pyperplan

Contents

1 Introduction

This is the documentation for pyperplan, a STRIPS planner built in the winter term 2010/2011 by the students of the AI planning practical course. It is written in Python3 and results showed that it is quite competitive when it comes to feature completeness. This document should help you to understand the planner's usage as well as the internal structure and where to extend it, if you want to.

2 Usage

To use the planner, you should have installed python3 on your system.

The planner's main program file is `pyperplan.py`. It takes at least one argument, the problem file. Currently only PDDL files are supported.

If you only pass a problem file, pyperplan will try to guess the domain file from the problem file's name. If the problem file contains a number, pyperplan will use a filename like `domainNUMBER.pddl` as a domain file candidate. If this file does not exist (or the problem name does not contain a number), it just uses `domain.pddl`. If this fails, an error will be given.

You can pass a specific domain file by using `pyperplan.py domainfile problemfile` to call the planner. **Note:** The domain comes before the problem, even though the domain is optional.

There are several command line arguments you can specify to define how pyperplan should solve the problem. By default, it uses breadth first search in the search space and the hff heuristic to estimate the goal distance (which is ignored by the breadth first search). By using the parameters -H or -s, you can select a different heuristic or search algorithm, respectively. If you leave one of them out, the default will be used. You can get a list of available search algorithms by executing `pyperplan.py --help`.

2.1 Solution

After a plan was found, it is saved in a file named just the same as the problem file but with a `.soln` appended.

A `soln` file contains actions in a LISP-like notation. The first element in each list is the action as taken from the domain specification, all the following elements are the arguments to the action. This format is a PDDL compatible plan representation that can be fed to PDDL validators to check if the plans are actually valid.

3 Internals

3.1 Logging

pyperplan uses the Logging facility for Python from the logging package that ships with any recent version of Python. Just add `import logging` in your file and use the function `logging.info(...)`. There are miscellaneous logging functions, just read the fine documentation of the logging package [1].

3.2 PDDL representation

The PDDL parsing and representation is in the package `pddl`, obviously. Currently both, the parser and the data structure, only support positive STRIPS, meaning that only positive preconditions are supported. For every element in a PDDL STRIPS problem or domain, there is a class in the `pddl.pddl` module. These classes are populated by the PDDL parser accordingly. The main parsing is done in `pddl.parser` where the `Parser` class is defined which provides methods for parsing a domain and a problem file. It parses the file by using a generic LISP parser that fills an abstract syntax tree. This tree is then traversed and nodes with PDDL-specific keywords are used to generate instances of the according class of from the `pddl.pddl` module. When more features are added to the STRIPS implementation, e.g. supporting the complete set of STRIPS features, it should be sufficient to modify the existing visitors.

3.3 Task and Grounding

After the domain and problem have both been parsed to an instance of `Domain` and `Problem` respectively, they pass the grounding step in which a concrete planning task is generated. The grounding is implemented in the module `grounding` and basically just takes a PDDL problem and returns a `Task` instance that represents the search task. This is a container class for a set of concrete Operator instances. The actions are no longer abstract definitions with types, but for any applicable object in the domain, there is an Operator that can be applied to the current state and that has specific results.

This is the crucial step before doing a search on the task. The `Task` class provides a function that helps building a search space: `get_successor_states` returns a list of all the possible states that can be reached using only valid operators. This creates a tree-like structure that, at some nodes, contain the goal state. The actual planning then consists in finding the shortest way to the goal state.

[1] http://docs.python.org/library/logging.html

3.4 Search

3.4.1 Summary

The search package contains a collection of search algorithms, like breadth first search or A*. The module searchspace contains a data structure SearchNode to create the search space, which stores information from the search and allows to extract the plan in a fast way.

3.4.2 Using the SearchNode class

The SearchNode class – in `searchspace.py` – is easy to use, just create a root node for the initial state of the planning task – use the function `searchspace.make_root_node(...)`. When applying an operator to a state you get a new state. This applied operator and the new state are stored in a new node, use the function `searchspace.make_child_node(...)` with the current search node as parent. This builds a tree-like structure.

In this way you can store the information from the search and, if you are in a goal state, you can read the plan in a nice way by calling `extract_solution()` on the goal node.

3.4.3 SAT planner

You can also find a SAT planner in the search package. It uses the minisat SAT solver to find a plan for a given problem. Pyperplan encodes the problem in a boolean formula in conjunctive normal form before it invokes minisat. To use the SAT planner you have to make the "minisat" executable available on the system PATH (e.g. /usr/local/bin/). Afterwards you can run the planner with the `--search sat` option. Please note that the executable is sometimes called "minisat2" in distribution packages (e.g. in some versions of Ubuntu). If that is the case you will have to rename the binary to "minisat".

3.4.4 Implementing new search algorithms

There is no base class to implement a new search algorithm, because each algorithm needs other arguments. So most of the algorithms are just single functions implemented in its own file. The naming convention is to name the module according to the algorithm and provide a function with a _search suffix that does the search. There should be no other side effects or things that need to be set up before doing the search. For convenient use, the `*_search()` function should be properly imported in the package's `__init__.py`.

3.5 Heuristics

3.5.1 Introduction

Most of the more advanced search algorithms are *informed* searches, which means they need information about how far a node is from the goal. Such information is taken from a heuristic which, given a node, returns how far the node is away from the goal state.

The heuristics in pyperplan are implemented as modules in the `heuristics` package.

3.5.2 Implementing new heuristics

For all the heuristics, there is a base class in the `heuristics.heuristic_base` package called `Heuristic`. Unlike search algorithms, which can be implemented as a single function, heuristics need to know something about the task and are hence a class. The class' constructor takes the planning task as an argument and can apply preprocessing of states if necessary. In most of the cases you want to at least store a reference to the goal state.

Each descendant of heuristic **must** implement the `__call__` magic method which has to take a node and return the estimated distance from the goal. For nodes that are known to be deadends (i.e. there is no valid plan that reaches the goal from this node), a heuristic value of `float('inf')` should be returned.

Pyperplan automatically finds all heuristics classes that reside in modules in the `heuristics` folder if the class name ends with "Heuristic".

4 Tests

A comprehensive unit test suite is available. It can be run with the py.test framework (http://pytest.org). To run the tests, install py.test for Python3.

4.1 Requirements

Ubuntu <= 12.04:

```
sudo apt-get install python3-setuptools
sudo easy_install3 -U pytest
```

Ubuntu >= 12.10:

```
sudo apt-get install python3-pip
sudo pip-3.x install -U pytest
```

4.2 Run tests

Change into the `src` directory and run `py.test-3.x`. Some of the tests have been marked as "slow" and will only be run if you pass py.test the `--slow` parameter.

5 Plan validation

If the plan validation tool VAL is found on the system PATH, all found plans will be validated. To use this feature

- Download VAL from http://planning.cis.strath.ac.uk/VAL/.
- Execute `make` in the extracted archive.
- Copy the executable `validate` to a directory on the path (e.g. /usr/local/bin/).
- If any problems occur during compilation, please consult VAL's homepage.