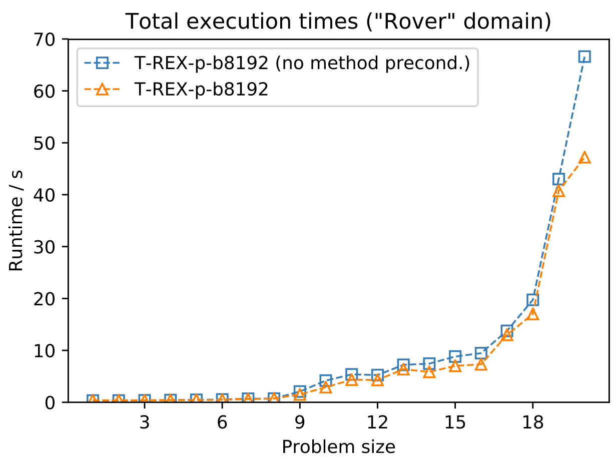

FIGURE 8.3: Comparison of T-REX on the Rover domain with and without method preconditions

8.5 T-REX Plan optimization

In the following, a qualitative demonstration of the plan length optimization stage of T-REX is provided and discussed on some exemplary instances.

A total of 15 exemplary instances from the three domains Barman, Childsnack, and Rover have been chosen, representing the full range of complexities for each domain. These domains represent the most relevant optimization scenarios: Childsnack is not optimizable at all, as it features a trivial hierarchy and will always directly lead to a solution of fixed length. Plans from the Barman domain are optimizable by a slight margin, and Rover plans are highly optimizable.

Two search strategies have been considered: The first strategy is a linear search over the possible plan lengths, beginning with the found plan length and then descending until an unsatisfiable plan length is reached. The step size is at least one, but the actual plan length found as the previous result is considered as well for the new plan length to test. The second strategy is a bisection search over the possible plan lengths, beginning with 1 and the initial plan length as the lower and upper bounds and then always testing the plan length corresponding to the mean of the two bounds.

Each run has been cut off after five minutes, reporting the shortest found plan.

8.5.1 Results

Both strategies terminated regularly for 14 out of the 15 considered instances, i.e. both strategies found a fully optimized plan for almost all considered instances within the time limit. Overall, the linear strategy was slightly faster to find the final plans than the bisection search. The results per instance are visualized in Fig B.5 in the Appendix.

In Figure 8.4, some illustrative examples of the results are provided. The progression of the plan length is illustrated by triangles showing each SAT solver result at some time for some forbidden plan length, and corresponding dashed lines indicate the best upper and lower bounds on the possible plan length which have been found so far.

At the Barman instance, it can be seen that both the linearly descending and the bisection search terminate reasonably fast with a fully optimized plan. Generally, it

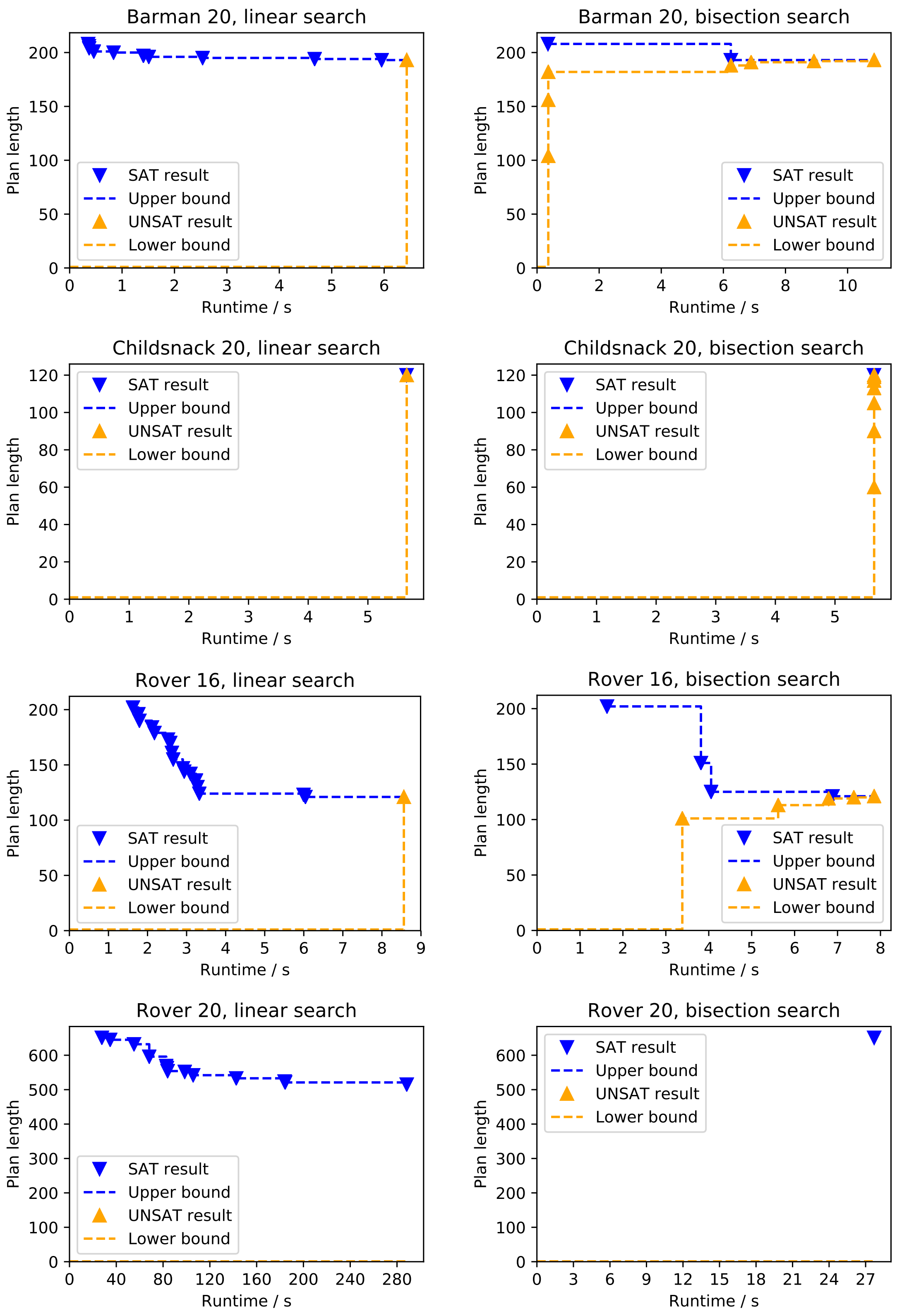

FIGURE 8.4: Illustration of the T-REX plan optimization process, with instances each of Barman, Childsnack and Rover, using a linear descent (left) and a bisection search (right)

is easy to see that the bisection search alternates between satisfying and unsatisfying SAT results whereas the linear search leads to exactly one unsatisfying result which concludes the computation.

The Childsnack domain has no potential for optimization under the chosen HTN model, and both search approaches were able to find this property virtually instantly (after finding the initial plan) on all considered instances.

Figure 8.4 also shows two instances from the Rover domain. Instance 16 has been quickly optimized by both strategies, and it demonstrates well the large margin by which a plan may be optimized (from 202 to 121 actions in this case).

For instance 20, none of the considered optimization strategies terminated. After finding an initial plan with 651 actions, the bisection search was not able to answer the very first test whether there is a plan of length shorter than $(651 + 1)/2 = 326$. As a consequence, this run terminated with the initial plan as the shortest found plan. In contrast, the linear search was able to find a couple of modest optimizations, with the shortest found plan at the cutoff time containing 515 actions.

The demonstrated examples illustrate that both search strategies can be employed in order to optimize the plan length of the T-REX approach. The linearly descending approach tends to lead to a more immediate payoff, and no resources are wasted by unnecessarily proving some unsatisfiability. The bisection search needs much less SAT solver calls, but the performed plan length tests are comparably hard to answer, leading to a more significant computational investment until some high quality plan is found. Overall, because of its fast payoff and slightly faster run times, the linear search tends to be more favorable compared to the binary search strategy on the considered problem instances.

8.6 Conclusion of Evaluation

By the evaluations presented in this chapter, the three proposed HTN-to-SAT encodings have been experimentally compared to one another as well as to state-of-the-art competition.

The incremental SMS encoding significantly outperforms the GCT encoding, which itself is based on the last published HTN-to-SAT encoding effort. Furthermore, the T-REX encoding and instantiation approach significantly outperforms SMS, and successfully competes with state-of-the-art classical SAT planning. The practical performances of both T-REX and Madagascar are highly dependent on the problem domain and, in the case of T-REX, on the modeling of HTN information.

T-REX has been demonstrated to be a practically useful approach due to its high efficiency, its additional optimizations, and its integrated plan optimization techniques. Currently, it does not achieve the run times of the conventional HTN planner GTOHP. Yet, an argument for the viability of T-REX can be made due to its plan optimization, its theoretical proving abilities, and a high robustness towards missing preconditions.

In total, the evaluations have yielded encouraging results for SAT planning on HTN domains, and they demonstrate that the contributions of this work successfully redefine the state-of-the-art in this field by a significant margin.

Chapter 9

Conclusion and Outlook

In this chapter, a conclusion on the presented work is provided and an outlook on potential future work is given.

9.1 Conclusion

The presented work has focused on solving Hierarchical Task Network planning problems using SAT solving techniques.

A number of previously proposed HTN-to-SAT encoding approaches have been discussed regarding their viability from today's point of view. An initial new encoding Grammar-Constrained Tasks (GCT) has been proposed, mending the central shortcomings of the previous encodings and being able to exploit modern HTN preprocessing.

In order to avoid the high complexity and the generally slow solving process of GCT encodings, a second original encoding called Stack-Machine Simulation (SMS) has been specifically designed for incremental SAT solving. It works reliably in all considered special cases.

However, as the SMS encoding still takes too many computational steps to find a solution, a technique has been developed to significantly reduce the amount of required computational steps, resulting in a third original encoding named Tree-like Reduction Exploration (T-REX). A specialized interpreter application has been developed to take abstract encoding files tailored to the problem and to instantiate all necessary clauses just as needed. Various encoding optimizations have been proposed in order to reduce the amount of variables and clauses and to speed up the solving procedure. Additionally, a plan length optimization stage within T-REX has been presented to reduce the plan length in an Anytime manner after an initial solution has been found.

Thorough evaluations have been conducted. A T-REX encoding variant which works particularly well in practice has been identified using the ParamILS tool, also validating the usefulness of the employed encoding optimizations. In a direct run time comparison featuring popular problem domains, SMS significantly outperformed GCT, and T-REX significantly outperformed SMS. Afterwards, T-REX has been compared to the state-of-the-art classical SAT planner Madagascar. The performance differences of the competitors have been discussed while taking into account the individual properties of each domain. T-REX outperformed Madagascar on some domains, which validates the use of the proposed SAT-based HTN technique over classical SAT planning.

T-REX did not achieve run times on par with the state-of-the-art HTN planner GTOHP, but can still be a viable alternative which offers an efficient optimization of the plan length as well as the ability to prove certain problem properties. Additionally, T-REX has been shown to solve problems robustly even when preconditions of methods are partly or fully missing and to reliably find solutions in such cases, whereas GTOHP does not.

To conclude, the contributions of this work are three original SAT encodings for HTN planning problems, each setting a new baseline for state-of-the-art SAT planning on HTN domains, and their practical implementation and evaluation.

By the theoretical design and the efficient implementation of the T-REX encoding and instantiation approach, it is shown that HTN planning via SAT solving is a viable option if engineered carefully. Based on the evaluation results, I suspect that it is generally difficult to achieve run times of conventional HTN planners with a SAT-based approach. However, numerous merits of the developed SAT-based technique have been identified, and the topic is definitely worthy of the attention of future research efforts — maybe to eventually outperform conventional HTN planning in some cases.

9.2 Outlook

In the following, an outlook towards possible future work is provided.

Specialized preprocessing

In order to optimize the process prior to the actual solving stage, an alternative preprocessing may be developed which is specialized to the T-REX approach. This can potentially lead to an overall more lightweight preprocessing and may allow to interleave the encoding process with the grounding procedure, further optimizing the total run time.

Extension of the model

The current T-REX approach features a total order on all tasks and subtasks, and it does not support *between* and *after* constraints for a method's expansion definition. By extending the preprocessing procedures and the T-REX approach by such additional constraints, the approach may become more versatile and more efficient. Likewise, introducing a T-REX variant which operates on a *partial* ordering of subtasks would make the approach more viable for further practical use cases. For such a scenario, further research can be done regarding the introduction of ∃-step semantics (Rintanen, Heljanko, and Niemelä, 2004) for multiple tasks and/or actions to co-occur at the same positions in an encoding based on T-REX.

Investigation of alternative SAT techniques

For further research, specialized SAT solver configurations may be investigated which are particularly well-fit to solve the formula structure as created by T-REX. In order to guide the solver to find solutions as fast as possible, specialized heuristics may be employed, and further SAT techniques such as randomized SAT solving can be explored to intelligently decide on the set of reductions to apply. In addition, alternative makespan scheduling strategies may be investigated instead of the current one-by-one increase of the makespan of T-REX. A significant speedup may be realized by combining an advanced makespan scheduling strategy with the execution of multiple SAT solver instances in parallel.

Appendix A

At-Most-One Encodings

Given a set $V := \{v_1, \ldots, v_n\}$ of variables, a common objective is to find a set of clauses which enforce that *at most one* of the variables is `true`.

The naïve method of achieving this is to add a simple NAND constraint for each possible pair of variables:

$$\forall i \neq j : \neg v_i \vee \neg v_j$$

Evidently, this leads to $\frac{1}{2}n(n-1) \in \mathcal{O}(n^2)$ clauses, so the amount of needed clauses may become huge for large n. However, this pairwise At-Most-One style does not require any additional variables, and the mutual exclusion property will be propagated instantly by any CDCL-based SAT solver: if some v_i is set to `true`, then all other variables in V instantly become `false` by unit propagation.

A second well-known method considered here is an approach based on the binary representation of i for any v_i. By adding $\log(n)$ helper variables $B := \{b_1, \ldots, b_{\log(n)}\}$, one for each binary digit of n, the following clauses can be introduced:

$$\forall i : \forall k \in \{1, \ldots, \log(n)\} : v_i \implies l_k,$$

where $l_k = b_k$ if the k-th bit of i is one, and $l_k = \neg b_k$ if the k-th bit of i is zero. As each b_k always represents *either* a one *or* a zero, there can never be multiple numbers represented by the bits B. Consequently, at most one of the v_i may be `true`.

This encoding only needs $\mathcal{O}(n \log n)$ clauses, and thus scales better for large n regarding the encoding size. However, it introduces additional variables to the problem and it does not propagate quite as directly as the pairwise method.

Appendix B

Supplementary Evaluation Graphs

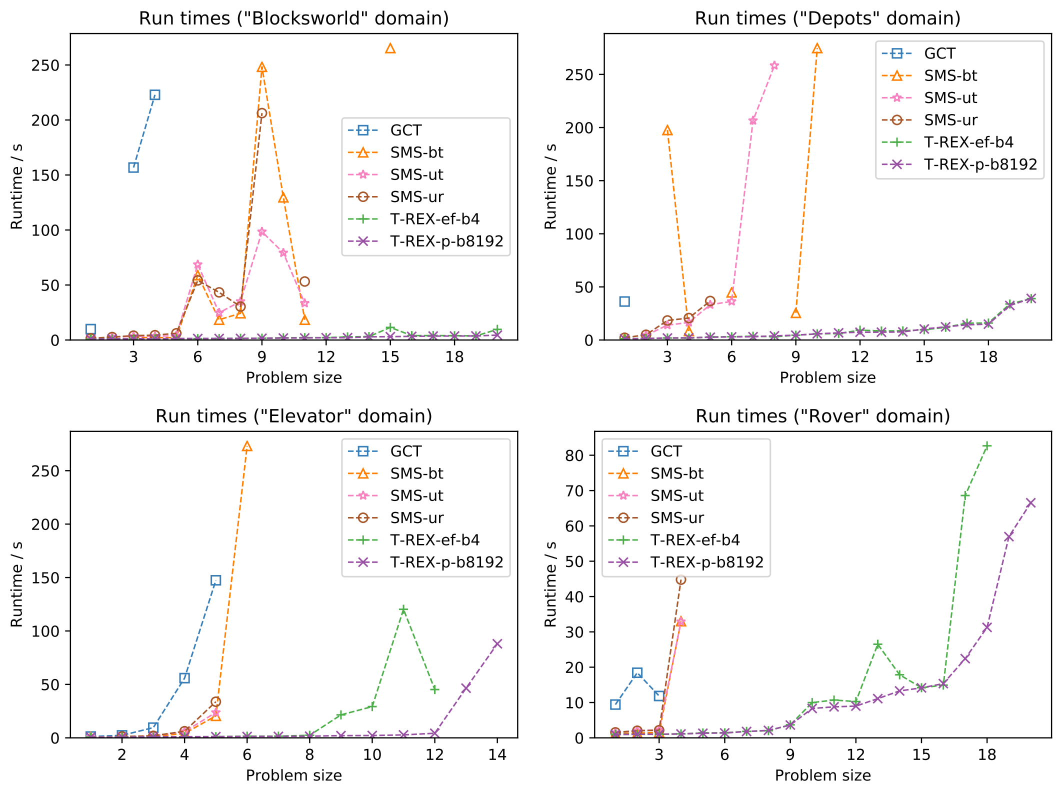

FIGURE B.1: Domain-dependent run times of compared encodings, with the instances ordered by the performance of T-REX-p-b8192

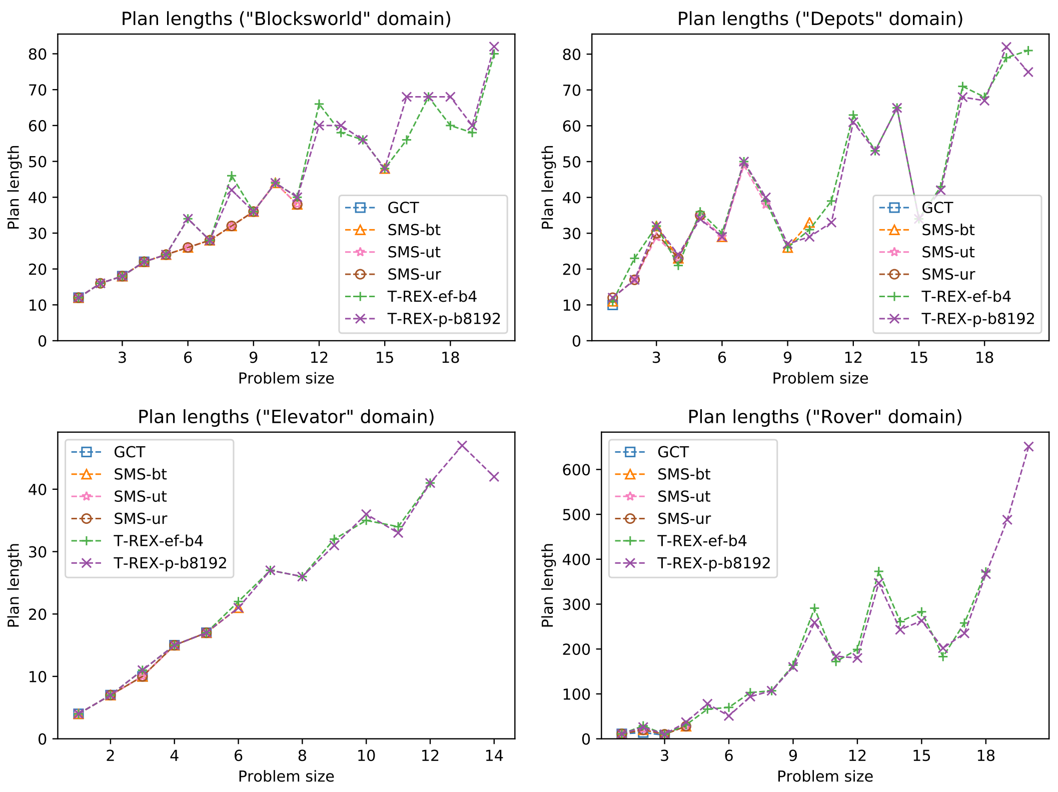

FIGURE B.2: Domain-dependent plan lengths found with compared encodings, again ordered by the performance of T-REX-p-b8192

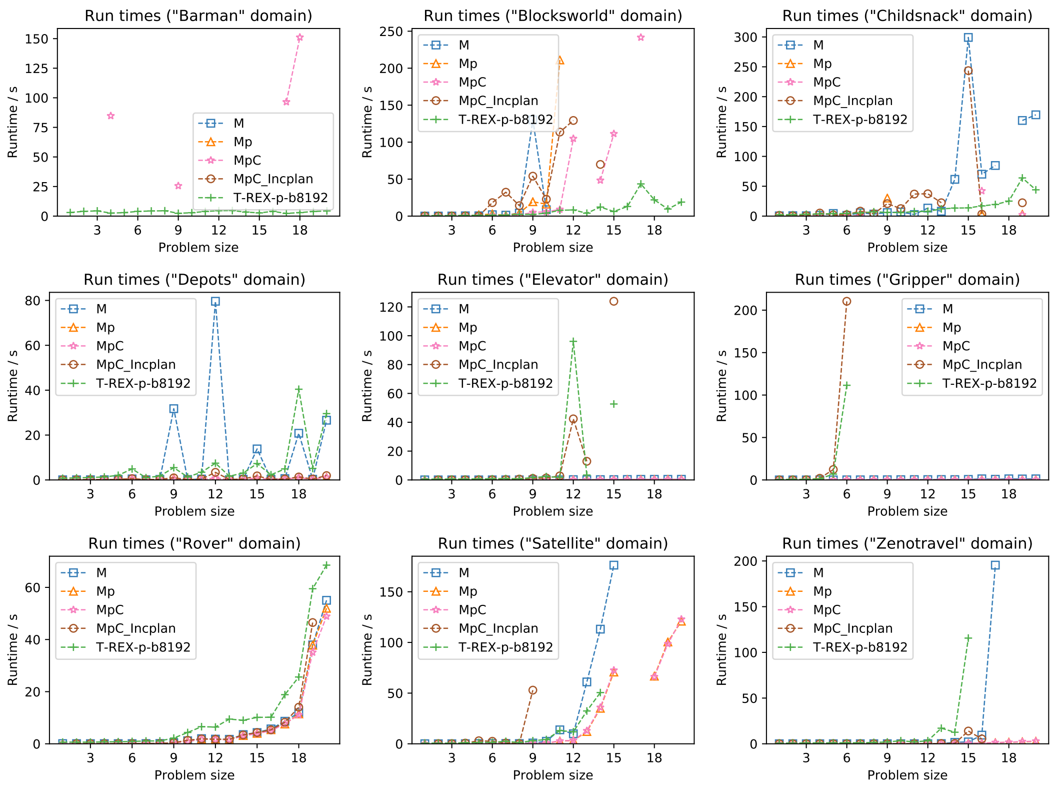

FIGURE B.3: Domain-dependent run times of tuned T-REX and Madagascar variants

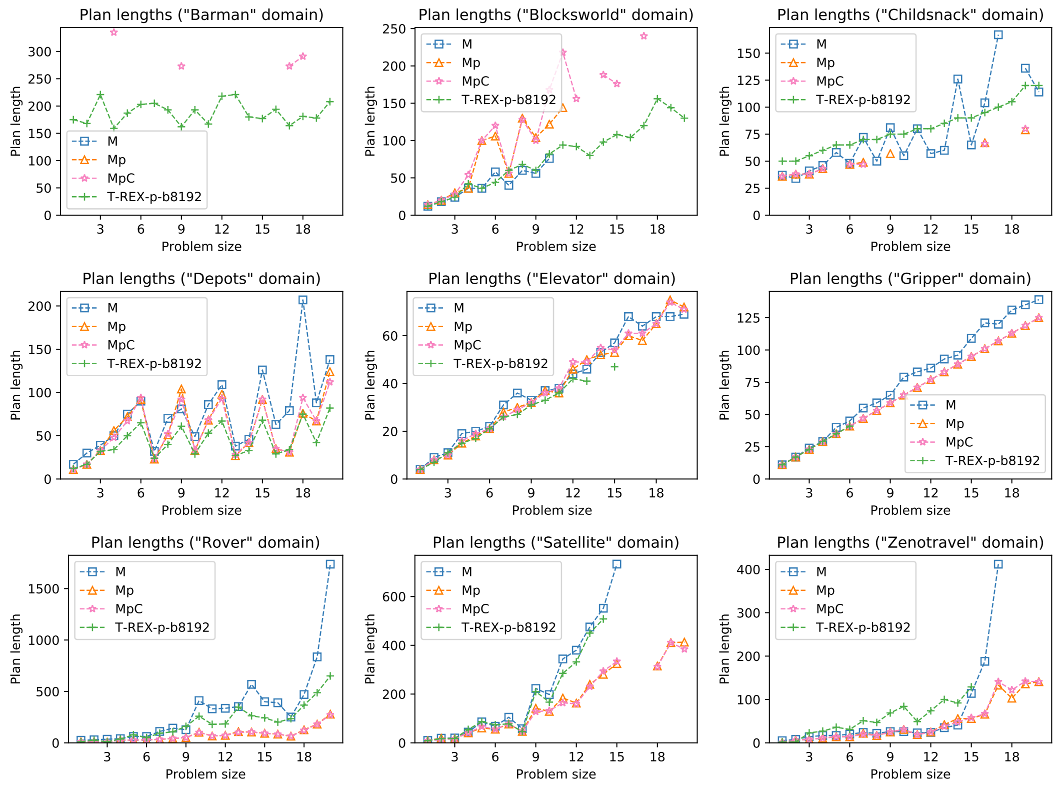

FIGURE B.4: Domain-dependent plan lengths found with tuned T-REX and Madagascar variants (excluding Incplan-patched variant)

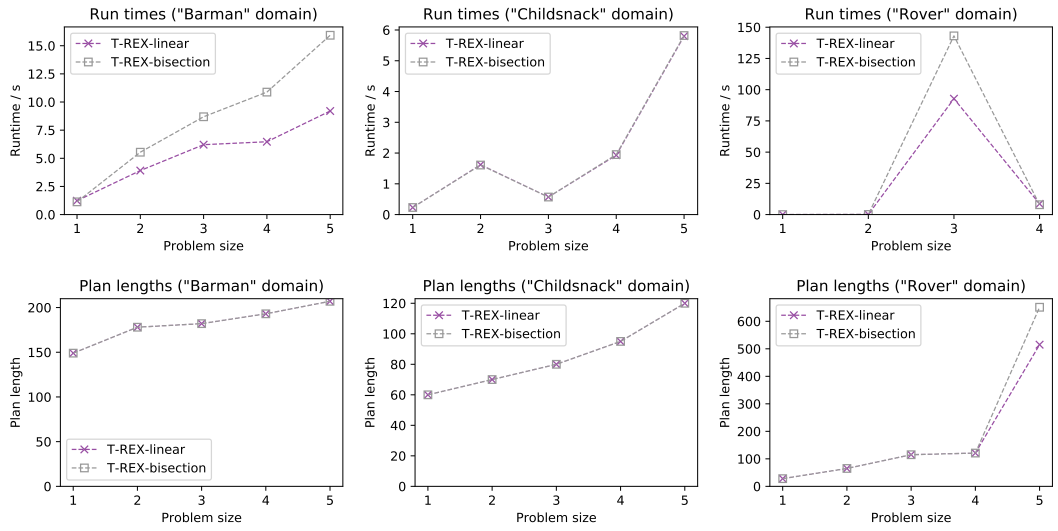

FIGURE B.5: Run times and found plan lengths of two search strategies of T-REX plan length optimization, on five problem instances per domain (instances 4, 8, 12, 16, 20 each, ordered by plan length)

Bibliography

Alhossaini, Maher and J.Christopher Beck (2012). “Macro Learning in Planning as Parameter Configuration”. In: *Advances in Artificial Intelligence*. Vol. 7310. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 13–24.

Audemard, Gilles and Laurent Simon (2009). “Predicting Learnt Clauses Quality in Modern SAT Solvers.” In: *IJCAI*. Vol. 9, pp. 399–404.

Balyo, Tomáš et al. (2016). “SAT race 2015”. In: *Artificial Intelligence* 241, pp. 45–65.

Bayardo Jr, Roberto J and Robert Schrag (1997). “Using CSP look-back techniques to solve real-world SAT instances”. In: *Aaai/iaai*, pp. 203–208.

Bevacqua, Giuseppe et al. (2015). “Mixed-Initiative Planning and Execution for Multiple Drones in Search and Rescue Missions.” In: *ICAPS*, pp. 315–323.

Biere, Armin (2008). “PicoSAT essentials”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 4, pp. 75–97.

— (2013). “Lingeling, Plingeling and Treengeling entering the SAT competition 2013”. In: *Proceedings of SAT competition* 51.

Blum, Avrim L and Merrick L Furst (1997). “Fast planning through planning graph analysis”. In: *Artificial intelligence* 90.1-2, pp. 281–300.

Bylander, Tom (1994). “The computational complexity of propositional STRIPS planning”. In: *Artificial Intelligence* 69.1-2, pp. 165–204.

Cook, Stephen A (1971). “The complexity of theorem-proving procedures”. In: *Proceedings of the third annual ACM symposium on Theory of computing*. ACM, pp. 151–158.

Currie, Ken and Austin Tate (1991). “O-Plan: the open planning architecture”. In: *Artificial intelligence* 52.1, pp. 49–86.

Davis, Martin, George Logemann, and Donald Loveland (1962). “A machine program for theorem-proving”. In: *Communications of the ACM* 5.7, pp. 394–397.

Eén, Niklas and Niklas Sörensson (2003). “An extensible SAT-solver”. In: *International conference on theory and applications of satisfiability testing*. Springer, pp. 502–518.

Erol, Kutluhan, James A Hendler, and Dana S Nau (1994). “UMCP: A Sound and Complete Procedure for Hierarchical Task-network Planning.” In: *AIPS*. Vol. 94, pp. 249–254.

Fikes, Richard E and Nils J Nilsson (1971). “STRIPS: A new approach to the application of theorem proving to problem solving”. In: *Artificial intelligence* 2.3-4, pp. 189–208.

Fukunaga, Alex et al. (1997). “ASPEN: A framework for automated planning and scheduling of spacecraft control and operations”. In: *Proc. International Symposium on AI, Robotics and Automation in Space*.

García, Javier et al. (2013). “Combining linear programming and automated planning to solve intermodal transportation problems”. In: *European Journal of Operational Research* 227.1, pp. 216–226.

Ghallab, Malik, Dana Nau, and Paolo Traverso (2004). *Automated Planning: theory and practice*. Elsevier.

Gocht, Stephan and Tomáš Balyo (2017). “Accelerating SAT Based Planning with Incremental SAT Solving”. In: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*, pp. 135–139.

Großmann, Peter et al. (2012). “Solving periodic event scheduling problems with SAT”. In: *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*. Springer, pp. 166–175.

Hoffmann, Jörg and Bernhard Nebel (2001). “The FF planning system: Fast plan generation through heuristic search”. In: *Journal of Artificial Intelligence Research* 14, pp. 253–302.

Howey, Richard, Derek Long, and Maria Fox (2004). “VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL”. In: *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on*. IEEE, pp. 294–301.

Hutter, Frank et al. (2007). “Boosting Verification by Automatic Tuning of Decision Procedures”. In: *Formal Methods in Computer Aided Design, 2007. FMCAD '07*, pp. 27 –34. DOI: 10.1109/FAMCAD.2007.9.

Hutter, Frank et al. (2009). “ParamILS: An Automatic Algorithm Configuration Framework”. In: *Journal of Artificial Intelligence Research* 36, pp. 267–306.

Kautz, Henry and Bart Selman (1996). “Pushing the envelope: Planning, propositional logic, and stochastic search”. In: *Proceedings of the National Conference on Artificial Intelligence*, pp. 1194–1201.

Kautz, Henry A, Bart Selman, et al. (1992). “Planning as Satisfiability.” In: *ECAI*. Vol. 92. Citeseer, pp. 359–363.

Mali, Amol Dattatraya (1999). “Hierarchical task network planning as satisfiability”. In: *European Conference on Planning*. Springer, pp. 122–134.

— (2000). “Enhancing HTN Planning as Satisfability.” In: *Artificial Intelligence and Soft Computing*, pp. 325–333.

Mali, Amol Dattatraya and Subbarao Kambhampati (1998). “Encoding HTN Planning in Propositional Logic.” In: *AIPS*, pp. 190–198.

Moskewicz, Matthew W et al. (2001). “Chaff: Engineering an efficient SAT solver”. In: *Proceedings of the 38th annual Design Automation Conference*. ACM, pp. 530–535.

Nabeshima, Hidetomo et al. (2006). “Lemma Reusing for SAT based Planning and Scheduling.” In: *ICAPS*, pp. 103–113.

Nau, Dana et al. (1999). “SHOP: Simple hierarchical ordered planner”. In: *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*. Morgan Kaufmann Publishers Inc., pp. 968–973.

Nau, Dana et al. (2005). “Applications of SHOP and SHOP2”. In: *IEEE Intelligent Systems* 20.2, pp. 34–41.

Nau, Dana S et al. (2003). “SHOP2: An HTN planning system”. In: *Journal of artificial intelligence research* 20, pp. 379–404.

Prasad, Mukul R, Armin Biere, and Aarti Gupta (2005). “A survey of recent advances in SAT-based formal verification”. In: *International Journal on Software Tools for Technology Transfer* 7.2, pp. 156–173.

Raja, Purushothaman and Sivagurunathan Pugazhenthi (2012). “Optimal path planning of mobile robots: A review”. In: *International Journal of Physical Sciences* 7.9, pp. 1314–1320.

Ramoul, Abdeldjalil et al. (2017). “Grounding of HTN Planning Domain”. In: *International Journal on Artificial Intelligence Tools* 26.05, p. 1760021.

Rintanen, Jussi (2014). “Madagascar: Scalable planning with SAT”. In: *Proceedings of the 8th International Planning Competition (IPC-2014)* 21.

Rintanen, Jussi, Keijo Heljanko, and Ilkka Niemelä (2004). "Parallel encodings of classical planning as satisfiability". In: *European Workshop on Logics in Artificial Intelligence*. Springer, pp. 307–319.

— (2006). "Planning as satisfiability: parallel plans and algorithms for plan search". In: *Artificial Intelligence* 170.12-13, pp. 1031–1080.

Sacerdoti, Earl D (1975). *A structure for plans and behavior*. Tech. rep. SRI INTERNATIONAL MENLO PARK CA ARTIFICIAL INTELLIGENCE CENTER.

"Satisfiability: Suggested Format" (1993). In: *DIMACS Challenge. DIMACS*.

Sellers, William Irvin and Phillip Lars Manning (2007). "Estimating dinosaur maximum running speeds using evolutionary robotics". In: *Proceedings of the Royal Society of London B: Biological Sciences* 274.1626, pp. 2711–2716.

Silva, João P Marques and Karem A Sakallah (1997). "GRASP—a new search algorithm for satisfiability". In: *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*. IEEE Computer Society, pp. 220–227.

Sirin, Evren et al. (2004). "HTN planning for web service composition using SHOP2". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 1.4, pp. 377–396.

Tange, O. (2011). "GNU Parallel - The Command-Line Power Tool". In: *;login: The USENIX Magazine* 36.1, pp. 42–47. DOI: `http://dx.doi.org/10.5281/zenodo.16303`. URL: `http://www.gnu.org/s/parallel`.

Tate, Austin (1976). *Project planning using a hierarchic non-linear planner*. Department of Artificial Intelligence, University of Edinburgh.

Weser, Martin, Dominik Off, and Jianwei Zhang (2010). "HTN robot planning in partially observable dynamic environments". In: *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. IEEE, pp. 1505–1510.

Wilkins, David E (1984). "Domain-independent planning Representation and plan generation". In: *Artificial Intelligence* 22.3, pp. 269–301.

Wood, R Glenn and Rob A Rutenbar (1998). "FPGA routing and routability estimation via Boolean satisfiability". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 6.2, pp. 222–231.