Principles of AI Planning

R. Mattmüller, Prof. Dr. B. Nebel
N. Mayer
Winter Term 2012/2013

University of Freiburg
Department of Computer Science

Project 1

Due: January 18th, 2013

In the project exercises, we will both use and alter `pyperplan`. As `pyperplan` is written in Python 3, all exercises can only be implemented in Python 3. All solutions should be sent by email to `mayern@informatik.uni-freiburg.de` and will be tested to verify their functionality.

Exercise 1.1 (STRIPS regression, 8 points)

In this part of the project, we will implement STRIPS regression search in `pyperplan`. Start out with a fresh copy of `pyperplan`:

```
hg clone https://bitbucket.org/malte/pyperplan
```

(a) Implement a module `src/search/regression.py` with a main function named `regression_search(task)` that performs regression breadth-first search as described in the lecture. Make sure to implement the optimization to only consider operators making at least one needed subgoal true. For the representation of search nodes, you can import and use the class `SearchNode` and the functions `make_root_node` and `make_child_node` provided in the module `src/search/searchspace.py`.

(b) Experiment with duplicate elimination and subsumption checking during regression search. E.g., if you encounter a conjunction $c = \varphi_1 \wedge \cdots \wedge \varphi_n$ such that another conjunction $\varphi_{i_1} \wedge \cdots \wedge \varphi_{i_k}$ with $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ of a subset of the conjuncts in c is already present in the closed list (or the open list), you can safely prune the search node representing the larger conjunction c. Investigate how this affects search performance in terms of node expansions and running times in the domains `blocks` and `logistics`. Report your findings by sending a table like the following with experimental results to `mayern@informatik.uni-freiburg.de`

	Duplicate elim.	Subs. checking	neither	both
Node expansions				
Running time				

Hint: In order to integrate your implementation into `pyperplan`, you still need to do the following:

(i) In the file `src/search/__init__.py`, add the following line:

```
from .regression import regression_search
```

(ii) In the `SEARCHES` dictionary in the file `src/pyperplan.py`, add the following entry:

```
'regr': search.regression_search
```

(iii) In the file `src/pyperplan.py`, change the conditional

```
if args.search in ['bfs', 'ids', 'sat']: heuristic = None
```

such that for `'regr'` search, no heuristic is used, either.

The first modification makes the `regression_search` function visible to `pyperplan`, the second adds it to the search algorithms available from the command line under the name `'regr'`, and the third one makes sure that `pyperplan` does not try to provide the regression search with a heuristic, which would be useless for a breadth-first search.

Eventually, you should be able to call `pyperplan` like this:

```
./pyperplan.py -s regr ../benchmarks/blocks/{domain.pddl,task01.pddl}
```

Bonus exercise (no marks): Investigate appropriate representations of the closed list that support the subsumption test mentioned above more efficiently than a proper list or array holding all states ever encountered during search.

Exercise 1.2 (Pattern-database heuristics, 12 points)

In this exercise, we will implement pattern-database heuristics in `pyperplan`. Since automatically selecting appropriate patterns (or pattern collections) is non-trivial, we will assume that pattern collections are handed to `pyperplan` along with the planning task to be solved.

Implement a module `src/heuristics/pattern_database.py` containing a class `PDBHeuristic` that is derived from the base class `Heuristic` from the module `src/heuristics/heuristic_base.py`. `PDBHeuristic` should implement the method `__call__(self, node)` which takes a search node and returns its heuristic value.

Make sure that when the `PDBHeuristic` is asked for a heuristic value, the PDBs containing optimal abstract goal distances have already been computed. This is most easily accomplished by initiating the computation of the PDBs in the constructor of the class `PDBHeuristic`. The constructor should take two arguments, a planning task `task` and a list of patterns `patterns`, where each pattern in this list should be a set of atomic propositions.

In more detail, the following has to be implemented as part of `PDBHeuristic`:

(a) In the preprocessing step used to compute PDBs, compute the abstract planning tasks of the given task for all patterns, i.e., perform the syntactic abstractions (projections) that discard variables outside the respective pattern as described in the lecture. You should end up with one abstract planning task for each pattern.

(b) For each abstraction, perform regression breadth-first search in the induced transition system of that particular abstract task, keeping track of the shortest abstract goal distances of all abstract states encountered. You can stop the regression search as soon as no more new abstract states are found. Backward-unreachable abstract states get a heuristic value of infinity.

Hint 1: You can probably reuse much of the STRIPS regression code from the previous exercise for the PDB computation.

Hint 2: When dealing with abstract states and with subgoals representing several abstract states at once, you may find the `product` and `combinations` functions from Python's `itertools` module helpful.

(c) Build one PDB for each abstraction and store the abstract goal distances in these PDBs.

(d) Implement the PDB lookup needed when `__call__(self, node)` is called. When returning heuristic values, you may assume that the given patterns are additive, i.e., you should return the *sum* of the heuristic values from the various PDBs.

Hint: In order to integrate your implementation into `pyperplan`, you still need to do the following:

(i) In the file `src/heuristics/__init__.py`, add the following line:

```
from .pattern_database import PDBHeuristic
```

(ii) In the `HEURISTICS` dictionary in the file `src/pyperplan.py`, add the following entry:

```
'pdb': heuristics.PDBHeuristic
```

(iii) Read the pattern collection from the file system. The file containing the patterns should consist of one pattern per line, where each pattern/line is a comma-separated list of atomic propositions (containing the surrounding parentheses, and completely in lower case). The file containing the patterns should reside in the same directory as the problem file to which it belongs, and should also be named the same, plus the suffix `'.patterns'`. E.g., the pattern collection file for the problem `benchmarks/blocks/task01.pddl` should be named `benchmarks/blocks/task01.pddl.patterns` and could have the following content:

```
(ontable a), (holding a), (on a b), (on a c), (on a d)
(ontable b), (holding b), (on b a), (on b c), (on b d)
(ontable c), (holding c), (on c a), (on c b), (on c d)
(ontable d), (holding d), (on d a), (on d b), (on d c)
```

This means that there are four pattern, one for the state (or position) of each of the four blocks `a`, `b`, `c`, and `d`.

In order to inform the `PDBHeuristic` about the pattern collection, you still need to perform the following modifications of the file `src/pyperplan.py`:

(A) Add a function `_parsePatterns(pattern_file)` that returns a list of sets of atomic propositions corresponding to the lines in the file `pattern_file`.

(B) Replace the conditional

```
if not heuristic_class is None:
    heuristic = heuristic_class(task)
```

by

```
if not heuristic_class is None:
    if heuristic_class is heuristics.PDBHeuristic:
        patterns = _parsePatterns('%s.patterns' % problem_file)
        heuristic = heuristic_class(task, patterns)
    else:
        heuristic = heuristic_class(task)
```

Evaluate your implementation in the blocksworld domain (and an arbitrary number of additional domains of your choice). For the blocksworld instances coming with `pyperplan`, generate appropriate additive pattern collections (leading to an admissible heuristic), and run the A* search of `pyperplan` using these pattern collections. Compare the performance with the performance of the `blind` heuristic and the `hff` heuristic implemented in `pyperplan` in terms of running times and numbers of node expansions. Report your findings by sending a table with experimental results to `mayern@informatik.uni-freiburg.de`.

Problem	Criterion	`pdb`	`hff`	`blind`
Blocksworld 1	Node exp.			
	Running time			
⋮	⋮	⋮	⋮	⋮

Eventually, you should be able to call `pyperplan` like this:

```
./pyperplan.py -s astar -H pdb ../benchmarks/blocks/{domain.pddl,task01.pddl}
```

Bonus exercise 1 (no marks): Implement an automatic pattern selection algorithm.

Bonus exercise 2 (no marks): Make it configurable whether to use the sum or the maximum over the component heuristics for aggregation.

Note: The exercise sheets may and should be worked on in groups of two students. Please state both names on your solution (this also holds for submissions by e-mail).