Fast Planning in Domains with Derived Predicates: An Approach Based on Rule-Action Graphs and Local Search

Alfonso Gerevini[+] **Alessandro Saetti**[+] **Ivan Serina**[*] **Paolo Toninelli**[+]
[+] Dip. di Elettronica per l'Automazione, Università degli Studi di Brescia, via Branze 38, 25123 Brescia, Italy
[*] Dept. of Computer and Information Systems, University of Strathclyde, Glasgow, UK
E-mail: {gerevini,saetti}@ing.unibs.it ivan.serina@cis.strath.ac.uk

Abstract

The ability to express "derived predicates" in the formalization of a planning domain is both practically and theoretically important. In this paper, we propose an approach to planning with derived predicates where the search space consists of "Rule-Action Graphs", particular graphs of actions and rules representing derived predicates. We present some techniques for representing rules and reasoning with them, which are integrated into a method for planning through local search and rule-action graphs. We also propose some new heuristics for guiding the search, and some experimental results illustrating the performance of our approach. Our proposed techniques are implemented in a planner that took part in the fourth International Planning Competition showing good performance in many benchmark problems.

Introduction

In classical domain-independent planning, derived predicates are predicates that the domain actions can only indirectly affect. Their truth in a state can be inferred by particular axioms, that enrich the typical operator description of a planning domain.

As discussed in (Thiebaux, et al. 2003; Edelkamp & Hoffmann 2004), derived predicates are practically useful to express in a concise and natural way some indirect action effects, such as updates on the transitive closure of a relation. Moreover, compiling them away by introducing artificial actions and facts in the formalization is infeasible because, in the worst case, we have an exponential blowup of either the problem description or the plan length (Thiebaux, et al. 2003). This suggests that it is worth investigating new planning methods supporting derived predicates, rather than using existing methods with "compiled" problems.

The first version of PDDL, the language of the International Planning Competitions, supports derived predicates as particular "axioms", and the recent PDDL2.2 (Edelkamp & Hoffmann 2004) version re-introduces them as one of the two new features for the benchmark domains of the 2004 International Planning Competition (IPC-4). Some methods for handling derived predicates have been developed and implemented in several planning systems, such as UCPOP (Barrett, et al. 1995) and the very recent planners DOWNWARD, SGPLAN and MARVIN (Edelkamp, et al. 2004).

In this paper, we present some techniques for planning with derived predicates, which are implemented in a new version of the LPG planner (Gerevini, et al. 2003) called LPG-td. Representing derived predicates in forward search planners is not difficult, because the current state is fully known (Edelkamp & Hoffmann 2004; Thiebaux, et al. 2003). However, in backward planners or non-directional planners like LPG and many other systems this is not trivial, because such planners work with *partial* descriptions of the world states. Moreover, since our general goal is fast planning, it is important to manage derived predicates efficiently and effectively in the search heuristics. This can be a complex task for both forward and non-forward planners.[1]

The main contributions of this work are (i) a method based on AND-OR-graphs for representing and managing the domain rules defining the derived predicates in the domain formalization (2nd section); (ii) a plan representation for domains with derived predicates based on particular graphs called *rule-action graphs* (3rd section); (iii) some techniques exploiting particular relaxed plans for the heuristic evaluation and restriction of the search neighborhood (4th section); (iv) some experimental results illustrating the effectiveness of our techniques (5th section).

Derived Predicates and the Rule Graph

In PDDL2.2, derived predicates are particular predicates that do not appear in the (positive or negative) effects of any domain action. The truth value of a derived predicate is determined by a set of *domain rules* of the form *if* $\Phi_{\overline{x}}$ *then* $P(\overline{x})$, where $P(\overline{x})$ is the derived predicate, $\overline{x}$ is a tuple of variables, the free variables in $\Phi_{\overline{x}}$ are exactly the variables in $\overline{x}$, and $\Phi_{\overline{x}}$ is a first-order formula such that the negated normal form (NNF) of $\Phi_{\overline{x}}$ does not contain any derived predicate in negated form. (In a NNF formula, negation occurs only in literals.) The last syntactic restriction has the semantical motivation of ensuring that there is never a negative interaction between the application of rules in a world state (for more details see (Edelkamp & Hoffmann 2004)).

Figure 1 shows a typical example of a derived predicate (`above`) in the blocks world. A block x is `above` y, if x is `on` y, or it is `on` a third block z, which is `above` y. `Above` is the transitive closure of the `on` relation.

[1] A derived predicate could be realized in different ways and, generally, more than one action is needed to achive it; in this case, the reachability information defined by a relaxed planning graph, like in the FF planner, could be inaccurate and the search could be guided toward a wrong direction. E.g., it happens for the derived goal `(not (affected ?b))` in the `PSR` test problems of IPC-4.

if $on(x,y) \vee \exists\, z\, \big(on(x,z) \wedge above(z,y)\big)$ **then** $above(x,y)$

s = { `ontable(A)`, `ontable(D)`, `on(C,D)`, `above(C,D)`, `on(B,C)`, `above(B,C)`, `above(B,D)` }

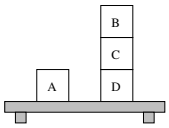


Figure 1: Example of a domain rule deriving a predicate in the blocks world, and of a state s where `above(B,C)`, `above(C,D)`, and `above(B,D)` are ground derived predicates.

In the rest of the paper, we call a ground predicate appearing in the initial state, problem goals, or in the preconditions or effects of a domain action a *basic fact*; we call a ground derived predicate obtained by substituting each variable in the derived predicate of a rule with a constant a *derived fact*.

A *grounded rule* is a rule where every predicate argument is a constant. Given a rule r = (*if* $\Phi_{\overline{x}}$ *then* $P(\overline{x})$) and a tuple of constants $\overline{c}$ ($|\overline{x}| = |\overline{c}|$), we can derive an equivalent *set* of grounded rules Γ by substituting in r the $\overline{c}$-constants for the corresponding $\overline{x}$-variables, and applying the following transformations to the resulting rule:

- $\Phi_{\overline{c}}$ is transformed into negated normal form;
- Each literal with an existentially quantified variable is replaced by a disjunction of literals where the variable is substituted by a constant of the planning problem (one disjunct for each constant);
- Each literal with a universally quantified variable is replaced by a conjunction of literals obtained by substituting the variable with every constant of the planning problem (one conjunct for each constant);
- $\Phi_{\overline{c}}$ is transformed into disjunctive normal form: $\Phi_{\overline{c}} = \phi^1 \vee ... \vee \phi^k$, where ϕ^i is a ground literal ($1 \leq i \leq k$);
- For each ϕ^i in $\Phi_{\overline{c}}$, the grounded rule *if* ϕ^i *then* $P(\overline{c})$ is added to Γ.

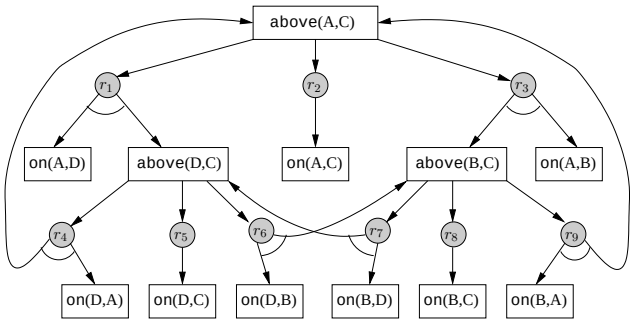
Given a planning problem and a set R of rules defining the derived predicates of the domain, we can then derive an equivalent set $\overline{R}$ of grounded rules. We call the left hand side (LHS) of each rule in $\overline{R}$ the *triggering condition* of the rule, and the conjoined facts forming the LHS of the rule the *triggering facts* of the rule.

We represent the domain grounded rules $\overline{R}$ through a *Rule Graph*, which is defined as follows.

Definition 1 *The* **rule graph** $\mathcal{R}$ *of a planning problem* Π *with derived predicates defined by a set of rules* $\overline{R}$ *is a directed* AND-OR*-graph such that:*

- AND*-nodes are either (i) leaf nodes labeled by basic facts of* Π*, or (ii) nodes labeled by derived facts of* Π*;* OR*-nodes are labeled by grounded rules in* $\overline{R}$ *and are not leaf nodes.*
- *Each* AND*-node* p *is connected to a set of* OR*-nodes representing the grounded rules deriving* p*. Each* OR*-node labeled* r *is connected to a set of* AND*-nodes representing the triggering condition of* r*.*

Figure 2 gives an example of a rule graph. For instance, the nodes labeled `on(A,D)` and `above(D,C)` are AND-nodes representing the triggering condition of rule r_1. `above(D,C)` is a derived fact that can be obtained by applying three rules (r_4, r_5 or r_6) represented by three OR-nodes with incoming edges from the AND-node labeled `above(D,C)`.



Grounded Rules (Or-Nodes of $\mathcal{R}$)
r_1: if `on(A,D)` $\wedge$ `above(D,C)` then `above(A,C)`
r_2: if `on(A,C)` then `above(A,C)`
r_3: if `on(A,B)` $\wedge$ `above(B,C)` then `above(A,C)`
r_4: if `on(D,A)` $\wedge$ `above(A,C)` then `above(D,C)`
r_5: if `on(D,C)` then `above(D,C)`
r_6: if `on(D,B)` $\wedge$ `above(B,C)` then `above(D,C)`
r_7: if `on(B,D)` $\wedge$ `above(D,C)` then `above(B,C)`
r_8: if `on(B,C)` then `above(B,C)`
r_9: if `on(B,A)` $\wedge$ `above(A,C)` then `above(B,C)`

Figure 2: A portion of the rule graph for a blocks world domain with the domain rule and objects of Figure 1. Circle nodes represent OR-nodes; square nodes represent AND-nodes. Multiple edges joined by an arc connect a domain rule to a set of AND-nodes representing the triggering condition of the rule.

Given a state s, and a set of domain rules R, we denote with $D(s,R)$ the set of the derived facts obtained by applying the rules in R to s with an arbitrary order until no new fact can be derived. In other words, $D(s,R)$ is the least-fixed point over all possible applications of the rules to the state where the derived facts are assumed to be false (because under the closed world assumption, they do not belong to s). An algorithm for deriving $D(s,R)$ is given in (Edelkamp & Hoffmann 2004).

In the following, we will abbreviate $s \cup D(s,R) \models \psi$ with $s \models^R \psi$, where $\models$ is the logical entailment under the closed world assumption on s, and ψ is a (basic or derived) fact.

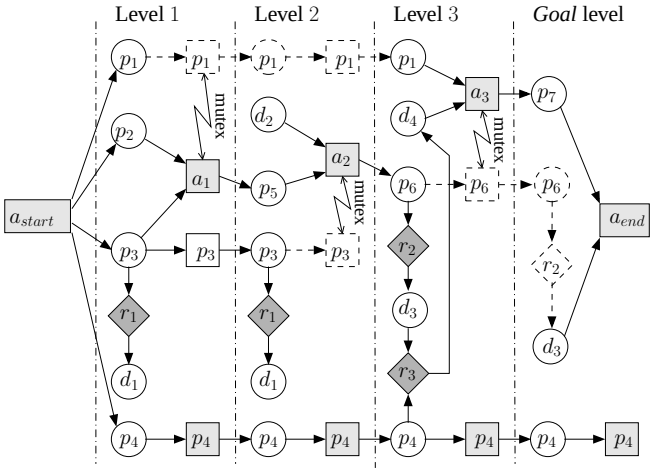
A Plan-based Search Space

Like in partial-order causal-link planning, e.g., (Penberthy & Weld 1992; McAllester & Rosenblitt 1991; Nguyen & Kambhampati 2001), in our approach we search in a space of partial plans, where each search state is a particular graph representing a plan under construction. In this section, we present our plan representation for domains with derived predicates, and the basic search steps (graph modifications) for exploring the search space.

Search State: Rule-Action Graph

We represent a (partial) plan in a domain with derived predicates through an extension of the linear action graph representation (Gerevini, et al. 2003), which we call *Rule-augmented Action Graph* or, shortly, *Rule-Action Graph*.

A linear action graph $\mathcal{A}$ for a planning problem Π is a directed acyclic leveled graph alternating a *fact level* and an *action level*. Fact levels contain *fact nodes*, each of which is labeled by a ground predicate of Π. Each fact node f at a level l is associated with a *no-op* action node at level l representing a dummy action having the predicate of f as its



ACTIVATED RULES:
r_1: if p_3 then d_1, r_2: if p_6 then d_3, r_3: if $d_3 \wedge p_4$ then d_4

Figure 3: A simple example of a rule-augmented action graph. Square nodes are action nodes; diamond nodes are rule nodes; circle nodes are (basic or derived) fact nodes. The square nodes marked by the facts p_1, p_3, and p_4 are no-op nodes. Dashed edges form chains of no-ops (rule nodes) that are blocked (deactivated) by mutex relations.

only precondition and effect. Each action level contains one action node labeled by the name of a domain action that it represents, and the no-op nodes corresponding to that level. An action node labeled a at a level l is connected by (i) incoming edges from the fact nodes at level l representing the preconditions of a (*precondition nodes*), and (ii) by outgoing edges to the fact nodes at level $l+1$ representing the effects of a (*effect nodes*). The initial level contains the special action node a_{start}, and the last level the special action node a_{end}. The effect nodes of a_{start} represent the positive facts of the initial state of Π, and the precondition nodes of a_{end} the goals of Π.

A pair of action nodes (possibly no-ops) can be related by a *persistent mutex relation*, i.e., a mutually exclusive relation holding at every level of the graph and imposing that the involved actions never occur in parallel in a valid plan.[2]

The definition of linear action graph can be made stronger by observing that the effects of an action node can be automatically propagated to the next levels of the graph through the corresponding no-ops, until there is an *interfering action* "blocking" the propagation, or the goal level has been reached (Gerevini, et al. 2003).

Definition 2 *A* **rule-action graph** *(RA-graph) of a problem* Π *with derived predicates is a linear action graph where*

- *each fact level can contain two additional types of nodes:* rule nodes *and* derived nodes;
- *each rule node is labeled by a grounded rule of* Π*, and each* derived node *is labeled by the fact derived by a grounded rule of* Π;
- *each rule node labeled* r *at a level* l *is connected by incoming edges to a set of fact nodes at* l *representing the triggering facts of* r*, and by an outgoing edge to a derived node at* l *representing the ground predicate derived by* r.

[2]Such relations are pre-computed by an extension of the algorithm given in (Gerevini, et al. 2003) handling derived predicates.

We call an action precondition node representing a derived fact a *derived precondition node*, and a node representing a triggering fact of a grounded rule *triggering node*.

Figure 3 gives an example of an RA-graph containing five action nodes, several fact nodes, some derived nodes representing four derived facts ($d_{1...4}$), and some rule nodes representing three grounded rules.

The rule nodes are automatically "activated" whenever the corresponding triggering nodes are supported, i.e., a rule node is inserted at a level of the graph (together with its derived fact node, if not already present) whenever its triggering nodes are all supported at that level. Notice that the no-op propagation can affect the activation of a rule at any level where the corresponding fact is propagated.

In the following, $S(l)$ indicates the world state obtained (under the closed world assumption) by applying to the problem initial state the actions in the RA-graph up to level $l-1$, ordered according to the level of their action node.

Definition 3 *A grounded rule* r *= (if* $\varphi_1 \wedge \cdots \wedge \varphi_n$ *then* ψ*) is* **activated** *at a level* l *of an RA-graph* $\mathcal{A}$ *iff, for each literal* φ_i *in* r*, either* $S(l) \models \varphi_i$ *or there exists an activated rule at* l *that derives* φ_i.

For instance, in the RA-graph of Figure 3, rule r_1 is activated at levels 1 and 2, rules r_2 and r_3 are activated at level 3, while rule r_2 is not activated at the goal level because of a_3 blocking the propagation of p_6.

Each RA-graph represents the partial plan formed by the actions associated with its action nodes, and can contain some *flaws*. A flaw at a level l of an RA-graph $\mathcal{A}$ is a precondition node of the action node at level l that is not supported in $\mathcal{A}$. If a level of an RA-graph has no flaw, we say that this level is *flawless*.

A basic fact node labeled q at a level l is *supported* if there is an action node at level $l-1$ representing an action with (positive) effect q. A derived precondition node p at a level l of an RA-graph is supported if and only if there is a rule node r at l such that p is the derived node of r. This is the case if and only if $S(l) \models^R \psi_p$, where ψ_p is the derived fact represented by p. An RA-graph without flaws represents a valid plan and it is called *solution RA-graph*.

Definition 4 *A* **solution RA-graph (valid plan)** *of a planning problem* Π *with derived predicates is a rule-action graph* $\mathcal{A}$ *of* Π *such that all levels of* $\mathcal{A}$ *are flawless.*

Search Steps for RA-graphs

Given an RA-graph $\mathcal{A}$ (search state) containing some flawed level(s), we can generate new RA-graphs (successor search states) by adding or removing an action node *helping to repair* a flawed level of $\mathcal{A}$. When we add an action node to a level l of the RA-graph, the graph is extended by one level and the nodes and edges at each level $l' \geq l$ are shifted one level forward. Similarly, when we remove an action node a, the RA-graph is "shrunk" by one level.

The definition of *Helpful Action Node*, that we can add to $\mathcal{A}$, and of *Harmful Action Node*, that we can remove from $\mathcal{A}$, relies on the notion of *Activation Fact Set* (shortly *Activation Set*). An activation set is a set of basic facts activating a set of rule nodes supporting a derived precondition node.

Definition 5 *Given an unsupported derived precondition node* d *at a level* l *of an RA-graph, an* **activation fact set** *for* d *is a minimal set* F *of basic facts such that* $S(l) \cup F \models^R \psi_d$*, where* ψ_d *is the derived fact represented by* d.

And-Search($n, A, PathNodes, Open, s$)
Input: An AND-node of the AND-OR rule graph $\mathcal{R}$ (n), the activation set under construction (A), the set of AND-nodes of $\mathcal{R}$ on the search tree path from the search tree root to n (*PathNodes*), the set of nodes to visit for A (*Open*), and a world state (s);
Output: An element of the activation set under construction, *false*, or the empty set.

1. **if** $n \in$ *PathNodes* **then return** *false*;
2. **if** $s \models^R n$ **then return** $\emptyset$;
3. **else if** n is a basic fact **then return** n;
4. **foreach** successor n' of n in $\mathcal{R}$ **do**
5. Or-Search($n', A, PathNodes \cup \{n\}$, *Open*, s);
6. **return** *false*.

Or-Search($n, A, PathNodes, Open, s$)
Side Effect: Update of the set of activation sets (Σ)

1. *Open* $\leftarrow$ *Open* $\cup \{n' \mid n'$ is a successor of n in $\mathcal{R}\}$;
2. **foreach** $t \in$ *Open* **do**
3. *Open* $\leftarrow$ *Open* $\setminus \{t\}$;
4. $n' \leftarrow$ And-Search(t, A, *PathNodes*, *Open*, s);
5. **if** $n' =$ *false* **then return**;
6. **else** $A \leftarrow A \cup \{n'\}$;
7. $\Sigma \leftarrow \Sigma \cup \{A\}$.

Figure 4: Algorithms for computing the activation sets (stored in the global variable Σ) of a derived precondition node by searching on the rule graph $\mathcal{R}$.

For example, suppose that r_4= (*if* $p_1 \wedge d_5$ *then* d_2) and r_5= (*if* $p_3 \wedge p_9$ *then* d_5) are two additional (inactive) rules for the RA-graph of Figure 1. Then $\{p_1, p_9\}$ is an activation set for d_2 at level 2 of the graph. Note that p_3 is not in the activation set for d_2, because at level 2 it is already supported.

Definition 6 *Given a flawed level l of an RA-graph $\mathcal{A}$, we say that an action node is* **helpful** *for l if its insertion into $\mathcal{A}$ at a level $i \leq l$ supports (i) a basic unsupported precondition node at l, or (ii) an (unsupported) node representing a fact in an activation set for an unsupported derived precondition node at l.*

For example, an action node representing an action with effect p_1 is helpful for level 3 of the RA-graph of Figure 3, if it is inserted into level 2 or 3.

Definition 7 *Given a flawed level l of an RA-graph $\mathcal{A}$, we say that an action node at a level $i \leq l$ is* **harmful** *for l if its removal from $\mathcal{A}$ either (i) would remove the unsupported precondition nodes at l ($i = l$), or (ii) would make an unsupported fact node f at l supported, where f is a basic precondition node, or it represents a fact in an activation set for a derived precondition node at l.*

For example, the action node a_3 of Figure 3 is harmful for level 3, because of the unsupported precondition node p_1 of a_3; a_3 is harmful for the goal level too, because it breaks the no-op propagation of p_6 at level 3, that would activate the rule r_2 supporting the derived node d_3 at the goal level.

We can identify the activation sets of a derived precondition node d at a level l by using the two mutually recursive algorithms described in Figure 4. These algorithms perform a complete backward search on the rule graph. And-Search visits an AND-node n of the rule graph and returns: (i) *false*, if n is a node already visited on the path from the root search tree to n ($n \in$ *PathNodes*), and hence the search is pruned to avoid looping; (ii) $\emptyset$, if n represents a fact that is entailed by $S(l) \cup D(S(l), R)$; (iii) n, if the previous cases do not apply and n is a basic fact (that will belong to the activation set under construction); (iv) *false*, otherwise (n together with its sibling AND-nodes have already been visited).

Or-Search visits an OR-node of the rule graph, and incrementally updates the set of activation sets, which are stored in the global variable Σ (initially set to the empty set). For example, if s defines the state described in Figure 1, And-Search(`above(A,C)`, $\emptyset, \emptyset, \emptyset$,$s$) searches in the portion of the rule graph of Figure 2, and identifies the set of possible activation sets of `above(A,C)`: $\Sigma = \{\{$`on(A,B)`$\}, \{$`on(A,C)`$\}, \{$`on(A,D)`, `on(D,C)`$\}, \{$`on(A,D)`, `on(D,B)`$\}\}$.

Note that, the maximum size of Σ depends on the planning problem and on the search states visited during the search process. In the worst case, the size of Σ can be exponential in the number of the problem objects involved by the grounded rules. However, in practice, for all the problems we tested from the IPC-4, the number of activation sets is polynomial. More on this in the experimental results section.

Local Search in the Space of RA-Graphs

The general scheme for searching a solution graph (a final state of the search) consists of a local search process in the space of all RA-graphs of the planning problem, starting from an initial RA-graph containing only a_{start} and a_{end}.

Each basic search step identifies the neighborhood $N(l, \mathcal{A})$ of the current RA-graph $\mathcal{A}$ for a flawed level l, i.e., the set of the RA-graphs obtained from $\mathcal{A}$ by adding a helpful action node for l, or by removing a harmful action node. The elements of the neighborhood are weighed according to a *heuristic evaluation function* estimating their quality, and an element with the best quality is then considered as the next possible RA-graph.

The basic search procedure that we use for searching RA-graphs is Walkplan (Gerevini, et al. 2003), a randomized procedure using a *noise parameter* p to escape from local minima. Given an RA-graph $\mathcal{A}$ and a flawed level l, if there is a modification helping to repair l that does not decrease the quality of $\mathcal{A}$, then the corresponding RA-graph is chosen as the next search state. Otherwise, with probability p one of the graphs in $N(l, \mathcal{A})$ is chosen randomly, and with probability $1 - p$ the next RA-graph is chosen according to the minimum value of the evaluation function.

Some heuristic evaluation functions for action graphs are proposed in (Gerevini, et al. 2003). In this section, we introduce a new heuristic function (E) for RA-graphs. The main differences with respect to the previous functions are: E gives a more accurate estimate of the search cost by taking account of *all* the flaws at a given level of the graph, instead of only one flaw; E estimates the search cost for supporting derived preconditions (derived nodes), which are not handled by the previous functions.

Relaxed Plans for RA-Graphs

The general idea for estimating the cost of making a level l flawless is to construct a relaxed plan π for the set of facts represented by the unsupported precondition nodes at l.

Suppose that we are evaluating the RA-graph obtained by adding an action node a to a level l_a, because it is helpful for l in the current RA-graph $\mathcal{A}$ ($l_a \leq l$). E uses a relaxed plan π to compute an estimate of a minimal set of new action nodes required to support

RelaxedPlan(G, I, A)

Input: A set of goal facts (G), an initial state for the relaxed plan (I), a set of reusable actions (A);
Output: A set of actions $Acts$ forming a relaxed plan for G from I

1. $G \leftarrow G - I$; $Acts \leftarrow A$;
2. $F \leftarrow \bigcup_{a \in Acts} Add(a)$;
3. $F \leftarrow F \cup D(I \cup F, R)$;
4. **while** $G - F \neq \emptyset$
5. **if** g is a basic fact in $G - F$ **then**
6. $b \leftarrow BestAction(g)$;
7. $Rplan \leftarrow$ RelaxedPlan$(Pre(b), I, Acts)$;
8. $Acts \leftarrow Aset(Rplan) \cup \{b\}$;
9. $F \leftarrow \bigcup_{a \in Acts} Add(a)$;
10. $F \leftarrow F \cup D(I \cup F, R)$;
11. **else** /* g is a derived fact */
12. $\Sigma \leftarrow \emptyset$; /* Σ is a set of activation sets for $\mathcal{R}$ */
13. And-Search$(g, \emptyset, \emptyset, \emptyset, I \cup F)$; /* Update Σ */
14. $H \leftarrow BestActivationSet(\Sigma)$;
15. $G \leftarrow G - \{g\} \cup \{H\}$;
16. **return** $Acts$.

Figure 5: Algorithm for computing a relaxed plan achieving a set of preconditions G from the state I using the domain rules R.

(1) *the unsupported precondition nodes of* a,
(2) *the flaws remaining at* l *after adding* a *to* $\mathcal{A}$, and
(3) *the supported precondition nodes of other action nodes in* $\mathcal{A}$ *that would become* un*supported by adding* a.

The larger such a set is, the higher is the estimated cost. Note that the heuristic functions previously proposed for action graphs do *not* consider (2). The new heuristic relaxes the "flaw-independence assumption", which in some domains is invalid and can mislead the search cost evaluation. This is particularly true for flaws at the same level: the way we repair a single flaw can affect the search cost of the other flaws at the same level.[3]

π is *relaxed* in the sense that it does not consider the negative interference between the actions of π and the domain rules activated by the positive effects of the actions of π. The initial state for π is obtained by applying the actions of $\mathcal{A}$ up to level $l_a - 1$, ordered according to their corresponding levels, and activating the possible domain rules.

The evaluation of an RA-graph in the search neighborhood that is derived by removing a harmful action node a is similar, and its description is omitted for lack of space.

Figure 5 shows the RelaxedPlan algorithm for computing relaxed plans. $Pre(a)$ denotes the set of facts corresponding to the preconditions of a, while $Add(a)$ the set of the positive effects of the action represented by a. Let $S^R(l)$ indicate the state $S(l) \cup D(S(l), R)$, where R is the set of the domain rules; l is the flawed level under reparation by adding an action a to level l_a. RelaxedPlan constructs a relaxed plan through a recursive backward process. The action chosen at step 6 to achieve a *basic* (sub)goal g is an action b such that (i) g is an effect of b; (ii) all preconditions of b are reachable from $S^R(l_a)$; (iii) it requires an estimated minimum number of additional actions required to support its preconditions from $S^R(l_a)$; (iv) b subverts the minimum number of supported (basic or derived) precondition nodes of $\mathcal{A}$. (iii) is computed as described in (Gerevini, et al. 2003).

When the (sub)goal g is a *derived* fact (steps 11–15), RelaxedPlan computes the set Σ of the activation sets for g, and it constructs a relaxed plan for the facts contained in the *best* activation set $H \in \Sigma$ (steps 12–14). In particular, the algorithm uses And-Search to compute Σ (step 13), and then selects from Σ a set H such that (i) all facts in H are reachable from $S^R(l_a)$; (ii) their reachability requires a minimum number of actions; and (iii) the insertion of an action a_h to achieve a facts h in H threats a minimum number of precondition nodes in the RA-graph.

Pruning the Neighborhood and the Activation Sets

In general, the effectiveness of a heuristic function evaluating the elements of the search neighborhood can be significantly affected by the size of the neighborhood: if this is too large, the neighborhood evaluation might require too much time. Since the number of activations sets for an unsupported derived preconditions p can in principle be very high, there might be many helpful actions for p, which could lead to a very large search neighborhood.

We have developed a heuristic method which evaluates the activation sets of p to select the "easiest one" to support in terms of search cost. Assume that at the flawed level under consideration we have a set U of unsupported precondition nodes. For each derived node d in U, we choose *one* of its activation sets by evaluating each of them using RelaxedPlan with the activation set as goal set. The selected activation set is one with the best relaxed plan (fewest number of actions and threats).

Moreover, during the generation of the activation sets, we prune those containing facts that are mutex with the preconditions at the flawed level l under consideration. (If this were the case, the truth of the facts in the activation set would make it impossible to support all precondition nodes at l.) In addition, when we search the rule graph we pre-evaluate the activation sets using reachability information from $S^R(l)$. The reachability cost of an activation set is defined as the maximum over the reachability cost of its facts. During their generation, the activations sets with cost exceeding the best set computed so far are pruned.

Experimental Results

In this section, we present some experimental results obtained using test domains and problems from IPC-4 and IPC-3, which illustrate the effectiveness of our techniques implemented in the LPG-td planner.[4]

Figure 6 shows the CPU-time (in logarithmic scale) of the IPC-4 planners for `Philosophers` and `PSR-Middle`. In the first domain, both LPG-td and DOWNWARD solve 48 problems, while MARVIN solves 30 problems. LPG-td is generally faster than the other planners, except for a few problems where DOWNWARD performs better than LPG-td. In `PSR-Middle` both SGPLAN and LPG-td solve all problems, but LPG-td is generally faster. Concerning plan quality, overall the plans generated by LPG-td are comparable to the plans generated by the other planners.

An interesting question is how many activation sets are generated in practice for a derived precondition (i.e., how

[3] An experimental analysis showed that the inclusion of (2) leads to significant speeds up in many benchmark problems. For lack of space, we omit these results (part of which are given in (Gerevini, et al. 2005)).

[4] For a description and formalization of these benchmark domains and problems, see the official web pages of IPC-3 and IPC-4.

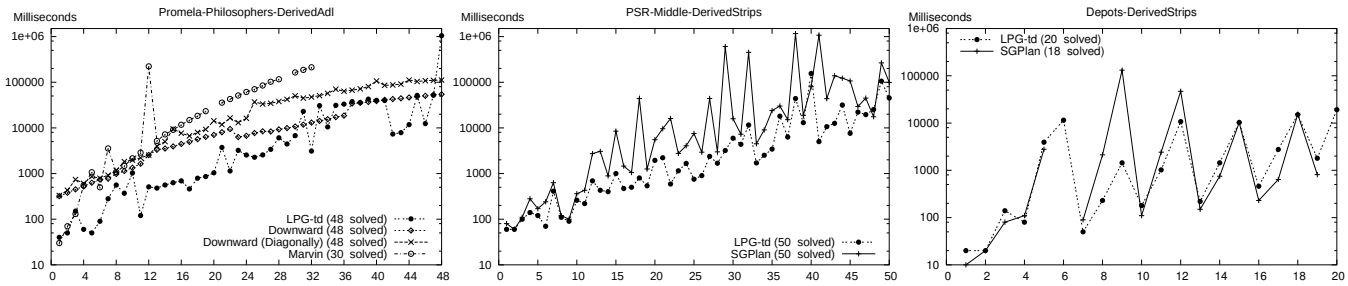


Figure 6: Performance of LPG-td and some IPC-4 planners in three benchmark domains involving derived predicates. On the x-axis we have the problem names (abbreviated by numbers). On the y-axis, we have CPU-time (log scale).

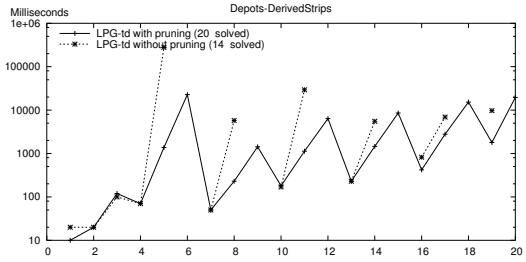


Figure 7: CPU-time of LPG-td with/out pruning of the activation sets for `Depots` with derived predicates.

Domains	μ_Σ (No Pruning)	μ_Σ	σ_Σ	CPU-time
Depots	330,597	35	461	5%
Philosophers	7	7	0	2%
PSR	275	259	3635	13%

Table 1: Number of activation sets without pruning for a derived precondition (mean), number of activation sets with pruning (mean and standard deviation), and average % of total CPU-time for deriving them with pruning in `Depots`, `Philosophers` and `PSR`.

large Σ is). Despite Σ can be exponentially large, we have experimentally observed that, for all the IPC-4 test problems attempted by LPG-td, $|\Sigma|$ was less than $1.2 \cdot n^2$, where n is number of the objects involved by the grounded rules. In the IPC-4 problems, the pruning techniques of Σ are not very effective, but the additional overhead is generally negligible.

To test the pruning techniques of Σ in a domain that can generate many more activation sets, we extended the IPC-3 domain `Depots` by adding derived predicates to it, and we modified the IPC-3 problems by replacing several "`on`" goals with the corresponding "`above`" goals, where `above` is a derived predicate. The 3rd plot of Figure 6 shows the performance of LPG-td and SGPLAN in the modified domain. Overall, the planners perform similarly, but LPG-td solves more problems, and in few cases it is significantly faster than SGPLAN. Figure 7 shows the usefulness of pruning in `Depots`. LPG-td with the pruning is generally faster and solves more problems.

Table 1 gives some statistical data on the size of the activation sets over all the test problems considered. Without pruning, in `Depots` the size of Σ can be up to four orders of magnitude larger, while in the other domains it is not significantly larger.

Finally, some experimental tests show that managing derived predicates in LPG-td using our methods (instead of compiling them away) significantly reduces planning time, confirming the obervation in (Thiebaux, et al. 2003).

Conclusions

We have presented some new techniques aimed at fast planning in domains involving derived predicates. Our methods extend the "planning through action graphs and local search" approach by (i) including a rule graph to support reasoning about derived predicates in the search states produced by the plan actions; (ii) augmenting the action graph representation with additional nodes and arcs representing (automatically triggered) domain rules; (iii) defining a new search space formed by such augmented action graphs, (iv) designing new heuristics to guide a local search process. These techniques are implemented in the LPG-td planner, which showed good performance in many benchmark problems.

We believe that our methods to handle derived predicates can be potentially useful for other non-directional or backward approaches to planning, and future work includes this investigation. For instance, the rule-graph and the activation fact set could be used in the Graphplan-based search phase to define the subgoals during regression in problems with derived predicates.

References

Barret, A., Christianson, D., Friedman, M., Kwok, C., Golden, K., Penberthy, S., Sun, Y., Weld, D. 1995. UCPOP User's Manual *T.R. 93-09-08d*, Univ. of Washington, Computer Science Dept.

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281–300.

Edelkamp, S., Hoffmann, J., Littman, M., Younes, H. (Eds.) 2004. *In Abstract Booklet of the IPC-4 Planners (ICAPS-04)*.

Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: The language for the Classic Part of the 4th International Planning Competition. *T.R. no. 195*: Institut für Informatik, Freiburg, Germany.

Fox, M., and Long, D. 2003 PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains *JAIR* 20:61–124.

Gerevini, A., Saetti, A., and Serina, I. 2003. Planning through Stochastic Local Search and Temporal Action Graphs. *JAIR* 20:239–290.

Gerevini, A., Saetti, A., Serina, I, and Toninelli, P. 2005. Planning with Derived Predicates through Rule-Action Graphs and Relaxed-Plan Heuristics. *T.R. 2005-01-40*, DEA, Univ. Brescia.

McAllester, D., and Rosenblitt, D. 1991. Systematic nonlinear planning. In *Proc. of AAAI-91*.

Nguyen, X., and Kambhampati, S. 2001. Reviving partial order planning. In *Proc. of IJCAI-01*.

Penberthy, J., and Weld, D. 1992. UCPOP: A sound, complete, partial order planner for ADL. *Proc. of KR'92*.

Thièbaux, S., Hoffmann, J., and Nebel, B. 2003. In defense of PDDL Axioms. *Proc. of IJCAI-03*.