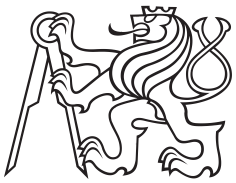
Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Inference of State Invariants for Domain-Independent Planning

Bc. Daniel Fišer

Supervisor: Ing. Antonín Komenda, Ph.D.
Study Programme: Open informatics
Field of Study: Artificial intelligence
May 2016

Acknowledgements

I would like to thank my supervisor Antonín Komenda for the valuable comments and remarks he has given me during the creation of this work.

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 20. May 2016

.....

Abstract

The mutual exclusion (mutex) state invariants are defined in a context of STRIPS planning as sets of facts from which maximally one can be true in any state reachable from the initial state. In this work, we introduce two new types of mutex invariants along with a detailed description of their structure and their relation to a general mutex invariant. Moreover, we provide a complexity analysis of decision problems that correspond to inference of different types of mutex invariants. Finally, three novel algorithms for inference of mutex invariants are described and experimentally compared with two different state-of-the-art algorithms.

Keywords: planning, STRIPS, invariant, mutex

Supervisor: Ing. Antonín Komenda, Ph.D.

Abstrakt

Stavové invarianty vzájemného vyloučení (mutexy) jsou v kontextu STRIPS plánování definovány jako množiny faktů, z nichž maximálně jeden může být platný v jakémkoli dosažitelném stavu. V této práci představíme dva nové typy mutexů společně s detailním popisem jejich struktury a vztahu k obecným mutexům. Dále poskytneme analýzu složitosti rozhodovacích problémů, které odpovídají problému odvozování různých typů mutexů. Nakonec popíšeme tři nové algoritmy pro odvozování mutexů a porovnáme je s dvěma různými state-of-the-art algoritmy.

Klíčová slova: plánování, STRIPS, invariant, mutex

Překlad názvu: Inference stavových invariantů pro doménově nezávislé plánování

Contents

Figures

Tables

Chapter 1
Introduction

State invariants in domain-independent planning are certain intrinsic properties of a particular planning problem that hold in all states reachable from the initial state. State invariants (as well as other types of invariants) tell something about the internal structure of the problem. This revealed structure can be further utilized in a process of solving the problem. State invariants could, for example, be used for a design of heuristic functions that can better guide search algorithms. They could be used for pruning the search space within which a plan is searched for or maybe even for reformulation of the original problem to some more simple form as a preprocessing step.

In this work, we are interested in inferring one particular type of state invariants called *mutual exclusion* (*mutex*) invariants. A mutex invariant states which facts cannot be true at the same time in any reachable state, i.e., a state can contain only one fact from the invariant or none of the facts. This definition corresponds to the mutexes as they were used in [Helmert, 2009]. The term mutex was also used in other works in a different meaning [Bonet and Geffner, 2001; Alcázar and Torralba, 2015]. Nevertheless, we decided to use this term because we think it well describes the essence of this type of invariant and it is well known to readers already familiar with this field of research.

The most straightforward application of mutex invariants is in translation to finite domain representation (FDR, or SAS$^+$) [Helmert, 2009]. Given a set of mutex invariants, FDR can be constructed by creating variables from those invariants that cover all facts. A special value "none of those" can be added to some variables if needed to cover a situation where none of the facts from the invariant is present in the state. Heuristic functions based on domain transition graphs (DTG) [Helmert, 2006; Torreño et al., 2014] could also benefit from mutex invariants. These heuristics can be constructed either from FDR representation or directly from the set of all inferred mutex invariants. State invariants (including mutex invariants) are also critical in improving the performance of SAT planners [Kautz and Selman, 1992; Sideris and Dimopoulos, 2010]. SAT planners are based on a formulation of planning tasks as a problem of satisfiability of logical formulas. State invariants expressed as a logical formula often significantly prune the search space and therefore improve efficiency of the solvers.

This work is not aimed at a design of any particular application using invariants but solely at analysis and inference of mutex invariants in a context of STRIPS planning [Fikes and Nilsson, 1971]. We introduce two new types of mutex invariants (fact-alternating mutex and restricted fact-alternating mutex) and we discuss in detail their properties and the relations between them and the general mutex invariant. Moreover, we provide a complexity analysis showing that inference of a maximum size mutexes is PSPACE-Complete whereas inferring a maximum size mutexes of the newly introduced types is NP-Complete.

Three new inference algorithms are introduced and compared from several different perspectives with two state-of-the-art algorithms. The first algorithm is able to extend a set of mutex invariants provided as its input. The second algorithm is designed to infer the newly introduced fact-alternating mutex invariants. It is based on a direct translation of the definition of fact-alternating mutexes into constraints of integer linear program (ILP). Solution of the ILP problem provides only one invariant at a time, so the ILP is refined in a cycle in such a way that all invariants are eventually discovered. The algorithm is proven to be complete with respect to maximal fact-alternating mutexes. The third algorithm infers restricted fact-alternating mutexes and it has a polynomial asymptotic complexity.

The work is organized as follows. A list of a related work is lied out in Chapter 2 where different types of invariants as well as different approaches to inferring the invariants are discussed. In Chapter 3, we formally define three types of mutex invariants, we discuss their properties and relations between them (Section 3.1) and we provide a formal analysis of their complexity (Section 3.2). In Section 3.3, 3.4, and 3.5 we describe three new inference algorithms that are experimentally evaluated in Chapter 4. We conclude with Chapter 5.

Chapter 2

Related Work

State invariants are formulas that are true in every state of a planning task reachable from the initial state by application of a sequence of operators. In this chapter we provide a brief discussion of different approaches to the inference of state invariants related to the approach presented in this work.

One of the first approaches to inference of state invariants was the DISCOPLAN system proposed by Gerevini and Schubert [1998, 2000]. The algorithm uses a *guess, check and repair* approach for generating invariants. Invariants are first hypothesized from definitions of operators. The consecutive steps involve verification that the invariants still hold in all reachable states and the unverified invariants are refined to form new invariants that are then in turn verified again. The refinements are based on sets of candidate supplementary conditions called "excuses" that are extracted during the verification phase. These "excuses" are extracted by an analysis considering all operators which allows the algorithm to make more informed choices in consequent refinement than the "excuses" that would be derived only from the first operator violating the invariant. However, this comes with an increased computational burden as noticed by Helmert [2009]. The algorithm is able to generate a wide range of different types of state invariants (or state constraints as they are called by Gerevini and Schubert) such as implicative constraints of a form $\phi \Rightarrow \psi$ stating that every state satisfying formulae ϕ also has to satisfy ψ, static constraints providing type information about predicates, or xor constraints providing an information about mutual exclusion of two literals given some additional conditions.

A Type Inference Module (TIM) proposed by Fox and Long [1998] and further extended by Cresswell et al. [2002] takes a domain description possibly without any type information and infers (or enrich) a type structure from the functional relationships in the domain. State invariants can be extracted from the way in which the inferred types are partitioned.

Rintanen [2000] proposed an iterative algorithm for generating state invariants. The algorithm uses a guess, check and repair approach and it is polynomial in time due to restrictions on a form and a length of invariants. The procedure starts with identification of an initial set of candidate invariants corresponding to the grounded facts in the initial state. In the following steps, the initial set of candidates is expanded with new invariants that are created

by expanding invariant candidates from the previous step using grounded operators. The invariant candidates that do not preserve their invariant property are rejected and new candidates that are weaker in the sense that they hold in more states than the original one are created. An interesting property of this algorithm is that it considers all invariant candidates during creation of new ones instead of expanding one invariant at a time.

Mukherji and Schubert [2005, 2006] took a completely different approach. Instead of analyzing operators of the planning task, state invariants are inferred from one or more reachable states. The set of reachable states can be obtained by random walks through a state space or by an exhaustive search with a bounded depth. State invariants are then inferred by an any-time algorithm employing a data analysis of the provided reachable states. The resulting invariants are not guaranteed to be correct in the sense that they do not have to hold in all reachable states besides those provided, but the authors suggest that some other algorithm, such that of Rintanen [2000], can be used for quick verification of the correctness of the invariants produced.

A generalization of h^{max} heuristic to a family of h^m heuristics [Haslum and Geffner, 2000; Haslum, 2009; Alcázar and Torralba, 2015] offers another method for generation of invariants. h^{max} is a widely known and well understood admissible heuristic for STRIPS planning. The heuristic value is computed on a relaxed reachability graph as a cost of most costly fact from a conjunction of reachable facts. The heuristic works with single facts, but it can be generalized to consider a conjunction of at most m facts instead. h^1 would then be equal to h^{max}, h^2 would build reachability graph with single facts and pairs of facts, h^3 would add also triplets of facts, and h^m would consider conjunctions of at most m facts. This heuristic is not bound by h^+ and even equals to the optimal heuristic for sufficiently large m, unfortunately the cost of computation increases exponentially in m.

The important property of h^m related to a generation of invariants is its ability to provide a set of fact conjunctions that are not reachable from the initial state. So, as well as we can say that the facts that does not appear in a reachability graph of h^1 (h^{max}) cannot affect a planning procedure, the same can be said about the unreachable conjunctions of m facts in the case of h^m. For example, an unreachable pair of facts in case of h^2 can be interpreted as an invariant stating that both facts from the pair cannot hold at the same time and similarly an unreachable triplet of facts in case of h^3 corresponds to an invariant stating that there is not any reachable state that contains all three facts at the same time.

The state invariants inferred by the algorithm introduced by Rintanen [2008] have a form of disjunction of facts possibly with a negation. The algorithm employs regression operators and satisfiability tests to check whether the clauses form invariants. Each clause initially consists of a single fact or a negation of a fact holding in the initial state. The clause that is not approved as an invariant is replaced by a set of weaker clauses each containing one additional fact (or its negation). The Rintanen's algorithm is able to produce invariants in a more general form than h^m invariants, because a h^m invariant

consisting of m facts corresponds to a disjunction $\neg f_1 \vee ... \vee \neg f_m$. Moreover, it was proven that the algorithm produces a superset of h^m invariants therefore it is a generalization of h^m heuristic.

An algorithm for translating PDDL planning tasks into a concise finite domain representation (FDR) was proposed by Helmert [2009]. The construction of FDR is based on identifying mutual exclusion (mutex) invariants. A mutual exclusion invariant states that at most one of the invariant facts can be present in any reachable state. The invariants are generated using guess, check and repair procedure running on the lifted PDDL domain. The procedure is initialized with small invariants counting only a single atom. The following step is proving the invariants through identification of so called threats. A threat emerges whenever there is an operator that has either two or more instances of invariant atoms in its add effects or instances in add effects are not compensated by the same number of instances in delete effects. The threatened invariants are then either discarded or refined by adding more atoms that could compensate the invariant in delete effects. The invariants that are not threatened are clearly invariants. The resulting invariants in a lifted form are grounded to a set of facts and in this final form they are used for a construction of variables in FDR.

Chapter 3

Mutual Exclusion Invariants

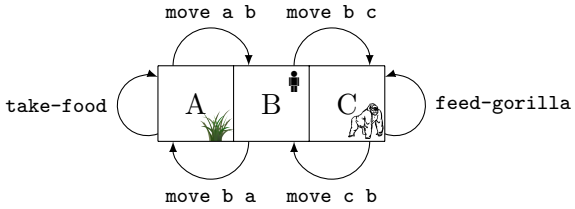
Definition 3.1. A STRIPS **planning task** Π is specified by a quadruple $\Pi = \langle \mathcal{F}, \mathcal{O}, s_{init}, s_{goal} \rangle$, where $\mathcal{F} = \{f_1, ..., f_n\}$ is a set of facts, $\mathcal{O} = \{o_1, ..., o_m\}$ is a set of grounded operators, $s_{init} \subseteq \mathcal{F}$ is an **initial state** and $s_{goal} \subseteq \mathcal{F}$ is a **goal** specification. A **state** $s \subseteq \mathcal{F}$ is a set of facts. An **operator** o is a triple $o = \langle \text{pre}(o), \text{add}(o), \text{del}(o) \rangle$, where $\text{pre}(o) \subseteq \mathcal{F}$ is a set of preconditions of an operator o, and $\text{add}(o) \subseteq \mathcal{F}$ and $\text{del}(o) \subseteq \mathcal{F}$ are sets of add and delete effects, respectively. All operators are well-formed, i.e., $\text{add}(o) \cap \text{del}(o) = \emptyset$ and $\text{pre}(o) \cap \text{add}(o) = \emptyset$. An operator o is **applicable** in a state s if $\text{pre}(o) \subseteq s$. The **resulting state** of applying an applicable operator o in a state s is the state $o[s] = (s \setminus \text{del}(o)) \cup \text{add}(o)$.

A sequence of operators $\pi = \langle o_1, ..., o_n \rangle$ is applicable in a state s_0 if there are states $s_1, ..., s_n$ such that o_i is applicable in s_{i-1} and $s_i = o_i[s_{i-1}]$ for $1 \leq i \leq n$. The resulting state of this application is $\pi[s_0] = s_n$. A state s is called a **reachable state** if there exists an applicable operator sequence π such that $\pi[s_{init}] = s$. A set of all reachable states is denoted by $\mathcal{S}^R$. An operator o is called a **reachable operator** iff it is applicable in some reachable state.

Consider the following simple example of the `gorilla-feeding` planning task depicted in Fig. 3.1. The planning task describes a zookeeper whose job is to feed a gorilla. The zookeeper can move between adjacent squares, he can take some food from a stock, carry it to the gorilla and feed it with the food if the gorilla is hungry. Moreover, if the gorilla was fed and it is hungry it escapes the zoo.

The planning task is described using six facts: `(at a)`, `(at b)`, and `(at c)` specify a position of the zookeeper, `(hungry)` and `(fed)` denote whether the gorilla is hungry or it was fed and `(carry-food)` specifies whether the zookeeper carries the food for the gorilla.

The operators in Fig. 3.1 are described using a simplified notation where preconditions are placed on the left hand side of the arrow symbol and the effects on the right hand side. The delete effects are listed with $\neg$ symbol in front of them and the add effects are listed without it. So for example operator `feed-gorilla` has three preconditions $\text{pre}(o) = \{$`(at c)`, `(hungry)`, `(carry-food)`$\}$, one add effect $\text{add}(o) = \{$`(fed)`$\}$, and two delete effects $\text{del}(o) = \{$`(hungry)`, `(carry-food)`$\}$. The planning task contains



Facts ($\mathcal{F}$):
`(at a)`, `(at b)`, `(at c)`,
`(hungry)`, `(fed)`
`(carry-food)`

Operators ($\mathcal{O}$):

`move a b:`	`(at a)` $\mapsto$ `(at b)`, $\neg$`(at a)`
`move b a:`	`(at b)` $\mapsto$ `(at a)`, $\neg$`(at b)`
`move b c:`	`(at b)` $\mapsto$ `(at c)`, $\neg$`(at b)`
`move c b:`	`(at c)` $\mapsto$ `(at b)`, $\neg$`(at c)`
`take-food:`	`(at a)`, `(hungry)` $\mapsto$ `(carry-food)`
`feed-gorilla:`	`(at c)`, `(hungry)`, `(carry-food)` $\mapsto$ `(fed)`, $\neg$`(hungry)`, $\neg$`(carry-food)`
`escape:`	`(fed)`, `(hungry)` $\mapsto$ $\neg$`(fed)`, $\neg$`(hungry)`

Initial state (s_{init}):
`(at b)`, `(hungry)`

Goal (s_{goal}):
`(fed)`

Figure 3.1: The `gorilla-feeding` planning task.

four operators for moving between adjacent squares (`move from-square to-square`), one operator for taking food from the square that contains the food stock (`take-food`), one operator for feeding the gorilla (`feed-gorilla`) that can be applied only on a square where the gorilla is and only when the zookeeper carries the food with him, and finally the operator describing escape of the gorilla from the zoo (`escape`). The initial state is set to $s_{init} = \{$`(at b)`, `(hungry)`$\}$ meaning that the zookeeper starts at the square `B` and the gorilla is hungry. The goal $s_{goal} = \{$`(fed)`$\}$ is to feed the gorilla.

All nine reachable states of the planning task are depicted in Fig. 3.2 along with all possible transitions between the states. The initial state is marked with the dashed box and all goal states with double border boxes. Fig. 3.2 shows that the zookeeper can freely move between adjacent squares which is reflected in the current state as exchange between `(at ...)` facts. Once the zookeeper takes food from the stock, the current state is extended by the fact `(carry-food)`. And once the gorilla is fed, the gorilla is not hungry anymore and the zookeeper does not carry the food. The operators `take-food` and `feed-gorilla` have no reverse operators which means that once they are used it is not possible to come back to the previous state.

Note also that the operator `take-food` can be used more than once but

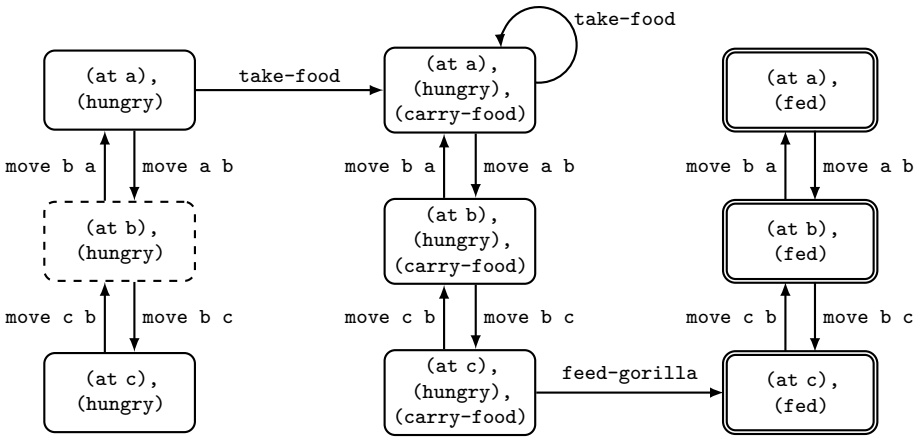


Figure 3.2: Reachable states and transitions between reachable states in the `gorilla-feeding` planning task.

without effect and the operator `escape` does not appear in Fig. 3.2 because it is not applicable in any reachable state (i.e., it is not a reachable operator). These two drawbacks could be fixed, but the `gorilla-feeding` planning task will be used throughout this chapter as a running example on which we will demonstrate different types of mutex invariants and this enables us to keep the example planning task very brief but with the ability to demonstrate the differences.

In Section 3.1 a mutual exclusion (mutex) invariant is formally defined and also two new (weaker) types of mutex invariants are introduced: a fact-alternating mutex invariant and a restricted fact-alternating mutex invariant. A complexity analysis of the defined invariant types is provided in Section 3.2. In Section 3.3 we introduce a novel algorithm that can be used for extending a list of already inferred mutex invariants. A complete algorithm for inference of fact-alternating mutex invariants is described in Section 3.4 and in Section 3.5 we introduce a polynomial algorithm for inference of restricted fact-alternating mutex invariants.

3.1 Mutex Hierarchy

Definition 3.2. A **mutex** $M \subseteq \mathcal{F}$ is a set of facts such that for every reachable state $s \in \mathcal{S}^R$ it holds that $|M \cap s| \leq 1$. A mutex that is not subset of any other mutex is called a **maximal mutex** (max-mutex).

A mutex (or mutex invariant) is defined as a set of facts from which maximally one can be true in any state reachable from the initial state by a sequence of operators. This is a very broad definition (adopted from [Helmert, 2009]) that covers all forms of mutex invariants which makes it hard to design a computationally feasible algorithm that would be able to produce all instances of mutex invariants. Therefore we introduce definitions of two new, more restricted, mutex invariants, namely the fact-alternating mutex and the restricted fact-alternating mutex. Both definitions are based on the

applicability of operators which makes them less complex then the general mutex as will be demonstrated in Section 3.2.

Definition 3.3. A **fact-alternating mutex** (fa-mutex) $M \subseteq \mathcal{F}$ is a set of facts such that $|M \cap s_{init}| \leq 1$ and for every operator $o \in \mathcal{O}$ it holds that $|M \cap \text{add}(o)| \leq |M \cap \text{pre}(o) \cap \text{del}(o)|$. A fa-mutex that is not subset of any other fa-mutex is called a **maximal fact-alternating mutex** (max-fa-mutex).

Proposition 3.4. *Every fact-alternating mutex is a mutex.*

Proof. (By induction) The first part $|M \cap s_{init}| \leq 1$ ensures a mutex property of M with respect to the initial state. Let s denote a state such that $|M \cap s| \leq 1$, i.e., the mutex property holds with respect to s. Now we need to make sure that the mutex property holds also for every state that is a resulting state from the application of an applicable operator o on s, i.e., for all $o \in \mathcal{O}$ such that $\text{pre}(o) \subseteq s$ an inequality $|M \cap o[s]| \leq 1$ holds. Since $|M \cap s| \leq 1$ and $\text{pre}(o) \subseteq s$ it follows that $|M \cap \text{pre}(o)| \leq 1$ and furthermore $|M \cap \text{pre}(o) \cap \text{del}(o)| \leq 1$. This means that three cases must be investigated. First, if $|M \cap \text{pre}(o) \cap \text{del}(o)| = 0$ then $|M \cap \text{add}(o)| = 0$ which means that no additional fact from M can be added to the resulting state and thus $|M \cap o[s]| \leq |M \cap s| \leq 1$. Second, if $|M \cap \text{pre}(o) \cap \text{del}(o)| = 1$ and $|M \cap \text{add}(o)| = 0$ then the same holds. Third, if $|M \cap \text{pre}(o) \cap \text{del}(o)| = 1$ and $|M \cap \text{add}(o)| = 1$ then $|M \cap \text{pre}(o)| = 1$ thus $|M \cap s| = 1$ (because $\text{pre}(o) \subseteq s$) so it follows that $M \cap \text{pre}(o) \cap \text{del}(o) = M \cap s \subseteq M \cap \text{del}(o)$. This means that $|M \cap (s \setminus \text{del}(o))| = 0$ so it follows that $|M \cap o[s]| = |M \cap ((s \setminus \text{del}(o)) \cup \text{add}(o))| = 1$, i.e., the mutex property is preserved also in the third case. Finally, since the mutex is defined for reachable states $\mathcal{S}^R$, every fact-alternating mutex must be also a mutex. □

The name fact-alternating mutex was chosen to stress its interesting property that lies in a mechanism by which facts from a fact-alternating mutex appear and disappear in particular states after application of operators. Consider some fact-alternating mutex M and some state s that does not contain any fact from M ($M \cap s = \emptyset$). Now we can ask whether any following state $\pi[s]$ can contain any fact from M. The answer is that it cannot because any operator o applicable in s that could add a new fact from M to the following state $o[s]$ would need to have a fact from M in its precondition ($M \cap \text{pre}(o) \neq \emptyset$) which is in contradiction with the assumption that s contain no fact from M. So it follows that facts from each particular fact-alternating mutex alternate between each other as new states are created and once the facts disappear from the state they cannot ever reappear again in any following state. This is formally proven in the following Proposition 3.5.

Proposition 3.5. *Let M denote a fact-alternating mutex and let s_i denote a state. If $|M \cap s_i| = 0$ then for every state $s_j = \pi[s_i]$ reachable from s_i by a sequence of applicable operators holds that $|M \cap s_j| = 0$.*

Proof. (By induction) Let us assume that $|M \cap s_k| = 0$ for some state $s_k = \pi[s_i]$ reachable from s_i. Now we will show that $|M \cap o[s_k]| = 0$ for any operator o applicable in s_k. Since o is applicable in s_k then $\text{pre}(o) \subseteq s_k$, and since

$|M \cap s_k| = 0$ then $|M \cap \text{pre}(o)| = 0$ therefore also $|M \cap \text{pre}(o) \cap \text{del}(o)| = 0$. So it follows that also $|M \cap \text{pre}(o) \cap \text{del}(o)| \geq |M \cap \text{add}(o)| = 0$ because M is a fa-mutex. Finally, this means that $|M \cap o[s_k]| = 0$ because o could not add any fact from M into $o[s_k]$. □

Proposition 3.5 describes a fact-alternating nature of fa-mutexes and also provides a method for proving that some facts are not reachable and therefore they can be safely removed from the planning task. It follows from Proposition 3.5 that given a fa-mutex that has empty intersection with the initial state, none of the facts the fa-mutex consists of can ever appear in any reachable state. Therefore they can be removed from the planning task entirely.

Similarly, fa-mutexes can be used for a detection of dead-end states and operators that can be safely removed from the planning task. A dead-end state is a state from which it is impossible to reach a goal by a sequence of applied operators. Consider a fa-mutex M having a non-empty intersection with the goal ($|M \cap s_{goal}| \geq 1$) and a reachable state s that does not contain any fact from M ($|M \cap s| = 0$). Such a state clearly must be a dead-end state, because it follows from Proposition 3.5 that all states reachable from s cannot contain any fact from M including the fact that appear in the goal.

The operators that have more than one fact from some mutex (and therefore also from some fa-mutex) in its preconditions cannot be applicable in any reachable state. Similarly, the operators with add effects containing more than one fact from some mutex (fa-mutex) are also unreachable, because the resulting state would be in contradiction with the mutex (fa-mutex). Such operators can be safely removed from the planning task. These two simple rules are not limited to the fact-alternating mutexes, but they can be used with any type of mutex invariant.

Fact-alternating mutexes provide one additional method for pruning superfluous operators. Consider a fa-mutex M such that $|M \cap s_{goal}| \geq 1$ and an operator o such that $|M \cap \text{add}(o)| = 0$ and $|M \cap \text{pre}(o) \cap \text{del}(o)| = 1$. The resulting state of application of the operator o would not contain any fact from fa-mutex M. Therefore such a state would be a dead-end state for the reasons already explained. This means that the operator o can be safely removed from the planning task because it can only produce dead-end states. In other words, the states resulting from application of the operator are not useful in finding a plan and therefore the operator itself is also not useful.

Definition 3.6. A **restricted fact-alternating mutex** (rfa-mutex) $M \subseteq \mathcal{F}$ is a set of facts such that $|M \cap s_{init}| \leq 1$ and for every operator $o \in \mathcal{O}$ it holds that $|M \cap \text{add}(o)| \leq |M \cap \text{pre}(o) \cap \text{del}(o)| \leq 1$. A rfa-mutex that is not subset of any other rfa-mutex is called a **maximal restricted fact-alternating mutex** (max-rfa-mutex).

Proposition 3.7. *Every restricted fact-alternating mutex is a fact-alternating mutex by Definition 3.3 and thus also a mutex by Definition 3.2.*

Proof. Definition 3.6 only adds two more restrictions on top of those put by Definition 3.3, namely $|M \cap \text{pre}(o) \cap \text{del}(o)| \leq 1$ and $|M \cap \text{add}(o)| \leq 1$. If

Table 3.1: A complete list of all mutexes, fa-mutexes and rfa-mutexes in the `gorilla-feeding` planning task. The maximal mutexes, maximal fa-mutexes and maximal rfa-mutexes are marked with a plus sign.

	mutex	**fa-mutex**	**rfa-mutex**
{}	✓	✓	✓
{(at a)}	✓	✗	✗
{(at b)}	✓	✗	✗
{(at c)}	✓	✗	✗
{(hungry)}	✓	✓	✓
{(fed)}	✓	✗	✗
{(carry-food)}	✓	✗	✗
{(at a), (at b)}	✓	✗	✗
{(at a), (at c)}	✓	✗	✗
{(at b), (at c)}	✓	✗	✗
{(carry-food), (fed)}	✓$^+$	✗	✗
{(hungry), (fed)}	✓$^+$	✓$^+$	✗
{(at a), (at b), (at c)}	✓$^+$	✓$^+$	✓$^+$

these restrictions hold then also those in Definition 3.3 must hold, therefore every rfa-mutex must be also a fa-mutex and therefore also a mutex. □

The triplet of mutexes, fa-mutexes, and rfa-mutexes forms a hierarchy of a different mutex types where rfa-mutexes are subset of fa-mutexes that are in turn subset of mutexes (rfa-mutex $\subseteq$ fa-mutex $\subseteq$ mutex). The mutex invariant is the most general definition of mutual exclusion between facts because it takes into account only reachable states. As we will demonstrate in the following section this has its consequences in the context of a complexity of this structure. A fact-alternating mutex covers a smaller number of sets of facts than mutex, but at least it provides a simple procedure for checking whether a given set of facts is a mutex. The restricted fact-alternating mutex adds more restrictions on fa-mutex that enables us to design a polynomial algorithm for its inference (Section 3.5).

A complete list of all types of mutexes in the example planning task is shown in Table 3.1. The maximal mutexes, fa-mutexes, and rfa-mutexes are marked with a plus sign. The simple corollary of the definition of the mutex is that every subset of any mutex is also a mutex. But the interesting property of fa-mutexes is that not every subset of this type of mutexes is also a fa-mutex and the same holds for rfa-mutexes. The reason being their strict definitions. This also means that even though it is always safe to consider a single fact to be a mutex this does not hold for fa-mutexes or rfa-mutexes. For example, `(at a)` is not a fa-mutex (and therefore neither a rfa-mutex) because the operator `move b a` has `(at a)` as its add effect, but it is not balanced by a delete effect and it cannot be because operators are not allowed to have the same facts in its add and delete effects. On the other hand, `(hungry)` is a fa-mutex (and also a rfa-mutex) because it is not listed as an add effect of any operator. This observation can be even generalized and we

can say that any single fact is a fa-mutex (and also a rfa-mutex) if and only if it does not appear in any add effect.

The facts `(carry-food)` and `(fed)` do not form a fa-mutex because of the operator `take-food` which adds `(carry-food)` fact, but does not delete the `(fed)` fact. This is exactly the type of mutex that is not covered by the fact-alternating mutex because the facts from this mutex appear in a state seemingly from nothing, i.e., the facts does not alternate between each other, but their appearance is conditioned on some other fact that is not part of the mutex.

The pair of facts `(hungry)` and `(fed)` is not a rfa-mutex because of the unreachable operator `escape` that increases the size of the intersection of the set of facts, preconditions and delete effects above the upper bound posed by Definition 3.6. This suggests that only unreachable operators can cause that some set of facts would be a fa-mutex but not a rfa-mutex. And indeed, if the planning task contains only reachable operators then fa-mutexes and rfa-mutexes both describe the exactly same set of mutexes which we formally prove in the following Proposition 3.8. Nevertheless, it does not make rfa-mutexes superfluous because a recognition of all reachable operators is as hard as the planning itself.

Proposition 3.8. *Let $\Pi = \langle \mathcal{F}, \mathcal{O}, s_{init}, s_{goal} \rangle$ denote a planning task and let $M \subseteq \mathcal{F}$ denote a set of facts. If the planning task contains only reachable operators then M is a fa-mutex iff M is a rfa-mutex.*

Proof. First, to prove by contradiction that if M is a fa-mutex then M is also a rfa-mutex, let us assume that M is a fa-mutex, but M is not a rfa-mutex. From the assumption it follows that there exist a reachable operator o and a reachable state s such that o is applicable in s, $|M \cap \mathrm{add}(o)| \leq |M \cap \mathrm{pre}(o) \cap \mathrm{del}(o)|$ (because M is a fa-mutex) and $|M \cap \mathrm{pre}(o) \cap \mathrm{del}(o)| > 1$ (because M is not a rfa-mutex). If $|M \cap \mathrm{pre}(o) \cap \mathrm{del}(o)| > 1$ then also $|M \cap \mathrm{pre}(o)| > 1$. Moreover, $\mathrm{pre}(o) \subseteq s$ because o is applicable in s, therefore also $|M \cap s| > 1$ which is in contradiction with the assumption that M is a fa-mutex. The other direction follows from Proposition 3.7 which concludes the proof. □

Proposition 3.8 is important in understanding the relationship between fa-mutexes and rfa-mutexes because what it says is that what makes the difference between these two types of mutexes are only unreachable operators.

The maximal mutexes, fa-mutex and rfa-mutexes are those that cannot be extended by any fact and still remain mutexes, fa-mutex or rfa-mutexes, respectively. Therefore all other mutexes, fa-mutexes or rfa-mutexes are already contained within the maximal ones, but we need to be careful while considering classification of the subsets of the maximal mutexes, fa-mutexes or rfa-mutexes. It is obviously true that every subset of a maximal mutex is also a mutex. Nevertheless, not every subset of a maximal fa-mutex (rfa-mutex) is also a fa-mutex (rfa-mutex) which we have already discussed. What remains to explain is a relationship between maximal mutexes and maximal fa-mutexes.

The question is whether every maximal fa-mutex is also a maximal mutex.

An intuitive answer would be that it is not because mutexes are more general then fa-mutexes. But to clearly demonstrate that, consider the following change in our running example. We can add a new fact `(at home)` and a new operator `go-home`: `(fed)` $\mapsto$ `(at home)`, $\neg$`(at a)`, $\neg$`(at b)`, $\neg$`(at c)`. This operator is clearly reachable and it adds one additional reachable state {`(fed)`, `(at home)`}. This means that in the altered planning task {`(at a)`, `(at b)`, `(at c)`} is no longer a maximal mutex because it is contained in the new maximal mutex {`(at a)`, `(at b)`, `(at c)`, `(at home)`}. Nevertheless, all maximal fa-mutexes still remain the same because the new operator is constructed in such a way that prevents to extend any fa-mutex listed in Table 3.1 by the fact `(at home)`.

3.2 Complexity Analysis

The complexity analysis of mutex structures we propose is based on an analysis of complexity classes of decision problems corresponding to the problems of finding the largest possible mutexes. We will show that such a decision problem for mutexes is asymptotically harder than for fa-mutexes and rfa-mutexes even though in general there can be exponentially many mutexes of all three types.

Definition 3.9. A **maximum mutex** (fa-mutex, rfa-mutex) is a mutex (fa-mutex, rfa-mutex) such that there is no other mutex (fa-mutex, rfa-mutex) consisting of more facts.

Definition 3.10. **Maximum-Mutex** (**Maximum-FA-Mutex**, **Maximum-RFA-Mutex**): Given a planning task Π, find a maximum mutex (fa-mutex, rfa-mutex). Decision problem: "Given Π and integer k, does Π contain a mutex (fa-mutex, rfa-mutex) of size at least k?"

A maximum mutex is a mutex that has the maximum possible number of facts in the corresponding planning task, i.e., maximum mutexes are the largest mutexes in a number of facts they consist of. It should be clear that every maximum mutex is also a maximal mutex by Definition 3.2 (but not the other way around) and the same holds for fa-mutexes and rfa-mutexes. Maximum-Mutex is a corresponding decision problem to the task of finding a maximum mutex. Similarly, Maximum-FA-Mutex is a decision problem corresponding to the finding a maximum fa-mutex and Maximum-RFA-Mutex deals with rfa-mutexes. In Section 3.2.1 we prove that Maximum-FA-Mutex and Maximum-RFA-Mutex are NP-Complete and in Section 3.2.2 we will show that Maximum-Mutex is PSPACE-Complete.

3.2.1 Maximum-FA-Mutex and Maximum-RFA-Mutex are NP-Complete

Definition 3.11. An undirected simple **graph** G is a tuple $G = \langle N, E \rangle$, where N denotes a set of nodes and E denotes a set of edges such that each edge $\{n_i, n_j\} \in E$ connects two different nodes ($n_i \neq n_j$) and there are no two

edges connecting the same nodes. A non-empty set C of nodes of G forms a **clique** if each node of C is connected by an edge to every other node of C. A clique that is not subset of any other clique is called a **maximal clique**. A **maximum clique** is a clique such that there is no other clique consisting of more nodes.

Definition 3.12. MAXIMUM-CLIQUE: Given a graph G, find a maximum clique. Decision problem: "Given G and integer k, does G contain a clique of size at least k?"

The MAXIMUM-CLIQUE problem is a well known NP-Complete decision problem which we use to show that MAXIMUM-FA-MUTEX and MAXIMUM-RFA-MUTEX are NP-Hard (Proposition 3.17). The reduction from MAXIMUM-CLIQUE is done using Algorithm 1 that translates any graph G into a planning task Π^G in a polynomial time. After the translation, it is shown that MAXIMUM-CLIQUE for G can be solved by solving MAXIMUM-FA-MUTEX or MAXIMUM-RFA-MUTEX for Π^G. In other words, we show that fa-mutex and rfa-mutex decision problems are at least as hard as some NP-Complete problem, in this case MAXIMUM-CLIQUE.

Proving that MAXIMUM-FA-MUTEX and MAXIMUM-RFA-MUTEX belong to NP is much easier because their definitions provide a verification algorithm running in a polynomial time which concludes the proof that MAXIMUM-FA-MUTEX and MAXIMUM-RFA-MUTEX are NP-Complete (Theorem 3.18). Moreover, it follows from the polynomial reduction, as we propose it, that the maximum possible number of maximal fa-mutexes or rfa-mutexes is exponential in a number of facts in the corresponding planning task (Proposition 3.19).

Algorithm 1: Translation of a graph into a planning task.

Input: A graph $G = \langle N, E \rangle$
Output: Planning task $\Pi^G = \langle \mathcal{F}, \mathcal{O}, s_{init}, s_{goal} \rangle$

$\mathcal{F} \leftarrow N \cup \{\top\}$, $\mathcal{O} \leftarrow \{\}$, $s_{init} \leftarrow \{\top\}$, $s_{goal} \leftarrow \{\}$;
for each $n_1, n_2 \in N$ *such that* $\{n_1, n_2\} \notin E$ **do**
    Create a new operator o and add it to $\mathcal{O}$;
    $\text{pre}(o) \leftarrow \{\top\}$;
    $\text{del}(o) \leftarrow \{\top\}$;
    $\text{add}(o) \leftarrow \{n_1, n_2\}$;
end

The proof that MAXIMUM-FA-MUTEX and MAXIMUM-RFA-MUTEX are NP-Hard starts with some auxiliary lemmas. Lemma 3.13 shows that we can use Algorithm 1 to translate graph G into a corresponding planning task Π^G and all cliques (including the maximum ones) are preserved during the translation in a form of fa-mutexes. More precisely, if C is a clique in G then $C \cup \{\top\}$ is a fa-mutex in Π^G. In Lemma 3.14 the other direction is proven, i.e., it is shown that every fa-mutex containing $\top$ corresponds to a clique in the original graph G. This leads to Lemma 3.15 that joins the previous two lemmas into equivalence between every clique C and the corresponding

fa-mutex $C \cup \{\top\}$. Moreover, it is also shown that the equivalence holds also for rfa-mutexes because all operators in Π^G are reachable and therefore every fa-mutex is also a rfa-mutex (Proposition 3.8). The last remaining piece of the proof of correctness of the polynomial reduction from Maximum-Clique problem is to show that there are not any maximum fa-mutexes (or rfa-mutexes) that does not contain $\top$, i.e., we must show that if we find a maximum fa-mutex (rfa-mutex) then we can reconstruct a maximum clique in the original graph G from it and therefore Maximum-Clique problem can be solved by solving Maximum-FA-Mutex or Maximum-RFA-Mutex. In Lemma 3.16, we prove even stronger statement saying that not only all maximum fa-mutexes (rfa-mutexes) but all fa-mutexes (rfa-mutexes) contain $\top$. Therefore Lemma 3.15 can be safely used to prove that the polynomial reduction from Maximum-Clique problem is correct, thus Maximum-FA-Mutex and Maximum-RFA-Mutex are both NP-Hard (Proposition 3.17).

The main contribution of this section is formulated in Theorem 3.18 stating that both Maximum-FA-Mutex and Maximum-RFA-Mutex are NP-Complete. Once we have proven that both decision problems are NP-Hard then it easily follows that they must be NP-Complete because any fa-mutex or rfa-mutex can be verified in a polynomial number of steps by checking the initial state and all operators.

For the rest of this section, let G denote a graph and let $\Pi^G = \langle \mathcal{F}, \mathcal{O}, s_{init}, s_{goal} \rangle$ denote a planning task generated by Algorithm 1 from G.

Lemma 3.13. *If C is a clique in G then $M = C \cup \{\top\}$ is a fa-mutex in Π^G.*[1]

Proof. To prove the lemma by contradiction let us assume that C is a clique and $M = C \cup \{\top\}$ is not a fa-mutex. Since $s_{init} = \{\top\} \subseteq M$ and $|M \cap \mathrm{pre}(o) \cap \mathrm{del}(o)| = 1$ for every operator o (because $\mathrm{pre}(o) = \mathrm{del}(o) = \{\top\}$) there must exist an operator $o^\star \in \mathcal{O}$ such that $|M \cap \mathrm{add}(o^\star)| \geq 2$. Since $\top$ is not part of any add effect and all add effects contain exactly two facts, it must hold that $\mathrm{add}(o^\star) \subseteq C$. This is in contradiction with assumption that C is a clique because all add effects are created only from the pairs of nodes that are not joined by an edge and there is no such pair of nodes in C by definition. Therefore if C is a clique then M is a fa-mutex. □

Lemma 3.14. *If $M = C \cup \{\top\}$ is a fa-mutex in Π^G then C is a clique in G.*

Proof. To prove the lemma by contradiction let us assume that $M = C \cup \{\top\}$ is a fa-mutex and C is not a clique. If C is not a clique then there exist $n_1, n_2 \in C$ such that n_1 and n_2 are not connected by an edge in G. So it follows that there exists an operator $o \in \mathcal{O}$ such that $\mathrm{add}(o) = \{n_1, n_2\}$ and $\mathrm{pre}(o) = \mathrm{del}(o) = \{\top\}$ therefore $|M \cap \mathrm{add}(o)| = 2 > |M \cap \mathrm{pre}(o) \cap \mathrm{del}(o)| = 1$. This is in contradiction with the assumption that M is a fa-mutex therefore if M is a fa-mutex then C is a clique. □

Lemma 3.15. *C is a clique in G iff $M = C \cup \{\top\}$ is a fa-mutex (rfa-mutex) in Π^G.*

Proof. All operators in Π^G are reachable because all have only $\top$ in their preconditions therefore they are applicable in the initial state. So it follows

[1] We slightly abuse notation here and in the following lemmas to simplify the notation.

from Proposition 3.8 that if M is a fa-mutex then M is also a rfa-mutex. Therefore Lemma 3.13 proves the direction from left to right and Lemma 3.14 the other direction for both equivalences. □

Lemma 3.16. *For every maximal fa-mutex M in Π^G it holds that $\top \in M$.*

Proof. Let N denote a fa-mutex such that $\top \notin N$. Now we prove that $M = \{\top\} \cup N$ is also a fa-mutex. Since $s_{init} = \{\top\}$ then surely $|M \cap s_{init}| \leq 1$. For every operator $o \in \mathcal{O}$ holds that $\text{pre}(o) = \text{del}(o) = \{\top\}$ and $\top \notin \text{add}(o)$ and $|N \cap \text{add}(o)| \leq |N \cap \text{pre}(o) \cap \text{del}(o)|$. So it follows that $|N \cap \text{add}(o)| = |N \cap \text{pre}(o) \cap \text{del}(o)| = 0$ therefore $|M \cap \text{add}(o)| = 0 \leq |M \cap \text{pre}(o) \cap \text{del}(o)| = 1$ therefore M is a fa-mutex. Finally, since every fa-mutex can be extended by $\top$ then surely every maximal fa-mutex must contain $\top$. □

Proposition 3.17. *Maximum-FA-Mutex and Maximum-RFA-Mutex are NP-Hard.*

Proof. We will reduce Maximum-Clique to Maximum-FA-Mutex (Maximum-RFA-Mutex). Any graph G can be translated into a planning task Π^G using Algorithm 1 running in a polynomial time, namely $O(n^2)$, where n is a number of nodes in G. From Lemma 3.16 it follows that a maximum fa-mutex (rfa-mutex) must contain $\top$ and then from Lemma 3.15 it follows that C is a maximum clique in G iff $M = C \cup \{\top\}$ is a maximum fa-mutex (rfa-mutex) in Π^G. Therefore Maximum-FA-Mutex (Maximum-RFA-Mutex) is NP-Hard. □

Theorem 3.18. *Maximum-FA-Mutex and Maximum-RFA-Mutex are NP-Complete.*

Proof. Maximum-FA-Mutex (Maximum-RFA-Mutex) is NP-Hard according to Proposition 3.17. It is easy to see that given a set of facts it can be verified as a fa-mutex (rfa-mutex) by checking the initial state and all operators according to Definition 3.3 (Definition 3.6). The verification procedure runs in a polynomial number of steps in a number of facts and operators. Therefore Maximum-FA-Mutex (Maximum-RFA-Mutex) is NP-Complete. □

Proposition 3.19. *The maximum possible number of maximal fa-mutexes (rfa-mutexes) in a planning task Π is exponential in a number of facts.*

Proof. It follows from Lemma 3.15 and Lemma 3.16 that for every possible graph it is possible to construct a planning task in which every maximal fa-mutex (rfa-mutex) corresponds to some maximal clique and vice versa. The maximum possible number of maximal cliques in a graph is exponential in a number of nodes (namely $c \cdot 3^{n/3}$ where n is number of nodes and $c \in \{1, 4/3, 2\}$ depending on n mod 3) [Moon and Moser, 1965]. This makes the lower bound an exponential. The upper bound is the maximum number of subsets of $\mathcal{F}$, which is also an exponential ($2^{|\mathcal{F}|}$). This makes the maximum possible number of maximal fa-mutexes (rfa-mutexes) exponential in a number of facts. □

3.2.2 Maximum-Mutex is PSPACE-Complete

Definition 3.20. Plan-Sat: Given a planning task Π, determine the existence of a solution.

Proposition 3.17 states that Maximum-FA-Mutex is NP-Hard, i.e., it is at least as hard as the hardest problems in NP. Since every fa-mutex is also mutex, inferring mutexes cannot be easier than inferring fa-mutexes therefore also Maximum-Mutex must be NP-Hard. In this section, we will precise this statement and we will show that Maximum-Mutex is PSPACE-Complete (Theorem 3.25). First, it will be proven that it is PSPACE-Hard (Proposition 3.22) using a polynomial reduction from Plan-Sat which is known to be PSPACE-Complete [Bylander, 1994]. Then we will present a PSPACE algorithm (Algorithm 3) solving the Maximum-Mutex problem which leads to a conclusion that Maximum-Mutex is PSPACE-Complete. Moreover, we will show that the maximum possible number of maximal mutexes is exactly the same as maximal fa-mutexes or maximal rfa-mutex and we will express this number exactly.

Algorithm 2: Translation of a planning task into another planning task where Maximum-Mutex can be used to solve Plan-Sat.

Input: Planning task $\Pi = \langle \mathcal{F}, \mathcal{O}, s_{init}, s_{goal} \rangle$
Output: Planning task $\Pi^M = \langle \mathcal{F}^M, \mathcal{O}^M, s_{init}, s_{goal} \rangle$

1 $\mathcal{F}^M \leftarrow \mathcal{F} \cup \{\bot\}$;
2 Create a new operator o^{sat};
3 $\text{pre}(o^{\text{sat}}) \leftarrow s_{goal}$;
4 $\text{del}(o^{\text{sat}}) \leftarrow \{\}$;
5 $\text{add}(o^{\text{sat}}) \leftarrow \mathcal{F}^M$;
6 $\mathcal{O}^M \leftarrow \mathcal{O} \cup \{o^{\text{sat}}\}$;

Lemma 3.21. *Let Π denote a planning task, let Π^M denote a planning task generated by Algorithm 2 from Π and let M denote a maximum mutex in Π^M. A solution of Π exists iff $|M| \leq 1$.*

Proof. A solution of Π exists iff there exists a reachable state s such that $s_{goal} \subseteq s$. If such s exists then o^{sat} is a reachable operator and thus its resulting state $s^{\text{sat}} = \mathcal{F}^M$ is also reachable. So it follows that every mutex of Π^M consist of at most one fact because any mutex N having more than one fact would violate a mutex property on s^{sat} ($|N \cap s^{\text{sat}}| \geq 2$). Therefore if a solution of Π exists then $|M| \leq 1$.

To prove the other direction by contradiction, let us assume that we have a maximum mutex M in Π^M such that $|M| \leq 1$ and Π has no solution. If Π has no solution then there does not exist a reachable state s such that $s_{goal} \subseteq s$, which means that o^{sat} is not applicable in any reachable state therefore also the state $s^{\text{sat}} = \mathcal{F}^M$ is not reachable. But the fact $\bot$ appears in Π^M only in s^{sat} therefore any mutex in Π^M can be extended by $\bot$ and it still remains a mutex. Finally, since in any planning task exists a mutex of size at least one (a single fact is always a mutex) and since such a mutex can be in

Π^M extended by $\bot$, then it follows that a maximum mutex M must have at least two facts ($|M| \geq 2$) which is in contradiction with the assumption that $|M| \leq 1$. Therefore if $|M| \leq 1$ then Π has a solution. □

Proposition 3.22. *Maximum-Mutex is PSPACE-Hard.*

Proof. We will reduce Plan-Sat to Maximum-Mutex. Any planning task Π can be translated to a different planning task Π^M by Algorithm 2 running in a polynomial time. It follows from Lemma 3.21 that we can determine whether Π has a solution by solving Maximum-Mutex problem on Π^M in the following way. If the maximum mutex in Π^M has at most one fact then the planning task Π has a solution. If the maximum mutex in Π^M has more then one fact then the planning task Π does not have a solution. Therefore Maximum-Mutex is PSPACE-Hard. □

Once we have proven that Maximum-Mutex is PSPACE-Hard, proving that it is also PSPACE-Complete requires to show that it is possible to infer a maximum mutex using a polynomial amount of space. Such an inferring algorithm is listed in Algorithm 3. The main idea of the algorithm is that every set of facts M of size at least two is a mutex if and only if every pair of facts from M is also a mutex (Proposition 3.23). In other words, if we are able to infer all mutexes containing exactly two facts, we can always use these pair mutexes for a construction of all other mutexes of size at least three. The main cycle of Algorithm 3 uses Plan-Sat to prove whether each pair of facts is mutex or it is not, i.e., whether the facts are part of some reachable state or not. The inferred pair mutexes are used for a construction of a graph where each edge corresponds to one pair mutex. And finally, Maximum-Clique is used to infer a maximum mutex. Such an algorithm clearly uses only a polynomial amount of space which is formally proven in Lemma 3.24. Theorem 3.25 just joins Proposition 3.22 (Maximum-Mutex is PSPACE-Hard) and Lemma 3.24 (Maximum-Mutex belongs to PSPACE) to formulate the main contribution of this section, i.e., the proof that Maximum-Mutex is PSPACE-Complete.

Proposition 3.23. *Let $M \subseteq \mathcal{F}$ denote a set of facts such that $|M| \geq 2$ and let $\mathcal{D}^M$ denote a set of all pairs of all facts from M, i.e., $\mathcal{D}^M = \{\{f_i, f_j\} \mid f_i, f_j \in M, f_i \neq f_j\}$. M is a mutex iff every $P \in \mathcal{D}^M$ is a mutex.*

Proof. It is easy to see that if M is a mutex (i.e., $|s \cap M| \leq 1$ for every reachable state s) then also every subset of M must be a mutex (i.e., for every $N \subseteq M$ holds that also $|s \cap N| \leq 1$ for every reachable state s).

To prove the other direction by a contradiction let us assume that every $P \in \mathcal{D}^M$ is a mutex, but M is not mutex. If M is not a mutex then there exists a reachable state s such that $|s \cap M| \geq 2$. This means that there must exist a pair of facts $\{f_1, f_2\}$ such that $f_1, f_2 \in M$ and $f_1, f_2 \in s$ which is in contradiction with the assumption that every $P \in \mathcal{D}^M$ is a mutex because $\{f_1, f_2\}$ must belong to $\mathcal{D}^M$ by definition. □

Lemma 3.24. *Given a planning task Π, Algorithm 3 returns a maximum mutex using a polynomial amount of space in the size of the input.*

Algorithm 3: MAXIMUM-MUTEX

Input: Planning task $\Pi = \langle \mathcal{F}, \mathcal{O}, s_{init}, s_{goal} \rangle$
Output: The largest mutex M

```
1 Construct a complete graph
  G = ⟨N = F, E = {{f_i, f_j} | f_i, f_j ∈ F, f_i ≠ f_j}⟩;
2 for each f_1, f_2 ∈ F such that f_1 ≠ f_2 do
3 |   if PLAN-SAT (Π^sat = ⟨F, O, s_init, {f_1, f_2}⟩) then
4 |   |   E ← E \ {f_1, f_2};
5 |   end
6 end
7 M ← MAXIMUM-CLIQUE (G);
```

Proof. Algorithm 3 starts with a complete graph constructed from the facts as its nodes. Then, in $|\mathcal{F}|^2$ steps, each pair of facts is checked by PSPACE-Complete PLAN-SAT whether they appear together in any reachable state. If they do, an edge connecting those two facts is removed from the graph. The edges remaining in the graph connect only those facts that never appear together in the same reachable state. Therefore every pair of facts connected by an edge is a mutex.

MAXIMUM-CLIQUE retrieves a maximum mutex in the last step because it follows from Proposition 3.23 that having all mutexes of size two is enough to construct any other mutex which covers also maximum mutexes and if there are no edges left in the graph then MAXIMUM-CLIQUE returns a single fact which is always a mutex.

Finally, since MAXIMUM-CLIQUE is NP-Complete and Algorithm 3 uses a polynomial number of calls of the PSPACE-Complete algorithm PLAN-SAT, Algorithm 3 must be in PSPACE. □

Theorem 3.25. *MAXIMUM-MUTEX is PSPACE-Complete.*

Proof. MAXIMUM-MUTEX is PSPACE-Hard (Proposition 3.22) and it also belongs to PSPACE (Lemma 3.24) therefore MAXIMUM-MUTEX is PSPACE-Complete. □

Proposition 3.19 states that the maximum number of maximal fa-mutexes (rfa-mutexes) is exponential in a number of facts. The proof of Proposition 3.19 is based on the proposed procedure that can translate any graph into a planning task in such a way that every maximal fa-mutex (rfa-mutex) corresponds to a maximal clique in the original graph. This enables us to enumerate the lower bound on the maximum possible number of maximal fa-mutexes (rfa-mutexes) as the maximum possible number of maximal cliques.

It follows from Proposition 3.23 that given a complete list of all mutexes consisting of exactly two facts, all maximal mutexes can be constructed using an algorithm for listing all maximal cliques. This means that the maximum possible number of maximal mutexes is exactly the same as the maximum possible number of maximal cliques. Furthermore, since the same number is

the lower bound on the maximum possible number of maximal fa-mutexes (rfa-mutexes) and since every fa-mutex (rfa-mutex) is also a mutex, the maximum possible number of maximal mutexes, maximal fa-mutexes and maximal rfa-mutexes are exactly the same, which we formally prove in the following Proposition 3.26.

Proposition 3.26. *Let $\Pi = \langle \mathcal{F}, \mathcal{O}, s_{init}, s_{goal} \rangle$ denote a planning task and let $n = |\mathcal{F}|$ denote a number of facts in Π. The maximum possible number $\mu(n)$, $\mu_{fa}(n)$, and $\mu_{rfa}(n)$ of maximal mutexes, maximal fa-mutexes, and maximal rfa-mutexes, respectively, for $n \geq 2$, is the following:*

$$\mu(n) = \mu_{fa}(n) = \mu_{rfa}(n) = \begin{cases} 3^{n/3}, & \text{if } n \bmod 3 = 0; \\ \frac{4}{3} \cdot 3^{n/3}, & \text{if } n \bmod 3 = 1; \\ 2 \cdot 3^{n/3}, & \text{if } n \bmod 3 = 2. \end{cases}$$

Proof. It follows from Proposition 3.23 that all maximal mutexes can be constructed from a complete set of pair mutexes using an algorithm for enumeration of all maximal cliques. Moreover, it is easy to see that given a set of facts it is always possible to construct a planning task that would contain any combination of pair mutexes. It follows from the proof of Proposition 3.19 that the same holds for maximal fa-mutexes and maximal rfa-mutexes. This means that the maximum possible number of maximal mutexes, fa-mutexes and rfa-mutexes in a planning task is exactly the same as the maximum possible number of maximal cliques in a graph [Moon and Moser, 1965]. □

3.3 Inference of Mutex Invariants

Several algorithms for inference of mutex invariants was already described in Chapter 2. In this section, we present a novel polynomial algorithm that takes a set of mutex invariants generated using a different method as its input and it uses a relatively simple inference rule to generate some additional mutexes. The inference rule is encapsulated in Proposition 3.29 preceded by two auxiliary definitions.

Definition 3.27. An **f_1-mutex-set** $\mathcal{M}^{f_1}$ is a set of mutexes such that for every mutex $M \in \mathcal{M}^{f_1}$ it holds that $f_1 \in M$.

Definition 3.28. An **f_1-f_2-relevant operator** o is an operator such that $f_1 \in \text{add}(o)$ and $f_2 \notin \text{del}(o)$. Given a planning task $\Pi = \langle \mathcal{F}, \mathcal{O}, s_{init}, s_{goal} \rangle$, the set of all f_1-f_2-relevant operators from $\mathcal{O}$ is denoted by $\mathcal{O}^{f_1, \not\prec f_2}$.

Proposition 3.29. *Let $\Pi = \langle \mathcal{F}, \mathcal{O}, s_{init}, s_{goal} \rangle$ denote a planning task. Let $M = \{f_1, f_2\} \subseteq \mathcal{F}$ denote a set of facts such that $f_1 \neq f_2$. Let $\mathcal{M}^{f_1}$ and $\mathcal{M}^{f_2}$ denote an f_1-mutex-set and an f_2-mutex-set, respectively. And let $\mathcal{O}^{f_1, \not\prec f_2}$ and $\mathcal{O}^{f_2, \not\prec f_1}$ denote all f_1-f_2-relevant and f_2-f_1-relevant operators, respectively.*

If

1. *$|M \cap s_{init}| \leq 1$, and*
2. *for every $o \in \mathcal{O}$ holds that $|M \cap add(o)| \leq 1$, and*

3. *for every* $o \in \mathcal{O}^{f_1, \not f_2}$ *holds that there exists a mutex* $N \in \mathcal{M}^{f_2}$ *such that* $|(N \setminus \{f_2\}) \cap pre(o)| \geq 1$, *and*
4. *for every* $o \in \mathcal{O}^{f_2, \not f_1}$ *holds that there exists a mutex* $N \in \mathcal{M}^{f_1}$ *such that* $|(N \setminus \{f_1\}) \cap pre(o)| \geq 1$

then M *is a mutex.*

Proof. (By induction) The first condition ensures the mutex property of M with respect to the initial state. Let s denote a reachable state such that $|M \cap s| \leq 1$, i.e., the mutex property holds with respect to s. Now we need to prove that for all $o \in \mathcal{O}$ such that $\text{pre}(o) \subseteq s$ an inequality $|M \cap o[s]| \leq 1$ holds. Since M consist of two facts, only three cases must be investigated.

- There is no fact from M in s, i.e., $M \cap s = \emptyset$. In this case the mutex property of any $o[s]$ can be violated only if o has $M \subseteq \text{add}(o)$ which is in contradiction with condition 2.
- $f_1 \notin s$ and $f_2 \in s$. In this case the mutex property of some $o[s]$ can be violated only if there exists an operator o such that $\text{pre}(o) \subseteq s$ and $f_1 \in \text{add}(o)$ and $f_2 \notin \text{del}(o)$ which means that $o \in \mathcal{O}^{f_1, \not f_2}$. So it follows from the condition 3 that $\text{pre}(o)$ must contain a fact mutually exclusive with f_2 and therefore also s must contain a fact mutually exclusive with f_2. Therefore s contains two mutually exclusive facts (one of which is f_2) which is in contradiction with the assumption that s is a reachable state because the mutex property must hold for all reachable states.
- $f_1 \in s$ and $f_2 \notin s$ is a similar case to the previous one except the condition 4 instead of 3 must be used. □

The first condition of Proposition 3.29 is self-evident: a pair of facts can be a mutex only if they are a mutex with respect to the initial state. The last three conditions are based on the assumption that all operators are reachable. Under this assumption, the second condition is also self-evident. If there is an operator that adds both facts to the resulting state then those facts cannot form a mutex. The last two conditions are the heart of the inference rule. What they say is that if there exists an operator that adds one of the facts to the resulting state, but does not deletes the other one, we can use the the operator's preconditions and the known mutexes to deduce whether the operator preserves the mutex property of the two facts. That is, if an operator o adds a fact f_1 and its precondition contains some fact f_m that is a mutex with f_2, we know that the state on which is o applicable must also contain f_m therefore it cannot contain f_2. This means that such an operator cannot violate a mutex property between f_1 and f_2.

The algorithm using Proposition 3.29 is encapsulated in Algorithm 4. The algorithm starts with removing unreachable operators that can be detected as such using the provided set of pair mutexes (mutexes consisting of two facts). This can be achieved by checking preconditions and add effects whether they violate any known pair mutex. If they do then the corresponding operator can be safely removed.

Algorithm 4: Inference of mutex invariants.

Input: Planning task $\Pi = \langle \mathcal{F}, \mathcal{O}, s_{init}, s_{goal} \rangle$, a set of pair mutexes $\mathcal{M}^2$
Output: A set of pair mutexes $\mathcal{M}^2$

1 Remove unreachable operators from $\mathcal{O}$ using $\mathcal{M}^2$;
2 $\mathcal{C} \leftarrow \{\{f_1, f_2\} \mid f_1, f_2 \in \mathcal{F}, f_1 \neq f_2\} \setminus \mathcal{M}^2$;
3 **for each** $\{f_1, f_2\} \in \mathcal{C}$ **do**
4 $\quad \mathcal{M}^{f_1} \leftarrow \{N \mid N \in \mathcal{M}^2, f_1 \in N\}$;
5 $\quad \mathcal{M}^{f_2} \leftarrow \{N \mid N \in \mathcal{M}^2, f_2 \in N\}$;
6 $\quad \mathcal{O}^{f_1, \not\in f_2} \leftarrow \{o \mid o \in \mathcal{O}, f_1 \in \mathrm{add}(o), f_2 \notin \mathrm{del}(o)\}$;
7 $\quad \mathcal{O}^{f_2, \not\in f_1} \leftarrow \{o \mid o \in \mathcal{O}, f_2 \in \mathrm{add}(o), f_1 \notin \mathrm{del}(o)\}$;
8 $\quad$ **if** $\{f_1, f_2\}$ *passes a check on all four conditions of Proposition 3.29* **then**
9 $\quad\quad \mathcal{M}^2 \leftarrow \mathcal{M}^2 \cup \{\{f_1, f_2\}\}$;
10 $\quad\quad$ **go to** 1;
11 $\quad$ **end**
12 **end**

A set of candidates for mutexes is created on the second line of the algorithm as all possible pairs of facts except those about which we already know they are mutexes. Then each pair of mutexes is tested according to Proposition 3.29 and the pair that passes the test is added to the list of inferred mutexes and the cycle starts again with removing unreachable operators. The algorithm terminates when there are not any candidates that pass the test.

It should be obvious that the algorithm runs in a polynomial time because every step of the algorithm is polynomial and it contains only polynomial number cycles because there is only $O(|\mathcal{F}|^2)$ candidate pairs possible.

The proposed algorithm can be used with a conjunction with any mutex inferring algorithm because any mutex can be decomposed into a set of pair mutexes (Proposition 3.23). This also includes the algorithms described in the following two sections. The performance of a combination of Algorithm 4 and several other inference algorithms is evaluated in Chapter 4.

3.4 Inference of Fact-Alternating Mutex Invariants

Definition 3.3 provides a concise description of fact-alternating mutexes that can be used for a design of an algorithm that generates them if it is used as a construction block of an integer linear program (ILP). Such an ILP can be constructed in the following way.

Each variable x_i of the ILP corresponds to a fact $f_i \in \mathcal{F}$ from the planning problem. Variables can acquire only binary values 0 or 1, 0 meaning that the corresponding fact is not present in the fa-mutex and 1 meaning the corresponding fact is part of the fa-mutex. For example having 3 facts f_1, f_2, f_3, the corresponding ILP would consist of 3 binary variables x_1, x_2, x_3

and an assignment to the variables $x_1 = 1, x_2 = 0, x_3 = 1$ would mean that the fa-mutex M consists of facts f_1 and f_3 ($M = \{f_1, f_3\}$).

The definition of a fact-alternating mutex can be rewritten into ILP constraints as follows:

$$\sum_{f_i \in s_{init}} x_i \leq 1, \tag{3.1}$$

$$\forall o \in \mathcal{O} : \sum_{f_i \in \mathrm{add}(o)} x_i \leq \sum_{f_i \in \mathrm{del}(o) \cap \mathrm{pre}(o)} x_i. \tag{3.2}$$

Eq. 3.1 is a constraint saying that the initial state must have at most one common fact with the fa-mutex and corresponds to the first condition in Definition 3.3 ($|M \cap s_{init}| \leq 1$). Eq. 3.2 corresponds to the second part of the definition and it ensures that the mutex property is preserved by all applicable operators ($|M \cap \mathrm{add}(o)| \leq |M \cap \mathrm{pre}(o) \cap \mathrm{del}(o)|$).

The objective function of the ILP is to maximize $\sum_{f_i \in \mathcal{F}} x_i$. The maximization enforces inference of a fa-mutex counting the maximal number of facts possible.

Unfortunately, the solution to this ILP is only one fa-mutex, so some mechanism enabling inference of all fact-alternating mutexes is required. This drawback can be resolved by solving the ILP repeatedly, each time with added constraints that exclude already inferred fa-mutexes. Let M denote a known fa-mutex. Such a fa-mutex and all its subsets can be excluded from the ILP solution by adding the constraint

$$\sum_{f_i \notin M} x_i \geq 1. \tag{3.3}$$

The constraint forces the ILP solver to add to the solution a fact that is not present in the known fa-mutex M and thus excluding M and all its subsets. In other words, since we are not interested in the fa-mutex M and its subsets, we know that any other fa-mutex must contain a fact that is not part of M.

Algorithm 5: Inference of fact-alternating mutex invariants using ILP.

Input: Planning task $\Pi = \langle \mathcal{F}, \mathcal{O}, s_{init}, s_{goal} \rangle$
Output: A set of fa-mutexes $\mathcal{M}$

```
Initialize ILP with constraints according to Eq. 3.1–3.2;
Set objective function of ILP to maximize ∑_{f_i ∈ F} x_i;
Solve ILP and save the resulting fa-mutex into M;
while |M| ≥ 1 do
    Add M to the output set M;
    Add constraint according to Eq. 3.3 using M;
    Solve ILP and save the resulting fa-mutex into M;
end
```

The whole fa-mutex inferring algorithm is encapsulated in Algorithm 5. First, ILP constraints are constructed according to Eq. 3.1–3.2 which ensures

that the solutions of the ILP will be fact-alternating mutexes. Then in turn a maximal fa-mutex is inferred through ILP solution and consequently removed from future solutions using added constraint corresponding to Eq. 3.3. The cycle continues until inferred fa-mutexes consist of at least one fact and since a maximal fa-mutex is produced at each step, arriving at smaller and smaller fa-mutexes means that the algorithm eventually terminates. The combination of maximization and removing the found fa-mutexes and all theirs subsets from the solutions in the following steps also ensures that every produced fa-mutex is unique and it is never a subset of any already found fa-mutex.

Theorem 3.30. *Algorithm 5 is complete with respect to maximal fact-alternating mutex invariants.*

Proof. To prove Theorem 3.30 by contradiction let us assume that Algorithm 5 was terminated and it produced a set of fact-alternating mutexes, and let us assume that there exists a fact-alternating mutex M that is not a subset of any mutex produced by Algorithm 5. Such a fa-mutex must satisfy constraints expressed by Eq. 3.1–3.2 and it must contain at least one fact that is not part of any fa-mutex produced by Algorithm 5. This is not violated by the ILP constraints in the last cycle of Algorithm 5 because they consist of Eq. 3.1–3.2 and a set of Eq. 3.3 constraints that force the next fa-mutex to include a fact that is not part of any fa-mutex found so far. This means that Algorithm 5 could not terminate thus such M does not exist and Algorithm 5 is complete with respect to maximal fact-alternating mutex invariants. □

Another interesting application of fa-mutexes is an inference of unreachable facts, i.e., the facts that are not part of any reachable state. It follows from Proposition 3.5 that we can slightly modify Algorithm 5 to produce fa-mutexes consisting exclusively from the unreachable facts. If Eq. 3.1 is replaced with a more strict constraint $\sum_{f_i \in s_{init}} x_i = 0$ the resulting fa-mutexes will be those that have empty intersection with the initial state. Therefore the facts they consist of can be safely removed from the planning task because they do not appear in any reachable state.

Moreover, if we run the modified algorithm for detection of the unreachable facts, we remove the inferred unreachable facts from the planning task and afterwards we execute Algorithm 5 for inference of fa-mutexes, we can change Eq. 3.1 in Algorithm 5 to more strict $\sum_{f_i \in s_{init}} x_i = 1$ because the resulting fa-mutexes can be only those that have a common fact with the initial state. Therefore, it still remains complete with respect to the modified planning task without unreachable facts removed in the previous step.

3.5 Inference of Restricted Fact-Alternating Mutex Invariants

The algorithm lied out in Section 3.4 was shown to be able to produce a complete set of all fact-alternating mutex invariants. In this section we will present a polynomial algorithm for inference of restricted fact-alternating mutexes. Such an algorithm, obviously, cannot be complete with respect to

maximal rfa-mutexes. The algorithm is based on inference of two sets for each fact we call a conflict set and a bind set whose definitions follow.

Definition 3.31. A fact $f \in \mathcal{F}$ is **in conflict** with a fact $g \in \mathcal{F}$ iff for every rfa-mutex M such that $f \in M$ it holds that $g \notin M$. The relation is symmetric, i.e., f is in conflict with g iff g is in conflict with f. A **conflict set** $C_f \subseteq \mathcal{F}$ is a set of facts such that for every $g \in C_f$ it holds that f is in conflict with g.

Definition 3.32. A fact $f \in \mathcal{F}$ **binds** a fact $g \in \mathcal{F}$ (or conversely g **is bound by** f) iff for every rfa-mutex M such that $f \in M$ it holds that $g \in M$. A **bind set** $B_f \subseteq \mathcal{F}$ is a set of facts such that for every $g \in B_f$ it holds that f binds g.

A conflict set C_f consists of the facts that cannot be part of any rfa-mutex containing f and a bind set B_f contains the facts that must be included in all rfa-mutexes also containing f if there are any. The algorithm for inference of rfa-mutexes is based on a gradual building of these two sets. We present a set of inference rules in a form of lemmas that can be used for proving whether some fact can be part of some conflict or bind set. And at the end of this section, we join the lemmas into a full inference algorithm. First, we start with simple properties of conflict and bind sets.

Lemma 3.33. *(Reflexivity of a bind relation) For every fact $f \in \mathcal{F}$ it holds that f binds itself.*

Proof. It directly follows from Definition 3.32 if we consider $g = f$. □

Lemma 3.34. *(Transitivity of a bind relation) If f binds g and g binds h then f binds h.*

Proof. If every rfa-mutex containing g also contains h and every rfa-mutex containing f contain also g then every rfa-mutex containing f must also contain h. □

Lemma 3.35. *(Transitivity between bind and conflict sets) If f binds g and g is in conflict with h then f is in conflict with h.*

Proof. If g is part of every rfa-mutex containing also f and h cannot be part of any rfa-mutex containing also g then h cannot be part of any rfa-mutex containing f. □

Lemma 3.36. *If there does not exist a rfa-mutex M such that $f \in M$ then f binds all facts from $\mathcal{F}$.*

Proof. It follows from the use of universal quantifier in Definition 3.32. □

Lemma 3.37. *$f \in \mathcal{F}$ is in conflict with all facts from $\mathcal{F}$ iff there does not exist a rfa-mutex M such that $f \in M$.*

Proof. If f is in conflict with all facts then f is in conflict with itself which is contradiction with $f \in M$. The other direction follows from the use of universal quantifier in Definition 3.31. □

Lemma 3.33 says that every bind set can be initialized with one fact, i.e., for every $f \in \mathcal{F}$ we can set $B_f = \{f\}$. Lemma 3.34 describes a simple inference rule based on a transitive property of bind sets. Similarly Lemma 3.35 can be used for extension of conflict sets through a bind relation. Lemma 3.36

and Lemma 3.37 help us to show that the inference algorithm terminates in a polynomial number of steps because even when we detect that some fact is not part of any rfa-mutex the corresponding conflict and bind sets grow. Thus we will be able to show that as conflict and bind sets grow by at least one fact per cycle, the maximal number of cycles is polynomial in a number of facts in the planning task. Therefore if we show that every cycle consist of only polynomial number of steps the whole algorithm terminates in a polynomial number of steps. The following lemma describes an initialization of conflict sets.

Lemma 3.38. *Let $f, g \in \mathcal{F}$ denote a pair of facts such that $f \neq g$. If*

1. *$f \in s_{init}$ and $g \in s_{init}$, or*
2. *there exists an operator $o \in \mathcal{O}$ such that $f \in add(o)$ and $g \in add(o)$, or*
3. *there exists an operator $o \in \mathcal{O}$ such that $f \in pre(o) \cap del(o)$ and $g \in pre(o) \cap del(o)$*

then f is in conflict with g.

Proof. Definition 3.6 states that every fact-alternating mutex can contain only one fact from the initial state, therefore all facts from the initial state are in conflict with each other. Moreover, any rfa-mutex can contain at most one fact from any add effect, therefore facts that appear in the same add effect must be in conflict with each other. Finally, any rfa-mutex can contain at most one fact from any intersection of precondition and delete effect, therefore facts that appear together in those intersections must be in conflict with each other. □

Lemma 3.38 stems directly from the definition of a rfa-mutex and it describes how conflict sets can be initialized just by checking the initial state and all operators.

Lemma 3.39. *If there exist a conflict set C_f and a bind set B_f such that $C_f \cap B_f \neq \emptyset$ then there is no rfa-mutex M such that $f \in M$ thus f binds all facts and it is also in conflict with all facts.*

Proof. If there exists a fact g such that $g \in C_f$ and $g \in B_f$ then every rfa-mutex containing f must contain g and also cannot contain g at the same time. So it follows that there is no such rfa-mutex. Therefore also $B_f = C_f = \mathcal{F}$ (Lemma 3.36, Lemma 3.37). □

Lemma 3.40. *If there exists an operator $o \in \mathcal{O}$, a bind set B_f, and a conflict set C_f such that $B_f \cap add(o) \neq \emptyset$ and $(pre(o) \cap del(o)) \setminus C_f = \emptyset$ then there does not exist any rfa-mutex M such that $f \in M$ thus f binds all facts and it is also in conflict with all facts.*

Proof. To prove the lemma by contradiction let us assume that there exists a rfa-mutex M such that $f \in M$. Since $B_f \subseteq M$ and $B_f \cap \mathrm{add}(o) \neq \emptyset$ then $|M \cap \mathrm{add}(o)| = 1$ therefore $|M \cap \mathrm{pre}(o) \cap \mathrm{del}(o)| = 1$. Furthermore, since $C_f \cap M = \emptyset$ then also $|M \cap ((\mathrm{pre}(o) \cap \mathrm{del}(o)) \setminus C_f)| = 1$. This is in contradiction with the assumption that $(\mathrm{pre}(o) \cap \mathrm{del}(o)) \setminus C_f = \emptyset$ therefore

M cannot be a rfa-mutex containing f. Therefore also $B_f = C_f = \mathcal{F}$ (Lemma 3.36, Lemma 3.37). □

Lemma 3.39 formally states a simple rule saying that if a fact is in conflict with the same fact as it binds, such a fact cannot be part of any rfa-mutex. Lemma 3.40 formulates a more complicated rule. If there is an operator o that adds some fact about which we know it must be part of all rfa-mutexes containing f, then we also know that $\text{pre}(o) \cap \text{del}(o)$ must contain some other fact from the same rfa-mutex to meet the definition of a rfa-mutex. But if $\text{pre}(o) \cap \text{del}(o)$ contains only the facts about which we know they cannot be part of any rfa-mutex containing f, then it follows that there are not any rfa-mutexes containing f at all. These two rules are used for identification of the facts that cannot be part of any rfa-mutex.

Lemma 3.41. *If there exists an operator $o \in \mathcal{O}$ and a bind set B_f such that $B_f \cap add(o) \neq \emptyset$ and a conflict set C_f and a fact g such that $(pre(o) \cap del(o)) \setminus C_f = \{g\}$ then f binds g.*

Proof. Two cases must be investigated, either:

1. there is no rfa-mutex containing f in which case f binds g according to Lemma 3.36,
2. or there exist some rfa-mutexes containing f. Therefore for every rfa-mutex M such that $f \in M$ must hold that $B_f \subseteq M$. So it follows from Definition 3.6 that if $B_f \cap \text{add}(o) \neq \emptyset$ then $|M \cap \text{add}(o)| = 1$ therefore also $|M \cap \text{pre}(o) \cap \text{del}(o)| = 1$. Finally, since g is the only fact from $\text{pre}(o) \cap \text{del}(o)$ that is not in conflict with f it must follow that g is bound by f. □

Lemma 3.41 corresponds to the rule extending bind sets. The main idea is that if there exists a rfa-mutex containing f then we can check all operators adding some fact about which we know it must be part of that rfa-mutex. If there is only one possible fact from an intersection of preconditions and delete effects that could satisfy a rfa-mutex condition on operators, then such a fact must be part of every rfa-mutex containing f.

Lemma 3.42. *If there exist a conflict set C_f and a bind set B_f and operators $o_1 \in \mathcal{O}$ and $o_2 \in \mathcal{O}$ such that $add(o_1) \cap B_f \neq \emptyset$ and $add(o_2) \cap B_f \neq \emptyset$ and $(pre(o_1) \cap del(o_1)) \setminus C_f \subseteq (pre(o_2) \cap del(o_2)) \setminus C_f$ then all facts from $((pre(o_2) \cap del(o_2)) \setminus C_f) \setminus ((pre(o_1) \cap del(o_1)) \setminus C_f)$ are in conflict with f.*

Proof. If there does not exist any rfa-mutex containing f then f is in conflict with all facts (Lemma 3.37). So we further assume that there exists at least one rfa-mutex containing f. Therefore for every rfa-mutex M containing f holds that $B_f \subseteq M$ and $C_f \cap M = \emptyset$. So it follows from Definition 3.6 that if $B_f \cap \text{add}(o_1) \neq \emptyset$ and $B_f \cap \text{add}(o_2) \neq \emptyset$ then $|M \cap \text{add}(o_1)| = |M \cap \text{add}(o_2)| = 1$ therefore $|M \cap \text{pre}(o_1) \cap \text{del}(o_1)| = |M \cap \text{pre}(o_2) \cap \text{del}(o_2)| = 1$ therefore $|M \cap ((\text{pre}(o_1) \cap \text{del}(o_1)) \setminus C_f)| = |M \cap ((\text{pre}(o_2) \cap \text{del}(o_2)) \setminus C_f)| = 1$. Finally, since $(\text{pre}(o_1) \cap \text{del}(o_1)) \setminus C_f \subseteq (\text{pre}(o_2) \cap \text{del}(o_2)) \setminus C_f$ it must follow that $|M \cap (((\text{pre}(o_2) \cap \text{del}(o_2)) \setminus C_f) \setminus ((\text{pre}(o_1) \cap \text{del}(o_1)) \setminus C_f))| = 0$ which concludes the proof. □

Lemma 3.42 follows a similar reasoning as Lemma 3.40 and Lemma 3.41. If an operator o has one of the facts from B_f as its add effect ($B_f \cap \text{add}(o) \neq \emptyset$) then one of the facts from $(\text{pre}(o) \cap \text{del}(o)) \setminus C_f$ must be also part of every rfa-mutex containing f. If we have two operators o_1, o_2 both having a fact from B_f as their add effect and $(\text{pre}(o_1) \cap \text{del}(o_1)) \setminus C_f$ is subset of $(\text{pre}(o_2) \cap \text{del}(o_2)) \setminus C_f$ then we may easily infer that all facts that are present in $(\text{pre}(o_2) \cap \text{del}(o_2)) \setminus C_f$ but not in $(\text{pre}(o_1) \cap \text{del}(o_1)) \setminus C_f$ cannot be part of any rfa-mutex containing f.

Algorithm 6: Inference of conflict and bind sets

Input: Planning task $\Pi = \langle \mathcal{F}, \mathcal{O}, s_{init}, s_{goal} \rangle$
Output: A set of conflict sets $\mathcal{C} = \{C_f\}_{f \in \mathcal{F}}$, a set of bind sets $\mathcal{B} = \{B_f\}_{f \in \mathcal{F}}$

```
for each f ∈ F do
    Initialize C_f according to Lemma 3.38;
    Initialize B_f = {f} (Lemma 3.33);
end
do
    for each f ∈ F do
        Set C_f = B_f = F if f cannot be part of any rfa-mutex (Lemma
        3.39, 3.40);
        Update B_f and C_f using transitivity rules (Lemma 3.34 and 3.35);
        Update B_f according to Lemma 3.41;
        Update C_f according to Lemma 3.42;
    end
while any C_f or B_f changed;
```

The algorithm for inference of conflict and bind sets is encapsulated in Algorithm 6. The algorithm starts with an initialization of conflict and bind sets using simple rules that check the initial state and all operators which requires only polynomial number of steps. In the main cycle of the algorithm all facts are processed sequentially. First it is checked whether the corresponding fact can be part of any rfa-mutex. This is achieved by checking whether the corresponding bind and conflict sets have empty intersection (Lemma 3.39) and by checking operators according to Lemma 3.40. Both operations require at most polynomial number of steps. Then conflict and bind sets are extended by more facts through a transitive property of bind sets (Lemma 3.34 and 3.35). This can be done also in a polynomial number of steps. Finally, the two most complicated rules are used (Lemma 3.41 and 3.42). Both these operations require checking all operators which can be done also in a polynomial number of steps. The whole algorithm terminates when there is no applicable rule that can be used for extension of some conflict or bind set. In every step of the algorithm, conflict and bind sets can only grow in a number of facts they consist of. Therefore the number of cycles can be at most polynomial in a number of facts. So it follows that the whole algorithm terminates in a polynomial number of steps in a number of facts

Algorithm 7: Inference of restricted fact-alternating mutex invariants using conflict and bind sets.

Input: Planning task $\Pi = \langle \mathcal{F}, \mathcal{O}, s_{init}, s_{goal} \rangle$
Output: A set of mutex invariants $\mathcal{M}$

```
1 for each B_f generated by Algorithm 6 do
2 |   if B_f is a rfa-mutex according to Definition 3.6 then
3 |   |   Add B_f to M;
4 |   end
5 end
```

and operators.

The algorithm for inference of rfa-mutex invariants is encapsulated in Algorithm 7. The algorithm is based on Algorithm 6 that is used for generating bind sets that are in turn verified whether they form rfa-mutexes. The verification must be performed because a correct bind set by itself does not guarantee that the facts from the bind set form a rfa-mutex. A bind set provides just a set of facts that must be part of every rfa-mutex containing the corresponding fact if there are any such rfa-mutexes. But they can be used as a guess for the subsequent verification which runs in a polynomial number steps because only the initial state and each operator must be checked. Therefore Algorithm 7 also terminates in a polynomial number of steps in a number of facts and operators.

Chapter 4

Experimental Results

All algorithms experimentally evaluated in this chapter were implemented[1] into Fast Downward's preprocessor [Helmert, 2006] written in Python programming language. The experiments were run on a computer with an Intel Core i7 3.60GHz processor and 16GB RAM. The algorithms were evaluated on all domains from the optimal deterministic track of the last International Planning Competition 2014 (IPC 2014) [Vallati et al., 2015] that do not contain any conditional effects after grounding (i.e., all except Citycar domain).

The algorithms proposed in this work were compared with two different methods for inferring mutex invariants that were already discussed in Chapter 2. One is the inference algorithm implemented in Fast Downward's preprocessor [Helmert, 2009] that will be from now on abbreviated by `fd`. The main purpose of `fd` is to infer mutex invariants that are consequently used for translation of a planning task from PDDL into Finite Domain Representation (FDR), therefore we compare our algorithms with `fd` also in a context of construction of FDR. The Helmert's algorithm was used by most planners that participated in the last IPC 2014 in the deterministic track. Therefore the comparison with `fd` is certainly relevant to the planning community.

The other state of the art algorithm that we use for comparison is a generalization of $h^{\max}$ heuristic into h^m heuristic [Haslum and Geffner, 2000]. More specifically, h^2 heuristic provides a method for inferring pairs of facts that cannot hold together in any reachable state. Such pairs of facts can be interpreted as mutex invariants because if two facts cannot be both part of the same reachable state, then at most one of them can be part of any reachable state which is exactly the definition of mutex invariant. This reasoning does not apply generally to h^m heuristic because for $m \geq 3$ stating that a set of three of more facts is unreachable does not necessarily mean that at most one of these facts can be part of the same reachable state. For example, if h^3 heuristic provides a set of three facts $\{f_1, f_2, f_3\}$ that is not part of any reachable state then it could be a case that there are reachable states that contain both f_1 and f_2 but not f_3, i.e., the set $\{f_1, f_2, f_3\}$ would not be a mutex invariant. Therefore from the whole family of h^m heuristics only h^2 can be used for inference of mutex invariants. We will refer to this mutex inference algorithm as $\mathtt{h}^2$.

[1] https://github.com/danfis/fast-downward-masters-thesis

The algorithm for inference of fa-mutex invariants (Algorithm 5) is implemented using CPLEX ILP solver (v12.6.1.0) running with default configuration in one thread. We will refer to this algorithm as `fa` and the algorithm for inference of rfa-mutexes (Algorithm 7) will be denoted by `rfa`.

The algorithm for extending a set of mutexes (Algorithm 4) needs a set of already inferred mutexes as its input. The algorithm will be denoted by `E[x]` where `x` is a name of the algorithm that provides the input mutexes. So for example `E[fa]` will refer to Algorithm 4 running on mutexes inferred by Algorithm 5 and `E[h`^{2}`]` to the same algorithm running on mutexes inferred by `h`^{2} as its input.

The presented algorithms (`fd`, `h`^{2}, `fa`, `rfa`, `E[fd]`, `E[h`^{2}`]`, `E[fa]`, and `E[rfa]`) are experimentally evaluated in three different ways. First, the algorithms are compared in terms of pair mutexes (Section 4.1). From Proposition 3.23 it follows that any mutex can be decomposed into a set of mutexes each having two facts. The decomposition allows us to compare the algorithms without considering differences in shapes and sizes of mutexes inferred by particular methods. In Section 4.2 the shapes and sizes of inferred mutex invariants will be taken into account and we will provide a discussion of reasons why the shapes and sizes of mutex invariants matter. The third way of comparing algorithms (Section 4.3) will consider translation into FDR as an example of application of mutex invariants. Finally, we will conclude this chapter by discussing the challenges coming from this work that should be addressed in a future work (Section 4.4).

4.1 Comparison in Terms of Pair Mutexes

As stated in Proposition 3.23, any mutex invariant can be decomposed into a set of pair mutexes by enumerating all pairs of facts from which the original mutex consists of. Such a decomposition provides a common ground for comparing algorithms in terms of sizes of the inferred mutexes. For example `h`^{2} is able to produce only pair mutexes, but `fa` is designed to produce maximal fa-mutexes. The pair decomposition provides a transparent method for comparing these two and all other algorithms for inference of mutex invariants.

On the other hand, this method of comparison clouds the fact that `fd`, `fa`, and `rfa` all provide a richer structure than just a set of pair mutexes. Although it is always possible to reconstruct back any mutex from its pair decomposition by using some algorithm for enumerating all maximal cliques in a graph [Bron and Kerbosch, 1973] (this was already discussed in Section 3.2), it must be taken into account that the reconstruction alone is NP-Hard and it can generate an exponential number of mutexes (i.e., possibly many more besides the original ones that were used for the decomposition). The significance of having richer mutex sets than just pairs of facts is discussed in more depth in Section 4.2 and 4.3.

The algorithms were evaluated on domains from the optimal deterministic track of IPC 2014 and all inferred mutexes were decomposed into pair mutexes.

Table 4.1: Sum of number of inferred pair mutexes.

domain	#ps	fd	h^2	fa	rfa	E[fd]	E[h^2]	E[fa]	E[rfa]
barman	20	1 792	**13 345**	11 645	7 901	1 792 (+0)	13 345 (+0)	13 273 (+1 628)	9 034 (+1 133)
cavediving	20	6 304	**67 847**	61 614	58 354	13 442 (+7 138)	67 847 (+0)	64 391 (+2 777)	64 391 (+6 037)
childsnack	20	**3 194**	**3 194**	**3 194**	324	3 194 (+0)	3 194 (+0)	3 194 (+0)	**3 194** (+2 870)
floortile	20	**17 572**	**17 572**	**17 572**	40	17 572 (+0)	17 572 (+0)	17 572 (+0)	40 (+0)
ged	20	48 452	**69 564**	68 326	125	50 061 (+1 609)	69 564 (+0)	**69 564** (+1 238)	496 (+371)
hiking	20	**2 505**	**2 505**	**2 505**	**2 505**	2 505 (+0)	2 505 (+0)	2 505 (+0)	2 505 (+0)
maintenance	20	0	**1 039**	**1 039**	0	0 (+0)	1 039 (+0)	1 039 (+0)	0 (+0)
openstacks	20	16 085	**21 340**	16 085	16 085	**21 340** (+5 255)	21 340 (+0)	**21 340** (+5 255)	**21 340** (+5 255)
parking	20	178 720	**260 880**	178 720	26 520	181 200 (+2 480)	260 880 (+0)	181 200 (+2 480)	26 520 (+0)
tetris	20	22 552	**10 997 274**	3 968 290	2 673 010	22 552 (+0)	10 997 274 (+0)	5 225 266 (+1 256 976)	2 673 010 (+0)
tidybot	20	240	**133 744**	**133 744**	**133 744**	240 (+0)	133 744 (+0)	133 744 (+0)	133 744 (+0)
transport	20	**114 168**	**114 168**	**114 168**	**114 168**	114 168 (+0)	114 168 (+0)	114 168 (+0)	114 168 (+0)
visitall	20	**227 268**	**227 268**	**227 268**	**227 268**	227 268 (+0)	227 268 (+0)	227 268 (+0)	227 268 (+0)
Σ	260	638 852	11 929 740	4 804 170	3 260 044	655 334 (+16 482)	11 929 740 (+0)	6 074 524 (+1 270 354)	3 275 710 (+15 666)

Table 4.2: Number of inferred pair mutexes in selected problems and number of all pair mutexes those problems contain.

domain	problem	all	fd	h^2	fa	rfa	E[fd]	E[h^2]	E[fa]	E[rfa]
barman	p433.1	336	48	**320**	265	177	48 (+0)	320 (+0)	317 (+52)	209 (+32)
ged	d-1-2	928	463	**607**	595	3	478 (+15)	607 (+0)	**607** (+12)	6 (+3)
hiking	ptesting-1-2-3	**19**	**19**	**19**	**19**	**19**	19 (+0)	19 (+0)	19 (+0)	19 (+0)
maintenance	1.3.010.010.1-001	**15**	0	**15**	**15**	0	0 (+0)	15 (+0)	15 (+0)	0 (+0)
tetris	p01-4	75 694	240	**75 626**	40 192	24 320	240 (+0)	75 626 (+0)	45 120 (+4 928)	24 320 (+0)
transport	p01	260	**124**	**124**	**124**	**124**	124 (+0)	124 (+0)	124 (+0)	124 (+0)
visitall	p-1-5	**300**	**300**	**300**	**300**	**300**	300 (+0)	300 (+0)	300 (+0)	300 (+0)

Table 4.3: Ratio of number of inferred pair mutexes.

domain	fd	h^2	fa	rfa	E[fd]	E[fa]	E[rfa]
barman	0.13	1.00	0.87	0.59	0.13	0.99	0.68
cavediving	0.09	1.00	0.91	0.86	0.20	0.95	0.95
childsnack	1.00	1.00	1.00	0.10	1.00	1.00	1.00
floortile	1.00	1.00	1.00	0.00	1.00	1.00	0.00
ged	0.70	1.00	0.98	0.00	0.72	1.00	0.01
hiking	1.00	1.00	1.00	1.00	1.00	1.00	1.00
maintenance	0.00	1.00	1.00	0.00	0.00	1.00	0.00
openstacks	0.75	1.00	0.75	0.75	1.00	1.00	1.00
parking	0.69	1.00	0.69	0.10	0.69	0.69	0.10
tetris	0.00	1.00	0.36	0.24	0.00	0.48	0.24
tidybot	0.00	1.00	1.00	1.00	0.00	1.00	1.00
transport	1.00	1.00	1.00	1.00	1.00	1.00	1.00
visitall	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Mean	0.57	1.00	0.89	0.51	0.60	0.93	0.61

Table 4.1 shows sums of number of inferred pair mutexes per domain and overall by all tested algorithms, the maximum numbers are highlighted. The results for `E[·]` (Algorithm 4) contain an additional information in parenthesis that show how many additional pair mutexes were inferred on top of those that were taken as its input. So, for example in `openstacks` domain `fd` inferred 16 085 pair mutexes and `E[fd]` 21 340 pair mutexes which means that `E[fd]` inferred 5255 additional pair mutexes because the output of `fd` was used as the input of `E[fd]`.

Table 4.3 shows the same results, but as ratios to the maximum number of inferred pair mutexes. The last row labeled as "Mean" shows arithmetic means of the values above it. In Fig. A.1, A.2, and A.3 (Appendix A) there are depicted comparisons between selected combinations of tested algorithms as scatter plots with logarithmic scales and with added zero. Each point in the scatter plots corresponds to one problem from the data set.

Table 4.4 shows sums of running times per domain and overall for each tested algorithm. The results for `E[·]` are listed with additional number in parenthesis expressing amount of time spent solely in Algorithm 4, i.e., without the time spent by the algorithm that produced the input for `E[·]`. Table 4.5 contains minimum and maximum running times per problem within each domain and over all problems within data set. In Fig. A.4, A.5, and A.6 are depicted comparisons of running times as scatter plots with logarithmic scales. Each point in the scatter plots corresponds to one problem.

Additionally, we managed to perform an exhaustive search of all reachable states on some problems from the data set, which enabled us to make a complete list of all mutex invariants of these problems and compare them with the results of the tested algorithms. The results are listed in Table 4.2 where the column labeled as "all" contains number of all pair mutexes in the particular problem. The maximum numbers of inferred pair mutexes are highlighted and the numbers in column "all" are highlighted if some algorithm managed to infer all pair mutexes in the corresponding problem.

In terms of a number of inferred pair mutexes, the poorest performance has

Table 4.4: Sum of running times in seconds of inference algorithms.

domain	#ps	fd	h^2	fa	rfa	E[fd]	E[h^2]	E[fa]	E[rfa]
barman	20	**0.23**	25.44	6.38	3.60	3.42 (+3.20)	28.97 (+3.54)	10.89 (+4.51)	7.95 (+4.36)
cavediving	20	**0.19**	380.15	137.48	77.04	134.42 (+134.23)	429.94 (+49.79)	194.43 (+56.96)	225.15 (+148.11)
childsnack	20	**0.10**	53.52	29.72	327.25	7.81 (+7.71)	61.22 (+7.69)	37.37 (+7.65)	338.90 (+11.65)
floortile	20	**0.17**	7.18	3.27	1.09	2.44 (+2.27)	9.23 (+2.05)	5.45 (+2.18)	2.65 (+1.56)
ged	20	3.23	31.68	39.26	**2.59**	11.59 (+8.36)	39.88 (+8.20)	48.36 (+9.09)	7.66 (+5.07)
hiking	20	**0.17**	55.91	4.03	13.39	6.43 (+6.27)	61.78 (+5.87)	10.25 (+6.22)	19.68 (+6.29)
maintenance	20	**0.01**	0.47	2.81	0.16	0.02 (+0.00)	0.84 (+0.37)	3.13 (+0.31)	0.16 (+0.00)
openstacks	20	**0.34**	122.99	23.07	22.45	16.00 (+15.66)	135.84 (+12.85)	38.69 (+15.62)	38.21 (+15.76)
parking	20	**0.15**	1 181.00	177.70	76.19	157.76 (+157.61)	1 332.08 (+151.08)	334.40 (+156.70)	193.42 (+117.24)
tetris	20	**0.64**	76.15	343.12	5 537.37	384.51 (+383.87)	5 112.94 (+5 036.79)	2 772.73 (+2 429.61)	7 173.39 (+1 636.01)
tidybot	20	**1.17**	8 461.57	43.89	492.31	243.12 (+241.95)	8 722.14 (+260.57)	303.14 (+259.25)	752.27 (+259.97)
transport	20	**0.12**	380.71	10.95	136.45	52.05 (+51.93)	431.43 (+50.72)	62.59 (+51.65)	188.01 (+51.56)
visitall	20	**0.03**	55.29	0.62	125.45	38.08 (+38.05)	92.56 (+37.27)	37.84 (+37.22)	161.05 (+35.60)
Σ	260	6.54	10 832.06	822.29	6 815.34	1 057.65 (+1 051.11)	16 458.85 (+5 626.79)	3 859.26 (+3 036.97)	9 108.51 (+2 293.17)

Table 4.5: Minimal and maximal running times in seconds of inference algorithms.

domain	fd		h^2		fa		rfa		E[fd]		E[h^2]		E[fa]		E[rfa]	
	min	max	min	max	min	max	min	max	min	max	min	max	min	max	min	max
barman	0.01	0.01	0.40	2.83	0.12	0.65	0.07	0.35	0.08	0.32	0.46	3.19	0.21	1.12	0.16	0.79
cavediving	0.01	0.02	0.31	78.78	0.17	28.99	0.08	15.84	0.09	31.34	0.37	88.79	0.24	40.68	0.18	49.85
childsnack	0.00	0.01	0.21	7.46	0.16	4.23	0.16	66.57	0.04	1.01	0.24	8.53	0.20	5.24	0.21	68.42
floortile	0.01	0.01	0.11	0.71	0.06	0.32	0.02	0.10	0.04	0.24	0.15	0.92	0.09	0.53	0.05	0.24
ged	0.14	0.19	0.09	3.98	0.25	4.70	0.01	0.30	0.19	1.18	0.11	4.97	0.28	5.84	0.03	0.90
hiking	0.01	0.01	0.03	10.27	0.01	0.61	0.01	2.42	0.01	1.12	0.03	11.28	0.02	1.72	0.02	3.50
maintenance	0.00	0.00	0.00	0.07	0.01	0.90	0.00	0.02	0.00	0.00	0.00	0.11	0.01	0.94	0.00	0.02
openstacks	0.01	0.04	0.78	16.69	0.18	2.73	0.19	2.52	0.14	1.91	0.88	18.21	0.31	4.60	0.32	4.30
parking	0.01	0.01	10.91	137.70	2.40	19.15	0.78	9.94	1.49	18.39	12.30	154.86	3.88	37.12	1.88	23.49
tetris	0.01	0.08	0.24	15.12	0.86	65.41	6.15	1150.97	0.07	91.41	2.36	1218.94	2.43	617.44	7.58	1513.90
tidybot	0.04	0.08	193.02	825.08	1.30	3.58	8.98	47.02	6.16	21.56	200.80	848.37	7.87	27.96	15.75	69.54
transport	0.00	0.01	0.15	74.24	0.04	1.87	0.05	28.77	0.03	10.66	0.17	84.80	0.07	12.67	0.07	39.98
visitall	0.00	0.00	0.03	15.21	0.01	0.09	0.03	51.14	0.01	11.88	0.04	26.84	0.02	11.67	0.04	60.17
Overall	0.00	0.19	0.00	825.08	0.01	65.41	0.00	1150.97	0.00	91.41	0.00	1218.94	0.01	617.44	0.00	1513.90

demonstrated algorithm `fd` with 638 852 inferred pair mutexes. The highest number of pair mutexes was inferred by $\mathtt{h}^2$ (11 929 740) and `fa` and `rfa` inferred overall 4 804 170 and 3 260 044 pair mutexes, respectively. This seems as a huge lead for $\mathtt{h}^2$ before all other algorithms, but careful investigation of Table 4.1 shows us that the most of the margin is disproportionately made in a single domain `tetris`. The reason for this is that domain `tetris` models a grid of positions and several objects of different shapes that can occupy the grid. After grounding, the problems from domain `tetris` contain a high number of facts that have strong restrictions on their co-occurrence in the grid therefore they contain a high number of mutex invariants. In this particular case, $\mathtt{h}^2$ is much more effective in inferring these mutexes than any other algorithm, but `fa` is still much more effective than the widely used algorithm `fd`. Without domain `tetris` the numbers are 932 466, 835 880, 616 300, and 587 034 for $\mathtt{h}^2$, `fa`, `fd`, and `rfa` respectively. This shows that $\mathtt{h}^2$ has still the best performance, but with much smaller margin from our algorithm `fa`. This result is even better reflected in Table 4.3. The algorithm `fa` has very similar results (in terms of number of pair mutexes) as $\mathtt{h}^2$ in most domains except aforementioned `tetris` and also in `parking` and `openstacks` (although the difference is much smaller).

Even though in overall numbers `rfa` is better than `fd`, without domain `tetris` the algorithm `rfa` is outperformed by `fd`. The number of additional pair mutexes inferred by the algorithm `E[·]` is not high. The algorithm had no effect what so ever with $\mathtt{h}^2$ (so $\mathtt{E[h}^2\mathtt{]}$ will not be considered anymore) and in six out of thirteen domains the algorithm was not able to produce a single additional pair mutex for any input algorithm. The biggest gain was recorded for `E[fa]` but it, again, was almost entirely only for domain `tetris`. Nevertheless, it provides some gain: in domains `cavediving`, `childsnack`, and `openstacks` it equals results from `fa` and `rfa`; in domain `cavediving` it more than doubles the number of inferred pair mutexes for `fd`; and in `childsnack`, `ged`, and `openstacks` it even equals the results with $\mathtt{h}^2$.

The algorithm $\mathtt{h}^2$ demonstrates the best performance in all domains, but in seven (out of thirteen) domains the results are the same as of `fa` and they are also the same in two additional domains if we consider `E[fa]`. `fd` achieves the same results as $\mathtt{h}^2$ in six domains (seven in case of `E[fd]`) and rfa in four domains (six in case of `E[rfa]`). Moreover, in the scatter plots comparing number of inferred pair mutexes between $\mathtt{h}^2$ and other algorithms in Fig. A.1 it can be seen that `fa` has almost the same results as $\mathtt{h}^2$ in most problems and the difference between these two algorithms makes just few problems. The scatter plots comparing `fd` with other algorithms (Fig. A.2) show that `fa` is strictly better than `fd` and `rfa` has comparable results with `fd`. Nevertheless, we still should bare in mind that $\mathtt{h}^2$ infers only pair mutexes, but `fd`, `fa`, and `rfa` are able to infer bigger mutexes that can provide more useful information.

We have also analyzed the inferred pair mutexes in pursuit of finding whether the results from different algorithms can be combined or if there is an algorithm that generates a superset of some other algorithm. The results are summarized in a schematic depiction of the outputs of the algorithms

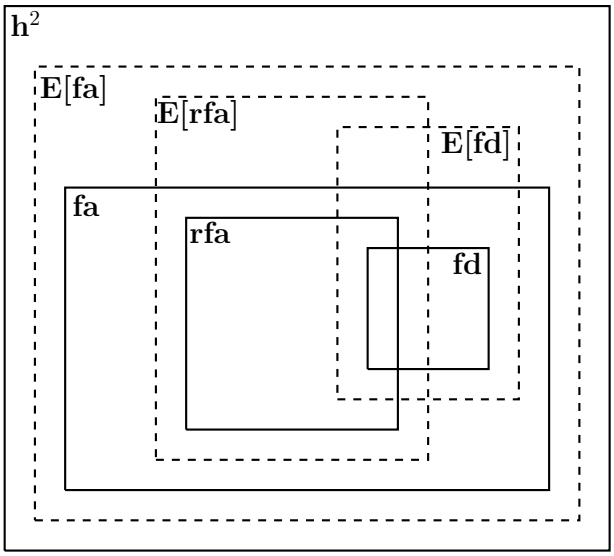


Figure 4.1: Schematic depiction of coverage of inferred pair mutexes by each inference algorithm.

as a set diagram (Fig. 4.1). The diagram shows that $\texttt{h}^2$ inferred in all cases a superset of all other algorithms and `fa` inferred a superset of `rfa` (which we have already formally proven) and `fd`. On the other hand, `rfa` and `fd` infer complementary sets of pair mutexes, so these two algorithms could be combined to increase a number of inferred pair mutexes. Similarly, `E[fa]` generates a superset of both `E[rfa]` and `E[fd]`, and `E[rfa]` and `E[fd]` generate complementary sets that can be reasonably combined.

Considering running times of the algorithms brings a different perspective to the results. As can be seen in Table 4.4 and Table 4.5, `fd` is several orders of magnitude faster than any other tested algorithm. The running time of `fd` never exceeds 200 milliseconds and the sum of running times over all problems in the data set is only 6.54 seconds. On the other hand, the slowest algorithm is $\texttt{h}^2$ that spent about 3 hours (10 832.06 seconds) on solving all problems from the data set, the maximum time spent on a single problem was more than 13.5 minutes (825.8 seconds). The main reason for such high numbers was a single domain `tidybot`, the overall running time of $\texttt{h}^2$ without considering `tidybot` was more than 39 minutes (2370.5 seconds) and the maximum of almost 2.3 minutes (137.7 seconds). The second slowest algorithm is `rfa` having the overall running time of more than 1.8 hours (6815.34 seconds) and the maximum being almost 20 minutes (1150.97 seconds). A disproportionate amount of time was in this case spent in `tetris` domain, the overall running time without `tetris` domain was 21.3 minutes (1278 seconds) and the maximum was 66.57 seconds. The overall running time of `fa` was more than 13 minutes (822.29 seconds) of which almost half of the time was spent in `tetris` domain. The maximum running time of `fa` was 65.41 seconds (without considering most time consuming domain `tetris` the maximum was only 28.99 seconds).

A surprising result is that `rfa` is so much slower than `fa` even though `rfa` is a polynomial algorithm. We think that the main reason for this is that the actual implementation of `rfa` does not use any optimization techniques, but it follows the description given in Section 3.5 which has an asymptotic complexity of high order polynomial. The second reason is that `rfa` is completely implemented in Python whereas `fa` calls CPLEX library that is implemented in C which should be also taken into account. Nevertheless, we believe that `rfa` can be implemented more efficiently which we should address in a future work.

Another interesting comparison in terms of a running time is between $\mathtt{h}^2$ and `fa`. $\mathtt{h}^2$ is more than ten times slower that `fa` both in the overall time and the maximum time (also clearly demonstrated in the scatter plot in Fig. A.4). Considering the similar results in terms of inferred pair mutexes, `fa` can still be contemplated as a viable replacement for $\mathtt{h}^2$ taking into account that `fa` is able to produce a richer set of mutexes than just a set of pair mutexes. But it also should be noted that $\mathtt{h}^2$ was implemented entirely in Python so it is reasonable to expect that at least a part of a difference in a running time between these two algorithms can be diminished by implementing them in C instead of Python.

Nevertheless, `fa` can be easily altered to an any-time algorithm just by setting a limit on a number of the cycles or by prematurely stopping the computation, because `fa` produces one correct fa-mutex per cycle. Whereas $\mathtt{h}^2$ must always run until the whole reachability graph is explored because if it is stopped prematurely the resulting mutexes could be incorrect ($\mathtt{h}^2$ must explore the whole reachability graph to prove that a pair of facts is really a mutex). Moreover, `fa` can be easily altered to provide mutex invariants of some specific shapes and sizes simply by putting more constraints into the ILP formulation that enables it.

The problems for which we were able to discover a complete list of pair mutexes is listed in Table 4.2. The table shows that `fd` and `rfa` were able to infer all pair mutexes for two out of seven problems. $\mathtt{h}^2$ and `fa` were more successful because they managed to infer a complete set of pair mutexes for three problems. Moreover, Table 4.2 confirms the previous results as it shows that $\mathtt{h}^2$ and `fa` are able to achieve similar results with exception in domain `tetris`. Considering that $\mathtt{h}^2$ is a polynomial algorithm and that inferring all pair mutexes is PSPACE-Complete, it is a rather surprising result that $\mathtt{h}^2$ was able to infer more than 99% of all pair mutexes over all of seven problems and 75% without considering the problem from domain `tetris` (`fa` inferred about 53% of all pair mutexes overall and almost 71% without the problem from `tetris`). This result motivates us in pursuing this method of analysis in a future work. It would be interesting to find out how do the pair mutexes not found by $\mathtt{h}^2$ look like, what are their structure and what is exactly the reason they were not discovered by $\mathtt{h}^2$ (or `fa`). This type of analysis could lead to new methods of inference of mutex invariants and possibly to new heuristic functions that would be somehow complementary to h^2 heuristics.

Table 4.6: Number of inferred mutex invariants.

domain	#ps	fd	fa	rfa	fa≻fd	fa≻rfa	rfa≻fd	fd≻rfa
barman	20	206	**498**	372	20	20	20	0
cavediving	20	184	**800**	492	20	20	20	0
childsnack	20	**618**	**618**	54	0	20	0	20
floortile	20	**575**	**575**	40	0	20	0	20
ged	20	**555**	**555**	125	20	20	0	20
hiking	20	**229**	**229**	**229**	0	0	0	0
maintenance	20	0	**460**	0	20	20	0	0
openstacks	20	**730**	**730**	**730**	0	0	0	0
parking	20	**820**	**820**	180	0	20	0	20
tetris	20	52	**676**	116	20	20	20	0
tidybot	20	60	**200**	**200**	20	0	20	0
transport	20	**206**	**206**	**206**	0	0	0	0
visitall	20	**20**	**20**	**20**	0	0	0	0
Σ	260	4255	6387	2764	120	160	80	80

4.2 Comparison of Inferred Mutexes

In the previous section, we provided an analysis of the algorithms for inference of mutex invariants in terms of pair mutexes that were obtained by decomposition of the actually inferred mutexes. As mentioned before, this type of analysis disregards the fact that the mutexes consisting of more than two facts can provide more useful information than mutexes formed just by a pair of facts. A translation from PDDL to FDR is one of the applications for which larger mutexes are desirable as we demonstrate in the next section. Another possible applications were discussed in Section 3.1 and there are possibly more to be discovered given the tight relation between satisfiability of planning tasks and inference of maximum mutexes as described in Section 3.2.2.

The algorithms $\mathtt{h}^2$ and `E[·]` infer only pair mutexes. We have already shown that given a set of pair mutexes, larger mutexes can be constructed from such a set using algorithms for enumerating maximal cliques in a graph. But this approach comes with its price because enumerating all maximal cliques in a graph is in itself NP-Hard (and discovering just one maximum clique is NP-Complete). In the data set we used, the construction of larger mutexes from pair mutexes adds a considerable computational burden because even though in some cases it took just a few seconds in other cases the construction of larger mutexes took more than five hours.

For this reasons, we have decided that in this section we compare only `fd`, `fa` and `rfa` because these algorithms are able to directly infer mutexes consisting of more than two facts and the comparison in terms of pair mutexes was already provided in the previous section.

In Table 4.6 we provide a number of inferred mutexes per domain and overall. The columns labeled as `a`≻`b` show a number of problems in which algorithm `a` generated a richer set of mutexes than `b`, i.e., a number of problems in which `a` inferred more mutexes than `b` or the number of inferred mutexes was the same but some mutex inferred by `a` was a proper superset of some mutex inferred by `b`. The latter happened only in a domain `ged`

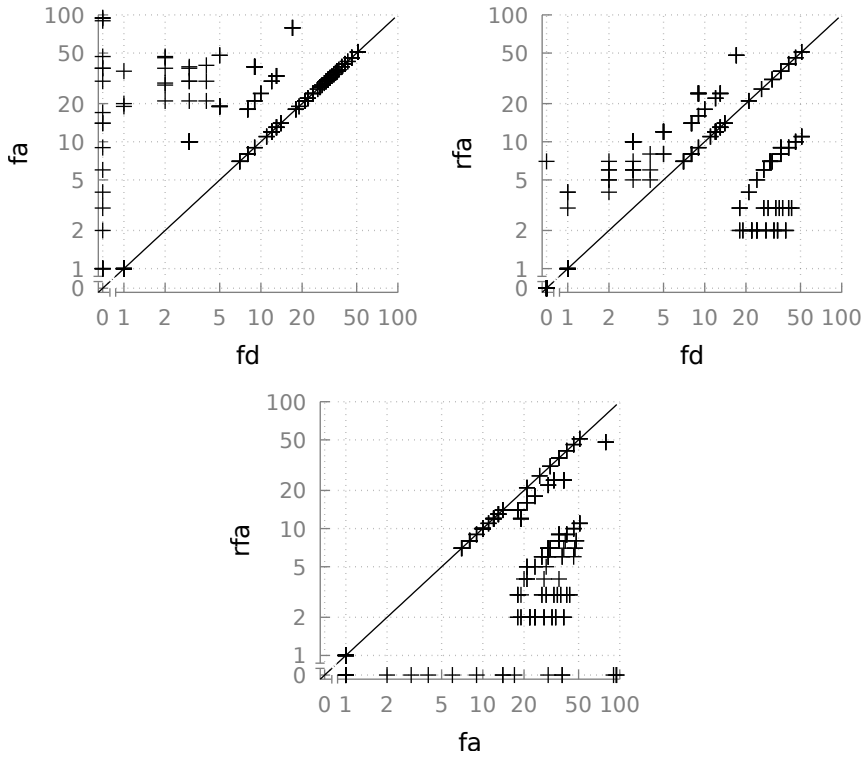


Figure 4.2: Number of inferred mutex invariants as scatter plots with logarithmic scales and added zero.

where `fa` and `fd` generated the same number of mutexes in every problem, but the mutexes inferred by `fa` contained more facts. In all other domains none of the algorithms generated a mutex that was a proper subset of any other mutex generated by any other algorithm.

The highest number of mutexes was generated by `fa` in all tested domains. `fd` generated less mutexes than `fa` in five domains (out of thirteen), overall number of mutexes inferred by `fd` approached only two thirds of the number of mutexes inferred by `fa`, and `fa` generated more mutexes than `fd` in 120 problems out of 260. Table 4.6 also shows that `fa` had the best performance not only in all domains, but also in every tested problem as it is also shown in scatter plots in Fig. 4.2. `rfa` was dominated by `fa` in 160 problems.

`rfa` achieved better results than `fd` in four domains and 80 problems and `fd` was better than `rfa` also in four domains and 80 problems. In overall numbers `fd` infers about 1.5 times more mutexes than `rfa`. In terms of inferred mutexes, `fd` and `rfa` are complementing each other (see also corresponding scatter plot in Fig. 4.2), but there is a huge difference in running times of those two algorithms which was already discussed in the previous section.

Considering that `rfa` always produce a subset of `fa` and that `rfa` is more than ten times slower, `fa` should always take precedence over `rfa`. This conclusion could be changed only if `rfa` was made available in much faster implementation (which we believe is possible).

Table 4.7: Number of variables in FDR.

domain	#ps	fd	fa	rfa	fa≻fd	fa≻rfa	rfa≻fd	fd≻rfa
barman	20	2210	**581**	971	20	20	20	0
cavediving	20	3726	**913**	**913**	20	0	20	0
childsnack	20	**1248**	**1248**	2318	0	20	0	20
floortile	20	**575**	**575**	2515	0	20	0	20
ged	20	**330**	**330**	3019	0	20	0	20
hiking	20	**229**	**229**	**229**	0	0	0	0
maintenance	20	1285	**536**	1285	20	20	0	0
openstacks	20	**1440**	**1440**	**1440**	0	0	0	0
parking	20	**1140**	**1140**	6100	0	20	0	20
tetris	20	16672	**676**	**676**	20	0	20	0
tidybot	20	7472	**3514**	**3514**	20	0	20	0
transport	20	**206**	**206**	**206**	0	0	0	0
visitall	20	**2434**	**2434**	**2434**	0	0	0	0
Σ	260	38 967	13 822	25 620	100	120	80	80

The comparison between `fa` and `fd` shows that `fa` generates a richer set of mutexes than `fd` in all cases, but `fa` is much slower than `fd`. On the other hand, the maximum amount of time that `fa` spent on a single problem was just 65.41 seconds which means that `fa` should be considered as a replacement for `fd` whenever it is allowed by time limitations posed on solving the corresponding task.

4.3 Translation to Finite Domain Representation

One of the applications of mutexes is a translation of planning tasks from PDDL to Finite Domain Representation (FDR) [Helmert, 2009]. A state in FDR (without axioms) is represented by a full assignment over all variables of the planning task, so the translation process must (besides other things) create a set of variables in such a way that all states that were reachable in the PDDL formulation are also reachable and expressible in the FDR formulation. This is the place where mutex invariants take an important part. Each mutex created from facts of a grounded PDDL consists of facts from which at most one can be part of any reachable state. This means that variables of FDR can be created directly from those mutexes that cover all facts. A special value "none of those" can be added to some variables if it is required to cover a situation where none of the facts from the corresponding mutex invariant is present in a state.

In this section, we evaluate `fd`, `fa` and `rfa` algorithms in context of translation from PDDL to FDR by Fast Downward's translator preprocessor. The results are presented in Table 4.7 which contains a number of variables created by the translation process utilizing different algorithms for mutex inference. The lower number the better because a smaller amount of variables usually allows a more compact representation requiring less memory for storing the reached states. The columns labeled as `a`≻`b` show a number of problems in which the translator with the algorithm `a` generated less variables than with `b` (i.e., a number of problems in which `a` performed better than `b`).

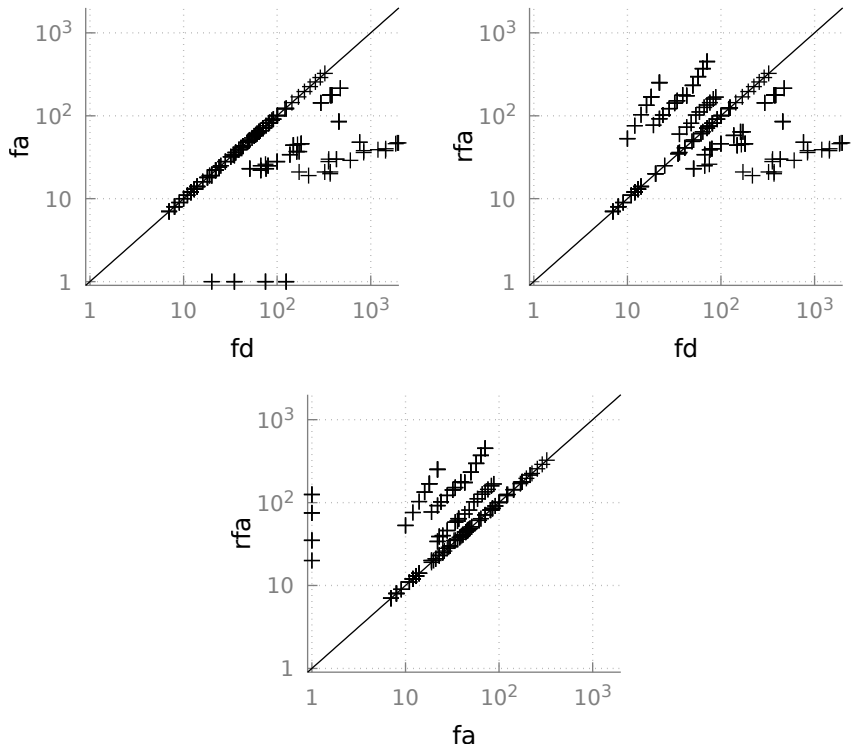


Figure 4.3: Number of variables in FDR as scatter plots with logarithmic scales.

The translator with `fa` created the lowest number of variables in all problems as it is also clearly shown in scatter plots in Fig. 4.3. Overall, the translator with `fd` created almost three times more variables than with `fa` and almost twice as much than with `rfa`. The translator with `fa` provides better variable encoding than with `fd` in 100 problems (out of 260) and in 120 problems the results are better than `rfa`. These results show how dramatic change in a number of created variables can provide a stronger algorithm for mutex inference.

Comparison between `fd` and `rfa` shows these two methods can complement each other because `fd` provides better results than `rfa` for 80 problems and `rfa` is better than `fd` also in 80 problems. The scatter plot comparing `fd` and `rfa` in Fig. 4.3 show that `fd` tend to generate more variables than `rfa` overall which is the reason why after summing results from all domains, the translator with `rfa` generates only two thirds of a number of variables than with `fd`.

The running times in seconds of the whole translation process (not just the mutex inference) are listed in Table 4.8. The table clearly shows that `rfa` is the slowest variant, which was expected given the poor running times of `rfa` algorithm (Section 4.2). But note that a disproportionately big amount of time is spent in a single domain `tetris`.

The fastest translator utilizes `fd` which was also expected given the running times of `fd` alone that were under a second for every problem in the data

Table 4.8: Running times in seconds of a whole translation process.

domain	fd			fa			rfa		
	sum	**min**	**max**	**sum**	**min**	**max**	**sum**	**min**	**max**
barman	**2.94**	0.10	0.21	8.90	0.20	0.84	6.08	0.15	0.53
cavediving	**14.82**	0.09	2.46	151.52	0.25	31.33	90.65	0.16	18.14
childsnack	**7.00**	0.06	0.81	36.55	0.22	5.03	334.50	0.23	67.41
floortile	**1.24**	0.04	0.09	4.32	0.09	0.40	2.25	0.06	0.19
ged	5.70	0.19	0.39	41.73	0.29	4.92	**5.42**	0.05	0.56
hiking	**11.19**	0.04	1.68	15.08	0.04	2.29	24.41	0.04	4.09
maintenance	**0.58**	0.01	0.07	3.34	0.01	0.95	0.72	0.01	0.09
openstacks	**8.84**	0.13	0.91	31.52	0.31	3.59	30.89	0.31	3.36
parking	**34.10**	0.62	3.13	211.60	3.02	22.24	111.62	1.45	13.14
tetris	**71.51**	0.72	9.37	392.17	1.36	71.74	5604.18	6.83	1159.70
tidybot	**162.64**	5.16	12.12	191.38	5.97	14.93	640.48	13.65	58.48
transport	**11.29**	0.05	1.55	22.07	0.09	3.39	147.61	0.09	30.31
visitall	**1.32**	0.01	0.19	1.91	0.02	0.28	126.77	0.04	51.33
Overall	333.18	0.01	12.12	1112.07	0.01	71.74	7125.59	0.01	1159.70

set. More surprising result is that the translator with `fa` is only about 3.4 times slower than with `fd` even though `fa` is more than 100 times slower than `fd`. If we compare the results from Table 4.4 and Table 4.8 it becomes clear that the parts of the translation process other than the inference of mutexes combined are about 50 times slower than `fd` which explains the smaller difference of running times in the case of utilization of `fd` and `fa` in the translation process.

Scatter plots in Fig. 4.4 provide more detailed view on the running times. The scatter plot comparing `fd` and `rfa` shows the translator with `rfa` is faster than the translator utilizing `fd` only in a few problems, but an overwhelming majority of problems was translated faster with `fd` (in many case more than ten times faster). The translator with `fa` is slower than the translator with `fd` in all tested problems, in some cases more than ten times slower.

Considering overall numbers, the translator with `fa` generates about three times less variables and runs about three times slower than the translator utilizing `fd`, which seems to be a reasonable trade off between the number of created variables and the running time of the translation process. The maximum amount of time given to optimal and satisficing planners in the deterministic track of IPC 2014 was 30 minutes. The maximum time the translator utilizing `fa` spent on a single problem was 71.74 seconds which means that the competition planners utilizing `fa` instead of `fd` would still be left with at least little less than 29 minutes for solving the problem, i.e., in the worst case the translation utilizing `fa` would consume only 1 minute out of 30 more than the original translation process utilizing `fd`. For these reasons, `fa` should be considered as a replacement for `fd` in the case of a translation from PDDL to FDR.

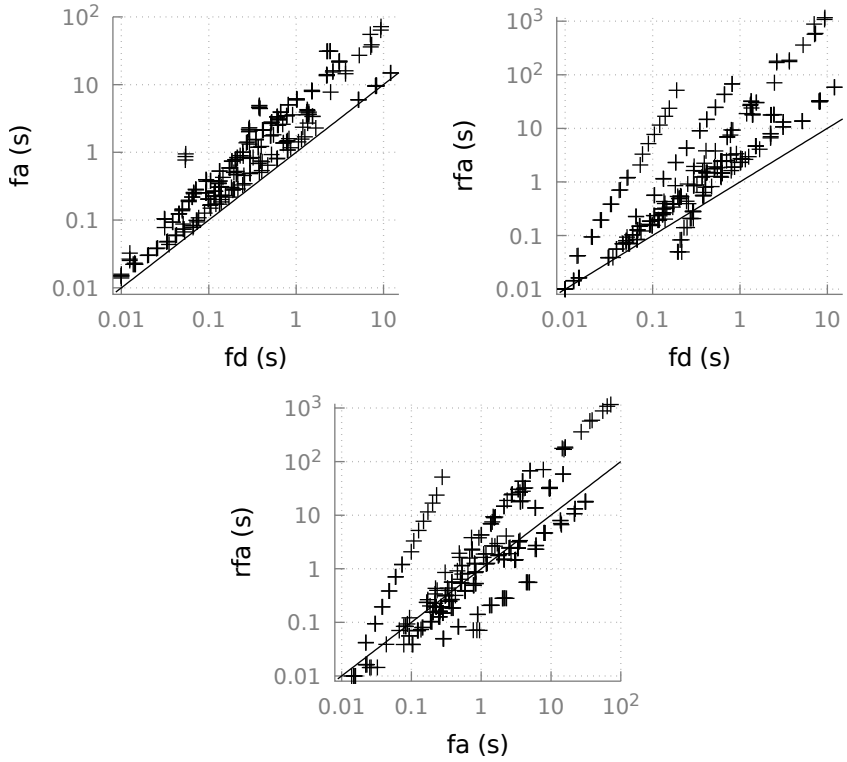


Figure 4.4: Running times in seconds of a whole translation process as scatter plots with logarithmic scales.

4.4 Future Work

The promising results of the algorithms we proposed are a motivation for further investigation of mutex invariants, their inference and application.

To support more PDDL domains, the algorithms needs to be extended for negative preconditions and conditional effects. For the algorithms described in this work, negative preconditions can be safely ignored because they cannot invalidate definition of fa-mutexes or rfa-mutexes and Algorithm 4 would also work the same way. However, the question is whether they can be used to extend the definitions and thus enable it to infer more invariants. On the other hand, conditional effects must be somehow taken into consideration in all presented algorithms to make them work.

The measured running times of the algorithm inferring rfa-mutexes clearly indicate that a faster implementation is required to make it a viable option. The number of inferred mutexes and above all the number of created variables in FDR certainly make this algorithm worth the effort.

The ILP formulation used in the algorithm for inference of fa-mutexes provides a strong tool for alternating the algorithm to deliver a more specific mutexes of certain shapes and sizes. For example, the translation to FDR could be encoded directly into ILP. Instead of spending time in inference of all fa-mutexes only to use some of them later for creation of variables,

we could modify the linear program in such a way it directly generates an encoding of FDR variables that fulfill some specified objective (e.g., minimal number of variables or minimal memory footprint of a state).

The applicability of the proposed algorithms will be further investigated. The most straightforward application of fa-mutexes (and rfa-mutexes) is its use in a translation to FDR (as was already demonstrated) and we should evaluate the impact of a different encoding of FDR variables on the actual planning process. Another application could be heuristic functions that reuse the inferred mutexes. One example could be a heuristic based on domain transition graphs (DTG) [Helmert, 2006; Torreño et al., 2014]. This heuristic was originally proposed for FDR where it is built on top of the variables of the task. The heuristic can be built directly from inferred mutex invariants which could provide better heuristic values.

Chapter 5
Conclusion

This work was focused on inference of mutual exclusion (mutex) state invariants in a context of STRIPS planning. We have introduced two new weaker types of mutex invariants called fact-alternating mutex (fa-mutex) invariant and restricted fact-alternating mutex (rfa-mutex) invariant. We have provided an extensive discussion of differences and relations between these types of mutex invariants including a complexity analysis based on decision problems corresponding to finding maximum mutexes, maximum fa-mutexes and maximum rfa-mutexes in a planning task. The complexity analysis has shown that the decision problems corresponding to fa-mutexes and rfa-mutexes are NP-Complete whereas the decision problem corresponding to a general mutex is PSPACE-Complete, i.e., it is as hard as a planning itself. We also proved that the maximum possible number of maximal mutexes, maximal fa-mutexes and maximal rfa-mutexes is in all three cases exponential in a number of facts in the corresponding planning task.

Three new algorithms for inferring mutex invariants were introduced. The first one is able to infer some additional pair mutexes on top of those that are already provided as its input. The second one is an ILP based algorithm for inference of fa-mutexes that was proven to be complete with respect to maximal fa-mutexes. The last one is a polynomial algorithm for inference of rfa-mutexes.

All three algorithms were experimentally evaluated and compared with two state-of-the-art algorithms for inferring mutex invariants, namely h^2 [Haslum and Geffner, 2000], which is a variant of a generalization of $h^{\max}$ heuristics, and an algorithm proposed by Helmert [2009] for a translation of planning tasks from PDDL to FDR that is widely used amongst the planning community. The algorithm for inference of additional pair mutexes has shown to be useful only in a limited number of domains. The algorithm for inference of rfa-mutexes demonstrated good results in a number of inferred mutexes, but very poor results in terms of running times due to its ineffective implementation that should be addressed in a future work. However, the algorithm for inference of fa-mutex proved to be a comparable with both state-of-the-art algorithms and it was also shown to be a viable replacement for Helmert's algorithm in a translation of planning tasks from PDDL to FDR.

Appendix A

Experimental Results: Additional Figures

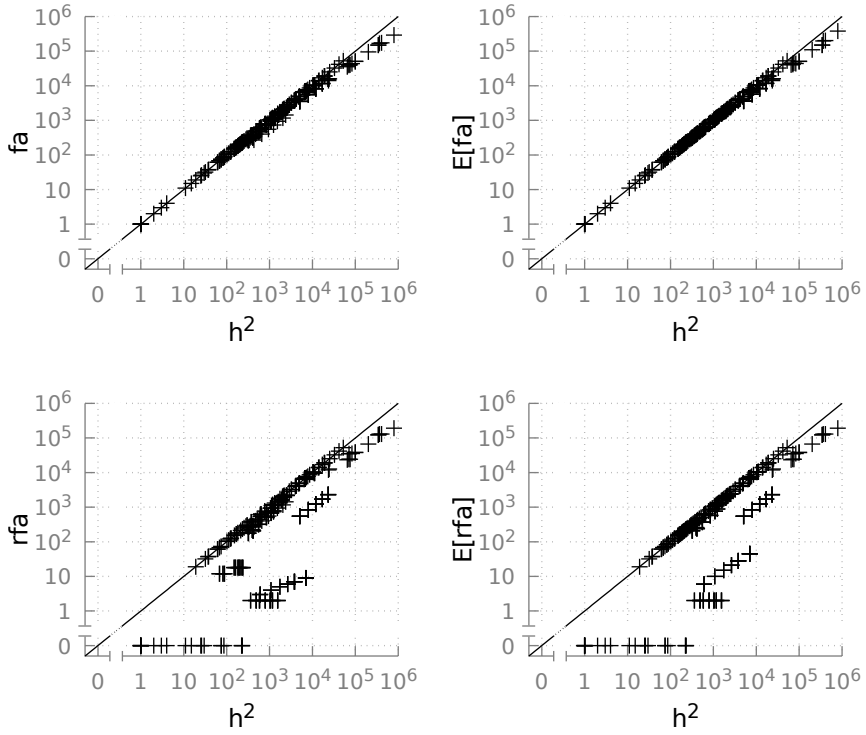


Figure A.1: Number of inferred pair mutexes as scatter plots with logarithmic scales with added zero. Comparison with $\mathtt{h}^2$.

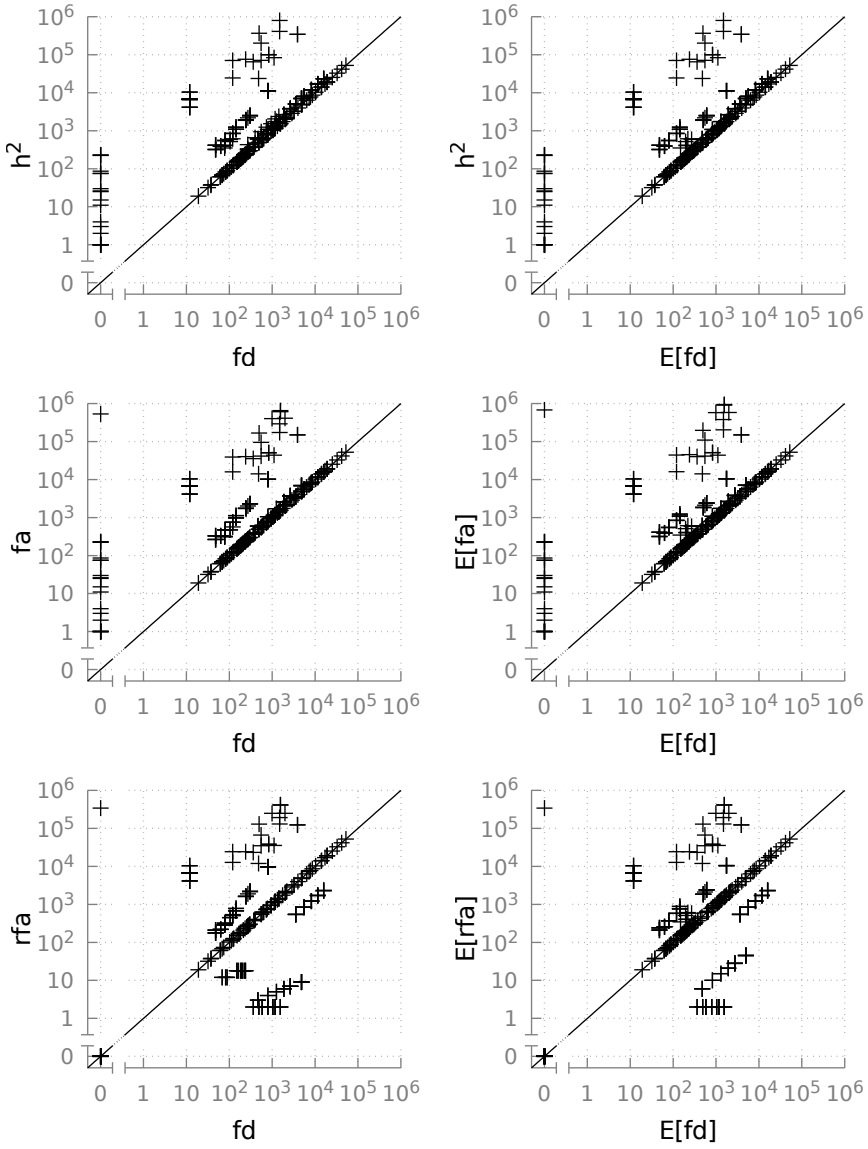


Figure A.2: Number of inferred pair mutexes as scatter plots with logarithmic scales with added zero. Comparison with `fd`.

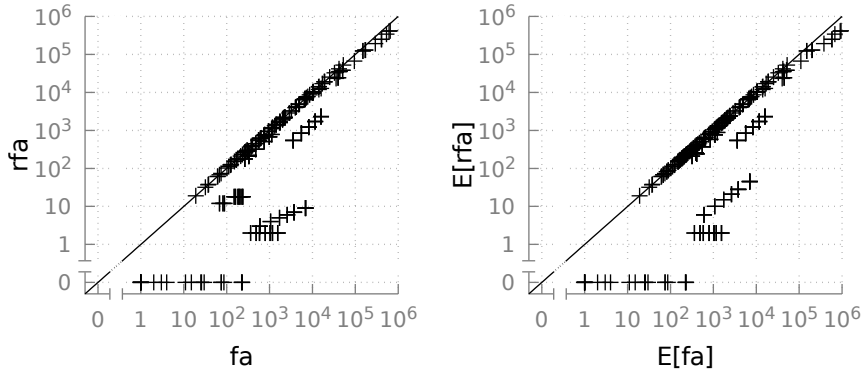


Figure A.3: Number of inferred pair mutexes as scatter plots with logarithmic scales with added zero. Comparison between `fa` and `rfa`.

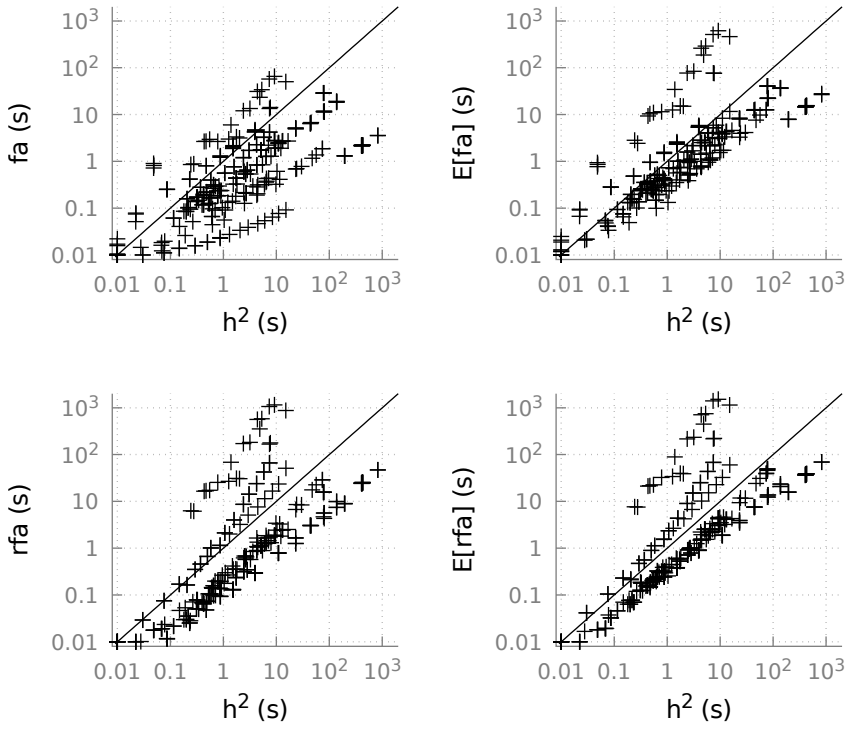


Figure A.4: Running times of inference algorithms as scatter plots with logarithmic scales. Comparison with $\mathtt{h}^2$.

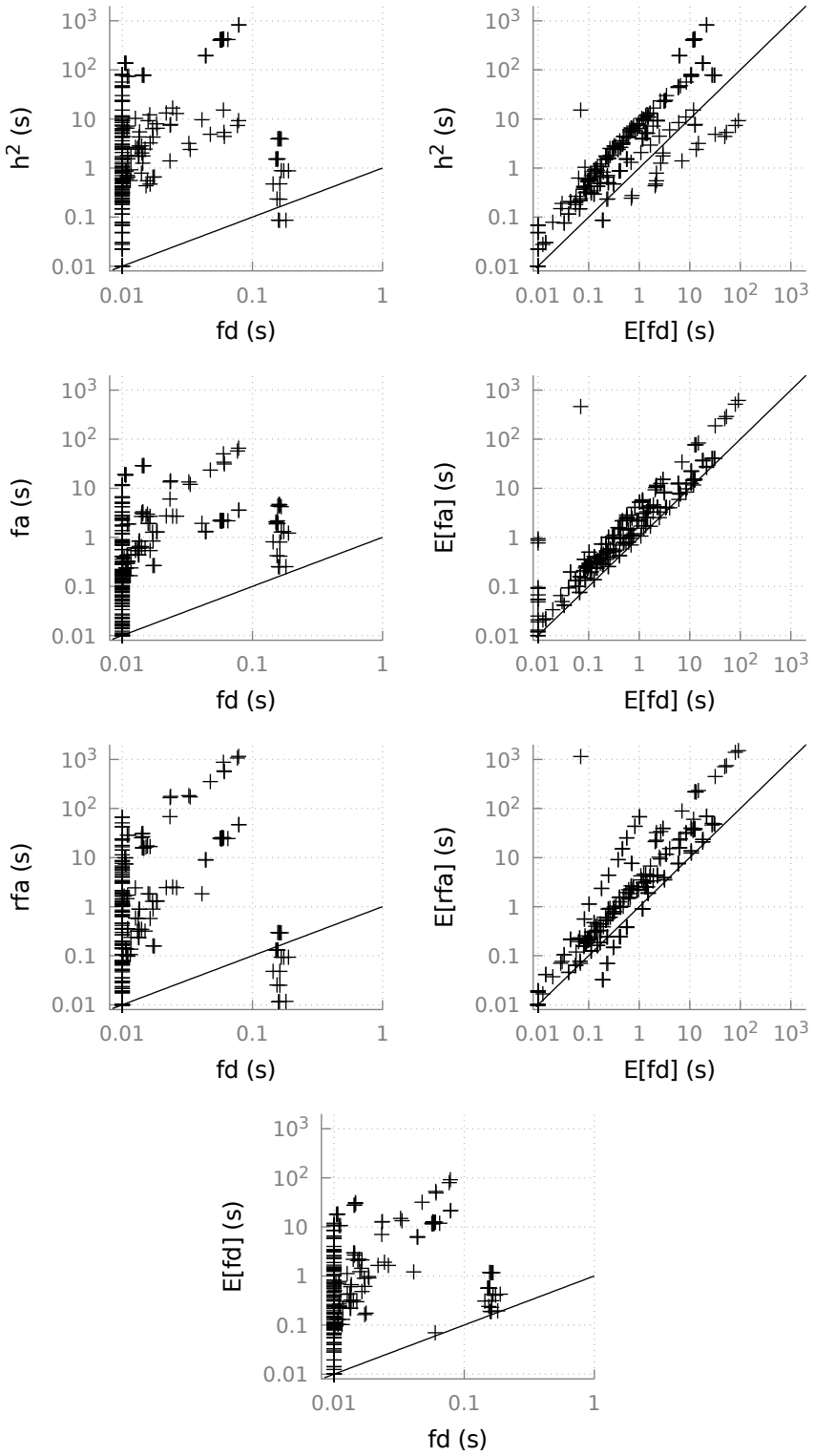


Figure A.5: Running times of inference algorithms as scatter plots with logarithmic scales. Comparison with `fd`.

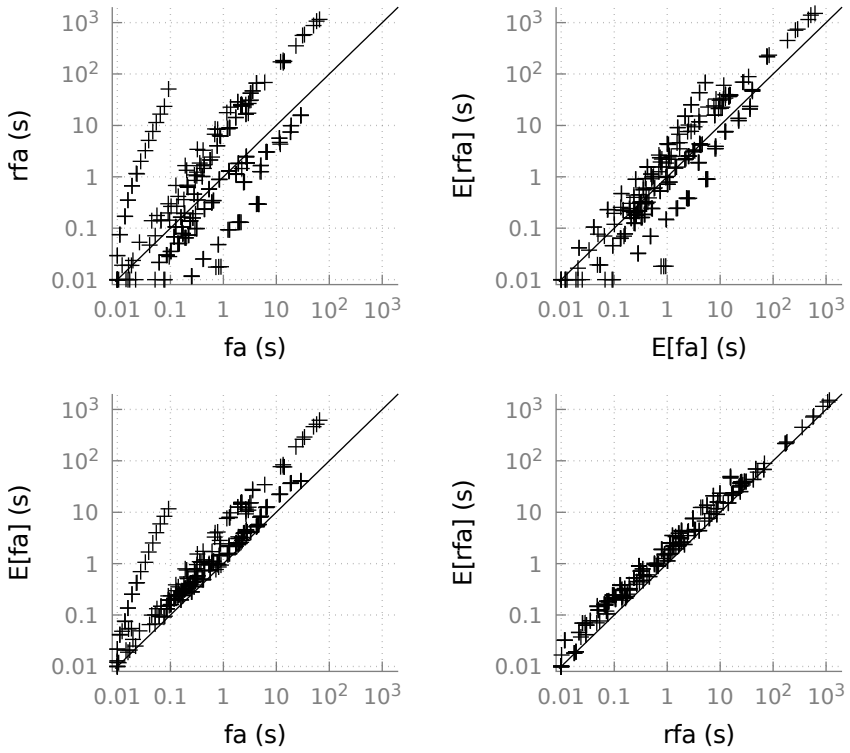


Figure A.6: Running times of inference algorithms as scatter plots with logarithmic scales. Comparison between `fa` and `rfa`.

Bibliography

Alcázar, V. and Torralba, Á. (2015). A reminder about the importance of computing and exploiting invariants in planning. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS*, pages 2–6.

Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *Artif. Intell.*, 129(1-2):5–33.

Bron, C. and Kerbosch, J. (1973). Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576.

Bylander, T. (1994). The computational complexity of propositional STRIPS planning. *Artif. Intell.*, 69(1-2):165–204.

Cresswell, S., Fox, M., and Long, D. (2002). Extending TIM domain analysis to handle ADL constructs. In *Knowledge Engineering Tools and Techniques for AI Planning: AIPS'02 Workshop*.

Fikes, R. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208.

Fox, M. and Long, D. (1998). The automatic inference of state invariants in TIM. *J. Artif. Intell. Res. (JAIR)*, 9:367–421.

Gerevini, A. and Schubert, L. K. (1998). Inferring state constraints for domain-independent planning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI, IAAI*, pages 905–912.

Gerevini, A. and Schubert, L. K. (2000). Discovering state constraints in DISCOPLAN: some new results. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence*, pages 761–767.

Haslum, P. (2009). $h^m(P) = h^1(P^m)$: Alternative characterisations of the generalisation from $h^{\max}$ to h^m. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS*, pages 354–357.

Haslum, P. and Geffner, H. (2000). Admissible heuristics for optimal planning. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems, AIPS*, pages 140–149.

Helmert, M. (2006). The fast downward planning system. *J. Artif. Intell. Res. (JAIR)*, 26:191–246.

Helmert, M. (2009). Concise finite-domain representations for PDDL planning tasks. *Artif. Intell.*, 173(5–6):503–535.

Kautz, H. A. and Selman, B. (1992). Planning as satisfiability. In *Tenth European Conference on Artificial Intelligence, ECAI*, pages 359–363.

Moon, J. W. and Moser, L. (1965). On cliques in graphs. *Israel Journal of Mathematics*, 3(1):23–28.

Mukherji, P. and Schubert, L. K. (2005). Discovering planning invariants as anomalies in state descriptions. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling, ICAPS*, pages 223–230.

Mukherji, P. and Schubert, L. K. (2006). State-based discovery and verification of propositional planning invariants. In *Proceedings of the 2006 International Conference on Artificial Intelligence, ICAI*, pages 465–471.

Rintanen, J. (2000). An iterative algorithm for synthesizing invariants. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, AAAI, IAAI*, pages 806–811.

Rintanen, J. (2008). Regression for classical and nondeterministic planning. In *Proceedings of the 18th European Conference on Artificial Intelligence, ECAI*, pages 568–572.

Sideris, A. and Dimopoulos, Y. (2010). Constraint propagation in propositional planning. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling, ICAPS*, pages 153–160.

Torreño, A., Onaindia, E., and Sapena, O. (2014). FMAP: distributed cooperative multi-agent planning. *Appl. Intell.*, 41(2):606–626.

Vallati, M., Chrpa, L., Grzes, M., McCluskey, T. L., Roberts, M., and Sanner, S. (2015). The 2014 International Planning Competition: Progress and trends. *AI Magazine*, 36(3):90–98.