Plan Verification

pandaPIparser contains a plan verifier for HTN planning problems formulated in HDDL (technically, the syntax to formulate solution plans is not part of the HDDL standard, but we extended it accordingly).

In order to run the verifier you have to provide pandaPIparser with the domain and problem instance as well as the plan you want to verify. You must provide them as three separate text files. To run the verifier, invoke pandaPIparser as follows

`./pandaPIParser -verify [DOMAIN] [PROBLEM] [PLAN]`

Instead of `-verify` you may specify `-vverify` (verbose verification) or `-vvverify` (very verbose verification) to get a more detailed output. Please note that this output may be *very* large, even for simple planning problems. However the output might be helpful if pandaPIparser rejects the plan.

Overview of the Plan Format

pandaPIparser is intended to read the full output of a planner and extract the plan from it. To do so it assumes that the planner will output the plan it found as its last output. No output may be produced after the plan. To indicate the start of the plan, pandaPIparser searches for the string `==>` in the input. Everything before that string will be ignored. You may not output any non-whitespace character after `==>` until the line ends. It is preferred that you put a line break immediately after `==>`.

After `==>`, pandaPIparser will interpret the rest of the planner's output line-based. The lines directly after `==>` will describe the derived primitive plan. You have to provide the actions of the plan in the order they are executed in. This section of the plan ends with a line starting (except for preceding whitespace characters) with `root`.

Each line describing a primitive action consists of three elements, which are separated by spaces (or more or other whitespace characters).

1. the action's ID
2. the action's name
3. the action's parameters

The action's ID may be any non-negative integer. It will be used to refer to the action when describing the decompositions used to obtain the plan. No two actions are allowed to have the same ID.

You must output the action's name as provided in the HDDL input.

Lastly, you must provide the arguments of the action. The arguments are separated by whitespace characters. For each parameter variable of the action, you have to specify the value it is assigned to. You have to provide these values in the order in which the variables appear in the action's parameter list. You may not add additional arguments or leave an argument blank.

Alternatively, you can provide the action's name and parameters in one of the following formats:

1. `(NAME ARG1 ARG2 .. ARGN)`
2. `NAME [ARG1,ARG2,..,ARGN]`

As an example, the beginning of a valid plan in the `transport` domain may look as follows:

```
==>
0 drive truck-0 city-loc-2 city-loc-1
1 pick-up truck-0 city-loc-1 package-0 capacity-0 capacity-1
2 drive truck-0 city-loc-1 city-loc-0
3 drop truck-0 city-loc-0 package-0 capacity-0 capacity-1
4 drive truck-0 city-loc-0 city-loc-1
5 pick-up truck-0 city-loc-1 package-1 capacity-0 capacity-1
6 drive truck-0 city-loc-1 city-loc-2
7 drop truck-0 city-loc-2 package-1 capacity-0 capacity-1
```

Decomposition

After the plan has specified the primitive actions it contains, the plan must also specify the decompositions that were applied in order to obtain these primitive actions. This section starts with a line declaring the root tasks. It contains the string `root` followed by a non-empty list of IDs separated by whitespace characters. We will describe the semantics of these IDs below.

Afterwards follow lines describing the applied decompositions. These first specify an abstract task that was decomposed, then the method applied to decompose it and lastly the tasks that resulted from applying the method. The first part of each such line – describing an abstract task – has exactly the same format as a line describing a primitive action. I.e. you may treat the abstract task for outputting it as if it was primitive. The IDs for abstract tasks must be distinct from those of the primitive actions.

To signal the end of the argument list of the abstract task, you have to output the string `->` separated with whitespace characters. After the `->` you have to provide the name of the method applied to the abstract task as it appeared in the HDDL domain. Lastly, you have to provide the tasks that were inserted due to the application of the decomposition. This is done by providing a whitespace-separated list of IDs of the inserted tasks. For each of these IDs, there must be a task in the input that has this ID.

You have to specify as many IDs as the method has subtasks declared in the input. Each of the IDs will correspond to one of the subtasks of the method.

If the method is totally-ordered, the ith ID will be that of the ith subtask. If the method is partially-ordered, you may output the IDs of the subtasks in any topological order of the subtasks. pandaPIparser will check any mapping of IDs to subtasks declared by the method as long as it is compatible with the declared task names and arguments. Note that this is necessary, as there is no means to uniquely determine an ordering of the subtasks of partially ordered methods. Not all input formats that HDDL can be compiled into allow for labels for the subtasks and some languages require a different order of appearance of the tasks in the domain. Thus pandaPIparser is as lax as possible. If the mapping is still unique (using task types and parameter values), it will be able to provide more helpful output. If not, you may have to refer to the verbose output if the plan is not correctly verified to determine the actual cause(s) of the error.

At the beginning of the decomposition section, you have to specify the root tasks. These are the IDs of the tasks that occur in the problem's initial task network. You may provide them in any topological ordering that is compatible with the order imposed on the tasks in the initial task network.

Instead of providing the IDs of the tasks in the initial task network, you may also add an artificial 'root' task and provide its ID as the root ID. It must be named `__top`. Its decomposition method must be named `__top_method` and is decomposed into the tasks of the initial task network, in any topological ordering.

As an example for specifying the decomposition of a plan, consider the following part of a solution to a `transport` problem.

```
root 15 14
8 load truck-0 city-loc-1 package-0 -> m-load 1
9 unload truck-0 city-loc-0 package-0 -> m-unload 3
10 get-to truck-0 city-loc-1 -> m-drive-to 0
11 unload truck-0 city-loc-2 package-1 -> m-unload 7
12 get-to truck-0 city-loc-0 -> m-drive-to 2
13 load truck-0 city-loc-1 package-1 -> m-load 5
14 deliver package-0 city-loc-0 -> m-deliver 10 8 12 9
15 deliver package-1 city-loc-2 -> m-deliver 16 13 17 11
16 get-to truck-0 city-loc-1 -> m-drive-to 4
17 get-to truck-0 city-loc-2 -> m-drive-to 6
```

End of Input

The plan either ends with the end of the planners output or the characters `<==`. Any output after `<==` will be ignored by the verifier. Also, you may print any whitespace character after the end of the plan.

Known Issues

For some specifically modelled domains, verification may be very slow. This issue arises if the domain contains an action (or method) that has an existentially quantified precondition with several variables. Since the plan does not contain any information on the values of these variables used by the planner in the plan, pandaPIparser must try to determine this value itself. In order to perform as little transformation as possible on the input pandaPIparser has no hint on which values to try and thus has to instantiate all combinations of variables. If this number is high, verification will take time, but will succeed.