Fast Downward
Making use of causal dependencies in the problem representation

Malte Helmert and Silvia Richter

Institut für Informatik, Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee, Gebäude 052, 79110 Freiburg, Germany
{helmert, srichter}@informatik.uni-freiburg.de

Abstract

Fast Downward is a propositional planning system based on heuristic search. Compared to other heuristic planners such as FF or HSP, it has two distinguishing features: First, it is tailored towards planning tasks with *non-binary* (but finite domain) state variables. Second, it exploits the *causal dependency* between state variables to solve relaxed planning problems in a hierarchical fashion.

Fast Downward is a planning system based on heuristic state space search, in the spirit of HSP or FF (Bonet & Geffner 2001; Hoffmann & Nebel 2001). It makes use of the *causal graph* (or CG) heuristic, introduced in an ICAPS 2004 paper (Helmert 2004). In this extended abstract, we aim at providing a high-level overview of Fast Downward, emphasizing the features that are not described in the CG article. While the CG heuristic was introduced for pure STRIPS domains, Fast Downward is capable of dealing with the complete propositional, non-temporal part of PDDL. In other words, it handles arbitrary ADL constructs and derived predicates (axioms).
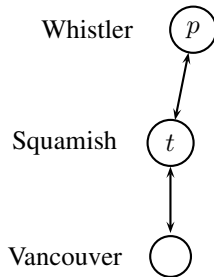

Figure 1: A simple planning task. Get the ICAPS participant *p* to Vancouver, using the taxi *t*.

The key feature of the CG heuristic — and the origin of Fast Downward's name — is the use of *hierarchical decomposition* to solve relaxed planning tasks. To illustrate this, consider the planning task in Fig. 1: The objective is to move the ICAPS participant *p* from Whistler (*W*) to Vancouver (*V*), using a taxi (*t*) initially located at Squamish (*S*).

The CG heuristic solves this problem hierarchically. The high-level goal is to change the state of the participant from
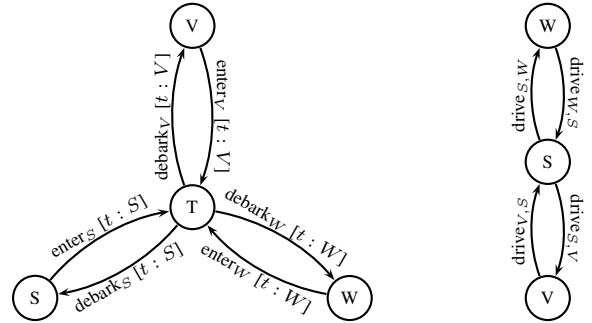

Figure 2: Domain transition graphs for the participant *p* (left) and taxi *t* (right).

"at Whistler" to "at Vancouver". The easiest way to do this is to board the taxi at Whistler and debark at Vancouver; at this point we do not care that these actions are not immediately applicable. This plan is found by looking at the ICAPS participant's *domain transition graph*, a directed graph depicting the ways in which *p* can change locations (Fig. 2). The different locations or *states* of *p* form the nodes of the graph, while the arcs correspond to operators affecting these states, annotated with their preconditions.

To estimate the cost of the "high-level plan" $p : W \rightsquigarrow T \rightsquigarrow V$, the heuristic solver inserts steps to satisfy the preconditions of the two operators by recursive invocations of the same algorithm. The transition $p : W \rightsquigarrow T$ requires the taxi to be at Whistler, as evidenced by the labeling of that arc in *p*'s domain transition graph. So we recursively find a (one-step) plan to move the taxi from its initial location Squamish to Whistler. Because there are no conditions on the transitions of the taxi (Fig. 2), there is no further recursion. We have thus computed that the cost of changing the state of the participant from W to T is 2, counting one action for the transition itself and one for the recursively calculated set-up cost. Similarly, we compute that the second transition $p : T \rightsquigarrow V$ is 3, because the taxi is now located in Whistler and thus needs two actions to get to Vancouver, in addition to the one action required to move *p* out of the taxi. Adding the transition costs together, the CG heuristic approximates the goal distance as $5 = 2 + 3$.

Observe that state transitions of the passenger are condi-

tioned on the state of the taxi, while the converse is not the case. We say that state variable p is *causally dependent* on state variable t. The set of causal dependencies of a planning tasked defines the *causal graph* of that task. Hierarchical decomposition is most suited to planning domains with acyclic causal graphs. In fact, the CG heuristic can only be calculated for tasks with acyclic causal graphs, and hence Fast Downward's heuristic estimator breaks causal cycles for the purposes of the heuristic estimator, by ignoring (some) operator preconditions. Contrast this relaxation to HSP's approach of ignoring (some) operator effects.

We hope that this small example provides the reader with some intuition of the basic ideas of the CG heuristic. Again, we point to the reference for a detailed exposition (Helmert 2004). In the following, we discuss the overall structure of the Fast Downward planner, emphasizing aspects that go beyond the STRIPS planner described in the conference paper.

Structure of the planner

Fast Downward currently consists of three independent programs:

1. the translator (written in Python),

2. the preprocessor (written in C++), and

3. the search engine (also written in C++).

To solve a planning task, the three programs are called in sequence; they communicate via text files. We have found that this clear separation facilitates simultaneous development of the planner by several people in its current prototype stage. Of course the current state of affairs leads to some inefficiencies, especially when solving easy or moderately difficult planning tasks. For hard tasks, runtime is typically dominated by the search engine.

Translator

The *translator* has the following responsibilities:

- Compiling away (most) ADL features.

- Grounding the operators and axioms.

- Converting the propositional (binary) representation to one with multi-valued state variables.

It is commonly known that some features of ADL can be compiled away easily, i.e. without significantly increasing the problem representation, while others cannot (Nebel 1999). However, in the presence of axioms, all ADL constructs except for conditional effects can be translated to STRIPS quite easily.

Fast Downward applies the following transformations, in order, to simplify the problem representation:

- Translate implications to disjunctions and translate all conditions to negation normal form (NNF).

- Compile away universal quantifiers in conditions.

- Translate conditions to prenex normal form.

- Translate the quantifier-free part of conditions into disjunctive normal form.

- Split operators or axioms with disjunctive conditions into several operators or axioms, and split conditional effects with disjunctive conditions into several effects.

All these transformations are fairly basic, except maybe for the elimination of universal quantifiers explained now. Using the equivalence $\forall x \varphi \equiv \neg \exists x \neg \varphi$, the translator introduces a new axiom for $\exists x \neg \varphi$ and replaces the universally quantified condition $\forall x \varphi$ by the literal $\neg \text{new-axiom}(\overline{V})$, where $\overline{V}$ is the set of free variables in $\exists x \neg \varphi$.

For example, the `blocked` axiom in the Promela domain contains the condition (ignoring types):

$$\forall t (\forall s' \neg \text{trans}(q, t, s, s') \lor \text{blocked-trans}(p, t)).$$

This is translated to the condition $\neg \text{new-axiom}(p, q, s)$, where $\text{new-axiom}(p, q, s)$ is defined as:

$$\exists t \neg (\forall s' \neg \text{trans}(q, t, s, s') \lor \text{blocked-trans}(p, t)),$$

which is translated to NNF, resulting in:

$$\exists t (\exists s' \text{trans}(q, t, s, s') \land \neg \text{blocked-trans}(p, t)).$$

After all transformations, all conditions are essentially simple conjunctions of literals (the remaining existential quantifiers can be considered action, axiom or effect parameters), so the resulting planning task is expressed in STRIPS with negation plus universal conditional effects and axioms.

For such planning tasks, efficient grounding is comparatively easy. Following the idea of Mips (Edelkamp & Helmert 1999), we avoid instantiating operators which can never be applied by first computing the set of propositions which are reachable in a *relaxed exploration*, ignoring negative conditions and effects. This amounts to the evaluation of a set of Horn logic rules derived from the actions and axioms. For example, the above axiom corresponds to the rule

```
new-axiom(p,q,s) :- trans(q,t,s,s').
```

The final translation step consists of replacing the set of binary state variables obtained by grounding with a smaller set of finite domain state variables capturing the same information. This is done by synthesizing invariants of the planning task, again using the algorithm of Mips.

To illustrate this, the variables p and t of our earlier example task are derived from the original PDDL representation by use of invariants. Specifically, the invariant

$$\exists_{=1} l : \text{taxi-at}(l),$$

justifies replacing the three binary variables `taxi-at(V)`, `taxi-at(S)` and `taxi-at(W)` by the variable t with domain $\{V, S, W\}$.

Preprocessor

The *preprocessor* is responsible for:

- Computing the causal graph of the planning task.

- Computing the domain transition graphs for each state variable.

- Computing the *successor generator*, a data structure that supports efficiently computing the successor states of a world state. (We do not discuss the successor generator in detail.)

Computing the causal graph is straight-forward: Variable A depends on variable B iff there is an operator (axiom) with A as an effect (consequence) and B as a condition or other effect. One notable optimization is employed at this point: All variables which are not mentioned in the goal and on which the goal does not depend directly or indirectly can be eliminated. For example, in the PSR domain, all instances of the `upstream` axiom for which the first parameter is not a circuit breaker may be safely removed.

As noted before, an acyclic causal graph is required for the CG heuristic. Therefore, for the purposes of the domain transition graphs, we compute an *acyclic skeleton* of the causal graph, i.e. a maximal acyclic subgraph. Cycles are broken by removing the weakest edges; this means that every dependency is weighted according to how often it occurs in the operators, and the edges with least weight are removed iteratively, until no cycle remains.

The central part of the preprocessor is the computation of the domain transition graphs. The domain transition graph of a variable contains arcs for all operators or axioms affecting this variable. For example, the graph for p in Fig. 2 contains an arc from V to T because there exists an operator with precondition $p = V$ and effect $p = T$, corresponding to the action of boarding the taxi in Vancouver. The arc is annotated with the condition $t = V$ because the operator requires the taxi to be in Vancouver as an additional precondition. We would omit this condition if the causal link between p and t were not part of the acyclic skeleton of the causal graph computed earlier. Thus, this is the part of the planner where some preconditions get ignored.

The reference (Helmert 2004) explains the details of domain transition graph construction for basic STRIPS-like operators; we note that the conditional effects present in the more general case do not lead to complications because domain transition graphs deal with operators one effect at a time, and for unary operators effect conditions can safely be considered part of the operator precondition.

Search Engine

After so much preprocessing, the actual search algorithm is not very mysterious. Fast Downward uses greedy best-first search, always expanding the node with the best heuristic estimate. The heuristic is computed from the domain transition graphs as follows: The goal distance of a state is taken to be the sum of the costs for all necessary changes of variables. The cost for changing the value of one variable V from v to v' is the sum of the costs for all transitions of V on the shortest path from v to v' in V's domain transition graph, computed using Dijkstra's algorithm.

The cost for traversing a single arc in the domain transition graph — the arc weight in Dijkstra's algorithm — is one plus the *set-up cost* of the transition, the sum of the (recursively computed) costs for achieving all necessary preconditions according to the arc label.[1] This follows the informal description of the CG heuristic in the introduction.

---

[1]If the arc corresponds to the derivation rule of an axiom, not to an action, then the weight is just the set-up cost, without adding 1.

Helpful Actions

As a further enhancement, Fast Downward incorporates the CG counterpart of FF's helpful actions: The planner collects all operators that correspond to domain transition graph arcs which contribute to the heuristic estimate of the given state. It then checks which of these operators are applicable in the current state. These form the set of helpful actions in that state. This set can be empty although the heuristic estimate is finite, because domain transition graphs do not respect all operator preconditions, as discussed before.

The overall best first search algorithm integrates helpful actions by maintaining two separate open lists; all states are first inserted into the first open list. When a state from this list is expanded, the "helpful" successors are generated and the state is inserted into the second open list. When a state from the second list is expanded, its "non-helpful" successors are expanded. The search control always selects that open list for expansion which has generated fewer search states so far. This means that if an average state encountered during search has 4 helpful and 40 other successors, the first open list is selected ten times out of eleven, thus biasing the exploration towards helpful actions.

Fast Diagonally Downward

As a final twist, we have also implemented a modified version of the search engine which combines CG heuristic and FF heuristic. This is based on the observation that CG and FF heuristic perform badly in different planning domains (Helmert 2004). Combining the forward and downward thrust by a simple vector addition, we have called this variant of the Fast Downward planner *Fast Diagonally Downward*.

Fast Diagonally Downward's search engine computes both the CG and FF heuristic for each state, as well as making use of helpful actions of both kinds. It uses separate open lists for the two heuristics, alternately expanding the node preferred by the FF estimate and the node preferred by the CG estimate. Newly generated states are always added to both open lists, making the approach different to simply running two planners in parallel. The hope is that the heuristics can lead each other out of their respective local minima, and indeed in some domains the combined approach works better than either of the original heuristics.

References

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *AIJ* 129(1):5–33.

Edelkamp, S., and Helmert, M. 1999. Exhibiting knowledge in planning problems to minimize state encoding length. In *Proc. ECP 1999*, 135–147.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In *Proc. ICAPS 2004*, 161–170.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

Nebel, B. 1999. What is the expressive power of disjunctive preconditions? In *Proc. ECP 1999*, 294–307.