PDDL4J

Release 4.0

Damien Pellier

Dec 08, 2021

CONTENTS

PDDL4J is a software library under LGPL license embedding Artificial Intelligence algorithms to find solutions for planning problems, that is to say time organized actions to achieve a goal. Solutions to planning problems are "todo lists" named **plan** representing operational features of actions like **who, how, where, when and what to do**. The code of the library is available on GitHub.

PDDL4J is a suite of solvers useful for decision problems that can be solved by a sequence of actions (plan). It is based on a declarative approach: the user **states** a decision problem in PDDL (Planning Domain Description language and PDDL4J generates possible solutions. **No programming language and/or background** is required. PDDL4J has lot of application fields like can be use in many industrial fields as smart homes/data/cities, autonomous systems and robotics, logistics, business processes etc.

Note: If you are not familiar with Automated Planning, start reading the chapter *Automated Planning in a Nutshell*. If you are not familiar with PDDL, start reading *PDDL Tutorial*.

PDDL was originally developed in 1998. It was inspired by the need to encourage the empirical comparison of planning systems and the exchange of planning benchmarks within the community. Its development improved the communication of research results and triggered an explosion in performance, expressivity and robustness of planning systems. PDDL has become a de facto standard language for describing planning domains, not only for planning competition but more widely, as it offers an opportunity to carry out empirical evaluation of planning systems on a growing collection of generally adopted standard benchmark domains. The emergence of a language standard will have an impact on the entire field, influencing what is seen as central and what peripheral in the development of planning systems.

The library contains:

- A PDDL 3.1 parser and all the classes need to manipulate its concepts. The parser can be configured to accept only specified requirements of PDDL language.
- A set of useful pre-processing mechanisms in order to instantiate and simply operators into ground actions.
- A set of already implemented classical heuristics, e.g., relaxed planning graph, critical path, etc.
- Several examples of planners using PDDL4J.

Note: To get PDDL4J and start running a planner start reading the chapter *Getting Started* or exploring the PDDL4J API documentation *API Documentation*.

Important: The library is open source. If you use it, please cite us:

D. Pellier & H. Fiorino (2017) PDDL4J: a planning domain description library for java, Journal of Experimental & Theoretical Artificial Intelligence, 30:1, 143-176, DOI: 10.1080/0952813X.2017.1409278

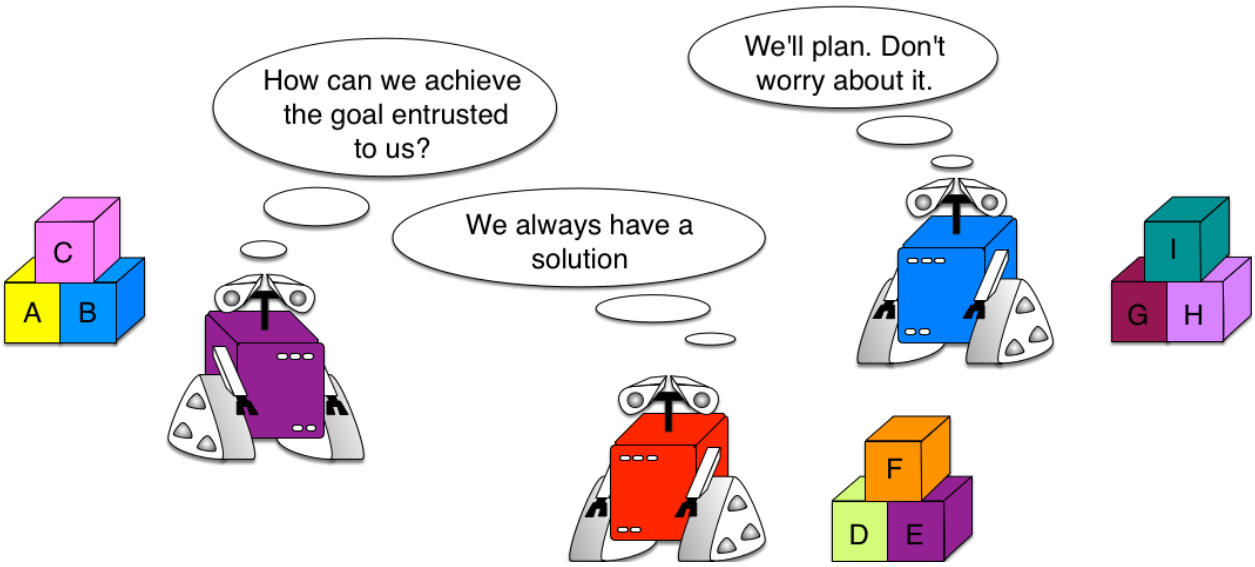
CHAPTER

ONE

AUTOMATED PLANNING IN A NUTSHELL

1.1 What is planning?

"Planning is a discipline of Artificial Intelligence that aims at the development of generic algorithms allowing **autonomous systems** to choose and organize their actions to achieve a goal by anticipating their effects (Ghallab, M. and al., 2004)"



1.2 A classical example: robot dockers

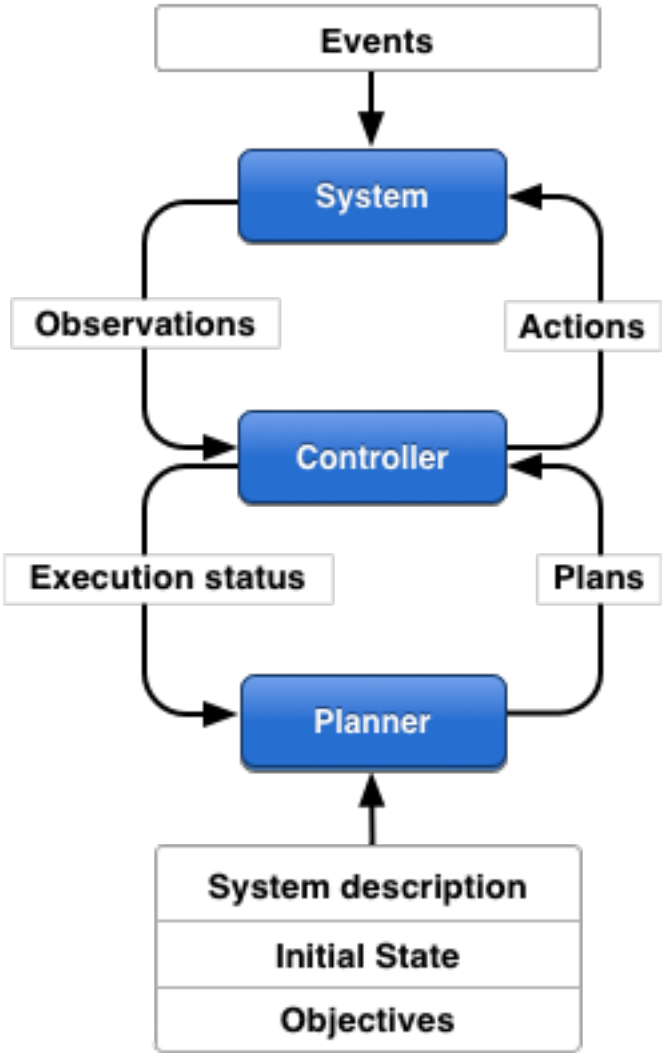
Imagine a system where robots have to transport containers and store them. In this system, every event and action could be predefined and hard-coded but this approach has many shortcomings: complexity, combinatory explosion of the possible scenarios, vulnerability to failures and unexpected events etc.

Another approach is to bet on autonomy technologies such as AI planning and let the robots make their own decisions according to their perceptions and the state of the world.

In the situation above, the goal is to stack container c2 on container c1 at location p2. Automated planning allows to compute a sequence of actions (plan) that will allow the autonomous robots r1, crane1, crane2 to achieve this task from the initial state (c1 is on c2 at p1).

Important: What is important to understand here is that if the initial state were different (due to an unexpected event or failure), the system would be able to compute another plan to fulfill the task.

1.3 The planning model



The conceptual model of planning is made of three components:

- **A state transition system** that models the evolution of the environment,
- **A controller** that chooses the next action to execute and controls its execution,
- **A planner** that, from the specification of a problem in PDDL, synthesizes a plan, i.e., a set of ordered actions, to achieve a goal.

1.4 Model Assumptions

Often, planning models do many simplifying assumptions about the real world:

- The environment is finished
- The environment is completely observable
- The environment is deterministics
- The solution plans are sequential
- The notion of time is implicit, e.g., actions have no duration
- Planning does not interleave action execution
- Planning is a centralized procedure etc.

Simplifying assumptions allow to avoid the combinatorial explosion of the search procedure.

1.5 A set of techniques

Classical representations of planning problems are based on logical representations, e.g., PDDL, OPL, etc. or state variables which are often more compact.

Planning algorithms can be classified in different approaches:

- State space planning
- Plan space planning
- SAT and CSP techniques
- Hierarchical planning techniques (HTN)
- Planning Graph techniques
- Markov Decision Process techniques
- Model Checking techniques
- etc.

CHAPTER

TWO

PDDL TUTORIAL

2.1 What is PDDL?

PDDL (Planning Domain Description Language) is a standard encoding language for "classical" planning.

The components of PDDL files are:

- *Requirements:* defining levels of abstraction in the language, e.g., "STRIPS", temporal, probabilistic effects etc.
- *Types:* sets of the things of interest in the world,
- *Objects:* instances of types,
- *Predicates:* Facts about objects that can be true or false,
- *Initial state* of the world: before starting the planning process,
- *Goal:* properties of the world true in goal states and achieved after the planning process,
- *Actions/Operators:* ways of changing states of the world and going from the initial state to goal states.

A planning task in PDDL is specified in two text files:

1. A *domain file* for requirements, types, predicates and actions,
2. A *problem file* for objects, initial state and goal specification.

Domain files are as follows:

```
(define (domain <domain name>)
    <PDDL code for requirements>
    <PDDL code for types>
    <PDDL code for predicates>
    <PDDL code for first action>
    [...]
    <PDDL code for last action>
)
```

where `<domain name>` is a string that identifies the planning domain.

Note: Many examples are available in `src/test/resources/benchmarks/pddl`. Domains and problems are classified by competition year and by track

---

Problem files are as follows:

```
(define (problem <problem name>)
    (:domain <domain name>)
    <PDDL code for objects>
    <PDDL code for initial state>
    <PDDL code for goal specification>
)
```

where:

- <problem name> is the string that identifies the planning task, e.g. gripper with 4 balls to move.
- <domain name> is the planning domain name corresponding to problem file.

Note: Novices in PDDL should start by doing the *A Simple Running Example: Logistics*. Advanced readers can go directly to the *An Industrial Use Case*.

2.2 A Simple Running Example: Logistics

We will use the Logistics domain to illustrate how to represent a planning task in PDDL.

In logistics, there are trucks and airplanes that can move packages between different airports and cities. We assume that in the initial state there is a truck in Paris airport. An airplane and two packages are in London airport. Paris has two places : south and north. The goal is to have one package in the north location and the other one in the south location.

Todo: First, create two text files respectively called `logistics.pddl` and `problem.pddl` and copy paste the PDDL code fragment given step by step in theses files. You can choose whatever text editor you want

---

Note: Remember that PDDL requires two files: 1. A *domain file* for requirements, types, predicates and actions, 2. A *problem file* for objects, initial state and goal specification.

2.2.1 Defining the Domain

Let start by defining in the file `logistics.pddl` the domain and its components :

- the requirements,
- the types,
- the predicates,
- the actions or the operators.

First, we have to define the name of the domain:

In PDDL, we write :

```
(define (domain logistics)
```

Requirements

The requirements for this logistics example are:

- **strips** : the actions will only use positive preconditions (predicates that must be true in the current state to trigger actions) and deterministic effects (effects that necessarily follow action triggering). Nothing else is allowed.
- **typing** : we will use "types" like in OO programming to represent sets of objects in the world.

In PDDL, we write :

```
(:requirements :strips :typing)
```

Types

We will use the following types:

- Places, cities and physical objects are considered as objects,
- Packages and vehicles are physical objects,
- Trucks and airplanes are vehicles,
- Airports and locations are places.

In PDDL, we write:

```
(:types city place physobj - object
      package vehicle - physobj
      truck airplane - vehicle
      airport location - place
)
```

Predicates

We will use the following predicates:

- *in-city(loc, city)* - true iff a place *loc* is in the city *city*
- *at(obj, loc)* - true iff a physical object *obj* is at place *loc*
- *in(pkg, veh)* - true iff the a package *pkg* is in a vehicle *veh*

In PDDL, question marks are used for variables:

```
(:predicates (in-city ?loc - place ?city - city)
    (at ?obj - physobj ?loc - place)
    (in ?pkg - package ?veh - vehicle)
)
```

Operators

We are going to define the operators of the actions of the logistics domain, i.e., the means to change the states of the world. The domains has 5 operators: *load-truck*, *load-airplaine*, *unload-truck*, *unload-airplane*, *drive-truck* and *fly-aiplane*.

Note: In this tutorial, we will use indifferently the words "action" and "operator" (though in planning community, actions are ground operators, i.e., operator where variables are replaced by constants).

Load Truck Operator

For instance, in the logistics domain, a truck can be loaded… And to load a truck, we need a package *pkg* and a truck *truck* at a place *loc*. To load *pkg* in *truck*, these two objects must be at the same place *loc*. The effects of loading *pkg* in *truck* are that *in(pkg, truck)* becomes true and *at(pkg, loc)* becomes false. Any other fact in the current state does not change:

```
(:action load-truck
    :parameters (?pkg - package ?truck - truck ?loc - place)
    :precondition (and (at ?truck ?loc) (at ?pkg ?loc))
    :effect (and (not (at ?pkg ?loc)) (in ?pkg ?truck))
)
```

Load Airplane Operator

Action/Operator :

- **Description** : Load a package *pkg* in an airplane *airplane* at a place *loc*,
- **Precondition** : *at(pkg, loc)* and *at(airplane, loc)* must be true,
- **Effect** : *in(pkg, airplane)* becomes true and *at(airplane, loc)* becomes false.

In PDDL:

```
(:action load-airplane
    :parameters (?pkg - package ?airplane - airplane ?loc - place)
    :precondition (and (at ?pkg ?loc) (at ?airplane ?loc))
    :effect (and (not (at ?pkg ?loc)) (in ?pkg ?airplane))
)
```

Unload Truck Operator

Action/Operator :

- **Description** : Unload a package *pkg* in a truck *truck* at a place *loc*,
- **Precondition** : *in(pkg, truck)* and *at(truc, loc)* must be true,
- **Effect** : *at(pkg, loc)* becomes true and *in(pkg, truck)* becomes false.

In PDDL:

```
(:action unload-truck
    :parameters (?pkg - package ?truck - truck ?loc - place)
    :precondition (and (at ?truck ?loc) (in ?pkg ?truck))
    :effect (and (not (in ?pkg ?truck)) (at ?pkg ?loc))
)
```

Unload Airplane Operator

Action/Operator :

- **Description** : Unload a package *pkg* in an airplane *airplane* at a place *loc*,
- **Precondition** : *in(pkg, airplane)* and *at(airplane, loc)* must be true,
- **Effect** : *at(pkg, loc)* becomes true and *in(pkg, airplane)* becomes false.

In PDDL:

```
(:action unload-airplane
    :parameters (?pkg - package ?airplane - airplane ?loc - place)
    :precondition (and (in ?pkg ?airplane) (at ?airplane ?loc))
    :effect (and (not (in ?pkg ?airplane)) (at ?pkg ?loc))
)
```

Fly-airplane Operator

Action/Operator :

- **Description** : Fly airplane *pkg* from a location *loc-from* to a location *loc-to*,
- **Precondition** : *at(pkg, loc-from)* must be true,
- **Effect** : *at(pkg, loc-to)* becomes true and *at(p, loc-from)* becomes false.

In PDDL:

```
(:action fly-airplane
    :parameters (?airplane - airplane ?loc-from - airport ?loc-to - airport)
    :precondition (at ?airplane ?loc-from)
    :effect (and (not (at ?airplane ?loc-from)) (at ?airplane ?loc-to))
)
```

Drive-truck Operator

Action/Operator :

- **Description** : Drive truck *truck* from a location *loc-from* to a location *loc-to*,
- **Precondition** : *at(truck, loc-from)* must be true,
- **Effect** : *at(truck, loc-to)* becomes true and *at(truck, loc-from)* becomes false.

In PDDL:

```
(:action drive-truck
    :parameters (?truck - truck ?loc-from - place ?loc-to - place ?city - city)
    :precondition (and (at ?truck ?loc-from) (in-city ?loc-from ?city) (in-city ?loc-to ?
↪city))
    :effect (and (not (at ?truck ?loc-from)) (at ?truck ?loc-to))
)
```

Note:

- Action preconditions and effects can be more complicated than seen so far. They can be universally or existentially quantified using PDDL statement of the form `(forall (?v1 ... ?vn) <effect >)`. In that case, specific requirements must be used, for instance `:adl`.
- They can be conditional : `(when <condition > <effect >)`
- Action They can have costs, duration, time constraints etc.

2.2.2 Defining the Problem

Now, let define in the file `problem.pddl` a simple problem and its components :

- the objects,
- the initial state,
- the goal to reach.

First, we have to define the name of the problem and indicate the domain associated with this problem:

In PDDL, we write :

```
(define (problem p01)
        (:domain logistics)
```

Objects

In this example, we use the following objects:

- A Truck : *truck*
- An airplane: *airplane*
- Two airports : *cdg*, *lhr*
- Two places : *north*, *south*
- Two cities : *london*, *paris*
- Two packages : *p1*, *p2*

In PDDL, we write:

```
(:objects plane - airplane
    truck - truck
    cdg lhr - airport
    south north - location
    paris london - city
    p1 p2 - package
)
```

Note: The types of the object can be only the types defined in the domain or the type `object`.

Initial State

The initial state is a set of ground predicates. A predicate is ground iff all the variables are bound to objects. The ground predicates in the initial state represent true facts in this state. Any fact that is not represented in a state is false: In our case:

```
(: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 Description

The goal is to have *at(p1, north)* and *at(p2, south)* in the final state (no matter the truth value of the other predicates). In PDDL, we write:

```
(:goal (and (at p1 north)
    (at p2 south))
)
```

Note: If you have not completed the files as you go, the domain `logistics.pddl` file and the problem `problem.pddl` file can be downloaded directly.

Todo: Run HSP planner on the logistics domain and the problem.

2.2.3 Running the logistics example

The procedure to run the logistic domain and problem with HSP planner of the library is given below:

1. Open a terminal
2. Create a directory called PDDL4J:

```
$ mkdir pddl4j
```

3. Go to the directory:

```
$ cd pddl4j
```

4. Download the binary of last release (X.X.X is the number of release - see *Download*.

```
$ wget http://pddl4j.imag.fr/repository/pddl4j/binaries/pddl4j-4.0.0.jar
```

5. Move the domain file `logistics.pddl` previously created or download it in the `pdd4j` directory.
6. Move the domain file `problem.pddl` previously created or download it in the `pdd4j` directory.

7. Test the example:

```
$ java -jar pddl4j-X.X.jar -server -Xms2048m -Xmx2048m -p HSP -o logistics.
↪pddl -f problem.pddl
```

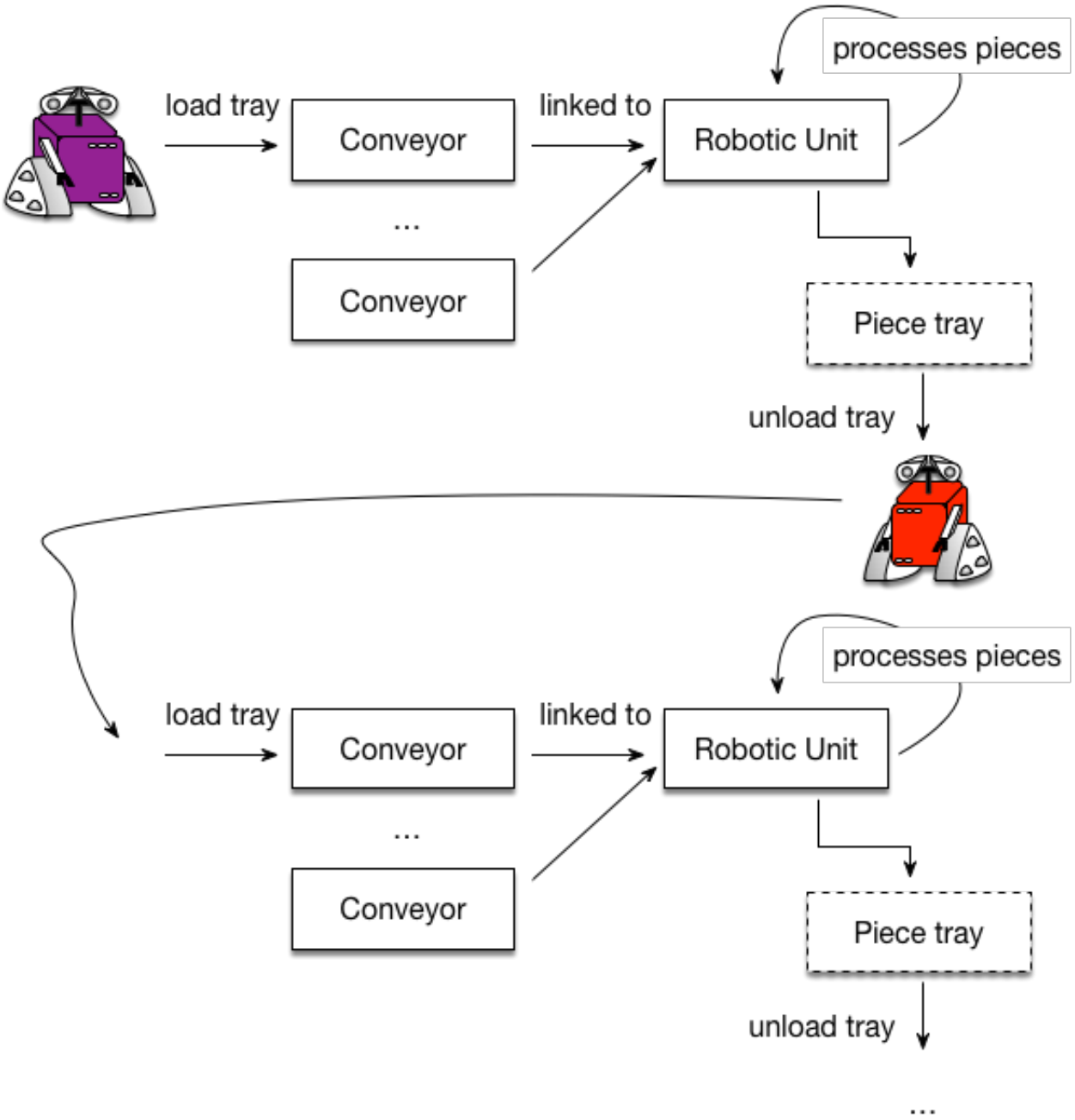
- **The JVM (Java Virtual Machine) arguments:**
  - `-jar` specified the executable jar of the library.
  - `-XMS` and `-XMX` allow to set the maximum heap size and the maximum memory size that can be used by the JVM

The result will be:

```
parsing domain file "logistics.pddl" done successfully
parsing problem file "problem.pddl" done successfully

encoding problem done successfully (28 ops, 17 facts)

found plan as follows:

00: (        loadd-airplane p1 plane lhr) [1]
01: (        loadd-airplane p2 plane lhr) [1]
02: (         fly-airplane plane lhr cdg) [1]
03: (       unload-airplane p1 plane cdg) [1]
04: (       unload-airplane p2 plane cdg) [1]
05: (            load-truck p1 truck cdg) [1]
06: (            load-truck p2 truck cdg) [1]
07: (  drive-truck truck cdg south paris) [1]
08: (        unload-truck p2 truck south) [1]
09: (drive-truck truck south north paris) [1]
10: (        unload-truck p1 truck north) [1]

plan total cost: 11,00

time spent:  0,09 seconds parsing
             0,03 seconds encoding
             0,01 seconds searching
             0,13 seconds total time
```

2.3 An Industrial Use Case

This is a real case that we tackled for a manufacturing company. This company devises supply chains to make pieces of medical equipments. A supply chain consists of independent robotized units/cells, which realize specific operations on the pieces: cleaning, checking, marking, assembling etc. The pieces are put on trays, and mobile robots are programmed to take and to transport the trays between the different units. The image below illustrates this process:



There are different kind of pieces at the beginning of the supply chain. A tray contains only one kind of pieces, and, each piece undergoes a sequence of operations from the beginning to the end of the supply chain. At the beginning of the supply chain, a unit is used to store all the trays. The units can have several inputs named "conveyors". The conveyors and the units are specific to a set of pieces: pieces are admissible to identified conveyors and units. Initially, every processing (unit loading/unloading, robot movements etc.) was hard-coded in a database by human operators.

Automated planning is now used to optimize unit/robot scheduling and to increase production efficiency.

Todo:

- Try to write your own domain file for this problem.
- Before reading the proposed solution below, write a simple problem file and test your domain. For instance, a simple problem is one type of pieces, a single tray, robot and conveyor; two units, a stocker storing this tray at the initial state and a processing unit. The goal is for the unit to perform three operations (op10 > op20 > op30) on the tray.

2.3.1 Defining the Domain

Requirements

Let start by creating the domain file. For instance, `rsc-domain.pddl` write the following PDDL to give a name to this domain and specify the requirements of the domain.

```
(define (domain robotic-manufactoring)
    (:requirements :strips :typing)
```

Types

Then, define the set of objects (types) that will be used in this domain. Quite obviously, we will have the types *robot* (mobile robots), *conveyor*, *unit*, *piece*, etc.

```
(:types
    robot - agent
    conveyor unit - location
    piece operation tray - object
)
```

Constants

We also declare a dummy operation called *stop* as a constant of the domain which will be used in one action:

Predicates

Note:

Now it is time to think to a model for the domain. It is based on the following ideas:

- *Producer/consumer:* the trays are resources consumed by the conveyors and produced by the units. A "one-to-many" relation is created between each unit and the conveyors. A tuple conveyors/unit is like a (Petri Net) "machine" that consumes and produces trays. The conveyors are the inputs and the unit is the output. Each input/output's capacity is one,
- *Operation stacks:* each tray is associated to a stack of operations that have to be performed on the pieces of the tray. The last operation of the stack is always _stop_. Each time a machine consumes a tray, the associated stack is pulled,

- *Goal:* to empty all the stacks by connecting the machines with robots transporting trays from units (outputs) to conveyors (inputs). The capacity of the robots is one.

Here is the vocabulary ("predicates") that will be used by the actions:

```
(:predicates
    ;;robot
    (robot_available ?robot - robot)
    ;; is the robot available? capacity is one
    (robot_at ?robot - robot ?l - location)
    ;; location of a robot. Either a conveyor or a unit

    ;;conveyor
    (conveyor_accepted_piece ?piece - piece ?conv - conveyor)
    ;; constraint on admissible pieces
    (conveyor_available ?conv - conveyor)
    ;; is the conveyor available? capacity is one
    (conveyor_unit ?conv - conveyor ?unit - unit)
    ;; "one-to-many" relation between units and conveyors

    ;;unit
    (unit_accepted_piece ?piece - piece ?unit - unit)
    ;; constraint on admissible pieces
    (unit_available ?unit - unit)
    ;; is the unit available? unit capacity is one
    (unit_operation ?op - operation ?unit - unit)
    ;; operation provided by the unit

    ;;tray
    (tray_on_unit ?tray - tray ?unit - unit)
    ;; the tray is in the unit
    (tray_on_conv ?tray - tray ?conv - conveyor)
    ;; the tray is input into the conveyor
    (tray_on_robot ?tray - tray ?robot - robot)
    ;; the robot is at the tray
    (tray_completed ?tray - tray)
    ;; all the scheduled operations are completed

    ;;piece
    (piece_on ?piece - piece ?tray - tray)
    ;; "one-to-one" relation: trays contain only one type of pieces

    ;;stack of operations
    (start ?op - operation ?tray - tray)
    ;; ?op is on top of the stack.
    ;; The stack has a one-to-one relation with the tray (same id)
    (todo ?opontop - operation ?nextop - operation ?tray - tray)
    ;; linked list of operations: ?nextop follows ?opontop. Last operation is stop
)
```

For instance, in the problem file, you can now specify an initial state beginning by:

```
(start op10 tray32)
```

(continues on next page)

(continued from previous page)

```
(todo op10 op20 tray32)
(todo op20 op30 tray32)
(todo op30 stop tray32)
```

This means that the sequence *op10*, *op20*, *op30* of operations is scheduled on *tray32*.

Likewise,

```
(conveyor_unit conv1 unit1)
(conveyor_unit conv2 unit1)
```

means that *unit1* has two inputs *conv1* and *conv2*.

Operators

The new step is to define all the actions. For this domain, we will need 6 actions:

```
(:action pickup_tray_on_unit
    :parameters (?robot - robot ?unit - unit ?tray - tray)
    :precondition (and (robot_available ?robot)
                       (robot_at ?robot ?unit)
                       (tray_on_unit ?tray ?unit)
                   )
    :effect (and (not (tray_on_unit ?tray ?unit))
                 (not (robot_available ?robot))
                 (tray_on_robot ?tray ?robot)
                 (unit_available ?unit)
            )
)
```

Action *pickup_tray_on_unit* allows a robot to pickup a tray on a unit provided the robot is available and located at this unit. The effects are that the tray is no more on the unit, the tray is on the robot and the robot is not available to pickup another tray. The unit becomes available to process another tray.

```
(:action drop_tray_on_conveyor
    :parameters (?robot - robot ?conv - conveyor ?tray - tray ?piece - piece)
    :precondition (and (conveyor_available ?conv)
                       (robot_at ?robot ?conv)
                       (tray_on_robot ?tray ?robot)
                       (conveyor_accepted_piece ?piece ?conv)
                       (piece_on ?piece ?tray)
                   )
    :effect (and (not (conveyor_available ?conv))
                 (not (tray_on_robot ?tray ?robot))
                 (tray_on_conv ?tray ?conv)
                 (robot_available ?robot))
)
```

Action *drop_tray_on_conveyor* is the counterpart of *pickup_tray_on_unit*. It allows a robot to put a tray on a conveyor. The preconditions are that the robot and the conveyor are in the same place, the conveyor is available and it accepts the same type of pieces than the tray. The effects are that the conveyor is no more available, the tray is no more on the robot (it is on the conveyor) and the robot is now available.

```
(:action robot_move
    :parameters (?robot - robot ?from - location ?to - location)
    :precondition (and (robot_at ?robot ?from))
    :effect (and (robot_at ?robot ?to)
                 (not (robot_at ?robot ?from))
            )
)
```

Action *robot_move* is trivial: it moves a robot from location *?from$ to location $?to*. Locations are either a conveyor or a unit (see *:types* keyword).

```
(:action conveyor_load_tray_in_unit
    :parameters (?conv - conveyor ?unit - unit ?tray - tray ?piece - piece)
    :precondition (and (unit_available ?unit)
                       (conveyor_unit ?conv ?unit)
                       (unit_accepted_piece ?piece ?unit)
                       (piece_on ?piece ?tray)
                       (tray_on_conv ?tray ?conv)
                  )
    :effect (and (not (tray_on_conv ?tray ?conv))
                 (not (unit_available ?unit))
                 (tray_on_unit ?tray ?unit))
    )
```

Action *conveyor_load_tray_in_unit* consumes a tray that is loaded on a conveyor linked to a unit provided the pieces of the tray are accepted by this unit. As a consequence, the tray is no more one the conveyor, the unit is not available and the tray is on the unit, ready for processing.

```
(:action unit_execute_operation
    :parameters (?unit - unit ?top - operation ?next - operation ?tray - tray)
    :precondition (and (unit_operation ?top ?unit)
                       (tray_on_unit ?tray ?unit)
                       (start ?top ?tray)
                       (todo ?top ?next ?tray)
                   )
    :effect (and (start ?next ?tray)
                 (not (todo ?top ?next ?tray))
                 (not (start ?top ?tray))
            )
    )
```

Action *unit_execute_operation* applies the operation pending on top of the tray's stack. The preconditions are that the unit is able to perform this operation, the tray is in the unit and this operation operation is on top of the stack. The effects are that the operation is pulled from the stack and the next operation becomes the top of the stack.

```
(:action tray_completed
    :parameters (?op - operation ?tray - tray ?unit - unit)
    :precondition (and (start stop ?tray)
                       (tray_on_unit ?tray ?unit)
                  )
    :effect (and (tray_completed ?tray)
                 (unit_available ?unit)
                 (not (tray_on_unit ?tray ?unit)))
    )
```

Action *tray_completed* is a dummy action which purpose is to check that all the scheduled operations on a tray have been done (*stop* operation on top of the tray's stack). It is used to build a goal state and to terminate the planning procedure for a given tray. Here we suppose that an operator picks up the tray once all the operations have been done and the unit becomes available to process another tray.

2.3.2 Defining the problem

Let start by creating the problem file, e.g., `rsc_problem_easy.pddl`. The problem we wish to define is a simple problem with one type of pieces, a single tray, robot and conveyor; two units, a stocker storing this tray at the initial state and a processing unit. The goal is for the unit to perform three operations (*op10* > *op20* > *op30*) on the tray.

Objects

Hence, the types and objects are as follows:

```
(:objects
    unit1 stocker - unit
    conv1 - conveyor
    robot1 - robot

    tray1 - tray
    piece1 - piece

    op10 op20 op30 - operation
)
```

Initial State

This snippet of code is the initial state:

```
  (:init
      ;; Operation schedule
      (start op10 tray1)
      (todo op10 op20 tray1)
      (todo op20 op30 tray1)
      (todo op30 stop tray1)

      ;; Initiate pieces on tray
      (piece_on piece1 tray1)

      ;; At the beginning, tray1 is on the stocker
      (tray_on_unit tray1 stocker)

      ;; Initiate robot
      (robot_at robot1 stocker)
      (robot_available robot1)
      ;; Initiate conveyor
      (conveyor_unit conv1 unit1)
      ;; Setup unit
      (unit_accepted_piece piece1 unit1)
      (unit_accepted_piece piece1 stocker)
```

(continues on next page)

(continued from previous page)

```
      (unit_operation op10 unit1)
      (unit_operation op20 unit1)
      (unit_operation op30 unit1)

      ;; Unit1 is ready
      (unit_available unit1)

      ;; Setup conveyor
      (conveyor_accepted_piece piece1 conv1)
      (conveyor_available conv1)

      ;; Setup robot
      (robot_available robot1)
)
```

Goal Description

The goal is simply the completion of *tray1*:

```
(:goal
    (and (tray_completed tray1))
)
```

Complete files can be downloaded here:

- `rsc_domain.pddl`
- `rsc_problem_easy.pddl`
- `rsc_problem_hard.pddl`

Note: To go further, we recommend the reader to refer to this good resource on PDDL

CHAPTER

THREE

GETTING STARTED

This section explains how to download PDDL4J from GitHub, create an executable of the library and run the Fast Forward planner implemented in the library.

3.1 Prerequisites

We assume that :

- Gradle is installed on your computer. For more information see the dedicated page How to install Gradle.
- Java JDK version 8 or higher is installed. To check, run :

```
java -version
```

3.2 Getting PDDL4J

To get PDDL4J just checkout the source from git repository:

```
git clone https://github.com/pellierd/pddl4j.git
cd pddl4j
```

3.3 Creating the executable jar

To build PDDL4J and creating the executable jar use the following command line:

```
./gradlew jar
```

This command build a single jar of the PDDL4J library containing all the dependency libraries used by PDDL4J. The jar generated is located in the directory `build/libs/pddl4j-X.X.jar` where X.X is the version of PDDL4J.

3.4 Example: Running Fast Forward planner

Several planners are implemented in PDDL4J (see section *Running Planners from Command line*.) to have the full list of planners implemented in the library and have the command lines to run them. As sample, find below the command line to launch Fast Forward planner implemented in the library.

```
$ java -cp build/libs/pddl4j-4.0.jar fr.uga.pddl4j.planners.statespace.FF \
   src/test/resources/benchmarks/pddl/ipc2000/logistics/strips-typed/domain.pddl \
   src/test/resources/benchmarks/pddl/ipc2000/logistics/strips-typed/p01.pddl
```

This command run the planner FF on the domain logics and the problem 1.

The output produces by the planner is as follow:

```
parsing domain file "domain.pddl" done successfully
parsing problem file "p01.pddl" done successfully

problem instantiation done successfully (140 actions, 56 fluents)
* starting enforced hill climbing
* enforced hill climbing succeeded

found plan as follows:

00: (     load-truck obj23 tru2 pos2) [0]
01: (     load-truck obj21 tru2 pos2) [0]
02: (     load-truck obj13 tru1 pos1) [0]
03: (     load-truck obj11 tru1 pos1) [0]
04: (drive-truck tru2 pos2 apt2 cit2) [0]
05: (   unload-truck obj23 tru2 apt2) [0]
06: (  load-airplane obj23 apn1 apt2) [0]
07: (   unload-truck obj21 tru2 apt2) [0]
08: (  load-airplane obj21 apn1 apt2) [0]
09: (    fly-airplane apn1 apt2 apt1) [0]
10: (unload-airplane obj23 apn1 apt1) [0]
11: (unload-airplane obj21 apn1 apt1) [0]
12: (drive-truck tru1 pos1 apt1 cit1) [0]
13: (     load-truck obj23 tru1 apt1) [0]
14: (     load-truck obj21 tru1 apt1) [0]
15: (   unload-truck obj13 tru1 apt1) [0]
16: (   unload-truck obj11 tru1 apt1) [0]
17: (drive-truck tru1 apt1 pos1 cit1) [0]
18: (   unload-truck obj23 tru1 pos1) [0]
19: (   unload-truck obj21 tru1 pos1) [0]

time spent:       0,02 seconds parsing
                  0,04 seconds encoding
                  0,02 seconds searching
                  0,07 seconds total time

memory used:      0,00 MBytes for problem representation
                  0,00 MBytes for searching
                  0,00 MBytes total
```

Most of the domains and the problems from IPC (International Planning Competition) are available for testing in the

directory `src/test/resources/benchmarks/`. They are classified by year and by competition track.

CHAPTER

FOUR

BUILDING THE LIBRARY

4.1 Building from source code

The general command line to build PDDL4J from source code is as follow:

```
./gradlew build [options]
```

Options are:

- `-PnoCheckStyle` to skip checkstyle verification
- `-PnoTest` to tests

Note: To speed up the compilation of PDDL4, we recommend not running tests and style checks.

The command line to build PDDL4J without test and checkstyle verification is:

```
./gradlew build -PnoCheckStyle -PnoTest
```

The different build stages are the following:

1. Compile Javacc (used to generate the parser)
2. Compile Java
3. Generate jar file
4. Checkstyle Main (automate the process of checking Java main code), report in `build/reports/checkstyle/main.xml`
5. Checkstyle Test (automate the process of checking Java test code), report in `build/reports/checkstyle/test.xml`
6. Test (use JUnit testing framework to test PDDL4J functions), report in `build/test-results/test`

4.2 Creating a jar of the library

To create a jar of the PDDL4J library with all dependency libraries just use the command line:

```
./gradlew jar
```

The jar generated is located in the directory `./build/libs/pddl4j-X.X.jar` where X.X is the version of PDDL4J. If the library is not build, the library is built before. The libraries include in the jar are :

- Log4j used for logging;
- JUnit used for unit tests;
- Picocli to deal with planners command line.
- JOL to analyze object layout schemes in JVMs.

4.3 Generating the Java documentation

To generate the java documentation of the library use the following command line:

```
./gradlew javadoc
```

To access to the documentation generated open the file `./build/docs/javadoc/index-all.html`.

4.4 Generating the BNF of the parser

To generate the BNF (Backus–Naur form) of the PDDL language accepted by the parser use the following command line:

```
./gradlew jjdoc
```

To access to the documentation generated open the file `./build/docs/PDDL4J_BNF/lexer.html`.

4.5 Generating the documentation

To generate the documentation just run the command line:

```
./gradlew site
```

To access to the documentation generated open the file `./build/docs/site/index.html`.

Note: The documentation is generated using Sphinx with readthedocs using reStructuredText. The source code of the documentation is available in `./docs`.

4.6 Running JUnit Tests

JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of the unit testing frameworks.

As a developer, it is important to include unit tests in your program to ensure that the functions/methods/algorithms return the expected results. In PDDL4J, unit tests are included to test the parser, the search strategies and planners with IPC benchmarks. To run all these tests, use the following command:

```
./gradlew test
```

The reports containing the results of JUnit tests are available in `./build/test-results/test/` folder.

Note: The execution of the tests is very time consuming. All planners are tested in several configurations on all IPC benchmarks they can handle. A search time is allocated to each problem of each domain. This time is usually set at 10 seconds. As soon as the planner fails to solve a problem within the time limit, the test procedure stops and checks whether the plans found are valid before finally moving on to the next domain and set of problems. For PDDL problems the plan validator used is VAL. For HDDL problems the plan validator used is Panda. Both plan validator are used in the international planning competition.

Warning: The planners JUnit tests will fail on windows os. The plan validators used are only available for linux or mac.

4.7 Checking source code convention

PDDL4J uses Checkstyle to check code source convention of the project. Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code to spare humans of this boring (but important) task. This makes it ideal for projects that want to enforce a coding standard.

The different programming rules are defined in the `./checkstyle.xml` file located in the `./config/checkstyle/` folder. A task has also been added in the `./build.gradle` configuration file.

Thus, when building PDDL4J, a report containing the various errors will automatically be generated allowing the developers to correct his/her code before committing it. The generated report is located in `./build/report/checkstyle/` folder.

To run only checkstyle on the PDDL4J source code, use the following command line:

```
./gradlew checkstyleMain
```

It is also possible to run checkstyle on the source code of the JUnit tests:

```
./gradlew checkstyleTest
```

It is possible to disable code analysis with Checkstyle by passing the following parameter to Gradle:

```
./gradlew build -PnoCheckStyle
```

CHAPTER

FIVE

RUNNING PLANNERS FROM COMMAND LINE

The planners currently implemented in the library are:

1. **FF** (*FastForward*). It is based on Enforced Hill Climbing algorithm and A* search and the relaxed planning graph heuristic devised by J. Hoffmann.
2. **HSP** (*Heuristic Search Planner*). It is based on A* search that can be combined with any goal cost heuristics functions developed in the library.
3. **GSP** (*Generic Search Planner*). This planner is a generic state space planner. It is possible to choose the search strategy and the goal cost heuristic function.
4. **TFD** (*Total-order Forward Decomposition*). It is based on dept first search search strategy and can only deal with total-order task decomposition.
5. **PFD** (*Partial-order Forward Decomposition*). It is based on dept first search search strategy and can deal with partial-order task decomposition.

5.1 FF (FastForward)

The command line syntax to launch the planner is as follow:

```
FF [-hV] [-l=<logLevel>] [-t=<timeout>] [-w=<weight>] <domain> <problem>

Description:

Solves a specified planning problem combining enforced hill climbing and A*
search strategies using the the delete relaxation heuristic.

Parameters:
   <domain>              The domain file.
   <problem>             The problem file.

Options:
  -l, --log=<logLevel>      Set the level of trace of the planner: ALL, DEBUG,
                            INFO, ERROR, FATAL, OFF, TRACE (preset INFO).
  -t, --timeout=<timeout>   Set the time out of the planner in seconds (preset
                            600s).
  -w, --weight=<weight>     Set the weight of the heuristic (preset 1.0).
  -h, --help                Show this help message and exit.
  -V, --version             Print version information and exit.
```

Command line example:

```
java -cp build/libs/pddl4j-4.0.0.jar fr.uga.pddl4j.planners.statespace.FF
       src/test/resources/benchmarks/pddl/ipc2002/depots/strips-automatic/domain.pdd
       src/test/resources/benchmarks/pddl/ipc2002/depots/strips-automatic/p01.pddl
       -t 1000
```

This command launches FF and allocates 1000 seconds to the search.

5.2 HSP (Heuristic Search Planner)

The command line syntax to launch the planner is as follow:

```
HSP [-hV] [-e=<heuristic>] [-l=<logLevel>]
                          [-t=<timeout>] [-w=<weight>] <domain> <problem>
Description:

Solves a specified planning problem using A* search strategy.

Parameters:
    <domain>              The domain file.
    <problem>             The problem file.

Options:
    -l, --log=<logLevel>      Set the level of trace: ALL, DEBUG, INFO, ERROR,
                                FATAL, OFF, TRACE (preset INFO).
    -t, --timeout=<timeout>   Set the time out of the planner in seconds (
                                preset 600s).
    -w, --weight=<weight>     the weight of the heuristic (preset 1.0).
    -e, --heuristic=<heuristic>
                              Set the heuristic : AJUSTED_SUM, AJUSTED_SUM2,
                                AJUSTED_SUM2M, COMBO, MAX, FAST_FORWARD,
                                SET_LEVEL, SUM, SUM_MUTEX (preset: FAST_FORWARD)
    -h, --help                Show this help message and exit.
    -V, --version             Print version information and exit.
```

Command line example:

```
java -cp build/libs/pddl4j-4.0.0.jar fr.uga.pddl4j.planners.statespace.HSP
     src/test/resources/benchmarks/pddl/ipc2002/depots/strips-automatic/domain.pdd
        src/test/resources/benchmarks/pddl/ipc2002/depots/strips-automatic/p01.pddl
     -e MAX
     -w 1.2
     -t 600
```

This command launches HSP using MAX heuristic with a weight of 1.2 and allocates 600 seconds to the search.

5.3 GSP (Heuristic Search Planner)

The command line syntax to launch the planner is as follow:

```
GSP [-hV] [-e=<heuristic>] [-l=<logLevel>]
                           [-t=<timeout>] [-w=<weight>] [-s
                           [=<strategies>...]]... <domain> <problem>

Description:

Solves a specified planning problem using a specified search strategy and heuristic.

 Parameters:
    <domain>              The domain file.
    <problem>             The problem file.

Options:
    -l, --log=<logLevel>      Set the level of trace: ALL, DEBUG, INFO, ERROR,
                                FATAL, OFF, TRACE (preset INFO).
    -t, --timeout=<timeout>   Set the time out of the planner in seconds (
                                preset 600s).
    -w, --weight=<weight>     Set the weight of the heuristic (preset 1.0).
    -e, --heuristic=<heuristic>
                              Set the heuristics: AJUSTED_SUM, AJUSTED_SUM2,
                                AJUSTED_SUM2M, COMBO, MAX, FAST_FORWARD,
                                SET_LEVEL, SUM, SUM_MUTEX (preset: FAST_FORWARD)
    -s, --search-strategies[=<strategies>...]
                              Set the search strategies: ASTAR,
                                ENFORCED_HILL_CLIMBING, BREADTH_FIRST,
                                GREEDY_BEST_FIRST, DEPTH_FIRST, HILL_CLIMBING
                                (preset: ASTAR)
    -h, --help                Show this help message and exit.
    -V, --version             Print version information and exit.
```

Command line example:

```
java -cp build/libs/pddl4j-4.0.0.jar fr.uga.pddl4j.planners.statespace.GSP
      src/test/resources/benchmarks/pddl/ipc2002/depots/strips-automatic/domain.pddl
      src/test/resources/benchmarks/pddl/ipc2002/depots/strips-automatic/p01.pddl
      -s ENFORCED_HILL_CLIMBING ASTAR
      -e FAST_FORWARD
      -t 1000
```

This command launches GSP using first ENFORCED_HILL_CLIMBING search strategy and if the search fails then ASTAR with the heuristic FAST_FORWARD and allocates 1000 seconds to the search.

5.4 TFD (Total-order Forward Decomposition)

The command line syntax to launch the planner is as follow:

```
TFD [-hiV] [-l=<logLevel>] [-t=<timeout>] <domain> <problem>

Description:

Solves a specified planning problem using a Total-order Forward Decomposition strategy.

Parameters:
    <domain>              The domain file.
    <problem>             The problem file.

Options:
    -t, --timeout=<timeout>   Set the time out of the planner in seconds (preset 600s).
    -l, --log=<logLevel>      Set the level of trace of the planner: ALL, DEBUG,
                                INFO, ERROR, FATAL, OFF, TRACE (preset INFO).
    -i, --interactive         Set the planner in interactive mode for debug
    -h, --help                Show this help message and exit.
    -V, --version             Print version information and exit.
```

Command line example:

```
java -cp build/libs/pddl4j-4.0.0.jar fr.uga.pddl4j.planners.htn.stn.TFD
         src/test/resources/benchmarks/hddl/ipc2020/barman/domain.hddl
         src/test/resources/benchmarks/hddl/ipc2020/barman/p01.hddl
         -t 600
```

This command launches TFD and allocates 600 seconds to the search.

Note: It is possible to use the iterative (-i) mode to debug and print step by step the task decomposition.

5.5 PFD (Partial-order Forward Decomposition)

The command line syntax to launch the planner is as follow:

```
TFD [-hiV] [-l=<logLevel>] [-t=<timeout>] <domain> <problem>

Description:

Solves a specified planning problem using a Partial-order Forward Decomposition strategy.

Parameters:
    <domain>              The domain file.
    <problem>             The problem file.

Options:
    -t, --timeout=<timeout>   Set the time out of the planner in seconds (preset 600s).
    -l, --log=<logLevel>      Set the level of trace of the planner: ALL, DEBUG,
```

(continues on next page)

(continued from previous page)

```
                              INFO, ERROR, FATAL, OFF, TRACE (preset INFO).
   -i, --interactive         Set the planner in interactive mode for debug
   -h, --help                Show this help message and exit.
   -V, --version             Print version information and exit.
```

Command line example:

```
java -cp build/libs/pddl4j-4.0.0.jar fr.uga.pddl4j.planners.htn.stn.PFD
        src/test/resources/benchmarks/hddl/ipc2020/barman/domain.hddl
        src/test/resources/benchmarks/hddl/ipc2020/barman/p01.hddl
        -t 600
```

This command launches PFD and allocates 600 seconds to the search.

CHAPTER

SIX

CONFIGURING PLANNERS BY PROGRAMMING

There are two ways to run a existing planner by programming: by direct manipulating or using the class PlannerConfiguration.

6.1 Pre-requisite Installations

For this tutorial you need:

- Java JDK version 8 or higher is installed.
- A text editor such as Sublime or Atom or IDE such as Eclipse NetBean or IntelliJ.

In the following, we will give the commands line so that the tutorial can be done independently of any IDE.

6.2 Step 1. Create a simple Java project with PDDL4J

First, open a terminal and create your development directory `PlannerConfigurationExamples`

```
mkdir PlannerConfigurationExamples
```

Then, create the sub-directories of your project

```
cd PlannerConfigurationExamples
mkdir -p src/fr/uga/pddl4j/examples/
mkdir classes
mkdir lib
```

Finally, get the last binary of PDDL4J and save it in the `lib` directory

```
wget http://pddl4j.imag.fr/repository/pddl4j/binaries/pddl4j-4.0.0.jar
mv pddl4j-4.0.0.jar lib/pddl4j-4.0.0.jar
```

You are now ready to configure an existing planner of the library by programming.

6.3 Step 2. By directly manipulating

Create and edit a file called `DirectPlannerConfigurationExample.java` in the directory `src/fr/uga/pddl4j/examples/`. The skeleton of this class is given bellow:

```
package fr.uga.pddl4j.examples;

import fr.uga.pddl4j.heuristics.state.StateHeuristic;
import fr.uga.pddl4j.planners.LogLevel;
import fr.uga.pddl4j.planners.statespace.HSP;

import java.io.FileNotFoundException;

/**
 * The class is an example. It shows how to create a planner by programming and running␣
↪it.
 *
 * @author D. Pellier
 * @version 4.0 - 30.11.2021
 */
public class DirectPlannerConfigurationExample {

    /**
     * The main method of the class.
     *
     * @param args the command line arguments. No argument is used.
     */
    public static void main(String[] args) {

        // The path to the benchmarks directory
        final String benchmarks = "src/test/resources/benchmarks/pddl/ipc2002/depots/
↪strips-automatic/";

        // Creates the planner
        HSP planner = new HSP();
        // Sets the domain of the problem to solve
        planner.setDomain(benchmarks + "domain.pddl");
        // Sets the problem to solve
        planner.setProblem(benchmarks + "p01.pddl");
        // Sets the timeout of the search in seconds
        planner.setTimeout(1000);
        // Sets log level
        planner.setLogLevel(LogLevel.INFO);
        // Selects the heuristic to use
        planner.setHeuristic(StateHeuristic.Name.MAX);
        // Sets the weight of the heuristic
        planner.setHeuristicWeight(1.2);

        // Solve and print the result
        try {
            planner.solve();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
```

(continues on next page)

(continued from previous page)

```
        }

    }
}
```

The code above configures and run the planner called HSP on the first problem of the depot domain from IPC 2002.

To test the above code use the following command line to compile the example:

```
javac -d classes -cp classes:lib/pddl4j-4.0.0.jar src/fr/uga/pddl4j/examples/
↪DirectPlannerConfigurationExample.java
```

and then the following command line to run the example:

```
java -cp classes:lib/pddl4j-4.0.0.jar fr.uga.pddl4j.examples.
↪DirectPlannerConfigurationExample
```

6.4 Step 3. Using the class planner configuration

Create and edit a file called `PlannerConfigurationExample.java` in the directory `src/fr/uga/pddl4j/examples/`. The skeleton of this class is given bellow:

```
package fr.uga.pddl4j.examples;

import fr.uga.pddl4j.heuristics.state.StateHeuristic;
import fr.uga.pddl4j.planners.LogLevel;
import fr.uga.pddl4j.planners.Planner;
import fr.uga.pddl4j.planners.statespace.HSP;

import java.io.FileNotFoundException;

/**
 * The class is an example. It shows how to create a HSP planner by programming and␣
↪running it using the class
 * {@code PlannerConfiguration}.
 *
 * @author D. Pellier
 * @version 4.0 - 30.11.2021
 */
public class PlannerConfigurationExample {

    /**
     * The main method of the class.
     *
     * @param args the command line arguments. No argument is used.
     */
    public static void main(String[] args) {

        // The path to the benchmarks directory
        final String benchmarks = "src/test/resources/benchmarks/pddl/ipc2002/depots/
↪strips-automatic/";
```

(continues on next page)

(continued from previous page)

```
        // Gets the default configuration from the planner
        fr.uga.pddl4j.planners.PlannerConfiguration config = HSP.
↪getDefaultConfiguration();
        // Sets the domain of the problem to solve
        config.setProperty(HSP.DOMAIN_SETTING, benchmarks + "domain.pddl");
        // Sets the problem to solve
        config.setProperty(HSP.PROBLEM_SETTING, benchmarks + "p01.pddl");
        // Sets the timeout allocated to the search.
        config.setProperty(HSP.TIME_OUT_SETTING, 1000);
        // Sets the log level
        config.setProperty(HSP.LOG_LEVEL_SETTING, LogLevel.INFO);
        // Sets the heuristic used to search
        config.setProperty(HSP.HEURISTIC_SETTING, StateHeuristic.Name.MAX);
        // Sets the weight of the heuristic
        config.setProperty(HSP.WEIGHT_HEURISTIC_SETTING, 1.2);

        // Creates an instance of the HSP planner with the specified configuration
        Planner planner = Planner.getInstance(Planner.Name.HSP, config);

        // Runs the planner and print the solution
        try {
            planner.solve();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }

    }
}
```

The above code configures and runs same HSP planner with the same configuration as by direct manipulating.

Note: The advantage of using the PlannerConfiguration object is that you can save or load specific configurations using the `store()` and `load()` methods of the class in XML format

---

To test the above code use the following command line to compile the example:

```
javac -d classes -cp classes:lib/pddl4j-4.0.0.jar src/fr/uga/pddl4j/examples/
↪PlannerConfigurationExample.java
```

and then the following command line to run the example:

```
java -cp classes:lib/pddl4j-4.0.0.jar fr.uga.pddl4j.examples.PlannerConfigurationExample
```

CHAPTER

SEVEN

WRITING YOUR OWN PLANNER

The objective of this tutorial is to design a simple planner based on A* search.

1. Create a simple Java project with PDDL4J
2. Create the main class of your planner
3. Get the planner arguments from the command line
4. Searching for a solution plan
5. Write your own A* search strategy
6. Make your planner configurable by programming

7.1 Pre-requisite Installations

For this tutorial you need:

- Java JDK version 8 or higher is installed.
- A text editor such as Sublime or Atom or IDE such as Eclipse NetBean or IntelliJ.

In the following, we will give the commands line so that the tutorial can be done independently of any IDE.

7.2 Step 1. Create a simple Java project with PDDL4J

First, open a terminal and create your development directory ASP (A* Planner)

```
mkdir ASP
```

Then, create the sub-directories of your project

```
cd ASP
mkdir -p src/fr/uga/pddl4j/examples/asp
mkdir classes
mkdir lib
```

Finally, get the last binary of PDDL4J and save it in the `lib` directory

```
wget http://pddl4j.imag.fr/repository/pddl4j/binaries/pddl4j-4.0.0.jar
mv pddl4j-4.0.0.jar lib/pddl4j-4.0.0.jar
```

You are now ready to write your own A* planner.

7.3 Step 2. Create the main class of our planner

Create and edit a file called `ASP.java` in the directory `src/fr/uga/pddl4j/examples/asp`. The skeleton of this class is given bellow:

```
1  package fr.uga.pddl4j.examples.asp;
2
3  import fr.uga.pddl4j.heuristics.state.StateHeuristic;
4  import fr.uga.pddl4j.parser.ParsedProblem;
5  import fr.uga.pddl4j.plan.Plan;
6  import fr.uga.pddl4j.plan.SequentialPlan;
7  import fr.uga.pddl4j.planners.AbstractPlanner;
8  import fr.uga.pddl4j.planners.Planner;
9  import fr.uga.pddl4j.planners.PlannerConfiguration;
10 import fr.uga.pddl4j.planners.SearchStrategy;
11 import fr.uga.pddl4j.planners.statespace.search.StateSpaceSearch;
12 import fr.uga.pddl4j.problem.ADLProblem;
13 import fr.uga.pddl4j.problem.State;
14 import fr.uga.pddl4j.problem.operator.Action;
15 import fr.uga.pddl4j.problem.operator.ConditionalEffect;
16 import org.apache.logging.log4j.LogManager;
17 import org.apache.logging.log4j.Logger;
18 import picocli.CommandLine;
19
20 import java.util.Comparator;
21 import java.util.HashSet;
22 import java.util.List;
23 import java.util.PriorityQueue;
24 import java.util.Set;
25
26 /**
27  * The class is an example. It shows how to create a simple A* search planner able to
28  * solve an ADL problem by choosing the heuristic to used and its weight.
29  *
30  * @author D. Pellier
31  * @version 4.0 - 30.11.2021
32  */
33 public class ASP extends AbstractPlanner<ADLProblem> {
34
35     /**
36      * The class logger.
37      */
38     private static final Logger LOGGER = LogManager.getLogger(ASP.class.getName());
39
40     /**
41      * Instantiates the planning problem from a parsed problem.
42      *
43      * @param problem the problem to instantiate.
44      * @return the instantiated planning problem or null if the problem cannot be
   ↪instantiated.
45      */
46     @Override
47     public ADLProblem instantiate(ParsedProblem problem) {
```

(continues on next page)

(continued from previous page)

```
48          final ADLProblem pb = new ADLProblem(problem);
49          pb.instantiate();
50          return pb;
51      }
52
53      /**
54       * Search a solution plan to a specified domain and problem using A*.
55       *
56       * @param problem the problem to solve.
57       * @return the plan found or null if no plan was found.
58       */
59      @Override
60      public Plan solve(final ADLProblem problem) {
61      }
62      /**
63       * The main method of the <code>ASP</code> planner.
64       *
65       * @param args the arguments of the command line.
66       */
67      public static void main(String[] args) {
68          try {
69              final ASP planner = new ASP();
70          } catch (IllegalArgumentException e) {
71              LOGGER.fatal(e.getMessage());
72          }
73      }
74  }
```

The class ASP extends the abstract class AbstractPlanner that contains the basic methods of any planners. The class is generic. You have to specify the type of problem that your planner is able to solve. In our case, our planner will be only able to solve simple problem called ADLProblem. ADL is a subset of the PDDL language.

Two methods must be overridden at least:

- The method `instantiate(ParsedProblem problem)` is an abstract method of the class `AbstractPlanner`. This method takes as parameter an instance of parsed problem and return the corresponding instantiated or grounding problem. The problem returned contains all the information related to the problem, i.e., the actions, the initial state, the goal of the problem, etc.
- The method `solve(Problem problem)` is the main method of the planner. The method takes as parameter the instantiated problem returned by the previous method overridden and returns a plan solution of null if no plan was found.

Note: The given skeleton contains also a `main()` method to launch the planner from the command line. We will return to this method in the next section.

Note: Every planner can have a logger instance. This logger is based on Log4j library developed by Apache and specialized in logging. A great benefit of Log4j is that different levels of logging can be set for your planner. The levels are hierarchical and are as follows: *TRACE*, *DEBUG*, *INFO*, *WARN*, *ERROR*, and *FATAL*. Have a look to the web site of Log4j for more details. The declaration of the logger is done line 38. To see an example use of the logger see line 71 of the `main()` method.

7.4 Step 3. Get the planner arguments from the command line

Before writing the search algorithm let's first look at how to get the arguments from the command line. Your planner takes as inputs at least a domain file that contains the description of the planning operators and a problem file that define the initial state and the goal to reach. Both files domain and problem rely on PDDL (Planning Domain Description Language). For those who are not familiar with PDDL, first have a look to the tutorial *PDDL Tutorial*. To deal with complex command line arguments, PDDL4J used picocli library. Picocli allow to create rich command line applications just by adding annotation on the main class of our planner.

7.4.1 Step 3.1. Activate the default command line arguments

By default, the class AbstractPlanner already handles the common arguments of all planners: the domain and the problem description, the time allocated to the search and the log level. The domain and the problem descriptions are mandatory parameters. The log level and the time allocated to search are optional.

To activate the default command line for your planner you have just to add the following annotation before the class declaration:

```
/**
 * The class is an example. It shows how to create a simple A* search planner able to
 * solve an ADL problem by choosing the heuristic to used and its weight.
 *
 * @author D. Pellier
 * @version 4.0 - 30.11.2021
 */
@CommandLine.Command(name = "ASP",
    version = "ASP 1.0",
    description = "Solves a specified planning problem using A* search strategy.",
    sortOptions = false,
    mixinStandardHelpOptions = true,
    headerHeading = "Usage:%n",
    synopsisHeading = "%n",
    descriptionHeading = "%nDescription:%n%n",
    parameterListHeading = "%nParameters:%n",
    optionListHeading = "%nOptions:%n")
public class ASP extends AbstractPlanner<ADLProblem> {
```

and complete the `main()` method with the code below:

```
    /**
     * The main method of the <code>ASP</code> planner.
     *
     * @param args the arguments of the command line.
     */
    public static void main(String[] args) {
        try {
            final ASP planner = new ASP();
            CommandLine cmd = new CommandLine(planner);
            cmd.execute(args);
        } catch (IllegalArgumentException e) {
            LOGGER.fatal(e.getMessage());
        }
    }
```

To test, first compile your planner:

```
javac -d classes -cp lib/pddl4j-4.0.0.jar src/fr/uga/pddl4j/examples/asp/ASP.java
```

and run it with the command line:

```
java -cp classes:lib/pddl4j-4.0.0.jar fr.uga.pddl4j.examples.asp.ASP --help
```

You will obtain the following message:

```
ASP [-hiV] [-l=<logLevel>] [-t=<timeout>] <domain> <problem>

Description:

Solves a specified planning problem using a A* search strategy.

Parameters:
    <domain>              The domain file.
    <problem>             The problem file.

Options:
    -t, --timeout=<timeout>   Set the time out of the planner in seconds (preset 600s).
    -l, --log=<logLevel>      Set the level of trace of the planner: ALL, DEBUG,
                                INFO, ERROR, FATAL, OFF, TRACE (preset INFO).
    -h, --help                Show this help message and exit.
    -V, --version             Print version information and exit.
```

7.4.2 Step 3.2 Adding new command line arguments

Now, we have to add the specific arguments of our planner to allow to choose the heuristic function to used and set the weight of the heuristic. This step is relatively simple and straightforward. We just need to declare two new attributes: one for the heuristic and one for its weight. The heuristic is of type GoalCostHeuristic.Name and the weight of type double. Note that the weight must be greater than 0.

```
1    /**
2     * The weight of the heuristic.
3     */
4    private double heuristicWeight;
5
6    /**
7     * The name of the heuristic used by the planner.
8     */
9    private StateHeuristic.Name heuristic;
```

To complete, we also add the corresponding getters and setters:

```
1    /**
2     * Sets the weight of the heuristic.
3     *
4     * @param weight the weight of the heuristic. The weight must be greater than 0.
5     * @throws IllegalArgumentException if the weight is strictly less than 0.
6     */
7    @CommandLine.Option(names = { "-w", "--weight" }, defaultValue = "1.0",
```

(continues on next page)

(continued from previous page)

```
        paramLabel = "<weight>", description = "Set the weight of the heuristic (preset 1.0).")
    public void setHeuristicWeight(final double weight) {
        if (weight <= 0) {
            throw new IllegalArgumentException("Weight <= 0");
        }
        this.heuristicWeight = weight;
    }

    /**
     * Set the name of heuristic used by the planner to the solve a planning problem.
     *
     * @param heuristic the name of the heuristic.
     */
    @CommandLine.Option(names = { "-e", "--heuristic" }, defaultValue = "FAST_FORWARD",
        description = "Set the heuristic : AJUSTED_SUM, AJUSTED_SUM2, AJUSTED_SUM2M, COMBO, "
            + "MAX, FAST_FORWARD SET_LEVEL, SUM, SUM_MUTEX (preset: FAST_FORWARD)")
    public void setHeuristic(StateHeuristic.Name heuristic)  {
        this.heuristic = heuristic;
    }

    /**
     * Returns the name of the heuristic used by the planner to solve a planning problem.
     *
     * @return the name of the heuristic used by the planner to solve a planning problem.
     */
    public final StateHeuristic.Name getHeuristic() {
        return this.heuristic;
    }

    /**
     * Returns the weight of the heuristic.
     *
     * @return the weight of the heuristic.
     */
    public final double getHeuristicWeight() {
        return this.heuristicWeight;
    }
```

To test, your complete command line compile once again your planner:

```
javac -d classes -cp lib/pddl4j-4.0.0.jar src/fr/uga/pddl4j/examples/asp/ASP.java
```

and run it with for instance the command line:

```
java -cp classes:lib/pddl4j-4.0.0.jar fr.uga.pddl4j.examples.ASP
     src/test/resources/benchmarks/pddl/ipc2002/depots/strips-automatic/domain.pddl
     src/test/resources/benchmarks/pddl/ipc2002/depots/strips-automatic/p01.pddl
     -e FAST_FORWARD
     -w 1.2
     -t 1000
```

Now the command line is set. The final command line of your planner is as follows:

```
ASP [-hV] [-e=<heuristic>] [-l=<logLevel>] [-t=<timeout>] [-w=<weight>]
    <domain> <problem>

Description:

Solves a specified planning problem using A* search strategy.

Parameters:
    <domain>              The domain file.
    <problem>             The problem file.

Options:
    -t, --timeout=<timeout>   Set the time out of the planner in seconds (preset
                                600s).
    -l, --log=<logLevel>      Set the level of trace of the planner: ALL, DEBUG,
                                INFO, ERROR, FATAL, OFF, TRACE (preset INFO).
    -w, --weight=<weight>     Set the weight of the heuristic (preset 1.0).
    -e, --heuristic=<heuristic>
                              Set the heuristic : AJUSTED_SUM, AJUSTED_SUM2,
                                AJUSTED_SUM2M, COMBO, MAX, FAST_FORWARD,
                                SET_LEVEL, SUM, SUM_MUTEX (preset: FAST_FORWARD)
    -h, --help                Show this help message and exit.
    -V, --version             Print version information and exit.
```

7.5 Step 4. Searching for a solution plan

You finally think we're here. How write my search procedure ? Two possibilities or the search procedure you want to use already exists in PDDL4J. In this case, it's extremely simple just call the right procedure in the `solve()` method. Otherwise, you have to write your own procedure. Let us first consider the first case. The second will consider in last part of this tutorial.

All the search strategies for state space planning already implemented in PDDL4J are available in the package fr.uga.pddl4j.planners.statespace.search.strategy search strategies. Thus, your `solve()` must look like as follows:

```
1     /**
2      * Search a solution plan to a specified domain and problem using A*.
3      *
4      * @param problem the problem to solve.
5      * @return the plan found or null if no plan was found.
6      */
7     @Override
8     public Plan solve(final ADLProblem problem) {
9         // Creates the A* search strategy
10        StateSpaceSearch search = StateSpaceSearch.getInstance(SearchStrategy.Name.ASTAR,
11            this.getHeuristic(), this.getHeuristicWeight(), this.getTimeout());
12        LOGGER.info("* Starting A* search \n");
13        // Search a solution
14        Plan plan = search.searchPlan(problem);
15        // If a plan is found update the statistics of the planner and log search␣
↪information
```

(continues on next page)

(continued from previous page)

```
16          if (plan != null) {
17              LOGGER.info("* A* search succeeded\n");
18              this.getStatistics().setTimeToSearch(search.getSearchingTime());
19              this.getStatistics().setMemoryUsedToSearch(search.getMemoryUsed());
20          } else {
21              LOGGER.info("* A* search failed\n");
22          }
23          // Return the plan found or null if the search fails.
24          return plan;
25      }
```

First, we create an instance of the search strategy for the problem to solve and then, we try to find a plan for this problem.

Note: If you need to get the goal node for printing for instance returned by the search strategy you can replace the call to `searchPlan()` line 14 by:

```
final Node goal = search.searchSolutionNode(problem);
Planner.getLogger().trace(problem.toString(goal));
Plan plan = search.extractPlan(goal, problem);
```

Now, your planner is ready to solve problems. After compiling the project run your planner with the command line to make a test:

```
java -cp classes:lib/pddl4j-4.0.0.jar fr.uga.pddl4j.examples.ASP
     src/test/resources/benchmarks/pddl/ipc2002/depots/strips-automatic/domain.pddl
     src/test/resources/benchmarks/pddl/ipc2002/depots/strips-automatic/p01.pddl
     -e FAST_FORWARD
     -w 1.2
     -t 1000
```

The output should be:

```
parsing domain file "domain.pddl" done successfully
parsing problem file "p01.pddl" done successfully

problem instantiation done successfully (90 actions, 46 fluents)

* Starting A* search
* A* search succeeded

found plan as follows:

00: (       lift hoist0 crate1 pallet0 depot0) [0]
01: ( lift hoist1 crate0 pallet1 distributor0) [0]
02: (        load hoist0 crate1 truck1 depot0) [0]
03: (        drive truck1 depot0 distributor0) [0]
04: (  load hoist1 crate0 truck1 distributor0) [0]
05: (unload hoist1 crate1 truck1 distributor0) [0]
06: (  drive truck1 distributor0 distributor1) [0]
07: ( drop hoist1 crate1 pallet1 distributor0) [0]
08: (unload hoist2 crate0 truck1 distributor1) [0]
```

(continues on next page)

(continued from previous page)

```
09: ( drop hoist2 crate0 pallet2 distributor1) [0]

time spent:       0,02 seconds parsing
                  0,03 seconds encoding
                  0,01 seconds searching
                  0,07 seconds total time

memory used:      0,00 MBytes for problem representation
                  0,00 MBytes for searching
                  0,00 MBytes total
```

7.6 Step 5. Write your own A* search strategy

Before writing your own A* search strategy, you need to create a class `Node`. This class represents a node of search tree developed by A*.

7.6.1 Step 5.1 Writing your own class Node

For state space planning a Node is a data structure with 5 components:

1. A state, i.e., the state in the state space to which the node corresponds;
2. A parent node, i.e., the node in the search tree that generated this node;
3. An action, i.e., the action that was applied to the parent node to produce this node;
4. A cost, i.e., the cost of the path from the initial state to the node, as indicated by the parent pointer; and
5. A heuristics value, i.e., a estimation of the cost from this node to a solution one.

The easiest way to write your own node class is to inherit the State class that models a state in a compact way. To do so, start creating a file `Node.java` in the repertory `src/fr/uga/pddl4j/examples/asp/` and copy and paste the the skeleton of the class `Node` is given below:

```
1  package fr.uga.pddl4j.examples.asp;
2
3  import fr.uga.pddl4j.problem.State;
4
5  /**
6   * This class implements a node of the tree search.
7   *
8   * @author D. Pellier
9   * @version 1.0 - 02.12.2021
10  */
11 public final class Node extends State {
12
13     /**
14      * The parent node of this node.
15      */
16     private Node parent;
17
18     /**
```

(continues on next page)

(continued from previous page)

```
     * The action apply to reach this node.
     */
    private int action;

    /**
     * The cost to reach this node from the root node.
     */
    private double cost;

    /**
     * The estimated distance to the goal from this node.
     */
    private double heuristic;

    /**
     * The depth of the node.
     */
    private int depth;

    /**
     * Creates a new node from a specified state.
     *
     * @param state the state.
     */
    public Node(State state) {
        super(state);
    }

    /**
     * Creates a new node with a specified state, parent node, operator,
     * cost and heuristic value.
     *
     * @param state     the logical state of the node.
     * @param parent    the parent node of the node.
     * @param action   the action applied to reached the node from its parent.
     * @param cost      the cost to reach the node from the root node.
     * @param heuristic the estimated distance to reach the goal from the node.
     */
    public Node(State state, Node parent, int action, double cost, double heuristic) {
        super(state);
        this.parent = parent;
        this.action = action;
        this.cost = cost;
        this.heuristic = heuristic;
        this.depth = -1;
    }

    /**
     * Creates a new node with a specified state, parent node, operator, cost,
     * depth and heuristic value.
     *
     * @param state     the logical state of the node.
```

(continues on next page)

(continued from previous page)

```
     * @param parent    the parent node of the node.
     * @param action    the action applied to reached the node from its parent.
     * @param cost      the cost to reach the node from the root node.
     * @param depth     the depth of the node.
     * @param heuristic the estimated distance to reach the goal from the node.
     */
    public Node(State state, Node parent, int action, double cost, int depth, double␣
↪heuristic) {
        super(state);
        this.parent = parent;
        this.action = action;
        this.cost = cost;
        this.depth = depth;
        this.heuristic = heuristic;
    }

    /**
     * Returns the action applied to reach the node.
     *
     * @return the action applied to reach the node.
     */
    public final int getAction() {
        return this.action;
    }

    /**
     * Sets the action applied to reach the node.
     *
     * @param action the action to set.
     */
    public final void setAction(final int action) {
        this.action = action;
    }

    /**
     * Returns the parent node of the node.
     *
     * @return the parent node.
     */
    public final Node getParent() {
        return parent;
    }

    /**
     * Sets the parent node of the node.
     *
     * @param parent the parent to set.
     */
    public final void setParent(Node parent) {
        this.parent = parent;
    }

```

(continues on next page)

(continued from previous page)

```
    /**
     * Returns the cost to reach the node from the root node.
     *
     * @return the cost to reach the node from the root node.
     */
    public final double getCost() {
        return cost;
    }

    /**
     * Sets the cost needed to reach the node from the root node.
     *
     * @param cost the cost needed to reach the node from the root nod to set.
     */
    public final void setCost(double cost) {
        this.cost = cost;
    }

    /**
     * Returns the estimated distance to the goal from the node.
     *
     * @return the estimated distance to the goal from the node.
     */
    public final double getHeuristic() {
        return heuristic;
    }

    /**
     * Sets the estimated distance to the goal from the node.
     *
     * @param estimates the estimated distance to the goal from the node to set.
     */
    public final void setHeuristic(double estimates) {
        this.heuristic = estimates;
    }

    /**
     * Returns the depth of this node.
     *
     * @return the depth of this node.
     */
    public int getDepth() {
        return this.depth;
    }

    /**
     * Set the depth of this node.
     *
     * @param depth the depth of this node.
     */
    public void setDepth(final int depth) {
        this.depth = depth;
```

(continues on next page)

(continued from previous page)

```
    }

    /**
     * Returns the value of the heuristic function, i.e.,
     * <code>this.node.getCost() + this.node.getHeuristic()</code>.
     *
     * @param weight the weight of the heuristic.
     * @return the value of the heuristic function, i.e.,
     * <code>this.node.getCost() + this.node.getHeuristic()</code>.
     */
    public final double getValueF(double weight) {
        return weight * this.heuristic + this.cost;
    }

}
```

7.6.2 Step 5.2 Writing your own A* search

A* is an informed search algorithm, or a best-first search, meaning that it solves problems by searching among all possible paths to the solution (goal) for the one that incurs the smallest cost (least distance traveled, shortest time, etc.), and among these paths it first considers the ones that appear to lead most quickly to the solution. It is formulated in terms of weighted graphs: starting from a specific node of a graph, it constructs a tree of paths starting from that node, expanding paths one step at a time, until one of its paths ends at the predetermined goal node (see the pseudocode A* for more details).

At each iteration of its main loop, A* needs to determine which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the cost (total weight) still to go to the goal node. Specifically, A* selects the path that minimizes $f(n) = g(n) + h(n)$ where :

- n is the last node on the path,
- $g(n)$ is the cost of the path from the start node to n, and
- $h(n)$ is a heuristic that estimates the cost of the cheapest path from n to the goal.

For the algorithm to find the actual shortest path, the heuristic function must be admissible, meaning that it never overestimates the actual cost to get to the nearest goal node.

Typical implementations of A* use a priority queue to perform the repeated selection of minimum (estimated) cost nodes to expand. This priority queue is known as the open set or fringe. At each step of the algorithm, the node with the lowest $f(x)$ value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty). The f value of the goal is then the length of the shortest path, since h at the goal is zero in an admissible heuristic. After this algorithm is run, the ending node will point to its predecessor, and so on, until some node's predecessor is the start node (see the `extract` procedure below).

Consider the implementation of A* now with PDDL4J and the new `solve()` procedure:

```
    /**
     * Search a solution plan for a planning problem using an A* search strategy.
     *
     * @param problem the problem to solve.
     * @return a plan solution for the problem or null if there is no solution
     */
```

(continues on next page)

(continued from previous page)

```
    public Plan astar(ADLProblem problem) {

        // First we create an instance of the heuristic to use to guide the search
        final StateHeuristic heuristic = StateHeuristic.getInstance(this.getHeuristic(), problem);

        // We get the initial state from the planning problem
        final State init = new State(problem.getInitialState());

        // We initialize the closed list of nodes (store the nodes explored)
        final Set<Node> close = new HashSet<>();

        // We initialize the opened list to store the pending node according to function f
        final double weight = this.getHeuristicWeight();
        final PriorityQueue<Node> open = new PriorityQueue<>(100, new Comparator<Node>() {
            public int compare(Node n1, Node n2) {
                double f1 =  weight * n1.getHeuristic() + n1.getCost();
                double f2 = weight * n2.getHeuristic() + n2.getCost();
                return Double.compare(f1, f2);
            }
        });

        // We create the root node of the tree search
        final Node root = new Node(init, null, -1, 0, heuristic.estimate(init, problem.getGoal()));

        // We add the root to the list of pending nodes
        open.add(root);
        Plan plan = null;

        // We set the timeout in ms allocated to the search
        final int timeout = this.getTimeout() * 1000;
        long time = 0;

        // We start the search
        while (!open.isEmpty() && plan == null && time < timeout) {

            // We pop the first node in the pending list open
            final Node current = open.poll();
            close.add(current);

            // If the goal is satisfied in the current node then extract the search and return it
            if (current.satisfy(problem.getGoal())) {
                return this.extractPlan(current, problem);
            }

            // Else we try to apply the actions of the problem to the current node
            else {
                for (int i = 0; i < problem.getActions().size(); i++) {
```

(continues on next page)

(continued from previous page)

```
                    // We get the actions of the problem
                    Action a = problem.getActions().get(i);
                    // If the action is applicable in the current node
                    if (a.isApplicable(current)) {
                        Node next = new Node(current);
                        // We apply the effect of the action
                        final List<ConditionalEffect> effects = a.
↪getConditionalEffects();
                        for (ConditionalEffect ce : effects) {
                            if (current.satisfy(ce.getCondition())) {
                                next.apply(ce.getEffect());
                            }
                        }
                        // We set the new child node information
                        final double g = current.getCost() + 1;
                        if (!close.contains(next)) {
                            next.setCost(g);
                            next.setParent(current);
                            next.setAction(i);
                            next.setHeuristic(heuristic.estimate(next, problem.
↪getGoal()));
                            open.add(next);
                        }
                    }
                }
            }
        }

        // Finally, we return the search computed or null if no search was found
        return plan;
    }
```

The method `extractPlan()` extracts a solution plan from the search space by backward chaining the path from the goal node to the root node. The code is given below:

```
    /**
     * Extracts a search from a specified node.
     *
     * @param node the node.
     * @param problem the problem.
     * @return the search extracted from the specified node.
     */
    private Plan extractPlan(final Node node, final ADLProblem problem) {
        Node n = node;
        final Plan plan = new SequentialPlan();
        while (n.getAction() != -1) {
            final Action a = problem.getActions().get(n.getAction());
            plan.add(0, a);
            n = n.getParent();
        }
        return plan;
    }
```

Finally, you have to change the call to the method `searchPlan()` in the method `solve()` by the explicit call to your `astar()` procedure. Your new `solve()` method is:

```
/**
 * Search a solution plan to a specified domain and problem using A*.
 *
 * @param problem the problem to solve.
 * @return the plan found or null if no plan was found.
 */
@Override
public Plan solve(final ADLProblem problem) {
    LOGGER.info("* Starting A* search \n");
    // Search a solution
    final long begin = System.currentTimeMillis();
    final Plan plan = this.astar(problem);
    final long end = System.currentTimeMillis();
    // If a plan is found update the statistics of the planner
    // and log search information
    if (plan != null) {
        LOGGER.info("* A* search succeeded\n");
        this.getStatistics().setTimeToSearch(end - begin);
    } else {
        LOGGER.info("* A* search failed\n");
    }
    // Return the plan found or null if the search fails.
    return plan;
}
```

7.7 Step 6. Make your planner configurable by programming

By default, your planner is configurable by programming (see MISSING REF for more details) because it inherits the class AbstractPlanner. But only for the common configurable properties of all planners, i.e., the domain, the problem, the timeout and the log level.

> **Note:** The common configurable properties and their values are defined in the interface Planner.

In order to allow the new properties of your planner to be configurable by programming, you have to :

1. declare for each new property a name and a default value
2. redefined the setter and the getter method to set and get the configuration of your planner
3. redefined a method `getDefaultConfiguration()`
4. redefined the method `hasValidConfiguration()`
5. redefined the constructors of your planner and deal with the class PlannerConfiguration

7.7.1 Step 6.1 Declaration of new properties

In the case of your planner, you have two new properties that you want configure: the heuristic and the weight the weight associated with it. The following code:

```
    /**
     * The HEURISTIC property used for planner configuration.
     */
    public static final String HEURISTIC_SETTING = "HEURISTIC";

    /**
     * The default value of the HEURISTIC property used for planner configuration.
     */
    public static final StateHeuristic.Name DEFAULT_HEURISTIC = StateHeuristic.Name.FAST_
↪FORWARD;

    /**
     * The WEIGHT_HEURISTIC property used for planner configuration.
     */
    public static final String WEIGHT_HEURISTIC_SETTING = "WEIGHT_HEURISTIC";

    /**
     * The default value of the WEIGHT_HEURISTIC property used for planner configuration.
     */
    public static final double DEFAULT_WEIGHT_HEURISTIC = 1.0;
```

7.7.2 Step 6.2 Setting and getting the configuration of your planner

To deal with the two properties and make your planner configurable by programming, it is necessary to redefined the setter and the getter of the class AbstractPlanner. This can be done in your case using the code below:

```
    /**
     * Returns the configuration of the planner.
     *
     * @return the configuration of the planner.
     */
    @Override
    public PlannerConfiguration getConfiguration() {
        final PlannerConfiguration config =  super.getConfiguration();
        config.setProperty(ASP.HEURISTIC_SETTING, this.getHeuristic().toString());
        config.setProperty(ASP.WEIGHT_HEURISTIC_SETTING, Double.toString(this.
↪getHeuristicWeight()));
        return config;
    }

    /**
     * Sets the configuration of the planner. If a planner setting is not defined in
     * the specified configuration, the setting is initialized with its default value.
     *
     * @param configuration the configuration to set.
     */
    @Override
```

(continues on next page)

(continued from previous page)

```
    public void setConfiguration(final PlannerConfiguration configuration) {
        super.setConfiguration(configuration);
        if (configuration.getProperty(ASP.WEIGHT_HEURISTIC_SETTING) == null) {
            this.setHeuristicWeight(ASP.DEFAULT_WEIGHT_HEURISTIC);
        } else {
            this.setHeuristicWeight(Double.parseDouble(configuration.getProperty(
                ASP.WEIGHT_HEURISTIC_SETTING)));
        }
        if (configuration.getProperty(ASP.HEURISTIC_SETTING) == null) {
            this.setHeuristic(ASP.DEFAULT_HEURISTIC);
        } else {
            this.setHeuristic(StateHeuristic.Name.valueOf(configuration.getProperty(
                ASP.HEURISTIC_SETTING)));
        }
    }
```

The code is quite simple. It call the method `getConfigration()` and `setConfiguration()` from the parent class AbstractPlanner. and set of get the new properties to an instance of the class PlannerConfiguration.

7.7.3 Step 6.3 Defining the default configuration of your planner

By convention all planner have a static method which returns the default configuration of a planner. In your case, the method `getDefaultConfiguration()` calls the eponymous method from the parent class AbstractPlanner. Then, in the same way as the previous method `getConfiguration()` it creates an instance of the class PlannerConfiguration with the default values.

```
    /**
     * This method return the default arguments of the planner.
     *
     * @return the default arguments of the planner.
     * @see PlannerConfiguration
     */
    public static PlannerConfiguration getDefaultConfiguration() {
        PlannerConfiguration config = Planner.getDefaultConfiguration();
        config.setProperty(ASP.HEURISTIC_SETTING, ASP.DEFAULT_HEURISTIC.toString());
        config.setProperty(ASP.WEIGHT_HEURISTIC_SETTING,
            Double.toString(ASP.DEFAULT_WEIGHT_HEURISTIC));
        return config;
    }
```

7.7.4 Step 6.4 Defining the method that checks if a configuration is valid or not

The `hasValideConfiguration()` method calls the eponymous method of the parent class AbstractPlanner and adds the checks on the added properties. For your planner, it checks that the weight associated to the heuristic is strictly greater than 0 and that a heuristic has been chosen among the GoalCostHeuristics already defined in the library

```
    /**
     * Checks the planner configuration and returns if the configuration is valid.
     * A configuration is valid if (1) the domain and the problem files exist and
     * can be read, (2) the timeout is greater than 0, (3) the weight of the
```

(continues on next page)

(continued from previous page)

```
     * heuristic is greater than 0 and (4) the heuristic is a not null.
     *
     * @return <code>true</code> if the configuration is valid <code>false</code>␣
↪otherwise.
     */
    public boolean hasValidConfiguration() {
        return super.hasValidConfiguration()
            && this.getHeuristicWeight() > 0.0
            && this.getHeuristic() != null;
    }
```

7.7.5 Step 6.5 Redefining the constructors of your planner

Redefining the constructors of your planner to create your planner from a PlannerConfiguration can be done with the code below:

```
    /**
     * Creates a new A* search planner with the default configuration.
     */
    public ASP() {
        this(ASP.getDefaultConfiguration());
    }

    /**
     * Creates a new A* search planner with a specified configuration.
     *
     * @param configuration the configuration of the planner.
     */
    public ASP(final PlannerConfiguration configuration) {
        super();
        this.setConfiguration(configuration);
    }
```

Note: The final code of the planner code is available here.

CHAPTER

EIGHT

USING THE PDDL PARSER

The library has a parser of the PDDL language. The language respects the 3.1 standard. To get the exact BNF (Backus–Naur form) of the PDDL language implemented in the library you can type the command:

```
./gradlew jjdoc
```

To access to the documentation generated open the file `./build/docs/PDDL4J_BNF/lexer.html` or simply by clicking on this link

8.1 Pre-requisite Installations

For this tutorial you need:

- Java JDK version 8 or higher is installed.
- A text editor such as Sublime or Atom or IDE such as Eclipse NetBean or IntelliJ.

In the following, we will give the commands line so that the tutorial can be done independently of any IDE.

8.2 Step 1. Create a simple Java project with PDDL4J

First, open a terminal and create your development directory

```
mkdir PDDLParserExample
```

Then, create the sub-directories of your project

```
cd PDDLParserExample
mkdir -p src/fr/uga/pddl4j/examples
mkdir classes
mkdir lib
```

Finally, get the last binary of PDDL4J and save it in the `lib` directory

```
wget http://pddl4j.imag.fr/repository/pddl4j/binaries/pddl4j-4.0.0.jar
mv pddl4j-4.0.0.jar lib/pddl4j-4.0.0.jar
```

8.3 Step 2. Create the main class of our example

Create and edit a file called `PDDLParserExample.java` in the directory `src/fr/uga/pddl4j/examples/`. The skeleton of this class is given bellow:

```
package fr.uga.pddl4j.examples;

import fr.uga.pddl4j.parser.ErrorManager;
import fr.uga.pddl4j.parser.Message;
import fr.uga.pddl4j.parser.PDDLParser;
import fr.uga.pddl4j.parser.ParsedProblem;

import java.io.FileNotFoundException;

/**
 * The class is an example class. It shows how to use the library to create a PDDL␣
↪parser and use it to parse PDDL
 * planning problem description.
 *
 * @author D. Pellier
 * @version 4.0 - 06.12.2021
 */
public class PDDLParserExample {

    /**
     * The main method the class. The first argument must be the path to the PDDL domain␣
↪description and the second
     * argument the path to the PDDL problem description.
     *
     * @param args the command line arguments.
     */
    public static void main(final String[] args) {

        // Checks the number of arguments from the command line
        if (args.length != 2) {
            System.out.println("Invalid command line");
            return;
        }

        try {
            // Creates an instance of the PDDL parser
            final PDDLParser parser = new PDDLParser();
            // Parses the domain and the problem files.
            final ParsedProblem parsedProblem = parser.parse(args[0], args[1]);
            // Gets the error manager of the parser
            final ErrorManager errorManager = parser.getErrorManager();
            // Checks if the error manager contains errors
            if (!errorManager.isEmpty()) {
                // Prints the errors
                for (Message m : errorManager.getMessages()) {
                    System.out.println(m.toString());
                }
            } else {
```

(continues on next page)

(continued from previous page)

```
                // Prints that the domain and the problem were successfully parsed
                System.out.print("\nparsing domain file \"" + args[0] + "\" done␣
↪successfully");
                System.out.print("\nparsing problem file \"" + args[1] + "\" done␣
↪successfully\n\n");
                // Print domain and the problem parsed
                System.out.println(parsedProblem.toString());
            }
            // This exception could happen if the domain or the problem does not exist
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

8.4 Step 3. Compile and Run the example

To test the above code use the following command line to compile the example:

```
javac -d classes -cp classes:lib/pddl4j-4.0.0.jar src/fr/uga/pddl4j/examples/
↪PDDLParserExample.java
```

and then the following command line to run the example:

```
java -cp classes:lib/pddl4j-4.0.0.jar fr.uga.pddl4j.examples.PDDLParserExample \\
    src/test/resources/benchmarks/pddl/ipc2000/logistics/strips-typed/domain.pddl \\
    src/test/resources/benchmarks/pddl/ipc2000/logistics/strips-typed/p01.pddl
```

CHAPTER

NINE

INSTANTIATING PLANNING PROBLEMS

The instantiation of a planning problem consists in transforming 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 or CSP for instance. However most planners use the instantiation to efficiently compute heuristics, speedup the search algorithm by using a compact encoding. In this tutorial, we present how to used PDDL4J to instantiate planning problems.

9.1 Pre-requisite Installations

For this tutorial you need:

- Java JDK version 8 or higher is installed.
- A text editor such as Sublime or Atom or IDE such as Eclipse NetBean or IntelliJ.

In the following, we will give the commands line so that the tutorial can be done independently of any IDE.

9.2 Step 1. Create a simple Java project with PDDL4J

First, open a terminal and create your development directory

```
mkdir ProblemInstantiationExample
```

Then, create the sub-directories of your project

```
cd ProblemInstantiationExample
mkdir -p src/fr/uga/pddl4j/examples
mkdir classes
mkdir lib
```

Finally, get the last binary of PDDL4J and save it in the `lib` directory

```
wget http://pddl4j.imag.fr/repository/pddl4j/binaries/pddl4j-4.0.0.jar
mv pddl4j-4.0.0.jar lib/pddl4j-4.0.0.jar
```

9.3 Step 2. Create the main class of our example

Create and edit a file called `ProblemInstantiationExample.java` in the directory `src/fr/uga/pddl4j/examples/`. The skeleton of this class is given bellow:

```
package fr.uga.pddl4j.examples;

import fr.uga.pddl4j.parser.ErrorManager;
import fr.uga.pddl4j.parser.Message;
import fr.uga.pddl4j.parser.PDDLParser;
import fr.uga.pddl4j.parser.ParsedProblem;
import fr.uga.pddl4j.problem.ADLProblem;
import fr.uga.pddl4j.problem.operator.Action;

import java.io.FileNotFoundException;

/**
 * The class is an example class. It shows how to use the library to create to ground planning problem.
 *
 * @author D. Pellier
 * @version 4.0 - 06.12.2021
 */
public class ProblemInstantiationExample {

    /**
     * The main method the class. The first argument must be the path to the PDDL domain description and the second
     * argument the path to the PDDL problem description.
     *
     * @param args the command line arguments.
     */
    public static void main(final String[] args) {

        // Checks the number of arguments from the command line
        if (args.length != 2) {
            System.out.println("Invalid command line");
            return;
        }

        try {
            // Creates an instance of the PDDL parser
            final PDDLParser parser = new PDDLParser();
            // Parses the domain and the problem files.
            final ParsedProblem parsedProblem = parser.parse(args[0], args[1]);
            // Gets the error manager of the parser
            final ErrorManager errorManager = parser.getErrorManager();
            // Checks if the error manager contains errors
            if (!errorManager.isEmpty()) {
                // Prints the errors
                for (Message m : errorManager.getMessages()) {
                    System.out.println(m.toString());
                }
```

(continues on next page)

(continued from previous page)

```
            } else {
                // Prints that the domain and the problem were successfully parsed
                System.out.print("\nparsing domain file \"" + args[0] + "\" done␣
↪successfully");
                System.out.print("\nparsing problem file \"" + args[1] + "\" done␣
↪successfully\n\n");
                // Create a ADL problem
                final ADLProblem problem = new ADLProblem(parsedProblem);
                // Instantiate the planning problem
                problem.instantiate();
                // Print the list of actions of the instantiated problem
                for (Action a : problem.getActions()) {
                    System.out.println(problem.toString(a));
                }
            }
            // This exception could happen if the domain or the problem does not exist
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

The first part of the main method parses the domain and the problem from the PDDL files (see for more details). If the parser succeeds, then a new ADLProblem is created from the parsed problem. Then, the method ``instantiate()``is called to instantiate the problem.

Warning: The call to the instantiate method can be long for complex problems.

The rest of the code print the list of actions of the instantiated problem.

9.4 Step 3. Compile and Run the example

To test the above code use the following command line to compile the example:

```
javac -d classes -cp classes:lib/pddl4j-4.0.0.jar src/fr/uga/pddl4j/examples/
↪ProblemInstantiationExample.java
```

and then the following command line to run the example:

```
java -cp classes:lib/pddl4j-4.0.0.jar fr.uga.pddl4j.examples.ProblemInstantiationExample␣
↪\\
    src/test/resources/benchmarks/pddl/ipc2000/logistics/strips-typed/domain.pddl \\
    src/test/resources/benchmarks/pddl/ipc2000/logistics/strips-typed/p01.pddl
```

CHAPTER

TEN

API DOCUMENTATION

The current version of the PDDL4J API Documentation is 4.0.0. Current and past API documentations of the library are available below:

- PDDL4J API Documentation is 4.0.0
- PDDL4J API Documentation is 3.8.3
- PDDL4J API Documentation is 3.8.2
- PDDL4J API Documentation is 3.8.1
- PDDL4J API Documentation is 3.8.0
- PDDL4J API Documentation is 3.7.3
- PDDL4J API Documentation is 3.7.2
- PDDL4J API Documentation is 3.7.1
- PDDL4J API Documentation is 3.7.0
- PDDL4J API Documentation is 3.6.0

The documentation of the PDDL and HDDL BNF syntax accepted by the planner is given below:

- PDDL4J BNF 4.0.0

CHAPTER

ELEVEN

DOWNLOAD

The simplest way to donwload PDDL4J is clone the project from GitHub. We also make available for download binary versions as well as the main releases. The latest version of the library is 4.0.0.

11.1 Binaries

- pddl4j.4.0.0.jar
- pddl4j.3.8.3.jar
- pddl4j.3.8.2.jar
- pddl4j.3.8.1.jar
- pddl4j.3.8.0.jar
- pddl4j.3.7.3.jar
- pddl4j.3.7.2.jar
- pddl4j.3.7.1.jar
- pddl4j.3.7.0.jar
- pddl4j.3.6.0.jar

11.2 Releases

- Release PDDL4J v3.8.3
- Release PDDL4J v3.8.2
- Release PDDL4J v3.8.1
- Release PDDL4J v3.7.2
- Release PDDL4J v3.7.1
- Release PDDL4J v3.7.0
- Release PDDL4J v3.6.0
- Release PDDL4J v3.5.0
- Release PDDL4J v3.1.0
- Release PDDL4J v3.0.0

- Release PDDL4J v2.0.0

CHAPTER

TWELVE

HOW TO CONTRIBUTE ?

To contribute to the project, it is important to consider the following procedure:

1. Fork PDDL4J repository
2. Checkout on devel branch
3. Create a new branch and work on it (optional)
4. Merge your work into devel branch (optional)
5. Create a merge request between your devel branch and pellierd/devel branch
6. As comments of your merge request, describe any changes or additions made
7. A contributor will review your code and accept/comment/reject it