Principles of AI Planning

R. Mattmüller, Prof. Dr. B. Nebel
T. Keller
Winter Term 2011/2012

University of Freiburg
Department of Computer Science

Project 3

Due: February 17th, 2012

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 `tkeller@informatik.uni-freiburg.de` and will be tested to verify their functionality.

Exercise 3.1 (Strong planning using AO* search, 8+6+4+2 points)

In this exercise, we implement AO* tree search to find strong plans for fully observable nondeterministic planning problems. We prepared a modified version of `pyperplan` that is able to parse and ground nondeterministic planning tasks. In PDDL, nondeterministic outcomes are encoded using the `oneof` keyword. In general, the nesting of the `oneof` keyword in effects is not restricted. In this exercise, however, we assume that all effects are either deterministic STRIPS effects e (in the form already supported by pyperplan, i.e., sequences of atomic add and delete effects, possibly conjoined by `and`) or sets of such effects, $\{e_1, \dots, e_n\}$, each element e_i of the set being one possible nondeterministic outcome. These sets are encoded as

$$\texttt{(oneof } e_1 \dots e_n\texttt{)}.$$

Download the `pyperplan` variant capable of reading such PDDL inputs from the class website. We prepared problem generators for two very simple nondeterministic planning domains, the `coinFlip` domain and the `chainOfRooms` domain. You can find an archive containing the two domain files and two problem generators written in Python on the class website. Both generators take exactly one argument, which is the problem size (the number of coins in the `coinFlip` domain and the number of rooms in the `chainOfRooms` domain). Download the archive and familiarize yourself with the domains.

(a) Implement AO* tree search in `pyperplan` to find strong plans for nondeterministic planning problems. To keep the algorithm simple, do not perform duplicate detection, but rather represent one state by several nodes in the search tree, if the same state is encountered along more than one path. This allows for a simple dynamic programming update procedure as well as a simple extraction of the currently most promising partial solution graph.

(b) Implement an interface between your AO* implementation and the heuristics defined in `pyperplan` for deterministic problems by applying the heuristics from the deterministic setting to the *all-outcomes determinization* of the planning task at hand. The all-outcomes determinization of a nondeterministic planning task is the deterministic planning task with the same variables, initial state and goal description, and with a set of operators that contains one operator for each operator-outcome pair in the nondeterministic task. E.g., if the nondeterministic task contains an operator $\langle \chi, \{e_1, \dots, e_n\} \rangle$, the all-outcomes determinization will contain the n operators $\langle \chi, e_1 \rangle$, ..., $\langle \chi, e_n \rangle$.

(c) To test the correctness of your implementation, implement the strategy output format defined below and verify the strategies you find using the `verify` tool.

The following description of the strategy output format is cited from Daniel Bryce and Olivier Buffet, "6th International Planning Competition":

In the output language, the file contains three sections separated by "%%":

```
<n> <atom-list>
%%
<m> <action-list>
%%
<plan>
```

where `<n>` is an integer, possibly 0, denoting the size of `<atom-list>` which is a space-separated list of atoms such as "`(on A B)`", `<m>`, possibly 0, is the size of `<action-list>` which is a space-separated list of operators such as "`(move A C B)`", and `<plan>` is the representation of the plan. [...]
For fully-observable non-deterministic problems, `<plan>` [...] is of the form:

```
policy <k> <map-list>
```

where `<k>` is the size of `<map-list>` which is a space-separated list of variable-sized elements. The elements of `<map-list>` define a partial function mapping states into actions. Each element is of the form:

```
<l> <atom-list> <action-index>
```

where `<l>` is the size of `<atom-list>` which is a space-separated list of integers in $\{0, \ldots, n-1\}$, each denoting the atom with such index, and `<action-index>` is an integer in $\{0, \ldots, m-1\}$ denoting the action with such index. Such element defines the mapping from the unique state that makes all and only all atoms in `<atom-list>` true into the action with appropriate index. For example, the file:

```
4 (on A B) (clear A) (clear B) (on B A)
%%
4 (pick A) (putdown A) (pick B) (putdown B)
%%
policy 2 2 0 1 0 2 3 2 2
```

denotes the policy π such that $\pi(s)$ and $\pi(s')$ are "`(pick A)`" and "`(pick B)`" respectively, and $s = \{\texttt{(on A B)}, \texttt{(clear A)}\}$ and $s' = \{\texttt{(on B A)}, \texttt{(clear B)}\}$.

(d) Compare the performance of your AO* implementation on `coinFlip` and `chainOfRooms` problems of different sizes using the FF-heuristic from `pyperplan`. Report results for all problems of a size still solvable within a 2 minute timeout.

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).