Fast Downward with Logging

1 Fast downward with logging

Fast Downward is a very configurable open source planner that does forward-chaining state space search. For example, it can emulate the search done by FF and does contain the code for LAMA2011 (in fact LAMA2011 was a fork from Fast Downward to begin with). However, like most planners it does not produce a lot of information about how the search was conducted. Therefore, this fork was created to log a lot of the information about the search process. This log can thereafter be sent to the Fast Downward Visualiser to show the search process that Fast Downward conducted. An unfortunate side effect of all the logging is that it is invasive which means that the planner takes considerable longer to solve problems.

2 Running the planner

The planner with logging can be run in the exact same way as the original planner (as of summer 2017). However, not all search engines, more on that later, are fully tested and the same applies to heuristics and open lists. Nevertheless, the planner can be executed with the following command:

```
$ fast-downward.py DOMAIN PROBLEM CONFIGURATION FLAGS
```

On the Linux lab computers at LiU / IDA:

```
$ /home/TDDC17/sw/fdlog/fast-downward.py DOMAIN PROBLEM CONFIGURATION
   FLAGS
```

Where DOMAIN and PROBLEM are the domain file and the problem file respectively. CONFIGURATION is a description of how the planner will search. As written in the previous section, the planner can be configured in many ways and therefore we refer to the website for fast downward (http://www.fast-downward.org) for the full description of the possible configuration. However, some of the basics will be covered in the following sections. This includes the terms Search engine, Evaluator, Open List, Heuristics and flags.

2.1 Flags

The planner should support most of the flags that are available in the original version of the planner. For more documentation abou those flags please see the original documentation. In addition the planner supports the following extra flags:

- –compress: Causes the planner to compress the log file to gzip format. This is recommended for all executions since the size of the log is an NP-function of the problem size.

2.2 Search engine

Fast Downward supports multiple search engines (also commonly referred to as search methods in other contexts) that are specified in the configuration with:

```
--search NAME(PARAMETERS)
```

A lot of the configurations for the search engine haven't been verified that the properly log the search. These have all been marked with "see original documentation". Lists are written in a command as [item1, item2, ...].

*2.2.1 A**

A* is a common search method that guarantees an optimal solution given that an admissible heuristic is used. This algorithm is a special case of the eager search. The engine can be used with the following:

```
--search astar(eval, mpd=false, pruning=null(), cost_type=NORMAL, bound=
   infinity, max_time=infinity)
```

Where the parameters are as follows:

- eval: an Evaluator.
- mdp: see original documentation.
- pruning: see original documentation.
- cost_type: see original documentation.
- bound: see original documentation.
- max_time: see original documentation.

2.2.2 Eager best-first search

Eager best-first search is an algorithm that visits the node with the best estimation from all the expanded nodes in each step in the search. Hence, A* can be acquired by defining the best as g+h. The search engine can be used with the following:

```
--search eager(open, reopen_closed=false, f_eval=<none>, preferred=[],
   pruning=null(), cost_type=NORMAL, bound=infinity, max_time=infinity)
```

Where the parameters are as follows:

- open: an Evaluator.
- reopen_closed: true if the planner should revisit search node when it finds a better way to it.
- f_eval: see original documentation.

- preferred: a list of heuristics that is used to determine if an operator is a preferred operator in a state (that is given higher priority than the other operators used in that state).
- pruning: see original documentation.
- cost_type: see original documentation.
- bound: see original documentation.
- max_time: see original documentation.

2.2.3 Greedy search (eager)

This is really similar to the eager search and can in most cases be seen as an eager search but with multiple evaluators that it selects from in an alternating manner (see Alternating open list). It can be used as follows:

```
--search eager_greedy(evals, preferred=[], boost=0, pruning=null(),
   cost_type=NORMAL, bound=infinity, max_time=infinity)
```

Where the parameters are as follows:

- evals: a list of Evaluators.
- preferred: a list of Heuristics.
- boost: How many extra (in addition to the alternative behaviour) operators will be selected from the preferred queues before the others will be considered again.
- pruning: see original documentation.
- cost_type: see original documentation.
- bound: see original documentation.
- max_time: see original documentation.

2.2.4 Lazy enforced hill-climbing

This search engine works in phases which each consists of breadth first search. A phase ends when the breadth first search reaches a state that has a lower heuristic value than the originating state for that phase. The found state will then be the originating state for the next phase. The search will not expand all the reachable states from a search state instead it will only note that those state exists and expand it when it is about to visit the state (this is the laziness). One uses the search engine with the following command:

```
--search ehc(h, preferred_usage=PRUNE_BY_PREFERRED, preferred=[],
   cost_type=NORMAL, bound=infinity, max_time=infinity)
```

Where the parameters are as follows:

- h: the Heuristic that is to be used.

- preferred_usage: see original documentation.
- preferred: see original documentation.
- cost_type: see original documentation.
- bound: see original documentation.
- max_time: see original documentation.

2.2.5 Iterated search

See original documentation.

2.2.6 Lazy best-first search

The search engine works by always selecting the most recently visited state with the lowest heuristic value and at least one possible applicable operator not tested in that state before. Given that node, an operator is selected to expand a new search node (which might lead to an already known state) which is directly visited and the whole thing repeats. Note that this is lazy since the visitation of a node happens directly after it's expanded. The search engine can be used as follows:

```
--search lazy(open, reopen_closed=false, preferred=[],
   randomize_successors=false, preferred_successors_first=false,
   random_seed=-1, cost_type=NORMAL, bound=infinity, max_time=infinity)
```

Where the parameters are as follows:

- evals: a list of Evaluators (if multiple Evaluators or a preferred queue exists then an Alternating open list is used otherwise a standard open list).
- preferred: a list of Heuristics.
- reopen_closed: see original documentation.
- boost: How many extra (in addition to the alternative behaviour) operators will be selected from the preferred queues before the others will be considered again.
- randomize_successors: see original documentation.
- preferred_successors_first: see original documentation.
- random_seed: see original documentation.
- cost_type: see original documentation.
- bound: see original documentation.
- max_time: see original documentation.

2.2.7 (Weighted) A (lazy)*

see original documentation.

2.3 Evaluator

Evaluators are either simple arithmetic operations that barely cost anything in terms of computing power or, potentially, more costly Heuristics. The simple arithmetic operations are as follows:

2.3.1 g-value

Returns the g-value (path cost) of a search node. Usage:

```
g()
```

2.3.2 Max evaluator

Returns the max of the given evalutors. Usage:

```
max(evals)
```

Where the parameters are:

- evals: a list of Evaluators.

2.3.3 Preference evaluator

See original documentation.

2.3.4 Sum evaluator

Returns the sum of all the given evaluators. Usage:

```
sum(evals)
```

Where the parameters are:

- evals: a list of Evaluators.

2.3.5 Weighted evaluator

Returns the multiplied value of an evaluator and a constant. Usage:

```
weight(eval, weight)
```

Where the parameters are:

- eval: an Evaluator.
- weight: the weight of the evaluator.

2.4 Heuristic

Heuristic is a function that estimates the cost to reach the goal from a current state. In Fast downward these are a subclass of Evaluators and can be used instead of an Evaluator. Sometimes one wants to use the same heuristic value in multiple places and it would be unnecessary to calculate it multiple times. Therefore, it's possible to pre-calculate a heuristic value for a state by:

```
--heuristic name=Heuristic
```

Where Heuristic is the call to a heuristic. Thereafter, the heuristic can be used as a parameter to, for example, the search engine by using the name. A heuristic can have the following properties in Fast Downward:

- Admissible: $h(s) \leq h^*(s)$ for all states s.
- Consistent: $h(s) \leq c(s, s') + h(s')$ for all states s connected to state s' by an action with cost $c(s, s')$.
- Safe: $h(s) = \infty$ is only true for states with $h^*(s) = \infty$
- Preferred operators: The heuristic identifies preferred operators.

2.4.1 Additive heuristic

```
add(transform=no_transform(), cache_estimates=true)
```

Where:

- transform: see original documentation.
- cache_estimates: see original documentation.

Supports language features:

- Action costs
- Conditional effects
- axioms

The heuristic has the following properties:

- Safe for tasks without axioms
- Preferred operators

2.4.2 Potential heuristic optimized for all states

See original documentation.

2.4.3 *Blind heuristic*

Returns the cost of the cheapest action for non-goal states and 0 for goal-states.

```
blind(transform=no_transform(), cache_estimates=true)
```

Where:

- transform: see original documentation.
- cache_estimates: see original documentation.

Supports language features:

- Action costs
- Conditional effects
- axioms

The heuristic has the following properties:

- Admissible
- Consistent
- Safe

2.4.4 *Context-enhanced additive heuristic*

```
cea(transform=no_transform(), cache_estimates=true)
```

Where:

- transform: see original documentation.
- cache_estimates: see original documentation.

Supports language features:

- Action costs
- Conditional effects
- axioms

The heuristic has the following properties:

- Preferred operators

2.4.5 *Additive CEGAR heuristic*

```
cegar(subtasks=[landmarks(),goals()], max_states=infinity,
   max_transitions=1000000, max_time=infinity, pick=MAX_REFINED,
   use_general_costs=true, transform=no_transform(), cache_estimates=true
   , random_seed=-1)
```

Where:

- subtasks: see original documentation
- max_states: see original documentation.
- max_transitions: see original documentation.
- max_time: see original documentation.
- pick: see original documentation.
- use_general_cost: see original documentation.
- transform: see original documentation.
- cache_estimates: see original documentation.
- random_seed: see original documentation.

Supports language features:

- Action costs

The heuristic has the following properties:

- Admissible
- Consistent
- Safe

2.4.6 Causal graph heuristic

```
cg(transform=no_transform(), cache_estimates=true)
```

Where:

- transform: see original documentation.
- cache_estimates: see original documentation.

Supports language features:

- Action costs
- Conditional effects
- axioms

The heuristic has the following properties:

- Preferred operators

2.4.7 Constant evaluator

Returns a constant value.

```
const(value=1,transform=no_transform(), cache_estimates=true)
```

Where:

- value: the value to return.
- transform: see original documentation.
- cache_estimates: see original documentation.

The heuristic has the following properties:

- Consistent

2.4.8 Canonical PDB

A pattern database heuristic, see original documentation for in depth description.

```
cpdbs(patterns=systematic(1), dominance_pruning=true, transform=
   no_transform(), cache_estimates=true)
```

Where:

- patterns: see original documentation.
- dominance_pruning: see original documentation.
- transform: see original documentation.
- cache_estimates: see original documentation.

Supports language features:

- Action costs

The heuristic has the following properties:

- Admissible
- Consistent
- Safe

2.4.9 Diverse potential heuristics

See original documentation.

2.4.10 FF heuristic

```
ff(transform=no_transform(), cache_estimates=true)
```

Where:

- transform: see original documentation.
- cache_estimates: see original documentation.

Supports language features:

- Action costs
- Conditional effects
- axioms

The heuristic has the following properties:

- Safe for task without axioms
- Preferred operators

2.4.11 Goal count heuristic

```
goalcount(transform=no_transform(), cache_estimates=true)
```

Where:

- transform: see original documentation.
- cache_estimates: see original documentation.

Supports language features:

- Conditional effects
- axioms

The heuristic has the following properties:

- Safe

2.4.12 h^m heuristics

```
hm(m=2, transform=no_transform(), cache_estimates=true)
```

Where:

- m: subset size [1, $infty$].
- transform: see original documentation.
- cache_estimates: see original documentation.

Supports language features:

- Action costs

The heuristic has the following properties:

- Admissible for tasks without conditional effects or axioms.

- Consistent for tasks without conditional effects or axioms.
- Safe for tasks without conditional effects or axioms.

2.4.13 Max heuristic

```
hmax(transform=no_transform(), cache_estimates=true)
```

Where:

- transform: see original documentation.
- cache_estimates: see original documentation.

Supports language features:

- Action costs
- Conditional effects
- axioms

The heuristic has the following properties:

- Admissible for tasks without axioms.
- Consistent for tasks without axioms.
- Safe for tasks without axioms.

2.4.14 Potential heuristic optimized for initial state

See original documentation.

2.4.15 iPDB

See original documentation for description of how the heuristic works.

```
ipdb(pdb_max_size=2000000, collection_max_size=20000000, num_samples
   =1000, min_improvement=10, max_time=infinity, random_seed=-1,
   dominance_pruning=true, transform=no_transform(), cache_estimates=true
   )
```

Where:

- pdb_max_size: see original documentation.
- collection_max_size: see original documentation.
- num_samples: see original documentation.
- min_improvement: see original documentation.
- max_time: see original documentation.

- random_seed: see original documentation.
- dominance_pruning: see original documentation.
- transform: see original documentation.
- cache_estimate: see original documentation.

Supports language features:

- Action costs

The heuristic has the following properties:

- Admissible
- Consistent
- Safe

2.4.16 Landmark-count heuristic

See original documentation.

2.4.17 Landmark-cut heuristic

```
lmcut(transform=no_transform(), cache_estimates=true)
```

Where:

- transform: see original documentation.
- cache_estimates: see original documentation.

Supports language features:

- Action costs

The heuristic has the following properties:

- Admissible
- Safe

2.4.18 Merge-and-shrink heuristic

See original documentation for in depth description of the heuristic.

```
merge_and_shrink(merge_strategy, shrink_strategy, label_reduction=<none>,
    prune_unreachable_states=true, prune_irrelevant_states=true,
   max_states=-1, max_states_before_merge=-1, threshold_before_merge=-1,
   transform=no_transform(), cache_estimates=true, verbosity=verbose)
```

Where:

- merge_strategy: see original documentation.
- shrink_strategy: see original documentation.
- label_reduction: see original documentation.
- prune_unreachable_states: see original documentation.
- max_states: see original documentation.
- max_states_before_merge: see original documentation.
- threshold_before_merge: see original documentation.
- transform: see original documentation.
- cache_estimates: see original documentation.
- verbosity: see original documentation.

Supports language features:

- Action costs
- Conditional effects (see note in original documentation)
- axioms

The heuristic has the following properties:

- Admissible
- Consistent
- Safe

2.4.19 Operator counting heuristic

See original documentation.

2.4.20 Pattern database heuristic

```
pdb(pattern=greedy(), transform=no_transform(), cache_estimates=true)
```

Where:

- pattern: see original documentation.
- transform: see original documentation.
- cache_estimates: see original documentation.

Supports language features:

- Action costs

The heuristic has the following properties:

- Admissible

- Consistent
- Safe

2.4.21 Sample-based potential heuristic

See original documentation.

2.4.22 Zero-One PDB

See original documentation for in depth description of the heuristic.

```
zopdbs(patterns=systematic(1), transform=no_transform(), cache_estimates=
   true)
```

Where:

- patterns: see original documentation.
- transform: see original documentation.
- cache_estimates: see original documentation.

Supports language features:

- Action costs

The heuristic has the following properties:

- Admissible
- Consistent
- Safe

2.5 Open list

Open lists in Fast downward are the queues that are keeping track of which node to visit next. These queues can either only contain nodes generated from preferred operators or all operators that hasn't been pruned. One should note that if multiple lists are used then it's possible that a queue contains already visited nodes. Disregarding if the node has been visited or not, it's counted as a node being selected from the open list.

2.5.1 Alternation open list

This list alternates between a set of lists always selecting the one with the lowest priority. The priorities are all initialised with one and increased with one when a node is selected from them. All sub-lists that only contains preferred operators can be rewarded during the search and thereby having their priorities reduced.

```
alt(sublists, boost=0)
```

Where:

- sublists: a list of open lists.
- boost: an integer describing how much the sub-queues will have their priorities decreased with when the queue is rewarded.

2.5.2 Epsilon-greedy open list

See original documentation.

2.5.3 Pareto open list

See original documentation.

2.5.4 Standard open list

A standard queue that sorts its entries on one evaluator and thereafter their insertion order.

```
single(eval, pref_only=false)
```

Where:

- eval: the evaluator.
- pref_only: only contains preferred operators.

2.5.5 Tie-breaking open list

A list that uses multiple heuristics the sort the queue. They are evaluated in order so that the second is only considered if the first is the same for two elements and so on.

```
tiebreaking(evals, pref_only=false, unsafe_pruning=true)
```

Where:

- evals: a list of Evaluators.
- pref_only: only insert nodes generated by preferred operators.
- unsafe_pruning: see original documentation.

2.5.6 Type-based open list

See original documentation.