FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Ivan Kuckir

Exploiting Structures in Automated Planning

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	prof. RNDr. Roman Barták, Ph.D.
Study programme:	Computer science
Study branch:	Theoretical Computer Science

Prague 2016

Title: Exploiting Structures in Automated Planning

Author: Bc. Ivan Kuckir

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: prof. RNDr. Roman Barták, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: This thesis focuses on improving the process of automated planning through symmetry breaking. The aim is to describe symmetries, which are often observed by human programmers, but haven't been properly theoretically formalized. After an analysis of available research, there are new definitions of symmetries proposed in context of classical planning, such as state equivalence, T1 automorphisms and more general automorphisms of constants. Several theorems are proved about new symmetries. As a result, an algorithm for detecting a special symmetry class is proposed, together with a method of exploiting such class during planning. Experimens are made to show the effect of symmetry breaking on the performance of the planner.

Keywords: automated planning symmetry breaking relation automorphism equivalence

I would like to thank my supervisor, prof. Roman Barták, for introducing me to the area of automated planning and for his precious advice and help, which I received during the work on this thesis. I am also very grateful to my friends, colleagues and parents, who supported me and encouraged me in my research.

Contents

Introduction

Automated planning is a branch of artificial intelligence, that studies the process of obtaining action sequences, which lead to fulfilling some goal. Such action sequences can be executed by robots, driverless cars and other machines. It can also be used to generate efficient plans for activities, which can be performed by people. Automated planning has gained a lot of interest and popularity in recent years.

The International Planning Competition, IPC, is an event, where automated planners compete with each other in solving many different problems. The special format, Planning Domain Definition Language (PDDL), was created for representing planning problems during the competition. Sample problems from the IPC in the PDDL format are publicly available at `https://helios.hud.ac.uk/scommv/IPC-14/`.

PDDL format can represent many completely unrelated problems. It has a rich structure, which allows defining both simple and also very complex models. It can be used for modelling real-life situations, but also completely fictional problems, which are made just for testing the automated planners during the IPC.

We can split the area of planning into stochastic planning (the effect of actions is not precisely given, different effects may happen with different probability) and deterministic planning (we know exactly the effect of each action). It can also be split into online planning (the goal may change during the process of planning, planning and the execution of the plan are performed simultaneously) and offline planning (we know the goal in advance, we can construct the entire plan and then execute it). In this thesis, we focus on deterministic, offline planning.

There are many areas of the research in automated planning.

Study of heuristics Heuristics say "the solution is probably this way". They may also focus on different strategies, used in the plan search, machine learning techniques etc.

Smart heuristics may let us solve problems, which are impossible to solve using other techniques. Analysis of previous plans may help us generate strategies, which would help finding future plans.

The efficiency of such methods may vary significantly from a problem to a problem. While some heuristic is efficient for one class of problems, it may be inefficient for another class. It is usually hard to estimate the result of application of heuristic method on a specific problem.

Knowledge of the domain These methods say "the solution is definitely (or definitely isn't) this way". This category contains the research about symmetries, partial order reduction techniques, domain control knowledge, control rules etc. Useful methods can be found for specific classes of problems, but it is hard to find methods, that would be helpful in many classes of distinct problems.

We can also put automatic planners for specific problems into this category. When programmers have to solve a specific kind of problem, they usually build specific rules into the planner, in order to make the search faster and avoid action sequences, which definitelly don't lead to the solution. Such rules are used ad-hoc. It is hard to study them from the broad perspective.

The typical example of such method is Automated Transformation of PDDL Representations (Riddle et al. [2015]). Authors describe a method, which detects similar objects inside a problem and edits the representation, such that the search space is smaller for a blind algorithm. Their method is demonstrated on several problems, but it has too many restrictions.

Modern automated planners usually combine both heuristics and the knowledge of the domain.

In this thesis, we will not examine heuristics, planning strategies and other similar tools, which try to predict the shortest path to the goal. We will take a closer look at symmetries. We will analyze existing techniques, talk about their advantages and disadvantages. Then we will introduce the new approach to symmetries, which can generalize and simplify some of the previous work in that field. As a result, we will get a general algorithm for detecting symmetries during the search.

1. Background

Many different models and representations were made in the field of automated planning in the past. For example, the propositional representation (also called *STRIPS*) used to be very pupular, see Fikes and Nilsson [1971]. It was used in the automated planner with the same name.

1.1 Basic terminology

The definitions, that we are going to use, correspond with the classical planning, described in Ghallab et al. [2004].

We start with a language of predicate logic with K predicate symbols and L constant function symbols (also called *objects*). Possible terms are either variables or constants. An atom (atomic formula) has a form of $p(t_0, t_1, ..., t_m)$, where p is a predicate symbol and t_i are terms (variables or constants).

Substitutions may be applied to atoms. Substitution is called "grounding", when variables are replaced by constants, and "lifting", when constants are replaced by variables. A ground atom contains constants only, a lifted atom contains variables only.

A state is a set of ground atoms. A literal is an atom or a negation of an atom.

An operator is a tuple

$$o = (name(o), precond(o), effects(o))$$

where name(o) is a name of an operator, precond(o) is a set of precondition literals and effects(o) is a set of effect literals (both sets contain lifted literals only). Parameters of an operator are all variables, that occur in its precondition or effect literals.

For a set of literals L, L^+ denotes the subset of all (positive) atoms inside the L, L^- denotes the subset of atoms, that are negated instide the L.

An operator can be converted into an action by applying a grounding substitution of its parameters to its preconditions and effects.

An action a is applicable to a state s, when

$$precond^+(a) \subseteq s, precond^-(a) \cap s = \emptyset$$

In such case, we can define a transition function:

$$\gamma(s, a) = (s - effects^-(a)) \cup effects^+(a)$$

Having a planning domain (constants, predicates and actions), we may define a planning problem as a pair (s_0, g), where s_0 is a state called an initial state and g is a set of ground literals.

A finite sequence of actions $(a_0, a_1, ..., a_n)$ is called a solution (a plan) for the problem (s_0, g), if

$$s^* = \gamma(\gamma(...\gamma(s_0, a_0)..., a_{n-1}), a_n)$$
$$g^+ \subseteq s^*, g^- \cap s^* = \emptyset$$

For a planning domain, let's define a *transition graph.* It is an oriented graph, vertices are all possible states. There is an oriented edge from the vertex s_0 to the vertex s_1, iff there exists an action a applicable to s_0, such that $\gamma(s_0, a) = s_1$. Such edge can be labeled with that action. Note, that the transition graph does not have to be a tree. There can be cycles, i.e. by performing some sequences of actions at one state we may get to the same state. It does not have to be connected either. Some states can be unreachable from another states.

Assume there is a cost function, which attaches a natural number to each action. We call a plan $(a_0, a_1, ..., a_n)$ optimal for a problem (s_0, g), if the plan cost $\sum_{i=0}^{n} cost(a_i)$ is smaller or equal to costs of all other plans for that problem.

Let's define one more useful term. The predicate is called *rigid* in the domain D, if it does not occur in any effect of any action. Otherwise, we call it *fluent.*

Atoms of a rigid predicate, that hold in the initial state, hold in all reachable states. Rigid predicates are used to describe the constant properties of a system, that don't change over time. E.g. it can be graph adjacency in logistic domains, processed types of objects, that were taken from the PDDL representation, or the equality relation between constants.

From the implementation point of view, rigid atoms are usually removed from the representation of a state and stored separately. Authors usually store them inside an efficient data structure, which allows to do the search in a constant time.

1.2 Similarities with PDDL

The definitions, that we mentioned earlier, were the key to creating the PDDL language. PDDL was extended by lots of additional features, such as object types, built-in predicates, quantifiers, conditional effects and others. Many techniques of converting a PDDL problem into a simplified version were described by Helmert [2009].

One such technique is compiling away types. Each object in a PDDL problem has its type and each parameter of a predicate and an operator is restricted by a certain type. We remove types by converting each type t into a new unary predicate with a name is_t. For each object of type t we add an atom $is\_t(o)$ into the initial state (including inherited types of that object). For each operator, that requires parameters to have types $t_0, t_1, ...$, we add new positive preconditions $\{is\_t_0(p_0), is\_t_1(p_1)...\}$.

Another commonly used technique is removing negative preconditions of operators. When there is a predicate p, which occurs as a negative precondition of some operator, we create a new predicate not_p with the same arity. Each precondition $\neg p(x_1, \dots x_n)$ is replaced by a precondition $not\_p(x_1, \dots x_n)$. For each positive effect $p(x_1, \dots x_n)$ we add a negative effect $\neg not\_p(x_1, \dots x_n)$ and for each negative effect $\neg p(x_1, \dots x_n)$ we add a positive effect $not\_p(x_1, \dots x_n)$. We also add new atoms to the initial state s_0. $not\_p(x_1, \dots x_n)$ is added iff $p(x_1, \dots x_n) \notin s_0$.

Now we describe, how to compile away some other properties of PDDL, that were not mentioned by Helmert [2009].

Equality PDDL tasks often use a "built-in" equality predicate, e.g. there may be a preconditoin of an operator $= (p_i, p_j)$. We get rid of it by explicitly constructing a new relation. First, we create a new predicate "=". Then, for each object (constant) c we add an atom $= (c, c)$ to the initial state.

Other logical connectives Our definition of the operator allows having sets of literals, that represent a logical conjunction of these literals. Preconditions of a PDDL opterator may have a complex structure, using connectives such as disjunction, implication, brackets etc.

Every logical formula can be converted into the disjunctive normal form (DNF) $t_1 \vee t_2 \vee ... \vee t_n$, where t_i is a conjunction of literals. We replace such operator with n new operators, that have preconditions $t_1, t_2, ...t_n$ respectively, and the same effects, as the original operator.

1.3 Example

Let's have a look at the example of the planning domain and the problem. We will describe the Childsnack domain, which is used throughout this thesis.

The domain describes the process of serving sandwiches to children. There are several constants:

$$\begin{aligned}
&child1, child2, ...child10,\\
&bread1, bread2, ...bread10,\\
&content1, content2, ...content10,\\
&tray1, tray2, tray3,\\
&table1, table2, table3,\\
&sandw1, sandw2, ...sandw10
\end{aligned}$$

The domain has following predicates:

- at_kitchen_bread(bread) - what bread is availabe
- no_gluten_bread(bread) - bread, which has no gluten in it
- at_kitchen_content(content) - what content is available
- no_gluten_content(content) - content, which has no gluten in it
- at_kitchen_sandwich(sandw) - sandwiches at the kitchen
- no_gluten_sandwich(sandw) - sandwiches, which have no gluten in them
- notexist(sandw) - sandwiches, which are not created yet
- at(tray, place) - stores the location of each tray
- ontray(tray, sandw) - specific sandwich is on the specific tray
- waiting(child, place) - specific child is waiting at the specific table
- allergic_gluten(child) - true for gluten-allergic children
- not_allergic_gluten(child) - true for not gluten-allergic children

One sandwich is made using one piece of bread and one piece of content. It can be done by following operators:

$$\begin{aligned}
&make\_sandwich(s, b, c),\\
&prec : \{at\_kitchen\_bread(b), at\_kitchen\_content(c), notexist(s)\},\\
&eff : \{\neg at\_kitchen\_bread(b), \neg at\_kitchen\_content(c), \neg notexist(s),\\
&at\_kitchen\_sandwich(s)\}
\end{aligned}$$

$$\begin{aligned}
&make\_sandwich\_no\_gluten(s, b, c),\\
&prec : \{at\_kitchen\_bread(b), at\_kitchen\_content(c), notexist(s)\\
&no\_gluten\_bread(b), no\_gluten\_sandwich(s)\},\\
&eff : \{\neg at\_kitchen\_bread(b), \neg at\_kitchen\_content(c), \neg notexist(s),\\
&at\_kitchen\_sandwich(s), no\_gluten\_sandwich(s)\}
\end{aligned}$$

These operators "remove resources" - atoms such as at_kitchen_bread(b), at_kitchen_content(c), notexist(s), but they add an atom at_kitchen_sandwich(s).

Another operator lets us put a sandwich onto a tray.

$$\begin{aligned}
&put\_on\_tray(s, t),\\
&prec : \{at\_kitchen\_sandwich(s), at(t, kitchen)\},\\
&eff : \{\neg at\_kitchen\_sandwich(s), ontray(s, t)\}
\end{aligned}$$

Next operator lets us move trays between places (kitchen, table1, table2, table3).

$$\begin{aligned}
&move\_tray(t, p1, p2),\\
&prec : \{at(t, p1)\},\\
&eff : \{\neg at(t, p1), at(t, p2)\}
\end{aligned}$$

Last two operators represent serving sandwiches to children.

$$\begin{aligned}
&serve\_sandwich(s, c, t, p),\\
&prec : \{at(t, p), ontray(t, s),\\
&waiting(c, p), not\_allergic\_gluten(c)\},\\
&eff : \{\neg ontray(t, s), served(c)\}
\end{aligned}$$

$$\begin{aligned}
&serve\_sandwich\_no\_gluten(s, c, t, p),\\
&prec : \{at(t, p), ontray(t, s), no\_gluten\_sandwich(s)\\
&waiting(c, p), allergic\_gluten(c)\},\\
&eff : \{\neg ontray(t, s), served(c)\}
\end{aligned}$$

There is the following initial state:

$at\_kitchen\_bread(bread[1|\dots|10])$
$no\_gluten\_bread(bread[2|6|7|8])$
$at\_kitchen\_content(content[1|\dots|10])$
$no\_gluten\_content(content[2|5|6|8])$
$notexist(sandw[1|\dots|10])$
$at(tray1, kitchen), at(tray2, kitchen), at(tray3, kitchen)$
$waiting(child1, table1), waiting(child2, table1), waiting(child3, table1),$
$waiting(child4, table3), waiting(child5, table2), waiting(child6, table2),$
$waiting(child7, table1), waiting(child8, table2), waiting(child9, table2),$
$waiting(child10, table1),$
$allergic\_gluten(child[2|3|7|9])$
$not\_allergic\_gluten(child[1|4|5|6|8|10])$

The goal atoms are:

$$served(child[1|\dots|10])$$

This description of the domain is incomplete. When some variable occurs only in effect atoms, but not in precondition atoms, any constant can be used during grounding (converting operators to actions) and an action would be still applicable. Additional type restrictions should be added, so `bread1` or `child2` can not be used as places to move trays to, etc. For a complete description of Childsnack, see attached PDDL files.

Let's mention several properties of our problem. There are exactly 10 pieces of bread and content, which allow us to make exactly 10 sandwiches, and there are exactly 10 children, who need to be served. We can make at most four gluten-free sandwiches and there are exactly four children, who require gluten-free sandwiches. .

The structure of operators allows us to make a classic sandwich using gluten-free ingredients (once we make such sandwich, the goal is not reachable from that state). It also allows us to serve gluten-free sandwiches to children, who are not gluten-allergic. We also can serve mutliple sandwiches to a single child. Such behavior is valid, but it prefents reaching the solution.

1.4 Planning algorithms

Many different algorithms are used today for solving planning problems.

Forward search This method represents the very straight-forward approach to planning. It explores the state space as a transition graph from the initial state to neighboring states using actions.

Backward search Backward search can be viewed as the search in the opposite direction. It starts with a goal state and tries to get into the initial state. It usually works with a partial representation of states, when it is not clear, if some atom is or is not in that state (e.g. in the goal state).

Partially ordered plans It is an approach, where the plan in the result consists of partially ordered actions. Any linearization of such plan creates a correct linear plan. Partial plans are useful, when the execution of actions in practice can be performed in parallel. The search can also be done by applying actions in parallel: we apply multiple actions to get to another state. The cost of such plans is usually defined as the *makespan*: the cost of the most expensive oriented path from the initial state to the goal state. When the cost of each action corresponds to the time of execution of that action, Makespan corresponds to the total time, when executing the plan in parallel, while the classical cost corresponds to the total time, when executing the plan sequentially. Algorithms, which generate partial plans (e.g. Graphplan) usually find optimal plans in terms of the makespan.

Reduction to other problems Planning problems are often reduced to other problems, e.g. into solving the constraint-satisfaction problem (CSP) or the problem of propositional satisfiability (SAT). Such planners benefit from the tremendous amount of work, that was spent e.g. on creating an efficient SAT solver, instead of inventing new methods for planning.

Examining Forward search In this thesis, we focus on automated planning based on forward search. The basic principle of the forward search is exploring states, one by one, and checking, if goal literals are satisfied.

It corresponds to a classic graph search of the transition graph for a specific domain. We have to find the shortest path from the initial state to some state, that satisfies goal literals. The arcs of such path, that correspond to actions, define a plan.

Let us write the classic A* algorithm, that is adjusted for planning purposes. We implement so-called "lazy deletion" from the queue. We allow the same state occur multiple times in the priority queue, which may lead to higher memory requirements. On the other hand, we don't need to check for each newly generated state, if it was generated before, which may lead to higher performance.

Many other algorithms for graph search may be used for planning, e.g. iterative depth-first search. The limit of DFS is the plan cost, which may not correspond to "depth". We still have to record visited vertices, to be sure, that

Algorithm 1 A* algorithm for planning

```
function ASTAR(Dom, S0, goal, heur)
    Q ← {} ; Q.PUSH(S0, heur(S0, goal) );          ▷ priority queue
    visited ← {}                                   ▷ visited states
    S0.cost ← 0
    while Q $\neq \emptyset$ do
        S ← Q.POP()
        if visited[S] = true then CONTINUE
        visited[S] ← true
        if SATISFIES(S,goal) then return PLAN(S)
        Acts ← ALLACTIONS(S, Dom)
        for all A $\in$ Acts do
            S2 ← GAMMA(S,A)
            S2.prev ← S;        ▷ predecessor, to build a complete plan
            S2.cost = S.cost + COST(A)
            Q.PUSH(S2, S2.cost + heur(S2, goal))
    return null
```

we have explored all reachable states, if no plan exists. Such record can be used to avoid cycles.

Forward search algorithms gradually explore the whole state space. The time complexity of the forward search algorithm usually corresponds to the number of analyzed vertices.

1.5 Domain-dependent and independent planners

In the context of automated planning, scientists often mention domain-dependent and domain-independent planning to describe a specific approach used inside a planner. The difference between domain-dependent and domain-independent planners may be formulated in many ways. This is a very frequent topic of discussion among scientists. Let's describe the usual formulations of domain-dependent and independent planning techniques.

To be able to categorize planners, we must first specify a problem, which we want to solve. Our problem is a planning problem described above, an initial and a final state. The solution is an optimal or near-to-optimal plan. The quality of a planner is measured by the time that the planner needs to find a plan.

First, let's consider a standard approach with a forward search, e.g. using A* algorithm. The planner is guaranteed to find a plan for any problem (with the initial state and the goal), that can be described inside a domain, if such plan exists. This approach can be called *domain-independent.* Planner can take any input and give an output.

In practice, domain-independent planners may perform actions, that seem to be useless from a human perspective. The planner often explores the part of state space, which does not lead to the solution. That exploration takes a lot of time.

Because of this, people often create their own planners for specific domains and specific classes of problems (specific initial and goal states). Usually, the goal is a set of atoms of one specific predicate. Such planners have restrictions, which should limit the search (by avoiding useless behavior) and make the state space smaller.

As a result, these planners are limited to a specific initial state and a specific goal as an input. They can not solve any possible problem, which can be described within a domain. These planners are called *domain-dependent*. They can be implemented in a similar way, as a domain-independent planner, where additional knowledge is added as the part of an input, so the "smartness" of a domain-dependent planner does not have to be hardcoded during its construction. Such planners are very fast and are massively used in practice Baier et al. [2007].

However, there may be planners, which use DCK when they detect a specific structure of the initial state and the goal, and use an uninformed search otherwise. In such case, it is not clear, if we should call them domain-dependent or independent. Describing domain-independent planners as planners, that can work with any input, is a little misleading.

2. Symmetry

In the area of a combinatorial search, the word "symmetry" usually denotes the property of a search space, where several branches or subtrees are similar (isomorphic, symmetrical), and searching through all of them is redundant. We can restrict the search just to one of several symmetrical branches, without any impact on reachability and the quality of the solution.

2.1 Available research

Object symmetry in the initial state Authors of the following paper Fox and Long [1999] were among first, who analyzed symmetries in planning. They defined symmetric objects to be those, which are indistinguishable from one another in terms of their initial and the final configurations. They implemented symmetry detection into the Graphplan algorithm.

When applying two actions to a state, if these actions differ just in a single constant and these constants are symmetric, it is enough to try just one of actions (trying the other action would result into a "symmetrical" behavior). Besides the informal definitions and the thorough description of the implementation, authors did not introduce any advanced theoretical formalization.

The problem of this approach is, that it defines equivalent objects only according to the initial and the final state. There may be more equivalency classes and symmetrical branches during the search, which are not described in the initial and the final states.

Symmetries of the transition graph In Shleyfman et al. [2015], authors defined symmetries as automorphisms of a transition graph, which preserve the action cost between two symmetric pairs of states and the "goalness" of symmetric states. Such automorphism can be represented as a permutation of states $\sigma(s)$. Note, that the transition between states s_0, s_1 may be performed by a completely different operator, than the transition between $\sigma(s_0), \sigma(s_1)$, as long as the actions have the same costs.

It is easy to see, that a sequence of states $[s_0, s_1, ...s_n]$ is a plan iff the sequence of states $[\sigma(s_0), \sigma(s_1), ...\sigma(s_n)]$ is a plan, and that both plans have equal costs. However, such definition is very broad and does not give any clue about an algorithm of detecting such symmetries. Two states can be completely unrelated and be symmetric, just because there is the same number of steps to the goal (having the same cost) and a similar graph structure around them.

Authors also define the *structural symmetry*, which is simply a renaming (permutation) of operators $\sigma_o(o)$ and atoms $\sigma_a(a)$ (the article speaks about propositions in STRIPS formalism), such that:

$$\begin{aligned} precond(\sigma_o(o)) &= \sigma_a(precond(o)) \\ effects(\sigma_o(o)) &= \sigma_a(effects(o)) \\ C(\sigma_o(o)) &= C(o) \\ \sigma_a(goal) &= goal \end{aligned}$$

We can see, that the structural symmetry, when extended to states (sets of atoms), is a symmetry of transition graph. Unlike Fox and Long [1999], it still does not give any method of restricting the search by avoiding symmetric paths. Authors use these definitions to create a heuristic, which keeps its properties under symmetry.

Bagged representation. In Riddle et al. [2015], authors defined so-called *Bagged representation.* The main idea is to represent some items as they are located in so-called bags (where they are indistinguishable) instead of sets (where they are distinguishable). They have manually rewritten several PDDL domains, where they applied the bagged representation. These new PDDL domains performed much better than the original domains, while being tested with modern automated planners.

Authors also proposed a method for finding such bags (and generating the new bagged representation of a problem) automatically. At the initial phase, algorithm creates the bags of objects. Objects end up in the same bag, if they fulfill several conditions.

Objects should be action-equivalent. It means, that each action should treat each object from the bag in the same way. E.g. it can not refer to any specific object as a constant inside a precondition or an effect. They must also satisfy several type restrictions. The distribution into bags also depends on the initial and the goal states.

Their system converts the original PDDL representation into the new representation. Instead of having several atoms representing the same bag, they create a single atom, which represents the count of objects in a specific bag. Operators are also edited, so they take objects from bags or put them into other bags. The domain is extended with predicates and objects representing simple arithmetics in order to keep track of the count of items in each bag.

The last part of the system is the module, which translates the solution plan in the new representation into the original representation by mapping indistinguishable objects from bags to real ”named” objects in the original representation. In the later research, authors managed to get rid of dependincies on the goal state by creating an algorithm, that converts any goal state of the bagged repserentation to the goal state of the original representation.

Orbit Search Another, very significant definition of symmetries, was done by Pochter et al. [2011]. Their methods are defined for SAS+ formalism, which seems to be completely unrelated to our way of research. These methods also closely depend on the initial and the goal state. We will get back to them later in this thesis.

Possible disadvantages of current approach All methods, that we have mentioned previously, seem to be using different formalisms and the relation between them is not evident from the first sight.

These methods are also very closely related to the structure of the initial and the goal state. If we performed several useful actions on the initial state and created the new initial state, these methods would detect much smaller groups of equivalent objects in these new initial states, since the objects would occur in different predicates. These methods don't consider symmetries, which may arise in other states on the path to the goal.

We will try to create more general and simpler definition of the symmetry without any reformulation of the problem and with minimal computational overhead.

3. A new approach to symmetry

In a planning task, constants alone don't have any actual meaning. They are just some simple distinguishable items. The atoms (relations) give them the meaning, the power to describe a complex structure and properties of the state.

In the following text, we will try to describe the representation of states and actions, which is not dependent on specific constants. We will define the equivalence of constants, which makes several states or actions equivalent in some sense. This equivalence will allow us to prune the search space and speed up the search process.

State equivalence Having a state (a set of ground atoms), if we decide to rename all constants, giving them new unique names, we would still consider that new state to be the same "world state". It may resemble the renaming of variables during the unification of two formulas, which share some variables. Example:

```
S0 = { (ontray sandw1 tray1), (ontray sandw2 tray2),
       (no_gluten_sandwich sandw1) }
S1 = { (ontray sandw2 tray1), (ontray sandw1 tray2),
       (no_gluten_sandwich sandw2) }
```

Both S0, S1 describe the same state ("two trays, a regular sandwich on one tray, a gluten-free sandwich on another"), they just use different constants for the same fact.

Let's take a look at the sets S0 and S1 from the previous example and add two new atoms to each of them.

```
S0 = { (ontray sandw1 tray1), (ontray sandw2 tray2),
       (no_gluten_sandwich sandw1),
       (at_kitchen_sandwich sandw3), (at tray1 kitchen) }
S1 = { (ontray sandw2 tray1), (ontray sandw1 tray2),
       (no_gluten_sandwich sandw2),
       (at_kitchen_sandwich sandw3), (at tray1 kitchen) }
```

Two states still have the same meaning. If we decide to apply the action `put_on_tray(sandw3, tray1)` to states S0 and S1, we "do the same thing" - put a new sandwich on a tray containing a gluten-free sandwich. However, if we manipulate `sandw2` in S0, it corresponds to manipulating `sandw1` in S1.

Equivalence of constants Now, let's take a look at equivalence of constants. Assume the following state:

```
S0 = { (at tray1 kitchen), (at tray2 kitchen),
      (at_kitchen_bread bread1), (at_kitchen_bread bread2),
      (at_kitchen_bread bread3), (no_gluten_bread bread3)  }
```

Constants `tray1`, `tray2` are in certain sense equivalent. They occur just once, as the first parameters of atoms of the predicate `at`, and the remaining parameter (`kitchen` in this case) is the same for both of them. The same is true for `bread1` and `bread2`. However, the same property does not hold for `bread1` and `bread3`, because `bread3` is gluten-free and occurs twice, while `bread1` occurs once.

If such an equivalence is properly defined, it may distribute all constants into equivalence classes. It may be enough to use just a single constant from each class, when applying actions to a state. The other actions are treated as symmetrical and may be pruned, which decreases the size of the search space.

More complex symmetries Let's take a look at the following state:

```
S1 = { (ontray sandw1 tray1), (ontray sandw2 tray2),
       (at tray1 kitchen),    (at tray2 kitchen)      }
```

There are two trays at the kitchen with one regular sandwich on each one of them. If we want to put a new sandwich on a tray, or move a tray to a table and serve children, it does not matter which tray we choose. However `tray1` and `tray2` are not equivalent in the sense of the previous example. `tray1` occurs in an atom with `sandw1`, while `tray2` occurs in an atom with `sandw2`, which is a different constant. However, if we decide to analyze occurrences of `tray1` with respect to `sandw1` and `tray2` with respect to `sandw2`, we may say, that the pair of constants `(tray1, sandw1)` is in certain sense equivalent to the pair `(tray2, sandw2)`.

This example requires a more general definition of equivalence. In the following text, we will try to come up with such definitions.

3.1 Relational automorphism

The equivalence in terms of Riddle et al. [2015] can be described in the following way: constants A and B are equivalent in the state S, if after swapping occurrences of A and B, we get a state identical to S.

Similarly, the equivalence of "pairs of constants" can be described this way: (A,B) is equivalent to (C,D), if after swapping A with C and B with D, we get the same state. In a general case, we are talking about the one-to-one function, that maps each constant to another constant (permutation), and after mapping constants according to that function, we will get the same state. Such permutations are called *automorphisms*.

From now on, each permutation will mean the permutation of constants, unless stated otherwise. Having a permutation P, let's define a notation, that will be used throughout this thesis:

- For a constant c, $P(c)$ is a constant, on which c is mapped by P (the standard meaning of a permutation)
- For an atom $a = pred_i(c_1, c_2, \ldots c_n)$, $P(a) = pred_i(P(c_1), P(c_2), \ldots, P(c_n))$
- For a set of atoms $S = \{a_1, \ldots a_k\}$, $P(S) = \{P(a_1), \ldots P(a_k)\}$

- For an action $a = (prec, eff^+, eff^-)$, $P(a) = (P(prec), P(eff^+), P(eff^-))$

Having this new notation, we can define the equivalence of states in the following way:

Definition 1. *Two states S_0, S_1 are equivalent (isomorphic), if there exists a permutation P of constants, such that $P(S_0) = S_1$.*

Definition 2. *The permutation P is an automorphism of the state S, if $P(S) = S$.*

Observation 1. *Let S and T be states and P be a permutation of constants. Then, P is an isomorphism between S and T iff for each predicate $pred_i$ with an arity n and for any set of constant $a_1, a_2 \dots a_n$:*

$$pred_i(a_1 \dots a_n) \in S \iff pred(P(a_1) \dots P(a_n)) \in T$$

Proof. The statement follows directly from the definition of an isomorphism. □

Theorem 2. *Let S and T be equivalent states and P is an isomorphism between them, $P(S) = T$. Then, each action A is applicable to S iff $P(A)$ is applicable to T and $S_0 = \gamma(S, A)$ is equivalent to $T_0 = \gamma(T, P(A))$ through the permutation P.*

Proof. Let $A = act_i(a_1 \dots a_n), P(A) = act_i(P(a_1) \dots P(a_n))$.

An action is applicable to a state, when all preconditions are satisfied. Applicability of $P(A)$ to T follows from Observation 1.

Let's analyze the equivalence of two new states obtained by applying A to S and P(A) to T. We will apply the Observation 1 to show the equivalence of these states by showing that:

$$pred_i(a_1 \dots a_n) \in S_0 \iff pred_i(P(a_1) \dots P(a_n)) \in T_0$$

We have to prove this property for states generated by applying A to S and $P(A)$ to T. When an atom $pred_i(a_1 \dots a_n) \in S_0$ is from S, i.e. it was not removed by A, then $pred_i(P(a_1) \dots P(a_n))$ was in T (according to Observation 1) and was not deleted by $P(A)$, thus $pred_i(P(a_1) \dots P(a_n)) \in T_0$. If $pred(a_1 \dots a_n) \in S_0$ was added by A, $pred_i(P(a_1) \dots P(a_n))$ must have been added by $P(A)$ into T_0. And since there exists P^{-1}, we can prove similar properties in the opposite direction to show, that there are no extra atoms in T_0. Hence we showed, that S_0 and T_0 are equivalent. □

Note, that at the beginning, we did not assume S and T to be different states. If they are equal, then P becomes an automorphism and we can write a special case of the theorem.

Theorem 3. *Let S be a state and P be an automorphism of S. Then each action A is applicable to S iff P(A) is applicable to S. $S_0 = \gamma(S, A)$ is equivalent to $S_1 = \gamma(S, P(A))$ through the permutation P.*

Theorem 4. *Let S_0 and T_0 be states and P be an isomorphism between them, $T_0 = P(S_0)$. If the state S_n is reached after applying the sequence of actions $A_1, \dots A_n$ on S_0, then a state $T_n = P(S_n)$ is reached after applying the sequence of actions $P(A_1), \dots P(A_n)$ on T_0.*

Proof. This statement can be proven by induction on the length of the action sequence using Theorem 2. It holds for the empty sequence of actions, $P(S_0) = T_0$. When a sequence $P(A_1) \dots P(A_i)$ of actions leads to $P(S_i) = T_i$ and we have an action A_{i+1} applicable to S_i leading to S_{i+1}, then $P(A_{i+1})$ is applicable to T_i leading to $P(S_{i+1}) = T_{i+1}$, thanks to the previous theorem. □

These theorems tell us, that for two equivalent states, if some action is applicable to one of them, then an equivalent action is applicable to the other one and resulting states are again equivalent using the same isomorphism. The last theorem tells us, that for two equivalent states, the whole state structure, that can be reached from one state, corresponds to the state structure, that is reachable from the other state, through the same isomorphism P.

Having two actions A and B applicable at one state S, the fact that $A = P(B)$ through some automorphism P of S tells us, that we can omit one of the actions, because it leads to the similar behavior. That is the main idea behind pruning of the search space that shall be described later in the thesis.

3.2 Reachability of the goal

In the planning problem, the goal is defined by the set of atoms $G = \{g_1, \dots g_n\}$. A state S is a goal state, if $G \subseteq S$.

Let's assume, that we are in the state S with an automorphism P, actions A and B are applicable on S and $A = P(B)$. There are following states $S_A = \gamma(S, A)$ and $S_B = \gamma(S, B)$ such that $S_A = P(S_B)$.

Assume that there is a goal state Sg, $G \subseteq Sg$, that is reachable at some point after applying B. Then, according Theorem 4, there is an equivalent state $P(Sg)$ reachable after applying A. But we can not say, whether $P(Sg)$ is also a goal state, i.e. whether $G \subseteq P(Sg)$. However, if $P(Sg)$ is not the goal state, then pruning B and applying just A may lead to an incomplete search.

Consider the state from the previous example:

```
S1 = { (ontray sandw1 tray1), (ontray sandw2 tray2),
       (at tray1 kitchen),    (at tray2 kitchen)      }
```

Moving `tray1` to `table1` is equivalent to moving `tray2` to `table1`, since both trays have similar content. However, if there is a goal `G = { (at tray2 kitchen) }`, moving just `tray1` and pruning the movement of `tray2` would lead to the plan, that is longer than the optimal plan. In the worst case, it can make all goal states unreachable.

In the following text, we will introduce two ways of preserving the reachability of the goal. Each of these methods is based on a special theorem.

Theorem 5. *Let S be a state and P be an automorphism of S, actions A and B be applicable to S, $A = P(B)$ and a certain goal state Sg be reachable after applying B. Let P be also an automorphism of the goal G, i.e. $P(G) = G$. Then $P(Sg)$ is reachable after applying A and it is also the goal state.*

Proof. The state $P(Sg)$ is reachable thanks to Theorem 4. $G \subseteq Sg$, thus $P(G) \subseteq P(Sg)$. And since $P(G) = G$, then $G \subseteq P(Sg)$. □

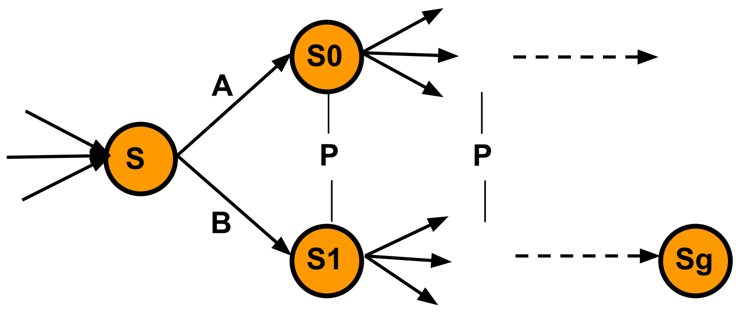


Figure 3.1: Reachability of the goal.

This simple condition, that each automorphism of the state, that we use, must also be an automorphism of the goal, is a sufficient condition for preserving the reachability of the goal during the pruning. On the other hand, it restricts the symmetry a lot (we can use a smaller number of automorphisms), resulting into a much smaller rate of pruning.

Up until now, we have been looking at symmetries from the local perspective of a single state. We have studied, how can the search process change, if we choose a different action from two equivalent actions in one state. But as the search continues, different automorphisms are found along the path. It can be interesting to study, how can the search process change, if we choose a different action from two equivalent actions in each state along the whole path, see the illustration in Figure 3.2.

Theorem 6. *Let there be an initial state S_0 and a sequence of actions $A_1, \dots A_n$, that leads from S_0 to a state S_n. For each state $S_0, \dots S_{n-1}$ along the path, let there be P_i automorphism of S_i, such that $P_0(A_1), P_0(P_1(A_2)) \dots P_0(..P_{n-1}(A_n)..)$ is the plan, leading to the goal state $P_0(..P_{n-1}(S_n)..)$. Let each P_i be also an automorphism of the initial state. Then $P = P_0 \circ \dots \circ P_{n-1}$ (the product of permutations) is also an automorphism of the initial state and $P(A_1) \dots P(A_n)$ is a plan leading to the same goal state $P(S_n) = P_0(\dots P_{n-1}(S_n) \dots)$.*

Proof. Obviously, the product of multiple automorphisms of the initial state is still an automorphism of the initial state.

It can be shown, that actions $P_0(A_1), P_0(P_1(A_2)) \dots P_0(..P_{n-1}(A_n)..)$ generate the path $S_0, P_0(S_1), P_0(P_1(S_2)) \dots P_0(..P_{n-1}(S_n)..)$ (can be easily shown by induction). The last state $P_0(..P_{n-1}(S_n)..) = P(S_n)$ is the goal state, so P is an isomorphism between S_n and the goal state $P(S_n)$.

Since $A_1, \dots A_n$ is a valid sequence of actions, applicable to the initial state and P is an automorphism of the initial state, $P(A_1) \dots P(A_n)$ is also a valid sequence of actions, applicable to the initial state. It leads to $P(S_n)$, so it is also a plan. $\square$

This theorem tells us, that if we use just the automorphisms of the initial state for pruning during the search and we can detect, that we have reached a state S_n equivalent to the goal through some automorphism of the initial state, it guarantees the reachability of the goal (we are not restricted by automorphisms of the goal, as in the previous case).

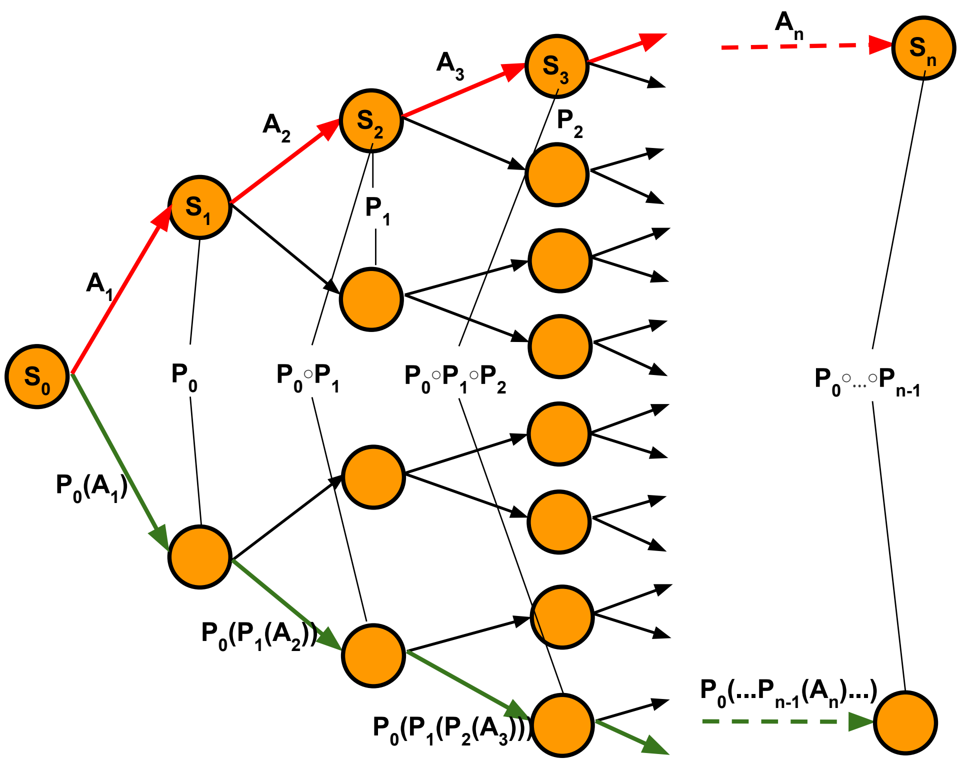


Figure 3.2: Automorphisms along the path

Once such S_n is reached using actions $A_1, \ldots A_n$, we just have to find the correct mapping P, such that $G \subseteq P(S_n)$. If P was just any isomorphism between the goal and the goal-equivalent state, the translation of $A_1, \ldots A_n$ to a real plan may be very hard or even impossible. But since P is the automorphism of the initial state, applying it to $A_1, \ldots A_n$ gives us a plan.

Such approach can be useful, when the initial state has many automorphisms. We will not present any method of detecting goal-equivalent states. However, we will use this theorem later to explain the methods behind Bagged representation in terms of relational automorphisms.

In general, reaching the goal-equivalent state does not necessarily mean, that our sequence of actions can be converted to a real plan. Those familiar with a Gripper domain can notice, that initial states are equivalent to goal states through a permutation, that swaps `rooma` and `roomb`. However, an empty sequence of actions (and even some longer sequences, that lead back to the initial state) can not be converted into a plan.

3.3 The complexity of automorphisms detection

When we look at states as the structures in terms of predicate logic, we see a finite set (constants) with a finite number of relations over that set (atoms grouped together by predicates), the permutation P between two equivalent states is an isomorphism between two isomorphic structures. Thus, deciding, whether two states are equivalent, corresponds to finding an isomorphism between two finite logical structures (with no functions, only relations).

There is a straightforward method to do that: for n elements in the carrier set, we can go through all $n!$ permutations (bijective functions between two carrier sets) and verify, whether some permutation P corresponds with each relation R_i (thus, P is an isomorphism).

$$\forall x_1 \dots x_k : R_i(x_1, \dots, x_k) \iff R_i(P(x_1), \dots, P(x_k))$$

Finding isomorphisms this way is computationally expensive. We can easily transform the problem of graph isomorphism (GI) into the structure isomorphism. Vertices would be converted into the finite carrier set, while edges give atoms of a single binary predicate. Such problems are called GI-hard. GI is in the complexity class NP (it is easy to make a polynomial-time deterministic algorithm checking that a certificate for graph isomorphism actually represents a graph isomorphism), but it is not known to be in P or to be NP-complete. See Köbler et al. [1993] for more information about graph isomorphism.

Automorphism candidates Let's define a relation, which may give us a clue about how many automorphisms there are. For each constant c we want to find other constants $d, \dots,$ to which c can be possibly mapped by automorphisms. The idea is to check, whether for each atom containing c there is a corresponding atom containing d at the same positions (without analyzing other constants in these atoms).

Definition 3. *For a state S, an Automorphism candidate relation AC of constants is defined in the following way: $AC(c_0, c_1)$ when there exists a permutation C of atoms in S, such that when $C(a) = b$, then atoms a, b belong to the same predicate and c_0 occurs in a at the same positions, as c_1 in b.*

If the planning problem had just a single binary predicate, the state can be viewed as a directed graph, in which constants are vertices and atoms are directed edges. Then, $AC(u, v)$ means, that vertices u, v have the same input and output degree. The permutation C (which does not have to be unique) maps incoming edges of u to incoming edges of v and outcoming edges of u to outcoming edges of v (edges that don't contain u can be mapped in any way).

Let's take a look at one of our previous examples:

```
S1 = { (ontray sandw1 tray1), (ontray sandw2 tray2),
       (at tray1 kitchen),    (at tray2 kitchen)      }
```

Here we see, that $AC(sandw1, sandw2)$, because the permutation of atoms C can map `(ontray sandw1 tray1)` to `(ontray sandw2 tray2)` to satisfy the condition: when `sandw1` occurs at some positions in an atom a, then `sandw2` occurs at the same positions in an atom $C(a)$ and both atoms belong to the same predicate.

We also see, that $\neg AC(sandw1, kitchen)$, simply because these constants occur in totally different predicates.

Observation 7. *The Automorphism candidate relation is an equivalence.*

Proof. The relation is reflexive - $AC(c, c)$ because identity is the permutation of atoms, which satisfies the conditions.

It is symmetric - once we have $AC(c_0, c_1)$, there is a permutation C of atoms that satisfies the conditions. The permutation C^{-1} satisfies the conditions for $AC(c_1, c_0)$.

It is transitive - when $AC(c_0, c_1)$ and $AC(c_1, c_2)$, there is a permutation of atoms C_0 mapping the occurences of c_0 to c_1 and a similar permutation C_1 mapping occurences of c_1 to c_2. The product $C = C_0 \circ C_1$ of these permutations must be a permutation mapping occurences of c_0 to occurences of c_2, that satisfies the conditions and proves $AC(c_0, c_2)$. □

Observation 8. *Let P be an automorphism of S. Then:*

$$\forall c : AC(c, P(c))$$

Proof. The automorphism P of constants also defines the permutation of atoms of the state S. When $P(a) = b$, then atoms a, b belong to the same predicate and c occurs in a at the same positions, as $P(c)$ in b. □

Observation 9. *Let $(C_1, \dots C_n)$ be the equivalence classes of AC for a state S. Then there exists at most $\prod_{i=1}^{n} |C_i|!$ automorphisms of S.*

Proof. Based on Observation 8, each automorphism can remap constants only within AC classes. Counting all permutations in each class and multiplying counts together gives an upper limit for the number of automorphisms. □

3.4 Working with automorphisms

Each planning domain usualy contains multiple predicates (relations). Automorphisms can map atoms only to the atoms of the same predicate.

When looking for automorphisms of the state, we can find automorphisms for each predicate separately. The automorphisms of the whole state will be the intersection of automorphisms of specific predicates (relations).

Since atoms of rigid predicates are present in every reachable state, this property allows us to precomute the automorphisms of rigid predicates in advance, and then look for automorphisms of fluent predicates only. Having automorphisms of fluent predicates of a specific state, we can intersect them with precomputed automorphisms of rigid predicates, to get automorphisms of the whole state.

Having two sets of automorphisms for states S_0, S_1, if we want to get those automorphisms, that work for both states, again, we can intersect their two sets.

Similarly, when looking for some equivalence relation for a state (e.g. AC equivalence), we can find an equivalence relation for each predicate separately. The equivalence of the whole state will be the intersection of equivalences of specific predicates. This feature can be used to build equivalences of rigid predicates in advance, and then intersect them with equivalences of fluent predicates during the search.

Having two equivalence relations for states S_0, S_1, if we want to get an equivalence, that works for both states, again, we can perform an intersection of two relations.

Automorphisms and permutations in general can be studied in a branch of Mathematics called the *group theory*. In the upcoming text, we will use several basic terms and concepts of the group theory for describing properties of automorphisms. Let's mention some of them.

Definition 4. *Let G be a group, $S \subseteq G$. Then $\langle S \rangle$ is the intersection of all subgroups of G containing S (the "smallest" subgroup of G containing S). Equivalently, $\langle S \rangle$ contains all elements of G, that can be expressed as the finite product of elements of S or their inverses.*

When $\langle S \rangle = G$, we say, that the set S generates *the group G. It is a set of generators of the group G.*

Definition 5. *Let $a \in G$, the* order *of a is the smallest positive integer m, such that $a^m = e$ (a composed m times with itself gives the identity element of the group), or a is said to have infinite order, when no such m exists.*

We will work with groups of permutations.

Definition 6. *The* parity *of the permutation P is the parity of the number of inversions, i.e. number of pairs x, y, such that $x < y$ and $P(x) > P(y)$. It splits all permutations into two classes of equal size: odd permutations (with an odd parity) and even permutations (with an even parity).*

A permutation is called a transposition*, when it exchanges two elements and keeps other elements fixed. We write the transposition as (a/b), when it exchanges elements a and b.*

Two transpositions $(a/b), (c/d)$ are disjoint*, when $\{a, b\} \cap \{c, d\} = \emptyset$ (when they exchange different elements).*

It is a well-known fact, that the group of all permutations can be generated by the set of all transpositions. Every transposition has the order of two (combining the transposition with itself gives the identity) and has odd parity (because it has just a single inversion).

3.5 T1 Automorphisms

In the previous chapter, we showed, that pruning of the search space is possible, when we know some automorphism of the current state and we know, that two actions are equivalent thanks to some automorphism. However in general, finding automorphisms can be a hard problem. The number of automorphisms can be exponential in terms of the size of the state, so representing automorphisms can also be tricky. Another problem is to decide, whether two actions are isomorphic through any of automorphisms, that we have previously found.

Instead of trying to find and work with all automorphisms of the state, we will focus on special cases of automorphisms, that are easy to detect, yet useful for pruning in practice.

Automorphisms can always be composed with one another or inverted (just as any permutation), giving new automorphisms. When studying automorphisms, it is much better to think about them as the whole subgroup of automorphisms, rather than some separate entities. In the following work, we will focus on detecting such subgroups, that are closed under composition and inversion.

Definition 7. *We will denote as T1 the set of automorphisms, which are transpositions. The relation L1 on constants is defined as* $L1(a,b) \iff (a = b) \vee (a/b) \in T1$.

Automorphisms in T1 correspond to object equivalencies, that were defined in the literature in the past. E.g. Fox and Long [1999]: "For our purposes we define symmetric objects to be those which are indistinguishable from one another in terms of their initial and final configurations."

Riddle et al. [2015] "Initial-state-equivalent objects are indistinguishable in the problem's initial state. They appear in the same predicates with the same other arguments (objects and constants) in the initial state".

In both cases, authors mention the initial and the goal states. We will consider such equivalences for any state. Let's take a look at the example, that we have mentioned previously:

```
S0 = { (at tray1 kitchen), (at tray2 kitchen),
      (at_kitchen_bread bread1), (at_kitchen_bread bread2),
      (at_kitchen_bread bread3), (no_gluten_bread bread3)  }
```

In this case, $T1 = \{(tray1/tray2), (bread1/bread2)\}$. If we decide to move some tray to `table1`, it does not matter, which tray we choose.

However, if the goal was `G =  (at tray1 table1)` , moving only `tray2` would lead to a plan, that is longer than an optimal plan (in general, incorrect pruning can prevent reaching the goal). We can solve it using one of the methods, that we have mentioned earlier. Namely, checking, if the used automorphism is also an automorphism of the goal G. In this case, $(tray1/tray2)$ would not be an automorphism of G, so moving `tray1` is not equivalent to moving `tray2`, we can not prune actions in this case.

Observation 10. *L1 is an equivalence relation.*

Proof. Reflexivity requirement is already in the definition.

Symmetry can be shown in a following way. $L1(a,b)$ implies that $(a/b) \in T1$, which is the same as $(b/a) \in T1$, thus $L1(b,a)$.

Transitivity: Let $L1(a,b)$ and $L1(b,c)$. Then $(a/b) \in T1, (b/c) \in T1$. By composing these automorphisms in the following way we get a new automorphism: $(a/b) \circ (b/c) \circ (a/b) = (a/c) \in T1$, which implies $L1(a,c)$. □

How does the closure of T1, $\langle T1 \rangle$, look like? Since L1 is an equivalence, it splits all constants into equivalence classes. The transposition of any two constants within this class is an automorphism. Since we have all transpositions for each pair of constants, we can generate all possible permutations within that class. $\langle T1 \rangle$ consists of all permutations, that mix constants, while preserving the same L1-equivalence class of each constant.

Observation 11. *For any state S, if* $L1_S = AC_S$*, then* $\langle T1_S \rangle$ *are all automorphisms, that exist on S.*

Proof. L1 equivalence tells us, that each permutation, that preserves the L1-equivalence class of constants, is an automorphism. AC tells us, that only permutations preserving AC-equivalence classes can be automorphisms. And since these two relations are equal, there can not be another automorphism. □

We have defined T1 automorphisms, which generate a subgroup of all automorphisms, and we have shown some properties of these automorphisms. In following chapters, we will present algorithms for finding T1 and using it for pruning. However, for the sake of pure theoretical science, let's try to examine another class of automorphisms.

3.6 T2 Automorphisms

T1 is quite a simple class of automorphisms. We would like to define some more complex class, but still not the class of all automorphisms.

Previously, we started with generators consisting of a single transposition, so now we can start with generators, that consist of two transpositions. There are several ways to do it.

Definition 8. *Let's define several generators of subgroups of automorphisms:*

- *Let T2* be automorphisms, that are the product of at most two transpositions.*
- *Let T2+ be automorphisms, that are the product of at most two disjoint transposition.*
- *Let T2 be automorphisms, that are the product of exactly two disjoint transpositions, and no transposition from the pair is an automorphism by itself.*

From the definition we see, that for any state:

$$T2 \subseteq T2{+} \subseteq T2{*}, \langle T2 \rangle \subseteq \langle T2{+} \rangle \subseteq \langle T2{*} \rangle$$

Theorem 12. *There exist states, where $\langle T2 \rangle \neq \langle T2{+} \rangle$ and where $\langle T2{+} \rangle \neq \langle T2{*} \rangle$.*

Proof. For the first inequality, let's consider the following state:

```
S = { (arc a b), (arc b a), (arc c d), (arc d c) }
T2+ = { (a/b)(c/d), (a/c)(b/d), (a/d)(b/c), (a/b), (c/d) }
T2  = { (a/b)(c/d), (a/c)(b/d), (a/d)(b/c) }
```

We see, that $(a/b) \in \langle T2{+} \rangle$. We have to prove, that $(a/b) \notin \langle T2 \rangle$. All automorphisms in $T2$ have even parity and the parity is preserved during the composition and the inversion of even permutations. However, (a/b) has an odd parity, so it can not be a product of elements of $T2$.

For the second inequality, let's consider the following state:

```
S = { (arc a b), (arc b c), (arc c a) }
T2* = { (a/b)(b/c), (b/c)(a/b) },          T2+ = { }
```

T2+ is empty for this state, because we don't have four different constants to create an automorphism with two disjoint transposition, and no single transposition is an automorphism. $\langle T2{+} \rangle = \{Id\} \neq \langle T2{*} \rangle$. □

In the remaining part of this section, we will focus on "the smallest class" of automorphisms, T2. The members of T2 are automorphisms consisting of two disjoint permutations, which we will denote as $(a/b)(c/d)$. One transposition will be called a *witness* of the other transposition, if they both create an automorphism. Each transposition needs a witness to be a member of T2.

Observation 13. *Let a, b, c, d be different constants and let $(a/b)(c/d)$ be an automorphism. Then $(a/b) \in T1 \iff (c/d) \in T1$.*

Proof. $(a/b) \in T1 \iff (a/b)$ is an automorphism $\iff (a/b) \circ (a/b)(c/d) = (c/d)$ is an automorphism (because the composition of two automorphisms is an automorphism) $\iff (c/d) \in T1$. □

The definition of T2 tells us, that transpositions used in automorphisms of T2 can not occur in T1. This observation tells us, that when one of transpositions occurs in T1, then another one must also occur in T1.

To get an idea of how the subgroup $\langle T2 \rangle$ looks like, we must study, how automorphisms of T2 interact with each other. When two different T2 automorphisms share constants (see Figure 3.3), they can share one constant (a), two constants (b,c,d), three constants (e,f) or four constants (g).

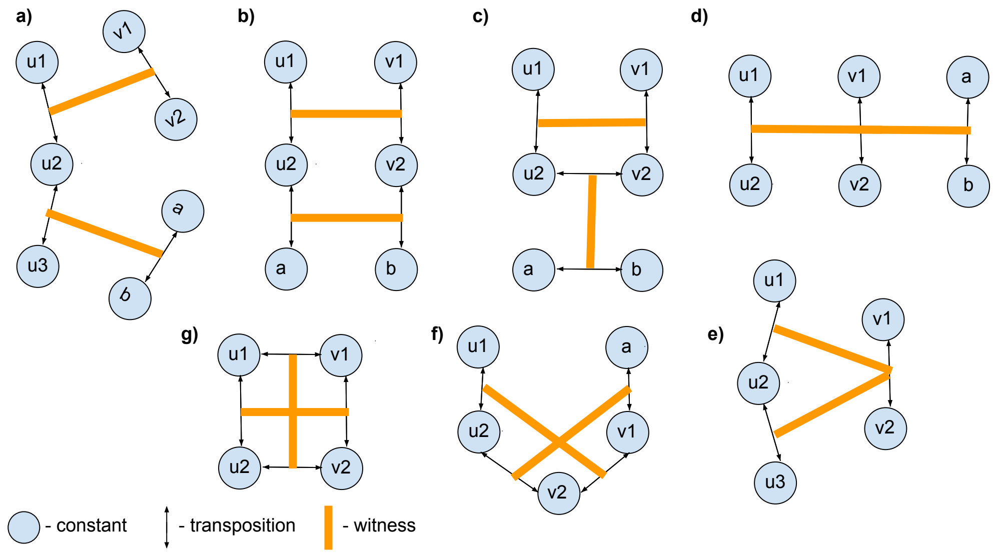


Figure 3.3: Interactions between automorphisms of T2.

Definition 9. *Let's define a relation L2 on constants: $L2(a, b)$ when $a = b$ or there exists a transposition (c/d) such that $(a/b)(c/d) \in T2$.*

Theorem 14. *When combinations of the type g. don't occur, L2 is an equivalence relation.*

Proof. Reflexivity is already in the definition, just as in the case of L1. Symmetry is obvious. We have to prove transitivity.

We have to show, that $L2(u1, u2) \wedge L2(u2, u3) \implies L2(u1, u3)$. Only the cases a., b., c., e., f. are relevant for this property (because the whole transposition is shared in d.) We have to show, that there exists an automorphism in T2, such that $(u1/u3)$ is one of its transpositions. It also requires to prove, that $(u1/u3)$ is not an automorphism by itself.

In a., $(u1/u2)(v1/v2) \circ (u2/u3)(a/b) \circ (u1/u2)(v1/v2) = (u1/u3)(a/b)$, and since (a/b) is not an automorphism, thus $(u1/u3)$ also is not.

In e., $(u1/u2)(v1/v2) \circ (u2/u3)(v1/v2) \circ (u1/u2)(v1/v2) = (u1/u3)(v1/v2)$, and since $(v1/v2)$ is not an automorphism, thus $(u1/u3)$ also is not.

It remains to prove the property for cases b., c. and f. In each case, a certain composition of automorphisms gives us a new automorphism, which proves the transitivity.

- b. $(u1/u2)(v1/v2) \circ (u2/a)(v2/b) \circ (u1/u2)(v1/v2) = (u1/a)(v1/b)$
- c. $(a/b)(u2/v2) \circ (u1/u2)(v1/v2) \circ (a/b)(u2/v2) = (u1/v2)(v1/u2)$
- f. $(u1/u2)(v1/v2) \circ (u2/v2)(a/v1) \circ (u1/u2)(v1/v2) = (u1/v1)(a/v2)$

Now we have to show, that these new automorphisms are in T2, i.e. that their transpositions are not automorphisms by themselves.

When $(u1/u2)(v1/v2) \in T2$, then there must be an atom $x \in S$, such that $(u1/u2)(v1/v2)[x] \in S$, but $(u1/u2)[x] \notin S$. x must contain constants from $\{u1, u2\}$ and from $\{v1, v2\}$.

Having an atom, we can apply automorphisms to it. When an atom is in S, all morphed versions also are in S. When an atom is not in S, all morphed versions also are not in S.

The remaining part of the proof for the cases b., c. and f. has the similar structure. Let's demonstrate it for the case b. For a contradiction, let's suppose, that $(u1/a)$ is an automorphism. $x = p(u1, u2, v1, v2, a, b) \in S$, $x2 = (u1/u2)[x] = p(u2, u1, v1, v2, a, b) \notin S$. We mechanically generate all morphed versions of x and $x2$ and show, that these sets are not disjoint - some atoms both are and are not in S. Let's do a thorough proof for the case b.:

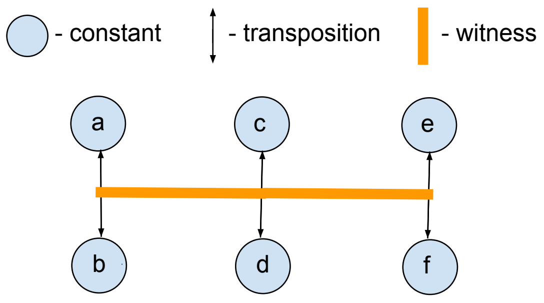
1. $p(u1, u2, v1, v2, a, b) \in S$ - precondition atom $x \in S$
2. $p(u2, u1, v1, v2, a, b) \notin S$ - since $(u1/u2)$ is not an automorphism
3. $p(u1, a, v1, b, u2, v2) \in S$ - applying $(u2/a)(v2/b)$ to 1.
4. $p(a, u1, v1, b, u2, v2) \notin S$ - applying $(u2/a)(v2/b)$ to 2.
5. $p(a, u1, v1, b, u2, v2) \in S$ - applying $(u1/a)$ to 3.

If $(u1/a)$ was automorphism, then $p(a, u1, v1, b, u2, v2)$ is and is not in S, which is a contradiction, $(u1/a)$ is not an automorphism, then $(v1/b)$ also isn't an automorphism and $(u1/a)(v1/b) \in T2$.

We have chosen $x = p(u1, u2, v1, v2, a, b)$ without loss of generality. If we omit some constants (only a constant from $\{u1, u2\}$ and a constant from $\{v1, v2\}$ must remain), the proof still holds. If some constants occured multiple times in x, the proof also holds. If we add some additinal constants to x, that are not manipulated by automorphisms, the proof also holds.

For the remaining cases c. and f., similar proofs have been done by a simple computer program. $\square$

Definition 10. *We call a state to be SW (single-witnessed), when for any different constants a, b, c, d, e, f: if $(a/b)(c/d)$, $(c/d)(e/f)$ are automorphisms, then (c/d) is an automorphism.*



SW property tells us, that each transposition, that occurs in automorphisms in T2, can have at most one witness. Automorphisms inside T2 can share constants, but they can not share transpositions (for SW states).

Theorem 15. *When the arity of all predicates is at most two, the state is SW.*

Proof. Let's look at atoms, that do not contain any of constants $\{a, b\}$. For these atoms, $(a/b)(c/d)$ is an automorphism, and since they don't contain $\{a, b\}$, (c/d) is an automorphism for them, too.

Let's look at atoms, that do not contain any of constants $\{e, f\}$. For these atoms, $(c/d)(e/f)$ is an automorphism, and since they don't contain $\{e, f\}$, (c/d) is an automorphism for them, too.

Let's look at atoms, that do not contain any of constants $\{c, d\}$. Obviously, (c/d) is an automorphism for them, too.

And since there are no atoms containing elements of all three transpositions (because the arity is at most 2), each atom must belong to one (or more) of subsets from above. Thus, the union of these subsets is the whole state and (c/d) is an automorphism of that state. □

Many domains from IPC have at most binary predicates (Childsnack, Barman, Gripper, ...), so all states in these domains are SW. If only rigid predicates have an arity larger than two, SW property can be checked for them in advance. It may be tricky to guarantee SW property in other cases.

Theorem 16. *When the state has SW property, only interactions depicted in b., f. and g. are possible.*

Proof. In cases d. and e., a transposition has two witnesses, which is not possible in a SW state. In cases a., c., new T2 automorphisms can be generated, which share a transposition with current automorphisms (see the previous proof), which is also not possible in a SW state. □

These theorems lead to an algorithm for detecting T2 automorphisms, which is slightly more complex, than the algorithm for T1 automorphisms, which will be presented later. It uses L1 equivalence and AC equivalence relations to find generators of $\langle T2 \rangle$. Our implementation of this algorithm did not offer any significant pruning, because T2 automorphisms are quite rare in IPC domains and problems.

Tk Automorphisms Just as we defined T1 and T2, we can define Tk in the same way.

Definition 11. *Let Tk be automorphisms, that are the product of exactly k disjoint transpositions, and no $k-1$ or less of these transposition create an automorphism by themselves.*

We see, that when an automorphism $A = t_1, \dots t_k \in Tk$, then, for each $j < k$, each subset of j transpositions from A is not in Tj.

When building Tk, we can build all $Tj, j < k$ first, which can be used for checking, if specific automorphisms can be in Tk. It also tells us, that the bigger are Tj, the smaller will be $Tk, k > j$. For example, if T1 contains all possible transpositions, then T2, T3, T4 ... are empty.

Definition 12. *Let Lk be a relation on constants: $Lk(a, b)$ when $a = b$ or there exist $k-1$ transpositions $t_1, \dots t_{k-1}$ such that $(a/b), t_1, \dots t_{k-1} \in Tk$.*

4. Algorithms

Now, we want to exploit previously defined symmetries during the search of the plan. Let's focus at the phase, in which the planner finds all possible actions applicable to the current state.

```
1: Acts ← AllActions(S, Dom)
2: Acts ← Prune(S, Acts)
```

Pruning mechanism can be modeled by calling a *Prune* subroutine, which receives all applicable actions and returns some subset of these actions. The algorithm with no pruning techniques can be modeled with a *Prune* subroutine, which returns the same set that it receives.

Our pruning mechanism, wich exploits T1 automorphisms, consists of two phases. The first phase is finding the L1-equivalence relation on the constants in a current state S. The second phase is removing actions from *Acts*, that are equivalent to other actions, which are preserved.

```
1: function Prune(S, Acts)
2:     L1 ← findL1(S)
3:     Acts ← pruneL1(Acts, L1)
4:     return Acts
```

4.1 Constructing L1-equivalence relation

We want the algorithm to find the L1-equivalence relation on a state S. At the first step, we create the structure, which we call the *occurence map*. It will help us find the actual L1-equivalence.

Occurence map For constants $c_1, \dots c_m$ and predicates $p_1, \dots p_n$, the occurence map is a matrix O of the size $m \times n$ of linked lists. Each linked list $O[i, j]$ contains all atoms (or pointers to atoms), where the constant c_i occurs in a predicate p_j.

The occurence map can be constructed using a simple algorithm, which goes through each atom of the state (and each constant in that atom) and creates the required map.

Algorithm 2 Finding occurence map for a state

```
1: function findOccurenceMap(S)
2:     omap ← Matrix(m,n) of empty linked lists    ▷ initialize an empty map
3:     for all atom ∈ S do
4:         for all c ∈ atom do
5:             omap[c, atom.predicate].add(atom)
6:     return omap
```

If there was just a single predicate, which was binary, the state can be viewed as a directed graph, where constatns are vertices and binary atoms are edges.

Finding occurence map would correspond to converting a graph from one representation to another. Input representation would be an unordered set of edges. The output would be adjacency lists, where each vertex has a list of incoming and outcoming neighbours.

Efficient implementation of Occurence map The work with an occurence map can be a bottleneck of the performance of the whole implementation (allocating, navigating and deallocating linked lists). In our implementation, the following representation turned out to be quite efficient.

When the linked list has the length 0, the matrix contains a special value *null*. When the linked list has the length 1, the matrix contains that single item, i.e. the single index of an atom. Only when the linked list is bigger than 1, the matrix value contains a pointer to that specific linked list.

For example, in a Childsnack domain, in the initial state of our problem, a huge majority of linked lists were empty (the constant did not occur in that predicate at all). Just a single list had the length bigger than 1 (the constant *kitchen* in the *at* predicate - all trays are located at the kitchen). The rest of linked lists had the length equal to 1.

Linked lists longer than 1 are usually quite rare in the occurence map. The problem of storing them efficiently is in some sense similar to dealing with collisions in hashing. Many ideas from hashing methods can lead to new ways of representing and accessing occurence maps.

Finding the relation The output of *findL1* should allow us to check, whether two constants are in the same equivalence class. For m constants, we want to construct an array of m integers, such that i-th and j-th constant are equivalent iff i-th and j-th integers of the array are equal.

Union-find data structure (also called disjoint-set data structure) keeps track of a finite set S of n elements and its partitioning into disjoint subsets $S_1, ...S_k$. At the beginning, each element is placed into its own single-item subset. The structure supports two operations: Union(x,y) and Find(x). Union(x,y) merges the subsets, in which x and y are located. Find(x) returns the identifier of the subset, in which x is located. Find operation lets us check, if two elements are in the same subset.

Union-find data structure is used for example in Kruskal's algorithm for a minimum spanning tree, where we create a single-item tree for each vertex, then we go through edges (sorted by weight) and connect trees together (merge sets of vertices), if they are not connected already.

Our algorithm puts each constant into a separate equivalence class at the beginning. Then it loops between each two pairs of constants and merges two classes, if constants satisfy the condition of L1-equivalence (after swapping occurences of the two constants, we should get the same state).

We first check, if two constants have the same number of occurences in all predicates. Then, for each predicate, for each occurence of the first constant in that predicate, we try to find a corresponding occurence of the second constant.

The subroutine *sameOccCount* loops through two rows of occurence map and checks, whether the linked list at $omap[i, p]$ has the same length, as the linked

Algorithm 3 Finding L1-equivalence

```
 1: function findL1(S)
 2:     omap ← findOccurenceMap(S)
 3:     uf ← new UnionFind(number of constants)
 4:     for all constant $i$ do
 5:         if uf.find(i)≠i then continue;
 6:         for all constant $j$, $j > i$ do
 7:             if uf.find(j)≠j then continue;
 8:             if ¬sameOccCount(omap, i, j) then continue;
 9:             eq ← true
10:             for all predicate $p$ do
11:                 used ← new List(omap[i,p].length)
12:                 for all ri ∈ omap[i,p] do
13:                     rfound ← false
14:                     for all rj ∈ omap[j,p] do
15:                         if used[rj]=0 & equiAtoms(ri,rj,i,j) then
16:                             rfound ← true; used[rj] ← 1; break;
17:                     if rfound = false then eq ← false; break;
18:                 if eq = false then break;
19:             if eq = true then uf.union(i,j)
20:     return uf.ToIntegerArray
```

Algorithm 4 Checking, if two atoms are equivalent

```
1: function equiAtoms(ri, rj, i, j)
2:     for all $a \in [1 \dots arity]$ do
3:         ok ← $(ri[a] = i$ & $rj[a] = j)$ OR $(ri[a] = j$ & $rj[a] = i)$
4:                 OR $(ri[a] = rj[a]$ & $ri[a] \neq i$ & $ri[a] \neq j)$
5:         if ok = false then return false
6:     return true
```

list at $omap[j, p]$. In other words, it checks, whether two constants occur in each predicate the same number of times.

Theorem 17. *The algorithm findL1 will finish. It will find the correct L1 equivalence.*

Proof. The algorithm consists of 5 nested loops, which iterate over finite sequences. Thus, the algorithm will finish.

The algorithm iterates over each pair of two constants (i-th and j-th) and skips to the next pair, as soon as it is clear, that i and j are or are not L1-equivalent. The correctness of this approach is given by four facts.

1. It is enough to check unordered pairs of different constants, thus we compare i,j, $i < j$.
2. If $uf.FIND(i) \neq i$, then i is already attached to some class C, the equivalence between i and some other constant k, $k < i$, was detected earlier. At that phase, equivalence with k was analyzed for every j,$j > k$. Thus, equivalence was also analyzed for all j,$j > i$, so we can skip that i.
3. If $uf.FIND(j) \neq j$, then j is already attached to some class C, we can skip that j. If j was equivalent to i, i and j were attached to class C during the analysis of k, the first representative of C, $k < i < j$.
4. If i and j occur different number of times in some predicate, the state can not be the same after swapping all occurences of i and j. Constants are not L1-equivalent.

These four facts mean, that $uf.Union(u, v)$ can be called only when $L1(u, v)$. If for some two constants $u, v : uf.Find(u) = uf.Find(v)$, then $L1(u, v)$ (we expect the correct implementation of UnionFind data structure).

Let $L1(u, v)$. WLOG $u < v$. Then, there exists the smallest $c \leq u < v$ such that $L1(c, u)$. During the loop, c could not be connected to any b smaller than c (it would contradict the fact, that c is the smallest representative of the equivalence class). At some point, c is the "outer constant" in our algorithm, while u, v will be "inner constants" (first u, then v). Inner constants can not be connected to any other constants (it would mean, that they were connected to some b smaller than c). Since $L2(c, u), L2(c, v)$, constants u, v will satisfy the conditions and will be connected to c. Since that moment, $uf.Find(u) = uf.Find(v)$. □

Observation 18. *Let's consider lists of N numbers having the sum S. The function F is defined on lists, such that when there is the same number K at two different positions, K^2 is added to F. Example:*

$$F([4, 4, 2, 2]) = 4^2 + 2^2 = 20$$
$$F([3, 3, 3, 3]) = 3^2 + 3^2 + 3^2 + 3^2 + 3^2 + 3^2 = 54$$

When N and S are fixed, the F has the maximal value iff all values in the list are the same.

Proof. Let there be n occurences of a and m occurences of b in the list, $a \neq b$. They add $\binom{n}{2}a^2 + \binom{m}{2}b^2$ to F. When we replace these occurences of a, b with the value $\frac{n \cdot a + m \cdot b}{n+m}$ (to preserve S), these occurences add $\binom{n+m}{2}\left(\frac{n \cdot a + m \cdot b}{n+m}\right)^2$ to F. We have to show, that $\binom{n}{2}a^2 + \binom{m}{2}b^2 \leq \binom{n+m}{2}\left(\frac{n \cdot a + m \cdot b}{n+m}\right)^2$.

$$
\begin{aligned}
&0 \leq \binom{n+m}{2}\left(\frac{n \cdot a + m \cdot b}{n+m}\right)^2 - \binom{n}{2}a^2 - \binom{m}{2}b^2 \\
&0 \leq \frac{(n+m-1)}{2}\frac{(na+mb)^2}{n+m} - \frac{n(n-1)}{2}a^2 - \frac{m(m-1)}{2}b^2 \\
&0 \leq (n+m-1)\frac{(na+mb)^2}{n+m} - n(n-1)a^2 - m(m-1)b^2 \\
&0 \leq (n+m-1)(na+mb)^2 - (n+m)\left(n(n-1)a^2 + m(m-1)b^2\right) \\
&0 \leq (n+m)\left((na+mb)^2 - n(n-1)a^2 - m(m-1)b^2\right) - (na+mb)^2 \\
&0 \leq (n+m)(2namb + na^2 + mb^2) - (na+mb)^2 \\
&0 \leq 2n^2amb + nmb^2 + 2nam^2b + nma^2 - 2namb \\
&0 \leq nm(2nab + 2mab - 2ab + b^2 + a^2) \\
&0 \leq 2ab(n+m-1) + b^2 + a^2
\end{aligned}
$$

And since in our case $ab > 0, m \geq 2, n \geq 2$, the last inequality is true. We showed, that if there are two different values in the list, the value of F can be increased by replacing them with the weighted average. The value of F is maximum iff all the values in the list are equal. □

Theorem 19. *Let the state S have n atoms and the problem have m constants. Then, the time complexity of the algorithm* $findL1$ *is* $O(\max(m^2, n^2))$.

Proof. Let the problem have p predicates. WLOG the arity of each predicate is a (we can fill in dummy constants, when a predicate has smaller arity).

The algorithm takes the biggest amount of time when no two constants are L1-equivalent. In that case, it has to check all $\binom{m}{2}$ pairs of constants. When analyzing some pair of constants i, j, we have to check for each predicate p_k, if the number of occurences of i in p_k is the same, as the number of occurences of j in p_k, which takes the time $\binom{m}{2}p \leq m^2 \cdot p$. In the worst case, these occurences correspond, so we proceed to the next step.

Each one of constants i, j has t occurences in total. For each predicate, we have to match each atom containing one constant with an atom containing another constant. In the worst case, constants occur just in one predicate, so we have to test each of t occurences of i with each of t occurences of j, which means testing t^2 pairs of atoms.

When the constants $c_1 \dots c_m$ occur $t_1 \dots t_m$ times in the state $(\sum t_i = n \cdot a)$ and two constants have the same number of occurences $t_i = t_j$, we have to do t_i^2 comparisons of atoms for them. What is the worst distribution of $t_1 \dots t_m$? According to Observation 18, we will make the biggest number of comparisons when all t_i are the same, i.e. when all constants occur the same number of times ($\frac{n \cdot a}{m}$ times).

Then, when matching occurences of some two constants, we will have to perform $\left(\frac{n \cdot a}{m}\right)^2$ comparisons of atoms. In total, we will make at most $m^2 \frac{(n \cdot a)^2}{m^2} = (n \cdot a)^2$

comparisons of two atoms. We can limit the maximum arity by some constant (so two atoms can be compared in a constant time). We can also limit the number of predicates by some constant. Then, the time complexity of the algorithm is $O(\max(m^2 \cdot p, (n \cdot a)^2)) = O(\max(m^2, n^2))$. □

Constructing AC relation The AC (Automorphism Candidate) relation is an equivalence, which gives us an upper limit on the number of automorphisms in the state. It can be computed using the same algorithm, as the L1 equivalence, with a small difference - instead of calling $equiAtoms(ri, rj, i, j)$ subroutine, we will call $candidates(ri, rj, i, j)$ subroutine.

Algorithm 5 Checking, if two atoms can be automorphism candidates

```
1: function CANDIDATES(ri, rj, i, j)
2:     for all $a \in [1 \ldots arity]$ do
3:         ok ← $(ri[a] = i \ \& \ rj[a] = j)$ OR $(ri[a] \neq i \ \& \ rj[a] \neq j)$
4:         if ok = false then return false
5:     return true
```

The proof of correctness will be similar, as in the case of L1 equivalence. The time complexity is also the same. AC relation can be used for detecting other subgroups of automorphisms, such as T2, T3, etc.

4.2 Pruning actions

Once we have constructed the L1-equivalence relation for a current state, we can proceed to pruning applicable actions according to this relation.

We will try to find equivalent actions for each operator separately, so from now on, let's consider just actions of one specific operator. We expect, that each action is identified by the list of its parameters. This list contains all constants, that occur in preconditions and effects of the action (the action is grounded, so there are no variables). If constants in the action A were substituted according to parameters of B, A would become equal to B. Thus, the problem of finding equivalent actions is reduced to the problem of finding equivalent lists of parameters. From now on, when we use the word action, we refer to the list of parameters. Instead of writing the list of specific constants as (c_2, c_5, c_1), we will write just the list of numbers $(2, 5, 1)$.

Based on the previous research, two actions $(a_1, \ldots, a_n), (b_1, \ldots, b_n)$ are equivalent iff two conditions are satisfied:

- $\forall i : L1(a_i, b_i)$
- there exists a valid permutation of constatns P, such that $\forall i : P(a_i) = b_i$

In other words, there must exist a permutation P, that preserves the L1-equivalency class for each parameter.

The second condition is necessary. Let's think about a state with constants $\{1, 2, 3\}$, L1-equivalence classes are $\{\{1, 2\}, \{3\}\}$. For two actions with parameters $(1, 2, 3)$ $(1, 1, 3)$, the first condition is satisfied (since $L1(1, 1)$, $L1(2, 1)$, $L1(3, 3)$).

However, no permutation can map both 1 to 1 and 2 to 1 (the second parameter of actions).

But, if we were sure, that no constant can occur in a list of parameters twice (e.g. when parameters have different types, such that no constant can satisfy both types), the second condition would be redundant. There always exists a valid permutation between two lists of different constants.

Note, that the equivalence of actions, which we have defined, is a true equivalence, and divides the set of actions into equivalence classes. Thus, our goal is to put just a single action of each class into the output.

The naive method of pruning would be going through each pair of actions, checking both conditions and when both of them are satisfied, throw one of actions away. But we will use more advanced method, which will require going through each action just once.

Our method will consist of two-level hashing. On the first level, we will sort actions into bins according to the first condition, and then according to the second condition.

Definition 13. *For an action $(a_1, \ldots, a_n)$, a* class-strip *is a tuple $(c_1, \ldots, c_n)$, such that c_i is the class (of L1-equivalence) of the constant a_i.*

When two actions have different class-strips, they can not be equivalent. The class-strip is used for hashing at the first level.

Once we know, that some subset of actions has the same class-strip, we can proceed to checking the second condition. When valid permutations exist between some pairs of actions, it again splits the subset into equivalence classes.

Let's get back to the previous example. Actions $(1, 1, 3)$, $(1, 2, 3)$, $(2, 1, 3)$, $(2, 2, 3)$ would all have the same class strip $(0, 0, 1)$ and would end up in the same bin. We see, that $(1, 1, 3)$ is L1-equivalent to $(2, 2, 3)$ and $(1, 2, 3)$ is L1-equivalent to $(2, 1, 3)$. But definitely not $(1, 1, 3)$ with $(2, 1, 3)$ (there doesn't exist a valid permutation between them).

Now we have to solve another problem. We have several k-tuples (lists of constants). Two k-tuples are equivalent, if there exists a valid permutation mapping one k-tuple to another. The goal is to return just a single representative of each equivalence class to the output. And there is a simple solution.

Definition 14. *For a k-tuple t and an element c, that occurs in t, an* occ-strip *is an ordered list of indices of c in t. E.g. for a 6-tuple $(2, 5, 5, 3, 5, 7)$ and an element 5 it would be $(1, 2, 4)$.*

For a k-tuple t, a perm-strip *is the lexicographically ordered list of all of its occ-strips. E.g. for $(2, 5, 5, 3, 5, 7)$ it is $((0), (1, 2, 4), (3), (5))$.*

Theorem 20. *There exists a valid permutation between two k-tuples iff they have the same perm-strips.*

Proof. $\rightarrow$: Let there be a valid permutation P between two k-tuples t, u. $\forall i : u_i = P(t_i)$. The occ-strip of the element t_i would be the same as the occ-strip of the element u_i. Both k-tuples have the same sets of occ-strips, so their perm-strips must also be the same.

$\leftarrow$: Same perm-strip of two k-tuples t, u tells us, that specific elements occur at the same indices the same number of times in both k-tuples. Let's define the map P between constants, such that $P(a) = b$ when a occurs in t and is represented by the same occ-strip, as the b in u. When there are m constants, that don't occur in t, there must be exactly m constants, that don't occur in u. We can define P for these constants in any way, so that it is bijection for them. Such P is a permuation, mapping t to u. □

Occ-strips will be used for the second level of hashing in our algorithm. Let's finally write a pseudocode for the *pruneL1* subroutine.

Algorithm 6 Pruning actions based on L1-equivalency

```
 1: function PRUNEL1(Acts, L1)
 2:     out ← []
 3:     map ← new HashMap()
 4:     for all operator o do
 5:         ActsO ← elements of Acts generated by o
 6:         for all a in ActsO do
 7:             cs ← CLASS-STRIP(a, L1)
 8:             if map[cs] = null then map[cs] ← new HashMap();
 9:             os ← PERM-STRIP(a)
10:             if map[cs][os] = null then
11:                 map[cs][os] ← 1
12:                 out.PUSH(a)
13:     return out
```

Efficient implementation Planners usually work with a grounded representation of the problem, which has a fixed amount of possible actions throughout the whole search. Perm-strip can be precomputed for each action in advance (which can't be done for class-strip, because L1 is different in each state).

In our pruning method, creating new second-level hash maps and second-level hashing took a huge amount of time. However, in common domains, many operators are unable to have the same constant twice in a parameter list. In that case, the second-level hashing may be redundant.

Our improvement is based on the following observation: Let's have two actions, that have the same class-strip. If this class strip does not contain any value twice, these actions must have the same perm-strip.

Precisely, if a class-strip consists only of unique classes, the action has to consist of unique constants only (if some constant occured twice in an action, both occurences would have to be in the same L1-equivalence class, which would lead to duplicate values in a class-strip). In that case, the occ-strip must be $((0), (1), (2), \ldots, (n))$.

When the algorithm detects a class-strip consisting of unique values, it can set $map[cs] \leftarrow 1$, put an action to the output and proceed right to the next action. In the future, we can omit an action each time we find this class-strip again.

For example, in a Childsnack domain, the only operator capable of having the same constant multiple times is the operator $move(tray, p1, p2)$. Here, $p1, p2$ can be grounded with the same constant, resulting in the second and the third value of a class-strip being the same, which requires a second-level hashing. Luckily in this case, such actions have the set of positive effects the same as the set of negative effects, which means, that an action leads to the same state and is redundant. The architecture of our planner allows us to avoid such actions completely. Thus, second-level hashing never occurs in our planner for a Childsnack domain.

4.3 Implementational details

Rigid and fluent predicates As we have shown previously, the L1-equivalence of two constants can aslo be analyzed on each predicate separately. Constants are L1-equivalent, iff they are L1-eqiuvalent on all predicates.

We have also mentioned, that modern planners separate the initial state into rigid and fluent predicates. Rigid predicates remain true for all future states, while the state is represented just by fluent predicates.

We can calculate the L1-equivalence relation of constants for rigid predicates just once at the beginning. The computation of L1-equivalence for fluent predicates must be done at each state. The subroutine *findL1(S)* can get the second parameter - another equivalence relation. Then, before calling the union method with two constants, it can check, whether these constants are eqiuvalent also in the attached relation. This mechanism lets us pass the equivalence for rigid predicates to the computation of the equivalence for fluent predicates, to get an equivalence for all predicates.

Also, distinguishing between rigid and fluent predicates lets us have smaller occurence maps (smaller number of columns) when computing the equivalence relation in each state.

Preserving the reachability of the goal As we have mentioned earlier, choosing blindly any of several equivalent actions (and pruning others) can lead to unreachability of the goal.

In our planner, we solve it by the method, that was also mentioned earlier. Namely, by assuring, that each used automorphism is also the automorphism of the goal state. I.e. we compute the L1-equivalence relation of constants for the "goal state" (atoms, that must be present in the goal) at the beginning and merge it to the equivalence generated by rigid predicates (by passing it as the second parameter to *findL1*, as mentioned earlier). The actual L1 equivalence, that is used during pruning, consists of L1 for rigid predicates, L1 for fluent predicats at the state S and of L1 for the goal state, all merged together.

Not pruning actions when we don't have to For some states in some domains, it may happen, that no two constants are L1-equivalent. I.e. each constant is in its own equivalence class. In that case, each action would have a different class-strip and all actions would get to the output. The call of *pruneL1* subroutine is unnecessary.

We can easily modify the *findL1* subroutine to return not only the equivalence relation, but also the number of equivalence classes. At the beginning, we will make the number of eq. classes equal to the number of constants. Each time we call $union(i, j)$, we decrease the number of classes by one. By the end, we will return the number of classes as the second part of the output.

After calling *findL1(S)*, if the number of classes is equal to the number of constants, we can skip the pruning part and return all the input actions at the output.

In the same way, we can analyze the number of equivalence classes for rigid predicates at the beginning. If it is equal to the number of constants, we know, that no equivalences can occur in the future, and we can disalbe the whole pruning machinery completely, including the computation of L1-eqiuvalence relation on fluent predicates in each state.

Of course, extending a forward-search planner by this kind of pruning has a computational overhead. Later in this thesis, we will analyze the process of deciding, when enabling the pruning mechanism is efficient and when it is not.

5. Comparison to existing approaches

In previous chapters, we have created tools for studying symmetries in a classical representation and proposed algorithms for detecting such symmetries.

Now, let's try to look at the related work about symmetries one more time, but from the perspective of relational automorphisms. We will see, that we often talk about the same symmetries, even when we describe them in a different way, or convert the problem to a different representation.

The research of Fox and Long [1999] about symmetries is very closely related to our research. Authors do not describe a clear algorithm, however from the description it is obvious, that they are detecting the T1 subgroup of automorphisms. These automorphisms (represented as groups of objects, classes of the L1 equivalence) are detected only at the initial state and are used throughout the search. From this perspective, our method can be considered more general, since it looks for T1 automorphisms in each state, not only in the initial one.

The reachability of the goal in their work is guaranteed by using the first method, that we introduced earlier, i.e. by the condition, that each used automorphism must be an automorphism of the goal state (objects must be goal-equivalent).

In Riddle et al. [2015], authors focus at detecting T1 automorphisms as well. However, there are two main differences in their approach.

In the previous case, authors implemented their new method into the Graphplan algorithm, editing the actual process of the plan search. In the Bagged representation, authors reformulate the problem into a different representation (new PDDL files), without requiring any changes in the actual planner. In this case, the new representation can be solved by any planner, which is usually much faster in comparison with the original representation, especially when the planner has no built-in symmetry detection.

The second difference is in the way of preserving the reachability of the goal. While the previous solution checked, whether each equivalence is also the equivalence of the goal, in the Bagged Representation, we do not care about the goal that much. Equivalent objects (classes of L1-equivalence) are represented by new constants - bags, and cardinalities of each bag. A new goal is fulfilled, when specific bags reach some specific cardinality. In each state, a bag with a specific cardinality corresponds to a group of several objects, that have been equivalent in the initial state. There are many ways of attaching specific objects to undistinguishable items of the bag (represented just by the cardinality), i.e. the state in this new bagged representation corresponds to several isomorphic states of the original representation. Reaching the goal state (where bags reach specific cardinalities) corresponds to reaching some goal-equivalent state in the original representation. Then, objects in each bag are given actual values from the bag, which gives us a plan, that leads to the goal-equivalent state in the previous representation. Because the bags were created at the initial state, the plan can be converted to a real plan using an automorphism of the initial state (remapping of bagged objects).

To sum it up, Bagged representation allows us to find and exploit much more symmetries, than the previous solution, by not requiring the goal-equivalence. However, both methods exploit only the structure (and the automorphisms) of the initial state, while our method detects automorphisms in any state. We also see, that it is possible to perform a similar process (exploiting T1 automorphisms) without changing the representation of a problem, but instead, describing the same properties in a new way.

5.1 Orbit Search

The work of Pochter et al. [2011], Orbit Search, is very frequently cited in a relation to symmetry breaking in state-based planning. Several planners are using this method or extensions of it. However, there is a big difference, when compared with our methods: we defined symmetries in a classical representation of a problem, while Orbit Search is defined in a SAS+ representation of a problem. To be able to see the relation between these two different approaches, let's define several terms, which will be needed only in this section of the work.

Definition 15. *The STRIPS representation of planning is defined over a set of propositional facts. The STRIPS problem is a tuple (F, A, I, G), where:*

- *F is a set of facts*
- *A is a set of actions. Each $a \in A$, is a triple: $a = (pre(a), add(a), del(a))$. $pre(a) \subseteq F$ is a set of preconditions, $add(a) \subseteq F$ is a set of positive effects and $del(a) \subseteq F$ is a set of negative effects.*
- *$I \subseteq F$ is the initial state and $G \subseteq F$ is the goal state*

A state is a subset of facts: $s \subseteq F$. An action a is applicable to a state s, when $pre(a) \subseteq s$. By applying an action to a state we get another state: $\gamma(s, a) = (s - del(a)) \cup add(a)$.

A sequence of actions $(a_1, \ldots, a_n)$ is a plan, when actions are sequentially applicable to the initial state and $G \subseteq \gamma(\gamma(...\gamma(I, a_1)..., a_{n-1}), a_n)$.

Definition 16. *The SAS+ representation of planning is defined over a set of variables with finite domains. The SAS+ problem is a tuple (V, A, I, G), where:*

- *$V = \{v_1, \ldots v_n\}$ is a set of variables , each one has a domain $Dom(v_i)$.*
- *A is a set of actions. Each $a \in A$, is a pair $a = (pre(a), eff(a))$. $pre(a)$ and $eff(a)$ are partial assignments of the variables in V. A partial assignemnt contains values for some variables, e.g. $v_i = u, u \in Dom(v_i)$*
- *I is the initial state - a full asignment, G is the goal state - a partial assignment*

A state is a full assignment of all variables of V by some values from their domains. An action a is applicable to a state s, when the partial assignment $pre(a)$ corresponds to s. By applying an action to a state we get another state: $\gamma(s, a)$, which correspond to the previous assignment with some values updated according to $eff(a)$.

A sequence of actions $(a_1, \ldots, a_n)$ *is called a plan, when actions are sequentially applicable to the initial state and* G *is fulfilled in a state* $\gamma(\gamma(...\gamma(I, a_1)..., a_{n-1}), a_n)$. ∎

A classical representation of planning can be converted into a STRIPS representation in a process called *grounding*. All predicates are valuated by all constants to create facts. All operators are grounded to actions by attaching all possible constants as their parameters. More efficient grounding is usually based on relaxed search.

By converting a problem into STRIPS, each atom becomes a simple item without any internal structure. A fact is usually represented by a boolean variable. A state is usually represented by a vector of bits. The True value of some bit means, that the fact is present in the current state, and the False means, that the fact is missing in that state. The notion of original constants, different predicates and the structure of the state as a set of different relations is completely lost.

Let's describe the naive conversion of STRIPS to SAS+. Each fact becomes a variable with a domain $\{0, 1\}$. Preconditions are converted into partial assignements, where specific facts must have the value 1. Positive effects become partial assignments, giving facts the value 1, and negative effects become assignments, that give facts the value 0. The real effect of the action is the union of these two partial assignments. The initial state becomes a full assignment having 1 for the facts that are present in the initial state of STRIPS, and 0 otherwise. The goal is converted to a partial assignment, giving the value 1 for the goal facts. More efficient conversion may be possible, but this conversion is sufficient for our purpose.

In this thesis, we have defined mappings between constants, which imply mappings between atoms, which imply mappings between actions and between the whole states. By converting a problem into STRIPS and later into SAS+, the relational structure is completely lost. With no ability to access constants and relations, we must analyze the structure of actions, the initial and the goal states, and the facts or variables, that are used inside them.

Now we can describe automorphisms of Pochter et al. [2011]. Having a SAS+ representation of a problem, we define a PDG (problem description graph) for that problem, which has four categories of vertices (colors of vertices). Each SAS+ variable is converted to a vertex with the first color. Each value in each domain is converted into a vertex of the second color. There is an edge between a variable and all its values.

Each action is converted into a vertex of the third color (for preconditions) and a vertex of the fourth category (for effects) with an edge between them. There is an edge between a precondition vertex and a value of some variable iff that value is required by that precondition. Edges between effect vertices and values of variables are made in a similar way.

In our example of such graph, we named variables and actions according to the actual structure of original atoms in the classic representation. These names are not used by SAS+ algorithms, they just help us see the relation between two definitions of automorphisms.

Orbit Search uses color-preserving automorphisms of the PDG. Such automorphism maps each variable to another variable, each value to another value, and each action to another action. It also maps each partial or full assignment (a state) to another partial or full assignment.

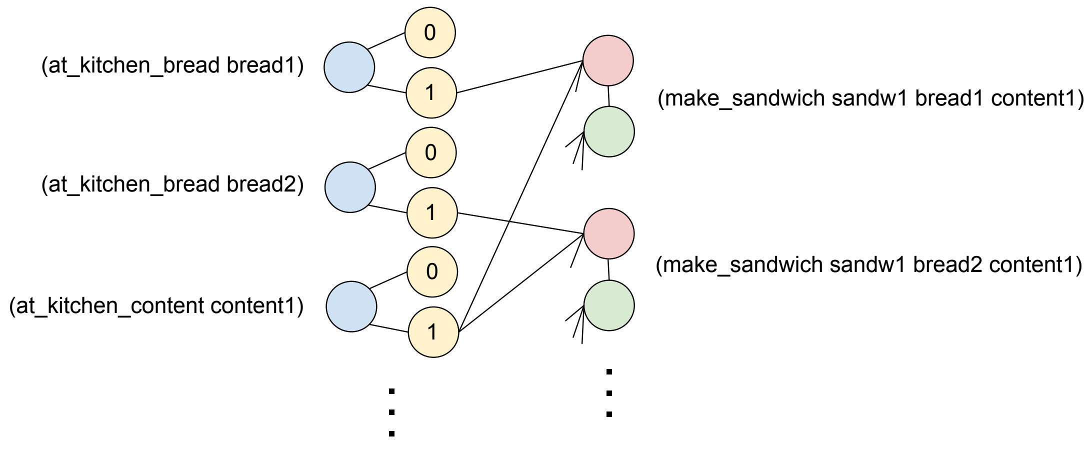

(at_kitchen_bread bread1)
(at_kitchen_bread bread2)
(at_kitchen_content content1)
0
1
0
1
0
1
(make_sandwich sandw1 bread1 content1)
(make_sandwich sandw1 bread2 content1)

Figure 5.1: PDG for Childsnack

Theorem 21. *Let S be a state in a classical (relational) representation and P be an automorphism of S. Actions A, B are applicable to S, $A = P(B)$, $\gamma(S, A) = P(\gamma(S, B))$.*

Let there be a naively created SAS+ representation for the same problem. For an atom X, let X' denote a corresponding variable, for an action A, let A' be a corresponding action in SAS+, for a state S, let S' be a corresponding state in SAS+.

Then, there exists a color-preserving automorphism P' of the PDG, such that A' and B' are applicable to S', $A' = P'(B')$, $\gamma'(S', A') = P'(\gamma'(S', B'))$

Proof. This property follows directly from the process of converting STRIPS to SAS+. When we define the mapping P' in correspondence with P (P maps an atom to an equivalent atom, so P' maps the variable to an equivalent variable with the corresponding name, etc.), the applicability of A', B' must be preserved, as well as the isomorphism of the subsequent states. It is easy to see, that such P' is definitely a color-preserving automorphism of the PDG. □

When in some state, `bread1` is equivalent to `bread2`, i.e. there is an automorphism consisting of a single transposition $(bread1/bread2)$, then the action `(make_sandwich sandw1 bread1 content1)'` and the action `(make_sandwich sandw1 bread2 content1)'` are equivalent and the subsequent states are isomorphic through a color-preserving automorphism, that swaps variable vertices, their value vertices, and action vertices, which have `bread1`, `bread2` in their names.

Advanced methods of converting STRIPS to SAS+ usually preserve this property. It means, that automorphisms detected by Orbit Search are more general than our relational automorphisms, since for each relational automorphism there exists a corresponding automorphism of the PDG. It also means, that in general, Orbit Search will always have a higher degree of pruning and will visit a smaller number of states.

But it does not necessarily mean, that Orbit Search is faster than our method. While our method performs *shallow pruning* - pruning the subsequent states from their parent in each state, Orbit Search works differently. When processing a new state, it tries to find an isomorphic state in previously visited states. If such state is found, the new state is omitted. Orbit Search is not implemented as the pruning of applicable actions, but as the modification of the A* algorithm itself.

The reachability of the goal in Orbit Search is preserved by ensuring, that each automorphism is an automorphism of the goal assignment (just like in Fox and Long [1999] and in our planner).

In this chapter, we have showed the relation between the original definition of object equivalence by Fox and Long [1999], the Bagged representation and the Orbit search. In short, we can say, that our method of pruning applicable actions according to T1 automorphisms in each state is more general than Bagged representation, but it is less general than the Orbit Search.

6. Experimental results

In previous chapters, we have introduced the concept of symmetries in a relational representation of planning and algorithms to detect and exploit a subclass of these symmetries. This detection requires a modification of the search process, it can not be done by simply reformulating the problem or extending some existing STRIPS / SAS+ planner, because we need an access to the relational representation of states.

We have created a planner called PLANR for the purpose of testing the proposed methods. It is a simple A* planner equipped with several basic heuristics. It can perform either a simple forward search in a relational representation, or the same search combined with "shallow pruning" of actions based on T1 automorphisms.

PLANR is equipped with three admissible and two inadmissible heuristics. The admissible heuristics are:

- h_{blind}: constant zero
- h_{goal}: corresponds to the number of goal atoms, that are not present in the state
- h_{max}: corresponds to g_s^{max} introduced by Bonet and Geffner [2001]

Inadmissible heuristics are:

- h_{add}: corresponds to g_s^+ introduced by Bonet and Geffner [2001]
- h_{FF}: corresponds h_{FF} heuristic introduced by Hoffmann and Nebel [2001]

Domains for testing We have selected six domains from the IPC, which are commonly used when demonstrating symmetry reduction techniques in planning. These domains are Gripper, Childsnack, Pipesworld-tankage, Pipesworld-notankage, Satellite, and Hiking. A brief description of these domains can be found in the Appendix of this work (1). Two of these domans come from the IPC 2014 (`https://helios.hud.ac.uk/scommv/IPC-14/`), others come from previous IPC competitions and are used as benchmarks of the Fast Downward planner (`https://bitbucket.org/aibasel/downward-benchmarks/src`).

In the Childsnack domain, it is very hard to find the optimal solution. During the latest IPC, the majority of participating planners was not able to find a single plan. We have created several simpler problems (with a smaller number of sandwiches and children), which we will use for testing.

The method of analyzing results There many ways of comparing planners according to their results. The usual way is to test all planners on multiple domains with a fixed set of problems for each domain. Then, planners are ranked according to the number of solved problems in each domain, which they have solved within a given time and memory limit. However, the complexity of problems within a specific domain varies a lot. There is usually one or two problems, which require more time than all other problems combined. Planner A can solve all problems a hudred times faster than planner B, but at the end, A may solve just one more problem than B for some domain.

Instead of focusing just on the number of solved problems, we will examine the actual time it took a planner to find a plan, and the number of visited states (states, that were taken from the heap in the A* algorithm). Since there are dozens of problems for each domain from the IPC, we have chosen just small subsets, which cover problems with different complexity, avoiding problems, that are too easy or too hard to solve.

We will split the tests into three sections. In the first section, we will compare the PLANR with other modern planners. In the second section, we will compare different configurations of the PLANR (different heuristics with enabled or disabled pruning) in finding optimal plans. In the third section, we will also compare different configurations of our planner, but in finding any satisfying plan.

For each problem, a planner will always have a time limit of 2 minutes and a memory limit of 2 GB to find the solution. Tests are run on a computer with Intel Core i5-3210M processor and 8 GB of RAM memory (no paging occurs). The best results are printed in bold. The cases, when a planner ran out of time or memory, are printed in gray.

6.1 Comparison with modern planners

In this section, we are going to compare PLANR with several modern planners. The goal is to find the optimal plan, so only admissible heuristics can be used.

The first configuration is PLANR combined with h_{goal} heuristic. This heuristic had the best results in our tests. h_{max} is usually more informative than h_{goal}, but it is also much harder to compute (see the following section).

The second configuration is the Metis planner. Metis is a planner composed out of three main blocks - incremental LM-cut heuristic, symmetry based pruning via Orbit Search algorithm and partial order reduction. It is a very advanced planner, which was one of the best planners of the latest International Planning Competition.

The third configuration is Fast Downward planner. It is also a very advanced planner, we will use it with the LM-cut heuristic. Many advanced planners, that participate in IPC, are based on Fast Downward (including Metis). It has been developed by multiple authors for several years.

	PLANR		Metis		FD	
	Time	States	Time	States	Time	States
prob15	3.4 s	156 273	**0.1 s**	187	120.0 s	-1
prob16	5.2 s	223 108	**0.1 s**	199	120.0 s	-1
prob17	7.5 s	315 595	**0.1 s**	211	120.0 s	-1
prob18	10.8 s	441 748	**0.1 s**	223	120.0 s	-1
prob19	15.8 s	612 797	**0.1 s**	235	120.0 s	-1
prob20	22.2 s	843 733	**0.1 s**	247	120.0 s	-1

Table 6.1: Gripper (runtime, visited states)

We can see, that PLANR is almost always faster than Fast Downward. It shows, that even a very simple planner with a trivial heuristic, which supports

	PLANR		Metis		FD	
	Time	States	Time	States	Time	States
p01_s2t2	0.0 s	336	**0.0 s**	78	0.0 s	592
p02_s3t3	0.2 s	13 413	**0.0 s**	1 792	2.1 s	142 145
p03_s4t3	5.3 s	271 533	**0.5 s**	20 176	108.4 s	4 958 025
p04_s5t3	92.9 s	4 075 587	**2.4 s**	61 083	120.0 s	-1

Table 6.2: Childsnack (runtime, visited states)

	PLANR		Metis		FD	
	Time	States	Time	States	Time	States
p03-net1-b8	**0.1 s**	3 011	0.1 s	292	1.1 s	7 506
p04-net1-b8	**0.3 s**	7 148	0.3 s	1 074	5.8 s	34 220
p07-net1-b1	**0.0 s**	380	0.4 s	133	28.0 s	53 542
p08-net1-b1	**1.1 s**	13 070	3.5 s	5 115	120.0 s	-1
p09-net1-b1	120.0 s	1 283 082	120.0 s	246 000	120.0 s	-1

Table 6.3: Pipesworld-t (runtime, visited states)

	PLANR		Metis		FD	
	Time	States	Time	States	Time	States
p07-net1-b1	**0.0 s**	193	0.1 s	116	0.1 s	775
p08-net1-b1	**0.0 s**	319	0.2 s	1 038	1.0 s	7 787
p09-net1-b1	5.4 s	128 904	**0.6 s**	3 795	7.0 s	35 219
p10-net1-b1	109.9 s	2 768 062	**46.9 s**	299 390	120.0 s	-1

Table 6.4: Pipesworld-nt (runtime, visited states)

	PLANR		Metis		FD	
	Time	States	Time	States	Time	States
p01-pfile1	0.0 s	58	0.0 s	24	**0.0 s**	42
p02-pfile2	0.1 s	8 297	**0.0 s**	50	0.0 s	68
p03-pfile3	0.1 s	5 861	**0.0 s**	92	0.0 s	326
p04-pfile4	19.8 s	573 750	**0.0 s**	119	0.0 s	432
p05-pfile5	41.5 s	753 507	**0.0 s**	391	0.8 s	14 308

Table 6.5: Satellite (runtime, visited states)

	PLANR		Metis		FD	
	Time	States	Time	States	Time	States
p-1-2-7	**0.6 s**	42 246	1.7 s	20 671	10.7 s	71 156
p-1-2-8	**1.5 s**	87 597	4.0 s	42 706	31.6 s	150 063

Table 6.6: Hiking (runtime, visited states)

pruning based on T1 automorphisms, can beat an advanced planner equipped with a state of the art heuristic. The only exception is the Satellite domain. We believe, that LM-cut heuristic (that is present in Metis and FD) is extremely helpful for this domain. The effect of this heuristic on the number of visited states is even more significant than the effect of pruning in PLANR.

Metis planner was almost always faster than PLANR. The exception was Pipesworld-tankage and Hiking domains. Even though Metis always visits fewer states than PLANR, it performs pruning in a different way, which may have an additional overhead.

6.2 Optimal plans

Pruning symmetries is supposed to reduce the number of visited states and the time of the search. However, there may be heuristics, for which the pruning does not lead to any improvement.

In following tests, we compare multiple configurations of PLANR with each other. We compare blind search with two simple admissible heuristics. All cases are tested with pruning enabled and disabled.

	h_{blind}		h_{goal}		h_{max}	
	none	T1	none	T1	none	T1
prob15	80.7 s	18.8 s	52.1 s	**3.4 s**	120.0 s	72.7 s
	6 762 586	844 483	1 935 682	156 273	1 019 916	677 365
prob16	67.3 s	29.3 s	57.7 s	**5.2 s**	120.0 s	112.4 s
	4 898 162	1 291 479	2 253 739	223 108	1 180 856	1 043 287
prob17	66.4 s	49.4 s	58.6 s	**7.5 s**	121.3 s	120.0 s
	3 820 583	1 951 344	2 597 648	315 595	1 012 098	1 041 990
prob18	59.2 s	73.6 s	64.6 s	**10.8 s**	120.0 s	120.0 s
	3 084 713	2 915 972	3 065 271	441 748	814 150	953 115
prob19	70.8 s	116.8 s	120.0 s	**15.8 s**	120.0 s	120.0 s
	3 531 554	4 313 562	3 571 183	612 797	532 171	940 181
prob20	71.7 s	120.0 s	120.0 s	**22.2 s**	120.0 s	120.0 s
	4 164 512	4 313 840	2 963 180	843 733	400 489	922 779

Table 6.7: Gripper (runtime, visited states)

	h_{blind}		h_{goal}		h_{max}	
	none	T1	none	T1	none	T1
p01_s2t2	0.0 s	0.0 s	0.0 s	**0.0 s**	0.1 s	0.0 s
	1 515	695	1 161	336	578	150
p02_s3t3	5.5 s	1.4 s	3.7 s	**0.2 s**	10.3 s	1.3 s
	251 746	47 774	141 157	13 413	123 672	17 290
p03_s4t3	120.0 s	29.6 s	120.0 s	**5.3 s**	120.0 s	32.3 s
	3 907 606	797 759	3 434 717	271 533	832 230	357 741
p04_s5t3	96.5 s	120.0 s	104.8 s	**92.9 s**	120.0 s	120.0 s
	1 159 921	2 548 206	1 063 889	4 075 587	304 576	918 359

Table 6.8: Childsnack (runtime, visited states)

Tests show, that for every domain and every heuristic, the search with pruning always performs better than the search with no pruning. As expected, heuristics h_{goal} and h_{max} always lead to visiting fewer states than during the blind search. Sometimes, h_{max} gives better performance than the blind search (e.g.

	h_{blind}		h_{goal}		h_{max}	
	none	T1	none	T1	none	T1
p03-net1-b8	1.7 s	0.7 s	0.4 s	**0.1 s**	2.1 s	0.3 s
	69 895	16 221	14 847	3 011	3 251	759
p04-net1-b8	7.9 s	5.8 s	0.9 s	**0.3 s**	48.9 s	7.7 s
	387 077	135 956	32 433	7 148	75 422	17 064
p07-net1-b1	52.1 s	80.1 s	0.4 s	**0.0 s**	120.0 s	41.2 s
	866 437	960 759	5 486	380	36 669	25 946
p08-net1-b1	58.3 s	120.0 s	33.0 s	**1.1 s**	120.0 s	120.0 s
	866 437	1 323 456	348 584	13 070	33 062	59 057
p09-net1-b1	56.1 s	122.4 s	71.0 s	120.0 s	120.0 s	120.0 s
	810 877	1 278 514	856 867	1 283 082	16 679	28 674

Table 6.9: Pipesworld-t (runtime, visited states)

	h_{blind}		h_{goal}		h_{max}	
	none	T1	none	T1	none	T1
p07-net1-b1	5.4 s	4.3 s	0.0 s	**0.0 s**	1.6 s	1.1 s
	123 146	66 025	291	193	5 601	4 034
p08-net1-b1	25.2 s	24.4 s	0.0 s	**0.0 s**	11.3 s	6.2 s
	580 725	375 493	550	319	36 549	20 244
p09-net1-b1	120.0 s	120.0 s	24.1 s	**5.4 s**	120.0 s	116.2 s
	2 160 336	1 425 366	373 557	128 904	203 203	215 633
p10-net1-b1	120.0 s	120.0 s	120.0 s	109.9 s	120.0 s	120.0 s
	2 180 743	1 387 878	1 931 395	2 768 062	189 778	198 950

Table 6.10: Pipesworld-nt (runtime, visited states)

	h_{blind}		h_{goal}		h_{max}	
	none	T1	none	T1	none	T1
p01-pfile1	0.0 s	0.0 s	0.0 s	**0.0 s**	0.0 s	0.0 s
	610	210	119	58	144	65
p02-pfile2	9.2 s	4.0 s	0.3 s	**0.1 s**	10.0 s	2.8 s
	1 125 328	303 352	27 444	8 297	125 457	38 410
p03-pfile3	45.4 s	60.7 s	**0.1 s**	0.1 s	39.6 s	38.6 s
	2 406 931	2 448 346	5 899	5 861	181 185	176 759
p04-pfile4	43.6 s	77.7 s	29.9 s	**19.8 s**	120.0 s	120.0 s
	1 903 913	2 239 917	947 352	573 750	373 199	404 098
p05-pfile5	31.9 s	52.5 s	43.6 s	**41.5 s**	120.0 s	120.0 s
	975 359	1 056 920	966 546	753 507	150 531	164 621

Table 6.11: Satellite (runtime, visited states)

in Pipesworld-notankage). However, h_{max} is computationally expensive and may lead to worse performance than the blind search (e.g. in Gripper).

	h_{blind} none	h_{blind} T1	h_{goal} none	h_{goal} T1	h_{max} none	h_{max} T1
p-1-2-7	0.8 s	0.6 s	1.0 s	**0.6 s**	120.0 s	103.8 s
	79 522	43 474	77 985	42 246	45 838	40 525
p-1-2-8	1.8 s	1.5 s	2.1 s	**1.5 s**	120.0 s	120.0 s
	165 882	89 513	163 487	87 597	30 603	31 397

Table 6.12: Hiking (runtime, visited states)

6.3 Satisfying plans

Finding satisfying plans, without the optimality requirement, can be considered as a special field of automated planning, mainly because specialized methods can be used, which don't guarantee optimality of the solution. We will find satisfying plans using heuristics, which are highly informative, but not admissible.

Highly informative inadmissible heuristics are known to drive the A* algorithm in a single direction. We can often see, that the number of visited states is equal to the length of the plan plus one (because of the initial state). It is not clear, if pruning can be useful even in these cases.

We have chosen slightly different subsets of problems for testing in this case. Since plans are usually easier to find, we can try solving harder problems than in the case of optimal plans. Besides the time of the search and the number of visited states, we also show a third value, the cost of the plan.

	h_{add} none		h_{add} T1		h_{FF} none		h_{FF} T1	
prob15	120.0 s	-1	**0.0 s**	125	120.0 s	-1	3.6 s	95
	310 855		495		396 411		16 669	
prob16	120.0 s	-1	**0.1 s**	133	120.0 s	-1	4.9 s	101
	260 329		580		373 056		21 711	
prob17	120.0 s	-1	**0.1 s**	141	120.0 s	-1	7.1 s	107
	238 714		682		345 690		27 946	
prob18	120.0 s	-1	**0.1 s**	149	120.0 s	-1	9.9 s	113
	222 366		798		332 880		35 780	
prob19	120.0 s	-1	**0.1 s**	157	120.0 s	-1	13.7 s	119
	213 313		921		326 193		45 543	
prob20	120.0 s	-1	**0.2 s**	165	120.0 s	-1	15.9 s	125
	186 949		1 060		315 321		57 587	

Table 6.13: Gripper (runtime, plan cost, visited states)

The h_{FF} heuristic always gives us better plans (with a smaller cost) than the h_{add} heuristic. Finding plans h_{FF} with usually take more time, but not always. If we look at Pipesworld-notangake, h_{FF} gives us better plans in a shorter time than h_{add}.

Let's focus at the effect of pruning. It is extremely helpful in Gripper and Childsnack domains. In Pipesworld-tankage and Pipesworld-notankage, pruning leads to finding better plans (with a smaller cost).

	h_{add}		h_{FF}	
	none	T1	none	T1
p02_s3t3	2.0 s 11	**0.1 s** 11	10.7 s 10	0.5 s 10
	24 736	639	30 221	1 698
p03_s4t3	83.8 s 14	**0.5 s** 14	120.0 s -1	19.8 s 14
	715 731	4 933	181 134	51 332
p04_s5t3	120.0 s -1	**11.8 s** 18	120.0 s -1	120.0 s -1
	579 399	87 715	64 391	216 048
p05_s6t3	120.0 s -1	**60.7 s** 21	120.0 s -1	120.0 s -1
	288 183	372 784	22 714	158 393

Table 6.14: Childsnack (runtime, plan cost, visited states)

	h_{add}		h_{FF}	
	none	T1	none	T1
p09-net1-b1	80.4 s 29	**1.4 s** 26	120.0 s -1	120.0 s -1
	29 033	360	5 050	10 831
p10-net1-b1	**1.2 s** 29	1.4 s 25	120.0 s -1	47.6 s 22
	248	734	7 367	3 658
p11-net2-b1	3.1 s 22	**1.3 s** 22	77.4 s 22	35.2 s 22
	8 843	4 141	56 243	29 703
p12-net2-b1	**25.6 s** 32	120.0 s -1	120.0 s -1	77.4 s 24
	20 456	242 351	41 617	35 079
p13-net2-b1	20.3 s 22	**0.3 s** 18	119.4 s 16	9.1 s 16
	16 808	501	24 352	2 612

Table 6.15: Pipesworld-t (runtime, plan cost, visited states)

	h_{add}		h_{FF}	
	none	T1	none	T1
p19-net2-b1	4.6 s 30	2.0 s 30	2.8 s 24	**1.6 s** 24
	2 939	1 226	1 108	714
p20-net2-b1	2.2 s 44	1.2 s 42	0.9 s 28	**0.7 s** 28
	1 652	868	352	282
p21-net3-b1	3.6 s 14	3.3 s 14	**0.4 s** 14	**0.4 s** 14
	9 032	8 241	335	335
p22-net3-b1	120.0 s -1	120.0 s -1	21.4 s 30	**21.2 s** 30
	155 503	154 150	14 534	14 478
p23-net3-b1	0.6 s 28	**0.5 s** 28	59.2 s 20	24.6 s 20
	748	788	32 345	12 591
p24-net3-b1	20.0 s 36	13.1 s 36	1.6 s 24	**1.4 s** 24
	54 216	33 044	730	642
p25-net3-b1	**5.1 s** 46	9.2 s 49	120.0 s -1	120.0 s -1
	2 171	9 194	29 652	12 052

Table 6.16: Pipesworld-nt (runtime, plan cost, visited states)

The effect of pruning is very interesting in the Satellite domain for h_{add} heuristic. The number of visited states corresponds to the cost of the plan. Pruning

	h_{add}				h_{FF}			
	none		T1		none		T1	
p17-pfile17	6.3 s	43	**4.6 s**	43	120.4 s	-1	120.4 s	-1
		44		44		129		178
p18-pfile18	0.7 s	32	**0.4 s**	32	120.1 s	-1	120.0 s	-1
		33		33		754		1 260
p19-pfile19	1.6 s	62	**1.3 s**	62	120.0 s	-1	120.1 s	-1
		64		64		586		665
p20-pfile20	120.0 s	-1	120.0 s	-1	120.3 s	-1	120.2 s	-1
		4 459		4 776		470		512
p21-HC-pfil	11.6 s	74	**4.1 s**	74	120.2 s	-1	56.5 s	73
		75		75		144		240
p22-HC-pfil	120.3 s	-1	**20.7 s**	91	121.3 s	-1	82.1 s	90
		312		215		76		194

Table 6.17: Satellite (runtime, plan cost, visited states)

	h_{add}				h_{FF}			
	none		T1		none		T1	
p-2-2-6	120.1 s	-1	120.0 s	-1	120.0 s	-1	120.0 s	-1
		2 127		3 337		1 751		2 676
p-2-2-7	2.0 s	11	**0.7 s**	10	120.1 s	-1	120.0 s	-1
		14		11		451		975
p-2-2-8	120.2 s	-1	120.0 s	-1	120.1 s	-1	120.1 s	-1
		790		1 445		929		1 218
p-2-3-6	120.1 s	-1	120.0 s	-1	120.1 s	-1	120.1 s	-1
		1 432		1 992		771		1 284

Table 6.18: Hiking (runtime, plan cost, visited states)

does not change the cost or the number of visited states. However, plans are found almost two times faster. Our pruning leads to fewer applicable actions for each state, i.e. to fewer states being inserted into the heap. The heuristic value must be computed for these states, even if they are never taken from the heap (never visited by the planner). Working with more states requires extra computation.

Our experiments show, that the search with pruning almost always performs better than the search without pruning for selected domains. Especially in highly symmetric domains such as Gripper or Childsnack, we can't see a chance of finding an optimal solution without the symmetry detection. We also see, that a very simple A* planner equipped with the proposed symmetry pruning can reach the performance of modern planners equipped with very advanced heuristics.

7. Estimating the pruning rate

Pruning methods, that have been introduced earlier, always have some computational overhead, depending on the quality of implementation and on the structure of a specific problem. Moreover, they aren't always useful. For example, if each constant has its own L1-equivalence class in all states, no action can ever be pruned and the detection of T1 automorphisms could be disabled completely.

We have already mentioned several cases, when we can disable pruning to improve performance. When there are no L1-equivalences between constants in rigid predicates, we can disable pruning, since rigid predicates are present in all states. However, this case is very rare. When no two constants are L1-equivalent in some state, we can skip filtering actions. However, we will still have to compute the L1-equivalence relation to detect such situation, including the construction of an occurence map for the state.

It would be very helpful to be able to estimate the rate of prunning in advance, before performing the actual search. Based on that estimation, we can disable pruning completely to avoid possibly useless computation.

Problems from IPC usually have very "regular" initial states, i.e. they have many relational automorphisms in these states. Some predicates are completely empty (e.g `at_kitchen_sandwich`, `no_gluten_sandwich`, `ontray` and `served` in problems of a Childsnack domain) and every two constants are L1-equivalent for an empty predicate.

We want to estimate the ratio R_p between the number of visited states when pruning is disabled and the number of visited states when pruning is enabled. E.g. if the planner visits 5000 states with pruning disabled and 1000 states with pruning enabled, $R_p = 5$. Our estimates should be based on the structure of L1 equivalence in initial states.

Definition 17. *Let the problem have #obj objects and #cls classes of L1 equivalence in the initial state. Then* $\phi_0 = \frac{\#obj}{\#cls}$.

This first estimate reflects the number of L1 equivalency classes, but it does not reflect the size of equivalence classes. If the problem has 10 constants, equivalency classes have sizes $(2, 2, 2, 2, 2)$ in the first case and $(5, 1, 1, 1, 1)$ in the second case, the estimate ϕ_0 will be still the same for both cases. However, there are much more T1 automorphisms in the second case (2^5 vs. $5!$). Let's present another estimate.

Definition 18. *Let* $(C_1, \dots C_n)$ *be the equivalence classes of L1 in the initial state. Then* $\phi_1 = \sqrt[3]{\prod_{i=1}^{n} |C_i|^2}$.

The second estimate reflects the sizes of classes of L1 equivalence much better than ϕ_0. These estimates consider the structure of L1 equivalence, but the actual pruning rate also strongly depends on the total number of actions, their structure, arity etc. Let's define another estimate, which takes actions into account.

Definition 19. *Let* A_0 *be all actions applicable to the initial state and* $A_1 \subseteq A_0$ *corresponds to these actions pruned by T1 automorphisms of the initial state. Then* $\phi_2 = \frac{|A_0|}{|A_1|}$.

The estimate ϕ_2 corresponds to the rate of pruning of the initial state. All three estimates equal to one when there are no T1 automorphisms in the initial state and they increase, as the number of automorphisms increases. Let's have a look at the actual values of ϕ_0, ϕ_1, ϕ_2 and R_p in our problems. We also print sizes of classes of L1 equivalence for each problem.

	ϕ_0	ϕ_1	ϕ_2	R_p h_{blind}	R_p h_{goal}	R_p h_{max}
prob15	9.00	16.00	32.50	>8.01	>12.39	>1.51
	(32, 2, 1, 1)					
prob16	9.50	16.66	34.50	>3.79	>10.10	>1.13
	(34, 2, 1, 1)					
prob17	10.00	17.31	36.50	>1.96	>8.23	
	(36, 2, 1, 1)					
prob18	10.50	17.94	38.50	>1.06	>6.94	
	(38, 2, 1, 1)					
prob19	11.00	18.57	40.50	>0.82	>5.83	
	(40, 2, 1, 1)					
prob20	11.50	19.18	42.50		>3.51	
	(42, 2, 1, 1)					

Table 7.1: Gripper

	ϕ_0	ϕ_1	ϕ_2	R_p h_{blind}	R_p h_{goal}	R_p h_{max}
p01_s2t2	1.40	6.35	3.60	2.18	3.46	3.85
	(2, 2, 2, 2, 1, 1, 1, 1, 1, 1)					
p02_s3t3	1.73	27.47	6.86	5.27	10.52	7.15
	(3, 3, 2, 2, 2, 2, 1, 1, 1, 1, 1)					
p03_s4t3	1.77	52.83	11.13	>4.90	>12.65	>2.33
	(4, 3, 2, 2, 2, 2, 2, 1, 1, 1, 1, . . .)					
p04_s5t3	1.93	105.26	19.25		>0.26	
	(5, 3, 3, 3, 2, 2, 2, 1, 1, 1, 1, . . .)					

Table 7.2: Childsnack

	ϕ_0	ϕ_1	ϕ_2	R_p h_{blind}	R_p h_{goal}	R_p h_{max}
p03-net1-b8	1.22	25.40	1.38	4.31	4.93	4.28
	(2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, . . .)					
p04-net1-b8	1.22	25.40	1.38	2.85	4.54	4.42
	(2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, . . .)					
p07-net1-b1	1.32	256.00	3.25	>0.90	14.44	>1.41
	(2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, . . .)					
p08-net1-b1	1.32	256.00	3.71		26.67	
	(2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, . . .)					

Table 7.3: Pipesworld-t

Let's have a closer look at the results for Gripper domain. There are four L1-equivalence classes at the initial state of each problem. It is easy to deduce,

	ϕ_0	ϕ_1	ϕ_2	R_p h_{blind}	R_p h_{goal}	R_p h_{max}
p07-net1-b1	1.10	2.52	1.13	1.87	1.51	1.39
	(2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, . . .)					
p08-net1-b1	1.10	2.52	1.29	1.55	1.72	1.81
	(2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, . . .)					
p09-net1-b1	1.14	4.00	1.22		2.90	>0.94
	(2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, . . .)					

Table 7.4: Pipesworld-nt

	ϕ_0	ϕ_1	ϕ_2	R_p h_{blind}	R_p h_{goal}	R_p h_{max}
p01-pfile1	1.50	5.24	1.75	2.90	2.05	2.22
	(3, 2, 2, 1, 1, 1, 1, 1)					
p02-pfile2	1.17	2.08	1.29	3.71	3.31	3.27
	(3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)					
p03-pfile3	1.00	1.00	1.00		1.01	1.03
	(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, . . .)					
p04-pfile4	1.13	2.52	1.24		1.65	
	(2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, . . .)					
p05-pfile5	1.04	1.59	1.09		1.28	
	(2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, . . .)					

Table 7.5: Satellite

	ϕ_0	ϕ_1	ϕ_2	R_p h_{blind}	R_p h_{goal}	R_p h_{max}
p-1-2-7	1.08	1.59	1.96	1.83	1.85	>1.13
	(2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)					
p-1-2-8	1.08	1.59	1.97	1.85	1.87	
	(2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, . . .)					

Table 7.6: Hiking

which constants are equivalent. The largest class contains all balls. The next class contains two empty grippers. Two rooms remain, each of them is in a separate class. When there are N balls, the value of ϕ_2 is always $N + 0.5$. In the unpruned case, we can pick one of N balls with each gripper (which gives us $2N$ applicable actions) or move to another room, $2N + 1$ actions in total. In the pruned case, there is just one action of picking a specific ball with a specific gripper, because other actions are equivalent, and we still can move to another room, two actions in total. Then, $\phi_2 = \frac{2N+1}{2} = N + 0.5$.

In practice, pruning can be disabled, when ϕ_0, ϕ_1 and ϕ_2 are sufficiently small (close to one). The actual limit will depend on a specific implementation, i.e. on how much extra computation is required by the pruning mechanisms.

Equivalent constants in the initial state don't always mean the reduction of the number of visited states. Let's have a look at the following problem.

```
Consts = { loc0, loc1, loc2, p0, p1 }
I = {  (arc loc0 loc1), (arc loc1,loc0),
       (arc loc1 loc2), (arc loc2 loc1),
       (at p0 loc0), (at p1 loc0)  }
```

```
G = {  (at p0 loc2), (at p1 loc2)  }
```

The problem consists of three locations and two packages. It is possible to move packages in the expected way. Packages are L1-equivalent at the initial state, they are also L1-equivalent at the goal. When our pruning is performed at the initial state, only one package can be moved. However, it is easy to see, that the number of reachable states does not change with pruning enabled.

Packages are equivalent, when they are at the same location. In that case, we always move only `p0`. When `p0` is at `locX` and `p1` is at `locY`, we can show, that the state with `p0` at `locY` and `p1` at `locX` is also reachable even with pruning enabled. When `locX = locY`, it holds trivially. Otherwise, packages are not equivalent, `p1` can be moved to `locX` (withoug being equivalent at any point in between). From that point, `p0` can be moved to `locY` even with pruning enabled.

On the other hand, if there are no equivalent constants in the initial state, some equivalent constants may appear in other reachable states. When we look at the third problem in the Satellite domain, there are no equivalent constants in the initial state (since $\phi_0 = \phi_1 = \phi_2 = 1$), but some pruning occured during the search (since $R_p > 1$).

The initial state is not a perfect way of estimating the pruning rate, but as we see from the results, it can be a very good guide for such estimation.

Conclusion

In this thesis, we analyzed existing approaches to symmetry breaking in automated planning. Such approaches usually have a very specific definition, e.g. they strongly depend on the initial and the goal states. Another class of symmetries, that was defined for a SAS+ representation, seemed to be completely unrelated to Bagged Representation and others.

We made a new definition of symmetries through relational automorphisms. We defined equivalency of states, T1 and T2 classes of automorphisms, and relations between them. It may seem like the previous work was progressing into definitions like these. We believe, that our definitions generalize and simplify some works of the previous research in this area.

We also proposed an efficient algorithm for detecting $\langle T1 \rangle$ subgroup of automorphisms, which turned out to be very helpful for many domains. Our methods are very simple and can be easily implemented into any existing planner based on forward search.

The results show, that even such simple method can reduce the search space significantly by avoiding symmetrical actions. Our simple planner with a trivial heuristic was able to outperform modern advanced planners with state of the art heuristics in solving some specific problems.

It may be very useful to define other special classes of automorphisms, that are easy to detect and exploit, or to apply our detection of T1 automorphisms in other areas, such as graph theory.

Bibliography

Jorge A. Baier, Christian Fritz, and Sheila A. McIlraith. Exploiting procedural domain control knowledge in state-of-the-art planners. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS-07)*, Providence, Rhode Island, September 22 - 26 2007. URL `http://www.cs.toronto.edu/~jabaier/publications/bai-fri-mci-icaps07.pdf`.

Blai Bonet and Hector Geffner. Planning as heuristic search. *Artif. Intell.*, 129 (1-2):5–33, 2001. doi: 10.1016/S0004-3702(01)00108-4. URL `http://dx.doi.org/10.1016/S0004-3702(01)00108-4`.

Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In *Proceedings of the 2Nd International Joint Conference on Artificial Intelligence*, IJCAI'71, pages 608–620, San Francisco, CA, USA, 1971. Morgan Kaufmann Publishers Inc. URL `http://dl.acm.org/citation.cfm?id=1622876.1622939`.

Maria Fox and Derek Long. The detection and exploitation of symmetry in planning problems. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, IJCAI '99, pages 956–961, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1-55860-613-0. URL `http://dl.acm.org/citation.cfm?id=646307.687897`.

M. Ghallab, D.S. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann Series in Artificial Intelligence. Elsevier/Morgan Kaufmann, 2004. ISBN 9781558608566. URL `https://books.google.cz/books?id=eCj3cKC_3ikC`.

Malte Helmert. Concise finite-domain representations for pddl planning tasks. *Artif. Intell.*, 173(5-6):503–535, April 2009. ISSN 0004-3702. doi: 10.1016/j.artint.2008.10.013. URL `http://dx.doi.org/10.1016/j.artint.2008.10.013`.

Jörg Hoffmann and Bernhard Nebel. The ff planning system: Fast plan generation through heuristic search. *J. Artif. Int. Res.*, 14(1):253–302, May 2001. ISSN 1076-9757. URL `http://dl.acm.org/citation.cfm?id=1622394.1622404`.

Johannes Köbler, Uwe Schöning, and Jacobo Torán. *The Graph Isomorphism Problem: Its Structural Complexity*. Birkhauser Verlag, Basel, Switzerland, Switzerland, 1993. ISBN 0-8176-3680-3.

Nir Pochter, Aviv Zohar, and Jeffrey S. Rosenschein. Exploiting problem symmetries in state-based planners. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, AAAI'11, pages 1004–1009. AAAI Press, 2011. URL `http://dl.acm.org/citation.cfm?id=2900423.2900583`.

Patricia J. Riddle, Michael W. Barley, Santiago Franco, and Jordan Douglas. Automated transformation of PDDL representations. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015, 11-13 June*

2015, Ein Gedi, the Dead Sea, Israel., pages 214–215, 2015. URL `http://www.aaai.org/ocs/index.php/SOCS/SOCS15/paper/view/11166`.

Alexander Shleyfman, Michael Katz, Malte Helmert, Silvan Sievers, and Martin Wehrle. Heuristics and symmetries in classical planning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 3371–3377. AAAI Press, 2015. ISBN 0-262-51129-0. URL `http://dl.acm.org/citation.cfm?id=2888116.2888185`.

Appendix

1. The list of used domains with a brief description of each domain

    - Gripper - there are two rooms, multiple balls are located at one room and the goal is to move all the balls to another room. There is a robot with two grippers, which can pick balls, move between rooms and drop balls.
    - Childsnack - there are several pieces of bread and content available, some of them are gluten-free. One piece of bread and one piece of content are required to make a sandwich. Sandwiches can be put on trays. Trays can be moved between kitchen and tables. Children are waiting by the tables. Some children are allergic to gluten and they must receive a gluten-free sandwich. A sandwich can be served to a child, when the tray is located at the right table. The goal is to serve all children, who are waiting for a sandwich.
    - Pipesworld-tankage - this domain models the flow of liquids through pipeline segments. It was inspired by the oil industry.
    - Pipesworld-notankage - very similar to the previous domain, but has several additional restrictions
    - Satellite - several satellites are located in space. They can turn to different directions, switch instruments, calibrate and take images. The goal is to take images of different space objects.
    - Hiking - there are several persons, that create couples, and a chain of several places. The goal is to get all couples from the first place to the last place by walking between these places. Before a couple can walk together to the next place, there must be a tent put up at the new location. There are several cars and tents, anybody can drive to any location and put up a tent there.