Bachelor of Science Thesis

Planning with Sets: Exploring Different Encodings

Leon Spitzer

Examiners:
Prof. Dr. Erika Ábrahám
Prof. Gerhard Lakemeyer, Ph.D.

Additional Advisor:
Francesco Leofante, Ph.D.

Aachen, 02.09.2022

Abstract

Much work has been put into extending the scope of planning beyond using propositional domains. We present an extension to the modelling language PDDL that allows for set creation and reasoning with set semantics in a planning domain. We show that modelling efforts can be reduced by exploiting set theory. We then present two encodings that reduce the problem of planning with sets to Satisfiability Modulo Theories, using propositional and a bit vector theories. We provide an implementation extending OMTPlan, an SMT-based planner, and evaluate it on existing benchmarks from the planning literature. The evaluation shows that the increased expressivity of the language does indeed ease the modelling for well-known planning domains; however this also introduces extra overhead that affect the performance of the planner.

Erklärung

Hiermit versichere ich, dass ich die vorgelegte Arbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Leon Spitzer
Aachen, den 02. September 2022

Acknowledgements

I want to thank Prof. Dr. Erika Ábrahám and Francesco Leofante, Ph.D. for providing me with excellent guidance throughout my thesis. I also thank Prof. Gerhard Lakemeyer for being my second supervisor.

Contents

Chapter 1

Introduction

1.1 Planning in AI

Planning in AI is the task of finding an applicable sequence of actions that lead to a satisfying outcome. The domain of a planning problem is a collection of states and the actions in a plan transform these states step by step. A planning problem can be visualized with a directed graph with nodes as states and edges as actions that switch between the states. A valid plan is a sequence of applicable actions that start in the initial state and end in a desired goal state.

A planner is a program that tries to find a valid plan to a planning problem. Explicitly searching the state space was the earliest method for automatically solving planning problems. While this approach can be efficient, it has scalability issues in larger state spaces. The *Planning as Satisfiability* approach, introduced by Kautz and Selman [KS92], reduces a planning problem to Satisfiability Checking (SAT) which in turn can be solved by existing SAT solvers. This approach can be extended to SAT Modulo Theories (SMT) which allows to reason over one or more theories. A planner can use the latest SMT solver and take advantage of higher expressivity that can be leveraged.

To describe planning problems the *Planning Domain Definition Language* (PDDL) was developed by McDermott in 1998 [McD00]. It allows for precise descriptions of tasks and facilitated the exchange and comparison of planners. PDDL was developed for the *International Planning Competition* (IPC) which hosts events for planners to compete over a growing collection of benchmarks since 1998.

Since the creation of PDDL it got regularly extended by the planning community. From using only propositional variables in PDDL1.2 [McD00], more sophisticated problems including time and numbers could be modelled in PDDL2.1 [FL03]. In this thesis we propose an extension of PDDL that allows for reasoning with set theory in planning problems. We formalize its semantics and present two different SMT encodings. We propose an implementation extending OMTPlan, an SMT based planner, that utilizes our extension and evaluate it on a collection of benchmarks from the IPC.

1.2 Outline

We first introduce a formal definition of Classical Planning in Chapter 2 Section 2.1 and Planning as Satisfiability in Section 2.2. Then we thoroughly look at PDDL in Section 2.3. We introduce our new set extension to PDDL in Chapter 3 Section 3.1 and show Propositional 3.2 and Bit Vector 3.3 encodings. In Chapter 4 we evaluate our extension on benchmarks from the IPC and compare them to equivalent problems modelled without our extension. Lastly, we summarize findings in Chapter 5 and conclude about future work in extending PDDL with sets.

Chapter 2

Preliminaries

2.1 Classical Planning

The formal model of a *classical planning problem* can be described as follows:

Definition 2.1.1. *(Classical Planning Problem). A planning problem* $\Pi = \langle V, A, I, G \rangle$ *consists of:*

- *A finite set of propositional variables* V*,*
- *A set of actions* A*,*
- *An initial condition* I*,*
- *A goal specification* G*.*

A state in Π is a function $s : V \rightarrow \mathbb{B}$, which assigns a value $s(v) \in \mathbb{B}$ to every variable $v \in V$. Each action $a \in A$ can be modelled by a tuple $a = (pre_a, eff_a, c_a)$.

pre_a is the precondition of a, which consists of a set of constraints and eff_a is an effect consisting of a set of assignments $v := e$ for $v \in V$ and $e \in \{\top, \bot\}$ meaning *true* or *false*. eff_a contains at most one assignment $v := e$ for each variable $v \in V$. Lastly, c_a is a positive rational which denotes the cost of action a. In this work we assume a uniform cost of 1 for each action.

The evaluation function $[\![\cdot]\!]_s$ and the satisfaction relation $\models$ for expressions, constraints and conditions are as usual. A state s' is called successor state from state s and effects Ψ, if s' results from applying effects in Ψ to s and the other variables not affected from Ψ remain unchanged. Formally, for some state s and effect Ψ, the successor of s and Ψ is the unique state s' with $s'(v) = [\![e]\!]_s$ for each assignment $v := e$ in Ψ, and $s'(v) = s(v)$ for each $v \in V$ that is not assigned in Ψ.

An action $a = (pre_a, eff_a, c_a)$ is called *applicable* in state s, if all constraints in precondition pre_a hold in s and $[\![e]\!]_s$ is defined for all assignments in effect eff_a.

I is called the *initial condition* which is satisfied by exactly one state s, called the *initial state*. I contains exactly one constraint for each variable $v \in V$. G is called the *goal condition*, which is satisfied by *goal states*.

A *serial plan* $\pi = (a_0, ..., a_{n-1})$ is a sequence of actions $a_0, ..., a_{n-1} \in A$ such that there exist (unique) states $s_0, ..., s_n \in S$ such that $s_0 \models I, s_{i-1} \models pre_{a_{i-1}}$ and s_{i-1},$s_i \models eff a_{i-1}$ for each $i = 1$,...,n and $s_n \models G$. We call π a valid plan for Π if it satisfies the initial condition at the beginning of the plan, if all states during the plan are properly defined and the goal condition at the end is satisfied.

With that in mind, we are going to look at an example of a classical planning problem.

Example 2.1.1. *Let us consider a logistics problem involving a truck driving between two cities. The goal of the truck is to pick up a package from one city and deliver it to the other city. We model this instance by a number of variables that define the state of our system. Without considering cost, we define a planning task* $\Pi = \langle V, A, I, G\rangle$ *with*

- $V = \{t@c1,\ t@c2,\ pck@c1,\ pck@c2,\ pck@t\}$
- $A = \{a_{move} = (\{t@c1\}, \{\neg t@c1 \wedge t@c2\}),$
  $a_{load} = (\{pck@c1 \wedge t@c1\}, \{\neg pck@c1 \wedge pck@t\}),$
  $a_{unload} = (\{pck@t \wedge t@c2\}, \{\neg pck@t \wedge pck@c2\})\}$
- $I = pck@c1 \wedge t@c1 \wedge \neg pck@c2 \wedge \neg pck@t \wedge \neg t@c2$
- $G = pck@c2$

Our example consists of five variables defining the location of the truck and the package. For instance, we use pck@c1 to model that the package (pck) is at city1 (c1). Similarly, we use t@c2 to model that the truck (t) is at city2 (c2). The precondition of a_{load} *is that package and truck are at city1. The effect of* a_{load} *enforces that the location of the package changes from city1 to truck. Initially the truck and the package are at city1, therefore these variables are true, and all other variables are false. The goal condition specifies that the package should be at city2. There exists a valid plan* $\pi = (a_{load},\ a_{move},\ a_{unload})$ *which solves this planning task.*

2.2 Planning as Satisfiability

Planning as Satisfiability works by reducing a planning problem to the propositional satisfiability problem (SAT). A first approach was introduced by Kautz und Selman in 1992 [KS92] which was extended in [KMS96]. Sequences of formulas are constructed that encode bounded versions of a planning problem Π. For a given horizon n, which denotes the length of a plan, a formula Π_n is defined whose solutions correspond to plans of length n. Starting with a small n, the horizon is continuously increased in the search for a valid plan. Π_n consists for each step of variables representing the value of each variable and action in Π. Solutions of Π_n contain a list of action variables that are set to true, which is the plan we are looking for.

Given a planning problem $\Pi = \langle V, A, I, G\rangle$, the encoding uses n action variable sets A_0, ..., A_{n-1} where each A_i contains a unique variable for every action $a \in A$ and $n + 1$ state variable sets V_0, ..., V_n where each V_i contains a unique variable for every propositional variable $v \in V$. With these variables we define following formulas:

Let $I(V_0)$ be the formula that specifies the assignment of variables in the initial state with variables $v \in V_0$. Let $G(V_n)$ be the formula that specifies the goal condition with variables $v \in V_n$. Note that every step has an own copy of state and action variables.

Let $T(A_i, V_i, V_{i+1})$ be a formula describing how actions affect states. T enforces that each action in A_i implies its preconditions over V_i and its effects over V_{i+1}. T also encodes frame axioms which enforce that variables keep their value over time unless they are modified by an actions effect. Lastly, T also encodes mutex axioms that specify execution semantics. In our case we allow exactly one action to be executed in each step. We encode bounded plans of length n with following formula:

$$\Pi_n := I(V_0) \wedge \bigwedge_{i=0}^{n-1} T(A_i, V_i, V_{i+1}) \wedge G(V_n)$$

By construction Π_n is satisfiable if and only if there exists a plan π_n with length n. Plans can be extracted by the model of Π_n. In the following example we are going to look at how a specific planning problem is encoded as a SAT formula.

Example 2.2.1. *Assume a simplified version of the planning problem from example 2.1.1 with one propositional variable* $V = \{pck@t\}$*, initial condition* $I = \neg pck@t$*, goal condition* $G = pck@t$*, and two actions* $A = \{a_{load}, a_{unload}\}$ *with* $a_{load} = (\{\neg pck@t\}, \{pck@t\})$ *and* $a_{unload} = (\{pck@t\}, \{\neg pck@t\})$*. Without considering cost, the encoding of* Π_1 *uses following constructs:*

- $A_0 = \{a_{load0},\ a_{unload0}\}$
- $V_0 = \{pck@t_0\},\ \ V_1 = \{pck@t_1\}$
- $I(V_0) = \neg pck@t_0,\ \ G(V_1) = pck@t_1$
- $\begin{aligned} T(A_0, V_0, V_1) = \ & a_{load0} \rightarrow (\neg pck@t_0 \wedge pck@t_1) \wedge \\ & a_{unload0} \rightarrow (pck@t_0 \wedge \neg pck@t_1) \wedge \\ & (\neg pck@t_0 \wedge pck@t_1) \rightarrow a_{load0} \wedge \\ & (pck@t_0 \wedge \neg pck@t_1) \rightarrow a_{unload0} \wedge \\ & a_{load0} \oplus a_{unload0} \end{aligned}$

where $\oplus$ *denotes mutual exclusion. Two state variable sets* V_0, V_1 *and one action variable set* A_0 *are created. The variables get the step as an index. The initial and goal condition depict that in step 0 there is no package at the truck and in step 1 the package is in the truck.* T *enforces first that if action* a_{load} *is executed in step 0 the precondition in step 0 hold and the effects in step 1 hold.* a_{unload} *is encoded similarly. Next the frame axioms state that if the variable* $pck@t$ *changes over a time step, meaning* $pck@t_0$ *and* $pck@t_1$ *have a different value, the responsible action must be the reason. Lastly, the execution semantic states that only one action can be executed in one time step. We notice that* Π_1 *has a solution:* $a_{load0} = pck@t_1 = true$*,* $a_{unload0} = pck@t_0 = false$*, which corresponds to the plan* $\pi_1 = (a_{load})$*.*

2.3 Introduction to PDDL

The Planning Domain Definition Language (PDDL) [McD00] is the community standard for the representation and exchange of planning domain models. It standardizes the formulation of planning problems and facilitates the comparison between different planners. With PDDL a user can define planning problems through domains and problems. A domain defines what type of objects can exist and what actions can be performed. A problem specifies which objects exist, what initial state we have and which goal state we pursue. The task for a planner is then to find an applicable set of actions which start in the initial state and end in the goal state. In the following we will explore in detail the specific parts that make up a planning task in PDDL1.2.

2.3.1 Domain file

The domain file specifies first the name of the domain and the requirements the planner must support to correctly solve the problem. By looking at this list the planner can already decide if it is able to handle this kind of domain. When *:typing* is used the user can define types of objects. PDDL1.2 models objects with predicate properties which are either true or false. The list of predicates is defined under the keyword *:predicates*.

In the following example we look at the logistics domain. It consists of trucks, locations and packages. Predicates are defined that specify properties between these object types and actions are defined which can modify predicates in their effect. In Listing 2.1 we see the whole domain file of the logistics domain.

```
(define (domain logistics)
  (:requirements :typing)
  (:types truck location package)
  (:predicates
    (truck-at ?truck - truck ?loc - location)
    (connected ?loc1 ?loc2 - location)
    (in ?pck - package ?truck - truck)
    (at ?pck - package ?loc - location))

  (:action move-truck
    :parameters (?start ?end - location ?truck - truck)
    :precondition (and (truck-at ?truck ?start)
                       (connected ?start ?end))
    :effect (and (truck-at ?truck ?end)
                 (not (truck-at ?truck ?start))))

  (:action load-one-package
    :parameters (?loc - location ?truck - truck ?pck - package)
    :precondition (and (truck-at ?truck ?loc)
                       (at ?pck ?loc))
    :effect (and (in ?pck ?truck)
                 (not (at ?pck ?loc))))

  (:action unload-one-package
    :parameters (?loc - location ?truck - truck ?pck - package)
    :precondition (and (truck-at ?truck ?loc)
                       (in ?pck ?truck))
    :effect (and (at ?pck ?loc)
                 (not (in ?pck ?truck))))
)
```

Listing 2.1: Domain file of the logistics domain defining requirements, types, predicates and actions.

The predicate `truck-at` takes as input one object of type truck and one object of type location and defines if that truck is at that location. `connected` specifies if two locations are connected and `in` and `at` define if a package is in a truck or at a location.

The actions can be executed if specified input parameters fulfil the precondition formula and change the assignment of predicates in the effect. For example the action move-truck can be executed if the truck is at the start location and the start and end locations are connected. The effect changes then the location of the truck by modifying the truck-at predicate. The action load-one-package checks if the truck and the package are at the same location and then modifies the location of the package. Similarly, the action unload-one-package unloads one package at the location of the truck if it is in the truck.

2.3.2 Problem file

The problem file defines a specific problem instance over a domain. It specifies what name the problem has and what domain it refers to. It then declares the objects used in the problem instance. In our example seen in Listing 2.2 we have one truck, three packages and three locations.

```
(define (problem instance-1)
  (:domain logistics)
  (:objects   truck1 - truck
              city1 city2 city3 - location
              pck1 pck2 pck3 - package)
  (:init
    (connected city1 city2)
    (connected city2 city3)
    (connected city3 city2)
    (connected city2 city1)
    (truck-at truck1 city2)
    (at pck1 city1)
    (at pck2 city2)
    (at pck3 city3))
  (:goal
    (and
      (at pck1 city2)
      (at pck2 city2)
      (at pck3 city2)))
)
```

Listing 2.2: Problem instance for the logistics domain consisting of one truck, three packages and three locations.

The initial state defines what predicates are true at the beginning of the problem. In this example the three cities form a line, the truck is in the middle city and each city has one package. PDDL applies the closed world assumption which is the presumption that what is not currently known to be true is false. Therefore, the rest of the predicates are set to false. The goal state is a formula of predicates which must be satisfied at the end of the plan, for the plan to be considered as a solution. In this problem we want all packages to be at city2.

A valid plan for this problem is for the truck to move to city1 first and pick up the package, move back to city2 and drop of the package and then repeat the same for city3. Note that this is not the only valid plan as the truck could go first to city3 or it could drop the packages at the end of the plan.

Chapter 3

Planning with Sets

Our extension introduces the new data type set into PDDL. It is inspired by Planning Modulo Theories (PMT) from [GLFB12] which treats arbitrary first order theories as parameters in a modelling language. Our set extension allows the user to define sets of objects in the problem file and reason with new semantics like union or intersection of sets in the preconditions and effects of actions. We aim to further improve modelling capabilities. For instance, using a set of packages in a truck instead of having individual predicates may be beneficial in describing loading or unloading operations. In the following we present our approach to modelling sets in PDDL.

3.1 Set Syntax and Semantics

We propose the following extension to the semantics of PDDL1.2. The extension allows to define sets in a new (:sets)-block in the domain file. The construction and assignment of sets is possible in the (:init) and (:goal)-block in the problem file. We present an overview of our syntax and semantics in Figure 3.1.

(construct-set x1 x2)	- returns set constant with objects x1 and x2
(emptyset)	- returns an empty set
(:= (myset s1) (myset s2))	- assigns all objects from myset s2 to myset s1
(add-element (myset s1) x1)	- adds object x1 to myset s1
(rem-element (myset s1) x1)	- removes object x1 from myset s1
(union (myset s1) (myset s2))	- returns union of myset s1 and myset s2
(intersect (myset s1) (myset s2))	- returns intersection of myset s1 and myset s2
(difference (myset s1) (myset s2))	- returns difference of myset s1 and myset s2
(member x1 (myset s1))	- returns true iff x1 is a member of myset s1
(subset (myset s1) (myset s2))	- returns true iff myset s1 is subset of myset s2

Figure 3.1: Syntax and semantics of set extension. Let $x1$, $x2$, $s1$, $s2$ be objects in the problem and *myset* a set variable for $s1, s2$ defined in the (:sets)-block.

Actions can reason with the new set operations in precondition and effect formulas. The goal condition can also use set operations to express goals. As an example we modified the logistics domain from the last chapter to reason with our new extension seen in Listing 3.1. All predicates got replaced with sets.

```
(define (domain set-logistics)
 (:requirements :sets :typing)
 (:types truck location package)
 (:sets
   (in ?truck - truck)
   (at ?loc - location)
   (connections ?loc - location)
 )
 (:action move-truck
   :parameters (?start ?end - location ?truck - truck)
   :precondition (and (member ?truck (at ?start))
                 (member ?end (connections ?start)))
   :effect (and (add-element (at ?end) ?truck)
                (rem-element (at ?start) ?truck)))

 (:action load-one-package
   :parameters (?loc - location ?truck - truck
                ?pck - package)
   :precondition (and (member ?truck (at ?loc))
                 (member ?pck (at ?loc)))
   :effect (and (add-element (in ?truck) ?pck)
                (rem-element (at ?loc) ?pck)))

 (:action unload-one-package
   :parameters (?loc - location ?truck - truck
                ?pck - package)
   :precondition (and (member ?truck (at ?loc))
                      (member ?pck (in ?truck)))
   :effect (and (add-element (at ?loc) ?pck)
           (rem-element (in ?truck) ?pck)))
)
```

Listing 3.1: The set-logistics domain. Predicates are replaced with sets. New operations like member, add-element and rem-element are used.

The `truck-at` and the `pck-at` predicate that specify the location of a truck or a package respectively are replaced with a set of objects for every location. When reasoning with the location of a truck or a package the member operation in combination with a location can get used. Changing the location of a truck or a package can be done with the add-element and rem-element operation. The `in-truck` predicate that holds information if a package is in a certain truck is replaced with a set of objects for every truck. Similarly to the set of a location the set of a truck holds a number of objects that can be modified with the same operations. Lastly, the `connections` predicate can also be replaced by a set of objects for every location.

Note that the sets we define are not typed, meaning that they can contain objects of any type. This is not a problem in a sense that any objects might end up in a set, as this is controlled by the user in the effects of actions, but rather a challenge for the encoding as the number of objects in a problem tends to be large. A problem file using our set extension can be seen in Listing 3.2.

```
(define (problem instance-1)
  (:domain set-logistics)
  (:objects truck1 - truck
            city1 city2 city3 - location
            pck1 pck2 pck3 - package)
  (:init
    (:= (connections city1) (construct-set city2))
    (:= (connections city2) (construct-set city1 city3))
    (:= (connections city3) (construct-set city2))
    (:= (at city1) (construct-set pck1))
    (:= (at city2) (construct-set pck2 truck1))
    (:= (at city3) (construct-set pck3))
    (:= (in truck1) emtpyset)
  )
  (:goal
    (and
      (= (at city2) (construct-set pck1 pck2 pck3 truck1))))
)
```

Listing 3.2: Problem instance-1 for set-logistics domain. Sets get created and assigned in the (:init)-block. The goal formula reasons about the equality of two sets.

Similar to the chapter before this problem instance describes a line of 3 cities which each have one single package and the truck needs to move to every city and bring the package to the starting point.

The creation of sets is done with `construct-set` and a list of objects. An empty set gets created with the keyword `emptyset`. The goal condition holds iff the set accessed by `(at city2)` equals the set consisting of objects `pck1, pck2, pck3, truck1`.

We now show how planning problems with our extended semantics can be encoded into SMT. We present a propositional and a bit vector encoding.

3.2 Propositional Encoding

The first SMT encoding we implemented is the propositional encoding. In the encoding, every set contains one variable for every object in the problem. Each of the variables in a set indicate for a single object in the problem instance if it is contained in this set. This essentially adds a constant number of new variables to the problem for every set we create. With these variables we can create formulas for Planning as Satisfiability as introduced in Chapter 2 Section 2.2, that encode the set operations introduced in the last section.

We introduce dedicated variable names for objects in a set. These are determined by the set name and the object name having the set reference. Additionally, we need to add an index i that specifies the time step of the variable. For example if we have objects $x1$, $x2$, $s1$ in our problem and the set *myset* exists for $s1$, then we have propositional variables $myset\text{-}s1_i^{x1}$, $myset\text{-}s1_i^{x2}$, $myset\text{-}s1_i^{s1}$ with $0 \leq i \leq n$ with n being the horizon, which indicate with their value if objects $x1$, $x2$, $s1$ are contained in the set $(myset\ s1)$ in step i. With this information about the extra set variables we present propositional encodings of set operations in Figure 3.2.

```
(:= (myset s1) (construct-set x1 x2))
```

$$\bigwedge_{obj \in \{x1,x2\}} myset\text{-}s1_0^{obj} \quad \bigwedge_{obj \in \{x3,s1,s2,s3\}} \neg myset\text{-}s1_0^{obj} \tag{3.1}$$

```
(:= (myset s1) emptyset)
```

$$\bigwedge_{\substack{obj \in \{x1,x2,x3,\\ s1,s2,s3\}}} \neg myset\text{-}s1_0^{obj} \tag{3.2}$$

```
(member x1 (myset s1))
```

$$a_i \rightarrow myset\text{-}s1_i^{x1} \tag{3.3}$$

```
(subset (myset s1) (myset s2))
```

$$\bigwedge_{\substack{obj \in \{x1,x2,x3,\\ s1,s2,s3\}}} a_i \rightarrow (myset\text{-}s1_i^{obj} \rightarrow myset\text{-}s2_i^{obj}) \tag{3.4}$$

```
(add-element (myset s1) x1)
```

$$a_i \rightarrow myset\text{-}s1_{i+1}^{x1} \tag{3.5}$$

```
(rem-element (myset s1) x1)
```

$$a_i \rightarrow \neg myset\text{-}s1_{i+1}^{x1} \tag{3.6}$$

```
(:= (myset s1) (union (myset s2) (myset s3)))
```

$$\bigwedge_{\substack{obj \in \{x1,x2,x3,\\ s1,s2,s3\}}} a_i \rightarrow (myset\text{-}s1_{i+1}^{obj} == (myset\text{-}s2_i^{obj} \vee myset\text{-}s3_i^{obj})) \tag{3.7}$$

```
(:= (myset s1) (intersect (myset s2) (myset s3)))
```

$$\bigwedge_{\substack{obj \in \{x1,x2,x3,\\ s1,s2,s3\}}} a_i \rightarrow (myset\text{-}s1_{i+1}^{obj} == (myset\text{-}s2_i^{obj} \wedge myset\text{-}s3_i^{obj})) \tag{3.8}$$

```
(:= (myset s1) (difference (myset s2) (myset s3)))
```

$$\bigwedge_{\substack{obj \in \{x1,x2,x3,\\ s1,s2,s3\}}} a_i \rightarrow (myset\text{-}s1_{i+1}^{obj} == (myset\text{-}s2_i^{obj} \wedge \neg myset\text{-}s3_i^{obj})) \tag{3.9}$$

Figure 3.2: Propositional Encoding. Let $x1, x2, x3, s1, s2, s3$ be objects in the problem and *myset* a set for objects $s1, s2, s3$. Let a be some action in the domain. We present propositional set encodings for gerneral set usage examples in the initial state 3.1, 3.2, the precondition of action a 3.3, 3.4 and the effect of action a 3.5 - 3.9.

Additionally to the shown encodings we need to add frame axioms for every added propositional set variable. This will increase the size of the formula significantly.

3.3 Bit Vector Encoding

The second encoding we implemented is the bit vector encoding. SMT-LIB already supports the theory of fixed sized bit vectors and we can directly use it to build SMT formulas for the encoding of sets.

Every set gets represented with one bit vector. According to a look up table, a single bit encodes if an object is included in the set. Our bit vector encoding adds for every set we create a single bit vector with size n, with n being the number of objects in the problem instance.

We introduce dedicated variable names for the bit vectors. They consist of the set name and the object having the set. Additionally, we also need an index i that specifies the time step of the bit vector variable. For example if we have objects $x1$, $x2$, $s1$ in our problem and the set *myset* exists for $s1$, then we have the bit vector variable *myset*-$s1_i$ of size 3, with $0 \leq i \leq n$, n being the horizon, which indicates with its value if objects $x1$, $x2$, $s1$ are contained in the set (*myset* $s1$). Set encodings with bit vectors are seen in Figure 3.3. We additionally need frame axioms for every bit vector variable.

```
(:= (myset s1) (construct-set x1 x2))
```

$$myset\text{-}s1_0 == (1,1,0,0,0,0) \tag{3.10}$$

```
(:= (myset s1) emptyset)
```

$$myset\text{-}s1_0 == (0,0,0,0,0,0) \tag{3.11}$$

```
(member x1 (myset s1))
```

$$a_i \rightarrow ((1,0,0,0,0,0) == (myset\text{-}s1_i \,\&\, (1,0,0,0,0,0))) \tag{3.12}$$

```
(subset (myset s1) (myset s2))
```

$$a_i \rightarrow ((0,0,0,0,0,0) == (myset\text{-}s1_i \,\&\, \sim myset\text{-}s2_i)) \tag{3.13}$$

```
(add-element (myset s1) x1)
```

$$a_i \rightarrow (myset\text{-}s1_{i+1} == (myset\text{-}s1_i \mid (1,0,0,0,0,0))) \tag{3.14}$$

```
(rem-element (myset s1) x1)
```

$$a_i \rightarrow (myset\text{-}s1_{i+1} == (myset\text{-}s1_i \,\&\, (0,1,1,1,1,1))) \tag{3.15}$$

```
(:= (myset s1) (union (myset s2) (myset s3)))
```

$$a_i \rightarrow (myset\text{-}s1_{i+1} == (myset\text{-}s2_i \mid myset\text{-}s3_i)) \tag{3.16}$$

```
(:= (myset s1) (intersect (myset s2) (myset s3)))
```

$$a_i \rightarrow (myset\text{-}s1_{i+1} == (myset\text{-}s2_i \,\&\, myset\text{-}s3_i)) \tag{3.17}$$

```
(:= (myset s1) (difference (myset s2) (myset s3)))
```

$$a_i \rightarrow (myset\text{-}s1_{i+1} == (myset\text{-}s2_i \,\&\, \sim myset\text{-}s3_i)) \tag{3.18}$$

Figure 3.3: Bit Vector Encoding. Let $x1, x2, x3, s1, s2, s3$ be objects in the problem and *myset* a set for objects $s1, s2, s3$. Let a be some action in the domain. Let & be a bitwise and operation, | be a bitwise or operation, and $\sim$ be a bitwise negation. We present bit vector encodings for gerneral set usage examples in the initial state 3.10, 3.11, the precondition of action a 3.12, 3.13 and the effect of action a 3.14 - 3.18.

Chapter 4

Experimental Results

To evaluate the proposed set extension we extended OMTPlan [LGÁT21] an SMT based planner. We first extended the parser to accept set syntax. Then we extended the encoder with new set semantics using propositional and bit vector encodings.

We evaluate our implementation on standard benchmarks from the IPC. The domains we use are Elevator, Logistics, Zeno-Travel and Gripper. An illustration of the domains is seen in Figure 4.1.

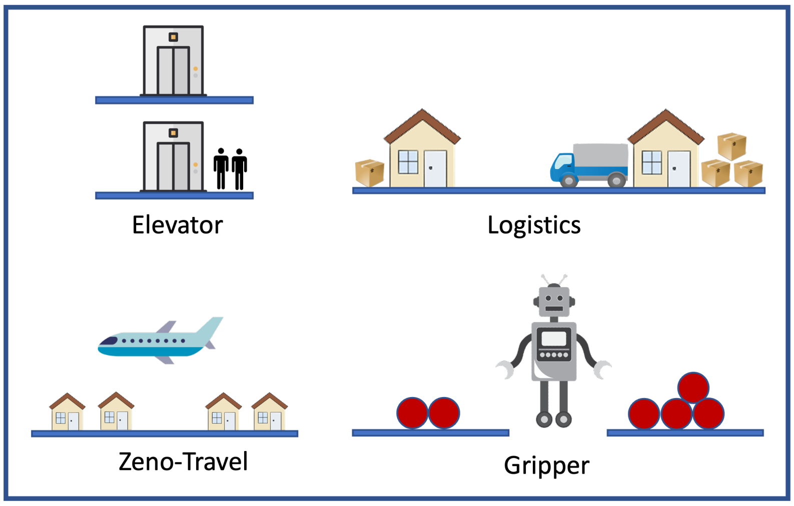

Figure 4.1: Illustration of domains from the IPC for evaluation.

We use the STRIPS domains with typed objects, meaning that objects can be typed and in the effects of actions, add and delete effects are allowed. To compare our extension with standard PDDL we created a set variant of each domain that utilizes our set extension. The set variants of the domains use sets as much as possible.

In the set-elevator domain we model a set of objects in the elevator and at floors. The set-logistics domain uses sets of objects in trucks and at locations. With the set-zeno-travel domain we use sets of objects in planes and at locations. Lastly the set-gripper domain has a set of objects for each room. Furthermore, the actions were modified to reason with set operations at appropriate places. With these changes we replaced predicates with sets while still modelling the same planning problem. Additionally to the domains, every problem instance had to be copied and changed to using our extension, so we have equivalent problems to compare. In the next section we present how our extension eased the modelling task compared to standard PDDL.

4.1 Easing the Modelling Task

Consider the Gripper domain. It consists of two gripper constants left and right, a number of rooms and a number of balls. The gripper can pick up and drop of the balls and move between the rooms. The gripper domain is seen in Listing 4.1.

```
(define (domain gripper)
  (:requirements :typing)
  (:types room ball gripper)
  (:constants left right - gripper)
  (:predicates
    (at-robby ?r - room)
    (at ?b - ball ?r - room)
    (free ?g - gripper)
    (carry ?o - ball ?g - gripper))
  (:action move
    :parameters  (?from ?to - room)
    :precondition (at-robby ?from)
    :effect (and (at-robby ?to)
                 (not (at-robby ?from))))
  (:action pick
    :parameters (?obj - ball ?room - room ?gripper - gripper)
    :precondition (and (at ?obj ?room)
                       (at-robby ?room)
                       (free ?gripper))
    :effect (and (carry ?obj ?gripper)
                 (not (at ?obj ?room))
                 (not (free ?gripper))))
  (:action drop
    :parameters (?obj - ball ?room - room ?gripper - gripper)
    :precondition (and (carry ?obj ?gripper)
                       (at-robby ?room))
    :effect (and (at ?obj ?room)
                 (free ?gripper)
                 (not (carry ?obj ?gripper))))
)
```

Listing 4.1: Domain file for gripper domain.

We modified this domain and replaced the `at` predicate with a set `in` for every room that holds the information what objects are in the room. The actions pick and drop must reason with member, add-element, rem-element operations together with a set and a specific object. The rest of the domain remains unchanged hence we do not include a Listing.

With this small change we dramatically changed the appearance of a problem file. When comparing two similar problem instances, one modelled with the set extension and one with standard PDDL it results in a cleaner look in favour of our set extension.

Listing 4.2 shows the problem instance modelled with standard PDDL and Listing 4.3 shows it modelled with the set extension.

```
(define (problem instance-1)
  (:domain gripper)
  (:objects rooma roomb - room
            b1 b2 b3 b4 b5 b6 b7 b8 - ball)
  (:init
    (at-robby rooma)
    (free left)
    (free right)
    (at b1 rooma)
    (at b2 rooma)
    (at b3 rooma)
    (at b4 rooma)
    (at b5 rooma)
    (at b6 rooma)
    (at b7 rooma)
    (at b8 rooma)))
  (:goal (and
    (at b1 roomb)
    (at b2 roomb)
    (at b3 roomb)
    (at b4 roomb)
    (at b5 roomb)
    (at b6 roomb)
    (at b7 roomb)
    (at b8 roomb)))
)
```

Listing 4.2: Problem file modelled with standard PDDL for gripper domain.

```
(define (problem instance-1)
  (:domain set-gripper)
  (:objects rooma roomb - room
            b1 b2 b3 b4 b5 b6 b7 b8 - ball)
  (:init
    (at-robby rooma)
    (free left)
    (free right)
    (:= (in rooma) (construct-set b1 b2 b3 b4 b5 b6 b7 b8))
    (:= (in roomb) emptyset))
  (:goal (and
    (= (in roomb) (construct-set b1 b2 b3 b4 b5 b6 b7 b8))))
)
```

Listing 4.3: Problem file modelled with set extension for set-gripper domain.

Many predicate definitions in the initialization get combined in a single set definition. This reduction of redundancy benefits the size of the file and a reader can get an overview of the problem quicker.

We created for each domain 5-15 problem instances of varying difficulty. First we compare the propositional encoding with the bit vector encoding on the Elevator domain. Then we compare the normal PDDL approach with our set extension using bit vector encodings.

4.2 Propositional versus Bit Vector Encoding

Results of comparing the propositional with the bit vector encoding are seen in Figure 4.2. We notice that the bit vector encoding performs better than the propositional encoding in the instances tested.

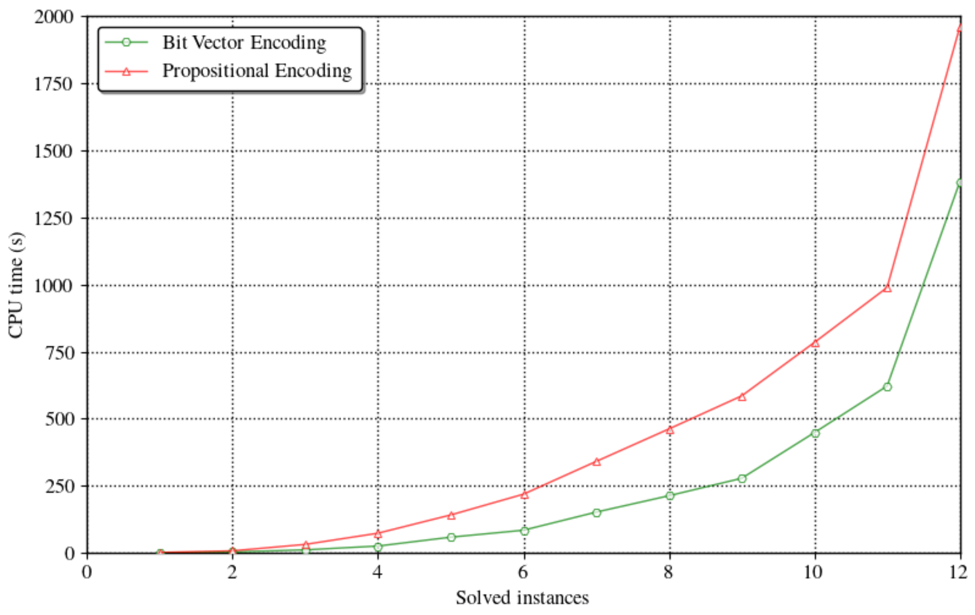

Figure 4.2: Cactus plot for Elevator domain comparing bit vector and propositional encodings of sets. Instances are ordered by increasing CPU time (in seconds).

Every instance of the elevator domain got solved faster with using bit vector encodings. This is due to extra variables that are created for the propositional encoding. Every variable needs its own frame which results into many assertions. And the more assertions we have, the longer the solver takes to find a solution. The bit vector encoding uses fewer variables and therefore creates a smaller formula which makes it the better choice for encoding sets. In the next section we evaluate how our set extension compares against a normal PDDL approach using predicates.

4.3 Set Extension versus Standard PDDL

Results of comparing the set extension using bit vector encodings to standard PDDL using the original domains without sets from the IPC, are seen in Figure 4.3.

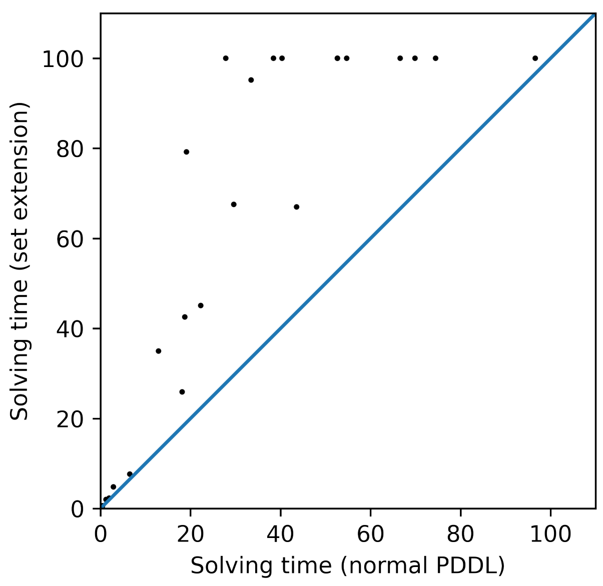

Figure 4.3: Comparison of solving times (sec) for all four domains. Set extension using bit vector encoding versus normal PDDL using predicates (timeout = 100sec).

We notice that the solving time for instances using sets are always higher than instances using normal PDDL. This may be fault due to extra overhead we create in the encoding. The approach we used to evaluate our extension was to take existing benchmarks and modify them to work with sets. Because the benchmarks were not specifically designed for sets they did not take advantage of all the set operations. We observe a linear relationship between the solving times of instances using our set extension and normal PDDL. This is an invitation for more exploration of sets in PDDL for the future.

Chapter 5

Conclusion

5.1 Summary

In this thesis we considered the problem of planning with sets. We proposed an extension of PDDL that allows reasoning with set theory. We discussed the approach of Planning as Satisfiability and presented two SMT encodings for our set operations. We developed an extension for OMTPlan an SMT-based planner, that implements our set extension in a non-optimal setting. We presented examples throughout the thesis where it is desirable to use sets from a modelling perspective.

In our propositional encoding of a set, every set contains a variable for every object. These variables indicate with true or false, if objects are contained in the set. Advantage of the approach is that no other theory except equality needs to be used in the encodings. This allows for a simple to grasp concept. The disadvantage is that with many extra variables many frame axioms are needed which does not scale well.

With the bit vector encoding of a set, every set gets represented by one bit vector. A single bit encodes if an object is included in the set. SMT-LIB supports the theory of fixed sized bit vectors which made it possible to directly encode them into a SMT formula. Advantage of bit vectors over a propositional encoding is that union or intersection operations can benefit directly from bit wise operations. Furthermore, there are fewer frame axioms needed.

Experimental results of our implementation over benchmarks from the IPC showed that our set extension does ease the modelling task of planning problems. The higher expressivity allows for more compact problem definitions. Experiments showed that bit vector encodings run faster than propositional encodings. When comparing problems modelled with sets using bit vector encodings to problems modelled with standard PDDL it shows that our implementation does not scale on larger problem instances. This may be faulted to extra overhead we introduce with bit vector variables and that the benchmarks were not specifically designed for set usage.

5.2 Future Work

First our implementation of sets should be optimized to reduce overhead it creates. Additionally, it is desirable to extend our proposed encodings to support PDDL2.1 featuring numeric variables and temporal constraints. With more sophisticated domains and problems, many new applications of sets can be explored. Furthermore, the search of plans in an optimal setting using the set extension should be considered.

Bibliography

[FL03] Maria Fox and Derek Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

[GLFB12] Peter Gregory, Derek Long, Maria Fox, and J Christopher Beck. Planning modulo theories: Extending the planning paradigm. In *Twenty-Second International Conference on Automated Planning and Scheduling*, 2012.

[KMS96] Henry Kautz, David McAllester, and Bart Selman. Encoding plans in propositional logic. *KR*, 96:374–384, 1996.

[KS92] Henry A Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, volume 92, pages 359–363. Citeseer, 1992.

[LGÁT21] Francesco Leofante, Enrico Giunchiglia, Erika Ábrahám, and Armando Tacchella. Optimal planning modulo theories. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 4128–4134, 2021.

[McD00] Drew M McDermott. The 1998 ai planning systems competition. *AI magazine*, 21(2):35–35, 2000.