PDDL4J: a planning domain description library for java.

Damien Pellier, Humbert Fiorino

To appear in the *Journal of Experimental & Theoretical Artificial Intelligence*
Vol. 00, No. 00, Month 20XX, 1–39

PDDL4J: A Planning Domain Description Library for Java

D. Pellier* and H. Fiorino

Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, F-38000 Grenoble France

(*v3.0 released July 2017*)

PDDL4J (Planning Domain Description Library for Java) is an open source toolkit for Java cross-platform developers meant (1) to provide state-of-the-art planners based on the PDDL language, and (2) to facilitate research works on new planners. In this article, we present an overview of the Automated Planning concepts and languages. We present some planning systems and their most significant applications. Then, we detail the PDDL4J toolkit with an emphasis on the available informative structures, heuristics and search algorithms.

Keywords: Automated planning, Heuristic Search, Planning Software, Java Programming.

Introduction

PDDL4J (Planning Domain Description Library for Java[1]) is an open source toolkit for Java cross-platform developers meant (1) to provide state-of-the-art planning tools based on the PDDL language, and (2) to facilitate research works on new planners. PDDL4J has been successfully used in several domains, e.g., automatic web services composition (Bevilacqua, Furno, Scotto di Carlo, & Zimeo, 2011), verification of business processes (Salnitri, Paja, & Giorgini, 2015), game (Cheng, Knoll, Luttenberger, & Buckl, 2011) and robotics (Zaman, Steinbauer, Maurer, Lepej, & Uran, 2013).

The PDDL language was originally developed by Malik Ghallab and the 1998 International Planning Competition (IPC) committee (Ghallab et al., 1998). It was inspired by the need to encourage the empirical comparison between planning systems and the diffusion of benchmarks within the planning community. Consequently, PDDL has improved planning system evaluation and led to important advances in performances and expressivity.

Since these pioneering works, PDDL has become a *de facto* standard language for the description of planning domains for IPC (its growing collection of domains is generally adopted as standard benchmarks) and for the applications embedding planners. The emergence of PDDL as a standard language has had a significant impact on the entire Artificial Intelligence field. It has facilitated the dissemination of planning techniques in different research and application fields where modelling and solving decision problems are challenging issues.

This paper is organized as follows: in Section 1, we present some of the most representative fields of application for planners. Section 2 describes how to get started with the PDDL4J toolkit. Section 3.1 introduces the background concepts of the automated

*Corresponding author. Email: damien.pellier@imag.fr

[1]http://pddl4j.imag.fr

planning as well as the world representations used by the planners. In Section 4, we present the PDDL language, and in Section 5, we detail the PDDL4J architecture and its different modules. Lastly, in Section 6, we compare PDDL4J with other exisiting planning toolkits.

1. Applications for Automated Planning

AI planners have been successfully used in a wide range of applications. It would be too long to review all of theses applications. In this section, we want to highlight the importance of the research works on automated planning, and consequently, of developing planning toolkits. We overview some of the most representative application fields showing where planning systems have a significant impact. Among these fields, we discuss the contributions of automated planning in developing the autonomy of robotic systems (see Section 1.1) and in facilitating IT knowledge engineering and model design (see Section 1.2).

1.1. *Autonomous systems*

1.1.1. Robotics

Automated planning systems have come a long way in robotics since the Shakey robot and the STRIPS planner (Fikes & Nilsson, 1971). Robotics is one of the most appealing application areas for automated planning, and, for several years, the PlanRob workshop at ICAPS (International Conference on Automated Planning and Scheduling) has become an important forum specifically dedicated to the challenges related to planning for autonomous robots. Among the challenges is the integration of planners in robot architectures, which is difficult and has been intensively investigated. Indeed, planning is an open loop activity that produces plans based on action models, the current state of the world and the targeted goal. On the other hand, executing actions is a closed loop activity that perceives the current state of the world, events modifying this state and feedbacks from the performed actions. Thus, planning in robotics needs an integrated approach to handle time, concurrency, synchronizations, deadlines and resources (Di Rocco, Pecora, & Saffiotti, 2013). For instance, FAPE (Dvořák, Bit-Monnot, Ingrand, & Ghallab, 2014) (Flexible Acting and Planning Environment) is a system integrating a planning language with expressive timeline representation, the planning process interacting with a dispatching mechanism that synchronizes observed time points of action effects and events with planned time. Another important issue is combining tasks and motion planning. In robot architectures, generally a "discrete" symbolic module reasons about causal and/or temporal relationships between actions, while "continuous" geometric/kinematic reasoning is entirely delegated to a specific module. For instance, picking up an object may require removing many others that obstruct this object. Identifying the exact obstructions requires geometric reasoning with results that are difficult to represent efficiently at the level of a symbolic planner. Different propositions based on new representation techniques have been made to synchronize continuous and discrete planning levels (Lagriffoul, 2014; Srivastava, Riano, Russell, & Abbeel, 2013; Garrett, Lozano-Pérez, & Kaelbling, n.d.; Ferrer-Mestres, Francès, & Geffner, 2015; Fernandez-Gonzalez, Karpas, & Williams, 2015). It is also possible to couple a symbolic planner with an ontology in order to reason about the action possibilities that are available to the planner as the set of known objects is updated (Cashmore et al., 2015). Furthermore, robots have

to interact with persons in an appropriate way and to act socially. HATP (Lallement, de Silva, & Alami, n.d.) is a planning framework that extends the planning representation by making Human-Robot interactions "first class" entities in the language. "Social rules" can be defined specifying which behaviours are acceptable. HATP interleaves symbolic planning with geometric reasoning to validate the human/robot interactions.

1.1.2. Space exploration

A typical challenge for space mission management is selecting observation targets for instruments and regularly building mission activities over a limited temporal horizon and resource constraints. Space mission management tries to optimally satisfy ground user requests. Data volume, observation timing, visibility windows, power needs, lighting, seasons, scientific objectives etc. have acute effects on real-time operations. Satellite, rover and spacecraft missions have management systems addressing these challenges, and automated planning and scheduling are technologies used in this field.

The operations team controlling Mars Exploration Rovers (MER) had to generate each day new plans describing the rover activities (Backes, Norris, Powell, & Vona, 2004; Bresina, Jónsson, Morris, & Rajan, 2005). Their objective was to fulfill as much observation requests as possible given the rover's resource limitations, while minimizing risk of vehicle failure. In order to accomplish this objective, given the short amount of planning time available, the operations team employed a Mixed-initiative Activity Plan Generator (MAGEN) system based on automated planning, scheduling and temporal reasoning, to assist operation staff in generating the daily activity plans (command sequences generated on the ground, and then sent to the spacecraft, which had to execute them). MAPGEN has been used for two MER rovers (Spirit and Opportunity), for the Phoenix Mars lander and for Mars Science Lander. Like any modern planner, it can evaluate thousands of alternative scenarios. However its goal is not to replace human experts, who in the context of critical operations may not trust computer decisions, but to assist them. The role of mixed-initiative planning is to "explain reasons for the decisions it makes, allowing the user to interactively explore alternatives, guide the search toward more desirable solutions, and to run various queries (e.g., what courses of action have not yet been explored with respect to some goal?)".

1.2. *Model design*

1.2.1. Video games and virtual agents

FEAR (First Encounter Assault Recon) is a survival horror first-person shooter video game developed and released in 2005. It uses AI techniques and Finite State Machines to control the characters' behaviors, and an A^* planner to generate sequences of actions. The objective is both to alleviate game developer burden, and to provide a realistic action movie experience to players, with intense combats and believable character behaviors (for instance, characters are able to use the environment to their advantage, taking cover, and even flipping over furniture to create their own cover rather than popping randomly in and out of cover, like a shooting gallery). FEAR characters use cover tactically, coordinating with others characters in a squad, leave a hiding place when threatened, and fire if they have no better option. The rapidly growing complexity of the combination and interaction of all of the behaviors becomes unmanageable for AI game developers, and planning tackles this issue by focusing developer effort on goal and action design, and letting the planner decide how to sequence actions to satisfy goals. Behavior designers do

not need to manually specify which actions are associated with which goals, which actions can transition to which other actions, or which goals can transition to other goals. The benefits of planning are numerous. Decoupling goals and actions allow different types of characters to satisfy goals in different ways. The second benefit of the planning approach is facilitating layering of behaviors and the incremental design of new characters from existing ones by adding/removing actions and goals (for instance, adding the "cover" goal, which makes a character who is not currently under cover get to cover etc.) Finally, an important benefit of a planning system is the capacity to dynamically solve problems: characters can re-plan, chznge their decisions etc. Characters are sensitive to the changes induced by the player in their environment: if the goal of a character is to open a door and this door is locked, then the character activates a new goal – enforce room access, and generates a new plan to satisfy its goal – kick in the door or get in through the window etc. Automated planning is used as an abstraction layer for model design. It allows non-technical developers to program AI systems, to design "interactive narratives" such as in AI based virtual agent games, intelligent tutoring systems, embodied conversational agents, etc. (Fernández, Adarve, Pérez, Rybarczyk, & Borrajo, 2006; Thomas & Young, 2006).

1.2.2. Business Process Management

In many companies, information systems have to undertake activities such as accessing remote data, integrating heterogeneous data, analyzing and deriving new data, etc. These activities are implemented as business processes controlling and interacting through different information flows. Usually, business processes are *manually* specified in languages such as Bpel in order to produce the adequate workflow in terms of data manipulations. Automated planning techniques have been used to *automatically* compose process skeletons (Hoffmann, Weber, & Kraft, 2009). Considered as a promising technique for Web Service Composition, several research works in non-deterministic planning have investigated how to extract planning models from existing industrial solutions (Hoffmann et al., 2009; Dongning, Zhihua, Yunfei, & Kangheng, 2010) and automatically connect the outputs of a process with the inputs of another (Ambite & Kapoor, 2007).

2. Getting Started with Pddl4j

Getting started with Pddl4j is simple. The command lines above (see Listing 1) show how to proceed.

```
1 $ git init
2 $ git clone https://github.com/pellierd/pddl4j.git
3 $ cd pddl4j
4 $ ./gradlew clean build
5 $ ./gradlew javadoc
6 $ ./gradlew run -PArgs=-o,<DOMAIN FILE PATH>,-f,<PROBLEM FILE PATH>
```

Listing 1 Pddl4j Installation, Compilation and Test

The `GIT` version control system allows to easily install Pddl4j (line 1 and 2). Pddl4j comes with all the configuration files for the Gradle build automation system (line 4 and 5): these command lines build Pddl4j on the host platform, and generate the documentation. Pddl files are available in `/pddl` directory to test `HSP` (Heuristic Seach Planner), our default planner (line 6). The plans generated by Pddl4j are compliant with Pddl3.1 and can be validated with Val, a plan validator (Howey, Long, & Fox, 2004).

Listing 2 shows how to directly run the `HSP` planner with the Java Virtual Machine. Note that the build generates a jar file in `/build/libs`.

```
$ java -javaagent:build/libs/pddl4j-VERSION.jar -server -Xms2048m -Xmx2048m fr.uga
    .pddl4j.planners.hsp.HSP -o <DOMAIN FILE PATH> -f <PROBLEM FILE PATH>
```

Listing 2 Java Run Command

Listing 3 shows how to use the `HSP` planner in a third-party code.

```
1  public static void main(String[] args) {
2
3        // Get the domain and the problem from the command line
4        String domain = args[0];
5        String problem = args[1];
6
7        // Create the problem factory
8        final ProblemFactory factory = new ProblemFactory();
9
10       // Parse the domain and the problem
11       ErrorManager errorManager = factory.parse(domain, problem);
12       if (!errorManager.isEmpty()) {
13             errorManager.printAll();
14             System.exit(0);
15       }
16
17       // Encode and simplify the planning problem in a compact representation
18       final CodedProblem pb = factory.encode();
19       if (!pb.isSolvable()) {
20             System.out.println("goal can be simplified to FALSE. "
21                         + "no search will solve it"));
22             System.exit(0);
23       }
24
25       // Create the planner and choose the Fast Forward heuristic
26       HSP planner = new HSP();
27       planner.setHeuristicType(Heuristic.Type.FAST_FORWARD);
28
29       // Search for a solution plan
30       final Plan plan = planner.search(pb);
31       if (plan != null) {
32              System.out.println(String.format("%nfound plan as follows:%n%n"));
33              System.out.println(pb.toString(plan));
34       } else {
35              strb.append(String.format("%nno plan found%n%n"));
36       }
37 }
```

Listing 3 Simple example of PDDL4J usage

First, we get the path of the domain and the path of the problem from the command line (lines 4 and 5). Then a planning problem "factory" is created (line 8). The factory's parser (line 11) takes both paths as input. Then, the planning problem is encoded by the factory (line 18). The method "`isSovable()`" (line 19) does a time-polynomial test asserting whether a solution is possible after the encoding step. Then, a planner "object" is created (line 26). In our case, we create an instance of the simple planner `HSP` (see Section 5.6 for more details about the planners available in PDDL4J). Then, a heuristic for the search is chosen (line 27). In our example, the Fast Forward heuristic is chosen (see section 5.5 for more details about the heuristics available in PDDL4J). Finally, a solution, namely a "plan" in Automated Planning terminology, is searched (line 30) in exponential time (Automated Planning is NP-hard) and the solution plan is displayed.

3. PDDL4J Background

3.1. *Planning Concepts*

Planners are general purpose decision problem solvers. Given a decision problem expressed in a logical high level language such as PDDL (see section 4), a planning system, namely a planner, builds a solution to this problem. The search algorithm embedded in planners is not specific to the problem to solve, it is a general purpose domain-independent search algorithm used whatever the input problem. Planning efficiency is based on the planner's ability to extract information from problem structures and promising paths to solutions, and consequently to use informed search heuristics.

Unlike domain-dependent approaches that build upon algorithms specifically devised for each type of problem, planners are based on a declarative (non-procedural and very high-level) language specifying what needs to be done rather than how to do it. The problem representation describes the *initial state* of the world – *objects* under consideration and their *properties*, a *goal* to achieve and *actions* that can be performed on these objects to achieve the goal. Actions have triggering *preconditions* and resulting *postconditions*, which are the actions' effects. The preconditions define under which circumstances (states of the world) actions can be performed. When triggered, actions change the state of the world as specified by their effects, and generate new states. As actions are not necessarily reversible (past states are no longer achievable from the present state), actions have precedence constraints and are not ordered in any sequence. A solution to a planning problem is a set of ordered actions, called a *plan*, starting from the initial state and ending in a goal state. For example, in graph theory, graph coloring is an assignment of colors to vertices (objects) such that no two connected vertices share the same color (constraint). A planning problem for graph coloring defines: a set of colours and vertices, and their connections and associated colours (the initial state); three actions that are solely applicable to two connected vertices (preconditions), and whose effect is always to disconnect them (i.e. color two unpainted vertices with two different colors; color one unpainted vertex with a color different from the another connected vertex, and simply disconnect two painted vertices). It is obvious that these actions are not reversible because painted vertices cannot be repainted. However, any valid combination (with respect to preconditions and precedence constraints) of actions satisfies the problem specification. Then, the goal here is to disconnect all of the vertices, and a plan is a sequence of colouring actions that reaches this goal. Another example of a planning problem is logistics (see section 4.1) where the objects are parcels, trucks, airplanes, airports and locations: the parcels can be loaded and unloaded (actions) from trucks or airplanes. Two other actions are driving trucks from one location to another one, and flying airplanes between airports. The initial state indicates the location of the parcels, trucks and airplanes (properties of the objects), and the goal specifies the parcels' destinations. A plan is a sequence of actions of loading/unloading parcels, driving trucks and flying airplanes between different locations in such a way that, in the final state reached by the last action, the goal is satisfied.

Therefore planning is the process consisting in choosing appropriate actions to bring the state of the world to the targeted goal. Classically, the dynamics of the world is modeled as a *State Transition System*.

Definition 3.1: A *State Transition System* is a tuple $\Sigma = (S, A, c, \gamma)$ such that:

- S is a set of states,
- A is a set of actions,

- $c : A \to \mathbb{R}^+$ is a cost function,
- $\gamma : S \times A \to S$ is a state transition function.

A State Transition System can be represented as a directed graph whose nodes are states of S, and vertices are actions of A. Given an action a and a state s, if $\gamma(s, a)$ is defined, then action a is *applicable* in the state s. Applying a to s produces a new state $s' = \gamma(s, a)$ with a cost $c(a)$. Σ is deterministic if for all states s and actions a, the transition function $\gamma(a, s)$ produces a unique state s'. Σ has a unit cost if, for all $a \in A$, $c(a) = 1$. A plan is any sequence of actions $\pi = [a_1, \cdots, a_k]$ (the size of π is k). The state produced by applying π to a state s is the state obtained by applying each action of π sequentially. We denote this by extending the state transition function to plans as follows:

$$\gamma(s, \pi) = \begin{cases} s, & \text{if } k = 0 \\ \gamma(\gamma(s, a_1), [a_2, \cdots, a_k]), & \text{if } k > 0 \text{ and } a_1 \text{ is applicable to } s \\ \text{undefined}, & \text{otherwise} \end{cases}$$

A state s' is reachable from a state s, if there exists a plan π such that $s' = \gamma(s, \pi)$.

3.2. *Classical Planning Representations*

It exists different, but equivalent ways to represent a classical planning problem as a State Transition System. Whatever a representation is adopted, the planning community mainly uses the PDDL language to specify planning problems. However, PDDL is not the only language used in the planning community: see for instance OCL (Object Constraints Language) (Liu & McCluskey, 2001) or OPL (Other Planning Language) (McDermott, n.d.). Most of the planners pre-process the PDDL language to encode planning problems in their internal representation. PDDL4J is able to manage two representations that are widely used in the planning community: the logical representation and the Finite Domain Representation (FDR).

3.2.1. Logical Representation

In the logical representation, each state s of the world is represented by a set of logical propositions p. If p is in s then p is *true* in the state s, and *false* otherwise. Each logical proposition is associated to a fact of the world (the presence of an airplane in an airport, the color of a node etc., see section 3.1). An action is an expression defining which propositions have to be true in a state in order to apply this action, and which propositions have to be added and removed from the state of world after the application of the action. In this representation, actions change the states of the world by modifying the truth values of the propositions. For simplicity and conciseness, in planning representations, states and actions are usually given in a first order logic language (without function symbols) rather than propositional logic. For instance, the literal "x is at l" (x and l are variable symbols) is a substitute for "x_0 is at t_1" or "x_1 is at f_1" (where x_0, x_1, t_1, f_1 are constant symbols denoting, for example, trucks and places in the logistics planning domain) etc. Similarly, the first order (abstract) action "drive truck x from place f to place t" is a substitute for ground actions like "drive truck x_0 from place f_1 to place t_1" or "drive truck x_1 from place t_1 to place f_1" etc. Here the quoted expressions denote terms of the PDDL language that will be presented in section 4. For the moment, it suffices to say that in planning terminology, first order actions are *operators*:

Definition 3.2: An operator o is a tuple $o = (\text{name}(o), \text{precond}(o), \text{effect}(o))$ whose elements are:

- name(o), the *name* of the operator,
- precond(o), the *preconditions* of the operator, i.e. a set of literals that must be true to apply the operator o,
- effect(o)$^-$ is a set of literals that are false after the application of the operator o,
- effect(o)$^+$ is a set of literals that are true after the application of the operator o.

Definition 3.3: An action is any ground instance of an operator. If a is an action and s is a state such that precond(a) are true in s, then a is *applicable* to s and the result of applying action a to state s is the state s':

$$s' = \gamma(s, a) = (s - \text{effects}^-(a)) \cup \text{effects}^+(a)$$

To transform the operators of the PDDL language (in the domain file) into the actions used by the planning algorithm, an instantiation process is triggered. This process is time consuming, and several speed-ups have been proposed in the literature. The instantiation process of PDDL4J is described in Section 5.2.

PDDL (problem and domain) files do not provide a procedure on how to solve a planning problem as in an imperative language (the "procedural" approach). They express a theory of action and its underpinning grammar of decisions asserting what satisfying combinations of actions can be tried to fulfill the targeted goal. This is a "declarative" approach on which the planners are based.

Definition 3.4: A planning problem in the logical representation is a quadruplet (P, A, c, I, G) where:

- P is a finite set of propositions,
- A is a finite set of actions, for each $a \in A$ precond(a) and effect(a)) are subsets of P and effects$^-(a)) \cap$ effects$^+(a) = \emptyset$,
- $c : A \rightarrow \mathbb{R}^+$ is a cost function,
- $I \subseteq P$ is the initial state,
- $G \subseteq P$ is the goal.

The *state space* of a planning problem (P, A, c, I, G) in the logical representation is a transition system (S, A, c, γ) where $S \subseteq 2^P$ is a subset of states defined on P. A *solution* to a planning problem (P, A, c, I, G) is a plan π such that $G \subseteq \gamma(I, \pi)$.

Once this specification is provided to the planner, the planning process uses a mean-ends reasoning to decide how to achieve goals given the available actions (see section 5.6 for planning algorithms). The computational hardness of the mean-ends planning process is high (NP-hard) because it entails non-deterministic decisions selecting objects, acting on objects, sorting out actions, as well as the backtracting from dead end decisions in combinatorial search spaces. For instance, in the graph coloring problem (see Section 3.1), the search space is the set of all the valid combinations of coloring actions: the bigger the graph, the higher the number of coloring actions, and the search space size is exponential in the number of actions. The `logistics` problem (see Section 3.1) emphasizes another determinant feature of planning problems: precedence constraints. Actions are not necessarily reversible and have to be performed in an appropriate sequence in order to achieve the planning goal: for example, packets have to be loaded before unloading etc. Furthermore, actions can have negative effects and the progression towards

the goal is not monotone. For instance, suppose that in a logistics problem, the goal is to have the truck x_0 at place f_1 and parcel p_0 at place t_1. If in the initial state, the truck x_0 is at place f_1 and p_0 is loaded in x_0, driving that truck x_0 from f_1 to t_1 to deliver p_0 will delete the fact "truck x_0 is at place f_1". This has to be added again (by driving back the truck x_0 from t_1 to f_1) to fulfill the goal.

The consequence of the planning computational complexity is that planners do not rely on brute-force search. They use informed search algorithms and extract heuristic information from planning problem representations (see Section 5.5).

3.2.2. Finite Domain Representation

Finite Domain Representation (FDR) is slightly different from logical representation. Each state is represented by a set of state variables and each action by a function that maps sets of state variables. This representation has been very popular since 2006 generally combined with the SAS+ formalism (Bäckström & Nebel, 1995). New efficient heuristics based on this representation have been developed. The advantages of the FDR over the logical representation are discussed in Section 4.2. However, the FDR is more compact (on the condition that the chosen variables do not have binary domains). For instance, the state variable v denoting the "location of truck x_0" is assigned with a value representing the place of the truck: $v = f_1$ if "x_0 is at f_1" etc. The preprocessing that chooses the state variables and their respective domains to obtain compact representations is implemented in PDDL4J and detailed in Section 5.3.

Definition 3.5: Let V be a finite set of variables, where each variable v has an associated finite domain $\mathrm{Dom}(v)$ that contains the default value *nil*. An operator in the FDR is a triple $o = (\mathrm{name}(o), \mathrm{precond}(o), \mathrm{effect}(o))$, where $\mathrm{name}(o)$, $\mathrm{precond}(o)$ and $\mathrm{effect}(o)$ are respectively the name, the preconditions and the effects of the operator o. $\mathrm{precond}(o)$ and $\mathrm{effects}(o)$ are subsets of V. The parameters of the operator o are the variables of V.

Definition 3.6: An action in the FDR is a triple $a = (\mathrm{name}(a), \mathrm{precond}(a), \mathrm{effect}(a))$ where $\mathrm{name}(a)$ is the name of the action and $\mathrm{precond}(a)$ and $\mathrm{effects}(a)$ are partial variable assignments referred to as the action preconditions and effects. A (partial) function on V maps each $v \in V$ with a member d of $\mathrm{Dom}(v)$. This assignment is noted $(v = d)$. Finally, a state s is an assignment of state variables, and action a is applicable to s if and only if $\mathrm{precond}(a) \subseteq s$.

Definition 3.7: A planning problem in FDR is a tuple (V, A, c, I, G) where:

- V is a finite set of state variables, each $v \in V$ has a finite domain $\mathrm{Dom}(v)$,
- A is a finite set of actions,
- $c : A \to \mathbb{R}^+$ is a cost function,
- I, the initial state, is a complete variable assignment,
- G, the goal, is a partial variable assignment.

Definition 3.8: Let (V, A, c, I, G) be a FDR planning problem. The state space of (V, A, c, I, G) is the state transition system (S, A, c, γ) where:

- states S are complete variable assignments,
- actions A and the cost function c are those of the planning problem,
- the transition function $\gamma(s, a)$, when a is applicable to s, is:

$$\text{if } (v = d) \in \mathrm{effect}(a) \text{ then } (v = d) \in \gamma(s, a)$$

In a way similar to the logical representation, a *solution* to a planning problem (V, A, c, I, G) is a plan π such that $G \subseteq \gamma(I, \pi)$.

4. PDDL: the Planning Definition Description Language

Proposed for the first time in 1998 (Ghallab et al., 1998), PDDL is mainly inspired by STRIPS (Fikes & Nilsson, 1971) (STanford Research Institute Problem Solver) and ADL (Pednault, 1994) (Action Description Language). It expresses planning problems in two files: (1) the *domain* file describing the operators, and (2) the *problem* file describing the initial state and the goal.

4.1. *A simple PDDL example*

To illustrate PDDL, we introduce a classical planning domain, `Logistics`, originally proposed by S. Bart and H. Kautz at the first International Planning Competition[2]. The domain consists in moving objects by planes and by trucks between different cities.

4.1.1. Domain Description

First, consider the head of the domain description:

```
(define (domain logistics)
  (:requirements :strips :equality :typing
                 :conditional-effects :universal-effects)
  (:types physobj - object
  obj vehicle - physobj
  truck airplane - vehicle
  location city - object
  airport - location)
  (:predicates (at ?x - physobj ?l - location)
       (in ?x - obj ?t - vehicle)
       (in-city ?l - location ?c - city))
  ...
```

A domain description always starts by the declaration of its name preceded by the keyword `domain`. Then, domain descriptions define its requirements, identified by the keyword `:requirements`. The requirements allow the expressive representation of the domain description. For instance, in this `Logistics` domain description, the requirements specify that actions descriptions can use typed terms, preconditions with equality terms, and effects with conditional and universal quantifier terms.

All the domains define a few primitive types, which are `object` representing any object in the planning world and `number`. From these primitive types, it is possible to derived all the types necessary for the domain description. In the `Logistics` domain, the type `vehicle` is derived from the type `physobj`, which is itself derived from the primitive type `object`. The type descriptions are identified by the keyword `:types`.

The predicates of a domain definition, identified by the keyword `:predicates`, specify which predicate names (denoting facts about objects in the world) can be used in the preconditions and the effects of the operators, their arguments (as well as their types,

[2] `http://icaps-conference.org/index.php/Main/Competitions/`

if the domain uses `typing` as a requirement). The symbol terms are alphanumeric characters, hyphens "-" and underscores "_". Any argument term starts with a question mark "?". For instance, `(at ?x - physobj ?l - location)` expresses that an object `?x` whose type is `physobj` is located at location `?l`.

In a domain description, operators, i.e. first order actions (see Section 3.1), specify the different ways to perform changes of the world state. Consider this simple operator description:

```
(:action load
  :parameters (?o - object ?v - vehicle ?l - location)
  :precondition (and (at ?o ?l) (at ?v ?l))
  :effect (and (in ?o ?v) (not (at ?o ?l))))
```

This action means that an object `?o` can be loaded in a vehicle `?v` at a location `?l`. The preconditions require that the object `?o` and the vehicle `?v` are at the same location `?l`. After the `load` action is executed, `load` effects express that object `?o` will be in the vehicle `?v`.

Likewise, `unload` can be specified as follows:

```
(:action unload
  :parameters (?o - object ?v - vehicle ?l - location)
  :precondition (and (in ?o ?v) (at ?v ?l))
  :effect (and (not (in ?o ?v))))
```

In this action, object `?o` can be unloaded from a vehicle `?v` at a location `?l`. The preconditions assert that object `?o` has to be in the vehicle `?v`, and that the vehicle `?v` is at the location `?l`. After the `unload` is executed, the fact `(in ?o ?v)` will be false.

Now, consider a more complicated action description with conditional effects and a universal quantifier:

```
(:action fly-airplane
   :parameters (?o - object ?p - airplane ?from ?to - airport)
   :precondition (and (at ?p ?from) )
   :effect (and (at ?p ?to) (not (at ?p ?from))
                (forall (?o - object) (when (and (in ?o ?p))
                        (and (not (at ?o ?from)) (at ?p ?to)))))))
```

This specifies that an airplane `?p` can fly from a departure airport `?from` to an arrival airport `?to` if the airplane is initially at the departure airport `?from`. The effect of `fly-airplane` is to move the airplane and all its objects to the airplane destination. The predicates `(at ?p ?from)` and `(not (at ?p ?from))` in the effect description are called the unconditional effects. Conversely, the predicates in a `when`-expression are called conditional effects. The term `(and (in ?o ?p))` is the antecedent of the conditional effects, and `(and (not (at ?o ?from)) (at ?p ?to))` is its consequence.

Finally, the `drive-truck` action allows moving an object by truck:

```
(:action drive-truck
  :parameters (?o - object ?t - truck ?from ?to - location ?c - city)
  :precondition (and (at ?t ?from)
                     (in-city ?from ?c)
                     (in-city ?t ?c))
  :effect (and (at ?t ?to) (not (at ?t ?from))
               (forall (?o - object) (when (and (in ?o ?t))
```

```
                    (and (not (at ?o ?from)) (at ?o ?to))))))
```

4.1.2. Problem Description

Now, let us see how to express a planning problem. In the `logistics` example:

```
(define (problem pb1)
   (:domain logistics)
   (:objects p1 p2 - obj
             london paris - city
             truck - truck
             plane - airplane
             north south east west - location
             lhr cdg  - airport)
   (:init (in-city cdg paris) (in-city lhr london)
          (in-city north paris) (in-city south paris)
          (at plane lhr) (at truck cdg)
          (at p1 lhr) (at p2 lhr))
   (:goal (and (at p1 north) (at p2 south))))
```

The problem `pb1` starts by listing all the objects of the world and their initial state. For instance, this initial state is composed of 2 packages `p1` and `p2` and a plane at London airport `lhr`, a truck at Paris airport `cdg`, and two depots located in the north and in the south of Paris. The goal is to find a plan moving respectively `p1` to the `north` depot of Paris and `p2` to the `south` depot of Paris.

4.2. *Finite Domain Representation*

PDDL semantics are easy to understand for researchers with a background in logics, and representing the properties of the world in terms of truth values is simple. Unfortunately, this simplicity comes with a price. PDDL representation is unstructured. For instance, consider the `logistics` problem: the proposition `(at p1 west)` (expressing that package `p1` is at `west` location) is semantically related to `(at p1 north)` (expressing that package `p1` is at `north` location) because both propositions express facts about the same object. Logically, the location of `p1` gives rise to 2^4 reachable states where `p1` is at `west`, simultaneously at `west` and `north` etc. Obviously, the number of meaningful states is only 4. In order to achieve such a compact representation, planning problems can be encoded by finite domain variables, rather than binary domains. This representation is the FDR. In our example, the variable `at_p1` with the domain {`north, south, east, west`} represents all the reachable states concerning `p1` locations.

The fact that PDDL representation gives rise to a larger number of reachable states is not necessary a problem. A planner such as FastForward (Hoffmann & Nebel, 2001) does not explore any infeasible states. Nevertheless, other planning approaches benefit from the FDR. For example, planners based on constraints programming (Kambhampati, 2001; van Beek & Chen, 1999) can generate more compact Constraints Satisfaction Problem (CSP) representations by using FDR than a direct encoding of the state variables in the PDDL representation. SAT planners such as MAXPLAN (Y. Chen, Zhao, & Zhang, 2007) can reduce the search space by using a FDR to compute mutual exclusions over facts and actions, not only at a given time step, but also across multiple steps. Planners based on problem decomposition and causal graph heuristics such as Fast Downward (Helmert, 2006) build on simpler FDR causal structures. FDR is implemented in PDDL4J, and

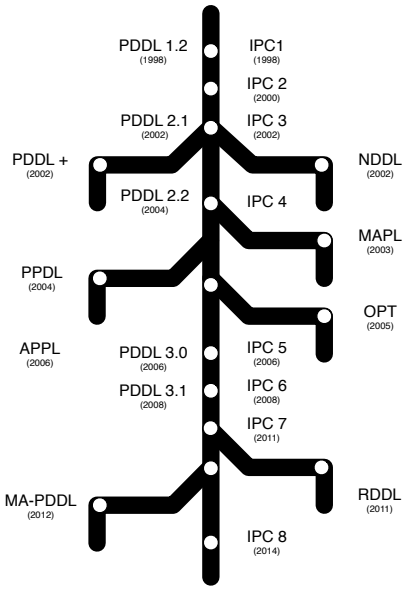

Figure 1. History and evolutions of the PDDL language

we present in Section 5.3 the algorithm used in PDDL4J to translate a PDDL planning problem into a FDR.

4.3. *PDDL evolutions and extensions*

The first version (Ghallab et al., 1998) of the PDDL 1.2 language was the official language of the 1st and the 2nd IPC (International Planning Competition) in 1998 and 2000. An overview of the history of PDDL and of its extensions is shown Figure 1. In 2002, a new version (Fox & Long, 2003), PDDL 2.1, became the official language of the 3rd IPC. To allow the representation of real-world problems, this version of PDDL introduced functions to express numerical objects, durative actions and plan-metrics assessing plan quality. In parallel, three extensions of PDDL 2.1 appeared. The first one (Fox & Long, 2002, 2006), called PDDL+, extended PDDL 2.1 to model predicable exogenous events and continuous changes. The second one (Brenner, 2003), called MAPL (Multi-Agent Planning Language), introduced finite domain state-variables, actions with duration determined at runtime, and explicit plan synchronization obtained through speech acts and communications among agents. The third one (McDermott, n.d.), called OPT (Ontology with Polymorphic Types), proposed a general-purpose notation integrating reasoning on ontologies in planning domains. At the same time, the NASA proposed the NDDL (Frank & Jónsson, 2003; Bernardini & Smith, n.d.) language. Unlike PDDL, NDDL is based on the FDR and the notions of activities and constraints on activities rather than states and actions.

In 2004, a minor release of PDDL, PDDL 2.2, was proposed for the 4th IPC. This version (Edelkamp & Hoffmann, 2004) introduced axioms in the language and the possibility of deriving predicates in order to model fact causality. It also allowed timed predicates to model exogenous events occurring at given time independently from plan execution. At the same time, PPDDL (Probabilistic PDDL) (Younes & Littman, 2004), extending PDDL 2.1 with probabilistic effects, was proposed. PPDDL has been the official language of the

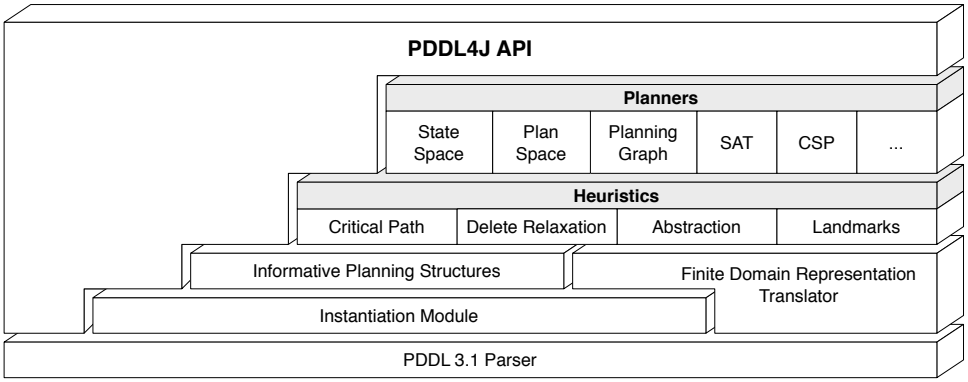

Figure 2. Overview of the PDDL4J toolkit

Probabilistic Track of the 4th and 5th IPC in 2004 and 2006 respectively.

In 2006, a new major release of PDDL was proposed for the 5th IPC. This version (Gerevini & Long, 2005c, 2005b, 2005a) introduced state-trajectory constraints, which have to be satisfied by the states generated by plan execution, and preferences, which are constraints that can be violated. In parallel, APPL (Butler & Muñoz, 2006) (Abstract Plan Preparation Language), a new version of NDDL was proposed. Its purpose was to simplify the formal analysis and specification of planning problems.

The lastest version of PDDL, PDDL 3.1 (Helmert, n.d.; Kovacs, n.d.-a) was proposed in 2008. It has been used from IPC6 to IPC8 in 2014. It introduces object-fluents, i.e. functions that manipulate any object type and not only numerical objects. Two extensions of PDDL 3.1 have to be mentioned. The first one is RDDL (Relational Dynamic influence Diagram Language) (Sanner, n.d.), which is an extension of PDDL 3.1 for probabilistic planning. The introduction of partial observability is one of the most important changes in RDDL with respect to PPDDL. RDDL has been used for the Uncertainty Track of the 7th IPC in 20011. The last extension is MA-PDDL (Multi-Agent PDDL) (Kovacs, n.d.-b) for multi-agent planning.

5. PDDL4J Architecture

PDDL4J is composed of several independent modules (see Figure 2). It has a full PDDL 3.1 parser that has been validated on all the available benchmarks of the International Planning Competitions (see Section 5.1 for the parser description and Section 6 for the validation tests). On top of the parser, PDDL4J provides a pre-processing module, the instantiation module (see Section 5.2), which is determinant for planner performances, as well as many state-of-the-art informative structures (see Section 5.4) and their corresponding heuristics (see Section 5.5). Lastly, some classical planners (see Section 5.6) are also provided.

5.1. *Domain and Problem Parsing*

The PDDL parser in PDDL4J implements PDDL 3.1 (Kovacs, n.d.-b). It ensures the lexical, syntaxical and semantical analysis of the domain and problem files, which are transformed in a compact internal representation by the instatiation module.

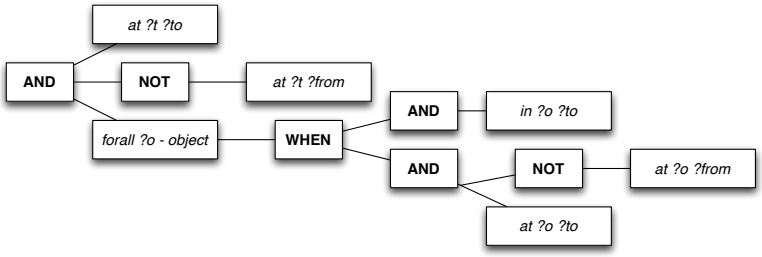

Figure 3. Tree representation of the effects of the operator *drive-truck*.

The PDDL4J parser is based on JavaCC[3], "the Java Compiler Compiler", which is a parser and lexical analyzer generator. It takes as input file a description of the PDDL language, i.e., its lexical and grammar specifications (the BNF), and generates the Java parser. JavaCC is particularly adapted to PDDL4J because it allows new PDDL language extensions and the different versions of PDDL to be easily managed without any hand-coding. Moreover, JavaCC has good error reporting, which is very useful for domain debugging.

The semantic layer of the parser proceeds to verifications. Some of them just raise warnings:

- a constant, a quantified parameter, a predicate, a function or a type declared and never used in the domain and problem files,
- a declared parameter never used in the operator,
- the domain name used in the domain file does not match with the domain name defined in the problem file, etc.

The parser also detects different errors:

- a type, a predicate, a constant or a function used but not declared,
- an inconsistent type hierarchy,
- a quantified parameter without type etc.

After these verifications, the parser generates a compact internal representation of the planning problem. First, all the parameters in the domain and problem files are renamed with a unique name. For instance, the logical formula $\psi(?x) \wedge \exists ?x\ \varphi(?x)$ is renamed $\psi(?x_1) \wedge \exists ?x_2\ \varphi(?x_2)$. Then, the formulas are encoded as tree structures with integer labels (see Figure 3): each integer maps a string, i.e., predicate name, parameter name, constant name, function name, etc.

5.2. *Instantiation Module*

The purpose of the instantiation module is to transform the operators of the planning domain into ground actions. Some planners use the instantiation in order to encode planning problems into different formalisms such as SAT (Kautz & Selman, 1992, 1999; Rintanen, 2012) or CSP (Barták, Salido, & Rossi, 2010; Kambhampati, 2000; Lopez & Bacchus, 2003) in order to benefit from the recent performance improvements of SAT and CSP solvers. Other planners use the instantiation to efficiently compute heuristics (Hoffmann & Nebel, 2001) or speedup the search algorithm by using a compact encoding (Helmert, 2009).

[3] `https://javacc.java.net`

In this section, we present the instantiation process implemented in PDDL4J. This instantiation process was first introduced by J. Koehler in the IPP planning system (Koehler, Nebel, Hoffmann, & Dimopoulos, 1997). Given an operator, for example `drive-truck` (see Section 4), the instantiation consists in replacing all the typed parameters (quantified or not) of this operator by all the corresponding typed constants declared in the planning problem. This process relies on some possible simplifications reducing the number of generated actions and consequently the planner's search space.

5.2.1. Overview of the instantiation process

The complete instantiation process has 6 steps:

(1) **Logical Normalization:** All the formulas of the form $\psi \rightarrow \varphi$ are replaced by the corresponding logical formula $\neg\psi \vee \varphi$. Then, all the negations are moved inward. For instance, the formula $\neg(\psi \wedge \varphi)$ is replaced by $\neg\psi \vee \neg\varphi$. Finally, all the universal and existential formulas are expanded to eliminate quantified formulas. For each expansion, a new tree structure of the formula and consequently of the operator is created (see Figure 4).

(2) **Inferring types:** This step is optional. It infers the types of the operator parameters whenever they are not given. The inferring process allows to the parameter domain (the domain of untyped parameters is the set of all the constants defined in the planning problem), and, consequently, the number of generated actions to be reduced.

(3) **Operator Instantiation:** All the operator parameters are replaced by typed constants. Then, these constants are propagated in the preconditions and effects formulas of the operator. For each replacement, a new tree structure of the operator is created. At the end of this step, all the operator parameters are eliminated. The instantiation process in PDDL4J makes the assumption that all the operator parameters are different.

(4) **Predicate Simplification:** This step consists, if possible, in the substituting the predicates in operators by *true* or *false*. For instance, the predicate `(in-city ?location ?city)` never occurs as a positive or negative effect of any operator. Therefore, the only possible instantiations of `(in-city ?location ?city)`, whatever the state, are those defined in the initial state. Thus, all the ground predicates matching `(in-city ?location ?city)`, like for instance `(in-city lhr paris)`, can be simplified to *false* in the operators if they do not hold in the initial state. Predicate simplifications are performed as soon as possible to reduce the number of operator instantiations.

(5) **Operator Simplification:** This step is a generalization of the predicate simplification to first-order logic formula and operators. For instance, consider an AND formula (see Figure 4), if one of the propositions $p_1, \ldots, p_n$ can be simplified to *false* then the whole AND formula can be simplified to *false*. By extension, an action whose preconditions can be simplified to *false* can be removed from the planning problem because this action will never be triggered.

(6) **Bit Set Problem Encoding:** Once all the operators are instantiated into actions and no more simplifications are possible, the instantiation process ends by encoding the planning problem within a compact bit set representation in order to reduce the memory consumption. A state is encoded with two bit sets: one to encode the positive ground predicates and one for the negative ground predicates. Each ground predicate corresponds to a position in a bit set. The set of relevant predicates is deduced from the simplification and instantiation steps. By extension, the action

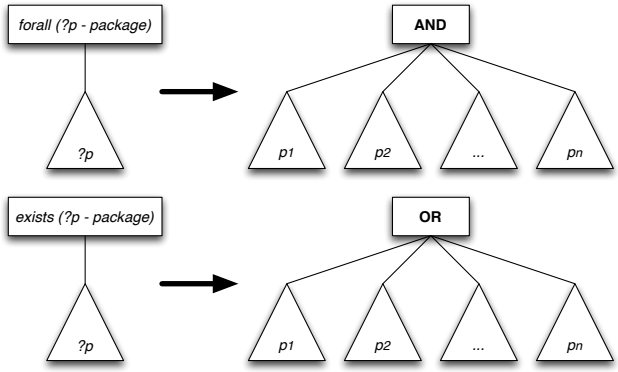

Figure 4. Elimination of the quantified formulas by expansions.

Table 1. Classification of the predicates used in the `logistics` domain.

Predicate name	**Positive effect**	**Negative effect**	**Predicates category**
in-city	no	no	inertia
at	yes	yes	fluent
in	yes	yes	fluent

preconditions are encoded as a state and the conditional effects as a list of couples of states, one for the condition and one for the effects.

5.2.2. Predicate Simplification

The predicate simplification of an operator consists, whenever possible, in replacing the predicates used in its preconditions and effects by *true* or *false*. Deciding if a simplification can be done relies on two steps: (1) the characterization of the predicates according to properties that can be preprocessed, and (2) the application of simplification rules whose triggering is based on the presence of the predicates in the initial state of the planning problem.

The predicate characterization is done by a single pass over the domain description. Each predicate is classified into four categories: a predicate is a *positive inertia* if and only if it never occurs positively in an unconditional effect or the consequent of a conditional effect of an operator. A predicate is a *negative inertia* if and only if it never occurs negatively in an unconditional effect or the consequent of a conditional effect of an operator. Predicates that are neither positive nor negative inertia are *fluents*. For instance, Table 1 summarizes the predicate categories in the `logistics` domain.

The simplification rules can be formally defined as follows. A partially instantiated predicate $(p\overrightarrow{a})$, where $\overrightarrow{a}$ defines uninstantiated parameters of p, can be simplified to *true* or *false* by applying the following rules during the instantiation:

Rule 1. If p is a positive inertia and $N(p\overrightarrow{a}) = 0$ then $(p\overrightarrow{a})$ is simplified to *false*,
Rule 2. If p is a negative inertia and $N(p\overrightarrow{a}) = Max(p\overrightarrow{a})$ then $(p\overrightarrow{a})$ is simplified to *true*.

Where in these rules:

- $N(p\overrightarrow{a})$ denotes the number of ground instances of p contained in the initial state,
- $Max(p\overrightarrow{a})$ denotes the number of ground instances of p that can be generated from the uninstantiated parameters in $\overrightarrow{a}$. $Max(p\overrightarrow{a})$ is the cartesian product of the

domain of each parameter in $\overrightarrow{a}$.

For instance, consider the predicate *(in-city ?location amsterdam)*. This predicate is a positive inertia. Because it does not match with any ground predicate in the initial state, N(`(in-city ?location amsterdam)` $= 0$, and the rule 1 can be applied: `(in-city ?location amsterdam)` is substituted by *false*. Now, consider the predicate `(in-city lhr london)`. This predicate is a negative inertia and matches one occurrence in the initial state, thus N(`(in-city lhr london)` $= 1$. Moreover, Max(`(in-city lhr london)` $= 1$ because `(in-city lhr london)` is a ground predicate. Therefore, rule 2 can be applied: (`(in-city lhr london)` is simplified by *true*. This means that `(in-city lhr london)` is *true* in the initial state and no operator will make it *false*.

When all the predicate simplifications have been done and the instantiation process ends, all the remaining ground predicates are necessarily fluent. All the other ground predicates have been simplified by *true* or *false*. Only fluents are relevant for planning and encoded in PDDL4J internal representations. PDDL4J simplification process is based on Koehler's work (Koehler & Hoffmann, 1999) (this algorithm requires time proportional to the maximal arity of the predicates).

5.2.3. Inferring types from unary predicates

Typing is important because it reduces the allowed instantiations and, thus greatly improves the efficiency of the instantiation process. However, formally, typing domain descriptions is not mandatory and, most of the time, unary predicates are used for typing. For instance, `(vehicle ?v)` indicates that parameter $?v$ is of type "vehicle". Unary predicates are inertias that are neither produced nor consumed by the operators. For instance, in the following untyped `load` operator:

```
(:action load
  :parameters (?o ?v ?l)
  :precondition (and (object ?o) (vehicle ?v) (location ?l)
                     (at ?o ?l) (at ?v ?l))
  :effect (and (in ?o ?v) (not (at ?o ?l))))
```

The three predicates `(object ?o)`, `(vehicle ?v)` and `(location ?l)` are unary inertias. They encode respectively the types *object*, *vehicle* and *location* of the `logistics` planning domain (see Section 4.1).

In order to speed-up the instantiation process, the PDDL4J instantiation module implements a type inference procedure inspired from IPP (Koehler, 1999). This procedure has three steps:

(1) The extraction of all the unary inertias $(p\ c) \in \mathcal{I}$ (the initial state) where c is a constant,
(2) The creation for every inertia $(p\ c)$ of a new type τ_p whose domain is $\text{Dom}(\tau_p) = \{c \mid (p\ c) \in \mathcal{I}\}$;
(3) The replacement of all the unary inertias by their corresponding inferred types in the operator descriptions.

Let o be an operator and $?x$ be one of its parameters. Let p be an unary inertia and $(p\ ?x)$ be a predicate that occurs in the preconditions or in the antecedents of one of the conditional effects of o: then o is rewritten as two operators o_1 and o_2:

- In o_1 all the occurrences of $(p\ ?x)$ are replaced by *true* and the type τ initially used to declare $?x$ is restricted to $\tau \cap \tau_p$. $\tau \cap \tau_p$ is a new type whose domain is

$\mathrm{Dom}(\tau \cap \tau_p) = \mathrm{Dom}(\tau) \cap \mathrm{Dom}(\tau_p)$,

- In o_2 all the occurrences of $(p\ ?x)$ are replaced by *false* and the type τ initially used to declare $?x$ is restricted to $\tau \backslash \tau_p$. $\tau \backslash \tau_p$ is a new type whose domain is $\mathrm{Dom}(\tau \backslash \tau_p) = \mathrm{Dom}(\tau) \backslash \mathrm{Dom}(\tau_p)$.

Similarly, quantified formulas in preconditions or antecedents of conditional effects are replaced. Consider once again the operator `load`. It is replaced by the two following operators such as $t_1 = object \cap\ \tau_{obj}$ and $t_2 = object \backslash\ \tau_{obj}$:

```
(:action load_1
  :parameters (?o - t1 ?v ?l)
  :precondition (and (true) (vehicle ?v) (location ?l)
                     (at ?o ?l) (at ?v ?l))
  :effect (and (in ?o ?v) (not (at ?o ?l))))

(:action load_2
  :parameters (?o - t2 ?v ?l)
  :precondition (and (false) (vehicle ?v) (location ?l)
                     (at ?o ?l) (at ?v ?l))
  :effect (and (in ?o ?v) (not (at ?o ?l))))
```

In the first operator, the simplification process removes *true*, and, in this second operator, invalidates all the preconditions: this operator will never be applicable and is removed from the planning domain. However, remember that preconditions can be any logical formula and simplification follows the usual contraction rules. The simplification process is iterated over all the operator parameters. It terminates when an inferred type is assigned to each parameter and all the unary inertias are removed from the operators.

5.2.4. Operator simplification

Action simplification is a generalization of predicate simplification applied to preconditions and effects in actions. Action simplifications are performed as soon as possible in order to reduce the cost of operator instantiations. They are based on the following syntaxical rules:

Rule 1. $\neg true \equiv false$ and $\neg false \equiv true$,
Rule 2. $true \wedge \varphi \equiv \varphi$, $false \wedge \varphi \equiv false$, $true \vee \varphi \equiv true$ and $false \vee \varphi \equiv \varphi$,
Rule 3. $\varphi \wedge \varphi \equiv \varphi$, $\varphi \wedge \neg\varphi \equiv false$, $\varphi \vee \varphi \equiv \varphi$ and $\varphi \vee \neg\varphi \equiv true$,
Rule 4. If a quantified parameter is never used in the quantified formula, the quantifier is removed, i.e., $\forall ?x\ \varphi(?y)$ is simplified to $\varphi(?y)$,
Rule 5. If a quantified parameter has an empty domain, the quantified formula is replaced by *true* in the case of an universal quantifier and by *false* in the case of an existential quantifier,
Rule 6. All equality formulas $?x = ?x$ ($?x$ is a parameter) and $c = c$ (c is a constant) are replaced by *true*. Equalities such as $c_1 = c_2$ are simplified to *false*.

When the precedent rules reduce the precondition, antecedent or consequent of an operator to *true* or *false*, the following rules apply:

Rule 1. If the antecedent of a conditional effect is replaced by *false*, the conditional effect is removed from the operator: this effect will never be applicable,
Rule 2. If the antecedent of a conditional effect is replaced by *true*, the conditional effect becomes unconditional: this effect is always applicable,

Rule 3. If the consequent of a conditional effect is replaced by *true*, the conditional effect is removed from the operator because it will never lead to any state transition,

Rule 4. If the precondition or an unconditional effect of an operation is replaced by *false*, the operator is removed from the planning domain because it will never be applied in any state,

Rule 5. If the effect of an operator is *true*, the operator is removed from the planning domain because it will never generate any new state.

5.3. *Finite Domain Representation Encoder*

Finite Domain Representation (FDR) is generally more compact than PDDL (see Section 4.2). FDR and logical representations are equivalent and translating one representation into the other one takes polynomial time (Helmert, 2009).

5.3.1. A naive encoding procedure

First, consider a naive procedure to encode logical representation into FDR. Let $\mathcal{P}_\mathcal{L} = (P, A, c, I, G)$ be a logical planning problem. The FDR translation of $\mathcal{P}_\mathcal{L}$ is the FDR planning problem $\mathcal{P}_\mathcal{F} = (V, A_\mathcal{F}, c, I_\mathcal{F}, G_\mathcal{F})$. This procedure creates for each proposition $p \in P$ of $\mathcal{P}_\mathcal{L}$ one boolean FDR variable v_p and applies trivial changes to A, I, G:

- $V = \{v_p \mid p \in P\}$ is the set of boolean variables, one for each proposition in P,
- $A_\mathcal{F} = \{a_\mathcal{F} \mid a \in A\}$ are actions where:
  - $pre(a_\mathcal{F}) = \{v_p = true \mid p \in pre(a)\}$ and
  - $effect(a_\mathcal{F}) = \{v_p = true \mid p \in effect^+(a)\} \cup \{v_p = false \mid p \in effect^-(a)\}$,
- $I_\mathcal{F} = \{v_p = true \mid p \in I\}$,
- $G_\mathcal{F} = \{v_p = true \mid p \in G\}$.

This naive procedure produces a FDR that is no more compact than the PDDL representation. A clever encoding consists in finding invariant states or sets of propositions that can never be true at the same time, i.e. "mutexes". Indeed, each set of mutex propositions can be encoded as a single variable whose values can be mapped with the propositional values.

5.3.2. Invariants Discovery

We are only interested in invariants that express mutual exclusive sets of propositions because they allow the grouping of a set of mutex propositions into a single finite domain variable. For instance in the `logistics` domain, $\forall ?p$ `(at ?p ?l)` and `(in ?p ?v)` state that whatever the package `?p` the propositions `(at ?p ?l)` and `(in ?p ?v)` are mutually exclusive. Then we can encode the proposition sets `(at p1 ?l)` and `(in p1 ?v)` with only one variable `p1` $\in$ {`at-lhr`, `at-cdg`, `at-north`, `at-south`, `in-truck`, `in-plane`}.

In practice, there are too many candidate invariants to enumerate them exhaustively. The PDDL4J invariant discovery procedure follows a guess, check, and repair approach. This approach was first implemented in the planner FastDownward (Helmert, 2006). The invariant discovery procedure starts with a set of few simple candidate invariants. Then, it tries to prove that each candidate invariant is really an invariant. If the proof succeeds, this candidate is kept as invariant, otherwise, it determines why this candidate fails and tries to refine it in order to generate other invariant candidates. At the top level, the discovery of invariants is a classical breadth-first search based on an opened and a closed list to store the pending candidates and the explored candidates. The complete

Table 2. Relevant propositions and mutex-groups of the `logistics` problem.

Mutex-Group 1	**Mutex-Group 2**	**Mutex-Group 3**	**Mutex-Group 4**
(at p1, lhr)	*(at p2, lhr)*	*(at truck, cdg)*	*(at plane, lhr)*
(at p1, cdg)	*(at p2, cdg)*	*(at truck, north)*	*(at plane, cdg)*
(at p1, north)	*(at p2, north)*	*(at truck, south)*	
(at p1, south)	*(at p2, south)*		
(in p1, truck)	*(in p2, truck)*		
(in p1, plane)	*(in p2, plane)*		

description of the PDDL4J algorithm is in (Helmert & Domshlak, 2009).

5.3.3. FDR Encoding

The FDR translation procedure implemented in PDDL4J is based on invariant discovery and the instantiation process (see Section 5.2). The FDR translation procedure involves 3 mains steps described below.

Variable Selection

Recall that each variable in the FDR corresponds to one or more facts of the world in the logical representation. In order to have a compact representation, we want to represent as many propositions as possible by a single variable induced by the computed invariants. The procedure implemented in PDDL4J does not guarantee finding the most compact representation (this is a NP-complete problem), but it uses the best approximation algorithm (Ausiello, Crescenzi, Gambosi, Kann, & Marchetti-Spaccamela, 1999). The variable selection procedure consists in four steps:

Step 1. The procedure starts by extracting the relevant propositions of the planning problem. This is quickly done by extracting all the propositions defined in the preconditions and the effects of the grounded operators. For instance, in the LOGISTICS problem (see §4.1), the relevant propositions are those listed in Table 2.

Step 2. The procedure instantiates the computed invariants. In the `logistics` domains, two invariants are extracted:

(1) ∀`?p (at ?p ?l) and (in ?p ?v)` are mutually exclusive: whatever the package `?p`, `?p` is either at a location `?l` or in a vehicle `?v`. This invariant produces two partial instantiations: `(at p1 ?l) is mutually exclusive with (in p1 ?v)`, and `(at p2 ?l) is mutally exclusive with (in p2 ?v)`,

(2) ∀`?v (at ?v ?l)` stating that whatever the vehicle `?v`, the vehicle is at only one location `?l`. This invariant produces two partial instantiations: `(at truck ?l)` and `(at plane ?l)`.

Step 3. Then, for each partial instantiation satisfying the initial state, a *mutex-group of propositions* is created. This mutex-group of propositions contains all the relevant propositions of the planning problem that unify with a predicate of the partial instantiations. In our example, four mutex-groups shown in Table 2, one for each partial instantiation, are generated;

Step 4. Finally, the procedure selects the mutex-group with the highest cardinality and creates a variable of the domain that contains all the propositions of the mutex-group plus the ⊥ value that stands for "none of propositions in the mutex-group is true". All the propositions of this mutex-group are removed from the other mutex-groups, and empty mutex-groups are removed from the list of mutex-groups. The process is repeated until this list is empty. In the `logistics` domain, the following variables are created:

- `?p1, ?p2` $\in$ {`at-lhr, at-cdg, at-north, at-south, in-truck, in-plane`},
- `?truck` $\in$ {`at-cdg, at-north, at-south`},
- `?plane` $\in$ {`at-lhr, at-cdg`}.

Initial state and goal encoding

Based on the variables selected, encoding the initial state and the goal is straightforward. For each proposition p of the initial state or goal, the corresponding variable is set to the value "p". The other variables are set to $\perp$. For instance, the initial state I in `logistics` is encoded as I = {`?p1 = at-lhr, ?p2 = at-lhr, ?truck = at-cdg, ?plane = at-lhr`}.

Action encoding

The encoding of the preconditions and the positive effects of an action is similar to the initial state and goal encoding. Nevertheless, encoding the negative effects requires some care as the same variable can be simultaneously involved in positive and negative effects. A variable encoding a negative effect is set to $\perp$ only if no positive effect is encoded with the same variable. For instance, the action `(load-truck p1 truck cdg)` is encoded with the preconditions {`?p1 = at-cdg, ?truck = at-cdg`} and the effect {`?p1 = in-truck`}.

5.4. *Informative Planning Structures*

Many research works have been investigating the use of informative data structures to facilitate automated planning. These structures are mainly used to encode the search space in a compact manner or to extract relevant information from the planning problem to build efficient heuristics. In this section, we introduce the two most important informative data structures used for automated planning and implemented in PDDL4J: the planning graph and the causal graph. These two structures are the basic ones used to compute the heuristics described in Section 5.5 and used in some of the planners implemented in PDDL4J (see Section 5.6).

5.4.1. Planning Graph

The planning graph structure was first proposed in the Graphplan (Blum & Furst, 1997) planner. It provides an efficient way to estimate which set of propositions is reachable from a given state. The estimation is achieved by a reachability analysis based on a relaxed condition of reachable states. The basic idea of the planning graph structure is to calculate, as a first approximation of reachable states, a set of propositions that can be reached by all the applicable actions at the same time from a given state. However, while a state is consistent, this set of propositions represents several possibly inconsistent states. Likewise, not all the actions are simultaneously applicable. Therefore, mutually inconsistent propositions and actions, *mutexes* in planning graph terminology (mutual exclusion relation), are labelled. An example of a planning graph based on the `logistics` domain is shown Figure 5.

Formally, a planning graph PG of a planning problem $\mathcal{P}_{\mathcal{L}} = (P, A, c, I, G)$ in the logical representation is a directed graph organized as an alternated sequence of "levels" composed of propositions and actions $\langle P_0, A_0, P_1, \ldots, A_{i-1}, P_i, \ldots\rangle$. Each proposition layer P_i is a subset of P. P_0, the first level of the planning graph contains all the propositions

of the initial state I. The level A_i with $i > 0$ is an action level containing all the actions $a \in A$ applicable from the proposition level P_i, and P_{i+1} contains all the propositions generated by the effects (both positive and negative) of the actions in A_i. Edges of the planning graph explicitly represent relations between actions and propositions. Actions in an action level A_i are connected by precondition edges to their preconditions in level P_i, by add-edges to their positive effects in level P_{i+1}, and by delete-edges to their negative effects in level P_{i+1}. At each level P_i, every proposition $p \in P_i$ is propagated to the next level P_{i+1} by a dummy action that has one precondition p and one positive effect p. Two actions a and b in level A_i are *mutex* if (i) a deletes a precondition of b, (ii) a deletes a positive effect of b or (iii) a precondition of a is mutex with a precondition of b. Two propositions p and q in P_i are mutex if every action in A_{i-1} that has p as a positive effect (including dummy actions) is mutex with every action that produces q. The sets of mutual exclusion relations at a proposition level P_i and an action level A_i is respectively μP_i and μA_i. For instance, in Figure 5, at the action level 1, the action `(fly-airplane plane lhr cdg)` is mutex with the action `(load-airplane p1 plane lhr)` and `(load-airplane p2 plane lhr)`.

Every planning graph PG has a *fixed-point level*. This fixed-point level is a level i such that for $\forall j > i$, level j is identical to level i, i.e., $P_i = P_j$, $\mu P_i = \mu P_j$, $A_{i-1} = A_{j-1}$ and $\mu A_{i-1} = \mu A_{j-1}$. It is easy to verify the following assertions:

- The number of actions required to produce a proposition is no less than the index of the smallest proposition level in the planning graph in which this proposition appears. This can be generalized to a pair of propositions. In this case, the number of actions to produce a pair of propositions is no less than the index of the smallest proposition level in which both propositions appear without mutex,
- The set of actions at the fixed-point level contains all the actions applicable to states reachable from the initial state.

These assertions give some indications on how the information in the planning graph can be used to guide the search of the planners. The first assertion shows that the level index in the planning graph can be used to estimate the "cost" of deriving a set of propositions. The second assertion shows which actions have to be considered by search algorithm, and which actions have to be discarded etc.

The good news is that the computation of a planning graph takes polynomial time: it depends on the number of actions and propositions of the problem (Kambhampati, Parker, & Lambrecht, 1997). In PDDL4J, the planning graph is based on an efficient implementation (a compact bit vector representation) proposed in the planner IPP (Koehler, 1999) (different implementations exist, e.g., STAN (Long & Fox, 1999), Blackbox (Kautz & Selman, 1999)).

5.4.2. Causal Graph

The concept of causal graph was first introduced by Knoblock (Knoblock, 1994). A causal graph captures the structure of a planning problem in a meaningful way: it captures the degree of independence among the state variables of a planning problem encoded in FDR. It has been used as a tool for describing tractable subclasses of planning problems (Brafman & Domshlak, 2003; P. Jonsson & Bäckström, 1998; Williams & Nayak, 1997), for decomposing planning problem into smaller ones (Brafman & Domshlak, 2006; A. Jonsson, 2007; Knoblock, 1994), as the basis for domain-independent heuristics to guide the search toward a solution plan (Helmert, 2006), for complexity analysis and particularly the identification of polynomial solvable sub-classes, and the analysis of search topology

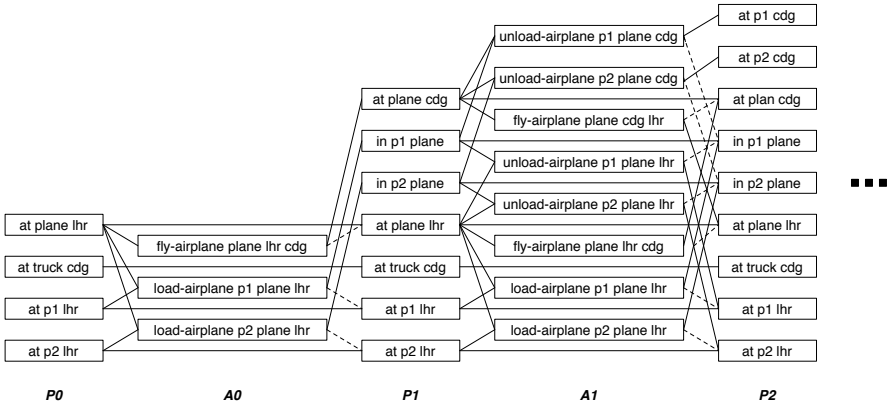

Figure 5. The two first levels of the planning graph for the `logistics` problem. Each box is either a proposition or an action. Solid lines represent preconditions and positive effects of actions, and dashed lines represent negative effects. For readability, mutexes are not shown.

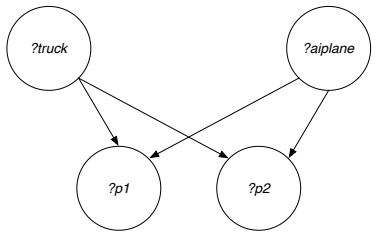

Figure 6. Causal graph of a `logistics` problem: vertices are state variables in Finite Domain Representation and arcs show variable dependencies.

of planning benchmarks (Hoffmann, 2011).

Formally, a causal graph CG for a FDR planning problem $\mathcal{P}_{\mathcal{F}} = (V, A, c, I, G)$ is a directed graph whose nodes are the state variables V of $\mathcal{P}_{\mathcal{F}}$. The arcs of the graph show the variable dependencies. There is an arc (v, u) between two variables v and u if and only if $v \neq u$ and there exists an action $a \in A$ such as $v \in \text{precond}(a)$ and $u \in \text{effects}(a)$, or $v \in \text{effects}(a)$ and $u \in \text{effects}(a)$.

The first condition means that modifying v may also modify u, and the second means that modifying v may have a side effect on u as well. To illustrate the concept of a causal graph, consider Figure 6. It depicts the causal graph of a `logistics` problem. We can observe that packages `p1` and `p2` are independent, but that moving the truck or the airplane modifies the state of packages `p1` and `p2`. Note that contrary to planning graphs, causal graphs capture only the structure of variables and actions, and cannot by design account for the influence of different initial states and goals.

5.4.3. Domain Transition Graph

A last useful informative structure often considered in connection with the causal graph and implemented in PDDL4J is the *Domain Transition Graph*. This graph describes how an individual state variable evolves with respect to the actions of a planning problem.

Let $\mathcal{P}_{\mathcal{F}} = (V, A, c, I, G)$ be a planning problem in FDR and $v \in V$. The Domain Transition Graph of v is a directed labeled graph $DTG(v, P)$ whose vertices are the values of $\text{Dom}(v)$ and arcs are labeled with actions $a \in A$. There is an arc (d, d') between two vertices d and d' if $(v = d) \in \text{precond}(a)$ and $(v = d') \in \text{effects}(a)$ or $v \notin \text{precond}(a)$

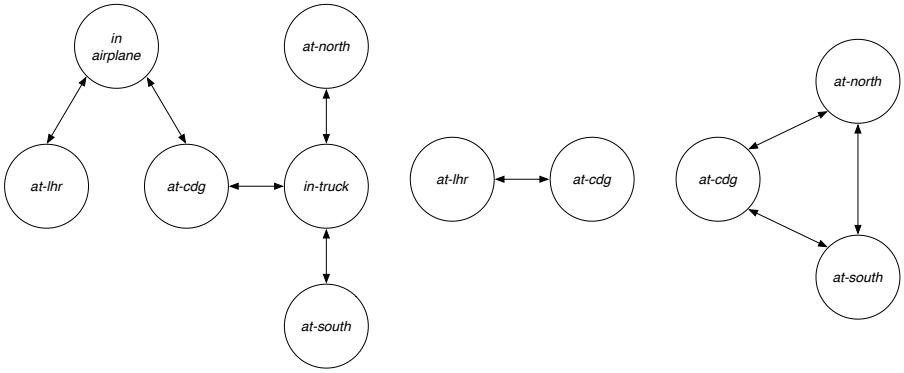

Figure 7. Domain Transition Graphs of a `logistics` problem for the variables `?p1` and `?p2` (left), the variable `?airplane` (centre) and the variable `?truck` (right): vertices are values of the state variables and arcs possible transitions. For simplicity, action labels are not depicted.

and $(v = d') \in \text{effects}(a)$. The Domain Transition Graph is said to be invertible if, for each arc (d, d'), there is an arc (d', d) in the graph. The Domain Transition Graphs of the state variables of the `logistics` problem are shown in Figure 7. It is proved (H. Chen & Giménez, n.d.) (1) that plan existence for a planning problem with an acyclic causal graph and whose Domains Transition Graphs are all invertible can be decided in polynomial time, and (2) that if the causal graph contains cycles deciding if a plan exists is NP-Hard.

5.5. *Planning Heuristics*

Since 2000, planners based on heuristic search have won all the deterministic tracks of the International Planning Competition. Heuristic search (Bryce & Kambhampati, 2007; Haslum, Botea, Helmert, Bonet, & Koenig, 2007; Liang, 2012) has become a de facto standard method to fin plans in large deterministic search space. This approach has revolutionized planning, both in terms of scalability and in terms of research methodology. Its principle is to guide the search by using a heuristic function h that estimates the cost h^* of an optimal path from a state s to the goal g. In other words, heuristic search procedures use h to decide which pending states s to expand first, i.e., the states with the smallest value $h(s, g)$. An important property for heuristics is their admissibility. Indeed, if an heuristic is *admissible*, it can be proved that the first solution plan returned by the classical A* search procedure is optimal in terms of plan length. A heuristic is admissible if it never overestimates the optimal cost h^* from s to g.

A common key idea to derive heuristics is "relaxation", i.e., ignoring some features in the original problem to obtain a simpler problem, which has polynomial time solutions. There are essentially four categories of methods to automatically generate heuristic functions (Helmert & Domshlak, 2009):

- *Critical path heuristics* (Haslum & Geffner, 2000; Haslum, Bonet, & Geffner, 2005) estimate the goal distance by computing lower bound estimates on the cost of achieving sets of facts of a predefined size,
- *Delete relaxation heuristics* (Nguyen, Kambhampati, & Nigenda, 2002; Hoffmann & Nebel, 2001; Domshlak, Hoffmann, & Katz, 2015) estimate the cost of reaching a goal state by ignoring the negative effects of actions,
- *Abstraction heuristics* (Ededlkamp, 2001; Helmert, Haslum, Hoffmann, & Nissim, 2014; Katz & Domshlak, 2008) try to collapse several states into one depending on

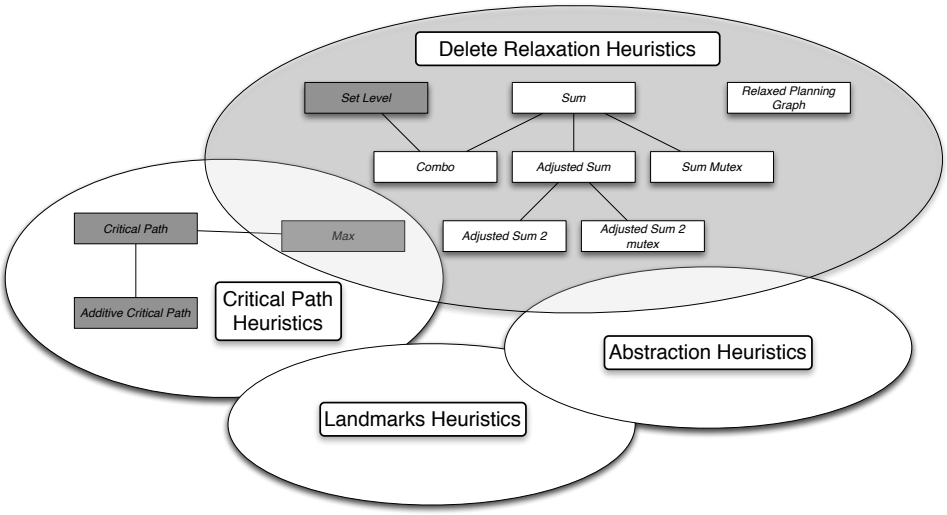

Figure 8. Overview of the heuristics available in the PDDL4J library

their properties in order to reduce the size of the search space. When the abstraction is small enough it is feasible to quickly find an optimal solution for the abstracted problem that is used as a heuristics value during the search,

- *Landmarks heuristics* (Hoffmann, Porteous, & Sebastia, 2004; Richter, Helmet, & Westphal, 2008) are based on the observation that some propositions are true at some point in every plan to reduce and decompose the search space.

For now, PDDL4J only implements critical path and delete relaxation heuristics (see Figure 8. Heuristics in gray are admissible heuristics).

5.5.1. Critical path heuristics

The critical path heuristic (Haslum & Geffner, 2000) h^m is defined by a relaxation of the planning problem in which the cost of deriving a set of propositions is approximated by the cost of the most costly subset of size m. Formally, it is defined as follows:

$$h(s,g)^m = \begin{cases} 0 & \text{if } \ g \subseteq s \\ min_{a \in A}(cost(a)) + h^m(s, \gamma^{-1}(g,a)) & \text{if } |g| \leq m \\ max_{g' \subseteq g \wedge |g'| \leq m} \ h^m(s,g') & \text{otherwise} \end{cases} \quad (1)$$

Where $\gamma^{-1}(g,a) = (g - \text{effet}^+(a)) \cup \text{precond(a)}$: a are "relevant" actions contributing to reach the goal g, and $\gamma^{-1}(g,a)$ is the new goal to achieve from a state s.

$h^m(s,g)$ is admissible. For fixed m, $h^m(s,g)$ can be computed in polynomial time: the greater m, the more informative is the heuristic. For small values of m, an explicit solution can be computed by a simple shortest-path algorithm such as Dijkstra. The solution can be stored in a table of heuristic values for sets of m or fewer predicates, so that only $max_{g' \subseteq g \wedge |g'| \leq m} \ h^m(s,g')$ has to be computed when a state s is evaluated during the search. This heuristic is relatively slow to compute at each state, and it is mainly used in backward search.

The PDDL4J library also implements an admissible additive variant (Haslum et al., 2005) of h^m. The main improvement is to partition the set of actions A, from which h^m is computed, into n disjoint subsets. Then, for each subset, the heuristic h^m is separately

computed and the sum of these individual computations is returned as the heuristic value. This computation requires less time than the original heuristic.

5.5.2. Delete relaxation heuristics

In these heuristics, the idea is to relax planning problems by neglecting the negative effects of the actions. Thus, once a proposition becomes true, it remains true forever. The transition function of the relaxed planning problem monotonically increases the number of propositions. Hence, it is easier to estimate the distances to the goal (recent works consider partial relaxation (Domshlak et al., 2015)). PDDL4J implements the following heurisitics:

Additive Heuristic. The additive heuristic (Bonet & Geffner, 2001) h^+ as in all the heuristics of the delete relaxation heuristics category ignores the negative effects of the actions. It approximates the distance from a state s to a goal g as the sum of the distances to the propositions $p \in g$. h^+ is given by the following equations:

$$h^+(s,g) = \Sigma_{p \in g}\ h^+(s,p) \tag{2}$$

where:

$$\begin{array}{ll} h(s,p)^+ = 0 & \text{if } \ p \in s \\ h(s,p)^+ = \infty & \text{if } \ \forall a \in A, p \notin \text{effect}^+(a) \\ h(s,p)^+ = min(cost(a)) + h^+(s,pre(a)) & \text{if } \ \exists a \in A, p \in \text{effect}^+(a) \end{array}$$

This heuristic requires polynomial time in the number of propositions and actions of the planning problem. However h^+ is not admissible.

Maximum Heuristic. In contrast to h^+, the maximum heuristic (Bonet & Geffner, 2001) h^{max} is admissible. It estimates the distance to the goal g as the largest distance to its propositions:

$$h^{max}(s,g) = max_{p \in g}\ h^+(s,p) \tag{3}$$

Although h^{max} is admissible, in practice, it is less informative than the additive heuristic. Note that h^{max} is equivalent to h^m when $m = 1$.

Additive Mutex Heuristic. The additive mutex heuristic h^{+M} improves the additive heuristic by using mutex relations as in Graphplan (Blum & Furst, 1997). The mutex relation used in h^{+M} are "structural" mutexes (time independent mutexes). Two propositions p and q are mutex when they cannot be present together in any reachable states from the initial state of the planning problem. The additive mutex heuristic is as follows:

$$h^{+M}(s,g) = \begin{cases} \infty & \text{if } \exists (p,q) \in s \text{ and } mutex(p,q) \\ \Sigma_{p \in g}\ h^+(s,p) & \text{otherwise} \end{cases} \tag{4}$$

h^{+M} is admissible, but empirical evaluations (Nguyen et al., 2002) show that h^{+M} is more informative than h^+ specially when subgoals are relatively independent. The procedure used to compute mutex relations starts with a large set of potential mutex pairs and

then precludes those that are not actually mutex. The initial set M of mutex pairs is defined as the union of two sets M_1 and M_2 defined as follows:

- M_1 is the set of pairs $P = \{p, q\}$ such that some action adds p and deletes q,
- M_2 is the set of pairs $P = \{r, q\}$ such that for some pair $m' = \{p, q\}$ in M_1 and some action a, $r \in pre(a)$ and $p \in \text{effect}^+(a)$.

Then, for each mutex pair $m = \{p, q\} \in M$, given a set of actions A and an initial state $\mathcal{I}$, m is not derived if and only if m is not true in $\mathcal{I}$. Therefore, for every $a \in A$ that adds p, either a deletes q, or a does not add q, and for some precondition r of a, $m' = \{r, q\}$ is a pair in M.

Set-Level Heuristic. The set-level heuristic (Nguyen et al., 2002) computation is based on the extraction of information from a planning graph structure. Suppose a planning graph $PG = \langle P_0, \mu P_0, A_0, \mu A_0, \ldots, P_n, \mu P_n \rangle$ is expanded until its fixed point is reached. P_i and A_i represent respectively the i^{th} propositional and action level, and μP_i and μA_i, the propositional and action mutex relation at level i. We call $level(p)$ the first propositional level of the planning graph where p appears. By extension, $level(g)$ denotes the first level of the planning graph where all the propositions of the goal g are present in the propositional level P_i and are mutex free, i.e., $g \notin \mu P_i$. The set-level heuristic is:

$$h^{lev}(s, g) = \begin{cases} level(g) \\ \infty \qquad \text{otherwise} \end{cases} \tag{5}$$

The set-level heuristic is admissible and more informative than the maximum heuristic. This is because the maximum heuristic is equivalent to the set-level heuristic without mutex constraints. Moreover, in practice, the set-level heuristic outperforms the additive heuristic because it subsumes most of the structural mutex relations computed by the additive heuristic.

Adjusted Additive Heuristic. The adjusted additive heuristic (Nguyen et al., 2002) tries to avoid the drawbacks of the set-level heuristic that tends to underestimate the goal distance as well as those of the additive heuristic that tends to overestimate the goal distance by neglecting the negative effects of the actions. The main idea of the adjusted additive heuristic is to make a partition of k subsets of the goal propositions and to apply the set-level heuristic on each subset. The adjusted additive heuristic is:

$$h^{+Adj}(s, g) = \underbrace{\Sigma_{p \in g}\ h^+(s, p)}_{h^+(s,g)} + \Delta(g) \tag{6}$$

where

$$\Delta(g) = level(g) - max_{p \in g} level(p) \tag{7}$$

$\Delta(g)$ can be considered as a measure of the interdependence between the goal propositions contained in g. In the best case, if there is no interdependence in g, i.e., all the goal propositions are independent, $\Delta(g) = 0$ and the adjusted additive heuristics is equivalent to h^+. However, the adjusted additive heuristics, like h^+, is not admissible.

PDDL4J also implements two improvements of the adjusted additive heuristic. The first improvement consists in taking into account the propositions that are derived by an action. The computation of h^+ in equation 6 is then given by:

$$cost(s, g) = cost(a) + cost(s, g \cup pre(a) - \text{effect}^+(a)) \tag{8}$$

where a is an action that derives $p \in g$ such that $level(p) = max_{p_i \in g} level(p_i)$. Note that recursively applying this equation implies extracting a sequence of actions. In this sense, this adjusted additive heuristic variant is similar to the relaxed plan heuristic develop by J. Hoffmann and B. Nebel in the FastForward planner (Hoffmann & Nebel, 2001).

The second improvement consists in improving the computation of $\Delta(g)$ by counting more precisely the interdependences between the goal propositions. Instead of considering goal propositions individually, the interdependence between each pair p and q of goal propositions is considered. $\Delta(g)$ in equation 7 is then given by:

$$\Delta_{max}(g) = max_{p,q \in g} level(\{p, q\}) - max(level(p), level(q)) \tag{9}$$

Combo Heuristic. As with the adjusted additive heuristic, the combo heuristic tries to take advantage of combining two heuristic: the set-level heuristic and the additive heuristic. Unlike the adjusted additive heuristic, it neglects the second term of equation 7. The Combo heuristic is calculated as follows:

$$h^{comb}(s, g) = h^+(s, g) + h^{lev}(s, g) \tag{10}$$

Empirical evaluations show that the combo heuristic is slightly faster than the adjusted additive heuristic.

Relaxed Planning Graph Heuristic. The relaxed planning graph heuristic (Hoffmann & Nebel, 2001) is based on the delete relaxation idea. The heuristic computation starts by constructing a relaxed variant of the planning graph (see § 5.4.1) in which the negative effects of the actions and the mutex relations are ignored. Then, a backward procedure is applied to extract a relaxed plan from the first level k of the planning graph containing the set of goal propositions. During the extraction, two sequences are maintained: a sequence of goal propositions $g_1, \ldots, g_k$ achieved at each propositional level and a sequence of set of actions $A_0, \ldots, A_{k-1}$ that represents the actions chosen to derive these goal propositions $g_1, \ldots, g_k$. The sequence $A_0, \ldots, A_{k-1}$ is the relaxed plan. Formally, the relaxed planning graph heuristic can be defined as follows:

$$h^r(s, g) = \Sigma_{i=0,\ldots,k-1} |A_i| \tag{11}$$

The relaxed planning graph heuristic returns the number of actions of the relaxed plan as an estimation of the goal achievement if such a plan exists, or ∞ otherwise. Note that building a relaxed planning graph and extracting a relaxed plan are polynomial time operations. However, the relaxed planning graph heuristic is not admissible.

5.6. *Planners*

Several state-of-the-art planners are implemented in PDDL4J. The objective of the toolkit is not to be exhaustive, but rather (1) to propose for pedagogical purposes an implementa-

tion of some canonical planners, and (2) to allow researchers to build on existing planners. We present in this section the implemented planners in three categories: state-space planning, graph planning, and SAT planning. Other planning techniques exists, for instance plan-space search (Penberthy & Weld, 1992), Model checking techniques (Triantafillou, Baier, & McIlraith, 2015) or Markov Decision Process (Mausam & Kolobov, 2012). But, for now, no planner based on these techniques is implemented in PDDL4J.

5.6.1. State-Space Planners

The planners in this category rely on search algorithms whose search space is a subset of the state space (see Section 3.1). Each node corresponds to a state of the world, each arc represents a state transition, i.e., an action, and a plan is a path from the initial state to a goal state in the search space.

Heuristic Search Planner (HSP) is an extended version of the planner proposed by B. Bonnet (Bonet & Geffner, 2001). This planner includes several basic search algorithms: best-first search, bread-first search, A* and iterative deepening search. Each search algorithm, except bread-first search, can be coupled with the heuristic described in Section 5.5.

Fast Forward is a state-space planner based on the relaxed planning graph heuristic (Hoffmann & Nebel, 2001). The search procedure is based on a variation of the hill-climbing search called enforced hill-climbing. The idea of this search procedure is to perform an exhaustive search only for the best states. The search uses also a heuristic called "helpful actions" to reduce the branching factor: only the actions of the relaxed planning graph at the first level are used to compute the successor states. This heuristic with the enforced hill climbing search makes Fast Forward incomplete. To guarantee the completeness, Fast Forward launches an A* search without helpful actions when the enforced hill-climbing search fails. Fast Forward does not find optimal plans. However, in practice, the plans are relatively close to the optimum.

Fast Downward is a state-space search planner based on FDR and the causal graph heuristic (Helmert, 2006). The main search procedure of Fast Downward is a greedy best-first search procedure, which is a modified version of classical best-first with deferred heuristic evaluations to mitigate the negative influence of wide branching factors. It extends the helpful actions proposed by Fast Forward in the context of causal graphs. This planner is the foundation of several other planners, as for instance LAMA (Richter & Westphal, 2010) etc.

5.6.2. Planning Graph Planners

In contrast to the state-space planners, the planners in this category encode their search space in a structure called the planning graph (see §5.4.1). This planning encoding is compact as, at every level, states are fused with sets of propositions. Moreover, mutex relations denoting inconsistencies among propositions and actions are stored at each level of the graph. All the planners in this category start by extending the planning graph until a proposition level is built that contains all the goal propositions without any mutex. Then, the planner starts its search: if it fails, the planning graph is extended to one more level, and a new search is carried out. Graph extension ends when the planning graph reaches its fixed-point (the two last levels are equal). In this case, there

is no solution and the planner returns failure. The planners presented below differ mainly in the search procedure used to extract a solution plan from the planning graph:

Graphplan was the first planner that introduced a planning graph structure (Blum & Furst, 1997). The extraction of a solution plan from the planning graph is a backward search from the last proposition level containing the goal to the first proposition level, which is the initial state of the planning problem. For each goal proposition at the last level of the planning graph, the search procedure non-deterministically chooses a set of mutex free actions producing the goal propositions. If such a set exists, the new goal becomes the union of the preconditions of these actions. Otherwise, there is no solution, the search procedure backtracks and tries another set of mutex free actions. The search is exponential in time. More information about Graphplan implementation in PDDL4J is available in IPP planner papers (Koehler, 1999; Koehler et al., 1997).

SATPlan is a graph-based planner where the search is carried out by a SAT solver. The implementation in PDDL4J relies on the papers of H. Kautz and B. Selman (Kautz, Selman, & Hoffmann, 2006; Kautz & Selman, 1999). Once the planning graph is extended, it is encoded as a SAT problem. This SAT problem is then solved with the SAT4J (Le Berre & Parrain, 2010) library.

GP-CSP is a graph-based planner where the search is carried out by a CSP solver. The implementation in PDDL4J is based on the M. Do and S. Kambhampati paper (Do & Kambhampati, 2001). In this case, the planning graph is encoded as a Constraints Satisfaction Problem (CSP). The CSP problem is then solved with the Choco library (Prud'homme, Fages, & Lorca, 2014). Note that alternative approaches (Lopez & Bacchus, 2003; Barták & Toropila, 2008; Cooper, de Roquemaurel, & Régnier, 2011) based on different planning graph encodings exist.

5.6.3. Propositional Satisfiability Planners

The general idea underlying propositional satisfiability planners is to encode planning problems into SAT problems for which very efficient algorithms exist. In the last decade, performances of SAT planners have considerably increased due to parallel SAT solvers that can now exploit several CPU cores (Rintanen, Heljanko, & Niemelä, 2006). Likewise, important improvements in compact SAT encodings and heuristic (Rintanen, 2012) have been achieved. In contrast to encodings that have quadratic sizes (Kautz et al., 2006; Kautz & Selman, 1999) (e.g. the case of action exclusion axioms), it is now possible to carry out planning problem encodings in asymptotically optimal linear-size (Rintanen et al., 2006). These different encodings can be classified into three categories with respect to action parallelism allowed in the solution plan. In the first category of encoding (Kautz & Selman, 1992), only sequential actions are allowed. In the second category (Rintanen et al., 2006; Robinson, Gretton, Pham, & Sattar, 2009), a set of actions can be simultaneously executed if they are pairwise independent. The actions parallelism used in this category is similar to the one used in the planning graph structure. In the third category (Dimopoulos, Nebel, & Koehler, 1997; Wehrle & Rintanen, 2007), a set of actions can be simultaneously executed if they can be totally ordered so that any action does not disable any later action or change its effects. This kind of encoding is the fastest encoding currently used by the most efficient SAT planner known to date and called Madagascar (Rintanen, 2014). These three kind of encodings are implemented in PDDL4J in the planner called PSPlan (the solving is done with the SAT4J library).

6. Evaluation and Experimental Results

In this section, we propose a comparison of PDDL4J with the two other planning toolkits: Fast-Downward (FD) (Helmert, 2006) and Fast-Forward (FF) (Hoffmann & Nebel, 2001) (the Lightweight Automated Planning ToolKiT (Ramirez, Lipovetzky, & Muise, 2015) is based on FD and FF). For each toolkit, we evaluate the comparable modules: *the parsing module* and *the encoding module*. These modules are written respectively in Java for PDDL4J, in Python for FD and in C for FF. The exhautive comparison was made on all the 29 STRIPS planning domains of the successive International Planning Competitions. The experiments were carried out on a 6-core Intel Xeon clocked at 2.2 GHz. For each experiment a memory of 16 GBytes was allocated.

The performances of the parsing modules are shown Figure 9. For every domain, Figure 9 shows the average time in seconds needed by PDD4J, FF and FD to parse all the domains and problems files. FF has the most performant parsing module. Then there is PDDL4J and finally FD. The difference of performance between, on the one hand, FD and, on the other hand, FF and PDDL4J, is significant for domains like `openstracks` or `optical-telegraph` where problem files are very large (more than 10 MBytes). The implementation of the parsing module of FD written in Python is definitely less performant than the implementation of PDDL4L based on JavaCC, and the implementation of FF based on Flex and Bison. However, whatever the parsing implementation, the average time for parsing remains relatively small and comparable (between 0.1 to 8 seconds).

The performances of the encoding modules are evaluated by : (1) the average time needed to encode all the planning problems of a domain (see Figure 10) and (2) the average memory needed to store all the encoded planning problems of a domain (see Figure 11). The time needed to encode planning problems remains relatively small (between 1 to 60 seconds for most of the domains). The encoding module of FD is much slower than PDDL4L and FF in this order. FD encoding module takes more time for the large problems (those identified in the evaluation of the parsing modules). However, FD outperforms PDDL4J and FF in two domains: `logistics` and `thoughtful`. These two domains have the particularity of having planning operators with a lot of parameters. For instance, the `thoughtful` domain has operators with more than 6 parameters. If we go into detail in the implementation of the encoding modules of FF and PDDL4J, we can notice that the computation of the inertias requires the creation of several multi-dimensions arrays whose size is exponential with the number of operator parameters. This explains why FF and PDDL4J, which implement inertia computation, are less performant to encode the `logistics` and `thoughtful` problems. Concerning the average memory used to store the encoded planning problems (see Figure 11), PDDL4J is broadly less memory-hungry than the other toolkits except for `logistics` and `thoughtful` for the reasons above.

To summarize, the experiments show that PDDL4J parsing and encoding modules are competitive with the two main research planning toolkits FF and FD, even if PDDL4J is written in Java. Moreover, PDD4J is more mature than research code. It has been completely refectored by professional developers and its development process follows software engineering standards: there is a source code repository available at http://pddl4j.imag.fr. Continuous inspection of code quality is managed by the SonarQube platform (see http://pddl4j.imag.fr/sonar). PDDL4J source code is implemented according "Continous Integration" (CI) principles (through Jenkins CI server, see http://pddl4j.imag.fr/jenkins): CI consists in verifying that each code modification does not produce code regression with respect to automated unit and integration tests. This is meant to avoid and/or efficiently identify bugs in code upgrades which could im-

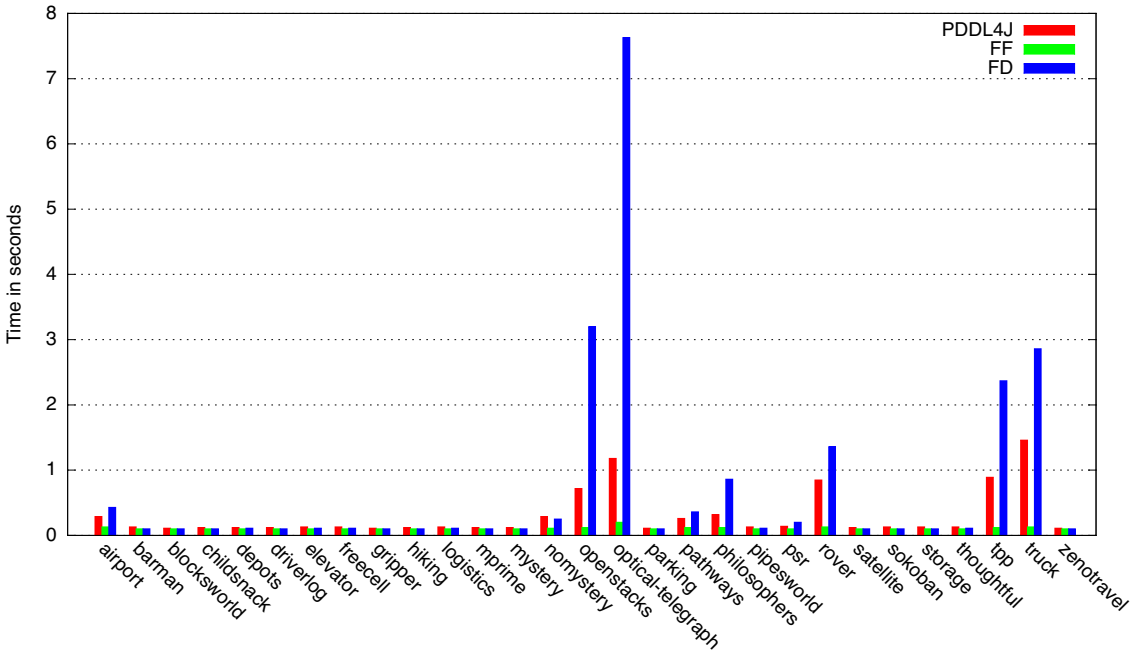

Figure 9. Average time for domain and problem parsing

pair PDDL4J functionalities and performances. We would like to highlight that PDDL4J comes with an exhaustive code documentation making PDDL4J a relevant Java toolkit for both industrial and research applications. Note that PDDL4J is released under LGPL license.

Conclusion

PDDL4J is a mature open-source toolkit providing access to a large number of planning-related technologies to software developers. Our objective is to facilitate developments of new planners and techniques in the planning community but also to disseminate planning techniques to other communities. PDDL4J provides state-of-the-art tools for the PDDL language, a Java API to design new algorithms and the most important heuristics developed for automated planning.

The future developments and extensions of PDDL4J will focus on the following points:

(1) Add new state-of-art planners and heuristics developed in the planning research community in order to integrate the lastest advances and keep the PDDL4J toolkit competitive.
(2) Develop the interfaces of PDDL4J to ease its integration and use in different contexts of applications. In particular, we will tackle the following implementions:
   (a) Simplified interfaces with other solvers based on SAT and CSP techniques such as SAT4J and CHOCO to bridge the gap between these communities, and
   (b) an interface to the *Robot Operating System* (Quigley, Gerkey, & Smart, 2015) (ROS) to facilitate PDDL4J usage in the robotics community. ROS is an open source framework aiming at writing softwares for robots. It is a collection

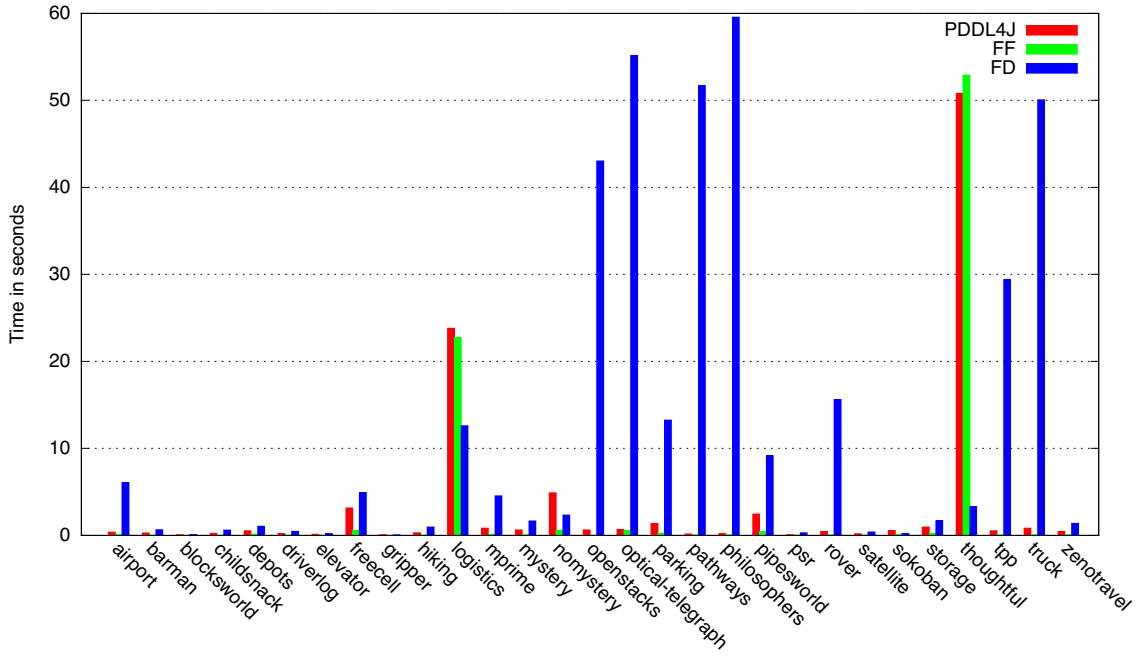

Figure 10. Average time for planning problem encoding

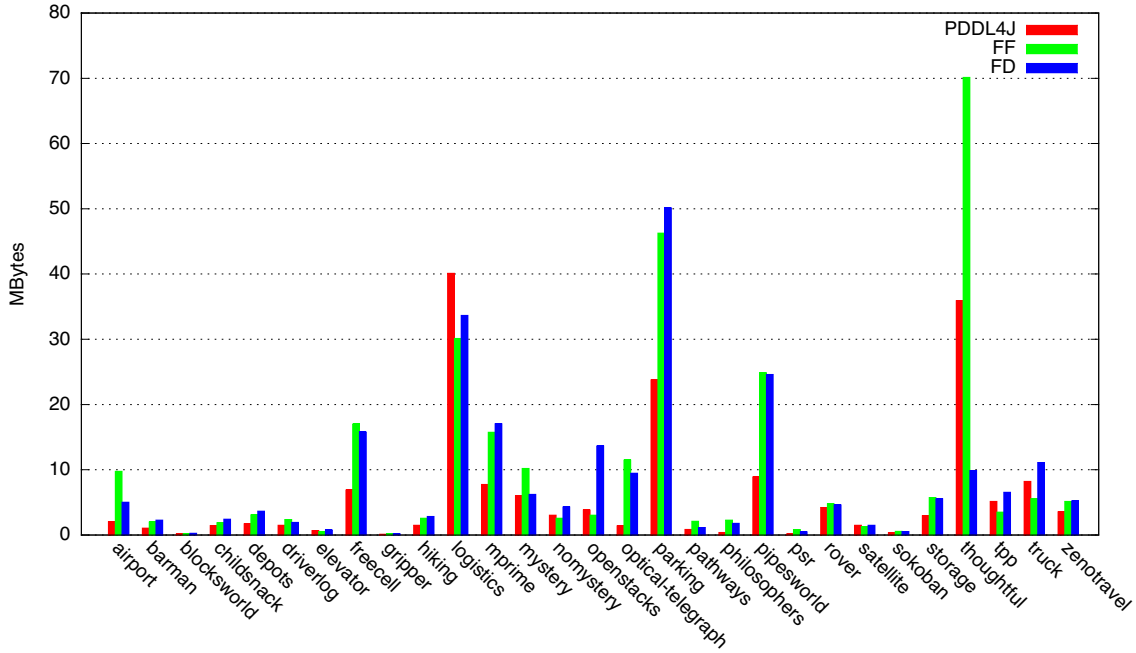

Figure 11. Average memory usage of the encodings of the planning problems

of tools, libraries, and conventions that help to create complex and robust robot behaviors on a wide variety of robotic platforms. It is currently used by hundreds of research groups and companies in the robotics industry. A

preliminary version of this interface is already available as an open source project[4].

References

Ambite, J. L., & Kapoor, D. (2007). Automatically Composing Data Workflows with Relational Descriptions and Shim Services. In *Proceedings of the International Semantic Web Conference* (pp. 15–29).

Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., & Marchetti-Spaccamela, A. (1999). *Complexity and Approximation.* Springer-Verlag.

Backes, P. G., Norris, J. S., Powell, M. W., & Vona, M. A. (2004). Multi-mission Activity Planning for Mars Lander and Rover Missions. In *Proceedings of the IEEE Aerospace Conference* (pp. 877–886).

Bäckström, C., & Nebel, B. (1995). Complexity Results for SAS+ Planning. *Computational Intelligence*, *11*, 625–656.

Barták, R., Salido, M., & Rossi, F. (2010). Constraint satisfaction techniques in planning and scheduling. *Journal Intelligent Manufacturing*, *21*(1), 5–15.

Barták, R., & Toropila, D. (2008). Reformulating constraint models for classical planning. In *Proceedings of The International Florida Artificial Intelligence Research Society Conference* (pp. 525–530).

Bernardini, S., & Smith, D. (n.d.).
In *Proceedings of the icaps workshop on heuristics for domain-independent planning.*

Bevilacqua, L., Furno, A., Scotto di Carlo, V., & Zimeo, E. (2011). A tool for automatic generation of WS-BPEL compositions from OWL-S described services. In *Proceedings of the International Conference on Software, Knowledge Information, Industrial Management and Applications.*

Blum, A., & Furst, M. (1997). Fast Planning Through Planning Graph Analysis. *Artificial Intelligence*, *90*(1-2), 281–300.

Bonet, B., & Geffner, H. (2001). Planning as Heuristic Search. *Artificial Intelligence*, *129*(1–2), 5–33.

Brafman, R., & Domshlak, C. (2003). Structure and Complexity in Planning with Unary Operators. *Journal of Artificial Intelligence Research*, *18*(1), 315–349.

Brafman, R., & Domshlak, C. (2006). Factored Planning: How, When, and When Not. In *Proceedings of the Association for the Advancement of Artificial Intelligence* (pp. 809–814).

Brenner, M. (2003). A Multiagent Planning Language. In *Proceedings of the ICAPS Workshop on Planning Domain Description Language.*

Bresina, J. L., Jónsson, A. K., Morris, P. H., & Rajan, K. (2005). Activity Planning for the Mars Exploration Rovers. In *Proceedings of the International Conference on Automated Planning and Scheduling* (pp. 40–49).

Bryce, D., & Kambhampati, S. (2007). A Tutorial on Planning Graph Based Reachability Heuristics. *Artificial Intelligence Magazine*, *27*(1), 47–83.

Butler, R., & Muñoz, C. (2006). *An Abstract Plan Preparation Language* (Tech. Rep. No. TM-2006-214518). NASA.

Cashmore, M., Fox, M., Long, D., Magazzeni, D., Ridder, B., De Carolis, V., ... Maurelli, F. (2015). Dynamically Extending Planning Models using an Ontology. In *Proceedings of the ICAPS Workshop on Planning and Robotics* (pp. 79–85).

Chen, H., & Giménez, O. (n.d.). Causal graphs and structurally restricted planning. *Journal of Computer System and Science*(10), 273–314.

Chen, Y., Zhao, X., & Zhang, W. (2007). Long-distance mutual exclusion for propositional

[4]https://github.com/pellierd/pddl4j-rospy

planning. In *Proceedings of the International Joint Conference on Artificial Intelligence* (pp. 1840–1845).

Cheng, C.-H., Knoll, A., Luttenberger, M., & Buckl, C. (2011). GAVS+: An Open Platform for the Research of Algorithmic Game Solving. In *Proceedings of the Joint European Conferences on Theory and Practice of Software* (pp. 258–261).

Cooper, M., de Roquemaurel, M., & Régnier, P. (2011). A weighted CSP approach to cost-optimal planning. *AI communication*, *24*(1), 1–29.

Dimopoulos, Y., Nebel, B., & Koehler, J. (1997). Encoding planning problems in nonmonotonic logic programs. In *Proceedings of the European Conference on Planning* (pp. 169–181).

Di Rocco, M., Pecora, F., & Saffiotti, A. (2013). Closed Loop Configuration Planning with Time and Resources. In *Proceedings of the ICAPS Workshop on Planning and Robotics* (pp. 36–44).

Do, M., & Kambhampati, S. (2001). Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence*, *132*(2), 151–182.

Domshlak, C., Hoffmann, J., & Katz, M. (2015). Red-black planning: A new systematic approach to partial delete relaxation. *Artificial Intelligence*, *221*, 73-114.

Dongning, R., Zhihua, J., Yunfei, J., & Kangheng, W. (2010). Learning Non-Deterministic Action Models for Web Services from WSBPEL Programs. *Journal of Computer Research and Development*, *47*(3), 445–454.

Dvořák, F., Bit-Monnot, A., Ingrand, F., & Ghallab, M. (2014). A Flexible ANML Actor and Planner in Robotics. In *Proceedings of the ICAPS Workshop on Planning and Robotics* (pp. 12–19).

Ededlkamp, S. (2001). Planning with Pattern Databases. In *Proceedings of the European conference on Planning* (pp. 13–24).

Edelkamp, S., & Hoffmann, J. (2004). *PDDL2.2: The Language for the Classical Part of the 4th International planning Competition* (Tech. Rep. No. 195). Institut für Informatik.

Fernández, S., Adarve, R., Pérez, M., Rybarczyk, M., & Borrajo, D. (2006). Planning for an AI Based Virtual Agents Game. In *Proceedings of the ICAPS Workshop on AI Planning for Synthetic Characters and Computer Games.*

Fernandez-Gonzalez, E., Karpas, E., & Williams, B. C. (2015). Mixed Discrete-Continuous Heuristic Generative Planning based on Flow Tubes. In *Proceedings of the ICAPS Workshop on Planning and Robotics* (pp. 106–115).

Ferrer-Mestres, J., Francès, G., & Geffner, H. (2015). Planning with State Constraints and its Applications to Combined Task and Motion Planning. In *Proceedings of the ICAPS Workshop on Planning and Robotics* (pp. 13–22).

Fikes, R., & Nilsson, N. J. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, *2*(3/4), 189–208.

Fox, M., & Long, D. (2002). PDDL+ : Modelling Continuous Time-dependent Effects. In *Proceedings of the International NASA Workshop on Planning and Scheduling.*

Fox, M., & Long, D. (2003). PDDL 2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, *20*(1), 61–124.

Fox, M., & Long, D. (2006). Modelling Mixed Discrete-Continuous Domains for Planning. *Journal of Artificial Intelligence Research*, *27*, 235–297.

Frank, J., & Jónsson, A. (2003). Constraint-Based Attribute and Interval Planning. *Constraints*, *8*(4), 339–364.

Garrett, C. R., Lozano-Pérez, T., & Kaelbling, L. P. (n.d.). Heuristic Search for Task and Motion Planning. In *Proceedings of the icaps workshop on planning and robotics.*

Gerevini, A., & Long, D. (2005a). *BNF Description of PDDL3.0.* (Unpublished manuscript from the International Planning Competition website)

Gerevini, A., & Long, D. (2005b). *Plan Constraints and Preferences in PDDL3* (Tech. Rep. No. 2005-08-47). Dipartimento di Elettronica per l'Automazione, Università degli Studi di Brescia.

Gerevini, A., & Long, D. (2005c). Preferences and Soft Constraints in PDDL3. In *Proceedings of the ICAPS Workshop on Preferences and Soft Constraints in Planning* (pp. 46–54).

Ghallab, M., Howe, A., Knoblock, G., McDermott, D., Ram, A., Veloso, M., ... Wilkins, D. (1998). PDDL: The Planning Domain Definition Language [Computer software manual].

Haslum, P., Bonet, B., & Geffner, H. (2005). New Admissible Heuristics for Domain-Independent Planning. In *Proceedings of the Association for the Advancement of Artificial Intelligence* (pp. 1163–1168).

Haslum, P., Botea, A., Helmert, M., Bonet, B., & Koenig, S. (2007). Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning. In *Proceedings of the association for the advancement of artificial intelligence* (pp. 1007–1012).

Haslum, P., & Geffner, H. (2000). Admissible Heuristics for Optimal Planning. In *Proceedings of the Artificial Intelligence Planning Systems* (pp. 140–149).

Helmert, M. (n.d., 2008). *Changes in PDDL 3.1.* (Unpublished manuscript International Planning Competition website)

Helmert, M. (2006). The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, *26*, 191–246.

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

Helmert, M., & Domshlak, C. (2009). Landmarks, Critical Paths and Abstractions: What is the difference anyway. In *Proceedings of the International Conference on Planning and Scheduling* (p. 162-171).

Helmert, M., Haslum, P., Hoffmann, J., & Nissim, R. (2014). Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *Journal of the ACM*, *61*(3), 16–63.

Hoffmann, J. (2011). Analyzing search topology without running any search: On the connection between causal graphs and h+. *Journal of Artificial Intelligence Research*, *41*, 155–229.

Hoffmann, J., & Nebel, B. (2001). The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, *14*(1), 253–302.

Hoffmann, J., Porteous, J., & Sebastia, L. (2004). Ordered Landmarks in Planning. *Journal of Artificial Intelligence Research*, *22*(1).

Hoffmann, J., Weber, I., & Kraft, F. (2009). Planning@SAP: An Application in Business Process Management. In *Proceedings of the ICAPS Workshop on Scheduling and Planning Applications.*

Howey, R., Long, D., & Fox, M. (2004). VAL: automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *Proceedings of the IEEE international conference on tools with artificial intelligence* (pp. 294–301).

Jonsson, A. (2007). The Role of Macros in Tractable Planning Over Causal Graphs. In *Proceedings of the International Joint Conference on Artificial Intelligence* (p. 1936-1941).

Jonsson, P., & Bäckström, C. (1998). Tractable plan existence does not imply tractable plan generation. *Annals of Mathematics and Artificial Intelligence*, *22*(3–4), 281–296.

Kambhampati, S. (2000). Planning Graph as a (Dynamic) CSP: Exploiting EBL, DDB and other CSP Search Techniques in Graphplan. *Journal of Artificial Intelligence Research*, *12*(1), 1–34.

Kambhampati, S. (2001). Planning as constraint Satisfaction: Solving the planning graph by compiling into CSP. *Artificial Intelligence*, *132*(2), 151–182.

Kambhampati, S., Parker, E., & Lambrecht, E. (1997). Understanding and Extending Graphplan. In *Proceedings of the European conference on Planning* (p. 260-272).

Katz, M., & Domshlak, C. (2008). Structural Patterns Heuristics via Fork Decomposition. In *Proceedings of the International Conference on Automated Planning and Scheduling* (pp. 182–189).

Kautz, H., & Selman, B. (1992). Planning as Satisfiability. In *Proceedings of the European Conference on Artificial Intelligence* (pp. 359–363).

Kautz, H., & Selman, B. (1999). Unifying SAT-based and Graph-based Planning. In *Proceedings of International Joint Conference on Artificial Intelligence* (pp. 318–325).

Kautz, H., Selman, B., & Hoffmann, J. (2006). Satplan: Planning as satisfiability. In *Abstracts of the 5th international planning competition.*

Knoblock, C. (1994). Automatically generating abstractions for planning. *Artificial intelligence*, *68*(2), 243–301.

Koehler, J. (1999). *Handling of conditional effects and negative goals in IPP* (Tech. Rep.). Freiburg University.

Koehler, J., & Hoffmann, J. (1999). *Handling of Inertia in a Planning System* (Tech. Rep. No. 122). Albert Ludwigs University.

Koehler, J., Nebel, B., Hoffmann, J., & Dimopoulos, Y. (1997). Extending planning graphs to an ADL subset. In *Proceedings of the European Conference on Planning* (pp. 273–285).

Kovacs, D. (n.d.-a, 2011). *BNF Definition of PDDL3.1: completely corrected, without comments.* (Unpublished manuscript from the International Planning Competition website)

Kovacs, D. (n.d.-b, 2012). *Complete BNF description of PDDL 3.1* (Tech. Rep.). (Unpublished manuscript from the International Planning Competition website)

Lagriffoul, F. (2014). Delegating Geometric Reasoning to the Task Planner. In *Proceedings of the ICAPS Workshop on Planning and Robotics* (pp. 54–59).

Lallement, R., de Silva, L., & Alami, R. (n.d.). HATP: An HTN Planner for Robotics. In *Proceedings of icaps workshop on planning and robotics.*

Le Berre, D., & Parrain, A. (2010). The SAT4J library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, *7*(59–64).

Liang, R. (2012). A Survey of Heuristics for Domain-Independent Planning. *Journal of Software*, *7*(9), 2099–2016.

Liu, D., & McCluskey, T. L. (2001). The ocl language manual [Computer software manual].

Long, D., & Fox, M. (1999). Efficient Implementation of the plan graph in STAN. *Journal of Artificial Intelligence Research*, *10*(87–115).

Lopez, A., & Bacchus, F. (2003). Generalizing GraphPlan by Formulating Planning as a CSP. In *Proceedings of the International Conference on Artificial Intelligence* (pp. 954–960).

Mausam, & Kolobov, A. (2012). *Planning with Markov Decision Processes: An AI Perspective.* Morgan Kaufmann & Claypool publishers.

McDermott, D. (n.d., 2005). OPT Manual Version 1.7.3 [Computer software manual].

Nguyen, X., Kambhampati, S., & Nigenda, R. (2002). Planning graph as the basis for deriving heuristics for plan synthesis by state space and CSP search. *Artificial Intelligence*, *135*(1-2), 73-123.

Pednault, E. (1994). ADL and the State-Transition Model of Action. *Journal of Logic and Computation*, *4*(5), 467–512.

Penberthy, J., & Weld, D. (1992). UCPO: A sound, complete, partial order planner for ADL. In C. R. B. Nebel & W. Swartout (Eds.), *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning* (pp. 103–114).

Prud'homme, C., Fages, J.-G., & Lorca, X. (2014). Choco3 Documentation [Computer software manual].

Quigley, M., Gerkey, B., & Smart, W. (2015). *Programming Robots with ROS.* O'Reilly.

Ramirez, M., Lipovetzky, N., & Muise, C. (2015). *Lightweight Automated Planning ToolKiT.* `http://lapkt.org/`.

Richter, S., Helmet, M., & Westphal, M. (2008). Landmarks Revisited. In *Proceedings of the Association for the Advancement of Artificial Intelligence* (pp. 975–982).

Richter, S., & Westphal, M. (2010). The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research*, *39*, 127–177.

Rintanen, J. (2012). Planning as satisfiability: Heuristics. *Artificial Intelligence*, *193*, 45–86.

Rintanen, J. (2014). Madagascar: Scalable Planning with SAT. In *Proceedings of the International Competition of Planning.*

Rintanen, J., Heljanko, K., & Niemelä, I. (2006). Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, *170*(12–13), 1031–1080.

Robinson, N., Gretton, C., Pham, D.-N., & Sattar, A. (2009). SAT-based parallel planning using a split representation of actions. In *Proceedings of the International on Automated Planning and Scheduling* (pp. 281–288).

Salnitri, M., Paja, E., & Giorgini, P. (2015). *Maintaining Secure Business Processes in Light of Socio-Technical System's Evaluation* (Tech. Rep.). DISI-University of Trento.

Sanner, S. (n.d., 2010). *Relational Dynamic Influence Diagram Language (RDDL): Language Description.* (Unpublished manuscript from the International Planning Competition website)

Srivastava, S., Riano, L., Russell, S., & Abbeel, P. (2013). Using Classical Planners for Tasks with Continuous Operators in Robotics. In *Proceedings of the ICAPS Workshop on Planning and Robotics* (pp. 27–35).

Thomas, J. M., & Young, M. R. (2006). Author in the Loop: Using Mixed-Initiative Planning to Improve Interactive Narrative. In *Proceedings of the ICAPS Workshop on AI Planning for Synthetic Characters and Computer Games.*

Triantafillou, E., Baier, J., & McIlraith, S. (2015). A Unifying Framework for Planning with LTL and Regular Expressions. In *Proceedings of the ICAPS Workshop on Model-Checking and Automated Planning* (p. 23-31).

van Beek, P., & Chen, X. (1999). CPlan: A constraint programming approach to planning. In *Proceedings of the Association for the Advancement of Artificial Intelligence* (pp. 585–590).

Wehrle, M., & Rintanen, J. (2007). Planning as satisfiability with relaxed ∃-step plans. In *Proceedings of the Australian Joint Conference on Artificial Intelligence* (pp. 244–253).

Williams, B., & Nayak, P. (1997). A reactive planner for a model-based executive. In *Proceedings of the International Joint Conference on Artificial Intelligence* (pp. 150–159).

Younes, H., & Littman, M. (2004). *PPDDL 1.0: an extension to PDDL for expressing planning domains with probabilistic effects* (Tech. Rep. No. CMU-CS-04-167). Carnegie Mellon University.

Zaman, S., Steinbauer, G., Maurer, J., Lepej, P., & Uran, S. (2013). An integrated model-based diagnosis and repair architecture for ROS-based robot systems. In *Proceedings of the International Conference on Robotics and Automation* (pp. 482–489).