Computing Schematic Invariants by Reduction to Ground Invariants

Lukas Vogel

Supervisor: Dr. Gabriele Röger

Universität Basel
Philosophisch-Naturwissenschaftliche Fakultät

19.07.2023

Abstract

This thesis is about implementing Jussi Rintanen's algorithm for schematic invariants. The algorithm is implemented in the planning tool Fast Downward and refers to Rintanen's paper *Schematic Invariants by Reduction to Ground Invariants*. The thesis describes all necessary definitions to understand the algorithm and draws a comparison between the original task and a reduced task in terms of runtime and number of grounded actions.

Contents

1 Introduction

An invariant is a statement that holds across the execution of specific program instructions. More specifically, invariants contains the reachability information of a state-space search algorithm. Furthermore, invariants for pruning can be used in planning algorithms. Many existing algorithms work with grounded actions, but if the number of these grounded actions are very high, it is impractical for an efficient implementation. In order to increase efficiency, it is advisable to work directly with schematic actions and to create invariants that also have a schematic form and are not grounded.

This work is based on the paper by Jussi Rintanen: *Schematic Invariants by Reduction to Ground Invariants*. Rintanen introduces an algorithm that combines the advantages of schematic algorithms and grounded algorithms. The algorithm works with schematic invariants, which are then checked whether a grounded action has an influence on the invariant. If this is the case, then a general regression is performed using a grounded instance of the schematic formula. If the result of the regression is satisfiable, the original schematic invariant has to be weakened.

Since Rintanen does not show any results of his algorithm, the task of this work is the implementation of the algorithm in the planning environment Fast Downward. With this planning environment, for example, a STRIPS or SAS+ planning task can be created. Fast Downward works with PDDL logic and supports classic problems like the Blocks World problem or the Sokoban problem. Rintanen's algorithm for creating invariants for classical planning is implemented in Fast Downward in this work. Fast Downward, provides a PDDL task that contains all the relevant information needed to compute the invariants. Fast Downward supports computation of invariants, but not schematic invariants created by reduction to grounded invariants. By implementing the algorithm, it should be determined whether the plannig task is more efficient when using schematic invariants instead of grounded ones. With the invariants of the algorithm obtained, mutex groups can be formed, which are then used by Fast Downward for the transformation to a finite-domain representation. The current implementation of Fast Downward works with grounded invariants which are also processed in the form of mutex groups and used for the transformation.

In this thesis, all definitions regarding Classical Planning, PDDL, Invariants and Mutex groups are given first, as these are all relevant in Fast Downward. Furthermore, the definitions are provided, which are derived from Rintanten's paper *Schematic Invariants by Reduction to Ground Invariants*. These definitions are necessary to understand the algorithm that Rintanen describes in his paper. After the introduction of the algorithm, it is shown how the algorithm is implemented in Fast Downward in the programming language Python, followed by the evaluation of the obtained invariant candidates that the algorithm returns.

Finally, a comparison is drawn between the runtime and the number of grounded actions for a normal task and a task that has been reduced, i.e. in which objects have been removed. The reduced task is based on limited grounding. The conclusion shows that the implemented algorithm, i.e. the implementation in Fast Downward, is not flawless, but Rintanen's approach is correct.

2 Background

In this section an introduction is given to the topics that are not directly defined in Rintanen's paper *Schematic Invariants by Reduction to Ground Invariants* but are required to understand the algorithm. The algorithm is related to Classical Planning, which is introduced in this section. The algorithm calculates invariants, which are also defined here. Since the algorithm is implemented in the Fast Downward planning environment and Fast Downward works with PDDL, a brief overview of PDDL is also provided and further explained in an example using Blocks World Problem. The calculated invariants are further processed in the form of mutex groups (in Fast Downward). Therefore, this section also contains a brief definition of mutex groups. In addition, the algorithm carries out a regression, which is defined here as a general regression.

2.1 Classical Planning

Classical planning is an approach to solving a problem by finding a sequence of actions. After executing these actions on the initial state, the goal state is reached, provided that the sequence of actions was found correctly. In Classical Planning, the environment is deterministic, as are the actions. The challenge in Classical Planning is to find this sequence of actions, which is also called a plan. (Lipovetzky, 2014)

Definition of Classical Planning A classical planning model $\Pi = \langle S, s_0, S_G, A, f \rangle$ consists of:

- A finite and discrete set of states S, i.e, the state space
- An initial state $s_0 \in S$,
- A set of goal states $S_G \in S$,
- A set of actions A,
- the actions applicable $A(s) \subseteq A$ in each state $s \in S$, and
- the deterministic transition function $s' = f(s, a)$ for $a \in A(s)$.

It follows that a classical plan is a sequence of actions $\pi = a_0, \ldots, a_n$. This sequence generates a sequence of states $s_0, \ldots, s_n$ so that $a_i \in A(s_i)$ is applicable in s_i. (Lipovetzky, 2014)

2.2 PDDL

Planning Domain Definition Language (PDDL) is a standardized method for representing planning tasks. PDDL is intended to express the physics of a domain. PDDL defines which predicates occur in planning tasks, which actions, which structure the actions have and the effects of the actions. In addition, PDDL also defines which requirements apply in order to be able to solve the task. (Aeronautiques et al., 1998)

Example Blocks World Problem In this thesis, the blocks world problem is used as an example. PDDL uses a domain file and a problem file. The domain file is used to define the requirements, the predicates and the actions that can be used in blocks world. The problem file, on the other hand, defines the objects that are used and also the initial and goal states.

2.2.1 Blocks World Domain File (PDDL)

```
(define (domain BLOCKS)
  (:requirements :strips)
  (:predicates (on ?x ?y)
               (ontable ?x)
               (clear ?x)
               (handempty)
               (holding ?x))
   (:action pick-up
               :parameters (?x)
               :precondition (and (clear ?x) (ontable ?x) (handempty))
               :effect
               (and (not (ontable ?x))
                    (not (clear ?x))
                    (not (handempty))
                    (holding ?x)))
   (:action put-down
               :parameters (?x)
               :precondition (holding ?x)
               :effect
               (and (not (holding ?x))
                    (clear ?x)
                    (handempty)
                    (ontable ?x)))
   (:action stack
               :parameters (?x ?y)
               :precondition (and (holding ?x) (clear ?y))
               :effect
               (and (not (holding ?x))
                    (not (clear ?y))
                    (clear ?x)
                    (handempty)
                    (on ?x ?y)))
   (:action unstack
               :parameters (?x ?y)
               :precondition (and (on ?x ?y) (clear ?x) (handempty))
               :effect
               (and (holding ?x)
                    (clear ?y)
                    (not (clear ?x))
                    (not (handempty))
                    (not (on ?x ?y)))))
```

2.2.2 Blocks World Problem File (PDDL)

```
(define (problem BLOCKS-4-0)
   (:domain BLOCKS)
   (:objects d a b c)
   (:init (clear c) (clear a) (clear b) (clear d) (ontable c)
   (ontable a) (ontable b) (ontable d) (handempty))
   (:goal (and (on d c) (on c b) (on b a))))
```

The domain file determines the rules for the task and the problem file states that there are four blocks a,b,c and d. In the init state, all these blocks are on the table and the target is to reach the goal state, which states that all blocks must be stacked on one another.

2.3 Invariants

An invariant of a planning task is a constraint that must be true for all world states reachable from the init state. There are many invariants that are not relevant and some that are useful but difficult to find. For example: *the goal is not satisfied* is an invariant if the task cannot be solved. This invariant is useful for marking the task as unsolvable from the beginning, but it is difficult to create this invariant before the task has passed a satisfiablility test. For example, an invariant that is not relevant but easy to find is: *at least five state variables are true.* While this invariant is true for most of the STRIPS planning task, it is not useful for solving the task faster because the invariant does not impose any restrictions on actions. (Helmert, 2009) Rintanen's algorithm is designed to calculate these invariants, but in their schematic state, i.e. the invariant does not contain any specific objects, only variables.

2.4 Mutex Group

Invariants which consits of mutually exclusive propositions are called mutexes. To be mutexes the invariants have to take the form of a binary clause. A set of literals is a mutex group if every subset of two literals is a mutex. (Saiz et al., 2014)

The invariants generated in Fast Downward are further processed in the form of mutex groups. These mutex groups are needed for the efficient solution of the classical planning problem.

2.4.1 Definition Mutex Group

A set of propositions $M = \{f_1, \ldots, f_m\}$ is a set of mutually exclusive propositions of size m (mutex size of m) if there is no state $s \subseteq S$ that may belong to a solution path such that $M \subseteq s$. (Saiz et al., 2014)

2.5 General Regression

The regression is used in the algorithm to test the possible invariant candidates that have arisen for satisfiability after an action, i.e. operator, has been applied to the grounded invariant candidate. The regression calculates all successors of a state, in this case an invariant candidate after the action on the State has been executed. The definition of general regression helps to apply the approach to Rintanen's algorithm. Finally, the regression of a formula is implemented by an action in the algorithm. The preceding definitions are used in the recursive calls to the function.

2.5.1 Definition Action

Let a be an action over state variables V:

- precondition of a written $pre(a)$ is a formula over V
- effect of a written $\mathit{eff}(a)$ is a action over V
- cost of a written $cost(a)$ is a number $n \in \mathbb{R}_0^+$

(Rintanen, 2008)

2.5.2 Definition Effect

An effect e is recursively defined as follows:

- v and $\neg v$ for state variables $v \in V$ are effects (atomic effect)
- $e_1 \wedge \cdots \wedge e_n$ is an effect if $e_1, \ldots, e_n$ are effects (conjunctive effect)
- $c \rhd e$ is an effect if c is a formula and e is an effect (conditional effect)

(Rintanen, 2008)

2.5.3 Definition Effect Condition

We recursively define the effect-conditon $\mathit{effcond}(e, e')$ as a propositional formula under which e triggers given effect e' as follows:

- $\mathit{effcond}(e, e) = \top$
- $\mathit{effcond}(e, e') = \bot$ for atomic effects $e' \neq e$
- $\mathit{effcond}(e, (e_1, \wedge \cdots \wedge e_n)) = \mathit{effcond}(e, e_1) \vee \cdots \vee \mathit{effcond}(e, e_n)$
- $\mathit{effcond}(e, (\chi \triangleright e')) = \chi \wedge \mathit{effcond}(e, e')$

(Rintanen, 2008)

2.5.4 Definition Regression of an state variable through an effect

The regression of $v \in V$ through an effect e is defined as follows:

- $regr(v, e) = \mathit{effcond}(v, e) \vee (v \wedge \neg \mathit{effcond}(\neg v, e))$

(Rintanen, 2008)

2.5.5 Definition Regression of a formula through an effect

Let φ be a propositional formula. The regression of φ through effect e is defined as follows:

- $regr(\top, e) = \top$
- $regr(\bot, e) = \bot$
- $regr(v, e) = \mathit{effcond}(v, e) \vee (v \wedge \neg \mathit{effcond}(\neg v, e))$
- $regr(\neg\psi, e) = \neg regr(\psi, e)$
- $regr(\psi \vee \chi, e) = regr(\psi, e) \vee regr(\chi, e)$
- $regr(\psi \wedge \chi, e) = regr(\psi, e) \wedge regr(\chi, e)$

(Rintanen, 2008)

2.5.6 Definition Regression of a formula through an action

Let φ be a propositional formula. The regression of φ through action a is defined as follows:

- $regr(\varphi, a) = pre(a) \wedge regr(\varphi,\ \mathit{eff}(a))$

(Rintanen, 2008)

3 Algorithm

In this chapter, we examine the algorithm proposed by Rintanen. The algorithm begins with a set of candidate schematic invariants that hold true in the initial state. These candidates are then converted into grounded invariants, tested on satisfiability, and if successful, a weakenend version of the original candidate is explored. If all invariants cannot be further weakened, the algorithm is terminated and the invariant candidates are returned in the form of mutex groups. To facilitate comprehension and clarity, we provide definitions for all essential elements of the algorithm.

Algorithms that work with invariants are nothing new. Blum and Furst 1997, Rintanen 1998/2008 have already developed algorithms that ground all schematic invariants. Algorithms have also been developed that work with ungrounded actions (Gerevini and Schubert 1998/2000, Edelkamp and Helmert 2000, Rintanen 2000, Lin 2004). Both variants are very slow in some cases. The grounded algorithm is very slow when there are thousands of ground actions. The schematic algorithm, on the other hand, is very slow if there are a large number of schematic actions. Also, schematic algorithms are very slow if the actions are very complex, regardless of the number of instances grounded. The advantage of grounded algorithms is that an efficient implementation is easier than with a schematic algorithm. This is because the schematic algorithm has to implement substitution. In order to get the advantages of both algorithms, a combination of these two algorithms is discussed in this work. More precisely, in the combination of the algorithms, schematic invariants are generated depending on grounded instances of actions, but without generating all grounded instances. If the goal is to generate schematic invariants, then it is sufficient to generate a small set of ground instances instead of generating all grounded instances of schematic actions. This approach must be correct and the found invariants must also be useful. This is shown by the following Lemma: (Rintanen, 2017)

Lemma 1 Assume a formula ϕ is an invariant of a problem instance Π. Let 0 be a set of objects and I the initial state. Let Π' be a probem instance such that

1. Π' has he same schematic actions,
2. $O \subset O'$, and
3. $I(x) = I'(x)$ for all state variables x of Π.

Then any schematic formula ϕ for which some ground instances are not invariants of Π, also has ground instances with respect to Π' that are not invariants of Π'. (Rintanen, 2017)

Proof: Consider a reachable state of Π that falsifies a ground instance of ϕ. Let π be an action sequence that reaches that state. Since all these actions are also ground actions of Π', and the initial state of Π is included in the initial state of Π', also this action sequence reaches a state that falsifies a ground instance of ϕ in the state-space of Π'. □
(Rintanen, 2017) The lemma shows that even if the number of objects is increased, not too many invariants are found, which are not invariants of the original problem. However, reducing the number of objects can lead to invariants that do not apply to the original problem. A simple example is the Blocks World Problem. If there is only one block in the problem, then $\neg on(x, y)$ is an invariant, since there are no two blocks to stack on top of each other. The goal is to find the smallest possible number of objects that are still considered invariants for the original problem. (Rintanen, 2017)

Another example is the invariant $P(x, y) \vee P(y, z)$. If x,y and z are all of different types, then the number of objects of each type is equal to the number of variables of that type. If x,y,z all have the same type, then five different cases must be considered.(Rintanen, 2017)

1. $x = y = z$
2. $x \neq y \neq z$
3. $x = y$
4. $x = z$
5. $y = z$

To test these five cases, the objects are instantiated with all their different variables. Suppose that the variables $\{a, b, c\}$ belong to this type, e.g. $dom(x) = dom(y) = dom(z) = \{a, b, c\}$. The five cases could be instantiated as follows:

1. $x = y = z = a$
2. $x \neq y \neq z \rightarrow x = a, y = b, z = c$
3. $x = y = a$, then $z = b \vee c$
4. $x = z = a$, then $y = b \vee c$
5. $y = z = a$, then $x = b \vee c$.

In the next step, the smallest possible number of objects of each type needed for computing invariants is defined. The invariants should be calculated using partial grounding. Furthermore, the invariant consists of disjunctions with a maximum number of N literals.(Rintanen, 2017)

Limited Instantiation : For a given action set A, predicate set P, domain function D, type t and integer $N \geq 1$, define: $L_t^N(A, P) = max(max_{a \in A} prms_t(a), max_{p \in P} prms_t(p)) + (N - 1) \cdot (max_{p \in P} prms_t(p))$ In the Limited Instantiation, $prms_t(a)$ is the number of schema variables of type t in action a and $prms_t(p)$ is the number of terms of type t in the schema formula with predicate p . (Rintanen, 2017)

Lemma 2 : Let a be a ground instance of a schematic action in A and ϕ a ground instance of a schematic disjunction of N literals, both formed from state variables with predicates from P. Assume a is relevant for ϕ. Then there are at most $L_t^N(A, P)$ different non-fixed objects of type t in $regr_a(\phi)$. (Rintanen, 2017) Limited instantiation is used for limited grounding. The task should be reduced so that the runtime of the algorithm is improved.

Theorem 1 states that if a schematic invariant candidate is falsified by a ground action set containing all grounded actions, then the invariant candidate is also falsifiable, with a much smaller grounded action set. On this basis, the task is reduced in Fast Downward. In this thesis, a comparison is then drawn between a task to which this therorem has been applied and a task that has been left in its original state.

Theorem 1 : Let A be a set of schematic actions, D domain function for types T, D' a domain function such that for every $t \in T$, either $D'(t) = D(t)$ or

1. $D'(t) \subset D(t)$
2. $|D'(t)| \geq L_t^N(A, P)$,
3. $D'(t)$ includes all fixed objects of type t and
4. $D'(t_0) \subset D'(t_1)$ *iff* $D(t_0) \subset D(t_1)$, for all $\{t_0, t_1\} \subseteq T$.

(Rintanen, 2017) Let C be a set of schematic formulas, C_D a grounded instance of C depending on D and C'_D a grounded instance of C depending on D'. Let ϕ be a schematic disjunction with a maximum number of N literals, then $C_D \cup \{regr_a(\phi\sigma)\}$ is satisfiable for a grounded instance of $a\sigma$ and $a \in A$ and a grounded instance $\phi\sigma$ of ϕ depending on D, for a $\sigma : V \rightarrow O$ and $a\sigma$ is relevant for $\phi\sigma$. Then $C'_D \cup \{regr_{a\sigma'}(\phi\sigma')\}$ is satisfiable for some σ' and σ' is contained in D'.
Proof: Let v be a valuation such that $v \models C_D \cup \{regr_{a\sigma}(\phi\sigma)\}$. We will construct a substitution σ' and then a valuation v' such that $v' \models C_{D'} \cup \{regr_{a\sigma'}(\phi\sigma')\}$. Let $R(t)$, for all $t \in T$, be any subset of $D(t)$ of same cardinality as $D'(t)$ that includes all objects of type t in $regr_{a\sigma}(\phi\sigma)$. Such sets $R(t)$ exist because of the assumption that $|D'(t)| \geq L_t^N(A, P)$ and because of Lemma 2. Let $R = \bigcup_{t \in T} R(t)$. Let π a bijective mapping $\pi : R \rightarrow \bigcup_{t \in T} D'(t)$ such that for all $o \in R$ and $t \in T$, $\pi(o) \in D'(t)$ iff $o \in D(t)$. Such a mapping exists because the cardinalities of $R(t)$ and $D'(t)$ are the same. Define $\sigma' = \sigma\pi$. Define a valuation v' by $v'(\pi(x)) = v(x)$ for every state variable x occuring in $a\sigma$ and $\phi\sigma$. Clearly, for every such $x, v \models x$ iff $v' \models \pi(x)$, and hence $v' \models regr_{\sigma'}(\phi\sigma')$. It remains to show that $v' \models C_{D'}$. Take any $\psi' \in C_{D'}$. Now $\psi' = \pi(\psi)$ for some $\psi \in C_D$ because C_D contains all ground instances of C with respect to D, and $\pi(\psi) \in C_{D'} \subseteq C_D$. By assumption, $v \models \psi$. Since $\psi' = \pi(\psi)$, also $v' \models \psi'$. □
(Rintanen, 2017)

3.1 Definitions used in Rintanen's Algorithm

The definitions below are all taken from Rintanen's paper and are used to adapt the previously introduced terms as they will eventually be used in the algorithm. For example, a new definition for a planning task and for invariants, in this case invariant candidates, is given.(Rintanen, 2017)

Definition Types Let O be a set of objects. Let there be a finite set T of types and to each type $t \in T$ a non-empty set $D(t) \subseteq O$ of objects is associated by the domain function $D : T \to O$. For this thesis we assume that if $D(t) \cap D(t') \neq \emptyset$, then either $D(t) \subseteq D(t')$ or $D(t') \subseteq D(t)$. (Rintanen, 2017)

Definition Terms Let V be a set of schema variables and O a set of objects. Terms (over O and V) are objects $o \in O$ and variables $v \in V$. Each variable has a type $\tau_{var}(v) \in T$. (Rintanen, 2017)

Definition Predicates Let P be a set of predicate symbols. Each predicate $p \in P$ has arity $ar(p) \in \mathbb{N}$ and an associated type $\tau_{pre}(p) \in T^{ar(p)}$, the latter given by the typing function τ_{pre}. (Rintanen, 2017)

Definition State Variables Let $p \in P$ be a predicate symbol of arity $n = ar(p)$ and type $\tau_{pre}(p) = (t_1, \ldots, t_n)$. Then schematic state variables are of the form $p(s_1, \ldots, s_n)$ where each s_i is either an object $o \in D(t_i)$ or a variable v with $\tau_{var}(v) = t_i$. The set $gnf(P, \tau_{pre}, D)$ of (ground) state variables consists of all $p(o_1, \ldots, o_n)$ such that $p \in P, ar(p) = n$, and $o_i \in D(t_i)$ where $\tau_{pre}(p) = (t_1, \ldots, t_n)$. In this thesis we consider only boolean state variables. (Rintanen, 2017)

Definition States Let P be a set of predicates and D a domain function. A state is a mapping from $gnf(P, \tau_{pre}, D)$ to $\{0, 1\}$, indicating the truth-values of every state variable. (Rintanen, 2017)

Definition Schematic Formulas Let O be a set of objects, V a set of variables, and P a set of predicates with arities $ar(p)$ for every $p \in P$. The following are schematic formulas over O and V:

- schematic state variables $p(s_1, \ldots, s_n)$ over V and O
- $\phi_1 \wedge \phi_2$, if ϕ_1 and ϕ_2 are schematic formulas
- $\phi_1 \vee \phi_2$, if ϕ_1 and ϕ_2 are schematic formulas
- $\neg\phi$, if ϕ is a schematic formula

(Rintanen, 2017)

Definition Schematic Effects Let $p(s_1, \ldots, s_n)$ be a schematic state variable. Then $p(s_1, \ldots, s_n)$ and $\neg p(s_1, \ldots, s_n)$ are schematic effects. (Rintanen, 2017)

Definition Schematic Actions A schematic action over O and V is a pair $\langle c, e \rangle$ where

- c is a schematic formula over O and V, and
- e is a set of schematic effects over O and V.

We define $prec(\langle c, e \rangle) = c$. (Rintanen, 2017)

Definition Planning Task A problem instance in planning $\sqcap = \langle O, T, D, P, \tau_{pre}, A, I \rangle$ consists of:

- a finite set O of objects
- a finite set T of types
- a domain function D
- a finite set P of predicates
- a typing function τ_{pre}
- a finite set A of schematic actions over O and V
- a initial state I

(Rintanen, 2017)

Definition Candidate Invariants Candidate invariants are schematic formulas (with free variables) of the forms:

- $\chi \rightarrow (l_1 \vee l_2)$
- $\chi \rightarrow l_1$

where χ is a possibly empty conjunction of inequalities $x \neq x'$ where x and x' are schema variables referring to objects, and the l_i are boolean schematic state variables or negated schematic state variables (literals). (Rintanen, 2017)

Definition Substitutions For sets V of schema variables and sets O of objects, a function $\sigma : V \rightarrow V \cup O$ is a substitution (Rintanen, 2017)

Definition Application of Substitutions For formulas ϕ, schematic actions a, and other syntactic objects, we define $\phi\sigma$ and $a\sigma$ as the syntactic objects respectively obtained from ϕ and a by replacing every occurrrence of $v \in V$ by $\sigma(v)$. (Rintanen, 2017)

Definition Substitution Composition Let V and Q be sets of schema variables and O a set of constant symbols (object names). Let $\sigma : V \rightarrow V \cup O$ and $\sigma' : V \rightarrow Q \cup O$ be substitutions. Then $\sigma\sigma' : V \rightarrow Q \cup O$ is the substitution defined by $\sigma\sigma'(v) = \sigma'(\sigma(v))$ (also denoted by $\sigma' \circ \sigma$) (Rintanen, 2017)

3.2 Satisfiable-Test with Vampire

Vampire is used to test the formula obtained from the regression for satisfiability. Vampire is a theorem prover that has its primary power in First Order Logic. The theorem prover receives the previous invariant candidates as so-called axioms and the result of the regression as a negated conjecture. Vampire then checks whether the negated conjecture is satisfiable with the given axioms or a refutation will be found. (Voronkov, 1990) In Rintanen's algorithm, $regr_a(\neg c\sigma)$ is executed. Since input is negated in the regression, the satisfiability test is a proof by refutation

Proof by Refutation Given a problem with axioms $X_1, \ldots X_n$ and a conjecture C. With proof by refutation, the conjecture is negated first. Then the negated conjecture is checked for unsatisfiability, given axioms $X_1, \ldots, X_n$. If the result is satisfiable, then the original conjecture C can be assumed to be satisfiable. In Rintanten's paper, an invariant candidate is grounded and negated. The grounded invariant candidate is thus passed as a negated conjecture and if the result is satisfactory, the original, non-negated, invariant candidate is satisfactory. (Kovács and Voronkov, 2013)

First Order Logic First order logic (FOL) often also called first-order formulas is a delimitation of the language or a logical formula. FOL uses the following operations as basic building blocks: $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\rightarrow$ (implies), $=$ (equal), $\forall$ (forall), $\exists$ (exists). In addition, an infinite set of variables is required. With these basics it is possible to create a logical formula, which then belongs to first-order logic. (Barwise, 1977)

Definition for FOL : The first-order formulas of a language L together form the smallest set of expressions containing atomic formulas and closed under the following formatting rules:

- If φ, ψ are formulas, then the expressions

  $$\neg\varphi, (\varphi \wedge \psi), (\varphi \vee \psi), (\varphi \rightarrow \psi)$$

  are also formulas.
- If φ is a formula and v is a variable, then $(\exists v\varphi)$ and $(\forall v\varphi)$ are also formulas. (Barwise, 1977)

TPTP The TPTP (Thousands of Problems for Theorem Provers) is a valuable resource for the automated theorem provers (ATP) community. It serves as a library of test problems to support the development and evaluation of ATP systems. The TPTP offers several main functions: TPTP provides a large number of ATP problems and serves as a template and reference to get started with creating tptp files for theorem provers like vampire. These problems help developers and researchers the to test and evaluate the performance of an atp system over many different complex problems. An example of such a problem would be the N-queens problem found in the library. TPTP also follows a concrete syntax with a specific format. This makes it easy to translate your own problems into tptp. (Sutcliffe Geoff, 2001)

A specific syntax is used to use FOL in TPTP. For the language, it should be noted that variables, functions and predicates are passed as symbols. Variable names always start with capital letters. For terms, it should be noted that all variables, constants and expressions $f(t_1, \ldots, t_n)$, where f is a function symbol of arity n and $t_1, \ldots, t_n$ are terms. A term determines domain elements. An atomic formula as an expression $p(t,1 \ldots, t_n)$ where p is a predicate symbol of arity n and $t_1, \cdots, t_n$ terms, determines the property of a domain element. All symbols are not interpreted, except equality = (Kovács and Voronkov, 2013)

FOL	TPTP
$\bot, \top$	$\$false, \$true$
$\neg F$	$\sim F$
$F_1 \wedge \cdots \wedge F_n$	$F1\& \ldots \& Fn$
$F_1 \vee \cdots \vee F_n$	$F1 \mid \cdots \mid Fn$
$F_1 \rightarrow F_n$	$F1 => Fn$
$(\forall x_1) \ldots (\forall x_n) F$	$![X1, \ldots, Xn] : F$
$(\exists x_1) \ldots (\exists x_n) F$	$?[X1, \ldots, Xn] : F$

As already mentioned, vampire receives several axioms and a negated conjecture. This is because vampires handle conjectures differently internally to make the proof more goal oriented. A Proof by Refutation is applied. Given is a problem with axioms $F_1, \ldots, F_n$ and conjecture G. First the conjecture is negated and vampire checks the set $\{F_1, \ldots, F_n, \neg G\}$ for unsatisfiability. (Kovács and Voronkov, 2013)

Typed First Order Formulas Typed First Order Formulas (tff) have almost the same syntax as First Order Formulas. Since the objects have different types in the PDDL task, these types must also be taken into account in the satisfiable test. To do this, each variable in a formula must be assigned a type and defined in the tptp file. In addition, each type and subtype must be defined in the tptp file. (Sutcliffe and Kotelnikov, 2018)

TFF extends the FOL language with types and type declarations. Each function, type and predicate must first be defined with its type signature. Each type is one of the following tff types:

- the predefined types \$i for i (individuals) and \$o for o (booleans),
- the predefined arithmetic types \$int (integers), \$rat (rationals), and \$real (reals) or
- user-defined types (constants)

User-defined types must always be declared before they are used. All user-defined types are of type \$tType and are declared as type. Each type signator declares either

- an individual type τ or
- a function type or predicate type $(\tau_1 \cdot \cdots \cdot \tau_n) > \tilde{\tau}$ for $n > 0$, where τ_i are argument types and $\tilde{\tau}$ is the result type.

(Sutcliffe and Kotelnikov, 2018)

For example, a tptp file with first order formulas would look like this:

fof(formula1, axiom, ![X0, X1]: ∼on(X0,X1)).
fof(formula0, negated_ conjecture, ![A,B,D]: (holding(A) & clear(B) & ∼on(D,D))).

Formula1 is defined as an axiom and is a formula that holds. Vampire then checks if the negated_conjecture is satisfiable with the given axioms. It is therefore checked whether holding(A) & clear(B) & ¬ on(D,D) applies, given that ¬ on(X0,X1) applies to all X0 and X1.

To convert a First Order Formula into Typed First order, the associated types for each variable $X0, X1, A, B$ and D are first defined. The tptp file will then look like this:

tff(type_ dec1, type, object: $tType).
*tff(on_ decl, type, on: (object * object) > $o).*
tff(ontable_ decl, type, ontable: object > $o).
tff(clear_ decl, type, clear: object > $o).
tff(handempty_ decl, type, handempty: > $o).
tff(holding_ decl, type, holding: object > $o).
*tff(equal_ decl, type, equal: (object * object) > $o).*
tff(formula1, axiom, ![X0:object, X1:object]: (∼ on(X0,X1))).
tff(formula0, negated_ conjecture, ![A:object,B:object,D:object]: (holding(A) & clear(B) & ∼ on(D,D))).

The basic types are defined here first. There is an type *object* that represents a *$tType* (*object* is a user defined type). Then for all existing predicates it must be defined which types are expected. Predicates *on* and equal expect two variables, both of type *object*. Predicates *ontable, clear, holding* expect only one variable of type *object*. The predicate *handempty*, on the other hand, expects no variable. After all types and predicates have been defined, the *axioms* and the *negated_ conjecture* can be added. To do this, each variable that occurs in the *axioms* or in the *negated_ conjecture* must be assigned to a type. In the *negated_ conjecture* you can see that A, B and D must all be of type *object*. Vampire then checks whether, for example, which kind of type *holding(A)* is expected and whether A also corresponds to the expected type. Thus, type restrictions can be easily incorporated into Vampire.

3.3 weakening

Weaken() is a function that takes a schematic formula as a parameter and logically weakens it. In this thesis weakening was done in two ways. Let c be a logical formula passed as a parameter to weaken.

1. Let l be a literal not contained in c. Weakening creates a new formula $c_{new} = c \vee l$. By adding a literal in the form of a disjunction, c is weakened without changing its satisfiability. If c already has the weakest possible form, $\emptyset$ is returned.

2. Adding inequalities also weakens c. The literal $at(p, b)$ says that object is p and place is b. Since an object can only be in one place at a time, the inequality $b_1 \neq b_2 \rightarrow (\neg at(p, b_1) \vee \neg at(p, b_2))$ can be added to c during weakening.

(Rintanen, 2017)

3.4 Algorithm

The following algorithm is provided b Rintanen's Paper and uses the folowing definitions, as well as all the definitions given in the previous section. Let C be a set of schematic formulas. Let A be a set of actions. Let $regr_a(x)$ be a regression of a formula through an action (Definition Regression of a formula through an action). Let be $c\sigma$ be a substitution and let weaken be a function with the following properties:

- as parameter weaken receives a schematic formula
- return a weakened form of the schematic formula, or if the formula was already of the weakest possible form, then $\emptyset$ will be returned

Algorithm 1 Algorithm for invariants for classical planning

```
1: C := schematic formulas true in initial state;
2: repeat
3:     C_0 := C;
4:     for all a ∈ A and c ∈ C do
5:         if C_0 ∪ {regr_a(¬cσ)} ∈ SAT for some cσ then
6:             C := (C \ {c}) ∪ weaken(c);
7:         end if
8:     end for
9: until C = C_0;
10: return C;
```

(Rintanen, 2017)

The regression is done for any possible instance of $c\sigma$. c is at this time a schematic invariant candidate and with the multiplication with σ the candidate is transformed in each possible grounded variant. If the satisfiability test is successful, the original schematic invariant candidate is removed from the set of all invariant candidates and a weakened version is added and tested again for satisfiability in a later pass. Since weakening only adds formulas that do not change satisfiability, we have a correct invariant candidate at the beginning of each pass through the forloop. Only by choosing an action, an existing invariant candidate can fail the satisfiability test and the candidate is deleted as an invariant. The implemented algorithm successfully finds invariants to most optimal stripts problems in Fast Downward. The implemented algorithm fails for some tasks due to the major differences in the various tasks regarding the Pddl definition. this is described in later sections.

4 Discussion

4.1 Fast Downward

Fast Downward is a classic planning system which is based on heuristic search and works with general deterministic planning problems, which are formalized in PDDL. Fast Downward is a progression planner as it searches the state space of a world state in a forward direction. The PDDL input is not used directly by Fast Downward, but first translated into a so-called multivalued planning task. This translation ensures that many of the implicit constraints of the planning task are now explicit constraints. (Helmert, 2006) (Helmert Malte, 2003)

4.2 Algorithm in Fast Downward

The main part of this thesis involves the implementation of Rintanen's algorithm in the Fast Downward environment. The algorithm receives a PDDL task from Fast Downward, which receives all actions that occur in the respective PDDL problem. The task also includes the init-state, the goal-state and the various objects contained in the PDDL problem with their respective predicates and types. From this task, the invariant candidates can be generated at the beginning of the algorithm. The invarant candidates that are true at the beginning of the problem (in the init state) are generated by simply counting them. To do this, a predicate is selected from task.predicates. As an example, we choose predicate P with objects x, y, z, in short $P(x, y, z)$. Each object belongs to a specific type. The number of literals are calculated as follows: $|x| \cdot |y| \cdot |z|$. These number of literals corresponds to all possible combinations with which the objects are replaced by variables. If we now assume that $typ(x) = typ(y) = typ(z)$ and $typ(x)$ can contain variables a and b, we get $2 \cdot 2 \cdot 2 = 8$ for the number of literals. Therefore it follows that here are 8 ways to represent $P(x, y, z)$ differently. This can be, for example, $P(a, a, a)$ or also $P(a, b, a)$, etc. To check whether $P(x, y, z)$ is an invariant candidate, 8 different (the number of calculated literals) literals have to be present in the init state of the task.

In order to understand the counting approach better, a concrete example is shown by using $clear(X)$. X is of type *object* and *object* contains variables a, b, c and d. For the calculation,the length of object, which is equals to 4, is used. The four choices are $clear(a), clear(b), clear(c)$ and $clear(d)$. These four literals must appear in the init-state of the task. If the number of $clear()$-literals in the init-state is 4, then $clear(X)$ is an invariant candidate. However, if not a single $clear()$ literal occurs in the init-state, then $\neg clear(X)$ is an invariant candidate.Expressed mathematically: For variables in predicate P in position 0 to N. Number of literals $p = |typ0| \cdot \cdots \cdot |typN|$. If $p ==$ *number of predicates* P in init-state then predicate P is an Invariant candidate. If $p > 0$ and *number of predicates* P in init-state $== 0$, then predicate $\neg P$ is an invariant candidate.

Form of Invariant Candidates The schematic formulas that are initially true are presented as invariant candidates. In this thesis an invariant candidate has three parameters. The first parameter is a list of literals. All literals in the list together form a disjunction. The second parameter concerns the inequalities. The inequalities consist of a set of several sets. Each of these sets contains variables which are contained in the literals and defines which two variables must not be the same. The invariant candidates receive the type restrictions as the third parameter. This is a list of TypedObjects. A TypedObject contains a name for the variable and an associated type. A possible invariant candidate looks like this:

List of literals: [NegatedAtom on(?x0,?x1), Atom =(?val0, ?val1)]. Sets of inequalities: {set(?x1, ?val1), set(?x1, ?val0), set(?x1, ?x0), set(?x0, ?val0), set(?x0, ?val1)}. List of type restrictions: [?val0: object, ?val1: object].

This candidate invariant states that either ¬ on(?x0,?x1) holds or that =(?val0, ?val1) holds. The inequalities states that ?x1 ≠ ?val1, ?x1 ≠ ?val0, ?x1 ≠ ?x0, ?x0 ≠ ?val0 and ?x0 ≠ ?val1 . So only ?val0 and ?val1 can be the same variable. The type restriction specifies that ?val0 and ?val1 must both be of type *object*.

Influence of Action on Invariant Candidate In order to carry out the regression and the satisfiable test, it is first checked whether the current invariant candidate could be made false by an action. This

is done with a simple check. An action contains *add* and *delete* effects just like in a STRIPS planning task. First, iterate over all literals in the invariant candidate. If the current literal is negated, then it is checked whether the predicate of this literal can be found in the add-effects of the action. If the current literal is not negated, it is chekced whether the predicate can be found in the delete-effects of the action. If a predicate is found, the action has an impact on the invariant candidate and $c\sigma$ can be built.

Instance $c\sigma$ $c\sigma$ is a substitution, where a schematic formula is converted into a grounded formula. While a schematic formula only contains variables like ?x0, ?x1, a grounded formula contains explicit objects. The schematic formulas apply to all possible objects, while the grounded formula only applies to the explicit objects. A schematic formula of the form *on(?x0, ?x1)* $\vee$ *ontable(?x0)* applies to all possible instances of ?x0 and ?x1. The predicates *on* and *ontable* both expect objects of type *object*, which are defined in the PDDL task. Assuming *a* and *b* are the only objects of type *object*, the following instances can be generated: *on(a,b)* $\vee$ *ontable(a)* and *on(b,a)* $\vee$ *ontable(b)*. The regression and the satisfiable test are then carried out on these instances. If the result of the satisfiable test is true, the schematic invariant is removed from the set of all invariant candidates and an invariant candidate that has been weakened is added back to the set.

4.3 Weakening in Algorithm

Weakening is the biggest hurdle in the algorithm. Simple weakening by adding a single literal works, but does not produce a useful result. To make the weakening more meaningful, not only is a literal added, but several new invariant candidates are generated, which receive the same literals as before the weakening plus a new literal. New literals are randomly generated, but with the restriction that the literal does not already exist in the invariant candidate. This generates a list of literals that are not yet contained in the invariant candidate. Then as many new invariant candidates are created as there are new literals found. As a result, not only a new candidate is discovered through the weakening, but a large number of new candidates, which are then passed again through the satisfiable test. Here is the problem from when exactly an invariant candidate cannot be further weakened. One possibility is that an invariant candidate may contain at most two literals. Thus, only new literals are added in weakening if the invariant candidate consists of only one literal.

Another form of weakening is achieved by adding inequalities. The invariant candidate $on(X, Y)$ could thus be weakened by an inequality stating that $X \neq Y$ must be true for $on(X, Y)$ to hold. So that the algorithm does not weaken too much on one step, the inequality is only introduced for invariant candidates that already consist of two literals, i.e. cannot be weakened by adding more literals. However, the maximum number of inequalities must also be defined. A simple way is to generate as many inequalities as there are variables in the candidate invariant. This means that an invariant candidate can no longer be weakened if it already contains two literals and has reached the maximum number of inequalities. At this point $\emptyset$ can be returned.

Since the algorithm always weakens an invariant candidate if the satisfiable test is successful, the list of invariant candidates does not become shorter. If the satisfiablility test was not successful, the candidate is not removed from the list and tested again at a later iteration. In order to reduce the list of invariant candidates, the case where $\emptyset$ is returned must be implemented correctly during weakening, otherwise the algorithm will quickly get stuck in an endless loop.

4.4 Implementation of the Algorithm in Python

The algorithm was implemented in Python as part of this work in the planning environment Fast Downward. Fast Downward has already a function implemented, which explores the task. This function returns the propositional action occurring in the task and the fluent ground atoms. A propositional action consists of a name, a list of preconditions, a list of effects that can be divided into add-effects and delete-effects, and the cost of the action. The algorithm for creating schmeatic invariants receives the propositional actions , the fluent ground atoms and the task itself as parameters. First, all types that occur in the task are stored in a map with their associated subtypes.

Example: type *truck* contains variables *truck1* and *truck2*, type *package* contains variables *p1* and *p2*. *Truck* and *package* are both supertypes of type *object*. So the type *object* contains the variables *truck1*,

truck2, *p1* and *p2*. In the next step, these types are parsed into the respective predicate. The predicate *at* expects a variable of type *location* as parameter 1 and a variable of type *truck* as parameter 2. This parsing creates a map that can check what types a given predicate expects. Before the algorithm starts, the first lines for the tptp file are generated in typed first order logic. All types and their associated supertypes are defined in the map. Each line in the tptp file that defines a type can already be prepared, since the types no longer change within the algorithm.

Example: The task contains types *object* and *truck*, and *truck* is of type *object*. So the two lines can be prepared:

- tff(object_type, type, object : $tType). and
- tff(truck_type, type, truck : object).

The predicates for the tptp file can also be prepared in advance. For example, the task contains the predicate *on* which expects two variables of type *object*. Then the line can be prepared:

- tff(on_type, type on: (object * object) > $o)

These tff lines can then be added to a list, which is then always written to the tptp file first, before the axioms and the negated_conjecture are added. The invariants can then be generated, which are true in the init state. This is done using the counting method explained in the previous section. Once the invariant candidates have been created, the algorithm is started. A queue is used instead of a list of invariant candidates because the invariant candidates are modified at runtime and during iteration over all candidates. Candidates are removed and also added. When iterating over a list, Python throws an error if the size of the list is changed during an iteration. This problem is avoided with a queue. An action and an invariant are chosen, and if the action has no effect on the invariant, then the next invariant is chosen. This happens until the action has an impact on the invariant candidate. If this is the case, a set of substitutions of the invariant candidate is formed by grounding the currently invariant to all possible variables existing in the task. Each of these grounded invariants are then passed through the regression function depending on the action.

Each result of the regression is then checked for satisfiability. This happens with Vampire theorem prover. To do this, the tptp file is created and the lines that have already been prepared are first written to the file, then all currently invariant candidates are added as axioms and the result of the regression is added as negated_conjecture. In this way it is tested whether the result of the regression is satisfiable, given the previous invariant candidates. If this is the case, then the original schematic invariant, which was previously grounded, is passed through the weakening. The weakening then returns a set of new invariant candidates, in schematic form. Each of these weakened invariant candidates are then added to a set called *seen_invariant_candidates* so that it can be checked whether a certain candidate has already been checked or created. This prevents a candidate from being created over and over again.

Since the implementation uses a queue for the invariant candidates, each invariant candidate is removed from the queue. If the action has no effect on the candidate, the candidate is simply added back to the queue so that the candidate can be tested again with another action. The candidate will also be added back to the queue if the satisfiability test fails. By doing this, the candidate is checked again at a later point in time with another action. If the queue has not changed after iterating over all actions and candidates in the queue, i.e. either that no action has an influence on the candidates, or that the satisfiable test fails for all candidates, then the queue with the invariant candidates is returned as a solution of the algorithm. All candidates which are then still in the queue are valid invariants.

Finally, the algorithm was tested on all optimal strips problems on sciCORE. Due to this extension to all these tasks, new errors occurred that did not occur with previously selected tasks. For example, special cases had to be built in, which either ignored tasks, or the parsing between objects and their possible associated types had to be approached anew. Due to the large number of different tasks, the domains of the tasks also differed greatly. Some domains, such as the *storage* problem, therefore had to be completely removed from the analysis.

5 Results of Algorithm

The result of the algorithm, i.e. the invariant candidates obtained, are shown on a concrete Blocks World problem. The example task has 8 objects or blocks of the type *obejct* called a, b, c, d, e, f, g and h. The init state is defined as follows:

clear(a) $\wedge$ *clear(d)* $\wedge$ *clear(b)* $\wedge$ *clear(c)* $\wedge$ *ontable(e)* $\wedge$ *ontable(f)* $\wedge$ *ontable(b)* $\wedge$ *ontable(c)* $\wedge$ *on(a,g)* $\wedge$ *on(g,e)* $\wedge$ *on(d,h)* $\wedge$ *on(h,f)* $\wedge$ *handempty()*

The goal state is defined as follows:

on(d,f) $\wedge$ *on(f,e)* $\wedge$ *on(e,h)* $\wedge$ *on(h,c)* $\wedge$ *on(c,a)* $\wedge$ *on(a,g)* $\wedge$ *on(g,b)*

At the start of the algorithm, the invariant candidates from which are true in the init state are generated by counting. This leads to the following invariant candidates:

- Invariant Candidate: handempty() [ ]
- Invariant Candidate: ?x0 $\neq$?x2 $\rightarrow$ $\neg$on(?x0, ?x1) $\vee\neg$on(?x2, ?x1) [?x1: object, ?x0: object, ?x2: object]
- Invariant Candidate: $\neg$holding(?x0) [?x0: object]
- Invariant Candidate:?x1$\neq$?x2 $\rightarrow$ $\neg$on(?x0, ?x2) $\vee\neg$on(?x0, ?x1) [?x1: object, ?x0: object, ?x2: object]

With these invariant candidates and any justified actions included in the task, the algorithm finds the following invariant candidates:

- Invariant Candidate 1: ?x0$\neq$?x2 $\rightarrow$ $\neg$on(?x0, ?x1) $\vee\neg$on(?x2, ?x1) [?x1: object, ?x0: object, ?x2: object]
- Invariant Candidate 2: ?x1$\neq$?x2 $\rightarrow$ $\neg$on(?x0, ?x2) $\vee\neg$on(?x0, ?x1) [?x1: object, ?x0: object, ?x2: object]
- Invariant Candidate 3: handempty() $\vee\neg$on(?var0, ?var0) [?var0: object]
- Invariant Candidate 4: $\neg$holding(?x0) $\vee\neg$clear(?x0) [?x0: object]
- Invariant Candidate 5: $\neg$holding(?x0) $\vee\neg$on(?var0, ?x0) [?var0: object, ?x0: object]
- Invariant Candidate 6: $\neg$holding(?x0) $\vee\neg$ontable(?x0) [?x0: object]
- Invariant Candidate 7: $\neg$on(?x0, ?x0) $\vee\neg$holding(?x0) [?x0: object]
- Invariant Candidate 8: $\neg$on(?x0, ?var0) $\vee\neg$holding(?x0) [?var0: object, ?x0: object]

The invariant candidates are considered as one large disjunction. So it applies that either invariant candidate 1 $\vee$ invariant candidate 2 $\vee\cdots\vee$ invariant candidate 8. No matter what state the problem is in, one of these invariants is true at that point. For example, if the hand is not holding any block, Invariant Candidate 3 holds. If a block is either not on table, or the block is not in the hand, then Invariant Candidate 6 holds. For any reachable state, one of these invariant candidates holds.

5.1 Reduction of the Task

As already described in the definitions, a task can be reduced using the limited instantiation and Theorem 1. If a schematic candidate is invariantly falsified with a grounded action set that contains all grounded actions from the task, it is also falsified with a significantly smaller set, if possible. Whether the set can be reduced is defined in Theorem 1.

The reduction of the task was also provided in the implementation. First, the Limited Instantiation is calculated as specified in the definition. This returns the value of the minimum number of objects of each type, that must be present in the task in order to still find correct invariants. In the next step,

objects are removed from the task until the task contains exactly as many objects of a given type, as specified in the limited instantiation. As mentioned in the limited instantiation, the reduced task, must still contain all fixed objects. In Fast Downward, constants are defined in the domain-file of a given pddl-problem. For this thesis, the task was expanded to include a field that now contains the constants. When reducing the task, it is checked whether the object to be removed is contained in the constants, and if this is the case, this object is not removed. This ensures that the reduced task still contains all constants even after the reduction.

After the task has been successfully reduced by the certain number of objects, limited grounding is applied. For this purpose, each schematic action in the task is grounded with all remaining objects. The comparison between the reduced task (limited grounding) and the normal task was done with sciCORE (see Acknowledgements for more information).

Applying the algorithm to multiple tasks shows that Rintanen's approach works. The algorithm finds invariant candidates that make sense and the limited grounding or the reduction of the task also works as described by Rintanen in his paper. However, due to the large number of different tasks, the evaluation via sciCORE has shown that the implementation is faulty and cannot be applied to all pddl files. This is due to the differently defined objects, predicates and actions that are defined in the domain of the task, as well as the requirements that the task needs. Many tasks could still be evaluated through error correction, but unfortunately there are some for which the evaluation fails.

5.2 Comparison: number of grounded actions in task

Figure1 shows the comparison between the number of grounded actions in the normal task and the grounded actions in the reduced task. A total of 323 tasks contain a small number of grounded actions when grounded. All other tasks have the same number of actions, whether the task is collapsed or not. All tasks lying on the diagonal contain exactly the same number of actions for the normal and the reduced task. The tasks located on the borders, are the ones which were not successful in sciCORE. Errors were vampire type errors, memory errors and normalize exceptions which are further described in the conclusion.

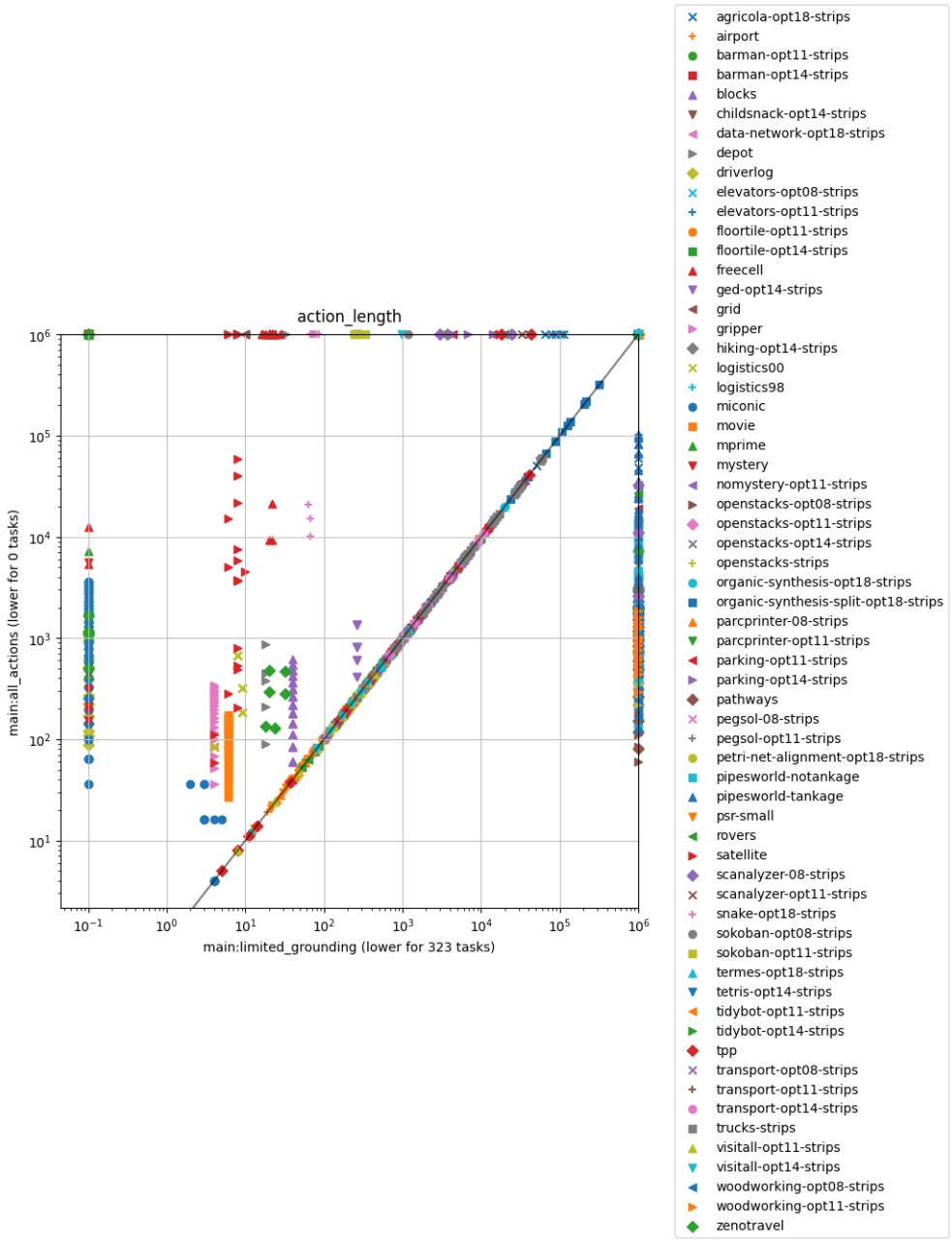

Figure 1: Comparison: Number of grounded actions in task

5.3 Comparison: runtime of algorithm

Figure 2 shows the runtime of the algorithm, i.e. for the creation of the invariant candidates compared to the normal task to the reduced one. A total of 245 reduced tasks on which limited grounding was applied ran through faster than the normal tasks. However, there are also normal tasks that run faster than the reduced tasks. A total of 107 normal tasks had a better runtime than the reduced tasks. The same applies here: all tasks that lie exactly on the diagonal have the same runtime, whether reduced or not. The tasks located on the borders, are the ones which were not successful in sciCORE. Errors were vampire type errors, memory errors and normalize exceptions which are further described in the conclusion.

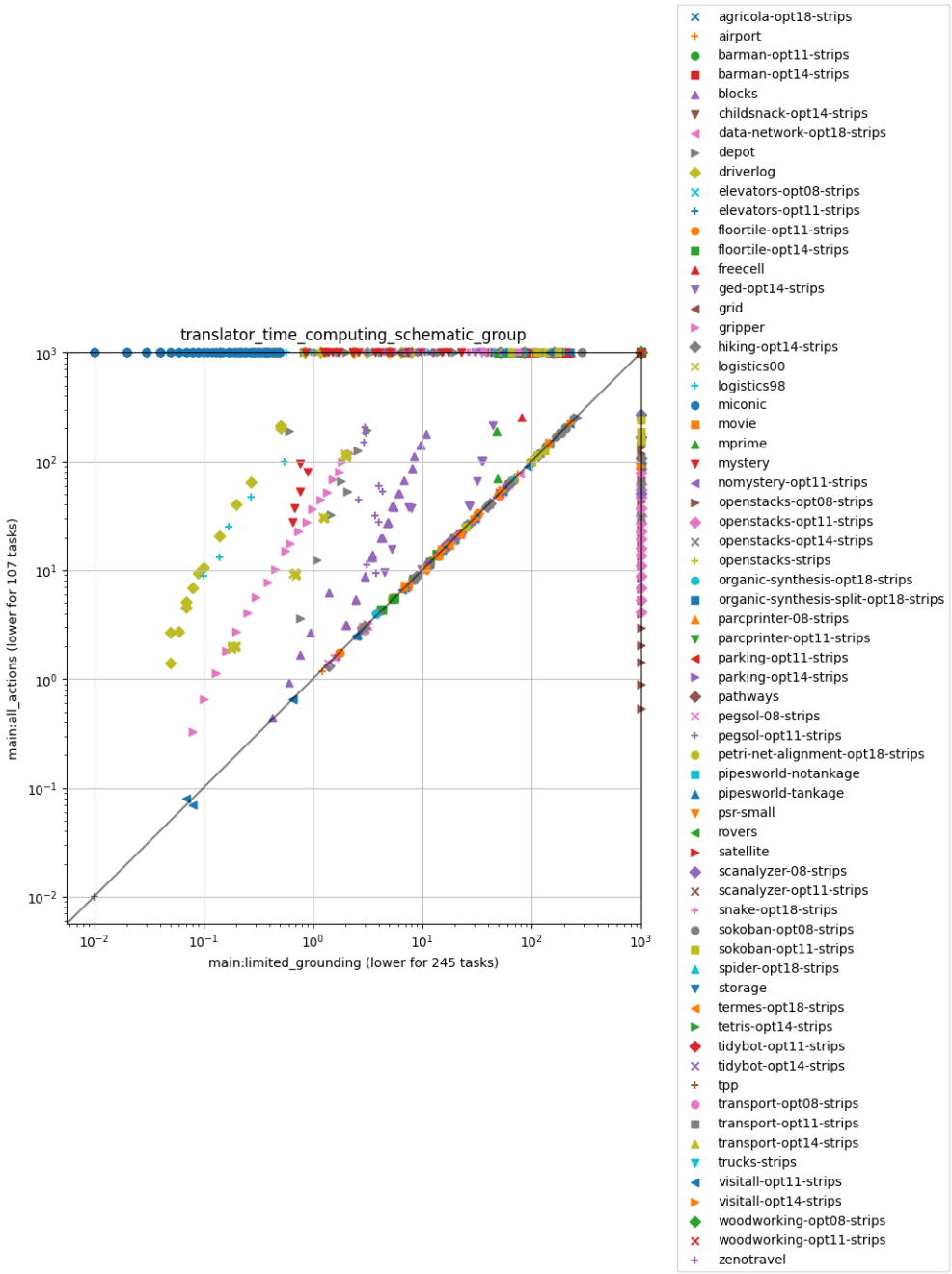

Figure 2: Comparison: Runtime of algorithm

6 Conclusion

Both figures show that Rintanen's approach works, but the implementation of the algorithm in Fast Downward is flawed. The reasons for this are detailed in the Conclusion. Rintanen's approach is confirmed by the fact that the task with limited grounding uses fewer schematic actions and finds the invariant candidates in a faster runtime. The exact number of tasks where this applies can be found in the figures. However, there are also normal tasks that had a better runtime than the task with limited grounding, which is because the limited grounding task still has to ground the action, if not all of it. Nevertheless, the original task receives all grounded actions without having to create them yourself, which leads to better runtimes in some tasks. However, the number of these tasks is smaller than the number of limited grounding tasks that had a better runtime.

In this thesis, the algorithm from Rintanen's paper *Schematic Invariants by Reduction to Ground Invariants* was implemented in the Fast Downward planning environment. All necessary definitions of the algorithm were shown and a short introduction to Fast Downward was given. Furthermore, the reduction of a task was discussed, which removes objects from a given task in order to significantly improve the runtime of the algorithm if the number of objects in the task is high.

The algorithm was tested with most optimal-strips-tasks, which are located in Fast Downward. There were some problems, which were excluded from the experiment, since they won't work with the current implementation. The first one is *storage*-task. This task includes a Typedobject ?x of type *either crate* or *storearea*. The algorithm implemented in this thesis, don't consider Typedobjects where a type has to be choosen, but only Typedobjects with a fixed type. The other one is *spider*-task. If this task is started, a *negated_ conjecture* is created which defines a variable X as type*card*. The formula contains a predicate *clear* and a predicate *in-play*. *clear* expects a variable of type *cardposition* and *in-play* expects a variable of type *card*. Since *clear(X)* and *in-play(X)* are both present in the formula, vampire stops with an error. *card* is a subtype of *cardposition*, thats why it is possible to pass to both predicates a variable of type *card*, but vampire have problems to understand this connection between the type and the subtype. This is due to the fact that the hierarchy in this example is as follows: *card* → *card_ or_ tableau* → *cardposition*. Since *card* is the subtype of the subtype of *cardposition*, vampire stops with an errorcode. This vampie specific error, where the types werent applicable in vampire occured over multiple tasks. At the beginning of the thesis I worked with first order formula files in vampire. The conversion to typed first order formula files was only implemented at a later point. This change made it possible to process significantly more tasks in a meaningful way, since each variable now had a type and each predicate expects a certain type. However, since vampire has problems converting several types into one variable, i.e. it could not distinguish between types and subtypes, a large number of tasks failed despite the switch to *tff*.

Another phenomenon is that tasks on sciCORE throw an error stating that the maximum recurrence depth has been exceeded when accessing a Python object. This is defined in Python as RecursionError. In local tests with exactly the same task, as the same domain files and the same problem files, the algorithm ran through without errors, and no recursion error or similar occurred. This is the case, for example, with *titdybot-opt11-strips* with problem-file *p10.pdd*. This task throws the RecursionError on sciCORE and locally it instantiates 19,255 grounded actions with limited grounding and calculates the schematic invariant candidates in 5,676s. Locally 20 invariant candidates are found. Some tasks also fail because of insufficient memory. This leads to a NormalizeException on sciCORE, since Python cannot recover from the memory errors.

The results of all tasks, were shown in the previous section. Thanks to the evaluation, done with the help of sciCORE, it is easy to see, that the algorithm of Rintaten works, as well as the reduction of the task. Therefore it is safe to say, that if one want to use a schematic invariant algorithm, the task should be reduced, to improve the runtime and memory used. Since in the reduced tasks, there are overall less grounded actions which have to be checked, the algorithm referring to the reduced task need less memory than without limited grounding.

In a future work, it could be investigate whether the planning environment Fast Downward can solve a task faster with these new obtained schematic invariants than with the already implemented version of grounded invariants. Furthermore, it would be interesting, which approach is faster and needs less memory, including the calculation of the invariants. One part of a future work would focus on the compar-

ison between only the search algorithm with either schematic or grounded invariants, and the other part would focus on comparison between translation as well as the search algorithm. If the search algorithm runs faster with schematic invariants than with grounded, but the translation needs a lot more time for schematic invariants, it could still be useful to work with the grounded invariants, since the overall time would be more effective. This is just a hypothesis and could be investigated in a future work.

Another topic would be the adaption of the algorithm to all PDDL-tasks which occur in Fast Downward. This would include the already mentioned excluded tasks (spider, storage) and as well the more complex tasks and tasks which have failed while testing on sciCORE. In this thesis only optimal-strips tasks were considered, but Fast Downward also supports tasks called which are satisficing. They include for example tasks *tetris-sat11-strips, sokoban-sat11-strips*. The optimal-strips tasks on the other hand includes the tasks *tetris-opt11-strips, sokoban-opt11-strips*. This adaption to the *sat*-tasks would be an improvement to the current implementation done in this thesis.

7 Acknowledgements and References

I would like to express my gratitude to my supervisor, Gabriele Röger, who helped me in understanding Rintanen's algorithm and also in implementing the algorithm in Fast Downward. Thanks to her, I was finally able to make an evaluation despite the great effort involved in implementing the algorithm.

Calculations were performed at sciCORE (http://scicore.unibas.ch/) scientific computing center at University of Basel. This refers to Figure1 and Figure2, in the section *Results of Algorithm*

References

Aeronautiques, C., Howe, A., Knoblock, C., McDermott, I. D., Ram, A., Veloso, M., Weld, D., SRI, D. W., Barrett, A., Christianson, D., et al. (1998). Pddl| the planning domain definition language. *Technical Report, Tech. Rep.*

Barwise, J. (1977). An introduction to first-order logic. In *Studies in Logic and the Foundations of Mathematics*, volume 90, pages 5–46. Elsevier.

Helmert, M. (2006). The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246.

Helmert, M. (2009). Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence*, 173(5-6):503–535.

Helmert Malte, R. S. (2003). Fast downward. `https://www.fast-downward.org/HomePage`. Accessed on 2023-07-14.

Kovács, L. and Voronkov, A. (2013). First-order theorem proving and vampire. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*, pages 1–35. Springer.

Lipovetzky, N. (2014). *Structure and inference in classical planning*. Lulu. com.

Rintanen, J. (2008). Regression for classical and nondeterministic planning. In *ECAI 2008*, pages 568–572. IOS Press.

Rintanen, J. (2017). Schematic invariants by reduction to ground invariants. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31.

Saiz, V. A., y Dra, D. D. B. M., and Fernández, D. S. (2014). *Generation and exploitation of intermediate goals in automated planning*. PhD thesis, Universidad Carlos III de Madrid.

Sutcliffe, G. and Kotelnikov, E. (2018). Tfx: The tptp extended typed first-order form. In *PAAR@ FLoC*, pages 72–87.

Sutcliffe Geoff, S. C. (2001). The tptp problem library for automated theorem proving. `https://www.tptp.org`. Accessed on 2023-06-27.

Voronkov, A. (1990). Vampire theorem prover website. `https://vprover.github.io`. Accessed on 2023-06-27.

Universität Basel

Philosophisch-Naturwissenschaftliche Fakultät

Erklärung zur wissenschaftlichen Redlichkeit und Veröffentlichung der Arbeit (beinhaltet Erklärung zu Plagiat und Betrug)

Titel der Arbeit:

Name Beurteiler*in: Dr. Gabriele Röger

Name Student*in Lukas Vogel

Matrikelnummer: 19-051-580

Mit meiner Unterschrift erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe. Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Ort, Datum: Reiden, 19.07.2023 Student*in: L. Vogel

Wird diese Arbeit veröffentlicht?

(●) Nein

( ) Ja. Mit meiner Unterschrift bestätige ich, dass ich mit einer Veröffentlichung der Arbeit (print/digital) in der Bibliothek, auf der Forschungsdatenbank der Universität Basel und/oder auf dem Dokumentenserver des Departements / des Fachbereichs einverstanden bin. Ebenso bin ich mit dem bibliographischen Nachweis im Katalog SLSP (Swiss Library Service Platform) einverstanden. (nicht Zutreffendes streichen)

Veröffentlichung ab: _____

Ort, Datum: Reiden, 19.7.2023 Student*in: L. Vogel

Ort, Datum: _____ Beurteiler*in: _____

Diese Erklärung ist in die Bachelor-, resp. Masterarbeit einzufügen.