Dissertation zur Erlangung des Doktorgrades der Technischen Fakultät der Albert-Ludwigs-Universität Freiburg im Breisgau

Analysis of Partial Order Reduction Techniques for Automated Planning

vorgelegt von
M. Sc. Yusra Al-Khazraji

2017

Betreut von
Prof. Dr. Bernhard Nebel

Tag der Disputation:
19.06.2018

Dekan:
Prof. Dr. Oliver Paul

Prüfungskommission:
Prof. Dr. Christoph Scholl (Vorsitz)
Prof. Dr. Bernd Becker (Beisitz)
Prof. Dr. Bernhard Nebel (Betreuer)
Prof. Dr. Andreas Podelski (Prüfer)

Zusammenfassung

Automatisierte Planung ist ein Teilbereich der Künstlichen Intelligenz, der eng mit Entscheidungsfindung zusammen hängt. Gegeben ein Planungsproblem, das durch einen Anfangszustand (d.h. eine Startkonfiguration des Problems), eine Menge von Aktionen und eine Zielbeschreibung spezifiziert wird, beschäftigt sich ein Planungsalgorithmus mit der Suche nach einer Abfolge von Aktionen oder einer Strategie, um die Zielbedingungen vom Anfangszustand aus zu erreichen. Planung, in all ihren Variationen, ist ein schweres Problem. Die klassische Planung ist eine der einfachsten Planungsformen. Trotz ihrer Einfachheit hat diese immer noch eine hohe Berechnungskomplexität. Daher können Planungsprobleme nicht durch Brute-Force-Algorithmen gelöst werden, sondern eher durch zielgerichtete Algorithmen, die die Komplexität der Planung abmildern. Planung als heuristische Suche ist eine führende Technik, die im Bereich der automatisierten Planung benutzt wird. Aktuelle Forschung hat jedoch gezeigt, dass Planung als heuristische Suche durch orthogonale Techniken komplementiert werden muss. Partial-Order-Reduktion (POR) ist ein bekannter und zuverlässiger Ansatz, der ursprünglich im Bereich der Computer-Aided-Verification entstanden ist, um die Zustandsraumexplosion zu bekämpfen. In der Regel wird diese für explizite Zustandsraum-Suche verwendet, um die Größe des generierten Zustandsraums zu reduzieren. Es gibt zwei Kategorien von POR-Techniken: Zustandsreduktion und Transitionsreduktion. Die Ersteren sind auf das Ignorieren einiger Zustände spezialisiert, die garantiert unkritisch sind, um Terminalzustände (z.B. Zielzustände für Planungsprobleme) zu erreichen, während die Letzteren entworfen sind, um redundante Zustände zu ignorieren. Später wurden neue POR-Techniken für optimale Planungsalgorithmen vorgeschlagen. Allerdings war der Zusammenhang zwischen den vorherigen und späteren Techniken unklar. Zudem waren einige der neuartigen Techniken fehlerhaft.

Im Rahmen der klassischen Planung untersuchen wir zwei Varianten einer mächtigen Zustandsreduktionstechnik aus dem Bereich des Model-Checkings, nämlich Stubborn Sets. Darüber hinaus schlagen wir eine neue Definition von Stubborn Sets für Fully Observable Nondeterministic Planning (FOND) vor. Für die klassische Planung zeigen wir, dass Stubborn Sets die Vollständigkeit und Optimalität von Suchalgorithmen erhalten, während diese für die FOND-Planung zumindest noch die Vollständigkeit erhalten. Darüber hinaus vergleichen wir zwischen Stubborn Sets und vorhandenen POR-Techniken, die für optimale Planung vorgeschlagen wurden. Außerdem studieren wir eine Transitionsreduktionstechnik aus dem Bereich des Model-Checkings, nämlich Sleep Sets, und klassifizieren diese Methode in vier Hauptvarianten. Danach untersuchen wir den Zusammenhang zwischen diesen Varianten und verschiedenen Suchalgorithmen. Weiterhin untersuchen wir theoretisch, welche Varianten für Suchalgorithmen (wie A*, IDA*, und Breitensuche) sicher sind. Wie auch Stubborn Sets vergleichen wir Sleep Sets mit anderen Transitionsreduktionstechniken für die Handlungsplanung. Des weiteren beschreiben wir eine Familie der Verallgemeinerung der Sleep-Sets-Methode und wir evaluieren die allgemeinste Form für optimale Planung. Schließlich evaluieren wir experimentell die Techniken, die wir für die Handlungsplanung vorschlagen.

Zusammenfassend lässt sich sagen, dass diese Arbeit POR-Techniken für Handlungsplanung untersucht, die formalen Beziehungen zwischen mehreren POR-Techniken darstellt und die theoretischen Beweise durch empirische Auswertungen stärkt.

Abstract

Automated planning is an area of Artificial Intelligence (AI) that is closely related to decision making. Given a planning problem, which is described as an initial state (i.e., a start configuration of the problem), a set of actions, and a goal description, a planning algorithm is concerned with finding a sequence of actions or a strategy for achieving the goal conditions from the initial state. Planning, in all its variations, is a hard problem. Classical planning is one of the simplest forms of planning. Despite its simplicity, it is still computationally intractable. Therefore, planning problems cannot be solved by brute-force algorithms, but rather by goal-directed algorithms that tackle the complexity of planning. Planning as heuristic search is a prominent technique that has been considered in the area of automated planning. However, recent research has shown that planning as heuristic search needs the aid of orthogonal techniques. Partial order reduction is a well-known and reliable approach that has originally been introduced in the area of computer aided verification to tackle the state explosion problem. It is usually used for explicit state-space search to reduce the size of the generated state space. There are two categories of partial order reduction techniques: State reduction and transition reduction techniques. The former are specialized to prune some states that are guaranteed to be uncritical for reaching some final states (e.g., goal states in planning), while the latter are designed to prune only redundant states. Later, new partial order reduction techniques have been proposed for optimal planning algorithms. However, the formal relationships between the previous and later techniques were unclear, and furthermore, some of the novel techniques were faulty.

In the context of classical planning, we investigate two variants of a powerful state reduction technique from the area of model checking, called stubborn sets. Furthermore, we propose a new definition of stubborn sets for Fully Observable Non-Deterministic planning (FOND). For classical planning, we show that stubborn sets are completeness and optimality preserving, whereas for FOND planning, we show that they are at least completeness preserving. In addition, we compare between stubborn sets and previous partial order reduction techniques proposed for optimal planning. Moreover, we study a transition reduction technique from the area of model checking, called sleep sets, and classify this method into four main variants. Then, we show the relationship between these variants and the different search algorithms. Precisely, we theoretically investigate which variants are safe for search algorithms like A*, IDA*, and breadth-first search. Like stubborn sets, we compare sleep sets with other transition reduction techniques for planning. In addition, we describe a family of generalization to the sleep sets method and evaluate the most general form for optimal planning. Finally, we experimentally evaluate the techniques we propose for planning.

In summary, this thesis investigates sound partial order reduction techniques for automated planning, establishes the formal relationships between several partial order reduction techniques, and strengthens the theoretical proof by empirical evaluation.

Acknowledgments

It would not have been possible for me to pursue the journey of my PhD research without the support of some people to whom I owe my success. First and foremost, I would like to express my special gratitude to my advisor Prof. Bernhard Nebel for giving me the opportunity to be a member of his research group throughout my doctoral research and for his continuous guidance and advice.

I am also deeply grateful to Martin Wehrle for his scientific cooperation from the very beginning of this journey. He continuously enriched my knowledge with many insightful and fruitful discussions. I have learned from him never to give up on a rejected paper and how to transform it to one that wins a best paper award.

Special thanks to Robert Mattmüller for providing me with advice and answering my scientific questions. I have learned a lot from his enthusiasm and love for research and knowledge.

In addition, I would like to thank Prof. Malte Helmert and Gabriele Röger for being great teachers of AI planning. I attended their lecture during my Master studies and it was the reason why I decided to pursue research in the area of planning.

Furthermore, I would like to thank Dominik Winterer for proof reading this thesis and Christian Dornhege for helping me improving the abstract.

I also want to thank Prof. Robert C. Holte and Michael Katz with whom I was happy to do research. Many thanks to my colleagues from the research group on Foundations of Artificial Intelligence at the university of Freiburg: Johannes Aldinger, Thorsten Engesser, Petra Geiger, Florian Geißer, Andreas Hertle, Uli Jakob, Felix Linder, Tim Schulte and Benedict Wright. My thanks are also for my former colleagues: Christian Becker-Asano, Patrick Eyerich, Moritz Göbelbecker, Roswitha Hilden, Julien Hué, Thomas Keller, Johannes Löhr, Manuela Ortlieb, Alexander Schimpf, Dali Sun, Stefan Wölfl and Dapeng Zhang.

Last but not least, I would like to thank Stefan Schäfer for his encouragement and support during the last phase of this journey. Thank you!

Contents

Chapter 1

Introduction

This chapter motivates the topic of this thesis, states its outline, and summarizes the contribution of the author.

1.1 Motivation

Automated planning is a discipline of Artificial Intelligence (AI) that performs reasoning before acting [GNT04]. It plays a key role in building autonomous intelligent agents that are capable of exhibiting goal-oriented behaviors. Given a formal and concise description of the problem, which usually consists of an *initial state* of the world, a set of *actions* that can be executed by the agent to modify the state of the world, and a *goal* description that is required to be achieved, a planning system computes a solution that leads to a state (or states) satisfying the goal. These solutions are called plans and their structures depend on the description of the problem provided to the planning system. There is a wide range of planning models that vary in the assumptions made about the agent capabilities and the environment in which she is supposed to act.

Classical planning is a planning form that considers a restricted model for describing problems. It is the basic and simplest planning model in the sense that it assumes the agent has full control of her actions and complete knowledge about the environment [GNT04]. Extended models can be obtained by relaxing some of the restricted assumptions. For example, the model that assumes nondeterminism in action execution and complete knowledge about the environment is known as fully observable nondeterministic planning (FOND). A solution for a classical planning problem is a sequence of actions, whereas a solution for a FOND planning problem is a strategy (a.k.a. policy) which is a function that maps states to actions. Indeed, a considerable amount of research has been dedicated to investigate efficient planning algorithms for solving classical planning problems. The main incentive for studying classical planning is the high computational complexity of planning: It is intractable in theory even when making very tight restrictions about the structure of the problem [Byl91; BN95].

State-space search is a well-known approach for solving classical planning problems [GNT04]. The aspects of a planning problem are described using variables and values, where each variable has a finite domain of values. A state is given as an assignment of variables to values from their domains. The *state space* of a planning problem is the Cartesian product of all possible variable assignments. The reachable state space is the set of all states that can be reached by applying sequences of actions starting from the initial state. State-space search solves a planning problem by exploring a part of the reachable state space. The main challenge to state-space search is the size of the state space, which is exponential in the number of state variables. Intuitively, exploring the whole state space is not affordable even for relatively small realistic problems. For this reason, efficient search algorithms, that explore only a reasonable size of the state space, are desired. Let's call the set of explored states by a search algorithm the *explored space*. These algorithms are usually armed with *heuristic functions* or *heuristics*, which map each state in the state space to a numeric value that estimates the distance of the state from a goal state. States with low heuristic values have high priority to be explored by the search algorithm. The scalability of search algorithms mostly depends on the quality of the heuristic function. Heuristic functions that provide values, which are close to the real distances of states from a goal state, are considered *informative*.

Planning as heuristic search has been one of the most prominent planning approaches for decades [McD96; McD99]. Therefore, early planning research has focused on developing informative heuristic functions to efficiently guide progression search algorithms towards goals [BG01; HN11]. In particular, the research on optimal planning (i.e., extracting plans with minimal costs) has witnessed the development of powerful admissible heuristics like merge-and-shrink and landmark-cut heuristics [HHH07; HD09].
Despite the success of heuristic search, it has been shown that using standalone heuristic functions can still lead to exploring a part of the state space whose size is necessarily exponential in the description of the problem [HR08]. Therefore, later research started to investigate orthogonal techniques to heuristic search that can further improve the performance of heuristic search planners.

Pruning techniques [Val89; Pel93; GHP95; God96; FL99; BH11; WH12; DKS13] are well-known approaches that have originally been developed in the area of computer-aided verification [CGP01; BK08] in order to tackle the state explosion problem. They help reduce the size of the explored space by executing only a subset of the applicable actions in a given state.

Partial order reduction [Val89; GW92; GHP93; KP95; GHP95; God96; CXY09; CY09; WH12] is a pruning technique that has reliably been used for decades in the area of model checking. Partial order reduction is beneficial when the components of the underlying system are independent of each other. These components are called *transitions* in computer aided verification, and *actions* or *operators* in planning. A transition (or an action) is a structure that transforms the system from one configuration (or state) to another. Executing independent transitions (or actions) in all possible orders can result in exploring unnecessary parts of the state space, which can be pruned by reducing the partial order between transitions.

Recently, partial order reduction methods have been utilized in the context of optimal classical planning in order to reduce the size of the explored state space, thereby improving the scalability of heuristic search [HG00; CXY09; CY09; BH11; WH12; Alk+12]. However, some partial order reduction techniques have been proposed for optimal classical planning without being related to the previous approaches from the area of model checking [CXY09; CY09].

Furthermore, some of the new techniques, in the original work, turned out to be faulty. In particular, they were neither completeness nor optimality preserving[1].

In this thesis, the main focus is on investigating sound partial order reduction techniques that can be combined with heuristic search for automated planning on both theoretical and empirical levels. Furthermore, the pruning power of investigated techniques are thoroughly analyzed and compared to other techniques.

1.2 Outline

This thesis consists of the following: The rest of this chapter states the contribution of the author in the area of automated planning throughout the doctoral process. Chapter 2 introduces the preliminaries of classical planning and presents the basic formal definitions needed throughout this thesis. Chapter 3 introduces partial order reduction and its categories, and states the preliminaries and the necessary definitions related to partial order reduction. In addition, this chapter discusses previous work related to partial order reduction. The first part of Chapter 4 presents the *stubborn sets* technique and its variants for planning, while the second part briefly summarizes the relationship between stubborn sets and a pruning technique from previous work. Chapter 5 introduces the preliminaries of FOND planning and a stubborn sets technique for nondeterministic planning tasks. Chapter 6 is concerned with applying the *sleep sets* technique, that has been introduced for computer aided verification, to classical planning. It provides a classification of sleep sets into four variants based on the contexts in which sleep sets have been presented in the literature. Furthermore, it contains a detailed analysis of the behavior of sleep sets with different search algorithms. In particular, it studies the combination of sleep sets and stubborn sets with duplicate elimination and graph search in the context of classical planning. Moreover, the pruning power of sleep sets relative to previous partial order reduction techniques is discussed in this chapter. Finally, this chapter presents a family of transition reduction techniques, called *generalized sleep sets*, that inherit their features from sleep sets and another technique called *move pruning*. Chapter 7 sheds light on combining state and transition partial order reduction techniques in details. In Chapter 8, empirical evaluations of the most important techniques presented in this thesis are shown and discussed. Finally, Chapter 9 provides a conclusion for the topics presented in this thesis and a summary of potential future work.

1.3 Contribution

The author of this thesis has made several contributions related to the application of partial order reduction techniques to the area of automated planning. These contributions are summarized in the following:

Stubborn sets for optimal classical planning. The author has investigated the application of *strong stubborn sets* and *weak stubborn sets* in the con-

[1]These techniques are stratified planning and expansion core [CXY09; CY09], which have been corrected later by Wehrle and Helmert [WH12].

text of classical planning, and provided a theoretical proof of their completeness and optimality. The contribution related to strong stubborn sets has been presented as a conference paper ECAI 2012 [Alk+12], whereas the work related to weak stubborn sets is published in a conference paper at ICAPS 2017 [Win+17]. In addition, the author has participated in the work that revealed the pruning power of stubborn sets and investigated their relationship to another partial order reduction technique. This research has been presented as a conference paper at ICAPS 2013 [Weh+13]. This paper won the *best paper award* at ICAPS 2013.

Sleep sets for optimal classical planning. The research on utilizing *sleep sets* in the context of optimal classical planning is the most prominent contribution of the author throughout her doctoral studies. In this context, sleep sets have been shown to be beneficial for planning when combined with tree search algorithms like IDA*. Furthermore, a family of *generalization* of sleep sets have been defined for optimal classical planning. Both of these contributions are the core ideas of a conference paper presented at AAAI 2015 [HAW15]. In addition, the author has contributed in studying the pruning power of sleep sets and their relationship to other partial order reduction techniques. This work has not been published yet.

Finally, the author classified the sleep sets technique into four main variants and showed the relationship of these variants with some AI search algorithms. In particular, one variant has been proven to be safe when combined with A* and can be used in the context of optimal classical planning. This work is the most significant contribution of the author and has been presented at SoCS 2016 [AW16]. This paper won the *best paper award* at SoCS 2016.

Stubborn sets for FOND planning. The author proposed a new stubborn sets pruning technique for FOND planning. This work is presented as conference paper at ICAPS 2017 [Win+17] in which the author has a significant contribution.

Stubborn sets in International Planning Competitions. Strong stubborn sets have been used in two planners that participated in two different International Planning Competitions (IPCs) to provide an empirical evidence of their pruning power:

- *Metis* [Alk+14] is a deterministic and optimal planning system that is based on the planning system Fast Downward [Hel06]. In addition to strong stubborn sets, Metis is powered by *symmetry reduction* [DKS13] and incremental LM-cut [PH13]. This planner participated in the deterministic track of IPC-14.
- *Aidos* [Sei+16] is a deterministic and optimal planning system that is also based on Fast Downward and designed for the detection of unsolvability in planning problems. This planner is armed with several orthogonal techniques whose combination has synergy effects in discovering unsolvable instances by detecting dead ends as early as possible. In particular, the planner combines a number of heuristics

specialized for fast dead ends detection with pruning techniques. Aidos participated in IPC-16 and was the *winner* of the unsolvability track.

The list of publications of the author in chronological order:

- Yusra Alkhazraji, Martin Wehrle, Robert Mattmüller, and Malte Helmert. "A Stubborn Set Algorithm for Optimal Planning". In: *Proceedings of Twentieth European Conference on Artificial Intelligence (ECAI 2012).* 2012, pp. 891–892
- Martin Wehrle, Malte Helmert, Yusra Alkhazraji, and Robert Mattmüller. "The Relative Pruning Power of Strong Stubborn Sets and Expansion Core". In: *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS 2013).* 2013, pp. 251–259. **(Best paper award).**
- Yusra Alkhazraji, Michael Katz, Robert Mattmüller, Florian Pommerening, Alexander Shleyfman, and Martin Wehrle. "Metis: Arming Fast Downward with Pruning and Incremental Computation (planner abstract)". In: *In the Eighth International Planning Competition (IPC 2014) (deterministic track).* 2014, pp. 88–92
- Robert C. Holte, Yusra Alkhazraji, and Martin Wehrle. "A Generalization of Sleep Sets Based on Operator Sequence Redundancy". In: *Proceedings of the Twenty-Ninth AAAI Conference (AAAI 2015).* 2015, pp. 3291–3297
- Jendrik Seipp, Florian Pommerening, Silvan Sievers, Martin Wehrle, Chris Fawcett, and Yusra Alkhazraji. "Fast Downward Aidos (planner abstract)". In: *In the First Unsolvability International Planning Competition (IPC 2016).* 2016. **(Winner of the unsolvability track at IPC-16).**
- Yusra Alkhazraji and Martin Wehrle. "Sleep Sets Meet Duplicate Elimination". In: *Proceedings of the Ninth Annual Symposium on Combinatorial Search (SOCS 2016).* 2016, pp. 2–9. **(Best paper award).**
- Dominik Winterer, Yusra Alkhazraji, Michael Katz, and Martin Wehrle. "Stubborn Sets for Fully Observable Nondeterministic Planning". In: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017).* 2017

In summary, this thesis is the first work that studies a wide range of partial order reduction techniques for automated planning both on theoretical and empirical levels.

Chapter 2

Classical Planning

Planning has several forms depending on the model of actions being considered to model planning problems [GNT04]. Classical planning is the basic form of AI planning that considers environments with several restrictive assumptions. Given a description of a planning problem, a classical planning system either finds and retrieves a sequence of actions as a solution (plan) or proves the problem unsolvable if no plan can be found.

This chapter introduces the mathematical model that is considered to represent classical planning problems, the specification language in which the problems are concisely described, and an introduction to planning algorithms and general search concepts.

2.1 Mathematical Model

The statement of the planning problem induces a mathematical model that has the following characteristic [RN10]:

- **Finite**: the system has a finite number of states.
- **Static**: no external events can contribute to changing the state of the system.
- **Deterministic**: every action applicable in a state leads to a unique successor state.
- **Fully-observable**: the states of the system should be fully known.
- **Discrete**: the actions and state variables of the system are discrete.

Note that full-observability is obtained by having a fully-observable initial state and deterministic actions. It is worthwhile to mention that extended planning models can be obtained by relaxing some of these restrictions to describe more specialized planning forms that are suitable for solving real-world planning problems.

Definition 1 (deterministic transition system). The classical model is represented mathematically as a *labeled deterministic transition system* (a.k.a. state space) which is depicted as a directed graph. A deterministic transition system $\mathcal{T}$ is a 5-tuple (S, s_0, A, T, G), where

- S is a finite set of *states*,
- $s_0 \in S$ is the *initial state*,
- A is a finite set of transitions (*actions*),
- $T : S \times A \mapsto S$ is the *transition relation*, and
- $G \subseteq S$ is a finite set of *goal states*.

2.2 Specification Language

The transition system model provides a theoretical representation of planning problems. However, it is computationally hard to generate the complete transition system even for relatively small planning problems. For this reason, specification languages are necessary to describe transition systems in a compact way. The first and most well-known planning specification language is the propositional STRIPS language [FN71]. A more concise language is called *finite-domain representation* language. We consider a restrictive form of this language called SAS$^+$ [BN95] that is similar to STRIPS except for the fact that SAS$^+$ considers multi-valued state variables instead of Boolean variables. It is worthwhile to mention that SAS$^+$ is conciser than STRIPS but not more expressive than it.

In the following, most of the preliminary definitions and notions that are used in this thesis are introduced.

Definition 2 (finite-domain state variable). A *state variable* v is a variable symbol that is associated with a non-empty finite domain of values $\mathcal{D}_v$ and can hold one value from its domain at a time.

Definition 3 (partial state and state). Given a set of state variables $\mathcal{V}$, a *partial state* s is a partial function defined on a subset $\mathcal{V}' \subseteq \mathcal{V}$ such that it assigns to each variable $v \in \mathcal{V}'$ a value from its domain. We write $s[v]$ to denote the value to which v is mapped by s. The set of state variables for which s is defined is denoted by $vars(s)$. For a state variable $v \notin vars(s)$, we write $s[v] = \bot$ to denote an undefined value for v. If $\mathcal{V}' = \mathcal{V}$, then s is called a *state*. A state s *satisfies* a partial state s', written $s \models s'$, *iff* $s[v] = s'[v]$ for all $v \in vars(s')$.

Definition 4 (operator). An *operator* o is a structure that consists of two parts: A *precondition* $pre(o)$ and an *effect* $\mathit{eff}(o)$, where both are partial states. Operator o is associated with a non-negative real number $cost(o) \in \mathbb{R}_0^+$ denoting its cost. We say that o is *applicable* in state s *iff* $s \models pre(o)$. The result of applying o in state s, denoted as $o(s)$, is defined as follows:

- $o(s)[v] = \mathit{eff}(o)[v]$ if $v \in vars(\mathit{eff}(o))$,
- $o(s)[v] = s[v]$, otherwise.

If o has an empty precondition (i.e., it has no precondition) then o is applicable in every state and its precondition is denoted by $\top$. We use $app(s)$ to denote the set of applicable operators in state s. For a variable v and a state s, an operator o is *v-applicable* in s *iff* either $v \notin vars(pre(o))$, or $pre(o)[v] = s[v]$.

At this point, we introduce the definition of SAS$^+$ planning tasks.

Definition 5 (SAS$^+$ planning task). An SAS$^+$ planning task Π is a 4-tuple $\langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ where

- $\mathcal{V}$ is a finite set of finite-domain state variables,
- $\mathcal{O}$ is a finite set of operators,
- s_0 is the *initial state*, and
- $s_\star$ is a partial state representing the set of goal states.

From now on, we call an SAS$^+$ planning task a planning task for short. Each classical planning task Π induces a deterministic transition system $\mathcal{T}_\Pi = (S, s_0, A, T, G)$ where S is the set of states over $\mathcal{V}$, s_0 is the same initial state of Π, $A = \mathcal{O}$, and $s \in G$ *iff* $s \models s_\star$ for all $s \in S$. A transition $(s, o, s') \in T$ *iff* o is applicable in s and $o(s) = s'$. If all $o \in \mathcal{O}$ have the same cost, the operators in $\mathcal{O}$ are called *unit-cost* operators. A state s' is *reachable* from a state s if there exist states $s_1, \ldots, s_n$, where $s = s_1$, and operators $o_1, \ldots, o_{n-1}$, with $i \in \{1, \ldots, n\}$, such that $o_i \in \mathit{app}(s_i)$, $o_i(s_i) = s_{i+1}$, and $s_n = s'$. The size of this transition system is exponential in the number of state variables of the planning task. Therefore, computing this system explicitly is computationally intractable. Next, we introduce more definitions that we need in this thesis.

Definition 6 (goal-related variable). Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task. A variable $v \in \mathcal{V}$ is *goal-related iff* $v \in \mathit{vars}(s_\star)$.

Definition 7 (domain transition graph). Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task and $v \in \mathcal{V}$ be a variable. The *domain transition graph* for v is a directed graph $\mathit{DTG}(v) = \langle V, E \rangle$, where the vertices are defined as $V = \mathcal{D}_v$, and there is an edge $(d, d') \in E$ *iff* there is an operator $o \in \mathcal{O}$ such that $\mathit{eff}(o)[v] = d'$, and either $v \notin \mathit{vars}(\mathit{pre}(o))$ or $\mathit{pre}(o)[v] = d$.

Definition 8 (fact). Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task. A *fact* is a pair $\langle v, d \rangle$, where $v \in \mathcal{V}$ and $d \in \mathcal{D}_v$. A fact $\langle v, d \rangle$ is *satisfied* by state s if $s[v] = d$; otherwise, it is unsatisfied. We say that a fact $\langle v, d \rangle$ is in partial state s *iff* $v \in \mathit{vars}(s)$ and $s[v] = d$.

2.3 Planning Algorithms

Since the emergence of AI planning, several classes of classical planning algorithms have been deployed by the research in this field. For example, state-space planning, plan-space planning (e.g. partial-order planning) [PW92], planning as satisfiability (or SAT-based planning) [KS92; Rin09; Rin12], and planning as a CSP (Constraint Satisfaction Problem) [DK01] are all well-known planning approaches.
State-space planning is the simplest approach and the most popular among others [GNT04]. The rationale behind the prominence of state-space planning is the direct correspondence between the mathematical model of classical planning and directed graphs by which deterministic transition systems are represented as we have seen in the last section [GB13]. Given this fact, a planning task can

be mapped to a path-finding problem which is typically solved by a path-finding search algorithm. State-space planning is performed by recruiting a search algorithm to explore the state space of the planning problem. The search process is done by applying operators starting from the initial state until a goal state is detected, and a plan is extracted and retrieved as a result. In the following, we formally introduce the definitions of explored spaces and plans.

Definition 9 (Explored space). Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task and $\mathcal{T}_\Pi = (S, s_0, A, T, G)$ be its induced transition system. Furthermore, let $\mathcal{A}$ be a planning algorithm that runs on Π to find a solution. The explored space of $\mathcal{A}$ is transition system $\mathcal{S}_\Pi = (S', s'_0, A', T', G')$, where $S' \subseteq S$, $s'_0 = s_0$, $A' \subseteq A$, $T' = T \cap (S' \times A' \mapsto S')$, and $G' = G \cap S'$.

Definition 10 (plan). Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task. A plan π for Π is a sequence of operators $o_1, \ldots, o_n$ where o_1 is applicable in s_0, and there are states $s_1, \ldots, s_n$ such that $s_i = o_i(s_{i-1})$, for all $i \in \{1, \ldots, n\}$, and $s_n \models s_\star$. The cost of a plan π is the sum of its operators costs, i.e.

$$cost(\pi) = \sum_{i=1}^{n} cost(o_i).$$

A plan $\pi = o_1, \ldots, o_n$ is *optimal* if it has the minimal cost among all plans for Π; otherwise, it is *suboptimal.*

2.4 General Search Concepts

In the following, we list the most important terminology associated with search algorithms in order for this thesis to be self-contained.

State expansion is the process of applying applicable operators in a given state, and generating its successor states.

State generation is the process of creating a successor state of a given state by applying an applicable operator in it.

Open list is a data structure used by search algorithms to keep track of generated states that may need to be expanded.

Closed list is a data structure used by search algorithms to store states that have been expanded.

Graph search algorithms are search algorithms that retain all expanded states in memory (i.e., in a closed list) in order to avoid expanding them again. In other words, if a state, which is already in the closed list, has been generated again, then it is pruned as duplicate. Graph search saves the effort of exploring redundant parts of the state space at the expense of high memory usage. Breadth-first search and A* with a closed list are examples of graph search algorithms.

Tree search algorithms are search algorithms that do not use a closed list. In other words, tree search algorithms do not remember expanded states, and if such states are generated again, they are added to the open list to be ready for expansion. Tree search algorithms save the memory (represented by the closed list) that is needed for storing expanded states. However, unlike graph search, tree search algorithms might explore redundant parts of the state space multiple times, which can excessively increase the runtime of the algorithm. Depth-first search and Iterative-deepening search (IDA*) are two well-known tree search algorithms.

The g-value of a state is the overall cost of operators needed to reach this state from the initial state.

The h-value of a state is the estimated cost needed to reach a goal state from the state (heuristic value).

Search node is a bookkeeping data structure that is used by some search algorithms to keep track of a state s, and some information related to it, like its g-value, h-value, parent state, and the operator that has been applied in the parent state to generate s.

Chapter 3

Partial Order Reduction

Partial order reduction is a pruning technique that was originally proposed in the area of computer aided verification (particularly in model checking) [Val89; GW92; GHP93; GHP95; God96]. Model checking is an automatic verification technique for concurrent systems [CGP01; BK08]. It is the problem of verifying whether a mathematical model of a system satisfies a formal property which ensures that the system does not express an undesired behavior that violates its specifications (e.g., the system is free from deadlocks). One of the most critical challenges that model checking needs to deal with is the combinatorial *state explosion problem*, where the size of the state space of the verified system blows up exponentially in the size of its individual components.

Several approaches have been used in model checking in order to tackle this problem, e.g., employing symbolic methods [McM93], abstraction-based methods [CGL94; CKV10], bounded model checking [Cla+01; Bie+03], and partial order reduction among others. An asynchronous concurrent system consists of a set of components where only one component can execute at a time. This behavior is modeled by interleaving the execution of the individual components. The interleaving model imposes an arbitrary ordering between concurrent components. In order to avoid prioritizing one order and neglect other orders, the events of the components are interleaved in all possible ways. Considering all possible orderings between transitions can contribute to the state explosion problem. Furthermore, the ordering between independent transitions is irrelevant if the specification language is concerned only about the final outcome of the independent transitions [CGP01].

As its name suggests, partial order reduction aims at reducing the size of the explored state space by reducing the partial oder between the transitions of the system. Some forms of partial order reduction impose an explicit total order on the transitions to be used during search. Roughly speaking, if two transitions are independent from each other, in the sense that executing them in any order leads to the same final consequence, then considering one execution order is sufficient to guarantee the reachability of the final outcome.

To have an intuition about the power of this technique, consider a system of n independent operations which can be executed in $n!$ different orderings, thereby leading to 2^n different states. Partial order reduction can reduce the number of explored states to $n + 1$ by executing only one ordering [CGP01]. Figure 3.1 is an example of pruning by partial order reduction in model checking

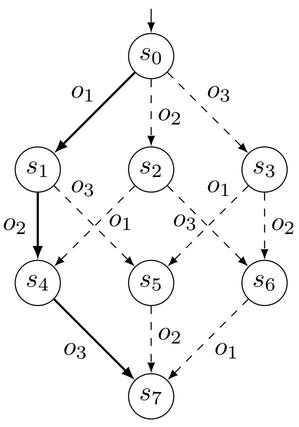


Figure 3.1: Pruning by partial order reduction [CGP01]

by Clarke, Grumberg, and Peled [CGP01]. The states preceded by dashed lines are pruned (not generated) by the search, and only states s_0, s_1, s_4 and s_7 are generated.

There are two main categories of partial order reduction methods in the literature. They differ from each other in the way of defining the dependency between transitions, and hence, in the form of pruning they perform. In order to define the effect of these categories on transition systems, we need to be familiar with graph homomorphism.

Definition 11 (graph homomorphism). Let $G = (V, E)$ and $G' = (V', E')$ be graphs, where V and V' denote the sets of vertices, and E and E' denote the sets of edges of G and G', respectively. A *graph homomorphism* ξ (or homomorphism for short) is a mapping from V' to V ($\xi : V' \mapsto V$) such that $(u, v) \in E'$ implies $(\xi(u), \xi(v)) \in E$.

State Reduction Techniques. In computer aided verification, state reduction techniques are partial order reduction techniques that can safely prune unique states from the explored space given that the absence of those pruned states is not crucial for verifying a given property of the system. Conceptually, a state reduction technique is a function that takes a explored space $\mathcal{T} = (S, s_0, A, T, G)$ as an argument and returns a *reduced* explored space $\mathcal{T}' = (S', s'_0, A', T', G')$ such that $S' \subseteq S$, $s'_0 = s_0$, $A' = A$, $T' = T \cap (S' \times A' \mapsto S')$, and $G' = G \cap S'$. Furthermore, there is a homomorphism ξ from $\mathcal{T}'$ to $\mathcal{T}$ ($\xi : \mathcal{T}' \to \mathcal{T}$) such that the following holds:

- $\xi(s'_0) = s_0$,
- $(s, a, s') \in T'$ implies $(\xi(s), a, \xi(s')) \in T$, and
- $g' \in G'$ implies $\xi(g') \in G$.

Transition Reduction Techniques. Similar to state reduction techniques, transition reduction techniques have also been proposed in the area of computer aided verification for the first time. Unlike state reduction techniques, transition

reduction techniques do not prune states from the state space but rather prune transitions that lead to states that have already been visited by the search or are guaranteed to be visited; hence, they prune *redundant* states.
Formally, a transition reduction technique is a function that takes a explored space $\mathcal{T} = (S, s_0, A, T, G)$ as an argument and returns a *reduced* explored space $\mathcal{T}' = (S', s'_0, A', T', G')$ where $S' = S$, $s'_0 = s_0$, $A' = A$, $T' = T \cap (S' \times A' \mapsto S')$, and $G' = G$. As in state reduction techniques, there is also a homomorphism ξ from the reduced transition system to the original transition system with the same properties mentioned above.

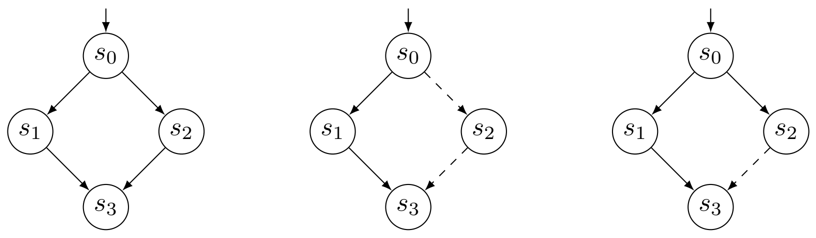


Figure 3.2: Original state space (left), reduced state space with a state reduction technique (middle), and reduced state space with transition reduction technique (right).

Figure 3.2 illustrates the differences between the original transition system and the reduced transition systems of state and transition reduction techniques, respectively.

3.1 Preliminaries

In the following, we will introduce the most important definitions and notations related to using partial order reduction for planning that are used in the rest of this thesis.

Definition 12 (path). A *path* is a sequence of operators $o_1, \dots, o_n$ that are sequentially applicable in a state s, i.e., there exist states $s_1, \dots, s_n$ with $s = s_1$, $o_i \in app(s_i)$, and $o_i(s_i) = s_{i+1}$, for $i \in \{1, \dots, n\}$. We say that $\sigma = o_1, \dots, o_n$ is applicable in state s if $o_1 \in app(s)$. We use $\sigma(s)$ as a shorthand for $o_n(\dots(o_1(s)))$ and $|\sigma|$ to denote the number of operators in σ (length of σ). The cost of σ is the accumulated cost of its operators, i.e.,

$$cost(\sigma) = \sum_{i=1}^{n} cost(o_i).$$

For path $\sigma = o_1, \dots, o_n$, we use $ops(\sigma)$ to denote the set containing the operators $o_1, \dots, o_n$. The empty path is denoted by ε. A prefix of σ is a nonempty initial segment of σ, i.e., $o_1, \dots, o_k$ with $k \in \{1, \dots, |\sigma|\}$. A suffix of σ is nonempty final segment of σ, i.e., $o_k, \dots, o_n$ with $k \in \{1, \dots, |\sigma|\}$.

In this thesis, we use the Greek letters $(\sigma, \rho, \delta, \dots)$ to denote paths. In addition, we use the terms "paths" and "operator sequences", interchangeably.

Definition 13 (permutation equivalent paths). Two paths σ and σ' are *permutation equivalent*, written as $\sigma \equiv \sigma'$, if $ops(\sigma) = ops(\sigma')$ and for all states s in which σ and σ' are applicable, it holds that $\sigma(s) = \sigma'(s)$.

Definition 14 (equivalence classes of paths). The *equivalence class of a path* σ, denoted as $[\sigma]_{\equiv}$, is the set of all paths that are permutation equivalent to σ.

Definition 15 (interference and commutativity of operators). Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task. Furthermore, let o and o' be operators in $\mathcal{O}$.

- o *disables* o' *iff* there exists $v \in vars(eff(o)) \cap vars(pre(o'))$ such that $eff(o)[v] \neq pre(o')[v]$.
- o *can enable* o' *iff* there exists $v \in vars(eff(o)) \cap vars(pre(o'))$ such that $eff(o)[v] = pre(o')[v]$.
- o and o' *conflict iff* there exists $v \in vars(eff(o)) \cap vars(eff(o'))$ such that $eff(o)[v] \neq eff(o')[v]$.
- o and o' *strongly interfere iff* o disables o', or o' disables o, or o and o' conflict.
- o *weakly interferes* with o' *iff* o disables o' or o and o' conflict[1].
- o and o' are *commutative iff* they do not strongly interfere, and none of them can enable the other.

Definition 16 (active operator). (Based on Wehrle & Helmert 2012 [WH12]). Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task and let s be a state. An operator $o \in \mathcal{O}$ is *active* in s if the following two conditions are satisfied:

1. For all $v \in vars(pre(o))$, there exists a path in $DTG(v)$ from $s[v]$ to $pre(o)[v]$. In addition, if v is goal-related, then there is a path from $pre(o)[v]$ to $s_\star[v]$.
2. For all $v \in vars(eff(o))$, where v is goal-related, there exists a path in $DTG(v)$ from $eff(o)[v]$ to $s_\star[v]$.

We use $act(s)$ to denote the set of active operators in state s.

3.2 Related Work

Theories about partial order reduction have emerged when it has been recognized that concurrency can immensely exacerbate the state explosion problem in the area of model checking. Both model checking and forward-search planning encounter the same fundamental problems that impair the scalability of algorithms in both research areas. The reason behind this similarity is that both areas utilize the construction of the reachability graphs for solving the problem at hand. Intuitively, the techniques developed to tackle the encountered problem in both fields independently can be based on similar theoretical concepts.

[1]In this thesis, the word "interference" refers to strong interference, i.e., "weak" should explicitly be mentioned if we speak of weak interference

The most well-known approaches for partial order reduction in model checking are *stubborn sets* introduced by Valmari 1989 [Val89], *sleep sets* introduced by Godefroid 1996 [God96], and *ample sets* introduced by Peled 1993 [Pel93]. Furthermore, Godefroid proposed *persistent sets* which is a general notion for techniques that perform state-based pruning like stubborn sets and ample sets [GP93; God96].

Later, researchers in planning and search techniques have proposed state space pruning techniques that share some features with the approaches of model checking. The *stratified planning* and *expansion core* methods by Chen at al. 2009; 2009 [CXY09; CY09] are two novel techniques that have been proposed for optimal planning. However, these techniques have had two errors. Firstly, the design of both techniques have encountered some subtle technical flaws that threaten their correctness. These problems have later been corrected by Wehrle and Helmert 2012 [WH12]. Secondly, the theoretical relationships between the new proposed techniques and the original ones from model checking were unclear. This thesis sheds light on the formal relationships between different partial order reduction methods.

Furthermore, there are other contributions in planning to prune parts of the state space and are similar to partial order reduction in their functionality [HG00; HB14]. A pruning method called *move pruning* has been developed by Burch and Holte to reduce the number of explored redundant states by tree search algorithms like IDA* [Kor85]. Move pruning follows an approach introduced by Taylor and Korf 1993 [TK93] for state space pruning. A pruning technique called *commutativity pruning* has been proposed by Haslum and Geffner [HG00], and it is a restricted form of move pruning.

Coles and Coles 2010 [CC10] proposed a pruning technique called *tunneling* which follows the idea of *tunnel macros* from Sokoban [JS01]. Inspired by this work, Nissim et al. 2012 [NAB12] have proposed a generalization of tunnel macros. Their technique is described as *partition-based pruning technique* because it depends on partitioning the set of operators to make pruning decisions.

The idea of *active operators* pruning has been first introduced by Chen and Yao 2009 [CY09] and further investigated by Wehrle and Helmert 2012 [WH12] for excluding some operators that do not belong to any plan. Later, Wehrle and Helmert 2014 [WH14] compared several definitions of strong stubborn sets and came up with a notion of *generalized strong stubborn sets* that unifies earlier definitions of strong stubborn sets.

Bounded intention planning is a pruning technique proposed by Wolfe and Russell 2011 [WR11] for optimal unary SAS$^+$ planning tasks. This technique has been classified, by the authors, as a variant of stubborn sets proposed by Valmari 1989 [Val89] without proving their claim. Sievers et al. 2014 [SWH14] have shown that bounded intention planning is indeed a variant of stubborn sets.

Another pruning technique orthogonal to partial order reduction, called *symmetry elimination*, has been introduced in model checking by Ip and Dill 1996 [ID96]. Symmetry elimination prunes some behaviors given alternative ones which might not lead to the same final outcomes, but the differences between final outcomes are not critical for solving the problem at hand. For example, given a robot with two grippers (right and left), whose task is to move an object from room A to room B, the state in which the robot holds the object with the *right* gripper is obviously different from the state in which she holds it with the *left* gripper. However, these states are symmetrical since the choice between the two gripper does not play a role in achieving the goal (i.e., having the object moved from room A to room B). The first work that investigated symmetries for planning was done by Fox and Long; Fox and Long 1999; 2002 [FL99; FL02]. Recently, symmetry elimination has been thoroughly investigated for classical planning [DKS12; DKS13; Weh+15; Sie+15a; Sie+15b; Shl+15; DHK15] and FOND planning [WWK16].

The combination of partial order reduction and symmetry elimination started in the area of model checking [EJP97; BS15].
The work by Wehrle et al. 2015 [Weh+15] proposed two integration ideas of partial order reduction and symmetry elimination for optimal classical planning, where strong stubborn sets for planning has been used as a partial order reduction technique [Alk+12]. The first idea is the straightforward application of strong stubborn sets to the computation of the space computed by symmetry elimination. The second idea is to derive a new notion of *symmetrical strong stubborn sets* which is a tighter integration of both techniques.

Chapter 4

Stubborn Sets for Classical Planning

Stubborn sets are a key notion that has been introduced by Valmari to describe a group of methods which adhere to a common theory with some differences in their definitions, implementations, and pruning power [Val89]. These methods are partial order reduction techniques that are state-dependent, i.e., are specialized for pruning states from the explored space of a search algorithm. Such pruned states may be unique in the state space, however, they are guaranteed to be uncritical for reaching some desired final states. Stubborn sets methods were originally designed to alleviate the state explosion problem in Petri Nets. Typically, stubborn sets are used when the search algorithm cares only about some final states (e.g., deadlocks), but is indifferent about the intermediate states being explored.
In the context of planning, we are concerned about reaching a goal state, which can be considered as a final state because the search does not need to continue beyond it. Roughly speaking, even if a pruned state leads to a goal state, there must be at least one alternative preserved state from which it is assured that the same goal state is reachable.

In this chapter, we investigate how two variants of stubborn sets can be used to perform partial order reduction in the context of optimal planning. Furthermore, the theoretical relationship between stubborn sets and other partial order reduction techniques from previous work is discussed.

4.1 Variants of Stubborn Sets

The notion of stubborn sets exploits the independence (the opposite of interference) between transitions to reduce the number of executed transitions in every state while maintaining the reachability of terminal states. Independent transitions have the property that by applying them in any order, the same final state is reached. The two methods that have been proposed for Petri Nets are *strong stubborn sets* and *weak stubborn sets*. Valmari introduced the former as a computationally more efficient variant but has less pruning power than the latter [Val89].

Next, we will introduce some necessary definitions, then the definitions of strong and weak stubborn sets.

Definition 17 (disjunctive action landmark). A *disjunctive action landmark* for a partial state t in state s, is a set of operators L_s^t such that every operator sequence, which starts in s and leads to a state s' with $s' \models t$, must contain some operator from L_s^t [HD09; Alk+12].

Definition 18 (necessary enabling set). A *necessary enabling set* for operator o in state s is a set of operators $\mathcal{N}_s^o$ such that every operator sequence, which starts in s and includes o, must contain some operator from $\mathcal{N}_s^o$ before the first occurrence of o in the sequence [God96; WH12].

Alternatively, we can say that a necessary enabling set for a inapplicable operator o in state s is a disjunctive action landmark for $pre(o)$ in s, but we prefer to explicitly speak of necessary enabling sets for chronological reasons, since they have been first introduced by Godefroid [God96]. In addition, it is easier for the reader to distinguish between operator enablers and goal achievers when this separation between necessary enabling sets and disjunctive action landmarks is made.

4.1.1 Strong Stubborn Sets

Valmari has introduced strong stubborn sets as the simplest among the group of methods based on the stubborn sets theory. Next, the definition of strong stubborn sets and an algorithm to compute them for SAS$^+$ planning are presented.

Definition 19 (strong interference relation). Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task. Furthermore, let o and o' be operators in $\mathcal{O}$. The *strong interference relation*, written as $\mathcal{I}_s$, is a binary relation on $\mathcal{O}$ ($\mathcal{I}_s \subseteq \mathcal{O} \times \mathcal{O}$) such that $(o, o') \in \mathcal{I}_s$ if o and o' strongly interfere. We write $\mathcal{I}_s(o)$ to denote the set of operators with which o interferes, i.e., $o' \in \mathcal{I}_s(o)$ *iff* $(o, o') \in \mathcal{I}_s$.

Definition 20 (strong stubborn sets). Given a state s, a *strong stubborn set* for s is a set of operators T_s such that:

1. For each $o \in T_s \cap app(s)$, we have $\mathcal{I}_s(o) \subseteq T_s$.
2. For each $o \in T_s \setminus app(s)$, we have $\mathcal{N}_s^o \subseteq T_s$ for some necessary enabling set $\mathcal{N}_s^o$.
3. $L_s^{s_\star} \subseteq T_s$ for some disjunctive action landmark $L_s^{s_\star}$.

Algorithm 1 is an iterative way of implementing stubborn sets. First, the stubborn set is initialized by a disjunctive action landmark for the goal (line 1). Then, the iteration includes the interfering operators with each operator o in the stubborn set T_s if $o \in app(s)$ (line 5); otherwise, it adds a necessary enabling set for o in s (line 7). The algorithm terminates when no more operators can be added to T_s (line 8).
Practically, a disjunctive action landmark in state s is computed as follows:

1. Select $v \in vars(s_\star)$ s.t. $s_\star[v] = d$ and $s[v] \neq d$.

Algorithm 1 Strong stubborn set for state s

```
1: $T_s \leftarrow L_s^{s_\star}$
2: repeat
3:     for $o \in T_s$ do
4:         if $o \in app(s)$ then
5:             $T_s \leftarrow T_s \cup \mathcal{I}_s(o)$
6:         else
7:             $T_s \leftarrow T_s \cup \mathcal{N}_s^o$
8: until $T_s$ is stable
9: return $T_s$
```

2. $L_s^{s_\star} := \{o \mid v \in vars(\mathit{eff}(o)) \wedge \mathit{eff}(o)[v] = d\}$.

In words, the choice of a disjunctive action landmark is done by choosing an unsatisfied goal fact in s, and adding the operators that achieve that fact to $L_s^{s_\star}$. Similarly, a necessary enabling set for an operator o in state s is computed as follows:

1. Select $v \in vars(pre(o))$ s.t. $pre[v] = d$ and $s[v] \neq d$.
2. $\mathcal{N}_s^o := \{o' \mid v \in vars(\mathit{eff}(o')) \wedge \mathit{eff}(o)[v] = d\}$.

Like a disjunctive action landmark, a necessary enabling set is computed by selecting an unsatisfied precondition fact, and adding the operators that achieve that fact to $\mathcal{N}_s^o$.

Example 1. Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task, where

- $\mathcal{V} = \{a, b\}$
- $\mathcal{O} = \{o_1, o_2\}$, where
  - $pre(o_1) = \{a \mapsto 0\}$, $\mathit{eff}(o_1) = \{a \mapsto 1\}$
  - $pre(o_2) = \{b \mapsto 0\}$, $\mathit{eff}(o_2) = \{b \mapsto 1\}$
- $s_0 = \{a \mapsto 0, b \mapsto 0\}$
- $s_\star = \{a \mapsto 1, b \mapsto 1\}$

A valid strong stubborn set for state s_0 is $T_{s_0} = \{o_1\}$: Variable a is a goal-related variable whose goal value is not satisfied by s_0, o_1 achieves the goal value of a, o_1 is applicable in s_0, and o_1 does not strongly interfere with o_2, i.e., $o_2 \notin \mathcal{I}_s(o_1)$. This means that o_2 is not applied in s_0, thereby pruning state $o_2(s_0)$ as shown in Figure 4.1.

4.1.2 Weak Stubborn Sets

Weak stubborn sets have been proposed by Valmari as a state reduction technique in addition to strong stubborn sets [Val89]. Weak stubborn sets exclude some operators that disable an operator already included in the set; hence, they can have, at least theoretically, more pruning power than strong stubborn sets. However, the conditions stated by Valmari to compute weak stubborn sets in the

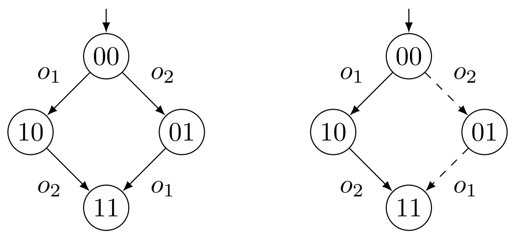


Figure 4.1: Original state space (left) and reduced state space with strong stubborn set (right).

context of Petri nets are rather complicated and difficult to implement. Next, we will see that, in the context of classical planning, the definition of weak stubborn sets is straightforward and does not incur an extra computational effort when defined for syntactic operators of planning tasks.

Definition 21 (weak interference relation). Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task. The *weak interference relation* is a binary relation on $\mathcal{O}$ ($\mathcal{I}_w \subseteq \mathcal{O} \times \mathcal{O}$) such that $(o, o') \in \mathcal{I}_w$ *iff* o weakly interferes with o' . We write $\mathcal{I}_w(o)$ to denote the set of operators where $o' \in \mathcal{I}_w(o)$ *iff* $(o, o') \in \mathcal{I}_w$.

Note that, in contrast to strong interference, weak interference is not a symmetric relation. The weak stubborn algorithm follows the same rules used by strong stubborn algorithm but considers $\mathcal{I}_w$ instead of $\mathcal{I}_s$.

Definition 22. (weak stubborn sets) Given a state s, a *weak stubborn set* for s is a set of operators T_s^w such that:

1. For each $o \in T_s^w \cap app(s)$, we have $\mathcal{I}_w(o) \subseteq T_s$.
2. For each $o \in T_s^w \setminus app(s)$, we have $\mathcal{N}_s^o \subseteq T_s^w$ for some necessary enabling set $\mathcal{N}_s^o$.
3. $L_s^{s_\star} \subseteq T_s$ for some disjunctive action landmark $L_s^{s_\star}$.

The definition of weak stubborn sets is obtained by dropping one of the conditions needed for the computation of the strong interference relation $\mathcal{I}_s$. Thus, no overhead is incurred for the computation of weak stubborn sets, and furthermore, they can prune more states than what strong stubborn sets prune. Weak stubborn sets can be computed using Algorithm 1 by replacing $\mathcal{I}_s(o)$ with $\mathcal{I}_w(o)$ and T_s with T_s^w.

The following example shows that weak stubborn sets can prune *exponentially more* states than strong stubborn sets.

Example 2. Let $n > 1$ and $\Pi_n = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task, where

- $\mathcal{V} = \{a, b, c, v_1, \ldots, v_n\}$
- $\mathcal{O} = \{o_b, o_c, o_1, \ldots, o_n, \bar{o_1}, \ldots, \bar{o_n}\}$, where
  - $pre(o_b) = \{a \mapsto 0\}$, $eff(o_1) = \{b \mapsto 1\}$
  - $pre(o_c) = \{a \mapsto 0\}$, $eff(o_2) = \{c \mapsto 1\}$

  - $pre(o_i) = \{\top\}$, $\mathit{eff}(o_i) = \{a \mapsto 1, v_i \mapsto 1\}$
  - $pre(\bar{o_i}) = \{v_i \mapsto 1\}$, $\mathit{eff}(\bar{o_i}) = \{a \mapsto 0, v_i \mapsto 0\}$

- $s_0 = \{a \mapsto 0, b \mapsto 0, c \mapsto 0, v_1 \mapsto 0, \dots, v_n \mapsto 0\}$
- $s_\star = \{b \mapsto 1, c \mapsto 1\}$

Before discussing how pruning by weak stubborn sets differs from pruning by strong stubborn sets, let's have a closer look at the characteristics of this planning task:
Π_n has $O(2^n)$ reachable states: There are variables $v_1, \dots, v_n$ with values $d \in \{0,1\}$, where all the combinations of facts $\langle v_i, d\rangle$ are reachable from s_0, i.e., every valuation from $v_1 \mapsto 0, \dots, v_n \mapsto 0$ until $v_1 \mapsto 1, \dots, v_n \mapsto 1$. Furthermore, Π_n has no dead ends because o_b and o_c are active in all reachable states (either $a \mapsto 0$ is true, or it can be achieved by applying an operator $\bar{o_i}$). In other words, there is a goal state reachable from every reachable state.

Weak stubborn sets. There are two possible minimal weak stubborn sets at state s_0: Either $\{o_b\}$ or $\{o_c\}$, depending on which variable (b or c) is chosen for including a disjunctive action landmark for $s_\star$ in s_0. Operators $o_1, \dots, o_n$ are applicable in s_0, but none of them is included in a weak stubborn set because neither o_b nor o_c weakly interferes with any of them, i.e., $o_i \notin \mathcal{I}_w(o_b)$ and $o_i \notin \mathcal{I}_w(o_c)$ for all $i \in \{1, \dots, n\}$. Note also that o_b and o_c are non-interfering with each other. Therefore, the reachable state space consists of 4 states, namely:

1. s_0
2. $o_b(s_0) = \{a \mapsto 0, b \mapsto 1, c \mapsto 0, v_1 \mapsto 0, \dots, v_n \mapsto 0\}$
3. $o_c(s_0) = \{a \mapsto 0, b \mapsto 0, c \mapsto 1, v_1 \mapsto 0, \dots, v_n \mapsto 0\}$
4. $o_c o_b(s_0) = o_b o_c(s_0) = \{a \mapsto 0, b \mapsto 1, c \mapsto 1, v_1 \mapsto 0, \dots, v_n \mapsto 0\}$

Depending on the choice of variable for computing a disjunctive action landmark, either $o_b(s_0)$ or $o_c(s_0)$ is pruned, which leads to generating only 3 states using weak stubborn sets.

Strong stubborn sets. We notice that any strong stubborn set in s_0 will contain *all* operators in $\mathcal{O}$. To illustrate this, assume that variable b is selected to compute a disjunctive action landmark in s_0, then o_b is included in the strong stubborn set. Since $o_b \in app(s_0)$ and all $o_i \in \mathcal{I}_s(o)$ (o_i operators disable o_b) for all $i \in \{1, \dots, n\}$, this means that all o_i will be included in the strong stubborn set. Now that all $o_i \in app(s_0)$ and they disable o_c, the latter will be in the strong stubborn set as well. Finally, operators o_i and $(\bar{o_i})$ conflict, which adds all $(\bar{o_i})$ to the strong stubborn set.
For other non-goal reachable states s from s_0, we distinguish two cases:

- $s[a] = 1$: $o_b \notin app(s)$ (the same holds for o_c), and operators $\bar{o_i}$ are included in the stubborn set as a necessary enabling set for o_b in s, for all $i \in \{1, \dots, n\}$. At least one operator $\bar{o_k} \in \{\bar{o_1}, \dots, \bar{o_n}\}$ is applicable in s, i.e., $s[v_k] = 1$, because at least one corresponding o_k led to state s; otherwise,

$s[a] \neq 1$. Next, all operators o_i are included in the stubborn sets because they conflict with $\bar{o_k}$. Finally o_c is added to the stubborn set since it is disabled by operators o_i, which are applicable in s. As a result, every operator is included in the stubborn set.

- $s[a] = 0$: $o_b \in app(s)$ (the same holds for o_c), and the computation of the stubborn sets is exactly like in state s_0, which includes all operators in the stubborn set.

For this example, we conclude that the reachable state space using strong stubborn sets has a size exponential in the number of variables ($O(2^n)$), whereas the size of the reachable state space is 4 when weak stubborn sets are being considered (only 3 are actually generated).

4.2 Stubborn Sets for Optimal Classical Planning

We show now that stubborn sets (strong and weak) can be utilized by an optimal search algorithm (e.g., A* guided by an admissible heuristic), such that the combination preserves completeness and optimality. In the following, A$^*_{sss}$ denotes A* using strong stubborn sets, and A$^*_{wss}$ denotes A* using weak stubborn sets.

Theorem 1. *A^*_{wss} is completeness and optimality preserving.*

Proof. Let T^w_s a weak stubborn set computed according to Definition 22 for state s. Let $T^w_{app(s)} \subseteq T^w_s$ be the set of applicable operators included in T^w_s. We show that for all states s from which there exists an optimal plan p with $|p| > 0$, $T^w_{app(s)}$ contains an operator that starts such a plan. Let $p = o_1 \dots o_n$ is an optimal plan that starts in s. Since T^w_s contains a disjunctive action landmark, at least one operator o_i must be included in T^w_s. Let o_k be the operator with the smallest index in p that is contained in T^w_s, i.e., $o_k \in T^w_s$ and $\{o_1, \dots, o_{k-1}\} \cap T^w_s = \emptyset$. We have:

1. $o_k \in app(s)$; otherwise a necessary enabling set $\mathcal{N}^{o_k}_s$ has to be included in T^w_s by rule 2 of definition 22, which means at least one operator from $\mathcal{N}^{o_k}_s$ must be applied before o_k in p to enable o_k, and this contradicts the assumption that o_k has the smallest index.

2. o_k does not weakly interfere with any o_i with $i \in \{1, \dots, k-1\}$, i.e., o_k does not disable o_i, and o_k and o_i do not conflict; otherwise, at least one operator from $o_1, \dots, o_{k-1}$ has to be contained in T^w_s by rule 1 of definition 22, again contradicting the assumption.

Therefore, we can obtain an optimal plan by shifting o_k to the front, i.e., we have an alternative permutation of the operators $o_1, \dots, o_k$ that starts with o_k and has the same cost as p: $o_k, o_1, \dots, o_{k-1}, o_{k+1}, \dots, o_n$.

□

Corollary 1. *Strong stubborn sets are completeness and optimality preserving.*

Practically, the implementations of stubborn sets consists of two steps: A *preprocessing step* in which the interference relation ($\mathcal{I}_s$ or $\mathcal{I}_w$) is precomputed for all operators of the planning task and cached in memory, and a *runtime step* where stubborn sets are computed during the search process for generated states by looking up the interference relation.

4.3 The Pruning Power of Stubborn Sets

An in-depth analysis has shown that stubborn sets[1], in their original form, strictly dominate previous state pruning techniques like *expansion core*. The expansion core method has been proposed as a state reduction technique [CY09]. However, the original version was marred with a technical flaw that has been discovered and corrected later by Wehrle & Helmert [WH12]. Previous work has shown that expansion core is an instantiation of the strong stubborn sets method [WH12]. Although both stubborn sets and expansion core are state reduction techniques, there are fundamental differences in the way both methods are presented. While stubborn sets are defined in an operator-based fashion, expansion core is defined with respect to variable dependencies (i.e. interference). For this reason, the relationship between both techniques is not straightforward to be inferred. Therefore, both methods have been represented as sets of rules to show the strict dominance of stubborn sets over expansion core.

This section summarizes the process used for showing the relationship between expansion core and stubborn sets. The detailed theoretical results exists in [Weh+13]. First, we briefly describe the expansion core method based on Wehrle & Helmert [WH12].

Expansion Core. Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task and let s be a state. Given variables v and $v' \in \mathcal{V}$, v is a *potential precondition* of v' in s if there is an operator $o \in act(s)$ such that $v \in vars(pre(o))$, o is v-applicable, and $v' \in vars(eff(o))$. v is a *potential dependent* of v' in s if there is an operator $o \in act(s)$ such that $v \in vars(pre(o))$, o is v-applicable, and $v' \in vars(eff(o))$. The *potential dependency graph* $PDG(s)$ is a directed graph $\langle V, E \rangle$, where $V = \mathcal{V}$. There is an edge from variable v to v' if v is a potential precondition of v' in s, or v is a potential dependent of v' in s, or there exists an operator $o \in act(s)$ such that $v \in vars(eff(o))$ and $v' \in vars(eff(o))$. Given a state s, the expansion core algorithm is computed a follows:

1. Compute the potential dependency graph $PDG(s)$ in s.
2. Select a goal-related state variable $v_\star$ that is unsatisfied in s (i.e., $s_\star[v_\star] \neq s[v_\star]$),
3. Compute a *dependency closure* $dc(s)$, which is a set of variables, such that $v_\star \in dc(s)$, and for variables v and v', if $v \in dc(s)$ and $PDG(s)$ contains an edge from v to v', then $v' \in dc(s)$.
4. Select a subset of $app(s)$, whose operators modify a variable in $dc(s)$, to be applied in s.

[1]By stubborn sets we mean the strong variant.

Sets of rules. For the purpose of the analysis, several sets of rules have been introduced to show the strict dominance of strong stubborn sets over expansion core (the detailed rules are stated in [Weh+13]):

- The expansion core is first reformulated as a set of monotonic rules that can be applied until a fixed point is reached. For a given state s, $EC(s)$ is the result of applying these rules. The first rule initializes $dc(s)$, the next three rules adds variables to $dc(s)$, and the fifth rule computes $EC(s)$. The final result is identical to the result of the original formulation of the algorithm when the same goal-related variables is selected for initialization.

- The next set of rule is obtained by computing the expansion core in an operator-based fashion, i.e., operators added to $EC(s)$ without building $dc(s)$ first. This variant of expansion core is called *operator-based expansion core* (OBEC), and it is an intermediate step between the original version of expansion core and strong stubborn sets. It has been shown that the pruning power of OBEC is *always at least as large as* the pruning power of EC. This fact has been concluded by comparing the previous set of rules with the ones of OBEC.

- Finally, an instantiation of strong stubborn sets has been proposed as a set of rules. This variant is called SSS-EC, and it is computed in a way that reflects definition 20, with special choices for disjunctive action landmarks and necessary enabling sets that achieve the dominance of SSS-EC over OBEC. The final and most important theorem in this analysis has shown that this particular instantiation of strong stubborn sets strictly dominates the expansion core in terms of pruning power. Furthermore, there exist families of planning tasks where strong stubborn sets explore exponentially less states than what expansion core explores [Weh+13].

Chapter 5

Stubborn Sets for FOND Planning

Planning agents can face situations where they are uncertain about the outcomes of executing their actions in the world. *Fully observable nondeterministic* (FOND) planning is a planning form used for modeling uncertainty by nondeterminism in actions of planning problems [Cim+03; Lev05; Kut+08; KE09; Fu+11]. A nondeterministic action has several outcomes, where only one of them is applied in a given state when the action is executed. However, it is unknown a priori which outcome is going to influence the successor state. Therefore, the state space of the nondeterministic model needs to represent all possible successor states that are obtained by different outcomes of nondeterministic actions. Like classical planning, FOND planning faces the exponential growth in the state space induced by the concise description of a nondeterministic planning problem. Therefore, pruning techniques might help reduce the size of the explored state space of a FOND planning problem. In this chapter, we establish the theoretical basis for a stubborn sets technique for FOND planning problems. Most of the content in this chapter has been published by Winterer et al. 2017 [Win+17].

5.1 Preliminaries

In the following, we state the most important definitions needed in this chapter. We start with the formal definition of *nondeterministic transition systems* used to represent nondeterministic planning tasks[1].

Definition 23 (nondeterministic transition system). A *nondeterministic transition system* $\mathcal{T}$ is a 5-tuple (S, s_0, A, T, G), where

- S is a finite set of *states*,
- $s_0 \in S$ is the *initial state*,
- A is a finite set of transitions (*actions*),

[1] The main difference between the definitions of deterministic transition system (Def. 1) and nondeterministic transition system is the nondeterministic transition relation T.

- $T : S \times A \times S$ is the *nondeterministic transition relation*, and
- $G \subseteq S$ is finite set of *goal states*.

Next, we define nondeterministic operators and FOND planning tasks.

Definition 24 (nondeterministic operator). A nondeterministic operator o is a structure that has a precondition $pre(o)$ and a set of effects $effs(o)$, where $pre(o)$ and $eff^i(o) \in effs(o)$ are partial states. The degree of an operator o, denoted as $deg(o)$, is the number of its nondeterministic effects. If $deg(o) = 1$, i.e., $effs(o) = \{eff^1(o)\}$, then o is deterministic. If o is a deterministic operator, we write $eff(o)$ instead of $eff^1(o)$.

Definition 25 (Fully observable nondeterministic planning task). A fully observable nondeterministic (FOND) planning task is a tuple $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$, where

- $\mathcal{V}$ is a finite set of finite-domain state variables,
- $\mathcal{O}$ is a finite set of *nondeterministic* operators,
- s_0 an initial state, and
- s$_\star$ is a partial state representing the set of goal states.

Definition 26 (all-outcome determinization). The all-outcome determinization of a nondeterministic operator o is $o^{[1]}, \dots, o^{[k]}$, where $k = deg(o)$ and every outcome $o^{[i]}$ has precondition $pre(o)$ and $eff(o^{[i]}) = eff^i(o)$. We use $\mathcal{O}_{det}$ to denote the set of all outcomes of operators in $\mathcal{O}$.

Solutions to FOND planning tasks are called *policies*.

Definition 27 (policy). A *policy* is a mapping $\pi : S \mapsto \mathcal{O} \cup \{\bot\}$, which maps states to operators or π is undefined (i.e., $\pi(s) = \bot$). A policy π is called *weak* if π defines at least one path from the initial state to a goal state when following π. In this case, π is called a *weak plan* for Π. A policy π is *closed* if following π either leads to a goal state, or to a state where the policy is defined. It is *proper* if from every state visited following π, there exists a path to a goal state following π. A policy that is closed and proper is called a *strong cyclic plan* for Π. Furthermore, π is *acyclic* if it does not revisit already visited states. A closed and proper acyclic policy is called a *strong plan* for Π.

Informally, a weak plan is a sequence of operators which may lead from the initial state to a goal state but are not guaranteed to do so [Cim+03]. Weak plans correspond to sequential plans in classical planning. A strong plan guarantees that a goal state is reached, where an upper bound on the number of plan steps exists. In contrast to strong plans, strong cyclic plans reach a goal state after a number of steps under the *fairness* assumption, i.e., there is a nonzero probability that a goal state can be reached when following a strong cyclic plan.

5.2 Pruning Techniques for FOND

Similar to classical planning, FOND planning faces the state explosion problem. Due to this fact, state space pruning techniques, like stubborn sets, are considered as eligible methods to tackle the state explosion problem in nondeterministic worlds. However, pruning techniques cannot be applied in the context of FOND planning in a straightforward way due to the complicated structure of the nondeterministic state space which is represented by AND/OR graphs. In this section, we present the theoretical basis for a stubborn sets method combined with LAO* [HZ01] search in the context of FOND planning. First, we show, by the following example, that using the original stubborn sets definition with the all outcome determinization is unsafe.

Example 3. Consider the following all-outcome determinization $\Pi_{det} = \langle \mathcal{V}, \mathcal{O}_{det}, s_0, s_* \rangle$ of nondeterministic planning task Π with:

- $\mathcal{V} = \{v_1, v_2, v_3\}$,
- $\mathcal{O} = \{o_1, o_2, o_3, o_4, o_5, o_6\}$, where
  - $pre(o_1^{[1]}) = \{v_1 \mapsto 0\}$, $eff(o_1^{[1]}) = \{v_1 \mapsto 1\}$
  - $pre(o_1^{[2]}) = \{v_1 \mapsto 0\}$, $eff(o_1^{[2]}) = \{v_1 \mapsto 2\}$
  - $pre(o_2^{[1]}) = \{v_2 \mapsto 0\}$, $eff(o_2^{[1]}) = \{v_2 \mapsto 1\}$
  - $pre(o_2^{[2]}) = \{v_2 \mapsto 0\}$, $eff(o_2^{[2]}) = \{v_2 \mapsto 2\}$
  - $pre(o_3) = \{v_1 \mapsto 1, v_2 \mapsto 1\}$, $eff(o_3) = \{v_3 \mapsto 1\}$
  - $pre(o_4) = \{v_1 \mapsto 1, v_2 \mapsto 2\}$, $eff(o_4) = \{v_3 \mapsto 1\}$
  - $pre(o_5) = \{v_1 \mapsto 2, v_2 \mapsto 0\}$, $eff(o_5) = \{v_3 \mapsto 1\}$
  - $pre(o_6) = \{\top\}$, $eff(o_6) = \{v_2 \mapsto 0\}$
- $s_0 = \{v_1 \mapsto 0, v_2 \mapsto 0, v_3 \mapsto 0\}$
- $s_\star = \{v_3 \mapsto 1\}$

The set $\{o_3, o_4, o_5\}$ is a disjunctive action landmark for $s_\star$ in s_0 and, therefore, is a candidate to initialize T_{s_0}. As all operators in this set are inapplicable in s_0, a necessary enabling set for each of them needs to be added to T_{s_0}. Assume the computation of stubborn sets chooses $o_2^{[1]}$, $o_2^{[2]}$ and o_6 as necessary enabling sets for o_3, o_4, and o_5 (with respect to variable v_2), respectively. At this point of the computation, T_{s_0} contains o_3, o_4, o_5, o_6 and both outcomes of o_2. o_6 is applicable and mutually conflicts with both outcomes of o_2, which are already in the stubborn set. The computation terminates with $T_{s_0} = \{o_3, o_4, o_5, o_6, o_2^{[1]}, o_2^{[2]}\}$. This means that o_1 is pruned in s_0, leading to a policy with deadends. Figure 5.1 shows that the only possible solution starts by applying o_1 in s_0.

Example 3 shows us that non-interference between operators' outcomes is *not* a sufficient criterion to safely apply the stubborn sets technique in the nondeterministic setting. In the following, we introduce the notion of *attachable* operators and show that it can be utilized to design a safe stubborn sets method for FOND planning.

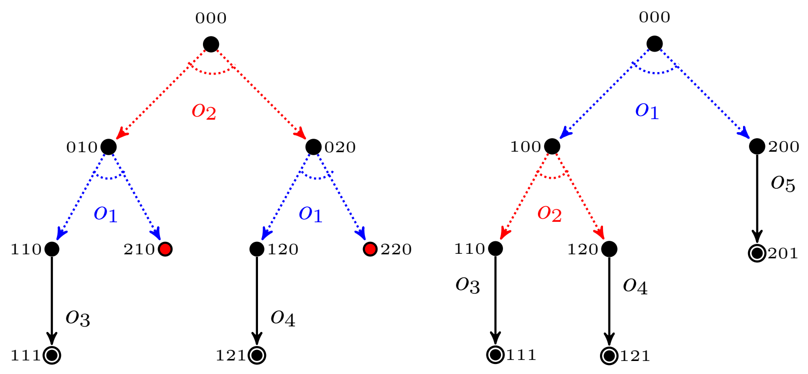


Figure 5.1: Applying o_2 before o_1 in s_0 only leads to a policy (left figure: red nodes are deadends) with dead-ends. The right figure is a strong plan.

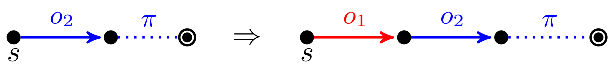


Figure 5.2: Adding o_1 to the front of the operator transforms weak plan $o_2\pi$ into weak plan $o_1o_2\pi$.

Definition 28 (attachable operator). Let o_1 and o_2 be operators in $\mathcal{O}_{det}$, π be an operator sequence, and s be a state. Operator o_1 is *attachable* to o_2 in state s if for every weak plan $o_2\pi$ from s, $o_1o_2\pi$ is a weak plan from s.

Checking the attachability property is computationally intractable. To be precise, this property is state-dependent, i.e., it needs to be checked in each visited state. Moreover, verifying this property amounts to finding two weak plans for s, one that starts with o_2 and another one that starts with o_1. Therefore, it is necessary to come up with an approximation based on a syntactic check of operators. Figure 5.2 shows the idea of attachability: If o_1 is attachable to o_2, then the weak plan $o_2\pi$ from s can be applied in state $o_1(s)$.

Definition 29 (disabled set and negated goal set). Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a FOND planning task and o_1 be an operator in $\mathcal{O}_{det}$.

- The *disabled set* $dis(o_1)$ is defined as the set of operator-variable pairs $(o_2, v) \in \mathcal{O}_{det} \times \mathcal{V}$, such that o_1 disables o_2 on v, i.e., $v \in vars(eff(o_1)) \cap vars(pre(o_2))$ and $eff(o_1)[v] \neq pre(o_2)[v]$.

- The *negated goals set* $neg(o_1)$ is defined as the set of goal variables with which o_1 conflicts, i.e., $eff(o_1)[v] \neq s_\star[v]$.

Definition 30 (syntactic attachable operator). Let o_1 and o_2 be two operators in $\mathcal{O}_{det}$. We say that o_1 is *syntactically attachable* to o_2 if

1. o_1 does not disable o_2,

2. $dis(o_1) \subseteq dis(o_2)$, and

3. $neg(o_1) \subseteq neg(o_2)$.

In the following theorem, we show that syntactic attachability is sufficient to check attachability.

Theorem 2. *Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a FOND planning task, s be a state of Π, and o_1 and o_2 be operators in $\mathcal{O}_{det} \cap app(s)$. If o_1 is syntactically attachable to o_2, then o_1 is attachable to o_2 in s.*

Proof. Let $\pi = o'_1 \dots o'_n$ be a weak plan from $o_2(s)$, where all operators o'_i are in $\mathcal{O}_{det}$. Since o_1 and o_2 are both applicable in s, we know from condition (1) (Definition 30) that o_2 is applicable in $o_1(s)$. To show that the entire operator sequence π is applicable in state $o_2(o_1(s))$, we compare the valuations of state variables in states $o_2(o_1(s))$ and $o_2(s)$. By condition (2), we know that states $o_2(o_1(s))$ and $o_2(s)$ cannot have different valuations on variables that occur in o_1's effect such that these valuations destroy at least one precondition of some operator in π. Therefore, it holds that $o_2(s)[v] = o_2(o_1(s))[v]$ for all $v \in \{v' \mid (o, v') \in dis(o_1)\}$. For all other $v \in vars(\mathit{eff}(o_1)) \setminus \{v' \mid (o, v') \in dis(o_1)\}$ with $o_2(o_1(s))[v] \neq o_2(s)[v]$ we distinguish two cases: Variable $v \notin vars(pre(o'_i))$ for all $i \in \{1, \dots, n\}$ or $v \in vars(pre(o'_i))$ for some $i \in \{1, \dots, n\}$. In the first case, operator sequence π is applicable in $o_2(o_1(s))$. In the second case, $\mathit{eff}(o_1)[v] = pre(o'_i)[v]$, otherwise $(o, v) \in dis(o_1)$ for some $o \in \{o'_1, \cdots, o'_n\}$ contradicting condition (2). Hence, π is applicable in $o_2(o_1(s))$ in this case as well. Finally, we have to show that $o_1 o_2 \pi$ is indeed a weak plan from s. From condition (3), we know that all goal facts that are satisfied in $o_2(s)$ must be satisfied in $o_2(o_1(s))$. Therefore $o_1 o_2 \pi$ is a weak plan from s concluding the proof. $\square$

5.2.1 Weak Interference vs. Attachability

Pruning using the notion of weak interference might result in equivalent weak plans that are permutations of non-interfering operators. On the other hand, attachability allows introducing new operators to the beginning of weak plans such that the resulting sequence is also a weak plan. To understand the design choices of the new pruning technique proposed next, we first clarify the relationship between attachability of operators and weak interference.

At first glance, it might seem to the reader that the notion of attachability subsumes the notion of weak interference. However, as we are going to see by examples, both notions are incomparable, i.e., if operator o_1 is syntactically attachable to operator o_2, this does not imply that o_1 does not weakly interfere with o_2.

Example 4. Consider a FOND planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$, where

- $\mathcal{V} = \{v_1, v_2, v_3, v_4\}$
- $\mathcal{O} = \{o_1, o_2\}$, where
  - $pre(o_1) = \{v_1 \mapsto 0\}$, $\mathit{effs}(o_1) = \{\{v_2 \mapsto 1, v_4 \mapsto 1\}\}$
  - $pre(o_2) = \{v_1 \mapsto 0\}$, $\mathit{effs}(o_2) = \{\{v_3 \mapsto 1, v_4 \mapsto 2\}\}$
- $s_0 = \{v_1 \mapsto 0, v_2 \mapsto 0, v_3 \mapsto 0, v_4 \mapsto 0\}$
- $s_\star = \{v_2 \mapsto 1, v_3 \mapsto 1\}$

We observe that operators o_1 and o_2 are mutually syntactically attachable to each other: $dis(o_1) \subseteq dis(o_2) = \emptyset$, and $neg(o_1) \subseteq neg(o_2) = \emptyset$. However, they also mutually weakly interfere with each other because they conflict on variable v_4.

In addition, attachability gives rise to a new concept of state space pruning: We obtain a stubborn set based solely on attachability. For simplicity, we assume classical planning tasks in the following definition.

Definition 31 (attachability-based stubborn sets). Given a state s, an *attachability-based stubborn set* for s is a set of operators T_a such that:

1. For each $o \in T_a \cap app(s)$, T_a contains all operators to which o is *not syntactically attachable.*
2. For each $o \in T_a \setminus app(s)$: $\mathcal{N}_s^o \subseteq T_a$ for some necessary enabling set $\mathcal{N}_s^o$.
3. $L_s^{s_\star} \subseteq T_s$ for some disjunctive action landmark $L_s^{s_\star}$.

If we review Definitions 20 and 22 from Chapter 4, we see that Definition 31 differs from them only in rule 1, which considers attachability instead of strong or weak interference.

Theorem 3. *Attachability-based stubborn sets technique is completeness preserving.*

Proof. Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task. Let o_1 and o_2 be operators in $\mathcal{O}$ such that o_1 is syntactically attachable to o_2. From Definition 30, we know that o_1 does not disable o_2, which is the first condition to check whether o_1 weakly interferes with o_2.

Now, assume that o_1 and o_2 conflict on some variable $v \in vars(eff(o_1)) \cap vars(eff(o_2))$, i.e., $eff[o_1](v) \neq eff[o_2](v)$. We show that conflicting on v is not critical for reaching a goal state by performing syntactic analysis. Assume that $v \in vars(pre(o'))$ for some $o' \in \mathcal{O}$ such that o_2 can enable o' on v and o_1 disables o' on v. This means that $(o', v) \in dis(o_1)$ and $(o', v) \notin dis(o_2)$, contradicting the second condition of attachability: $dis(o_1) \subseteq dis(o_2)$. Now assume that v is a goal-related variable such that o_1 sets it to a value other than its goal value, and o_2 sets v to its goal value. Again, this implies $v \in neg(o_1)$ and $v \notin neg(o_2)$ contradicting the third condition of attachability: $neg(o_1) \subseteq neg(o_2)$. Hence, no such o' can exist and the claim of the theorem holds. $\square$

It is worthwhile to mention that, due to the nature of attachability, it can lead to suboptimal weak plans. We further show, by the following examples, that stubborn sets based on weak interference and stubborn sets based on attachability are two different pruning techniques such that none of them dominates the other with respect to pruning power.

Example 5. Consider a FOND planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$, where

- $\mathcal{V} = \{v_1, v_2, v_3\}$
- $\mathcal{O} = \{o_1, o_2, o_3\}$
  - $pre(o_1) = \{\top\}$, $eff(o_1) = \{\{v_1 \mapsto 2, v_2 \mapsto 1\}\}$
  - $pre(o_2) = \{\top\}$, $eff(o_2) = \{\{v_1 \mapsto 1\}\}$

  - $pre(o_3) = \{v_1 \mapsto 1, v_2 \mapsto 1\}$, $\mathit{eff}(o_3) = \{\{v_3 \mapsto 1\}\}$

- $s_0 = \{v_1 \mapsto 0, v_2 \mapsto 0, v_3 \mapsto 0\}$
- $s_\star = \{v_1 \mapsto 1, v_2 \mapsto 1, v_3 \mapsto 1\}$

Let T_a denote a stubborn set that uses only syntactic attachability. Since $\{o_2\}$ is a disjunctive action landmark for $s_\star$ in s_0, we can use it to initialize the stubborn set: $T_a(s_0) = \{o_2\}$. Because $o_2 \in app(s_0)$, we need to add the operators to which o_2 is not attachable. Since $dis(o_2) = \emptyset$ and $neg(o_2) = \emptyset$, o_2 is attachable to both o_1 and o_3. This means that the computation will terminate on $T_a(s_0) = \{o_2\}$. On the other hand, if we compute T_s, the stubborn set based on weak interference, the result is different. We initialize $T_s(s_0)$ with o_2 as before. o_2 is applicable, which means we need to add operators with which o_2 weakly interferes. Obviously, o_2 weakly interferes with o_1 because they conflict on variable v_1. Therefore, o_1 is added to $T_s(s_0)$. Also, o_1 is applicable and it disables o_3, hence o_3 is added to $T_s(s_0)$, which means all operators are applied in s_0 ($T_s(s_0) = \{o_1, o_2, o_3\}$).

Furthermore, each method can result in a different plan: Depending on the search algorithm, using only syntactic attachability might result in plan $o_2 o_1 o_2 o_3$, while using only weak interference might lead to plan $o_1 o_2 o_3$ which is optimal.

Example 6. Consider a FOND planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$, where

- $\mathcal{V} = \{v_1, v_2\}$
- $\mathcal{O} = \{o_1, o_2, o_3\}$, where
  - $pre(o_1) = \{\top\}$, $\mathit{eff}(o_1) = \{\{v_1 \mapsto 1\}\}$
  - $pre(o_2) = \{\top\}$, $\mathit{eff}(o_2) = \{\{v_2 \mapsto 1\}\}$
  - $pre(o_3) = \{v_1 \mapsto 0\}$, $\mathit{eff}(o_3) = \{\{v_2 \mapsto 1\}\}$
- $s_0 = \{v_1 \mapsto 0, v_2 \mapsto 0\}$
- $s_\star = \{v_1 \mapsto 1, v_2 \mapsto 1\}$

We compute $T_a(s_0)$: The set $\{o_1\}$ is a disjunctive action landmark in s_0, so we add it to $T_a(s_0)$. Since o_1 is applicable, we add operators to which it is not attachable. Because $dis(o_1) = \{o_3\} \not\subseteq dis(o_2) = \emptyset$, o_1 is not attachable to o_2 and therefore o_2 is added to $T_a(s_0)$. In addition, o_3 is added because o_1 disables o_3 and hence not attachable to it, ending in $T_a(s_0) = \{o_1, o_2, o_3\}$. Using weak interference: We add o_1 to $T_s(s_0)$ as a disjunctive action landmark. o_1 is applicable so we need to add the operators with which it weakly interferes. We add o_3 only because o_1 disables o_3, but we do not add o_2 because neither o_1 nor o_3 weakly interferes with o_2. Therefore, $T_s(s_0) = \{o_1, o_3\}$.

We conclude this section with the following corollary, summarizing our observations.

Corollary 2. *Stubborn sets based on weak interference and stubborn sets based on attachability are incomparable in terms of pruning power.*

5.2.2 Nondeterministic Weak Stubborn Sets

While attachability is a sufficient criterion for stubborn sets pruning, it could lead to unnecessary state explorations (e.g., see plan length in Example 5). We propose a stubborn set method based on a combination of attachability and weak interference for FOND planning. The definition of a disjunctive action landmark can be extended to FOND planning in a straightforward way. In the context of FOND planning, a *disjunctive action landmark* for a partial state s' in a state s is a set of nondeterministic operators with the property that some outcome of an operator from this set occurs on every path from s to any state that satisfies s'. A *necessary enabling set* for an operator o in s is a disjunctive action landmark for $pre(o)$ in s. The following definitions extend syntactic attachability and weak interference to nondeterministic operators.

Definition 32 (operator accordance). Let o_1 and o_2 be operators in $\mathcal{O}$. We say that o_1 *accords* with o_2 if outcome $o_1^{[i]}$ is syntactically attachable to outcome $o_2^{[j]}$ for all $i \in \{1, \ldots, deg(o_1)\}$ and all $j \in \{1, \ldots, deg(o_2)\}$.

Definition 33 (weak interference of nondeterministic operators). Let o_1 and o_2 be operators in $\mathcal{O}$. We say that o_1 *weakly interferes* with o_2 if at least one outcome $o_1^{[i]}$ weakly interferes with at least $o_2^{[j]}$ for some $i \in \{1, \ldots, deg(o_1)\}$ and some $j \in \{1, \ldots, deg(o_2)\}$.

Based on these definitions, we define now weak stubborn sets for FOND planning.

Definition 34 (nondeterministic weak stubborn sets). Let Π be a *nondeterministic* planning task, and s be a state. A set $T_s \subseteq \mathcal{O}$ is a *nondeterministic weak stubborn set* (NWSS) in s if the following conditions hold:

1. T_s contains a disjunctive action landmark for $s_\star$ in s.
2. For each operator $o \in T_s$ with $o \notin app(s)$, T_s contains a necessary enabling set for o in s.
3. For each operator $o \in T_s$ with $o \in app(s)$, T_s contains all nondeterministic operators o' ($deg(o') > 1$) with which o does not accord.
4. For each operator $o \in T_s$ with $o \in app(s)$, T_s contains all operators with which o weakly interferes.

Nondeterministic weak stubborn sets can be seen as a hybrid notion combining attachability with weak interference. Points (1) and (2) of Definition 34 are equivalent to points (3) and (2) in Definition 22, respectively[2]. Accordance, in point (3), is a necessary property for changing the application order of nondeterministic operators in AND/OR graphs. However, in some cases, pruning can be performed by merely permuting operator sequences (e.g., if the involved operators are deterministic) without the need to attach extra operators, which cannot be guaranteed by accordance alone. For this reason, we make the distinction between deterministic operators and operators of degree at least two, and use *only* weak interference when processing deterministic operators (point

[2]The order of rules in a definition of stubborn sets is not important.

4). This means, if a planning task has only deterministic operators, then only points (1), (2), and (4) of Definition 34 are relevant.

In the following theorem, we see that pruning with nondeterministic weak stubborn sets is a safe option for FOND planning.

Theorem 4. *Restricting the successor generation to a NWSS in every state is completeness-preserving for strong cyclic planning.*

Proof. Let s be a state from which a strong cyclic plan exists and let π be such a plan. Let T_s be a *nondeterministic weak stubborn set* for s. We show that either (i) the operator $\pi(s)$ is in T_s, or (ii) there exists $o \in T_s$ such that $o \in \mathit{app}(s)$ and for each state $s' \in o(s)$, there exists a strong cyclic plan from s'.

Let $\pi' = o_1 \dots o_n$ be any sequence of operators from π that defines an acyclic weak plan from s. T_s contains a disjunctive action landmark for $s_\star$ in s, and thus it contains an operator from π'. Let o_i be such an operator with smallest index. Then $o_i \in \mathit{app}(s)$ (otherwise its necessary enabling set would be contained in T_s, mandating an operator from the necessary enabling set to appear in π' before o_i). If $i = 1$, we are done; otherwise, we distinguish two cases: Operator o_i is a strictly nondeterministic operator or operator o_i is a deterministic operator. In the first case, we observe: For every outcome $o_i^{[j]}$, $o_i^{[j]}\pi'$ is a weak plan as $o_i^{[j]}$ accords with every operator of smaller index within π'. For the second case, we see that $o_i^{[j]}o_1 \dots o_{i-1}o_{i+1}o_n$ is a weak plan from s. As $o_i^{[j]}$ is an arbitrary outcome of o_i, the resulting structure is a strong cyclic plan. □

We also provide a nondeterministic *strong* variant of stubborn set (NSSS) by modifying Definition 34 as follows: In point 3, all operators that do not *mutually* accord with o are added to the stubborn set. In point 4, all operators that *strongly* interfere with o are added to the stubborn set.

Corollary 3. *NSSS inherit the completeness property from NWSS.*

Proof. The definition of NSSS subsumes the definition of NWSS; hence, the proof of Theorem 4 shows also that NSSS is a completeness-preserving pruning technique. □

Example 7. Consider the following nondeterministic planning task $\mathit{\Pi} = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$, where

- $\mathcal{V} = \{v_1, v_2, v_3\}$
- $\mathcal{O} = \{o_1, o_2, o_3, o_4, o_5, o_6, o_7\}$, where
  - $\mathit{pre}(o_1) = \{v_1 \mapsto 0\}$, $\mathit{effs}(o_1) = \{\{v_1 \mapsto 1\}, \{v_1 \mapsto 2\}\}$
  - $\mathit{pre}(o_2) = \{v_2 \mapsto 0\}$, $\mathit{effs}(o_2) = \{\{v_2 \mapsto 1\}, \{v_2 \mapsto 2\}\}$
  - $\mathit{pre}(o_3) = \{v_1 \mapsto 2\}$, $\mathit{effs}(o_3) = \{\{v_3 \mapsto 1\}, \{v_3 \mapsto 2\}\}$
  - $\mathit{pre}(o_4) = \{v_1 \mapsto 1, v_2 = 1\}$, $\mathit{effs}(o_4) = \{\{v_2 \mapsto 3\}\}$
  - $\mathit{pre}(o_5) = \{v_1 \mapsto 1, v_2 = 2\}$, $\mathit{effs}(o_5) = \{\{v_2 \mapsto 3\}\}$
  - $\mathit{pre}(o_6) = \{v_1 \mapsto 2, v_3 = 1\}$, $\mathit{effs}(o_6) = \{\{v_2 \mapsto 3\}\}$
  - $\mathit{pre}(o_7) = \{v_1 \mapsto 2, v_3 = 2\}$, $\mathit{effs}(o_7) = \{\{v_2 \mapsto 3\}\}$
- $s_0 = \{v_1 \mapsto 0, v_2 \mapsto 0, v_3 \mapsto 0\}$

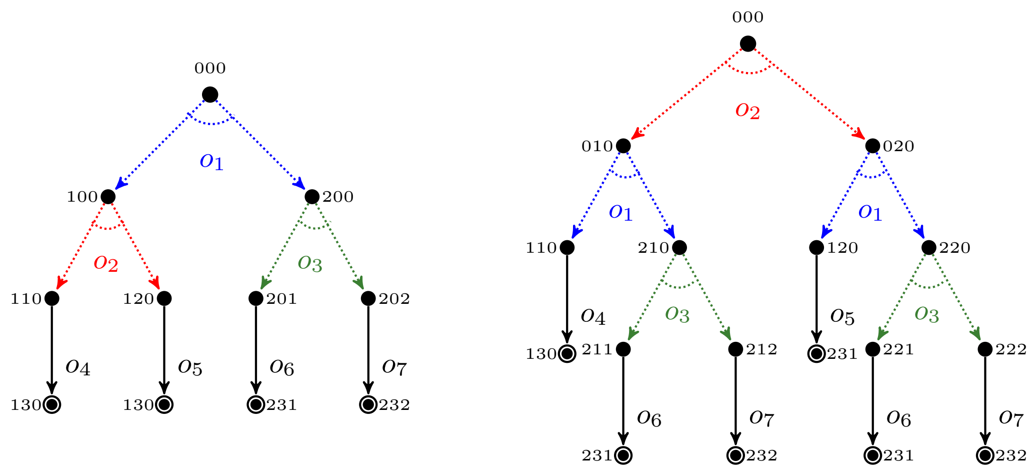


Figure 5.3: Both structures are policies for the planning task.

- $s_\star = \{v_2 \mapsto 3\}$

We compute T_{s_0} according to Definition 34: The set $\{o_4, o_5, o_6, o_7\}$ is a disjunctive action landmark for $s_\star$ in s_0 which we add to T_{s_0}. Obviously, all of these operators are inapplicable, which means we need to add a necessary enabling set for each of them. By considering variable v_1 in the precondition of all operators in T_{s_0}, operator o_1 can be added to T_{s_0} as a necessary enabling set. Clearly, o_1 is applicable in s_0, weakly interferes with o_3, and accords with o_2; hence, only o_3 will be added to T_{s_0}. Operator o_3 is inapplicable and $\{o_1\}$ is a necessary enabling set for it. Consequently, the computation terminates with $T_{s_0} = \{o_1, o_3, o_4, o_5, o_6, o_7\}$ with only o_1 being applied in s_0 and o_2 is pruned. The left graph in Figure 5.3 shows a strong plan that starts with o_1 in s_0.

On the other hand, considering variable v_2 for operators o_4 or o_5, as a basis to choose a necessary enabling set, leads to including o_2 to the stubborn set in addition to o_1, which is included for any choice of variables for operators o_6 and o_7. The right graph in Figure 5.3 shows the strong plan that starts with o_2 in s_0.

Unlike permuting operators of sequential plans, changing the application order of nondeterministic operators might result in new structures that differ in size from the original ones. For example, Figure 5.3 shows two strong plans that can be obtained by two different application orders for operators o_1 and o_2. For future work, this observation is a motive to investigate an optimized version of stubborn sets algorithm tailored to enforce the choice of small structures over larger ones (e.g., the left graph vs. the right graph in Figure 5.3).

Chapter 6

Sleep Sets for Classical Planning

The sleep sets method has originally been introduced in the area of computer-aided verification [GW92]. Sleep sets are path-dependent, i.e., they prune transitions in a state depending on information collected along the path through which the state has been reached. Several variants of sleep sets have been proposed by Godefroid [GHP93; GHP95; God96]. However, some of these variants were faulty when combined with search algorithms that perform full duplicate elimination as it has been shown in later work [KP95; Bos+09]. Although necessary modifications have been performed to guarantee the correctness of sleep sets, some theoretical questions remained unanswered. For instance, it was still unclear which search algorithms can safely be combined with the original form of sleep sets.

In this chapter, we eliminate the ambiguity related to sleep sets' variants and come up with a taxonomy of them. The definition of sleep sets relies on the notion of commutativity between operators (Definition. 15). The commutativity relation, denoted as $\bowtie$, is a binary relation on the set of operators. Informally, two operators are commutative if they do not affect the applicability or the effects of each other. Because the sleep sets technique is a transition reduction technique, it preserves all reachable states and prunes only redundant ones. As a path-dependent method, sleep sets require a total order on operators and operator sequences.

Next, we introduce the definitons of total orders and minimal paths.

Definition 35 (total orders on paths). [HAW15] Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task. We introduce the following notions:

- Let α, β, γ and λ be paths. A total order $<_o$ on operator sequences is *nested* if $\varepsilon <_o \alpha$ for all $\alpha \neq \varepsilon$, and $\alpha <_o \beta$ implies $\gamma\alpha\lambda <_o \gamma\beta\lambda$.
- Let α and β be paths. Let $<_o$ be a total order defined on operators in $\mathcal{O}$, e.g., $o_1 <_o o_2 <_o \ldots <_o o_n$. This order induces a total order on paths which is *length-lexicographic*. A total order $<_o$ on paths is length-lexicographic if the following holds: $\alpha <_o \beta$ *iff* either $|\alpha| < |\beta|$, or $|\alpha| = |\beta|$ and $o_i <_o o'_i$, where o_i and o'_i are the leftmost operators in α and β, respectively, where for all k, $1 \leq k \leq i$, it holds that $o_k = o'_k$, and $o_i \neq o'_i$.

Proposition 1. Every length-lexicographic order is a nested order.

Definition 36 (minimal path). [HAW15] Let $<_o$ be a nested order on operator sequences. For any two states s and t, we define $min(s,t)$, the minimal path from s to t, to be the least-cost path from s to t that is smallest according to the nested order $<_o$. $min(s,t)$ is undefined if there is no path from s to t.

Let $<_{ss}$ denote a total order on operators that is used by sleep sets. We emphasize that $<_{ss}$ is an arbitrary total order on operators enforced by sleep sets, i.e., sleep sets are not restricted to a specific order. To explain the notion of minimal paths, consider the following example: Let o_1 and o_2 be operators such that $o_1 <_{ss} o_2$. Furthermore, let s and t be states such that t is reachable from state s via paths o_1o_2 and o_2o_1. Then, $min(s,t) = o_1o_2$. In the following, we introduce the definition of sleep sets.

Definition 37 (sleep set). Given a path $\sigma_n = o_1 \ldots o_n$, its *sleep set* is defined as follows:

- $ss(\sigma_n) = \{o \in O \mid (o <_{ss} o_n \vee o \in ss(\sigma_{n-1})) \wedge o \bowtie o_n\}$ with $\sigma_{n-1} = o_1, \ldots, o_{n-1}$, and
- $ss(\varepsilon) = \emptyset$.

For every explored path σ, a sleep set is inductively computed and stored along with state $s = \sigma(s_0)$. In what follows, we will associate a sleep set with a path σ or with a state $s = \sigma(s_0)$, interchangeably.

When s is ready for expansion, only applicable operators that are *not* contained in $ss(\sigma)$ are applied. To make sleep sets technique practically efficient, sleep sets are computed in an interleaving fashion after generating path σo as follows:

(1) $ss(\sigma o) = \{o' \in ss(\sigma) \mid o' \bowtie o\}$, and

(2) o is *locally* included in $ss(\sigma)$.

Step (2) is executed to maintain the ordering used by sleep sets, and therefore it becomes unnecessary to perform an explicit comparison for the order of operators in step (1). In other words, the mere purpose of (2) is to propagate operators to the rest of the successor sequences. "Locally" is used to indicate that o is allowed to be used for propagating operators only during the current expansion. This means that o does not have to be included in $ss(\sigma)$ if the state which is reached by σ is re-expanded later. This will be clarified later.

Example 8. Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task, where

- $\mathcal{V} = \{a, b, c\}$
- $\mathcal{O} = \{o_1, o_2, o_3\}$, where
  - $pre(o_1) = \{\top\}$, $\mathit{eff}(o_1) = \{a \mapsto 1\}$
  - $pre(o_2) = \{\top\}$, $\mathit{eff}(o_2) = \{b \mapsto 1\}$
  - $pre(o_3) = \{b \mapsto 1\}$, $\mathit{eff}(o_3) = \{c \mapsto 1\}$
- $s_0 = \{a \mapsto 0, b \mapsto 0, c \mapsto 0\}$

- $s_\star = \{a \mapsto 1, c \mapsto 1\}$

The total ordering between operator is: $o_3 <_{ss} o_1 <_{ss} o_2$.

We see that $o_1 \bowtie o_2$ and $o_1 \bowtie o_3$. Consider the explored space in Figure 6.1. The initial state s_0 is labeled with an empty sleep set because $ss(\varepsilon) = \emptyset$. State s_2 is labeled with $ss(o_2) = \{o_1\}$ because $o_1 \bowtie o_2$ and $o_1 <_{ss} o_2$. State s_4 is labeled with $ss(o_2o_3) = \{o_1\}$ because $o_1 \bowtie o_3$ and $o_1 \in ss(o_2)$. Therefore, operator o_1 is not applied in states s_2 and s_4 (shown as dashed transitions). As a consequence, sleep sets preserved path $o_1o_2o_3$ (from s_0 to s_5) and pruned both $o_2o_1o_3$ and $o_2o_3o_1$.

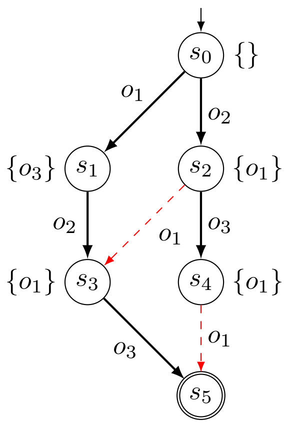


Figure 6.1: Sleep sets pruning.

6.1 Variants of Sleep Sets

Sleep sets have been considered within different contexts. In the verification literature, sleep sets have mainly been used with combination of full duplicate elimination and depth-first search. In the following, four main variants of sleep sets are listed [AW16]:

(A) Let s be a state generated by path σ for the first time. A sleep set $ss(\sigma)$ is computed and cached with s in order to be used for pruning whenever s is ready to be expanded. If s is revisited by an alternative path σ', then s is immediately pruned. This means that sleep sets are computed only once per state. This is the first version of sleep sets that has been combined with depth-first search and full duplicate elimination [GW92]. However, as sleep sets technique is path-dependent, reaching a state by a different path σ' can allow some operators to be applied in s, which are contained in $ss(\sigma)$, i.e., there exists some $o \in ss(\sigma)$ and $o \notin ss(\sigma')$. This is unsafe, since the successor state $s' = o(s)$ can be pruned from the reduced transition graph. Previous work has shown, by counter-examples, that combing this variant with depth-first search and full duplicate elimination is incomplete [KP95; Bos+09].

Algorithm 2 DFS$_{ss}$ (using variant A)

```
1: $Stack \leftarrow \emptyset; H \leftarrow \emptyset$    /*Hash Table*/
2: $Stack.push(s_0)$
3: while $Stack \neq \emptyset$ do
4:     $s \leftarrow Stack.pop()$
5:     $H.insert(s)$
6:     $app(s) \leftarrow app(s) \setminus ss(\sigma)$    /*$s = \sigma(s_0)$ */
7:     for $o \in app(s)$ do
8:         $s' \leftarrow o(s)$
9:         if $s' \notin H$ then
10:            $ss(\sigma o) \leftarrow \{o' \mid o' \in ss(\sigma) \wedge o \bowtie o'\}$
11:            $Stack.push(s')$
12:            $ss(\sigma) \leftarrow ss(\sigma) \cup \{o\}$
```

Algorithm 2 presents depth-first search with full duplicate elimination combined with variant A which has been proposed in [GW92]. It uses a stack *Stack* for storing the generated but not yet expanded states, and a hash table H to store expanded states. Each time a state s is popped from *Stack* (line 4), the set of applicable operators $app(s)$ is restricted to the ones outside the sleep set $ss(\sigma)$ (line 6). Upon expansion of s, if a successor state s' is not expanded before (i.e., not in H) (line 9), then its sleep set $ss(\sigma o)$ is computed (line 10), the state is pushed on the stack (line 11), and the operator o is added locally to $ss(\sigma)$ (line 12) to be propagated to sleep sets of future successors of s if commutativity holds.

(B) Similar to variant A, states are pruned when they are revisited, but the algorithm has been modified to exclude some operators from sleep sets. It is worthwhile to mention that a kind of unsafe interaction can occur between variant A and the existence of cycles in the state space. The first modification to variant A was to exclude operator o from $ss(\sigma)$ if σo closes a cycle. However, this does not fully eliminate the outcome of the unsafe interaction as it has been shown later [KP95]. In a further refinement to the algorithm, upon expansion of a state, operators that lead to cycles are removed from the sleep set [GHP93]. This version solves the incompleteness problem that is caused by cycles. These modifications are shown in algorithm 3 in lines 7, 13 and 14. Line 7 excludes operators from $ss(\sigma)$ if they lead to states that already exist on the stack (i.e., cause cycles), and lines 13-14 do not add operator o locally to $ss(\sigma)$ if it closes a cycle.

(C) Let s be a state reached by paths $\sigma_1 \ldots \sigma_n$ in this particular order, i.e., first by σ_1 and last by σ_n. We will use the notation $ss(\sigma_1 \ldots \sigma_n)$ to denote the set of operators that are allowed to be pruned after visiting s by σ_n. The sleep set $ss(\sigma_1 \ldots \sigma_n)$ is inductively defined as $ss(\sigma_1 \ldots \sigma_{n-1}) \cap ss(\sigma_n)$. This means that some operators, which are pruned according to $ss(\sigma_1 \ldots \sigma_{n-1})$, can be applied according to $ss(\sigma_1 \ldots \sigma_n)$ because they are not included in $ss(\sigma_n)$. Duplicated states are pruned if $ss(\sigma_1 \ldots \sigma_n) = \emptyset$. This variant has been shown to be complete when used with full duplicate elimination, and it is independent of the order in which paths are explored by the

Algorithm 3 DFS$_{ss}$ (using variant B)

```
 1: $Stack \leftarrow \emptyset; H \leftarrow \emptyset$
 2: $Stack.push(s_0)$
 3: while $Stack \neq \emptyset$ do
 4:     $s \leftarrow Stack.pop()$
 5:     $H.insert(s)$;
 6:     $app(s) \leftarrow app(s) \setminus ss(\sigma)$
 7:     $ss(\sigma) \leftarrow \{o \mid o \in ss(\sigma) \wedge s' = o(s) \wedge s' \notin Stack\}$
 8:     for $o \in app(s)$ do
 9:         $s' \leftarrow o(s)$
10:         if $s' \notin H$ then
11:             $ss(\sigma o) \leftarrow \{o' \mid o' \in ss(\sigma) \wedge o \bowtie o'\}$
12:             $Stack.push(s')$
13:             if $s' \notin Stack$ then
14:                 $ss(\sigma) \leftarrow ss(\sigma) \cup \{o\}$
```

search algorithm [God96]. Algorithm 4 represents DFS with full duplicate

Algorithm 4 DFS$_{ss}$(using variant C)

```
 1: $Stack \leftarrow \emptyset$; $H \leftarrow \emptyset$
 2: $Stack.push(s_0)$
 3: while $Stack \neq \emptyset$ do
 4:     $s \leftarrow Stack.pop()$
 5:     if $s \notin H$ then
 6:         $H.insert(s)$
 7:         $app(s) \leftarrow app(s) \setminus ss(\sigma_1)$    /*$n = 1$*/
 8:     else
 9:         $app(s) \leftarrow ss(\sigma_1, \ldots, \sigma_{n-1}) \setminus ss(\sigma_n)$
10:         $ss(\sigma_1, \ldots, \sigma_n) \leftarrow ss(\sigma_1, \ldots, \sigma_{n-1}) \cap ss(\sigma_n)$
11:     for $o \in app(s)$ do
12:         $X \leftarrow ss(\sigma_1, \ldots, \sigma_n) \cup \{o' \mid o' <_{ss} o \wedge o' \in app(s)\}$
13:         $s' \leftarrow o(s)$
14:         $ss(\sigma_n o) \leftarrow \{o' \mid o' \in X \wedge o \bowtie o'\}$
15:         $Stack.push(s')$
```

elimination combined with variant C of sleep sets. The algorithm follows the lines of the search algorithm suggested by Godefroid in his monograph adapted to the notation of this thesis [God96]. Godefroid proposed the algorithm to combine persistent sets and sleep sets, while here we have only sleep sets (the combination of sleep sets with state reduction techniques is discussed in the next chapter).

If state s is to be expanded for the first time, which means it has been generated only once via path σ_1, then $app(s)$ is computed as before (lines 5-7); otherwise, $app(s)$ contains only the operators which are in the updated sleep set $ss(\sigma_1, \ldots, \sigma_{n-1})$ but not in $ss(\sigma_n)$, and the sleep set $ss(\sigma_1, \ldots, \sigma_n)$ is computed (lines 8-10). The set X, in line 12, is used to avoid adding operators locally as before, since these operators are not supposed to be maintained in $ss(\sigma_1, \ldots, \sigma_n)$ when s is to be revisited later.

(D) This variant is designed to be used with tree-search algorithms with only cycle elimination (not full duplicate elimination). Let s be a state that is first generated by path $\sigma = o_1 \ldots o_n$, then by path $\sigma' = o_1 \ldots o_n o_{n+1} \ldots o_{n+k}$ for $k \geq 1$ (s is a part of a cycle). Sleep set $ss(\sigma)$ is computed, but s is pruned as duplicate when reached by σ', i.e., $ss(\sigma, \sigma')$ is not computed. Furthermore, if s is visited by $\sigma = o_1 \ldots o_n$, and then by $\sigma' = o'_1 \ldots o'_m$, then a new sleep set $ss(\sigma')$ is independently computed and used for s. In other words, if s is to be expanded after being generated by σ, then $ss(\sigma)$ is used for pruning, and when generated later by σ', $ss(\sigma')$ is used upon expansion. This version has been shown to be completeness and optimality preserving when combined with IDA* [HAW15]. For consistency, we state Algorithm 5 using DFS that performs only cycle detection combined with sleep set variant D. The superscript i in s^i is used to denote the i-th visit of s, which is tracked by the algorithm, but it is used here to associate it with σ^i, which is the particular path through which s is generated in its i-th visit. Because this algorithm performs only cycle elimination, no hash table is being used. Like previous variants, the operators in sleep sets are excluded from the set of applicable operators (line 5). If a successor state of s does not already exist on the search stack (line 8), a new sleep set associated with it is computed (line 9), the state is pushed on the stack (line 10), and operator o is locally added to $ss(\sigma^i)$ (line 11).

Algorithm 5 DFS$_{ss}$ (using variant D)

```
1: $Stack \leftarrow \emptyset$
2: $Stack.push(s_0)$
3: **while** $Stack \neq \emptyset$ **do**
4:     $s^i \leftarrow Stack.pop()$
5:     $app(s^i) \leftarrow app(s^i) \setminus ss(\sigma^i)$
6:     **for** $o \in app(s^i)$ **do**
7:         $s' \leftarrow o(s^i)$
8:         **if** $s' \notin Stack$ **then**
9:             $ss(\sigma^i o) \leftarrow \{o' \mid o' \in ss(\sigma^i) \land o \bowtie o'\}$
10:            $Stack.push(s')$    /*$ss(\sigma^i o)$ is associated with $s'$*/
11:            $ss(\sigma^i) \leftarrow ss(\sigma^i) \cup \{o\}$
```

6.2 Sleep Sets and Search Algorithms

In this section, we show how to safely combine variants of sleep sets with various search algorithm, i.e., which combinations are completeness and optimality preserving.

Given a search algorithm Alg, a set of operators $\mathcal{O}$, and a total order $<_{Alg}$ on $\mathcal{O}$ that Alg uses to generate successor states during search, we say that Alg is *order-consistent* if $<_{Alg}$ and $<_{ss}$ are identical. In the following, we assume that search algorithms, that we want to combine with sleep sets, are order-consistent.

6.2.1 Breadth-First Search

Breadth-First Search (BFS) is a blind search algorithm that performs full duplicate elimination, and retains completeness and optimality. We prove that variant A of sleep sets suffices to be combined with BFS while preserving its original characteristics, namely, all states are generated modulo duplicates, and an optimal solution is extracted. We call the algorithm, which combines BFS with variant A of sleep sets, BFS$_{ss}$.

Before stating the main theorem about completeness and optimality of BFS$_{ss}$, we need to show that sleep sets preserve at least one permutation of each path in the state space. The following theorem is a special case from a more general one for a more general definition of sleep sets by Holte et al. [HAW15].

Theorem 5. *For any states s and t s.t. t is reachable from s, it holds that $o_k \notin ss(o_1 \dots o_{k-1})$ for all k with $1 \leq k \leq |min(s,t)|$.*

Proof. Let $min(s,t) = \sigma_{i-1} o_i \delta_{i+1}$, where σ_{i-1} is a prefix of $min(s,t)$ consisting of operators $o_1 \dots o_{i-1}$, and δ_{i+1} is a suffix consisting of operators $o_{i+1} \dots o_{|min(s,t)|}$. If $i = 1$, then σ_{i-1} is empty, and therefore $ss(\sigma_{i-1}) = \emptyset$, o_i is preserved. If $i \geq 2$, then we assume that $o_i \in ss(\sigma_{i-1})$. We distinguish two cases:

- $o_i <_{ss} o_{i-1}$:
  Since $o_i \in ss(\sigma_{i-1})$, it follows that $o_i \bowtie o_{i-1}$, and $\sigma_{i-2} o_i o_{i-1} \delta_{i+1} <_{ss} \sigma_{i-2} o_{i-1} o_i \delta_{i+1} = min(s,t)$. This contradicts that assumption that $min(s,t)$ is minimal according to $<_{ss}$.

- $o_i \not<_{ss} o_{i-1}$:
  Let j be the largest index in σ_{i-2} such that $o_j \bowtie o_i$ and $o_i <_{ss} o_j$. This means that $o_i \in ss(\sigma_j)$ and has persisted in the sleep sets from σ_j to σ_{i-1}. This implies that $o_i \bowtie o_k$, where $j \leq k < i$, and hence $\sigma_{j-1} o_i oj \dots o_{i-1} \delta_{i+1} <_{ss} \sigma_{j-1} oj \dots o_{i-1} o_i \delta_{i+1} = min(s,t)$, which again contradicts the definition of $min(s,t)$.

□

The above theorem shows that sleep sets have the property to preserve a particular optimal path between any two states, namely, the path that is smallest with respect to $<_{ss}$. Now we need to show that this path is not pruned by duplicate elimination executed by breadth-first search.

Lemma 1. *Let s and t be states with t is reachable from s. Then it holds for all i with $1 \leq i \leq n-1$ and $s_i = o_1 \dots o_i(s)$ that $min(s, s_i) = o_1 \dots o_i$ is the minimal path from s to s_i according to $<_{ss}$.*

Proof. Let $\sigma_i = o_1 \dots o_i$ for some $i \in \{1, \dots, o_{n-1}\}$. First, we show that $cost(\sigma_i)$ is minimal among all paths that lead from s to s_i. Assume that there exists σ such that $s_i = \sigma(s)$ and $cost(\sigma) < cost(\sigma_i)$. This implies that $cost(\sigma o_{i+1} \dots o_n) < cost(\sigma_i o_{i+1} \dots o_n) = min(s,t)$ which contradicts the fact that $min(s,t)$ has a minimal cost among all paths that lead from s to t. Second, consider a path σ such that $s_i = \sigma(s)$, $cost(\sigma) = cost(\sigma_i)$, and $\sigma <_{ss} \sigma_i$. Then it holds that $\sigma o_{i+1} \dots o_n <_{ss} \sigma_i o_{i+1} \dots o_n = min(s,t)$, which again contradicts that $min(s,t)$ is minimal according to $<_{ss}$ among paths from s to t. □

Lemma 1 shows that prefixes of the minimal path between any two states s and t are also minimal paths.

Theorem 6. *[AW16] BFS$_{ss}$ is complete. When applied with unit-cost operators, BFS$_{ss}$ is optimal.*

Proof. Let s be a state reachable from s_0. Let $min(s_0, s) = o_1 \dots o_n$, and $\sigma = o'_1 \dots o'_m$ be a path with $\sigma \neq min(s_0, s)$ that reaches s. We show that standard breadth-first search using $<_{Alg}$ generates s with $min(s_0, s)$ first, i.e., before it generates s with σ. To see this, consider the following cases:

1. $n < m$: $min(s_0, s)$ is explored before σ because breadth-first search explores shorter paths before longer ones.

2. $n > m$ cannot occur because it would contradict the assumption that $min(s_0, s)$ is minimal.

3. $n = m$: Let i be the left-most position where $min(s_0, s)$ and σ differ, i.e., $o_i \neq o'_i$ and $o_j = o'_j$ for $j < i$. By assumption, $<_{ss}$ is equal to $<_{Alg}$, and $\min(s_0, s) <_{ss} \sigma$, hence $\min(s_0, s) <_{Alg} \sigma$ and $o_i <_{Alg} o'_i$. It follows that breadth-first search explores the path $o_1 \dots o_i$ before $o'_1 \dots o'_i$, and hence (by exploring states in a first-in first-out manner) also their completion $min(s_0, s)$ before σ.

From Lemma 1, we know that all the prefixes of $min(s_0, s)$ are minimal as well, hence it follows that all states $s_1, \dots, s_n$, generated on the path from s_0 to s, are generated on the path $min(s_0, s)$ first. It follows that $s_1, \dots, s_n$ are not pruned by breadth-first search as duplicate states. By Theorem 5, it follows that additionally computing sleep sets for these prefix paths that generate $s_1, \dots, s_n$, which yields BFS$_{ss}$, preserves $min(s_0, s)$, showing the claim.

□

The theorem shows that $min(s_0, s)$ is guaranteed to be generated first by breadth-first search among all paths that lead from s_0 to state s, and therefore the states on $min(s_0, s)$ cannot be pruned as duplicates. However, we do not have this property in other search algorithms that perform duplicate elimination, like Dijkstra's algorithm and A*. Algorithm 6 describes BFS$_{ss}$.

Algorithm 6 BFS$_{ss}$

```
1: $Open \leftarrow \emptyset$; $H \leftarrow \emptyset$
2: $Open.insert(s_0)$
3: while $Open \neq \emptyset$ do
4:     $s \leftarrow Open.pop()$
5:     $H.insert(s)$
6:     $app(s) \leftarrow app(s) \setminus ss(\sigma)$
7:     for $o \in app(s)$ do
8:         $s' \leftarrow o(s)$
9:         if $s' \notin H$ then
10:            $ss(\sigma o) \leftarrow \{o' \mid o' \in ss(\sigma) \land o \bowtie o'\}$
11:            $Open.insert(s')$
12:            $ss(\sigma) \leftarrow ss(\sigma) \cup \{o\}$
```

The main difference between Algorithm 6 and Algorithm 2 is that the former uses a First In-First Out container for storing generated states (*Open*), and the latter uses a Last In-First Out container (*Stack*). However, Algorithm 2 is incomplete, whereas Algorithm 6 is, indeed, complete. In fact, this leads us to the conclusion that the order in which states are expanded by the search algorithm plays a role in the decision of which variant of sleep sets the algorithm can safely combined with.

6.2.2 A* Search

A* search algorithm is a graph search algorithm with full duplicate elimination, and is usually equipped with heuristics. A* uses an open list as a priority queue and a closed list as a hash table, and yields optimal solutions when combined with admissible heuristics.

This section presents a new algorithm obtained by using A* with variant C of sleep sets, and we call the algorithm A$^*_{ss}$. For simplicity, we assume that A* uses a consistent heuristic. Previously, we presented Algorithm 4, using DFS, as an example of a search algorithm with full duplicate elimination combined with variant C of sleep sets. In fact, A$^*_{ss}$ differs from algorithm 4 in three main points:

1. Algorithm 4 uses a stack data structure to keep track of generated states. However, the original completeness proof presented by Godfroid does not rely on the stack behavior [God96]. For this reason, this algorithm remains complete if a priority queue is used instead of a stack.

2. Similar to A*, A$^*_{ss}$ needs to check for the goal condition when a state is popped from the priority queue (open list). This step does not affect the computation or functionality of sleep sets pruning.

3. Finally, assume a state s is generated first by paths $\sigma_1, \dots, \sigma_n$, and generated by path σ later. If s is contained in the open list and not yet in the closed list, then the sleep set of s is updated according to σ, i.e., $ss(\sigma_1, \dots, \sigma_n, \sigma) := ss(\sigma_1, \dots, \sigma_n) \cap ss(\sigma)$.

Algorithm 7 A$^*_{ss}$

```
1: $Open \leftarrow \emptyset$; $Closed \leftarrow \emptyset$
2: $n_0 \leftarrow make\_node(s_0)$
3: $Open.insert(n_0)$;
4: while $Open \neq \emptyset$ do
5:     $n \leftarrow Open.pop\_min()$
6:     $s \leftarrow n.get\_state()$
7:     if $is\_goal(s)$ then
8:         $plan \leftarrow \sigma^s_{min}$   /*minimal cost generating path of $s$*/
9:         return SOLVED
10:    $app(s) \leftarrow app(s) \setminus ss(\sigma^s_1, \dots, \sigma^s_n)$
11:    EXPAND$(s, app(s), ss(\sigma^s_1, \dots, \sigma^s_n))$
12: return UNSOLVED
```

In the following, we will describe A^*_{ss} in more detail. In order to make clear which state the algorithm is considering, we use σ^s, instead of σ, to denote the path that ended in state s, i.e., $s = \sigma(s_0)$. Accordingly, the sleep set of state s reached by paths $\sigma^s_1, \dots, \sigma^s_n$ in this order is denoted with $ss(\sigma^s_1, \dots, \sigma^s_n)$.

Algorithm 7 represents the usual A^* algorithm combined with variant (C) of sleep sets. The algorithm uses search nodes to keep track of states and their information. Algorithm A^*_{ss} differs from A^* in the computation of successor states as follows:

1. **Computing the expansion set:** while A^* considers *all* applicable operators in a given state s for generating its successor states, A^*_{ss} considers a *subset* of applicable operators. We call this set the *expansion set*, and is defined as
$$app(s) \setminus ss(\sigma^s_1, \dots, \sigma^s_n),$$
where $\sigma^s_1, \dots, \sigma^s_n$ are the paths by which s has been generated at the time when s is expanded.

2. **Operator application and sleep set updates:** A^*_{ss} applies the operators in $app(s) \setminus ss(\sigma^s_1, \dots, \sigma^s_n)$ and computes (or updates, respectively) the corresponding sleep set of the successor states. The pseudo code of this expansion step, called EXPAND$(s, app(s), ss(\sigma^s_1, \dots, \sigma^s_n))$ in the following, is given in Algorithm 8 to be used with consistent heuristics and Algorithm 9 for inconsistent heuristics.

Algorithm 8 Successor generation & sleep set updates with a consistent h

```
1: function EXPAND($s, app(s), ss(\sigma^s_1, \dots, \sigma^s_n)$)
2:     for $o \in app(s) \setminus ss(\sigma^s_1, \dots, \sigma^s_n)$ do
3:         $s' \leftarrow o(s)$
4:         $\sigma^s \leftarrow$ minimal cost generating path of $s$
5:         $X \leftarrow ss(\sigma^s_1, \dots, \sigma^s_n) \cup \{o' \mid o' <_{ss} o \wedge o' \in app(s)\}$
6:         $ss(\sigma^s o) \leftarrow \{o' \mid o' \in X$ and $o' \bowtie o\}$
7:         $ss(\sigma^{s'}_1, \dots, \sigma^{s'}_m, \sigma^s o) \leftarrow ss(\sigma^{s'}_1, \dots, \sigma^{s'}_m) \cap ss(\sigma^s o)$
8:         if $s' \in Closed$ then
9:             $applicable\_sleep \leftarrow ss(\sigma^{s'}_1, \dots, \sigma^{s'}_m) \setminus ss(\sigma^s o)$
10:            EXPAND $(s', applicable\_sleep, \emptyset)$
11:        else
12:            $n' \leftarrow make\_node(s')$
13:            $Open.insert(n')$
```

A^*_{ss} with consistent heuristics

In Algorithm 8, the search is guided by a consistent heuristic function. Assuming that σ^s is the path on which s has been reached last, function EXPAND$(s, app(s), ss(\sigma^s_1, \dots, \sigma^s_n))$ computes the sleep set of the successor state s' reached on the path $\sigma^s o$ (Line 5–6). The sleep set of s' is updated according to variant C (Line 7). If s' is closed, then s' is further expanded by generating all successors that are not pruned according to the most recently computed sleep set (Line 8–10). Recall that $\sigma^{s'}_1, \dots, \sigma^{s'}_m$ are the paths by which s' has

Algorithm 9 Successor generation & sleep set updates with an inconsistent h

```
1: function EXPAND($s, app(s), ss(\sigma_1^s, \ldots, \sigma_n^s)$)
2:     for $o \in app(s) \setminus ss(\sigma_1^s, \ldots, \sigma_n^s)$ do
3:         $s' \leftarrow o(s)$
4:         $\sigma^s \leftarrow$ minimal cost generating path of $s$
5:         $X \leftarrow ss(\sigma_1^s, \ldots, \sigma_n^s) \cup \{o' \mid o' <_{ss} o \wedge o' \in app(s)\}$
6:         $ss(\sigma^s o) \leftarrow \{o' \mid o' \in X$ and $o' \bowtie o\}$
7:         $ss(\sigma_1^{s'}, \ldots, \sigma_m^{s'}, \sigma^s o) \leftarrow ss(\sigma_1^{s'}, \ldots, \sigma_m^{s'}) \cap ss(\sigma^s o)$
8:         if $s' \in Closed$ then
9:             if $cost(\sigma^s o) < cost(\tilde{\sigma})$ for all $\tilde{\sigma} \in \{\sigma_1^{s'}, \ldots, \sigma_m^{s'}\}$ then
10:                EXPAND ($s', app(s'), ss(\sigma_1^{s'}, \ldots, \sigma_m^{s'}, \sigma^s o)$)
11:            else
12:                $applicable\_sleep \leftarrow ss(\sigma_1^{s'}, \ldots, \sigma_m^{s'}) \setminus ss(\sigma^s o)$
13:                EXPAND ($s', applicable\_sleep, \emptyset$)
14:        else
15:            $n' \leftarrow make\_node(s')$
16:            $Open.insert(n')$
```

been reached before reaching s' on $\sigma^s o$. At this point, we also observe that the particular function signature (which includes *app*(s) and the sleep set of s) is convenient for the recursive call in Line 10. Finally, in lines 11-13, we cover the case where s' is either generated for the first time, or previously generated but not expanded yet, i.e., s' is already in the open list.

A^*_{ss} with inconsistent heuristics

A search algorithm guided by an inconsistent heuristic function might re-generate states via cheaper paths than the paths via which those states have previously been generated. This might lead to suboptimal plans, unless this case is handled by the search separately. A^* search deals with this situation by checking the cost of the paths that lead to every generated state s and performs a *re-opening* step if the cost of the new path to s is less than the cost of every previous paths to s. Re-opening means inserting s again in the open list to be ready for future expansion, and considering the new path instead of the previous paths, i.e., the minimal cost path via which s has been reached so far.
Now, consider Algorithm 9 that assumes an inconsistent heuristic function. The only difference from Algorithm 8 can be seen in lines 9-10: Line 9 checks if $\sigma^s o$ is the cheapest among previous paths to the generated state s'. In that case, s' is expanded with the updated sleep set computed in Line 7. This means all the operators that have already been applied in s' (during a previous expansion process) are going to applied again in s' after being reached via $\sigma^s o$. The following example shows that neglecting this modification to Algorithm 8 (i.e., ignoring Lines 9-10 in Algorithm 9) can result in suboptimal plans.

Example 9. Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task, where

- $V = \{a, b, c, d, g\}$
- $O = \{o_1, o_2, o_3, o_4, o_g\}$, where

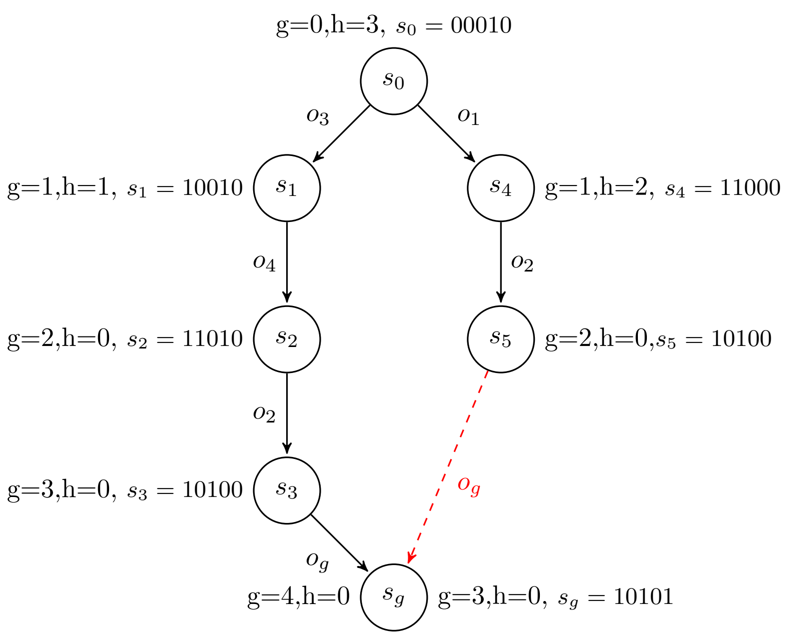


Figure 6.2: Using Algorithm 8 with an inconsistent heuristic leads to a suboptimal plan.

- $pre(o_1) = \{a \mapsto 0\}$, $eff(o_1) = \{a \mapsto 1, b \mapsto 1, d \mapsto 0\}$
- $pre(o_2) = \{b \mapsto 1\}$, $eff(o_2) = \{b \mapsto 0, c \mapsto 1, d \mapsto 0\}$
- $pre(o_3) = \{a \mapsto 0, d \mapsto 1\}$, $eff(o_3) = \{a \mapsto 1\}$
- $pre(o_4) = \{a \mapsto 1, d \mapsto 1\}$, $eff(o_4) = \{b \mapsto 1\}$
- $pre(o_g) = \{c \mapsto 1\}$, $eff(o_g) = \{g \mapsto 1\}$

- $s_0 = \{a \mapsto 0, b \mapsto 0, c \mapsto 0, d \mapsto 1, g \mapsto 0\}$
- $s_\star = \{g \mapsto 1\}$

The total order is given as $o_g <_{ss} o_4 <_{ss} o_3 <_{ss} o_2 <_{ss} o_1$. We also consider unit-cost operators.

In Example 9, all states have empty sleep sets: Although o_g is commutative with o_1, o_3 and o_4, o_g is applicable in only one reachable state in which none of these operators is applicable. The states are annotated with their g-values, h-values, and the values of state variables in the order "*abcdg*". The expansion order of states is indicated by their indices $i \in \{0 \ldots 5\}$. If we consider Algorithm 8, then it is possible to expand state s_3 before s_5 (where $s_3 = s_5$), but o_g will be pruned in s_5 because the set-difference between both sleep sets (i.e., $ss(o_3o_4o_2)$ and $ss(o_1o_2)$) is empty. However, by using Algorithm 9, o_g will be applied in s_5, thereby preserving the optimal plan $o_1o_2o_g$.

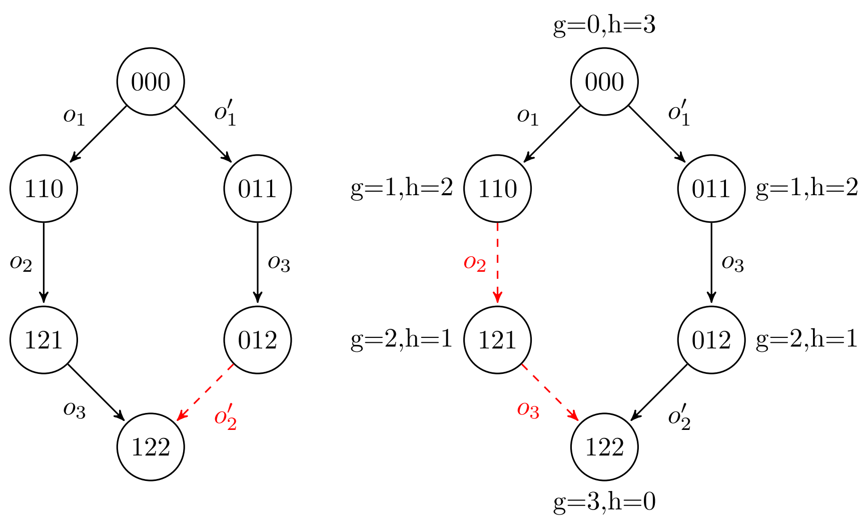


Figure 6.3: A^*_{ss} generates $o'_1 o_3 o'_2$ (right) and BFS_{ss} generates $o_1 o_2 o_3$ (left). The dashed lines refer to the operators that are not applied because the goal has already been reached.

Comparing A^*_{ss} to BFS_{ss}

While BFS_{ss} is guaranteed to generate the minimal path according to $<_{ss}$ between two given states s and t first (i.e., $min(s,t)$), A^*_{ss} does not have this property. The following example illustrates this fact.

Example 10. Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task, where

- $\mathcal{V} = \{a, b, c\}$
- $\mathcal{O} = \{o_1, o_2, o_3, o'_1, o'_2\}$, where
  - $pre(o_1) = \{a \mapsto 0, b \mapsto 0\}$, $\mathit{eff}(o_1) = \{a \mapsto 1, b \mapsto 1\}$
  - $pre(o_2) = \{b \mapsto 1, c \mapsto 0\}$, $\mathit{eff}(o_2) = \{b \mapsto 2, c \mapsto 1\}$
  - $pre(o_3) = \{c \mapsto 1\}$, $\mathit{eff}(o_3) = \{c \mapsto 2\}$
  - $pre(o'_1) = \{b \mapsto 0, c \mapsto 0\}$, $\mathit{eff}(o'_1) = \{b \mapsto 1, c \mapsto 1\}$
  - $pre(o'_2) = \{b \mapsto 1, c \mapsto 2\}$, $\mathit{eff}(o'_2) = \{a \mapsto 1, b \mapsto 2\}$
- $s_0 = \{a \mapsto 0, b \mapsto 0, c \mapsto 0\}$
- $s_\star = \{b \mapsto 2, c \mapsto 2\}$

The operator ordering is $o_1 <_{ss} o_2 <_{ss} o_3 <_{ss} o'_1 <_{ss} o'_2$.

In Figure 6.3, the left graph refers to the explored space of BFS_{ss}, while the right one refers to the explored space of A^*_{ss}. The sleep sets of all states are empty. This is easy to check: The only commutativity that holds is $o_1 \bowtie o_3$ but there is no reachable state in which both operators are applicable (i.e., state 001). The minimal path according to $<_{ss}$ is $o_1o_2o_3$. Given that BFS$_{ss}$ uses $<_{ss}$ for the generation of successor states, path $o_1o_2o_3$ is guaranteed to be generated first. Consider the right graph of the figure. The states are annotated by their h-values and their g-values. First, the initial state is expanded (also using $<_{ss}$) by generating state 110 first and then state 011. Both states have the same $f = h + g$ value. If the tie-breaking used by the search chooses 011 to be expanded next, then state 012 is generated. Again, assume the tie-breaking decides to expand 012 next instead of 110 and the goal state 122 is generated. Finally, states 110 and 122 have the same f value and we assume that the latter state is popped from the open list to be expanded, which means that path $o'_1o_3o'_2$ is generated first.

The following theorem states that A* remains complete and optimal when combined with variant C of sleep sets.

Theorem 7. *For admissible heuristics, A^*_{ss} is complete and optimal.*

Proof. The proof is a special case of the proof of Theorem 12 in the next chapter, which shows the claim for A$^*_{ss}$ with additional state pruning based on strong stubborn sets. □

6.2.3 IDA* Search

IDA* with only cycle detection has been combined with variant D of sleep sets [HAW15]. We call the resulting algorithm IDA$^*_{ss}$ and we will see that combining sleep sets with cycle detection and heuristic cutoffs is safe. The following two theorems and their proofs follow the structure of the proofs by Holte and Burch to show the safety of move pruning [HB14].

Theorem 8. *[AW16] Sleep sets are safe to use in conjunction with cycle detection.*

Proof. Let s be a state that is reachable from s_0. We show that cycle detection does not eliminate $min(s_0, s)$. Assume that cycle detection eliminates $min(s_0, s)$. This means that $min(s_0, s)$ must contain a cycle, i.e., $min(s_0, s) = \alpha\gamma\beta$ for operator sequences α, γ, β, with $|\gamma| > 0$ and $\alpha\gamma[s_0] = \alpha[s_0]$. This implies that $\alpha\beta$ is a path from s_0 to s with $cost(\alpha\beta) \leq cost(min(s_0, s))$ and $\alpha\beta <_{ss} min(s_0, s)$, which contradicts the definition of $min(s_0, s)$, hence showing that $min(s_0, s)$ is not eliminated. From Theorem 5, it follows that $min(s_0, s)$ is preserved by sleep sets as well. Together with the properties that IDA* is complete (and optimal for admissible heuristics), this shows the claim. □

Theorem 9. *[AW16] Sleeps sets are safe to use in conjunction with heuristic cutoff for any bound $b \geq f^*$, and any admissible heuristic h.*

Proof. Let h be an admissible heuristic, and let s be a state that is reachable from s_0. We show that heuristic cutoffs do not eliminate min(s_0, s). Let operator sequence α be a prefix of min(s_0, s). As h is admissible, we have $cost(\alpha) + h(\alpha[s_0]) \leq cost(min(s_0, s))$ since α is a prefix of $min(s_0, s)$. As $cost(min(s_0, s)) = C^*$, we have $cost(\alpha) + h(\alpha[s_0]) \leq C^*$. Heuristic cutoffs can only prune paths with costs strictly larger than C^*, hence α is not pruned. Since α has been chosen as an arbitrary prefix of $min(s_0, s)$ (including $min(s_0, s)$ itself), this shows that heuristic cutoffs do not prune $min(s_0, s)$. □

Corollary 4. *IDA$^*_{ss}$ combined with cycle detection is completeness and optimality preserving.*

Proof. The proof follows directly from Theorems 8 and Theorem 9. □

Like stubborn sets, the implementation of sleep sets consists of two steps: A preprocessing step in which the commutativity relation for all operators of the planning task is precomputed and cached in memory, and a runtime step where sleep sets are computed for generated states using the cached commutativity relation.

6.3 The Pruning Power of Sleep Sets

Previous work has introduced some other transition reduction techniques that can be employed by combining them with tree search algorithms to reduce the number of duplicates. All of these techniques have been proposed in the area of planning. In the following, we present three transition reduction techniques and compare them to sleep sets.

6.3.1 Sleep Sets and Commutativity Pruning

Commutativity pruning is a transition partial order reduction technique that has been proposed to produce less duplicated states during search performed by IDA* search [HG00]. Like sleep sets, commutativity pruning relies on the notion of commutativity and uses a predefined total ordering $<_{cp}$ on the operators of the planning task. While performing search, an applicable operator o' in state s is allowed only if $o <_{cp} o'$, where o is the operator that led to s.

Wehrle & Helmert have shown that sleep sets *dominate* commutativity pruning given that both methods use the same total ordering on operators [WH12]. This means that sleep sets can prune all paths that are pruned by commutativity pruning but not vice versa. This fact can be shown by rewriting the definition of sleep sets for path $\sigma = o_1, \ldots, o_n$ as follows:

$$ss(\sigma) = \{o \in O \mid o <_{ss} o_n \wedge o \bowtie o_n\} \cup \{o \in O \mid o \in ss(\sigma_{n-1}) \wedge o \bowtie o_n\}$$

We observe that the definition of sleep sets of path σ has been split into two parts:

1. Commutative operators with smaller order than o_n, and
2. Commutative operators propagated from the sleep set of the prefix σ_{n-1}.

The first part is responsible for capturing duplicates caused by swapping two consecutive commutative operators in a sequence. For example, if $o_1 <_{ss} o_2$ and $o_1 \bowtie o_2$, then o_2o_1 is pruned and o_1o_2 is preserved. This is equivalent to how commutativity pruning behaves when $<_{ss}$ is equivalent to $<_{cp}$. The second part of the definition is more powerful in the sense that it can propagate operators included in sleep sets down sequences of arbitrary length leading to more reduction in duplicates. For example, given operators $o_1o_2o_3$ with $o_1 <_{ss} o_2 <_{ss} o_3$, $o_1 \bowtie o_2$, $o_3 \bowtie o_2$, and $<_{cp}$ is equivalent to $<_{ss}$, then:

- $o_3o_1o_2$ is *pruned* by sleep sets because $o_2 \in ss(o_3o_1)$ which triggers the second part of the definition.

- $o_3o_1o_2$ is *not pruned* by commutativity pruning because $o_1 <_{cp} o_2$.

To summarize, we notice that the pruning power of sleep sets lies in the second part of the definition which allows an operator to leapfrog over a sequence of operators of arbitrary length. On the other hand, commutativity pruning is capable only of permuting sequences of length two.

6.3.2 Sleep Sets and Stratified Planning

Stratified planning is a transition partial order reduction technique that has been proposed in the context of optimal planning [CXY09; Xu+11]. The original work was faulty [CXY09] and has been corrected in a later version which is considered here [Xu+11]. Before we explain the algorithm, we need to introduce the definition of causal graphs for planning tasks.

Definition 38. (causal graph). [Kno94; Hel06] Given a planning task $\Pi = (\mathcal{V}, \mathcal{O}, s_0, \mathcal{G})$, the causal graph of Π is a directed graph $CG = (V, E)$, where $V = \mathcal{V}$, and there exists an edge $(v, v') \in E$ *iff* $v \neq v'$ and there exists an operator $o \in \mathcal{O}$ such that $v \in vars(o)$ and $v' \in vars(\mathit{eff}(o))$.

First, the stratified planning technique computes the strongly connected components $C_1, \ldots, C_n$ of the causal graph[1]. Then, a topological ordering $C_1 < \cdots < C_n$ is imposed on these components such that there is an edge from a variable in component C_i to a variable in component C_j only if $i \leq j$. Next, the algorithm performs the following:

1. Every variable $v \in \mathcal{V}$ is assigned a level according to the topological component it belongs to, i.e., $level(v) = i$ *iff* $v \in C_i$.
2. Every operator $o \in \mathcal{O}$ is assigned a level according to the level of the variables it modifies, i.e., $level(o) = i$ *iff* o modifies a variable v with $level(v) = i$. By Definition 38, it holds that all variables modified by the same operator are in the same component, and hence have the same level.

To show how stratified planning performs pruning, we need to define the notion of *follow-up operator.*

Definition 39 (follow-up operator). (Based on Chen et al. 2009 [CXY09]) Given a planning task $\Pi = (\mathcal{V}, \mathcal{O}, s_0, \mathcal{G})$ and operators $o, o' \in \mathcal{O}$, o' is a *follow-up operator* of o if $vars(o') \cap vars(\mathit{eff}(o)) \neq \emptyset$, i.e., o' reads or modifies a variable that is modified by o.

Stratified planning works as follows: Let s be a state reached by operator o, and o' is an applicable operator in s. Then o' is pruned in s if $level(o') > level(o)$ and o' is not a follow-up operator of o.

Again, it has been shown by Wehrle & Helmert that commutativity pruning dominates stratified planning in the sense that the former prunes more paths than the latter [WH12]. The idea of the proof is to show first that commutativity pruning prunes all paths pruned by stratified planning, and then show that commutativity pruning strictly dominates stratified planning in a particular situation. To illustrate, let a path $\sigma o o'$ be a path pruned by stratified planning. Now, consider the following facts:

[1]A strongly connected component of a directed graph is a maximal strongly connected subgraph.

1. Stratified planning prunes o' if it is not a follow-up operator of o and $level(o') > level(o)$. The first condition shows the independence between o' and o in one direction, i.e., o' does not read or modify a variable modified by o. In other words, o does not disable o', o cannot enable o' and they do not conflict. Second, from the connected components and imposing levels on variables and operators, the independence between o and o' holds in the other direction, i.e., o is not affected by o' in the same way explained for the first situation. Putting these two facts together, we have $o \bowtie o'$.

2. Consider defining the total order $<_{cp}$ for commutativity pruning such that $level(o) > level(o')$ implies $o <_{cp} o'$. By using this order, commutativity pruning prunes *all* paths pruned by stratified planning.

3. If two commutative operators exist in the same connected component, then stratified planning would not prune any path in which the two operators take part because they are in the same level. On the other hand, commutativity pruning has a more fine-grained ordering for operators such that the two operators must have different orderings, i.e., for operators o and o', either $o <_{cp} o'$ or $o' <_{cp} o$. For example, given operators o and o', where $o <_{cp} o'$, $o \bowtie o'$, and both operators belong to the same connected component in the causal graph, then $level(o) = leve(o')$. In this case, commutativity pruning prunes path $o'o$, while stratified planning preserves both paths oo' and $o'o$.

In a nutshell, stratified planning is a less powerful pruning technique than commutativity pruning due to imposing a less informative ordering on operators than the arbitrary total order used by commutativity pruning: Stratified planning does not offer any pruning power for operators with equal levels. Finally, we need to mention the relationship between sleep sets and stratified planning. From the fact that sleep sets strictly dominates commutativity pruning, and the latter strictly dominates stratified planning, we conclude that sleep sets strictly dominates stratified planning.

To summarize, we considered two previous transition reduction techniques that have been proposed in the context of optimal planning, and we have seen that sleep sets strictly dominates both.

6.4 Generalized Sleep Sets

As we have seen, sleep sets allow exploring only one permutation of commutative operators that lead to some state and prune all other permutations that lead to the same state. In previous work, a transition reduction technique, known as *moving pruning*, has been proposed in the context of single-agent search using the notion of *redundancy* between operator sequences [HB14]. On the other hand, sleep sets use commutativity which is a restricted form of redundancy. In this section, we show how sleep sets can be generalized to a family of transition reduction techniques by using different forms of redundancy. Furthermore, we will see the relationship between generalized sleep sets and move pruning. First, we introduce the notion of total orders on operator sequences.

Before presenting the generalizations, we need to consider the behavior of the sleep sets method. As we mentioned before, the original definition of sleep sets distinguishes between two different situations: Operator o is pruned after a path $\sigma = o_1 \dots o_n$ if $o \bowtie o_n$ and either $o <_{ss} o_n$ *or* $o \in ss(o_1 \dots o_{n-1})$. These two cases can explicitly be expressed by the following definitions. These definitions are skeletons that are used by the relations we are going to see afterwards.

Definition 40 (anchor point). An ω-anchor point for operator o in operator sequence $o_1 \dots o_n$ is an index $i \in \{1, \dots, n\}$ such that $o \omega o_i$ and $o <_{ss} o_i$.

Informally, a ω-anchor point corresponds to the first condition in the original definition of sleep sets.

Definition 41 (relay point). An ω-relay point for o in operator sequence $o_1 \dots o_n$ is an index with $2 \leq i \leq n$ such that $o \omega o_i$, and $(i-1)$ is a ω-anchor point or ω-relay point for o in $o_1 \dots o_{i-1}$.

Again, an ω-relay point reflects the second condition in the original definition. Furthermore, we state the definition of generalized sleep sets.

Definition 42 (generalized sleep set). A generalized sleep set for an operator sequence σ with respect to an ω-relation is a set of operators $gss^{\omega}(\sigma)$ such that:

- $gss^{\omega}(\sigma) = \emptyset$, if $\sigma = \varepsilon$, and
- $gss^{\omega}(\sigma) = \{o \mid n \text{ is an } \omega\text{-anchor or } \omega\text{-relay point for } o \text{ in } \sigma\}$, if $|\sigma| = n > 0$.

In all definitions, the symbol ω can be instantiated to one of several relations on operator sequences as we will see next. In the following, we show the whole family of generalized sleep sets starting with commutativity.

Commutativity $\bowtie$. We have previously seen the syntactic definition of sleep sets. In the following, a state-dependent definition is presented in order to have a comparable version to redundancy afterwards.

Given an operator sequence σ, we use $pre(\sigma)$ to denote the set of states in which σ can be applied, and $\sigma(s)$ to denote the state resulting from applying σ in a state $s \in pre(\sigma)$.

Definition 43 (commutative operators). Let o_1 and o_2 be two operators. We say that o_1 and o_2 are commutative, denoted as $o_1 \bowtie o_2$, *iff* the following holds for every state s:

1. $s \in pre(o_1)$ and $s \in pre(o_2) \implies o_1(s) \in pre(o_2)$ and $o_2(s) \in pre(o_1)$ and $o_1(o_2(s)) = o_2(o_1(s))$,
2. $s \in pre(o_1)$ and $s \notin pre(o_2) \implies o_1(s) \notin pre(o_2)$, and
3. $s \in pre(o_2)$ and $s \notin pre(o_1) \implies o_2(s) \notin pre(o_1)$.

To illustrate the connection with the syntactic definition, it is easy to see that the first condition implies that o_1 does not syntactically interfere with o_2, i.e., o_1 does not disable o_2, o_2 does not disable o_1, and they do not conflict. The second condition means that if $o_2 \notin app(s)$, then $o_2 \notin app(o_1(s))$, which is syntactically approximated by requiring that

o_1 cannot enable o_2. Similarly, the third condition is concerned with the applicability of o_1.

By replacing ω by $\bowtie$ in the previously stated definitions, we get $\bowtie$-anchor point, $\bowtie$-relay point, and consequently a generalized sleep set definition with respect to $\bowtie$, i.e., for operator sequence σ, a generalized sleep sets is denoted as $gss^{\bowtie}(\sigma)$.

Since both definitions (i.e., the original sleep sets definition and generalized sleep sets using $\bowtie$) use the same relation ($\bowtie$), these definition are equivalent, i.e., for all operator sequences σ, it holds that $gss^{\bowtie}(\sigma) = ss(\sigma)$.

Equivalence $\equiv$. The equivalence relation on operator sequences, denoted as $\equiv$, is a special case of redundancy, which will be defined later.

Definition 44 (equivalent operator sequences). Let σ and ρ be two operator sequences. We say that σ and ρ are equivalent, denoted as $\sigma \equiv \rho$, *iff* the following holds:

1. $cost(\sigma) = cost(\rho)$,
2. $pre(\sigma) = pre(\rho)$, and
3. $s \in pre(\sigma) \implies \sigma(s) = \rho(s)$.

It is obvious that $\sigma \equiv \rho$ holds if $\sigma \leq \rho$ and $\rho \leq \sigma$. Equivalence is more general than commutativity. The following lemma shows this fact.

Lemma 2. *For any two operators o_1 and o_2, $o_1 \bowtie o_2 \implies o_1o_2 \equiv o_2o_1$.*

Proof. By definition 44, $o_1o_2 \equiv o_2o_1$ *iff* $pre(o_1o_2) = pre(o_2o_1)$ and $s \in pre(o_1o_2) \implies o_1o_2(s) = o_2o_1(s)$. Since $o_1 \bowtie o_2$, it follows from Definition 43 that $pre(o_1o_2) = pre(o_2o_1) = (pre(o_1) \cap pre(o_2))$ and that $s \in (pre(o_1) \cap pre(o_2)) \implies o_2o_1(s) = o_1o_2(s)$ (condition 1 in Definition 43). □

Note that the implication does not always hold in the other direction: There can be operators such that $o_1o_2 \equiv o_2o_1$ but it does *not* hold that $o_1 \bowtie o_2$. This can happen if $pre(o_1o_2) = pre(o_2o_1) \neq (pre(o_1) \cap pre(o_2))$, i.e., there exists a state $s \in (pre(o_1) \cap pre(o_2))$ in which both o_1o_2 and o_2o_1 cannot be applied (o_1 disables o_2 and o_2 disables o_1).

Like commutativity, for an operator sequence σ, a generalized sleep set $gss^{\equiv}(\sigma)$ can be defined based on $\equiv$-anchor and $\equiv$-relay points.

However, this generalization does not result in more pruning than in the case of sleep sets using commutativity since the only difference between the two relations ($\equiv$ and $\bowtie$) is that the former allows operators that mutually disable each other, and hence, neither anchor nor relay points can introduce any pruning. For example, let o_1 and o_2 be operators such that o_1 disables o_2, o_2 disables o_1, and $o_1o_2 \equiv o_2o_1$. The operators are ordered as follows: $o_1 <_{ss} o_2$. Clearly, $o_1 \in gss^{\equiv}(o_2)$, but o_1 is not applicable after o_2. Similarly, propagating o_1 down the sequence would not result in any pruning because o_1 remains inapplicable until an enabling operator o' for o_1 is applied, but then the equivalence does not hold and o_1 is removed from $gss^{\equiv}(o_2 \ldots o')$. Thus, the equivalence relation does not add any pruning power to sleep sets.

Redundancy $\leq$. Similar to equivalence, redundancy is defined on operator sequences.

Definition 45 (redundant operator sequences). Let σ and ρ be two operator sequences. We say that ρ is redundant with σ, denoted with $\sigma \leq \rho$, *iff* the following holds:

1. $\mathit{cost}(\rho) \geq \mathit{cost}(\sigma)$
2. $\mathit{pre}(\rho) \subseteq \mathit{pre}(\sigma)$
3. $s \in \mathit{pre}(\rho) \implies \rho(s) = \sigma(s)$

This definition is general in the sense it considers any operator sequences of arbitrary length and operators in both sequences might differ. Next, we show several types of redundancy starting from the simplest to the most general:

Simple redundancy. This type of redundancy between operator sequences is restricted to two operators, e.g., $o_1o_2 \leq o_2o_1$. From Definition 45, it is obvious that even simple redundancy $\leq$ is more general than equivalence and commutativity. To explain this, consider the following example:

Let o_1 and o_2 be operators with the following components:

- $\mathit{pre}(o_1) = \{\top\}, \mathit{eff}(o_1) = \{a \mapsto 1\}$
- $\mathit{pre}(o_2) = \{a \mapsto 1\}, \mathit{eff}(o_2) = \{b \mapsto 1\}$

Clearly, $o_1o_2 \leq o_2o_1$ is true for the following reasons:

1. $\mathit{cost}(o_1o_2) = \mathit{cost}(o_2o_1)$,
2. $\mathit{pre}(o_2o_1) \subseteq \mathit{pre}(o_2o_1)$, and
3. $s \in \mathit{pre}(o_2o_1) \implies o_1o_2(s) = o_2o_1(s)$.

Condition 2 holds because o_1 is applicable in every state (its precondition is trivially satisfied), and o_2 has one precondition that is achieved by o_1. This means that o_1o_2 is applicable in every states. On the other hand, o_2o_1 is applicable only in states with $a = 1$. Condition 3 holds because o_1 and o_2 do not conflict on any variable.

Let's examine commutativity now. Let $s = \{a \mapsto 0, b \mapsto 0\}$ be a state. $o_1 \in \mathit{app}(s)$ and results in state $o_1(s) = \{a \mapsto 1, b \mapsto 0\}$. Obviously, $o_2 \notin \mathit{app}(s)$ but $o_2 \in \mathit{app}(o_1(s))$, which violates condition 2 in Definition 43. Therefore, o_1 and o_2 are not commutative. Alternatively, it is easy to syntactically check that o_1 can enable o_2 meaning that they are not commutative.

Flexible redundancy. A more general definition is to relax the left hand side of the inequality $o_1o_2 \leq o_2o_1$ such that o_2 can be replaced by an arbitrary operator x such that the inequality holds, i.e., $o_1x \leq o_2o_1$.

Practically, the drawback of computing a sleep set based on this type of redundancy is the linear overhead needed during the runtime step for checking all operators x instead of a fixed lookup for o_2.

Full flexible redundancy. By further relaxing the left hand side of the inequality such that both o_1 and o_2 can be replaced by arbitrary operators x and y, i.e., $xy \leq o_2 o_1$, we obtain the most general form of redundancy for operator sequences of length two.

The overhead during runtime is quadratic in the number of operators since we have free choices of operators for x and y.

Long distance leapfrogging. So far we have considered several forms of redundancy restricted to operator sequences of length two. Now we show that an operator can leapfrog over an entire sequence of operators of arbitrary length such that it does not need to leapfrog over intermediate operators in that sequence, i.e. , $oo_1 \dots o_n \leq o_1 \dots o_n o$ and $oo_i \nleq o_i o$ for all $i \in \{1, \dots, n\}$.

Given an operator sequence σ, $\overleftarrow{\sigma}$ is used to denote the prefix of σ up to the last operator in σ, i.e., if $|\sigma| \geq 1$, then $\overleftarrow{\sigma} = \sigma_1 \dots \sigma_{|\sigma|-1}$; $\overleftarrow{\varepsilon}$ is defined to be ε. Let $<_{gss}$ be a length-lexicographic order used by generalized sleep sets. The definitions of anchor and relay points for $\leq$ are as follows:

Definition 46. A $\leq$-anchor point for operator o in sequence $o_1 \dots o_n$ is an index $i \in \{1, \dots, n\}$ for which there exists an index $k \in \{1, \dots, i\}$, and an operator sequence α such that:

(A1) $\alpha \leq o_k \dots o_i o$, and

(A2) $\overleftarrow{\alpha} <_{gss} o_k \dots o_i$

Definition 47. A $\leq$-relay point for operator o in sequence $o_1 \dots o_n$ is an index $i \in \{2, \dots, n\}$ for which there exists an index $k \in \{2, \dots, i\}$, operator x, and operator sequence α such that:

(R1) $x\alpha \leq o_k \dots o_i o$,

(R2) $|\alpha| \leq |o_k \dots o_i|$, and

(R3) $(k-1)$ is a $\leq$-anchor or $\leq$-relay point for x in $o_1 \dots o_{k-1}$.

6.4.1 Generalized Sleep Sets are Safe

In this section, we show that generalized sleep sets based on $\leq$ are completeness and optimality preserving. For a state s, we use $gss(s)$ instead of $gss^{\leq}$ for simplicity. Furthermore, we use anchor points and relay points instead of $\leq$-anchor and $\leq$-relay points, respectively. All the following have been shown by Holte et al. [HAW15][2].

Lemma 3. *The relation $\leq$ is transitive.*

Proof. Let α, β, γ be operator sequences such that $\alpha \leq \beta$ and $\beta \leq \gamma$. We show that $\alpha \leq \gamma$. Consider the three conditions in the redundancy definition:

1. $cost(\alpha) \leq cost(\gamma)$ holds because $cost(\alpha) \leq cost(\beta) \leq cost(\gamma)$.

2. $pre(\gamma) \subseteq pre(\alpha)$ holds because of $pre(\gamma) \subseteq pre(\beta) \subseteq pre(\alpha)$.

3. $s \in pre(\gamma) \implies \alpha(s) = \gamma(s)$ holds because $s \in pre(\gamma) \implies s \in pre(\beta)$, $\gamma(s) = \beta(s)$, and $s \in pre(\beta) \implies \beta(s) = \alpha(s)$. Therefore $\gamma(s) = \alpha(s)$ and hence, $\alpha \leq \gamma$.

[2]We slightly reformulated the content to be consistent with the style of this thesis.

□

Lemma 4. *Let α and β be any operator sequences such that $\alpha \leq \beta$, and let γ be any operator sequence. then $\alpha\gamma \leq \beta\gamma$ and $\gamma\alpha \leq \gamma\beta$.*

Proof. We show that $\alpha\gamma \leq \beta\gamma$ (the proof for $\gamma\alpha \leq \gamma\beta$ is analogous). For this purpose, we need to show the following:

1. It holds that $cost(\alpha\gamma) \leq cost(\beta\gamma)$ because $cost(\alpha\gamma) = cost(\alpha) + cost(\gamma) \leq cost(\beta) + cost(\gamma) = cost(\beta\gamma)$.
2. It holds that $s \in pre(\beta\gamma) \implies s \in pre(\beta)$ and $\beta(s) \in pre(\gamma)$. From $\alpha \leq \beta$ it follows that $s \in pre(\alpha)$ and $\alpha(s) \in pre(\gamma)$, i.e., that $s \in pre(\alpha\gamma)$.
3. $s \in pre(\beta\gamma) \implies \beta\gamma(s)$ and $\alpha\gamma(s)$ are both defined. $\beta\gamma(s) = \gamma(\beta(s)) = \gamma(\alpha(s)) = \alpha\gamma(s)$.

□

Lemma 5. *Let α, β, γ, δ, and σ be any operator sequences and let $<_o$ be a length-lexicographic order on operator sequences such that $\alpha <_o \beta$ and $|\gamma| \leq |\delta|$. Then $\sigma\alpha\gamma <_o \sigma\beta\delta$.*

Proof. If $|\alpha| < |\beta|$ or $|\gamma| < |\delta|$ then $|\sigma\alpha\gamma| < |\sigma\beta\delta|$ and therefore $\sigma\alpha\gamma <_o \sigma\beta\delta$. Alternatively, if $|\alpha| = |\beta|$ and $|\gamma| = |\delta|$, then $|\sigma\alpha\gamma| = |\sigma\beta\delta|$. If o^α and o^β are the leftmost operators where α and β differ, i.e., o^α in α and o^β in the corresponding position in β, then o^α and o^β are also the leftmost operators where $\sigma\alpha\gamma$ and $\sigma\beta\delta$ differ. From $\alpha <_o \beta$ we have $o^\alpha <_o o^\beta$ and therefore $\sigma\alpha\gamma <_o \sigma\beta\delta$. □

Theorem 10. *Let $\sigma = o_1 \dots o_n$ be an non-empty operator sequence, where $n \geq 1$, and o be an operator. If $o \in gss(\sigma)$, then there exists an operator sequence σ' such that:*

(P1) $\sigma' \leq \sigma o$,

(P2) $\overleftarrow{\sigma'} <_{gss} \sigma$.

Proof. The proof is shown by induction on n.
Base case: $n = 1$. Because $gss(\varepsilon) = \emptyset$, n must be an anchor point for o in σ (n cannot be a relay point), i.e., there exists k and α such that A1 and A2 hold with $i = n = 1$ and $o' = o$. $k = 1$ because $n = 1$. By setting σ' to be α, we get that P1 and P2 and exactly A1 and A2, respectively. Therefore, the theorem holds for the base case. Inductive case: Assume that the theorem holds for all m, $1 \leq m \leq n$. We show that it holds for $n + 1$. Let $\sigma = o_1 \dots o_{n+1}$ be an operator sequence of length $n + 1$ and o be an operator such that $o \in gss(\sigma)$.

If $n + 1$ is an anchor point for o in σ, then there exist k and α such that A1 and A2 hold with $i = n + 1$ and $o' = o$. By setting σ' to $o_1 \dots o_{k-1}\alpha$, the theorem holds for the following reasons:

- P1 follows directly from A1 by appending $o_1 \dots o_{k-1}$ to both sides of the inequality of A1 (Lemma 4).
- If $\alpha \neq \varepsilon$, then $\overleftarrow{\sigma'}$ is equal to $o_1 \dots o_{k-1}\overleftarrow{\alpha}$ and P2 follows from A2 and Lemma 5 by appending $o_1 \dots o_{k-1}$ to both sides of the inequality in A2. If $\alpha = \varepsilon$, then P2 holds trivially because $\overleftarrow{\sigma'} = o_1 \dots o_{k-2}$ is a prefix of σ.

If $n+1$ is not an anchor point for o in σ, it must be a relay point. Therefore, there exist k, z and α such that R1, R2, and R3 hold with $i = n+1$ and $o' = o$. From R3 and the inductive hypothesis, there exists an operator sequence $\overline{\sigma'} = \overline{o'_1} \dots \overline{o'}_{|\overline{\sigma'}|}$ such that:

($\overline{P1}$) $\overline{\sigma'} \leq o_1 \dots o_{k-1} z$,

($\overline{P2}$) $\overleftarrow{\overline{\sigma'}} <_{gss} o_1 \dots o_{k-1}$.

By setting σ' to $\overline{\sigma'}\alpha$ the requirements of the theorem are satisfied for the following reasons:

- We need to show $\sigma' \leq \sigma o$, i.e., that $\overline{\sigma'}\alpha \leq \sigma o$. From $\overline{P1}$, we have $\overline{\sigma'} \leq o_1 \dots o_{k-1} z$. By appending α to both sides of the inequality (Lemma 4), we get $\overline{\sigma'}\alpha \leq o_1 \dots o_{k-1} z\alpha$. By R1 and Lemma 4 (appending $o_1 \dots o_{k-1}$ to both sides of the inequality), we have $o_1 \dots o_{k-1} z\alpha \leq o_1 \dots o_{k-1} o_k \dots o_{n+1} o = \sigma o$. By the transitivity of $\leq$, we have $\sigma' = \overline{\sigma'}\alpha \leq \sigma o$, i.e. P1 is true.

- We need to show that $\overleftarrow{\sigma'} <_{gss} \sigma$. Let $\gamma = \overleftarrow{(\tilde{o}\alpha)}$, where $\tilde{o}$ is the last operator in $\overline{\sigma'}$ if $\overline{\sigma'} \neq \varepsilon$, and $\tilde{o} = \varepsilon$ if $\overline{\sigma'} = \varepsilon$. Then $|\gamma| = |\overleftarrow{(\tilde{o}\alpha)}| \leq |\alpha|$. Combining this with R2, we get $|\gamma| \leq |o_k \dots o_{n+1}|$. Using this fact in Lemma 5 together with the inequality $\overleftarrow{\overline{\sigma'}} <_{gss} o_1 \dots o_{k-1}$ from $\overline{P2}$ we have $\overleftarrow{\overline{\sigma'}}\gamma <_{gss} o_1 \dots o_{k-1} o_k \dots o_{n+1} = \sigma$. But $\overleftarrow{\overline{\sigma'}}\gamma = \overleftarrow{(\overline{\sigma'}\alpha)}$, hence P2 is true.

□

Theorem 11. *Let s and t be states such that t is reachable from s. Let $\sigma_{k-1} o_k \rho_{k+1} = min(s,t)$ for all k, $1 \leq k \leq |min(s,t)|$. Then, $o_k \notin gss(\sigma_{k-1})$ for all k, $1 \leq k \leq |min(s,t)|$.*

Proof. The theorem holds for $k = 1$, since $\sigma_0 = \varepsilon$ and $gss(\varepsilon) = \emptyset$. For $k \geq 2$, if $o_k \in gss(\sigma_{k-1})$ then, by Theorem 10, there exists an operator sequence σ' such that:

(P1) $\sigma' \leq \sigma_{k-1} o_k = \sigma_k$,

(P2) $\overleftarrow{\sigma'} <_{gss} \sigma_{k-1}$.

Let $\sigma = \sigma' \rho_{k+1}$. By appending ρ_{k+1} to both sides of the inequality in $P1$ (Lemma 4), we get $\sigma = \sigma' \rho_{k+1} \leq \sigma_k \rho_{k+1} = min(s,t)$. Hence, σ is a least-cost path from s to t. Let o' be the last operator in σ' if $\sigma' \neq \varepsilon$, and $o' = \varepsilon$ otherwise. By appending $o' \rho_{k+1}$ to the left hand side of the inequality in P2, and $o_k \rho_{k+1}$ to the right hand side, by Lemma 5, we get $\sigma' \rho_{k+1} <_{gss} \sigma_k \rho_{k+1}$, i.e., $\sigma <_{gss} min(s,t)$. There two facts about σ contradict $min(s,t)$ being the least-cost path that is smallest according to $<_{gss}$. □

6.4.2 Generalized Sleep Sets and Move Pruning

Burch and Holte introduced move pruning as an optimality-preserving duplicate pruning technique for single-agent search problems. The method has been designed to detect sequences of moves that result in duplicate states in tree search algorithms like DFS or IDA* [BH11; BH12; HB14]. Move pruning builds on an earlier algorithm that has been introduced by Taylor and Korf to reduce redundancy in some combinatorial games like the N-puzzle sliding tile puzzle and Rubik's cube [TK93]. Like sleep sets, move pruning relies on redundancy between operator sequences and need a total ordering on operators.

Let $<_{mp}$ be a nested order on operator sequences used by move pruning, σ be an operator sequence, and o be an operator. Furthermore, let δ be an operator sequence such that $\delta <_{mp} \sigma o$ and $\delta \leq \sigma o$, then move pruning prunes (does not apply) o after σ. The safety proof is given in [HB14]. Similar to sleep sets, the implementation of move pruning has two steps: A preprocessing step where the redundancy relation is computed for operator sequences of length L or less (L is a parameter), and a runtime step in which operators are pruned during the search. For instance, for operators $o_1, \dots, o_n$ with unit costs, move pruning checks $o_i o_j \leq o_k o_l$ and $o_i \leq o_j o_k$, for $i, j, k, l \in \{1, \dots, n\}$.

Let $\sigma_1 = o_1 \dots o_n$ and $\sigma_2 = o'_1 \dots o'_m$ be operator sequences such that $n \leq m \leq L$. Let $s \in pre(\sigma_1) \cap pre(\sigma_2)$ be a state. If $\sigma_1 \leq \sigma_2$ and $\sigma_1 <_{mp} \sigma_2$, then operator o'_m is not applied in state $o'_1 \dots o'_{m-1}(s)$. To be comparable to generalized sleep sets, move pruning can be redefined in terms of *move pruning points.*

Definition 48. A move pruning point for operator o in operator sequence $o_1 \dots o_n$ is an index i where $1 \leq i \leq n$ for which there exists an index k where $1 \leq k \leq i$ and an operator sequence λ such that:

(MP1) $\lambda \leq o_k \dots o_i o$, and

(MP2) $\lambda <_{mp} o_k \dots o_i o$.

If a point in a path is a move pruning point for operator o, then o is pruned. This is similar to the fact that sleep sets would prune o at anchor points. However, there is an interesting difference between move pruning points and anchor points. If we consider definitions 46 and 48, we notice the difference between rules (A2) and (MP2). Let σ_1 and σ_2 be two operator sequences such that $\sigma_1 \leq \sigma_2$. Assume that operator orderings $<_{gss}$ and $<_{mp}$ are identical. Definition 46 requires $\overleftarrow{\sigma_1} <_{gss} \overleftarrow{\sigma_2}$ (A2), while Definition 48 requires $\sigma_1 <_{mp} \sigma_2$ (MP2). Indeed, rule (A2) implies (MP2) but not vice versa. For example, given operators o, o' and o'' such that $o <_{mp} o' <_{mp} o''$, then $oo' <_{mp} oo''$ is true although $o <_{mp} o$ is false. Clearly, move pruning points are more general than anchor points, and hence they can prune paths that anchor points cannot prune. On the other hand, relay points of sleep sets have the property that they can prune path of arbitrary length by propagating operators along paths, while the length of pruned paths is bounded by L for move pruning.

In a nutshell, we see that generalized sleep sets and move pruning are pairwise incomparable in terms of pruning power. We have also shown that generalized sleep sets dominates the original sleep sets method in terms of pruning power.

Chapter 7

Combining Sleep Sets with Stubborn Sets

While stubborn sets are directed towards goal states due to employing disjunctive action landmarks in their definition, sleep sets have no clue about the direction of goal states but rather they rely on the information gathered throughout the search. In his monograph, Godfroid proposed an algorithm for combining sleep sets with a state reduction technique called *persistent sets* [God96]. A persistent set is a general notion used to define the reduced set of transitions produced by state reduction techniques (like stubborn sets) [Ove81; Val89]. The intuition behind this combination is that persistent sets are sometimes forced to include some transitions that can safely be pruned according to sleep sets since they lead to redundant states. In the following, we see how sleep sets can be combined with stubborn sets in a straightforward fashion. Afterwards, we show that a tighter integration of both techniques can be unsafe.

7.1 Loose Integration

The algorithm that Godefroid suggested, combines sleep sets and persistent sets in a straightforward fashion: For a given state s, a persistent set (e.g., a stubborn set) and a sleep set for s are independently computed, then only the applicable transitions that belong to the persistent set but outside the sleep set are used to expand s. We followed this method to combine A^*_{ss} with state pruning.

Algorithm 10 shows how A* search can be combined with sleep sets and stubborn sets. We call this algorithm A^*_{sssss}. The only difference to Algorithm 7 appears in line 10, where $app(s)$ are restricted to the applicable operators in a stubborn set T_s, which is computed according to Algorithm 1, then the new resulting $app(s)$ is restricted again to the operators outside the sleep set (line 11). The next theorem shows the safety of A^*_{sssss}. For $[\sigma]_\equiv$, we use $\Sigma^{s,\sigma} := \{o^i_1 \mid \sigma \in [\sigma]_\equiv, \sigma = o^i_1 o^i_2 \dots o^i_{n_i}\}$ to denote the *set of initial operators* of $[\sigma]_\equiv$, i.e., the set that contains the first operators of all paths in $[\sigma]_\equiv$.

Theorem 12. *[AW16] Let s be a state, and let s_g be a goal state reachable from s via operator sequence σ (i.e., $\sigma(s) = s_g$). Let $\sigma_1, \dots, \sigma_n$ be the paths*

Algorithm 10 A$^*_{sssss}$

```
1: $Open \leftarrow \emptyset$; $Closed \leftarrow \emptyset$
2: $n_0 \leftarrow make\_node(s_0)$
3: $Open.insert(n_0)$;
4: while $Open \neq \emptyset$ do
5:     $n \leftarrow Open.pop\_min()$
6:     $s \leftarrow n.get\_state()$
7:     if $is\_goal(s)$ then
8:         $plan \leftarrow \sigma^s_{min}$    /*minimal cost generating path of $s$*/
9:         return SOLVED
10:    $app(s) \leftarrow app(s) \cap T_s$
11:    $app(s) \leftarrow app(s) \setminus ss(\sigma^s_1, \dots, \sigma^s_n)$
12:    EXPAND$(s, app(s), ss(\sigma^s_1, \dots, \sigma^s_n))$
13: return UNSOLVED
```

explored by A^*_{sssss} that generated s in this particular order before termination (i.e., s is generated by σ_1 first, and by σ_n last).

If $\Sigma^{s,\sigma} \cap ss(\sigma_1, \dots, \sigma_n) = \emptyset$, then there is a permutation $\bar{\sigma} \in [\sigma]_\equiv$ that is preserved by A^*_{sssss}.

Before giving the proof, let us discuss the claim and its implications in some more detail. Theorem 12 states that if the (updated) sleep set of a state s eventually does not contain any first operator of the sequences in $[\sigma]_\equiv$, then at least one of these sequences is preserved. As discussed by Godefroid, this particularly implies the completeness of A$^*_{sssss}$ because the sleep set of the initial state is empty by definition. In addition, we observe that A$^*_{sssss}$ remains optimal because for all solutions, at least one permutation is preserved.

Proof. Consider the permutation equivalent paths $[\sigma]_\equiv$ of σ, and the set of initial operators $\Sigma^{s,\sigma}$ of $[\sigma]_\equiv$. We show by induction on the length of σ that at least one permutation sequence $\bar{\sigma} \in [\sigma]_\equiv$ is preserved by A$^*_{sssss}$. If $|\sigma| = 0$, the result is immediate.

If $|\sigma| > 0$, then there is an operator sequence of length $|\sigma|$ from s to s_g in the state space induced by A* and strong stubborn sets. The proof will show that such an operator sequence to reach s_g still exists in the state space induced by A$^*_{sssss}$.

First, we observe that there is $o \in \Sigma^{s,\sigma}$ that is applied by A$^*_{sssss}$ in s: To see this, consider the first sequence σ^s_k $(1 \leq k \leq n)$ by which state s is generated such that $\Sigma^{s,\sigma} \cap ss(\sigma^s_1, \dots, \sigma^s_k) = \emptyset$ (i.e., $\Sigma^{s,\sigma} \cap ss(\sigma^s_1, \dots, \sigma^s_i) \neq \emptyset$ for $1 \leq i \leq k-1$). Such σ^s_k must exist because $\Sigma^{s,\sigma} \cap ss(\sigma^s_1, \dots, \sigma^s_n) = \emptyset$ by assumption, and by definition, sleep sets can only reduce when a state is revisited.

Now consider the expansion process of s when s is reached by σ^s_k. Let o be the operator in $\Sigma^{s,\sigma}$ that is applied in s and is smallest among the remaining operators in $\Sigma^{s,\sigma}$ (according to $<_{ss}$) that have not yet been applied in s. Such an operator must exist because $\Sigma^{s,\sigma} \cap ss(\sigma^s_1, \dots, \sigma^s_{k-1}) \neq \emptyset$. Let $s' := o(s)$. As $o \in \Sigma^{s,\sigma}$, the goal state s_g is reachable from s' with an operator sequence σ' with $|\sigma'| = |\sigma| - 1$.

Consider the paths $\rho_1, \dots, \rho_t$ explored by A$^*_{sssss}$ that generate s'. To conclude the inductive proof argument, we will show (by contradiction) that $\Sigma^{s',\sigma'} \cap$

$ss(\rho_1^{s'}, \dots, \rho_t^{s'}) = \emptyset$. Assume $\Sigma^{s',\sigma'} \cap ss(\rho_1^{s'}, \dots, \rho_t^{s'}) \neq \emptyset$. Then there exists an operator $\bar{o} \in \Sigma^{s',\sigma'}$ with $\bar{o} \in ss(\rho_1^{s'}, \dots, \rho_m^{s'})$ for all $1 \leq m \leq t$. In particular, $\bar{o} \in ss(\rho_1^{s'}, \dots, (\sigma_k o)^{s'})$, which implies that o and $\bar{o}$ are commutative. It follows that $\bar{o}$ is applicable in s (because $\bar{o}$ is applicable in s', and $\bar{o}$ is not disabled by o), and furthermore, $\bar{o}$ is an initial operator of a permutation of σ which leads to s_g, i.e., $\bar{o} \in \Sigma^{s,\sigma}$. On the other hand, as $\bar{o} \in ss(\rho_1^{s'}, \dots, (\sigma_k o)^{s'})$, it follows that $\bar{o} \in ss(\sigma_1^s, \dots, \sigma_k^s)$ already (and $\bar{o}$ is propagated to $ss(\rho_1^{s'}, \dots, (\sigma_k o)^{s'})$ afterwards), or $\bar{o}$ has been added to $ss(\rho_1^{s'}, \dots, (\sigma_k o)^{s'})$ after applying o in s (meaning that $\bar{o}$ is applied before o in s, i.e., $\bar{o} <_{ss} o$). However, both of these cases cannot happen: The former case is a contradiction to the fact that $\bar{o} \in \Sigma^{s,\sigma}$ (because $\Sigma^{s,\sigma} \cap ss(\sigma_1^s, \dots, \sigma_k^s) = \emptyset$), and the latter case is a contradiction to the choice of o being the smallest operator according to $<_{ss}$. The induction continues from s' until s_g is reached. □

Corollary 5. *[AW16] A^*_{sssss} inherits the completeness and optimality properties from A^*_{ss}.*

Proof. Completeness follows because the sleep set of the initial state is empty by definition. Optimality follows because for *every* solution π, a permutation of π is preserved, hence in particular for every optimal solution. □

In the following, we see an example of the loose integration of sleep sets and stubborn sets.

Example 11. Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task, where

- $\mathcal{V} = \{a, b, c, g\}$
- $\mathcal{O} = \{o_1, o_2, o_3\}$, where
  - $pre(o_1) = \{\top\}$, $eff(o_1) = \{a \mapsto 1, c \mapsto 1\}$
  - $pre(o_2) = \{\top\}$, $eff(o_2) = \{b \mapsto 1, c \mapsto 1\}$
  - $pre(o_3) = \{c \mapsto 1\}$, $eff(o_3) = \{a \mapsto 0, g \mapsto 1\}$
- $s_0 = \{a \mapsto 0, b \mapsto 0, c \mapsto 0, g \mapsto 0\}$
- $s_\star = \{g \mapsto 1\}$

Furthermore, the total order on $\mathcal{O}$ is given as follows: $o_1 <_{ss} o_2 <_{ss} o_3$.

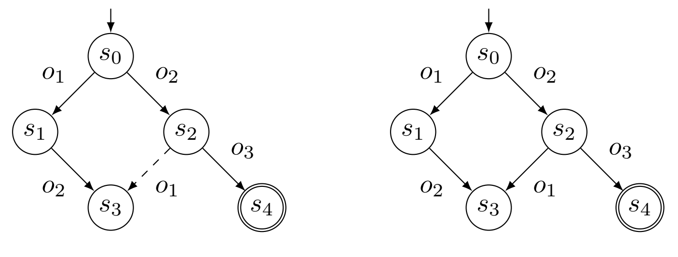


Figure 7.1: Operator o_1 is pruned by sleep sets but not by stubborn sets.

This example shows that sleep sets can safely prune operators that cannot be pruned by stubborn sets. In the initial state s_0, neither sleep sets nor stubborn sets prune anything: For sleep sets, $ss(\varepsilon) = \emptyset$. For stubborn sets, T_{s_0} is initialized with o_3 as a disjunctive action landmark. $o_3 \notin app(s_0)$ and hence o_1 and o_2 are added to T_{s_0} as a necessary enabling set. Therefore, both o_1 and o_2 will be applied in s_0. Now assume that the search selects state s_2 to be expanded next. Obviously, operators o_1 and o_2 are commutative ($o_1 \bowtie o_2$). Because $o_1 <_{ss} o_2$, $o_1 \in ss(o_2)$ and o_1 can safely be pruned at state s_2. Now, consider the computation of a stubborn set T_{s_2} for state s_2. First, T_{s_2} is initialized with o_3 as a disjunctive action landmark for $s_\star$ in s_2. Indeed, o_3 is applicable in s_2; therefore, interfering operators need to be added to T_{s_2}. Consequently, o_1 is added to T_{s_2} since it conflicts with o_3 on variable a. Figure 7.1 illustrates the example.

7.2 Tight Integration

The following interesting theoretical question arose throughout the research on sleep sets and stubborn: Is there a way to integrate both techniques in a tighter fashion? i.e., can one method make use of the computation of the other? Let us consider the idea of utilizing sleep sets in the computation of stubborn sets. For instance, for a state s, an operator o is excluded from being added to the stubborn set of s if o is already in the sleep set of the path that ended in s, even though o is member of a selected disjunctive action landmark, a necessary necessary enabling set, or an interferer of another operator which is already included in the stubborn set. The following counterexample shows that it is unsafe to utilize sleep sets in the computation of stubborn sets directly.

Example 12. Let $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task, where

- $\mathcal{V} = \{a, b, c, g\}$
- $\mathcal{O} = \{o_1, o_2, o_3\}$, where
  - $pre(o_1) = \{\top\}$, $eff(o_1) = \{a \mapsto 1, c \mapsto 1\}$
  - $pre(o_2) = \{\top\}$, $eff(o_2) = \{b \mapsto 1\}$
  - $pre(o_3) = \{b \mapsto 1, c \mapsto 0\}$, $eff(o_3) = \{g \mapsto 1\}$
- $s_0 = \{a \mapsto 0, b \mapsto 0, c \mapsto 0, g \mapsto 0\}$
- $s_\star = \{a \mapsto 1, b \mapsto 1, g \mapsto 1\}$

The total order on $\mathcal{O}$ is given as follows: $o_1 <_{ss} o_2 <_{ss} o_3$.

The only possible plan for this planning task is $o_2 o_3 o_1$. For state s_0, variable a is selected as a seed for a disjunctive action landmark, which leads to including o_1 in T_{s_0}. Since o_1 is applicable, o_3 is added to T_{s_0} as it interferes with o_1 (o_1 disables o_3 on variable c). Finally, o_2 is included in T_{s_0} as a necessary enabling set for o_3 (o_3 is inapplicable in s_0).

Let's consider now state s_2 in Figure 7.2. Due to commutativity of o_1 and o_2, operator o_1 is in the sleep set $ss(o_2)$. If variable a has been selected as the seed of a disjunctive action landmark at s_2, then o_1 needs to be included in the stubborn set, but $o_1 \in ss(o_2)$, therefore, it would be excluded from T_s which results in an empty stubborn set, thereby pruning the only plan. Consequently,

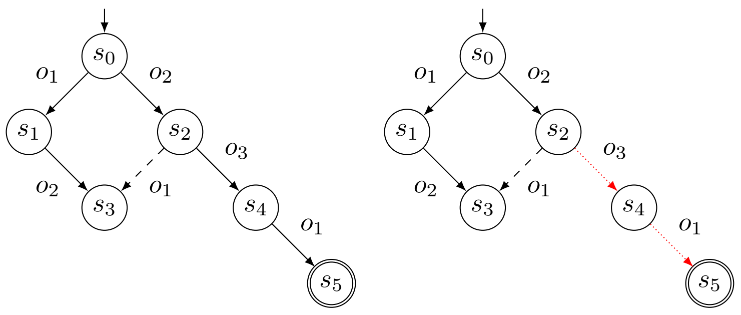


Figure 7.2: Operator o_3 is pruned in s_2 when o_1 is not added to T_{s_2}.

the direct integration of sleep sets into the computation of stubborn sets is unsafe.

In future work, it would be interesting to investigate a safe way to utilize sleep sets for computing stubborn sets.

Chapter 8

Experimental Evaluation

In this chapter, we experimentally evaluate the partial order reduction techniques presented in this thesis. All techniques are implemented on top of the planning system Fast Downward [Hel06].

Next, we will see the empirical evaluation of stubborn sets (strong and weak).

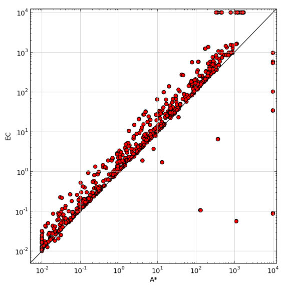
8.1 Strong Stubborn Sets vs. Expansion Core

All state reduction techniques are implemented with A^* search algorithm. The strong stubborn sets method (SSS) is evaluated by comparing it to pure A^* (without partial order reduction) and to the expansion core method (EC). Furthermore, the variant of SSS that dominates EC (SSS-EC) is also evaluated (mentioned briefly in section 4.3 and in details in [Weh+13]). The variant of SSS used for this experiment selects the first unsatisfied precondition or goal fact as a basis for adding a necessary enabling set or a disjunctive action landmark, respectively. Three sets of experiments have been performed for all configurations: A^* is guided by the landmark-cut heuristic [HD09], A^* with the merge-and-shrink heuristic [DFP09; HHH07], and A^* as blind search.

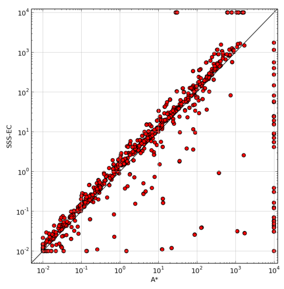
Table 8.1 provides an evaluation of the configurations mentioned above using the landmark cut heuristic. Table 8.2 shows the evaluation of all configurations using the merge-and-shrink heuristic. Finally, Table 8.3 is an overview of the results using blind search. The domains listed in the tables are those where coverage (i.e., number of solved problems) is not the same for all algorithms. The rest of the domains are counted in "remaining domains". The number of generated states is provided for the instances that are solved by all configurations, i.e., by pure A^* as well as by EC, SSS-EC and SSS.

Figures 8.1, 8.2, and 8.3 are scatterplots that show the search time for all configurations with the landmark-cut heuristic, the merge-and-shrink heuristic, and the blind heuristic, respectively.

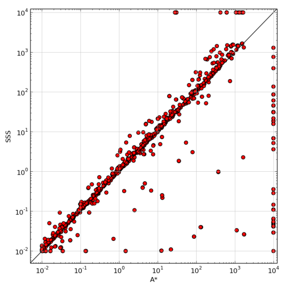
Experiment setting and benchmarks. The experiments are performed on a cluster with Intel Xeon E5-2650v2 2.6 GHz CPUs, with a timeout of 30 minutes and a memory bound of 2 GB per run. The benchmarks are all optimal STRIPS planning instances from the International Planning Competitions (IPCs) up to 2014, with an overall number of 79 domains and 2107 problems.



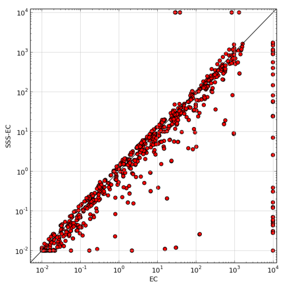
(a) A* vs. EC



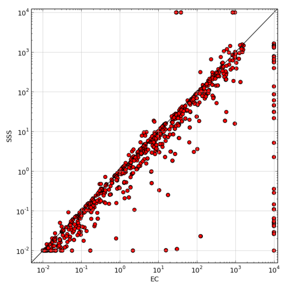
(b) A* vs. SSS-EC



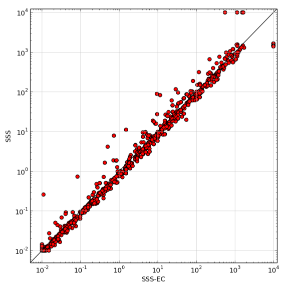
(c) A* vs. SSS



(d) EC vs. SSS-EC

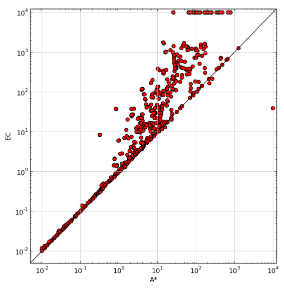


(e) EC vs. SSS

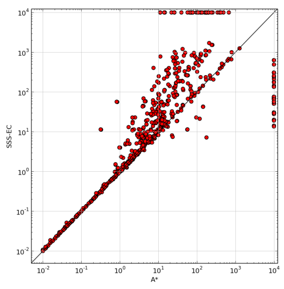


(f) SSS-EC vs. SSS

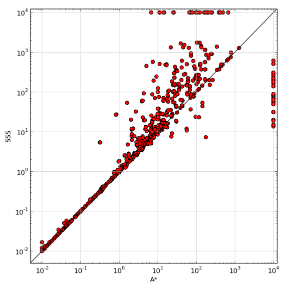
Figure 8.1: Search time for state reduction techniques using the landmark-cut heuristic.



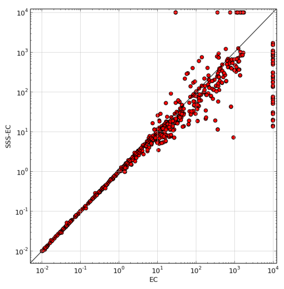
(a) A* vs. EC



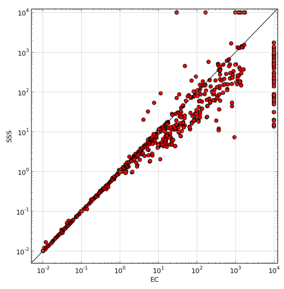
(b) A* vs. SSS-EC



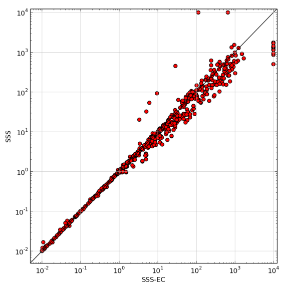
(c) A* vs. SSS



(d) EC vs. SSS-EC



(e) EC vs. SSS

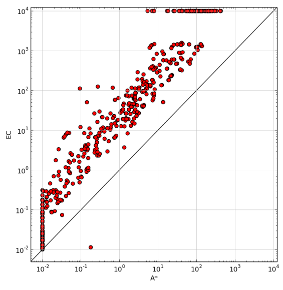


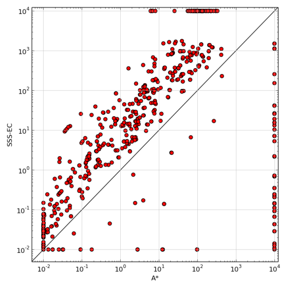
(f) SSS-EC vs. SSS

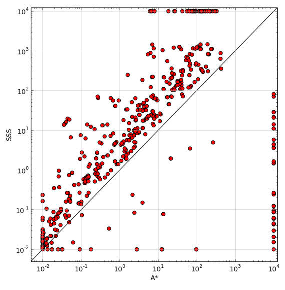
Figure 8.2: Search time for state reduction techniques using the merge-and-shrink heuristic.

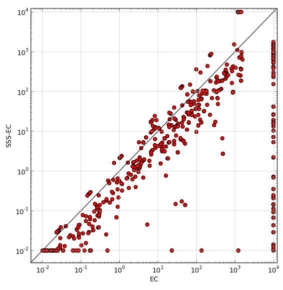
	Coverage				Nodes generated			
Domain (problems)	A*	+EC	+SSS-EC	+SSS	A*	+EC	+SSS-EC	+SSS
PARCPRINTER-08 (30)	19	−1	**+11**	**+11**	2452034	100%	**<1%**	**<1%**
PARCPRINTER-OPT11 (20)	14	−1	**+6**	**+6**	2452025	100%	**<1%**	**<1%**
WOODWORKING-OPT08 (30)	17	+6	**+10**	**+10**	6109688	9%	**<1%**	**<1%**
WOODWORKING-OPT11 (20)	12	+4	**+7**	**+7**	6109399	9%	**<1%**	**<1%**
SATELLITE (36)	7	±0	**+5**	**+5**	5057697	83%	**2%**	**2%**
ROVERS (40)	8	−1	**+1**	**+1**	1894389	99%	**22%**	24%
OPENSTACKS-OPT08 (30)	**21**	−4	−2	−1	4324209	100%	**63%**	**63%**
OPENSTACKS-OPT11 (20)	**16**	−4	−2	−1	4300589	100%	**63%**	**63%**
OPENSTACKS-OPT14 (20)	**3**	−2	**±0**	**±0**	333319	100%	**64%**	**64%**
TIDYBOT-OPT14 (20)	**10**	−2	−1	−2	359568	100%	**75%**	**75%**
HIKING-OPT14 (20)	**9**	**±0**	−1	**±0**	7791154	100%	**99%**	**99%**
LOGISTICS00 (28)	20	±0	**+1**	±0	12849032	94%	**19%**	99%
FREECELL (80)	**15**	**±0**	**±0**	−1	12318970	**100%**	**100%**	**100%**
PARKING-OPT11(20)	**3**	−1	−1	−1	560427	**100%**	**100%**	**100%**
PEGSOL-08 (30)	**28**	−1	−1	**±0**	6039901	**100%**	**100%**	**100%**
PEGSOL-OPT11 (20)	**18**	−1	−1	**±0**	6402682	**100%**	**100%**	**100%**
SCANALYZER-08 (30)	**15**	**±0**	−3	−3	13942542	**100%**	**100%**	**100%**
SCANALYZER-OPT11 (20)	**12**	**±0**	−3	−3	13942534	**100%**	**100%**	**100%**
SOKOBAN-OPT08 (30)	**30**	−2	−2	−1	20469867	**100%**	**100%**	**100%**
TETRIS-OPT14 (17)	**6**	−1	−1	−1	1280030	**100%**	**100%**	**100%**
TPP (30)	**7**	−1	−1	**±0**	233191	**100%**	**100%**	**100%**
VISITALL-OPT11 (20)	**11**	−1	−1	**±0**	1991026	**100%**	**100%**	**100%**
REMAINING DOMAINS (1496)	**592**	**±0**	**±0**	**±0**	476297502	99.9%	**89%**	93%
SUM (2107)	893	−14	+21	**+26**	607511775	98%	**85%**	90%

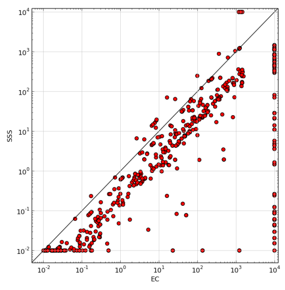
Table 8.1: Results overview for the landmark-cut heuristic.


(a) A* vs. EC


(b) A* vs. SSS-EC


(c) A* vs. SSS


(d) EC vs. SSS-EC


(e) EC vs. SSS

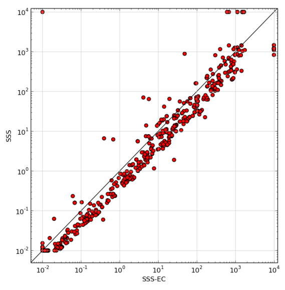

(f) SSS-EC vs. SSS

Figure 8.3: Search time using the blind heuristic.

Coverage. In Table 8.1, the dominance of SSS-EC over EC is obvious by observing the number of solved instances by both configurations: SSS-EC solved

	Coverage				Nodes generated			
Domain (problems)	A*	+EC	+SSS-EC	+SSS	A*	+EC	+SSS-EC	+SSS
PARCPRINTER-08 (30)	14	±0	**+11**	**+11**	69744790	100%	**<1%**	**<1%**
PARCPRINTER-OPT11 (20)	10	±0	**+8**	**+8**	69744790	100%	**<1%**	**<1%**
WOODWORKING-OPT08 (30)	13	+1	**+7**	**+7**	37957178	28%	**<1%**	1%
WOODWORKING-OPT11 (20)	8	±0	**+6**	**+6**	37957178	28%	**<1%**	1%
ROVERS (40)	8	±0	**+1**	**+1**	267804769	97%	**8%**	9%
PATHWAYS-NONEG (30)	4	±0	**+1**	**+1**	9372	100%	**16%**	19%
SATELLITE (36)	6	±0	**+1**	**+1**	1550	94%	**28%**	29%
MYSTERY (30)	**16**	−1	−1	−1	14836242	100%	**32%**	44%
OPENSTACKS-OPT08 (30)	**20**	−3	−1	**±0**	4969076	100%	**60%**	**60%**
OPENSTACKS-OPT11 (20)	**15**	−3	−1	**±0**	4960916	100%	**60%**	**60%**
OPENSTACKS-OPT14 (20)	**3**	−2	**±0**	**±0**	424367	100%	**60%**	**60%**
ZENOTRAVEL (20)	**12**	**±0**	−2	**±0**	18998735	99%	**76%**	77%
FLOORTILE-OPT11 (20)	4	±0	**+1**	**+1**	34167799	100%	**79%**	83%
AIRPORT (50)	**18**	−2	**±0**	**±0**	1992660	100%	**61%**	90%
MPRIME (35)	**23**	−1	**±0**	**±0**	15102564	100%	**92%**	**92%**
NOMYSTERY-OPT11 (20)	**18**	−1	−1	−3	401945	100%	**97%**	**97%**
HIKING-OPT14 (20)	**13**	−3	−4	−3	10851210	100%	**99%**	**99%**
HIKING-MCO14 (20)	**3**	−1	−1	−1	1412978	100%	**99%**	**99%**
PIPESWORLD-NOTANKAGE (50)	**16**	−1	−1	−1	16538093	100%	**99%**	**99%**
DEPOT (22)	**6**	−1	−1	**±0**	14883344	100%	**99%**	100%
BLOCKS (35)	**26**	−1	**±0**	**±0**	37868664	**100%**	**100%**	**100%**
FREECELL (80)	**20**	−6	−6	−6	6147833	**100%**	**100%**	**100%**
MICONIC (150)	**72**	**±0**	−3	−2	89056243	**100%**	**100%**	**100%**
PIPESWORLD-TANKAGE (50)	**14**	−2	−2	−1	12223901	100%	**98%**	100%
SCANALYZER-08 (30)	**13**	−2	−5	−4	23908080	**100%**	**100%**	**100%**
SCANALYZER-OPT11 (20)	**10**	−2	−5	−4	23908080	**100%**	**100%**	**100%**
SOKOBAN-OPT08 (30)	**26**	−6	−2	−1	37392445	100%	**94%**	100%
SOKOBAN-OPT11 (20)	**20**	−3	**±0**	**±0**	37391596	100%	**94%**	100%
TRANSPORT-OPT14 (20)	**7**	−1	−1	−1	40851713	100%	**98%**	100%
TRUCKS (30)	**7**	−2	−2	−1	4711045	100%	**98%**	100%
REMAINING DOMAINS (1079)	**313**	**±0**	**±0**	**±0**	1457314640	99%	**89%**	**89%**
SUM (2107)	758	−43	−4	**+7**	2393533796	97%	**61%**	73%

Table 8.2: Results overview for the merge-and-shrink heuristic.

21 instances *more* than the baseline A* while EC solved 14 instances *less* than A*. In addition, we observe that SSS yields a higher coverage than SSS-EC: SSS solves additional 26 instances compared to the baseline configuration.
Table 8.2 shows that both EC and SSS-EC solve less instances than the baseline A* when the merge-and-shrink heuristic is used. However, the difference in coverage loss is still big between the two configurations: EC loses 43 instances, while SSS-EC loses only 4 instances.
We notice the low coverage increase for SSS (compared to the result for the landmark-cut heuristic) and significant decrease in coverage for EC and SSS-EC compared to the baseline configuration. This can be justified as follows: The computation of the merge-and-shrink heuristic involves a pre-processing step and a look-up step. After performing the pre-processing step, looking-up heuristic values for explored states is rather fast compared to the computation of the landmark-cut heuristic. This means that exploring states is less expensive with merge-and-shrink than with landmark-cut which makes the pruning effect less significant. In addition, the overhead required to compute strong stubborn sets or expansion core could worsen the overall performance.
Finally, Table 8.3 is an overview of the results for all configurations using A* with the blind heuristic. The results are similar to the results for the merge-and-shrink heuristic. We observe that all the three pruning methods lead to decrease in coverage. In particular, we see the significant degradation of the performance of EC with the blind heuristic compared to its performance with

	Coverage				Nodes generated			
Domain (problems)	A*	+EC	+SSS-EC	+SSS	A*	+EC	+SSS-EC	+SSS
PARCPRINTER-08 (30)	10	±0	**+20**	**+20**	70870124	100%	**<1%**	**<1%**
PARCPRINTER-OPT11 (20)	6	±0	**+14**	**+14**	70854677	100%	**<1%**	**<1%**
WOODWORKING-OPT08 (30)	7	±0	**+7**	+6	40894144	72%	**<1%**	1%
WOODWORKING-OPT11 (20)	2	±0	**+6**	+5	38105951	71%	**<1%**	1%
ROVERS (40)	5	±0	**+1**	**+1**	135111806	98%	**3%**	**3%**
SATELLITE (36)	5	1	**+1**	**+1**	5926926	88%	**4%**	**4%**
MYSTERY (30)	**15**	−3	−3	−2	16036138	100%	**31%**	47%
SOKOBAN-OPT08 (30)	21	−5	**±0**	**±0**	46997936	100%	**42%**	100%
SOKOBAN-OPT11 (20)	18	−5	**±0**	**±0**	46987680	100%	**42%**	100%
NOMYSTERY-OPT11 (20)	**8**	−1	**±0**	−1	5527015	100%	**49%**	69%
AIRPORT (50)	21	−3	**±0**	**±0**	5859635	100%	**49%**	78%
OPENSTACKS-OPT08 (30)	**20**	−3	**±0**	−1	5017325	100%	**59%**	60%
OPENSTACKS-OPT11 (20)	**15**	−3	**±0**	−1	4992597	100%	**60%**	**60%**
OPENSTACKS-OPT14 (20)	**3**	−2	**±0**	**±0**	424367	100%	**60%**	**60%**
TIDYBOT-OPT11 (20)	**13**	−10	−6	−8	67404	100%	**61%**	**61%**
PSR-SMALL (50)	**49**	−1	**±0**	**±0**	13379581	100%	**64%**	**64%**
ELEVATORS-OPT08 (30)	11	±0	**+1**	±0	349270958	100%	**65%**	87%
ELEVATORS-OPT11 (20)	9	±0	**+1**	±0	348702574	100%	**65%**	87%
ZENOTRAVEL (20)	**8**	−1	−1	−1	10750124	99%	**84%**	85%
TRUCKS (30)	**6**	−2	−1	−1	37961719	100%	**95%**	**95%**
MPRIME (35)	**19**	−5	−4	−2	18473788	100%	**98%**	**98%**
HIKING-OPT14 (20)	**11**	−3	−3	−3	40366687	100%	**98%**	99%
THOUGHTFUL-AGL14 (20)	**5**	−4	**±0**	**±0**	10083954	100%	**98%**	99%
TRANSPORT-OPT14 (20)	**6**	−2	−2	−1	18867064	100%	**99%**	100%
PIPESWORLD-NOTANKAGE (50)	**14**	**±0**	−2	**±0**	14036756	100%	**99%**	**99%**
DEPOT (22)	**4**	**±0**	−1	**±0**	23390231	100%	**99%**	100%
PIPESWORLD-TANKAGE (50)	**11**	−3	−5	−3	3537883	100%	**99%**	100%
FREECELL (80)	**15**	−2	−6	−7	8024017	100%	**99%**	100%
GED-OPT14 (20)	**15**	−2	**±0**	**±0**	**9521324**	**100%**	**100%**	**100%**
MICONIC (150)	**50**	**±0**	−5	**±0**	**387690342**	**100%**	**100%**	**100%**
SCANALYZER-08 (30)	**12**	−6	−6	−6	**1798292**	**100%**	**100%**	**100%**
SCANALYZER-OPT11 (20)	**9**	−6	−6	−6	**1005252**	**100%**	**100%**	**100%**
TETRIS-OPT14 (17)	**8**	−5	−4	−3	**366690**	**100%**	**100%**	**100%**
TPP (30)	**6**	−1	**±0**	**±0**	**119411**	**100%**	**100%**	**100%**
REMAINING DOMAINS (977)	**187**	**±0**	**±0**	**±0**	845750046	99%	**91%**	93%
SUM (2107)	624	−90	−6	−4	2636770418	99%	**71%**	80%

Table 8.3: Results overview for blind search (i.e., A* with the blind heuristic).

the merge-and-shrink heuristic: 90 vs 43 less solved instances. Moreover, we see that SSS-EC and SSS lose instances in several domains. This is due to the overhead in the computation of strong stubborn sets, specially in domains where no pruning occurs.

Generated nodes. The decreased number of generated search nodes is the main reason for the increase in coverage for partial order reduction techniques[1]. Table 8.1 shows that both SSS-EC and SSS have by far more pruning power than EC. For every domain, the reported numbers of generated nodes refer to the instances that are commonly solved by all configurations. The most interesting domains in which SSS-EC and SSS perform significantly well are PARCPRINTER and WOODWORKING where both configurations prune more than 99% of the explored space (i.e., generate less than 1% of the states generated by A*). Overall, the number of generated states is reduced to 98% with EC, 85% with SSS-EC, and 90% with SSS using the landmark-cut-heuristic. Using the merge-and-shrink heuristic, the amount of reduction is 97% with EC, 61%

[1]The reported number of generated nodes is until the last f-layer to eliminate the effect of tie-breaking.

with SSS-EC, and 73% with SSS. Finally, using the blind heuristic, the amount of reduction is 99% with EC, 71% with SSS-EC, and 80% with SSS.

Although SSS-EC outperforms SSS in the terms of node generation, SSS solves more instances than SSS-EC. This happens because SSS is computationally faster than SSS-EC. The experimental results reveal that using SSS is preferable in practice even though SSS-EC is slightly better in terms of pruning power.

Search time. Figure 8.1 shows that reducing the number of generated search nodes using SSS and SSS-EC using the landmark-cut heuristic leads to a faster search on average compared to the baseline and to EC. On the other hand, Figures 8.2 and 8.3 show that all three pruning configurations worsen the search time on average using the merge-and-shrink and the blind heuristics.

In summary, the empirical evaluation in this section emphasizes the following points:

1. The experiments reflect the dominance of a particular instantiation of strong stubborn sets (SSS-EC) over the expansion core method.

2. Partial order reduction techniques are more effective when combined with heuristics that are relatively expensive, as the relative overhead of computing partial order reduction techniques is low. On the other hand, combining them with cheap heuristics (i.e., fast to compute) does not pay off, due to the increase of the relative overhead needed during the computation of partial order reduction techniques, which worsen the overall performance of the search.

3. For domains where the pruning power is insignificant, using partial order reduction techniques can affect scalability of the underlying search algorithm, due to the computational overhead which can extensively degrade both memory consumption and runtime. For example, in SCANALYZER domains, the computation of the interference relation for SSS and SSS-EC runs out of memory, wheres the overhead in the actual computation of strong stubborn sets during search leads to timeouts in PARKING, SOKOBAN, and VISITALL domains.

8.2 Weak Stubborn Sets vs. Strong Stubborn Sets

In chapter 4, we have seen that weak stubborn sets have, at least theoretically, exponentially more pruning power than strong stubborn sets. In this section, we experimentally compare both variants to each other. We consider the SSS version used in the previous section and the landmark-cut heuristic. Weak stubborn sets are referred to as WSS. In Table 8.4, we provide an overview of the results on the domains where the pruning power of SSS and WSS differs.

Experiment setting and benchmarks. The same setting and benchmarks as in section 8.1.

	Coverage		Nodes generated		# problems
Domain (problems)	SSS	WSS	SSS	WSS	w. diff. gen.
OPENSTACKS-OPT08 (30)	**20**	**±0**	6129805	**99.968%**	10
OPENSTACKS-OPT11 (20)	**15**	**±0**	6116635	**99.968%**	10
OPENSTACKS-OPT14 (20)	**3**	**±0**	4138032	**99.552%**	2
PATHWAYS-NONEG (30)	**5**	**±0**	153443	**99.791%**	2
PSR-SMALL (50)	**49**	**±0**	18109203	**99.998%**	6
SATELLITE (36)	**12**	**±0**	69748950	**92.914%**	12
ROVERS (40)	**10**	**±0**	96609018	**97.213%**	4
STORAGE (30)	**15**	**±0**	2374214	**97.725%**	12

Table 8.4: Weak compared to strong stubborn sets (using landmark-cut heuristic).

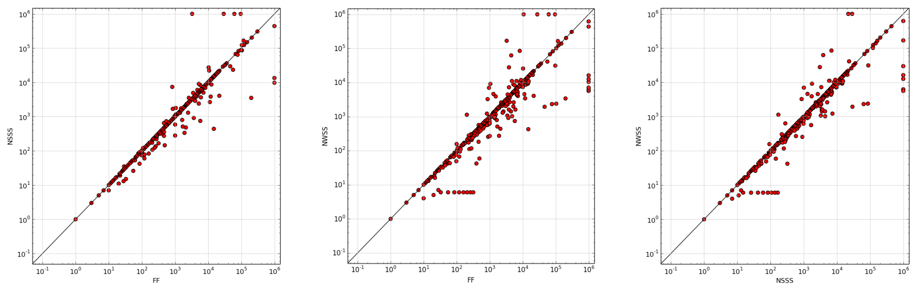
Although WSS can be exponentially more powerful than SSS, the experimental results show that the difference between both variants is almost negligible for the available IPC optimal planning domains: The same coverage for both algorithms, and very few domains where WSS reduce the number of generated search nodes. Despite the fact that the difference between SSS and WSS is experimentally small, we believe that the results are interesting for the following reasons:

1. This is the first experimental evaluation of weak stubborn sets. In the area of model checking, weak stubborn sets have been only theoretically analyzed [Val89], but there is no empirical evaluation for them due to their complicated conditions.

2. In the context of SAS^+ language, weak stubborn sets are straightforward computable by dropping one of the conditions needed to compute the interference relation for SSS. Furthermore, they exponentially dominate SSS in theory, and we think, there might be new planning domains in future, where WSS can practically be more beneficial than SSS.

8.3 Stubborn Sets for FOND planning

In this section, we empirically investigate the pruning power of nondeterministic stubborn sets when applied as the only pruning technique and also on top of other pruning techniques such as structural symmetries and active operators pruning. For this purpose, strong and weak stubborn sets variants as well as active operators pruning have been implemented on top of an adaptation of Fast Downward [Hel06] to FOND planning [WWK16], which already included symmetry based pruning. As a baseline search algorithm, we employ LAO^* using the FF heuristic [HN11] and we refer to this configuration as FF for simplicity. The implementation is based on the all-outcome determinization of FOND planning tasks. All configurations are built on top of this baseline. For all stubborn sets approaches, the disabling relation and achievers were entirely precomputed, while the interference relation was computed during the search and then cached for later use on demand.



(a) FF vs. NSSS (b) FF vs. NWSS (c) NSSS vs. NWSS

Figure 8.4: Generated nodes comparison for FOND planning problems.

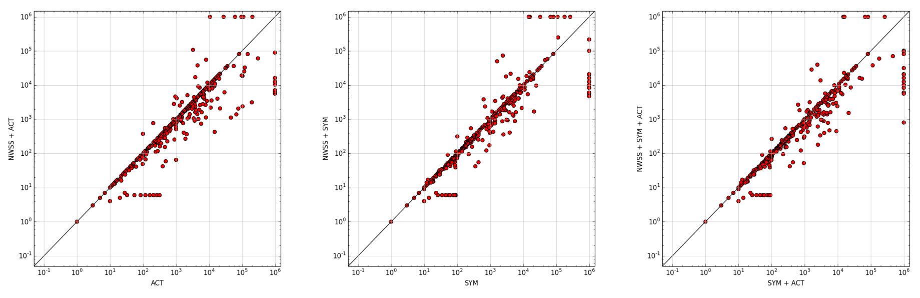
Experiment setting and benchmarks. All our experiments were conducted on a cluster equipped with Intel Xeon E5-2650 v2 CPUs running at 2.6 GHz. For each run, the time limit and memory bound were 30 minutes and 2 GB, respectively. The benchmark set consists of all IPC-08 FOND domains, scaled-up versions of these domains by Christian Muise and other FOND domains commonly used in the literature.

8.3.1 NSSS and NWSS

Domain (problems)	FF	NSSS	NWSS
BLOCKSWORLD (30)	**22**	−1	−1
CHAIN-OF-ROOMS-FIXED (10)	**10**	**±0**	**±0**
EARTH-OBSERVATION (40)	**33**	**±0**	**±0**
FAULTS (55)	**55**	**±0**	**±0**
FIRST-RESPONDERS (100)	**98**	−1	**±0**
FOREST (90)	8	+2	**+5**
PRP-BLOCKSWORLD-NEW (50)	**16**	−1	−1
PRP-ELEVATORS (15)	14	±0	**+1**
PRP-EX-BLOCKSWORLD (15)	**8**	**±0**	**±0**
PRP-FAULTS-NEW (190)	**190**	**±0**	**±0**
PRP-FIRST-RESPONDERS-NEW (95)	30	±0	**+1**
PRP-FOREST-NEW (90)	7	±0	**+1**
TIDYUP-MDP (10)	**10**	**±0**	**±0**
TIREWORLD (15)	**15**	**±0**	**±0**
TRIANGLE-TIREWORLD (40)	**7**	**±0**	**±0**
SUM (845)	523	−1	**+6**

Table 8.5: Coverage of baseline (FF), nondeterministic strong stubborn sets (NSSS) and nondeterministic weak stubborn sets (NWSS).

Coverage. Table 8.5 is an overview about coverage per domain for the baseline algorithm LAO* using the FF heuristic, NSSS, and NWSS both built on top of the baseline. We notice that NSSS performance is similar to the baseline algorithm (FF), loosing one problem each in three domains, and increasing coverage by two problems in one domain. In total, coverage of NSSS is reduced by one instance. Comparing NWSS to FF, it looses one task each in two domains and gains five problems in one domain and one problem each in three domains, overall increasing coverage by six problems.



(a) ACT vs. NWSS + ACT (b) SYM vs. NWSS + SYM (c) SYM + ACT vs. NWSS + SYM + ACT

Figure 8.5: Generated nodes comparison for NWSS with (a) active operators, (b) symmetries, and (c) both active operators and symmetries.

Generated nodes. Figure 8.4 shows the number of generated nodes per instance comparing the three configurations to each other.
Figure 8.4a compares NSSS to the baseline FF. Most of the tasks appear on or near the diagonal. This result is consistent with the small difference in coverage between FF and NSSS. Comparing NWSS to FF: Figure 8.4b shows that pruning by NWSS helps in most cases to reduce the number of generated nodes. In some cases, the decrease in node generations is almost reaching two orders of magnitude. Note that there is an exponential reduction in generated nodes on a part of the FIRST-RESPONDERS domain. Figure 8.4c shows that this gain carries over to the comparison with NSSS, reflecting the theoretical dominance of nondeterministic weak stubborn sets over the strong variant. It should be pointed out that due to the non-optimal nature of the LAO* algorithm, the theoretical dominance does not necessarily translate to a dominance in the number of generated nodes.

Search time. In domains where no pruning is possible, the stubborn sets computation can slow down the search on certain instances. We observe such an effect in the domains BLOCKSWORLD and PRP-BLOCKSWORLD-NEW where less instances are solved with stubborn sets than without. One possibility to avoid loss in coverage is to stop the computation of stubborn sets after a predefined threshold of unsuccessfully pruning attempts is exceeded as has been done in Metis [Alk+14].

To summarize, nondeterministic weak stubborn are beneficial to LAO* search whereas nondeterministic strong stubborn sets to a lesser extent.

8.3.2 Combining NWSS with other Pruning Techniques

Recently, it has been shown that structural symmetries and stubborn sets can successfully be combined for classical planning [Weh+15]. Furthermore, structural symmetries have been applied in the context of planning [WWK16]. Here, we experiment whether combining stubborn sets with structural symmetries and with active operators pruning can have synergy effects. First, we shortly summarize the idea of symmetries and active operators:

Algorithm	**Coverage** *all operators*	*active operators*
FF	523	**+1**
NWSS	**529**	−2
SYM	526	**+1**
NWSS + SYM	532	**+6**

Table 8.6: Coverage results for FF, NWSS, SYM and NWSS + SYM with and with and without active operator pruning.

- *Symmetry elimination*: Symmetry elimination considers equivalence classes of symmetrical states and allows for using representative states of each equivalence class. Many approaches have shown their potential in several contexts in classical planning. We consider for the variant of structural symmetries for FOND planning proposed by [WWK16].
- *Active operators pruning*: This pruning technique was introduced by Chen and Yao 2009 [CY09] and later further investigated by Wehrle et al. 2013 [Weh+13]. In a nutshell, given a state s, an operator o is considered *active* if there exists a weak plan from s starting with o. A sufficient criterion can be formulated based on domain transition graphs (DTGs). The successor generation in a state s is restricted to active operators.

Combining these pruning techniques in a straightforward way results in a combined pruning technique that is safe. For this evaluation, we consider all possible combinations of symmetry elimination and active operators pruning with and without nondeterministic weak stubborn sets.

Coverage. The coverage results are reported in Table 8.6. We observe that all non-combined pruning techniques improve coverage. In particular, nondeterministic weak stubborn sets is the most beneficial configuration out of those (529 solved instances), followed by symmetry elimination (526 solved instances) and active operator pruning (524 solved instances). Most importantly, the configuration that combines NWSS with symmetries and active operators pruning leads to a substantial coverage increase of 15 additionally solved instances (538) (compared to the baseline FF). In addition, we observe that applying NWSS on top of symmetry reduction (SYM) leads to a clear performance improvement with and without active operators pruning. By evaluating the gain from applying active operators pruning, we see that this pruning technique is only moderately beneficial both as a single pruning technique and when combined with symmetry reduction and nondeterministic stubborn sets.

Generated nodes. Figure 8.5 depicts the generated nodes comparison. In general, we notice that the configurations that uses NWSS are more powerful regarding pruning than others.

8.4 Sleep Sets vs. Commutativity Pruning vs. Stratified Planning

In this section, we experimentally compare sleep sets (SS), commutativity pruning (CP) and stratified planning (SP) to each other. All the three techniques have been implemented with IDA* search algorithm with full cycle elimination, but no full duplicate elimination. Table 8.7 provides an overview of the coverage for the three pruning techniques and the baseline plain IDA*. It shows only the domains for which coverage is not the same for all configurations. The rest of the domains are considered in "remaining domains". These experiments are performed using the operator ordering imposed by the authors of stratified planning [CXY09] under which the dominance relationships hold (Section 6.3).

Experiment setting and benchmarks. The same setting and benchmarks as in section 8.1.

Coverage. Obviously, Table 8.7 shows that the landmark-cut heuristic yields the most powerful configurations for IDA* for the same reason we discussed in the previous section. Most importantly, the theoretical dominance relationship of sleep sets over commutativity pruning and of commutativity pruning over stratified planning is confirmed by the coverage results for all heuristics.

	Landmark-cut				Merge-and-shrink				Blind			
Domain (problems)	IDA*	+SP	+CP	+SS	IDA*	+SP	+CP	+SS	IDA*	+SP	+CP	+SS
airport (50)	32	±0	+1	**+6**	**11**	**±0**	**±0**	**±0**	7	±0	+4	**+8**
depot (22)	2	±0	±0	**+2**	2	**±0**	**±0**	**±0**	1	±0	±0	+1
driverlog (20)	8	+1	**+2**	**+2**	**11**	**±0**	**±0**	**±0**	1	+1	**+2**	**+2**
elevators-opt08 (30)	0	+2	+3	**+5**	**0**	**±0**	**±0**	**±0**	**0**	**±0**	**±0**	**±0**
elevators-opt11 (20)	0	+1	+1	**+3**	**0**	**±0**	**±0**	**±0**	**0**	**±0**	**±0**	**±0**
floortile-opt11 (20)	1	±0	**+1**	**+1**	**4**	**±0**	**±0**	**±0**	**0**	**±0**	**±0**	**±0**
freecell (80)	2	±0	**+2**	**+2**	**2**	±0	**1**	**1**	**1**	**±0**	**±0**	**±0**
gripper (20)	2	±0	**+1**	**+1**	**20**	**±0**	**±0**	**±0**	1	**±0**	**±0**	**±0**
logistics00 (28)	**7**	+7	+7	**+13**	10	+4	+4	**+9**	1	+2	+2	**+5**
logistics98 (35)	3	±0	±0	**+3**	**2**	±0	±0	**+2**	**0**	**±0**	**±0**	**±0**
miconic (150)	**138**	**±0**	**±0**	**±0**	**65**	+1	+1	+1	**15**	**±0**	**±0**	**±0**
movie (30)	**30**	**±0**	**±0**	**±0**	**30**	**±0**	**±0**	**±0**	3	±0	**+17**	+8
mprime (35)	**20**	**±0**	**±0**	**±0**	**11**	−1	**±0**	**±0**	10	±0	**+3**	+1
mystery (30)	15	±0	**+1**	**+1**	**7**	**±0**	**±0**	**±0**	9	+2	+5	**+6**
nomystery-opt11 (20)	**12**	**±0**	**±0**	**±0**	15	**+3**	**+3**	**+5**	5	**+1**	**+1**	**+1**
openstacks-opt08 (30)	4	±0	+2	**+4**	7	±0	+2	**+3**	4	±0	**+4**	**+4**
openstacks-opt11 (20)	1	±0	+1	**+3**	2	±0	+2	**+3**	1	±0	**+3**	**+3**
parcprinter-08 (30)	13	**+2**	**+2**	**+2**	14	**+1**	**+1**	**+1**	3	**+3**	**+3**	**+3**
parcprinter-opt11 (20)	8	+1	+1	**+2**	9	**+1**	**+1**	**+1**	0	**+2**	**+2**	**+2**
parking-opt11 (20)	0	±0	±0	**+1**	**0**	**±0**	**±0**	**±0**	**0**	**±0**	**±0**	**±0**
pathways-noneg (30)	**4**	**±0**	**±0**	**±0**	3	±0	**+1**	**+1**	2	±0	±0	**+2**
pipesworld-notankage (50)	8	±0	+2	**+3**	**6**	**±0**	**±0**	**±0**	4	±0	+2	**+3**
pipesworld-tankage (50)	4	**±0**	+1	**+2**	**6**	**±0**	**±0**	**±0**	2	±0	**+2**	**+2**
psr-small (50)	32	±0	**+10**	**+10**	41	−1	**+2**	**+2**	31	±0	**+9**	+8
rovers (40)	4	±0	**+2**	**+2**	5	±0	±0	**+1**	**4**	**±0**	**±0**	**±0**
satellite (36)	5	±0	±0	**+1**	**6**	**±0**	**±0**	**±0**	**1**	**±0**	**±0**	**±0**
scanalyzer-08 (30)	12	−3	**+1**	**+1**	9	−2	**+1**	**+1**	**3**	**±0**	**±0**	**±0**
scanalyzer-opt11 (20)	9	−3	**+1**	**+1**	6	−2	**+1**	**+1**	**1**	**±0**	**±0**	**±0**
storage (30)	11	±0	+1	**+3**	11	±0	+2	**+3**	9	±0	**+1**	**+1**
tetris-opt14 (17)	2	±0	±0	**+1**	**0**	**±0**	**±0**	**±0**	1	±0	**+2**	**+2**
thoughtful-agl14 (20)	0	±0	+1	**+4**	**0**	**±0**	**±0**	**±0**	**0**	**±0**	**±0**	**±0**
tidybot-opt11 (20)	4	±0	**+2**	**+2**	**0**	**±0**	**±0**	**±0**	1	±0	**+2**	**+2**
tpp (30)	5	**+1**	**+1**	**+1**	5	±0	±0	**+1**	4	±0	±0	**+1**
transport-opt14 (20)	0	±0	±0	**+1**	0	±0	**+1**	**+1**	**0**	**±0**	**±0**	**±0**
trucks(30)	3	±0	**+1**	**+1**	**3**	**±0**	**±0**	**±0**	**0**	±0	**+1**	±0
woodworking-opt08 (30)	10	±0	+1	**+2**	**8**	**±0**	**±0**	**±0**	2	**+2**	**+2**	**+2**
woodworking-opt11 (20)	5	±0	+1	**+2**	**3**	**±0**	**±0**	**±0**	**0**	**±0**	**±0**	**±0**
zenotravel (20)	9	**+1**	**+1**	**+1**	9	**+1**	**+1**	**+1**	**4**	**±0**	**±0**	**±0**
Remaining domains (854)	**86**	**±0**	**±0**	**±0**	**116**	**±0**	**±0**	**±0**	**54**	**±0**	**±0**	**±0**
Sum (2107)	511	+11	+52	**+89**	459	+5	+24	**+36**	185	+13	**+67**	**+67**

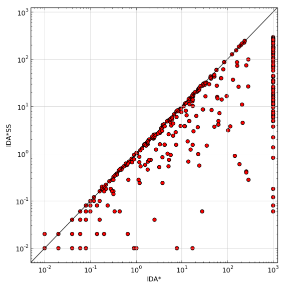
Table 8.7: Coverage overview for the transition reduction techniques.

Generated nodes. Tables 8.8, 8.9, and 8.10 provide the number of node generations (until the last f-layer to eliminate the effect of tie-breaking) for the three pruning techniques using the landmark-cut, the merge-and-shrink, and the blind heuristics, respectively. The listed domains are those whose overall number of generated nodes is not the same for all configurations. As before, the given numbers refer to the instances that are solved by all configurations. The dominance relation of sleep sets over commutativity pruning and of the latter over stratified planning are depicted in the results. However, we notice that sleep sets and commutativity pruning do not differ much in the overall performance, i.e., the overall number of generated states.

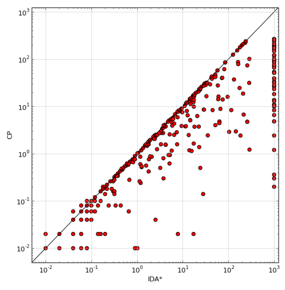
Domain (problems)	IDA*	+SP	+CP	+SS
LOGISTICS00 (7)	2894764	**<1%**	**<1%**	**<1%**
DRIVERLOG (8)	3383376	9%	2%	**<1%**
TPP (5)	48142	**1%**	**1%**	**1%**
TETRIS-OPT14 (2)	337371	100%	3%	**2%**
FLOORTILE-OPT11 (1)	1745622	99%	6%	**2%**
LOGISTICS98 (3)	29607	7%	7%	**3%**
ZENOTRAVEL (9)	1196887	7%	7%	**4%**
WOODWORKING-OPT08 (10)	220234	22%	5%	**4%**
WOODWORKING-OPT11 (5)	219406	22%	5%	**4%**
PSR-SMALL (32)	18271312	99%	**8%**	**8%**
MICONIC (138)	1833990	17%	**11%**	**11%**
DEPOT (2)	4378	19%	15%	**11%**
PIPESWORLD-NOTANKAGE (8)	507961	100%	17%	**13%**
MYSTERY (17)	101347	62%	15%	**14%**
PIPESWORLD-TANKAGE (4)	118359	100%	19%	**15%**
GRIPPER (2)	215486	100%	**19%**	**19%**
SATELLITE (5)	8444	**20%**	**20%**	**20%**
TRUCKS (3)	3369059	100%	**22%**	**22%**
NOMYSTERY-OPT11 (12)	173254	**25%**	**25%**	**25%**
SCANALYZER-08 (9)	231950	100%	36%	**33%**
SCANALYZER-OPT11 (6)	231940	100%	36%	**33%**
TIDYBOT-OPT11 (4)	37613	100%	**35%**	**35%**
MPRIME (20)	237976	73%	37%	**35%**
PARCPRINTER-OPT11 (8)	193	**40%**	**40%**	**40%**
ROVERS (4)	615	51%	49%	**41%**
HIKING-OPT14 (1)	428322	100%	**45%**	**45%**
FREECELL (2)	97468	100%	49%	**45%**
PARCPRINTER-08 (13)	243	**53%**	**53%**	**53%**
OPENSTACKS-OPT08 (4)	5384	100%	**54%**	**54%**
STORAGE (11)	166774	100%	56%	**55%**
PATHWAYS-NONEG (4)	14825	70%	67%	**64%**
TRANSPORT-OPT08 (4)	995	90%	90%	**85%**
OPENSTACKS-OPT11(1)	27	100%	**96%**	**96%**
GED-OPT14 (10)	2069012	100%	**99%**	**99%**
REMAINING DOMAINS (133)	**33085018**	**100%**	**100%**	**100%**
SUM (507)	68504304	86%	53%	**52%**

Table 8.8: Overview of node generations using landmark-cut heuristic.

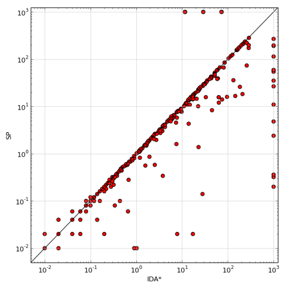
Search time. Figures 8.6, 8.7, and 8.8 compare search time per instance for all configurations using the landmark-cut heuristic, the merge-and-shrink heuristic, and the blind heuristic, respectively. Using the landmark-cut and the blind heuristics, the number of generated nodes of each of the four algorithms reflects on search time. On the other hand, the effect of pruning on search time is less obvious using the merge-and-shrink heuristic and the blind heuristic.



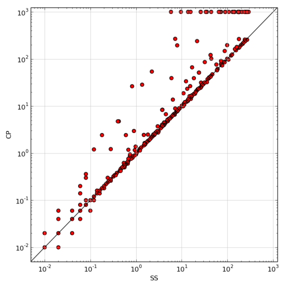
(a) IDA* vs. SS



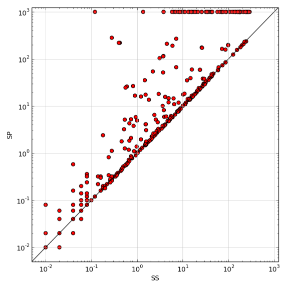
(b) IDA* vs. CP



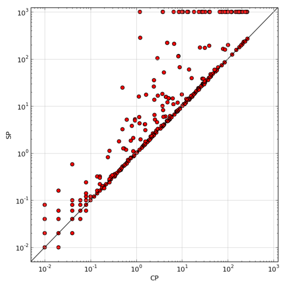
(c) IDA* vs. SP



(d) SS vs. CP

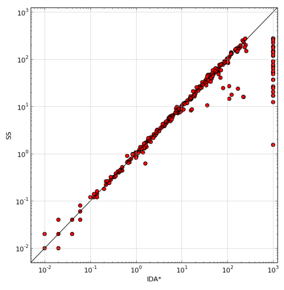


(e) SS vs. SP

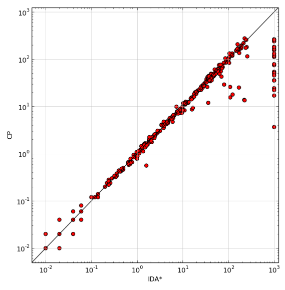


(f) CP vs. SP

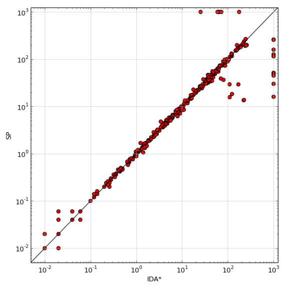
Figure 8.6: Search time for transition reduction techniques using the landmark-cut heuristic.



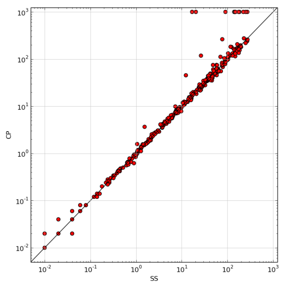
(a) IDA* vs. SS



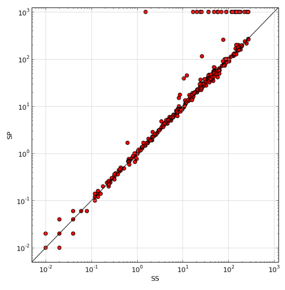
(b) IDA* vs. CP



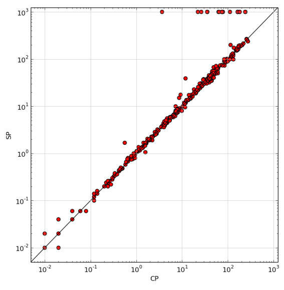
(c) IDA* vs. SP



(d) SS vs. CP



(e) SS vs. SP

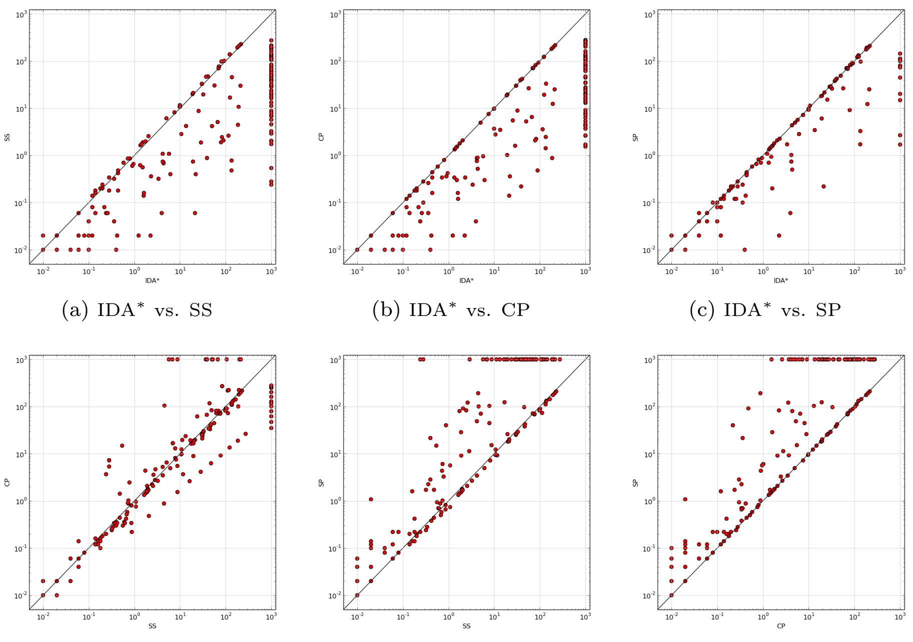


(f) CP vs. SP

Figure 8.7: Search time for transition reduction techniques using the merge-and-shrink heuristic.

Domain (problems)	IDA*	+SP	+CP	+SS
DEPOT (2)	16705038	2%	1%	**<1%**
DRIVERLOG (11)	32500074	3%	1%	**<1%**
ZENOTRAVEL (9)	7114300	3%	3%	**1%**
NOMYSTERY-OPT11 (15)	15793616	**2%**	**2%**	**2%**
TRANSPORT-OPT08 (6)	37599870	6%	6%	**5%**
TRANSPORT-OPT11 (1)	37599870	6%	6%	**5%**
PSR-SMALL (40)	260922	99%	**7%**	**7%**
LOGISTICS98 (2)	22789	9%	9%	**8%**
MYSTERY (10)	89067	42%	10%	**9%**
MPRIME (10)	9799098	36%	12%	**10%**
PIPESWORLD-TANKAGE (6)	20437	100%	16%	**13%**
ROVERS (5)	3679	37%	30%	**15%**
PIPESWORLD-NOTANKAGE (6)	53394	100%	22%	**18%**
FREECELL (2)	22464	100%	59%	**57%**
STORAGE (11)	4496056	100%	**68%**	**68%**
SCANALYZER-08 (7)	2054	100%	82%	**75%**
SCANALYZER-OPT11 (4)	2054	100%	82%	**75%**
GRID (2)	2204405	100%	**89%**	**89%**
MICONIC (65)	15367	**93%**	**93%**	**93%**
GED-OPT14 (13)	3233519	100%	**99%**	**99%**
REMAINING DOMAINS (239)	**127023216**	**100%**	**100%**	**100%**
SUM (456)	294561289	50%	**48%**	**48%**

Table 8.9: Overview of node generations using merge-and-shrink heuristic.



(a) IDA* vs. SS (b) IDA* vs. CP (c) IDA* vs. SP

(d) SS vs. CP (e) SS vs. SP (f) CP vs. SP

Figure 8.8: Search time for transition reduction techniques using the blind heuristic.

8.5 Sleep Sets vs. Move Pruning

Sleep sets and move pruning are evaluated using IDA*search algorithm that performs full cycle detection (but no full duplicate elimination) combined with

Domain (problems)	IDA*	+SP	+CP	+SS
DEPOT(1)	4801951	1%	**<1%**	**<1%**
TRANSPORT-OPT08(3)	21543128	2%	1%	**<1%**
MOVIE(3)	78283206	100%	**<1%**	**<1%**
LOGISTICS00(1)	517972	1%	1%	**<1%**
ROVERS(4)	15880760	4%	3%	**1%**
WOODWORKING-OPT08(2)	18851767	11%	6%	**1%**
TPP(4)	837171	**1%**	**1%**	**1%**
AIRPORT(7)	218305	90%	2%	**2%**
DRIVERLOG(1)	22854	6%	**2%**	**2%**
PSR-SMALL(31)	22592573	99%	**4%**	**4%**
PATHWAYS-NONEG(2)	2789361	27%	7%	**5%**
SATELLITE(1)	610089	**6%**	**6%**	**6%**
PIPESWORLD-TANKAGE(2)	6394300	100%	10%	**6%**
ZENOTRAVEL(4)	10671973	10%	10%	**7%**
PIPESWORLD-NOTANKAGE(4)	16039104	100%	13%	**8%**
MYSTERY(11)	410785	54%	12%	**11%**
TETRIS-OPT14(1)	10808	100%	**14%**	**14%**
FREECELL(1)	2843765	100%	24%	**19%**
STORAGE(9)	14303454	100%	21%	**20%**
NOMYSTERY-OPT11(5)	46361789	**22%**	**22%**	**22%**
MPRIME(10)	23026942	77%	26%	**24%**
GRIPPER(1)	82188	100%	35%	**33%**
TIDYBOT-OPT11(1)	294	100%	**38%**	**38%**
OPENSTACKS-OPT08(4)	6884	100%	**55%**	**55%**
OPENSTACKS-OPT11(1)	71	100%	**79%**	**79%**
MICONIC(15)	2028636	**96%**	**96%**	**96%**
GED-OPT14(9)	34795897	100%	**98%**	**98%**
GRID(1)	179738	100%	**99%**	**99%**
REMAINING DOMAINS (228)	**60332338**	**100%**	**100%**	**100%**
SUM (187)	478052982	76%	**45%**	**45%**

Table 8.10: Overview of node generations using blind heuristic.

the landmark-cut heuristic. In the figures, IDA*SS refers to IDA* combined with sleep sets, and IDA*MP refers to IDA* with move pruning.
For move pruning we set $L = 2$, where L is the length of operator sequences that are considered for redundancy check. Times that are less than or equal to 0.1 seconds are shown in the plots as 0.1[2].

Experiment setting and benchmarks. The evaluation is performed on Intel Xeon E5-2660 CPUs that run at 2.2 GHz, with a time limit of 30 minutes and a memory limit of 2 GB per run. The benchmarks are all optimal STRIPS planning instances up to IPC-11, with an overall number of 44 domains and 1396 problems.

Node generations and search time. Figure 8.9 is a comparison between IDA* (y-axis) and IDA*SS (x-axis) in terms of the number of nodes generated (left plot) and search time (without preprocessing time) in seconds (right plot) using logarithmic scales. The points on the diagonal lines represent problem instances. The lines $y = 2x$, $y = 10x$, and $y = 50x$ are used to easily compare the performance of the algorithms against each other. The points on the diagonal line represent instances for which the performance of both configurations is equal, while points above the diagonal are instances for which IDA*SS outperformed IDA*. The instances that were solved by IDA*SS but not by IDA* (due to exceeding the time limit) are shown at the very top of the plot. There are

[2]These results have been presented at AAAI 2015 [HAW15].

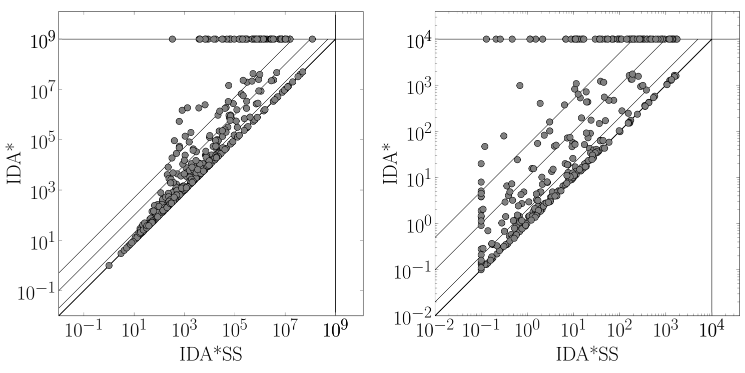


Figure 8.9: IDA* (y-axis) vs. IDA*SS (x-axis). Left: Number of nodes generated. Right: Search time [HAW15].

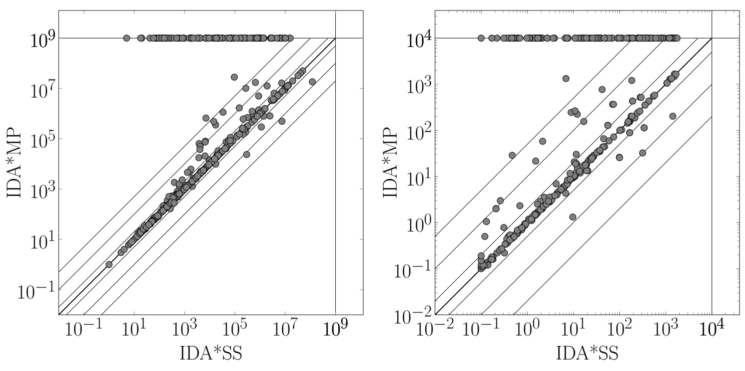


Figure 8.10: IDA*MP (y-axis) vs. IDA*SS (x-axis). Left: Number of nodes generated. Right: Search time [HAW15].

many instances for which both configurations generate exactly the same number of search nodes (shown on the diagonal line). In these instances, sleep sets pruning does not fire either because there is no commutativity between operators or the commutative operators are not used during search. For other instances, IDA*SS generates substantially fewer nodes than what IDA* generates (up to 3 orders of magnitude). The right plot (search time) shows that the reduction of node generations leads to a corresponding reduction in search time.
Similarly, Figure 8.10 compares IDA*SS with IDA*MP. The points at the very top of the plot represent the instances that are solved by IDA*SS but not by IDA*MP due to exceeding the memory bound during the preprocessing step, i.e., while computing the redundancy table which consumes more memory than the commutativity table computed by sleep sets.
We observe that, for most of the problem instances, IDA*SS and IDA*MP generate the same number of nodes. This happens because either no commutativity and no redundancy have been detected, or both methods detected the same commutativity (note that commutativity is a special case of redundancy).
The points below the diagonal are instance for which IDA*MP outperformed

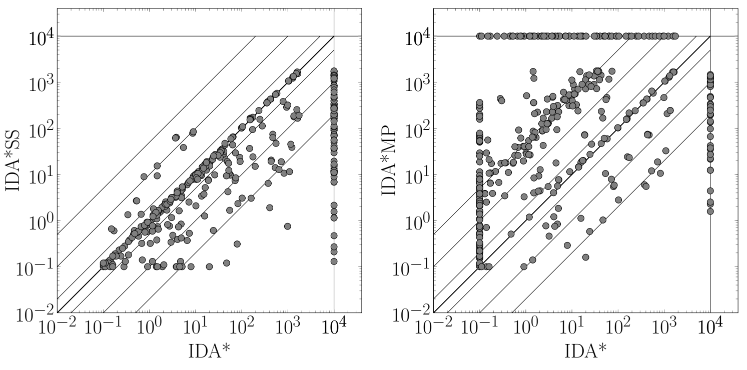


Figure 8.11: Total time for IDA* (x-axis) vs. IDA*SS (left plot, y-axis) and IDA*MP (right plot, y-axis) [HAW15].

IDA*SS, while the ones above the diagonal are instances for which IDA*SS performs better than IDA*MP. This shows that move pruning can sometimes outperform sleep sets when it detects redundancies that sleep sets cannot detect. On the other hand, sleep sets can sometimes be more powerful than move pruning because it has the property of passing commutative operator along operator sequences of length greater than 2. The plot of the search time shows how the pruning power is proportional to the search time of both configurations.

Coverage and total time. Figure 8.11 shows the total time required to solve each problem instance by IDA*, IDA*SS, and IDA*MP where the total time is the sum of the preprocessing time and the search time.
Preprocessing time for sleep sets is the time consumed for computing the commutativity relation, and for move pruning is the time needed to computing the redundancy relation for sequences of length 2 ($L = 2$). The total time of plain IDA* is its search time (no preprocessing is performed).
From the left plot of the figure (y-axis), we notice that the total time of IDA*SS is less than the total time (i.e., the search time) consumed by IDA*. This is because the preprocessing time of IDA*SS is negligible, and hence does not affect the overall performance of the algorithm. Moreover, the pruning power of sleep sets improves the search time of IDA*SS. As a result, IDA*SS solves more instance than IDA* (570 vs 477). All instances that are solved by IDA*SS, but not by IDA* are due to timeouts (exceeding 30 minutes).
As opposed to IDA*SS, the total time of IDA*MP is worse than IDA*'s mostly by more than an order of magnitude. Consequently, the IDA*MP's coverage is smaller than IDA*'s (443 vs 477).

We conclude that sleep sets outperform move pruning in terms of total time and coverage (570 vs 443). Furthermore, because of the expensive preprocessing of move pruning, it can be suitable in practice when the preprocessing step is used across many problem instances (the computation of the redundancy relation is independent of the initial state and goal description).

In summary, we have seen that both sleep sets and move pruning can remarkably enhance the performance of IDA*, and that none of the two pruning techniques dominates the other.

8.6 Generalized Sleep Sets

In this section, we compare sleep sets (based on commutativity) with the long distance leapfrogging generalization we presented in Section 6.4 for $L = 2$ (which is full flexible redundancy). We use IDA*GSS to denote this variant.

Experiment setting and benchmarks. The same setting and benchmarks as in section 8.5.

Node generations and search time. Figure 8.12 shows that the number of node generations is exactly the same for both IDA*GSS (y-axis) and IDA*SS (x-axis) for most instances. This means that, for these instances, the generalized variant does not capture more than the commutativity that sleep sets computes. For other instances, IDA*GSS can outperform IDA*SS by at most an oder of magnitude. The search time plot shows that the overhead for computing the generalized method is higher than the overhead for computing basic sleep sets.

Coverage and total time. Similar to IDA*MP, the preprocessing time for IDA*GSS is costly, which leads to less coverage than plain IDA* (421 vs 477). Like move pruning, we think that the generalized method is more appropriate when the preprocessing step can be utilized across many instances.

8.7 Sleep Sets with Duplicate Elimination and Stubborn Sets

In this section, we evaluate variant C of sleep sets (section 6.1) combined with strong stubborn sets within A* search. These results have been presented in [AW16].

Node generation. Figure 8.13 shows an overview of the generated search nodes per domain by considering only instances solved by both configurations.

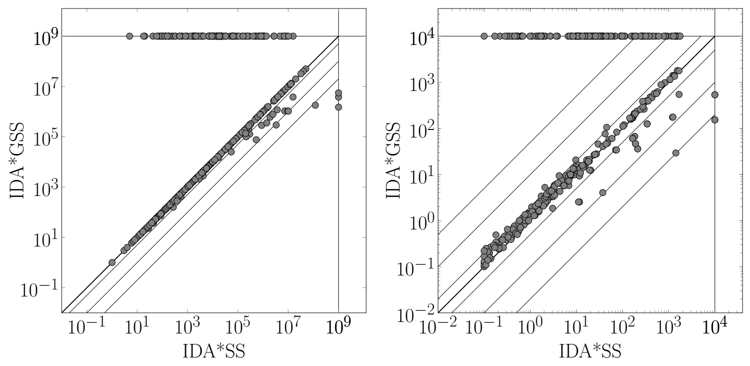


Figure 8.12: IDA*GSS (y-axis) vs. IDA*SS (x-axis). Left: Number of nodes generated. Right: Search time [HAW15].

A^*_{sss} refers to A^* combined with strong stubborn sets (SSS in the experiments of section 8.1). A^*_{sssss} is A^*_{sss} combined with variant C of sleep sets. Although sleep sets can be applied with any heuristic, we use the landmark-cut heuristic because, as we have already mentioned, pruning techniques can be more effective when combined with expensive heuristics. The numbers of commonly solved problems are given in parenthesis after the domain names, best results are shown in bold. The number of node generations is shown until the last f-layer to avoid tie-breaking issues.

The results reveal a consistent improvement regarding the number of generated nodes per domain. Although the savings in node generations are slight, the results show that the node generations *can* be further reduced compared to pure strong stubborn set pruning. Figure 8.14 shows the number of generated nodes and the search time per problem for the domains from the IPC-14. These are the domains for which the two configurations differ in the number of generated nodes. From the results we observe that the number of node generations for A^*_{sssss} is at most as high as for A^*_{sss} except for the Hiking domain. The reason behind this, is that inconsistent heuristics (like landmark-cut) can cause A^*_{sssss} to generate more nodes than A^*_{sss}. This presumably happens in the three Hiking problems where slightly more nodes are generated when sleep sets are applied in addition to stubborn set pruning.

The node savings are slight in general. However, some domains show that more additional pruning can be gained. For example, the number of generated nodes are roughly cut in half in a large instance of the Transport domain (problem #14).

Coverage. We notice that the savings in node generations do not increase coverage for the considered direct combination (or loose integration) of sleep sets and strong stubborn sets: the same coverage of A^*_{sss} and A^*_{sssss} with landmark-cut in all of the 33 domains.

Generated nodes	$\mathbf{A}^*_{sssss}$	$\mathbf{A}^*_{sss}$
barman-opt11-strips (4)	**22731591**	22882501
elevators-opt08-strips (22)	**38380306**	48873617
elevators-opt11-strips (18)	**36666539**	46248291
floortile-opt11-strips (7)	**29401436**	34912718
floortile-opt14-strips (6)	**52200140**	61804871
ged-opt14-strips (15)	**9064612**	**9064612**
hiking-opt14-strips (9)	**30638123**	30742124
nomystery-opt11-strips (14)	**387045**	410776
openstacks-opt08-strips (20)	**5777074**	6129805
openstacks-opt11-strips (15)	**5763904**	6116635
openstacks-opt14-strips (3)	**3866657**	4138032
parcprinter-08-strips (30)	**4877**	**4877**
parcprinter-opt11-strips (20)	**1884**	**1884**
parking-opt11-strips (2)	**555418**	560427
parking-opt14-strips (3)	**2253957**	2274968
pegsol-08-strips (28)	**54045223**	**54045223**
pegsol-opt11-strips (18)	**54408002**	**54408002**
scanalyzer-08-strips (12)	**13504754**	13942542
scanalyzer-opt11-strips (9)	**13504746**	13942534
sokoban-opt08-strips (29)	**38525983**	**38525983**
sokoban-opt11-strips (20)	**8310909**	**8310909**
tetris-opt14-strips (5)	**1150721**	1280023
tidybot-opt11-strips (14)	**299325**	308515
tidybot-opt14-strips (8)	**269184**	269891
transport-opt08-strips (11)	**302942**	426271
transport-opt11-strips (6)	**300508**	423268
transport-opt14-strips (6)	**2936311**	3396159
visitall-opt11-strips (11)	**23775034**	**23775034**
visitall-opt14-strips (5)	**2530507**	**2530507**
woodworking-opt08-strips (27)	**1722277**	2583855
woodworking-opt11-strips (19)	**976547**	1431694
Sum (427)	**454256536**	493766548

Figure 8.13: Sum of generated search nodes per domain on commonly solved problems (excluding the last f-layer) [AW16].

	Generated nodes		Search time	
Problem	$\mathbf{A}^*_{sssss}$	$\mathbf{A}^*_{sss}$	$\mathbf{A}^*_{sssss}$	$\mathbf{A}^*_{sss}$
floortile-opt14-strips				
p01-4-3-2	**1242815**	1510800	**32.11**	35.13
p01-4-4-2	**29688755**	35050715	**1090.50**	1186.70
p01-5-3-2	**4724922**	5612842	**153.86**	170.11
p02-5-3-2	**7206855**	8537779	**226.70**	245.06
p03-4-3-2	**1784070**	2141977	**44.69**	48.49
p03-5-3-2	**7552723**	8950758	**238.01**	257.55
hiking-opt14-strips				
ptesting-1-2-3	**1901**	**1901**	**0.01**	**0.01**
ptesting-1-2-4	**15294**	**15294**	**0.18**	**0.18**
ptesting-1-2-5	69869	**69749**	1.26	**1.23**
ptesting-1-2-7	634939	**634379**	21.81	**21.50**
ptesting-1-2-8	1496276	**1495364**	67.06	**66.22**
ptesting-2-2-3	**97479**	98151	2.99	**2.72**
ptesting-2-2-4	**4702120**	4710935	273.92	**265.36**
ptesting-2-3-4	**22911601**	22969357	1590.73	**1490.90**
ptesting-2-4-3	**708644**	746994	33.43	**28.86**
openstacks-opt14-strips				
p20_1	**1710721**	1764993	816.36	**803.53**
p20_2	**1939861**	2156773	608.38	**596.98**
p20_3	**216075**	216266	20.34	**19.83**
parking-opt14-strips				
p_12_7-01	**648718**	653952	439.45	**430.86**
p_12_7-02	**1305778**	1318576	868.17	**853.79**
p_12_7-03	**299461**	302440	216.37	**212.79**
tetris-opt14-strips				
p01-8	**599145**	645908	1530.18	**1135.26**
p02-4	**71**	140	0.09	**0.08**
p02-6	**496359**	568559	1208.25	**893.27**
p03-4	**6442**	7635	2.63	**2.18**
p05-6	**48704**	57781	36.93	**33.32**
tidybot-opt14-strips				
p02	**27104**	27128	540.48	**534.46**
p03	**31957**	32420	**532.31**	534.35
p04	**3202**	3203	53.44	**52.65**
p08	**9748**	9822	**152.59**	153.74
p11	**10775**	10817	104.06	**103.98**
p12	**153555**	153583	1545.73	**1526.25**
p13	**29865**	29938	**283.56**	283.98
p14	**2978**	2980	**50.51**	50.58
transport-opt14-strips				
p01	**1916**	3073	**0.14**	0.17
p02	**210118**	227350	**23.25**	24.41
p03	**215554**	268080	**67.49**	89.86
p07	**2310840**	2513397	236.03	**233.54**
p13	**10979**	19434	**4.51**	5.53
p14	**186904**	364825	**164.77**	230.93

Figure 8.14: Node generations (excluding the last f-layer) and search time on a per-problem basis for IPC-14 domains [AW16].

Chapter 9

Conclusion

In this thesis, we have seen that partial order reduction techniques can be applied to planning as heuristic search in order to reduce the size of the generated state space. In particular, the original techniques proposed for the area of computer aided verification are more powerful than other techniques which have been proposed later for automated planning. In the following, we summarize the contribution of this thesis and shed light on possible future work.

9.1 Summary

The most important topics that this thesis presented can be summarized as follows:

- Adapting the stubborn sets technique from its original form proposed by Valmari [Val89] to optimal classical planning and showing its dominance over the expansion core [CY09] in terms of pruning power.
- Coming up with a syntactic definition of weak stubborn sets and showing that it has exponentially more pruning power than strong stubborns sets (at least theoretically).
- Proposing a pruning technique for FOND planning based on the theory of stubborn sets.
- Adapting the sleep sets technique from its original form proposed by Godefroid [God96] to optimal planning and showing that it dominates both commutativity pruning [HG00] and stratified planning [CXY09] in terms of pruning power.
- Performing a detailed analysis for the variants of sleep sets proposed in the literature. In particular, we have seen that variant A of sleep sets is safe with BFS, and that variant C can safely be combined with A* and stubborn sets to be used in the context of optimal classical planning. In addition, variant D can safely be combined with IDA* search that performs only cycle detection.
- Proposing a family of transition reduction techniques (generalized sleep sets) that incorporate the characteristics of sleep sets and move pruning.

9.2 Future Work

We briefly mention the potential future work related to pruning techniques for planning. One possible topic is to investigate a completeness and optimality preserving tight combination of sleep sets and stubborn sets which has more pruning power than the loose combination we considered in our research (originally proposed by Godefroid).

Another worth-investigating idea is to apply partial order reduction methods (or pruning methods in general) with backward search algorithms. To the best of our knowledge, the research on pruning techniques for planning, that has been done so far, considered only forward search algorithms.

Finally, it is interesting to investigate combinations of pruning techniques to planning models beyond classical planning. For example, Fully Observable Nondeterministic Planning (FOND), Partial Observable Nondeterministic Planning (POND), and Multi-agent planning (MAP).

List of Figures

List of Tables

Bibliography

[Alk+12] Yusra Alkhazraji, Martin Wehrle, Robert Mattmüller, and Malte Helmert. "A Stubborn Set Algorithm for Optimal Planning". In: *Proceedings of Twentieth European Conference on Artificial Intelligence (ECAI 2012).* 2012, pp. 891–892.

[Alk+14] Yusra Alkhazraji, Michael Katz, Robert Mattmüller, Florian Pommerening, Alexander Shleyfman, and Martin Wehrle. "Metis: Arming Fast Downward with Pruning and Incremental Computation (planner abstract)". In: *In the Eighth International Planning Competition (IPC 2014) (deterministic track).* 2014, pp. 88–92.

[AW16] Yusra Alkhazraji and Martin Wehrle. "Sleep Sets Meet Duplicate Elimination". In: *Proceedings of the Ninth Annual Symposium on Combinatorial Search (SOCS 2016).* 2016, pp. 2–9.

[BG01] Blai Bonet and Hector Geffner. "Planning as heuristic search". In: *Journal of Artificial Intelligence (JAIR)* 129.1-2 (2001), pp. 5–33.

[BH11] Neil Burch and Robert C. Holte. "Automatic Move Pruning in General Single-Player Games". In: *Proceedings of the Fourth International Symposium on Combinatorial Search (SoCS 2011).* 2011, pp. 31–38.

[BH12] Neil Burch and Robert C. Holte. "Automatic Move Pruning Revisited". In: *Proceedings of the Fifth International Symposium on Combinatorial Search (SoCS 2012).* 2012.

[Bie+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. "Bounded model checking". In: *Journal of Advances in Computers* 58 (2003), pp. 117–148.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, 2008. ISBN: 978-0-262-02649-9.

[BN95] Christer Bäckström and Bernhard Nebel. "Complexity Results for SAS+ Planning". In: *Journal of Computational Intelligence* 11 (1995), pp. 625–656.

[Bos+09] Dragan Bosnacki, Edith Elkind, Blaise Genest, and Doron Peled. "On commutativity based Edge Lean search". In: *Annals of Mathematics and Artificial Intelligence* 56.2 (2009), pp. 187–210.

[BS15] Dragan Bosnacki and Mark Scheffer. "Partial Order Reduction and Symmetry with Multiple Representatives". In: *Proceedings on the Seventh International NASA Formal Methods Symposium (NFM 2015).* 2015, pp. 97–111.

[Byl91] Tom Bylander. “Complexity Results for Planning”. In: *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (AI 1991).* 1991, pp. 274–279.

[CC10] Amanda Jane Coles and Andrew Coles. “Completeness-Preserving Pruning for Optimal Planning”. In: *Proceedings of the Nineteenth European Conference on Artificial Intelligence (ECAI 2010).* 2010, pp. 965–966.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. “Model Checking and Abstraction”. In: *Journal of Transactions on Programming Languages and Systems (ACM)* 16.5 (1994), pp. 1512–1542.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking.* MIT Press, 2001. ISBN: 978-0-262-03270-4.

[Cim+03] Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. “Weak, strong, and strong cyclic planning via symbolic model checking”. In: *Journal of Artificial Intelligence (JAIR)* 147.1-2 (2003), pp. 35–84.

[CKV10] Edmund M. Clarke, Robert P. Kurshan, and Helmut Veith. “The Localization Reduction and Counterexample-Guided Abstraction Refinement”. In: *Time for Verification, Essays in Memory of Amir Pnueli.* 2010, pp. 61–71.

[Cla+01] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. “Bounded Model Checking Using Satisfiability Solving”. In: *Journal of Formal Methods in System Design* 19.1 (2001), pp. 7–34.

[CXY09] Yixin Chen, You Xu, and Guohui Yao. “Stratified Planning”. In: *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI 2009).* 2009, pp. 1665–1670.

[CY09] Yixin Chen and Guohui Yao. “Completeness and Optimality Preserving Reduction for Planning”. In: *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI 2009).* 2009, pp. 1659–1664.

[DFP09] Klaus Dräger, Bernd Finkbeiner, and Andreas Podelski. “Directed model checking with distance-preserving abstractions”. In: *STTT* 11.1 (2009), pp. 27–37.

[DHK15] Carmel Domshlak, Jörg Hoffmann, and Michael Katz. “Red-black planning: A new systematic approach to partial delete relaxation”. In: *Journal of Artificial Intelligence (JAIR)* 221 (2015), pp. 73–114.

[DK01] Minh Binh Do and Subbarao Kambhampati. “Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP”. In: *Journal of Artificial Intelligence* 132.2 (2001), pp. 151–182.

[DKS12] Carmel Domshlak, Michael Katz, and Alexander Shleyfman. “Enhanced Symmetry Breaking in Cost-Optimal Planning as Forward Search”. In: *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, (ICAPS 2012).* 2012.

[DKS13] Carmel Domshlak, Michael Katz, and Alexander Shleyfman. "Symmetry Breaking: Satisficing Planning and Landmark Heuristics". In: *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, (ICAPS 2013).* 2013.

[EJP97] E. Allen Emerson, Somesh Jha, and Doron A. Peled. "Combining Partial Order and Symmetry Reductions". In: *Tools and Algorithms for Construction and Analysis of Systems, Third International Workshop, (TACAS 1997).* 1997, pp. 19–34.

[FL02] Maria Fox and Derek Long. "Extending the Exploitation of Symmetries in Planning". In: *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems (AIPS 2002).* 2002, pp. 83–91.

[FL99] Maria Fox and Derek Long. "The Detection and Exploitation of Symmetry in Planning Problems". In: *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, (IJCAI 1999).* 1999, pp. 956–961.

[FN71] Richard Fikes and Nils J. Nilsson. "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving". In: *Journal of Artificial Intelligence* 2.3/4 (1971), pp. 189–208.

[Fu+11] Jicheng Fu, Vincent Ng, Farokh B. Bastani, and I-Ling Yen. "Simple and Fast Strong Cyclic Planning for Fully-Observable Nondeterministic Planning Problems". In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI 2011).* 2011, pp. 1949–1954.

[GB13] Hector Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning.* Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2013. ISBN: 9781608459698.

[GHP93] Patrice Godefroid, Gerard J. Holzmann, and Didier Pirottin. "State-Space Caching Revisited". In: *Proceedings of the Fourth International Conference on Computer Aided Verification (CAV 1992).* 1993, pp. 178–191.

[GHP95] Patrice Godefroid, Gerard Holzmann, and Didier Pirottin. "State-space Caching Revisited". In: *Journal of Formal Methods in System Design* 7.3 (1995), pp. 227–241.

[GNT04] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning - theory and practice.* Elsevier, 2004. ISBN: 978-1-55860-856-6.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem.* Vol. 1032. Springer-Verlag, 1996.

[GP93] Patrice Godefroid and Didier Pirottin. "Refining Dependencies Improves Partial-Order Verification Methods (Extended Abstract)". In: *Computer Aided Verification, Fifth International Conference, (CAV 1993).* 1993, pp. 438–449.

[GW92] Patrice Godefroid and Pierre Wolper. “Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties”. In: *Proceedings of the Third International Conference on Computer Aided Verification (CAV 1991).* 1992, pp. 332–342.

[HAW15] Robert C. Holte, Yusra Alkhazraji, and Martin Wehrle. “A Generalization of Sleep Sets Based on Operator Sequence Redundancy”. In: *Proceedings of the Twenty-Ninth AAAI Conference (AAAI 2015).* 2015, pp. 3291–3297.

[HB14] Robert C. Holte and Neil Burch. “Automatic Move Pruning for Single-Agent Search”. In: *Journal of AI Communications* 27.4 (2014), pp. 363–383.

[HD09] Malte Helmert and Carmel Domshlak. “Landmarks, Critical Paths and Abstractions: What’s the Difference Anyway?” In: *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009).* 2009, pp. 162–169.

[Hel06] Malte Helmert. “The Fast Downward Planning System”. In: *Journal of Artificial Intelligence Research (JAIR)* (2006), pp. 191–246.

[HG00] Patrik Haslum and Héctor Geffner. “Admissible Heuristics for Optimal Planning”. In: *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling Systems (AIPS 2000).* 2000, pp. 140–149.

[HHH07] Malte Helmert, Patrik Haslum, and Jörg Hoffmann. “Flexible Abstraction Heuristics for Optimal Sequential Planning”. In: *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007).* 2007, pp. 176–183.

[HN11] Jörg Hoffmann and Bernhard Nebel. “The FF Planning System: Fast Plan Generation Through Heuristic Search”. In: *Journal of Computing Research Repository (CoRR)* abs/1106.0675 (2011).

[HR08] Malte Helmert and Gabriele Röger. “How Good is Almost Perfect?” In: *Proceedings of the Twenty-Third Conference on Artificial Intelligence (AAAI 2008).* 2008, pp. 944–949.

[HZ01] Eric A. Hansen and Shlomo Zilberstein. “LAO*: A heuristic search algorithm that finds solutions with loops”. In: *Journal of Artificial Intelligence* 129.1-2 (2001), pp. 35–62.

[ID96] C. Norris Ip and David L. Dill. “Better Verification Through Symmetry”. In: *Journal of Formal Methods in System Design* 9.1/2 (1996), pp. 41–75.

[JS01] Andreas Junghanns and Jonathan Schaeffer. “Sokoban: Enhancing general single-agent search methods using domain knowledge”. In: *Journal of Artificial Intelligence* 129.1-2 (2001), pp. 219–251.

[KE09] Peter Kissmann and Stefan Edelkamp. “Solving Fully-Observable Non-deterministic Planning Problems via Translation into a General Game”. In: *Proceedings of Advances in Artificial Intelligence, Thirty-Second Annual German Conference on AI.* 2009, pp. 1–8.

[Kno94] Craig A. Knoblock. "Automatically Generating Abstractions for Planning". In: *Journal of Artificial Intelligence (JAIR)* 68.2 (1994), pp. 243–302.

[Kor85] Richard E. Korf. "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search". In: *Journal of Artifitial Intelligence (JAIR)* 27.1 (1985), pp. 97–109.

[KP95] Maciej Koutny and Marta Pietkiewicz-Koutny. *On the Sleep Sets Method for Partial Order Verification of Concurrent Systems.* Tech. rep. 495. Department of Computing Science, University of Newcastle upon Tyne, 1995.

[KS92] Henry A. Kautz and Bart Selman. "Planning as Satisfiability". In: *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI 1992).* 1992, pp. 359–363.

[Kut+08] Ugur Kuter, Dana S. Nau, Elnatan Reisner, and Robert P. Goldman. "Using Classical Planners to Solve Nondeterministic Planning Problems". In: *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, (ICAPS 2008).* 2008, pp. 190–197.

[Lev05] Hector J. Levesque. "Planning with Loops". In: *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI 2005).* 2005, pp. 509–515.

[McD96] Drew V. McDermott. "A Heuristic Estimator for Means-Ends Analysis in Planning". In: *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems (AIPS 1996).* 1996, pp. 142–149.

[McD99] Drew V. McDermott. "Using Regression-Match Graphs to Control Search in Planning". In: *Journal of Artificial Intelligence (JAIR)* 109.1-2 (1999), pp. 111–159.

[McM93] Kenneth L. McMillan. *Symbolic model checking.* Kluwer, 1993. ISBN: 978-0-7923-9380-1.

[NAB12] Raz Nissim, Udi Apsel, and Ronen I. Brafman. "Tunneling and Decomposition-Based State Reduction for Optimal Planning". In: *Proceedings of the Twentieth European Conference on Artificial Intelligence (ECAI 2012).* 2012, pp. 624–629.

[Ove81] William T. Overman. "Verification of Concurrent Systems: Function and Timing". AAI8121023. PhD thesis. 1981.

[Pel93] Doron A. Peled. "All from One, One for All: on Model Checking Using Representatives". In: *Proceedings of the Fifth International Conference on Computer Aided Verification (CAV 1993).* 1993, pp. 409–423.

[PH13] Florian Pommerening and Malte Helmert. "Incremental LM-Cut". In: *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, (ICAPS 2013).* 2013.

[PW92] J. Scott Penberthy and Daniel S. Weld. “UCPOP: A Sound, Complete, Partial Order Planner for ADL”. In: *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR 1992).* 1992, pp. 103–114.

[Rin09] Jussi Rintanen. “Planning and SAT”. In: *Handbook of Satisfiability.* 2009, pp. 483–504.

[Rin12] Jussi Rintanen. “Planning as satisfiability: Heuristics”. In: *Journal of Artificial Intelligence* 193 (2012), pp. 45–86.

[RN10] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)* Pearson Education, 2010. ISBN: 978-0-13-604259-4.

[Sei+16] Jendrik Seipp, Florian Pommerening, Silvan Sievers, Martin Wehrle, Chris Fawcett, and Yusra Alkhazraji. “Fast Downward Aidos (planner abstract)”. In: *In the First Unsolvability International Planning Competition (IPC 2016).* 2016.

[Shl+15] Alexander Shleyfman, Michael Katz, Malte Helmert, Silvan Sievers, and Martin Wehrle. “Heuristics and Symmetries in Classical Planning”. In: *Proceedings of the Twenty-Ninth Conference on Artificial Intelligence (AAAI 2015).* 2015, pp. 3371–3377.

[Sie+15a] Silvan Sievers, Martin Wehrle, Malte Helmert, and Michael Katz. “An Empirical Case Study on Symmetry Handling in Cost-Optimal Planning as Heuristic Search”. In: *Proceedings of the Thirty-Eighth Annual German Conference on AI (KI 2015) – Advances in Artificial Intelligence.* 2015, pp. 166–180.

[Sie+15b] Silvan Sievers, Martin Wehrle, Malte Helmert, Alexander Shleyfman, and Michael Katz. “Factored Symmetries for Merge-and-Shrink Abstractions”. In: *Proceedings of the Twenty-Ninth Conference on Artificial Intelligence (AAAI 2015).* 2015, pp. 3378–3385.

[SWH14] Silvan Sievers, Martin Wehrle, and Malte Helmert. “Bounded Intention Planning Revisited”. In: *Proceedings of the Twenty-First European Conference on Artificial Intelligence (ECAI 2014).* 2014, pp. 1097–1098.

[TK93] Larry A. Taylor and Richard E. Korf. “Pruning Duplicate Nodes in Depth-First Search”. In: *Proceedings of the Eleventh National Conference on Artificial Intelligence.* 1993, pp. 756–761.

[Val89] Antti Valmari. “Stubborn sets for reduced state space generation”. In: *Proceedings of Tenth International Conference on Applications and Theory of Petri Nets – Advances in Petri Nets 1990.* 1989, pp. 491–515.

[Weh+13] Martin Wehrle, Malte Helmert, Yusra Alkhazraji, and Robert Mattmüller. “The Relative Pruning Power of Strong Stubborn Sets and Expansion Core”. In: *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS 2013).* 2013, pp. 251–259.

[Weh+15] Martin Wehrle, Malte Helmert, Alexander Shleyfman, and Michael Katz. “Integrating Partial Order Reduction and Symmetry Elimination for Cost-Optimal Classical Planning”. In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, (IJCAI 2015).* 2015, pp. 1712–1718.

[WH12] Martin Wehrle and Malte Helmert. “About Partial Order Reduction in Planning and Computer Aided Verification”. In: *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012).* 2012.

[WH14] Martin Wehrle and Malte Helmert. “Efficient Stubborn Sets: Generalized Algorithms and Selection Strategies”. In: *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, (ICAPS 2014).* 2014.

[Win+17] Dominik Winterer, Yusra Alkhazraji, Michael Katz, and Martin Wehrle. “Stubborn Sets for Fully Observable Nondeterministic Planning”. In: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017).* 2017.

[WR11] Jason Wolfe and Stuart J. Russell. “Bounded Intention Planning”. In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI 2011).* 2011, pp. 2039–2045.

[WWK16] Dominik Winterer, Martin Wehrle, and Michael Katz. “Structural Symmetries for Fully Observable Nondeterministic Planning”. In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, (IJCAI 2016).* 2016, pp. 3293–3299.

[Xu+11] You Xu, Yixin Chen, Qiang Lu, and Ruoyun Huang. “Theory and Algorithms for Partial Order Based Reduction in Planning”. In: *CoRR* abs/1106.5427 (2011).