Planning with Derived Predicates through Rule-Action Graphs and Relaxed-Plan Heuristics

Alfonso Gerevini Alessandro Saetti Ivan Serina Paolo Toninelli
Dipartimento di Elettronica per l'Automazione,
Università degli Studi di Brescia
Via Branze 38, I-25123 Brescia, Italy
{gerevini,saetti,serina}@ing.unibs.it

Abstract

The ability to express "derived predicates" in the formalization of a planning domain is both practically and theoretically important. The recent PDDL2.2 language supports derived predicates, which can be expressed by "domain rules".

We propose an approach to planning with derived predicates where the search space consists of particular graphs of actions and rules, called *rule-action graphs*, representing partial plans. We present (i) some techniques for managing domain rules in the context of a local search process for rule-action graphs, (ii) new heuristics for guiding the search, and (iii) a new method for restricting the search neighborhood to speed up the search.

The proposed approach and techniques are implemented in a new version of the LPG planner, which took part in the fourth International Planning Competition showing good performance in many benchmark problems.

1 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 [17, 5], 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 blow up of either the problem description or the plan length [17]. This suggests that is worth investigating new planning representation and algorithms supporting derived predicates, rather than using existing methods with "compiled" problems.

The first version of PDDL [12], the language of the International Planning Competitions, supports derived predicates as particular "axioms", and the recent PDDL2.2 [5] 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 [2] and the very recent planners DOWNWARD [10], SGPLAN [3] and MARVIN [4].

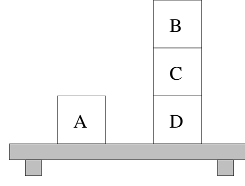
In this paper, we present some techniques for planning with derived predicates, which are implemented in a new version of the LPG planner [8] called LPG-td.[1] Our methods extend an approach to planning in a space of *action graphs* representing partial plans, which is explored by a local search process.

The main contributions of this work are:

- a method based on AND-OR-graphs for representing and managing the domain rules defining the derived predicates in the domain formalization;
- a plan representation for domains with derived predicates based on particular graphs called *rule-action graphs*;
- some techniques exploiting particular relaxed plans for the heuristic evaluation and restriction of the search neighborhood (both for simple STRIPS domains and domains involving derived predicates);
- some experimental results from IPC-4 illustrating the effectiveness of our techniques.

[1] The "td" suffix is an abbreviation of "Derived predicates and Timed initial literals", the two new features of PDDL2.2 with respect to PDDL2.1 [6].

$$\textbf{if} \quad on(x,y) \lor \exists\, z \left(on(x,z) \land above(z,y)\right) \textbf{ then } above(x,y)$$



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

Figure 1: Example of 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.

The rest of the paper is organized as follows. The second section introduces our representation of derived predicates; the third section presents rule-action graphs; the fourth section gives some new search techniques; the fifth section describes some experimental results from IPC-4; finally, the last section gives the conclusions.

2 Representing Derived Predicates: 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

$$\textit{if } \Phi_{\overline{x}} \textit{ then } P(\overline{x}),$$

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.[2] 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 [5]).

Figure 1 shows a typical example of 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.

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 an 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: $\Phi_{\overline{c}} = \phi^1 \lor ... \lor \phi^k$, 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** *of a planning problem* Π *with derived predicates defined by a set of rules* R *is a directed* AND-OR*-graph such that:*

[2] In a NNF formula, negation occurs only in literals.

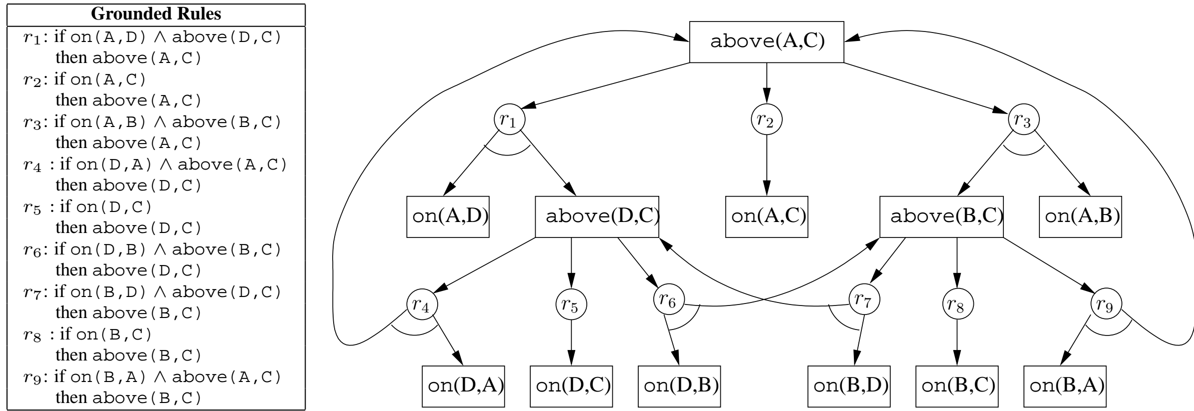


Figure 2: A portion of the rule graph for a blocks world domain with the domain rule and objects of Figure 1. Circle nodes represent OR-nodes; square nodes represent AND-nodes. Multiple edges joined by an arc connect a domain rule to a set of AND-nodes representing the triggering condition of the rule. 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 and r_6) represented by three OR-nodes with incoming edges from the AND-node labeled `above(D,C)`.

- 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 rule graph. Notice that, in general, each OR-node in a rule graph has a single incoming edge, because of the syntax of the rules defining derived predicates.

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 [5].

In the rest of the paper, we 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.

3 A Plan-based Search Space

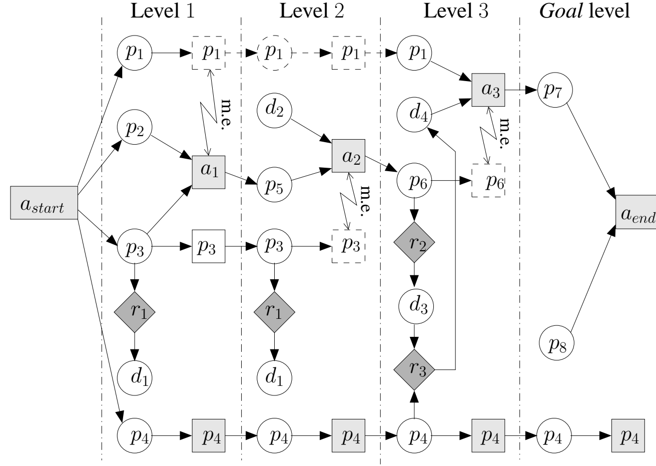
Like in partial-order causal-link planning, e.g., [14, 11, 13], in our approach we search in a space of partial plans, where each search state is a particular graph representing a plan under construction. In this section, we present our plan representation for domains with derived predicates, and the basic search steps (graph modifications) for exploring the search space.

3.1 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 [7, 8], 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 basic *fact nodes*, each of which is labeled by a ground predicate of Π. Each action level contains one action node labeled by the name of a domain action that it represents, and *no-op nodes* defined as in [1]. 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.



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 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 that are blocked by a mutex relation. "m.e." indicates mutual exclusion.

A pair of action nodes (possibly no-op nodes) can be related by a *persistent mutex relation*, i.e., a mutually exclusive relation holding at every level of the graph and imposes that the involved actions never occur in parallel in a valid plan. Such relations are pre-computed by an extension of the algorithm presented in [8] to deal with derived predicates (for lack of space, in this paper we omit a description of the revised algorithm).[3]

Definition 2 *A* **rule-action graph** *(RA-graph) of a planning 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.*[4]

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 a *triggering node*.

Figure 3 gives a simple example of RA-graph containing five action nodes ($a_{start}, a_1, a_2, a_3, a_{end}$), several fact nodes representing eight basic facts, some derived nodes representing four derived facts, and some rule nodes representing three grounded rules.

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 a RA-graph $\mathcal{A}$ is either a precondition node of the action node (or of a no-op node) at level l that is not supported in $\mathcal{A}$, or a triggering node of a rule node at level l that is not supported. A basic fact node labeled q at a level l is *supported* if there is an action node (or a no-op node) at level $l-1$ representing an action with (positive) effect q. A derived node labeled p at level l is supported if there is a rule node at level l representing a grounded rule deriving p. If a level of a RA-graph has no flaw, we say that this level is *flawless*.[5]

[3] Mutex relations between actions are defined in accordance with the rule of *no moving targets* defined by Fox and Long [6], and extended by Edelkamp and Hoffmann ([5]) to domains with derived predicates.

[4] Notice that, unlike planning graphs [1] and action graphs, RA-graphs are not acyclic graphs, because they can contain cycles involving rule nodes and fact nodes.

[5] Under the assumption that the initial state of the planning problem and the preconditions/effects of each action are consistent, according to our definition of plan flaw, we have that a level l of a RA-graph $\mathcal{A}$ contains a flaw if and only if the state obtained by (1) applying to the initial

For example, in the RA-graph of Figure 3, node d_4 is supported by the rule node labeled r_3, node p_5 is supported by the action node labeled a_1, while p_8 and d_2 are not supported.

A RA-graph without flaws represents a valid plan and it is called a *Solution RA-graph*.

Definition 3 *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.*

Note that, as shown in [8], having only one action in each level of a RA-graph does not prevent the generation of parallel (partially ordered) plans.

The notion of supported facts and the definition of RA-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 last level of the graph has been reached [7]. Moreover, the rule nodes can be automatically "activated" whenever the corresponding triggering nodes are supported, i.e., a rule node is automatically 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.

The next definitions incorporate the no-op propagation and the automatic rule activation into the rule-action graph. The main advantage of this extension is that it leads to a smaller search space, because there is no need to treat the insertion/removal of no-op and rule nodes as search steps during planning.

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 corresponding action node.

Definition 4 *A* **rule-action graph with no-op propagation and automatic rule activation** *of a planning problem* Π *is an rule-action graph* $\mathcal{A}$ *such that*

- *p is a no-op node at a level l of $\mathcal{A}$ iff (i) there is a node at a level h representing an action with effect p, and (ii) there is no action node at a level k which is mutex with the no-op of p, and such that $h \leq k < l$;*
- *r is a rule node at a level l of $\mathcal{A}$ iff the rule represented by r is* activated *by $S(l)$.*

Definition 5 *A grounded rule r = (if $\varphi_1 \wedge \cdots \wedge \varphi_n$ then ψ) is* **activated** *at a level l of a 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, while rules r_2 and r_3 are activated at level 3.

A derived precondition node p at a level l of a RA-graph with no-op propagation and automatic rule activation is supported if and only if there is a rule node r at l such that p is the derived node of r. It is easy to see that this is the case if and only if $S(l) \models^R \psi_p$, where ψ_p is the derived fact represented by p. Finally, it is important to observe that in a RA-graph with no-op propagation and automatic rule activation, the only possible flaws are unsupported precondition nodes of domain action nodes.

Since in the rest of this paper we will consider only rule-action graphs with no-op propagation and automatic rule activation, we will abbreviate their name simply to rule-action graphs (leaving implicit that they include the no-op propagation and automatic rule activation).

3.2 Search Steps for RA-graphs

Given a 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 $\mathcal{A}$. In order to better exploit the heuristics described in the next section, in LPG-td such a level is the *first* flawed level of $\mathcal{A}$.[6]

state the actions in $\mathcal{A}$ up to level $l-1$, and (2) augmenting such a state with the preconditions of the action at level l and with the facts of the triggering nodes of the rule nodes at level l, is inconsistent.

[6] We have designed and experimentally tested different flaw selection strategies, that are described in [9]. The one preferring flaws at the earliest level of the graph tends to perform better than the others, and is used as default strategy. For more details and a discussion on this strategy see the paper mentioned above.

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** $\emptyset$.

Or-Search(n, A, $PathNodes$, $Open$, s)

Input: An OR-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);

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 $\mathcal{R}$.

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.[7]

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*). Essentially, an activation set is a set of basic facts activating a set of rule nodes supporting a derived precondition node.

Definition 6 *Given an unsupported derived precondition node d at a flawed level l of a 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.*

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 3. We have that $\{p_1, p_9\}$ 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 7 *Given a flawed level l of a 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; while it is not helpful, if it is inserted into level 1 because a_1 blocks the propagation of p_1.

Definition 8 *Given a flawed level l of a 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.*

[7]As discussed in [8], the removal of an action node a can induce the (automatic) removal of other redundant action nodes (i.e., those action nodes supporting only the precondition nodes of a and, recursively, the activation nodes supporting a precondition node of a removable action).

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_1 is harmful for level 3, because it breaks the no-op propagation of p_1 at level 1, that would support the precondition node p_1 at level 3. Notice that a_1 is an harmful action node for level 3, but it is also an helpful action node for level 2 because it supports the precondition node p_5.

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, providing that s defines the state described in Figure 1, And-Search(above(A,C), $\emptyset$, $\emptyset$, $\emptyset$, s) searches in the portion of 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), on(B,C)$\}\}$.

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 n of the problem objects involved by the grounded rules. However, in practice, for all problems we tested from the IPC-4, the number of activation sets is less than $1.2 \cdot n^2$.

4 Local Search in the Space of RA-Graphs

In this section, first we give some background on the stochastic local search procedure used in our approach; then we present new search heuristics for RA-graphs and a method for restricting the search neighborhood.

4.1 Search Procedure and Basic Neighborhood

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

The quality of a RA-graph depends on the number of the flaws it contains, the estimated number of the search steps required to remove them (the *search cost*), and the overall *execution or temporal cost* (depending on the specified plan metric) of the represented plan. In this paper, we focus on the search cost.

The search strategy used by LPG-td is Walkplan, a method similar to the well-known Walksat procedure for solving propositional satisfiability problems [16]. According to Walkplan, the best element in the neighborhood is the RA-graph which has the *lowest decrease of quality* with respect to the current RA-graph, i.e., it does not consider possible improvements.

Walkplan uses a *noise parameter* p to randomize the search. Given a RA-graph $\mathcal{A}$ and a flawed level l, if there is a modification aimed at repairing 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. In addition to the use of the noise parameter, in order to escape local minima, the new version of our planner uses a short *tabu list* ensuring that the last five search states (RA-graphs) are different.

4.2 Search Heuristics based on Relaxed Plans

In [8, 9], we presented some heuristic evaluation functions implemented in the previous version of our planner. 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.

$\mathsf{EvalAdd}(a, l)$

Input: An action node a that does not belong to $\mathcal{A}$ and the earliest flawed level of $\mathcal{A}$;
Output: A set of actions forming a relaxed plan.

1. $I \leftarrow S^R(l_a);\;\; G \leftarrow Unsup(l)$;
2. $Rplan \leftarrow \mathsf{RelaxedPlan}(Pre(a), I, \emptyset)$;
3. $A \leftarrow Rplan \cup \{a\};\;\; I \leftarrow I - Threats(a) \cup Add(a)$;
4. $Rplan \leftarrow \mathsf{RelaxedPlan}(G \cup Threats(a), I, A)$;
5. **return** $Rplan$.

$\mathsf{EvalDel}(a, l)$

Input: An action node a that does not belong to $\mathcal{A}$ and the earliest flawed level of $\mathcal{A}$;
Output: A set of actions forming a relaxed plan.

1. $I \leftarrow S^R(l_a);\;\; G \leftarrow UnsupDel(l)$;
2. $Rplan \leftarrow \mathsf{RelaxedPlan}(Sup(a), I, \emptyset)$;
3. **if** $(l_a < l)$ **then**
4. $\quad Rplan \leftarrow \mathsf{RelaxedPlan}(G, I, Rplan)$;
5. **return** $Rplan$.

Figure 5: Algorithms for estimating the search cost of adding/removing an helpful/harmful action node for the earliest flawed level l of the current RA-graph $\mathcal{A}$.

The first of these differences is an attempt to relax the "flaw-independence assumption" of the previous functions, which in some domains is invalid, and can mislead the search cost evaluation.

The general idea for estimating the search 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, because it is helpful for l in the current RA-graph $\mathcal{A}$. E uses a relaxed plan π to compute an estimate of a minimal set of new action nodes required to support

(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 search cost. In the following, $Pre(a)$ denotes the set of facts corresponding to the flaws of (1), $Unsup(l)$ denotes the sets of facts corresponding to the flaws of (2), and $Threats(a)$ denotes the set of facts corresponding to the flaws of (3). Moreover, we will use some additional notation: $Add(a)$ is the set of the positive effects of the action represented by a, l_x is the level of the node x in the RA-graph, ψ_p denotes the fact represented by the fact node p, and $b^{f\rightarrow c}$ indicates that action node c has a precondition node representing the fact f, which is supported by action node b.

The next definition states more precisely when a fact belongs to $Threats(a)$.

Definition 9 *Given an action node a in a RA-graph $\mathcal{A}$, a fact f is* **threatened** *by a iff the no-op of f and a are mutex, and:*

- *there exist (i) two action nodes $b, c \in \mathcal{A}$ such that $b^{f\rightarrow c}$, $l_b < l_a < l_c$, and (ii) no action node $a' \in \mathcal{A}$ such that $f \in Add(a')$ and $l_a < l_{a'} < l_c$, or*
- *there exist (i) a derived precondition node $p \in \mathcal{A}$ such that $l_p > l_a$, $S(l_p) \models^R \psi_p$, $S(l_p) - \{f\} \not\models^R \psi_p$, and (ii) no action node a' such that $f \in Add(a')$ and $l_a < l_{a'} < l_p$.*[8]

Let $S^R(l)$ indicate the state $S(l) \cup D(S(l), R)$, where R is the set of the domain rules. The relaxed plan consists of two subplans: one for $Pre(a)$, and one for $Unsup(l)$ and $Threats(a)$. The initial state of the first subplan is $S^R(l_a)$, while the initial state of the second is $S^R(l_a)$ modified by the effects of a. Moreover, the second subplan can reuse the actions of the first.

The evaluation of an RA-graph in the search neighborhood that is derived by *removing* an harmful action node a is similar. E uses a relaxed plan π to estimate a minimal set of new action nodes required to support

[8] We use a definition of mutex relation between an action and a no-op which is slightly weaker than the one given by Fox and Long: the no-op of f and an action with positive effect f are not mutex.

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: The 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 */
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 6: Algorithm for computing a relaxed plan achieving a set of action preconditions from the initial state I.

(1) the precondition nodes supported by a (possibly through no-op propagation of its effects and automatic rule activation) that would become *un*supported by removing a;

(2) when l_a precedes the flawed level l under reparation, the unsupported precondition nodes at level l that do not become supported by removing a.[9]

We denote the set of facts corresponding to the precondition nodes of (1) with $Sup(a)$, and the set of facts corresponding to the preconitions nodes of (2) with $UnsupDel(l)$.

More formally, the heuristic evaluation of the RA-graph obtained by adding a helpful action node a ($E(a,l)^i$) or by removing a harmful action node a ($E(a,l)^r$) for a flawed level l is defined as follows:

$$E(a,l)^i = |\pi(a,l)^i| + \sum_{a' \in \pi(a,l)^i} |Threats(a')|$$

$$E(a,l)^r = |\pi(a,l)^r| + \sum_{a' \in \pi(a,l)^r} |Threats(a')|$$

where $\pi(a,l)^i$ and $\pi(a,l)^r$ are sets of actions forming two relaxed plans, and are computed by the algorithms EvalAdd(a, l) and EvalDel(a, l) given in Figure 5, respectively. Such sets are incrementally constructed using the RelaxedPlan subroutine given in Figure 6.

EvalAdd(a, l) runs RelaxedPlan twice, first with goals $Pre(a)$ (step 2), and then with goals $Unsup(l) \cup Threats(a)$ (step 4). EvalDel(a, l) runs RelaxedPlan on $Sup(a)$ (step 2) and, if $l_a < l$, on $UnsupDel(l)$ (step 4).

RelaxedPlan constructs a relaxed plan through a recursive backward process that can reuse a possibly non-empty input set of actions A. The action chosen at step 6 to achieve a *basic* (sub)goal g is an action a' such that (i) g is an effect of a'; (ii) all preconditions of a' are reachable from I; (iii) reachability of the preconditions of a' requires a minimum number of actions, estimated as the maximum of the heuristic number of actions required to support each precondition p of a' from $S^R(l_a)$ ($Num_acts(p, l_a)$); (iv) a' subverts the minimum number of supported precondition nodes of $\mathcal{A}$.

Formally, at step 6 $BestAction(g)$ returns an action satisfying

$$\underset{\{a' \in A_g\}}{ARGMIN} \left\{ \underset{p \in Pre(a') - F}{MAX} Num_acts(p, l_a) + |Threats(a')| \right\},$$

where $A_g = \{a' \in \mathcal{O} \,|\, a \in Add(a)$, $\mathcal{O}$ is the set of all actions, $\forall p \in Pre(a)\ Num_acts(p, l_a) \geq 0\}$; F is the set of positive effects of the actions currently in *Acts* augmented with the facts derived using the domain rules R on $I \cup F$.[10] $Num_acts(p, l_a)$ is computed by *reachability analysis* using a polynomial algorithm similar to the

[9] When we remove a, we remove both a and all its precondition nodes. If $l_a < l$, some precondition nodes at l can remain unsupported.

[10] The set $\mathcal{O}$ does not contain operator instances with mutually exclusive preconditions. In principle, A_g can be empty because g might not be reachable from $S^R(l_a)$ (i.e., $b = \emptyset$). RelaxedPlan treats this special case by forcing its termination and returning a set of actions

Planner	Solved	Attempted	Success ratio	Planning capabilities at IPC-4
LPG-td	845	1074	79%	Propositional + DP, Metric-Temporal +TIL
SGPLAN	1090	1415	77%	Propositional + DP, Metric-Temporal +TIL
P-MEP	98	588	17%	Propositional, Metric-Temporal +TIL
CRIKEY	364	594	61%	Propositional, Metric-Temporal
LPG-IPC3	306	594	52%	Propositional, Metric-Temporal
DOWNWARD (DIAG)	380	432	88%	Propositional + DP
DOWNWARD	360	432	83%	Propositional + DP
MARVIN	224	432	52%	Propositional + DP
YAHSP	255	279	91%	Propositional
MACRO-FF	189	332	57%	Propositional
FAP	81	193	42%	Propositional
ROADMAPPER	52	186	28%	Propositional
TILSAPA	63	166	38%	TIL
OPTOP	4	50	8%	TIL

Table 1: Number of problems attempted, solved, and success ratio of the planners that took part in the 4th IPC. "DP" means derived predicates; "TIL" means timed initial literals; "Propositional" means STRIPS or ADL. The planning capabilities are the PDDL2.2 features in the test problems attended by each planner at IPC-4.

one proposed in [8], which for lack of space we omit. The main difference concerns the treatment of the derived preconditions affecting the reachability information of other basic and derived facts.[11]

When the (sub)goal g is a *derived* fact (step 11), 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

- all facts in H are reachable from I, and their reachability requires a minimum number of actions;
- 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.

More formally, at step 14 $BestActivationSet(\Sigma)$ returns an activation set satisfying

$$\underset{\{H \in \Sigma\}}{ARGMIN} \left\{ \underset{h \in H-F}{MAX} \left[Num_acts(h, l_a) + |\, Threats(a_h)| \right] \right\},$$

where

$$a_h = \underset{\{a' \in A_h\}}{ARGMIN} \left\{ \underset{p \in Pre(a')}{MAX} \; Num_acts(p, l_a) \right\},$$

and A_h is defined analogously to A_g in $BestAction(g)$.

4.3 Neighborhood Restrictions

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, and a less accurate (but faster) evaluation function could be more adequate. Since the basic search neighborhood can be very large, we developed some techniques for restricting it, which are described and experimentally evaluated in [9]. Here we present a new additional restriction technique for both STRIPS domains and domains involving derived predicates. We tested this techniques on the IPC-4 benchmark domains, where it was very effective.

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, the facts in the selected activation set must not be mutex with the preconditions nodes at the flawed level l under consideration. (If this were the case, the truth of the facts in activation set would make impossible to support all precondition nodes at l.)

including a special action with very high execution cost, leading E to consider the element of the neighborhood under evaluation a bad possible next search state [8]. For clarity, we omit these details from the description of the algorithm in Figure 6.

[11] If a (basic or derived) fact p is not reachable, then $Num_acts(p, l_a)$ is set to a negative number.

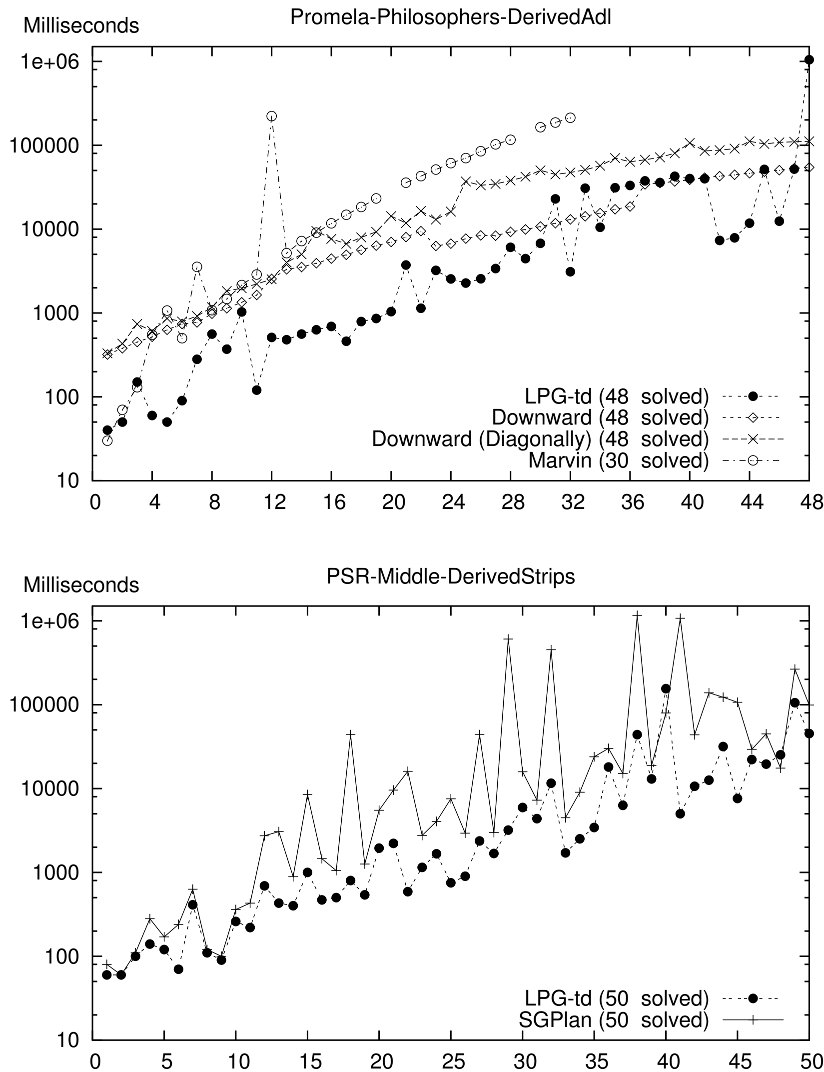


Figure 7: Performance of LPG-td and some IPC-4 planners in two benchmark domains involving derived predicates. On the x-axis we have the problem names (abbreviated by numbers). On the y-axis, we have CPU-time (log scale).

Then, we consider the union of all selected activation sets (one for each d in U) and the set of facts corresponding to the basic precondition nodes that are not supported at l. From the resulting set of facts K, we choose one element k and we restrict the neighborhood $N(l, \mathcal{A})$ of the current RA-graph $\mathcal{A}$ to contain only the RA-graphs of N without the action node at l and the RA-graphs with a new action node supporting k at l.

In order to choose k from K, we use a strategy similar to the "least-commitment flaw-selection" strategy in partial-order causal link planning [15]: *k is the fact that can be supported by the fewest number of graph modifications to $\mathcal{A}$* (either inserting of a helpful action or removal of a harmful action).

5 Experimental Results

The techniques presented in this paper have been implemented in the LPG-td planner, which took part in the fourth International Planning Competition (IPC-4) obtaining the 2nd prize in the suboptimal metric-temporal track, and showing good performance in the suboptimal propositional track. [12] In this section, we present some experimental results illustrating the performance of LPG-td using the test problems of the IPC-4.[13] These problems belong to

[12]The system is available from `http://lpg.ing.unibs.it`.

[13]All tests were conducted on the competition machine, an Intel Xeon(tm) 3 GHz 1 Gbytes of RAM. The CPU-time limit for each test was 30 minutes. We ran LPG-td with the same default settings for every problem attempted.

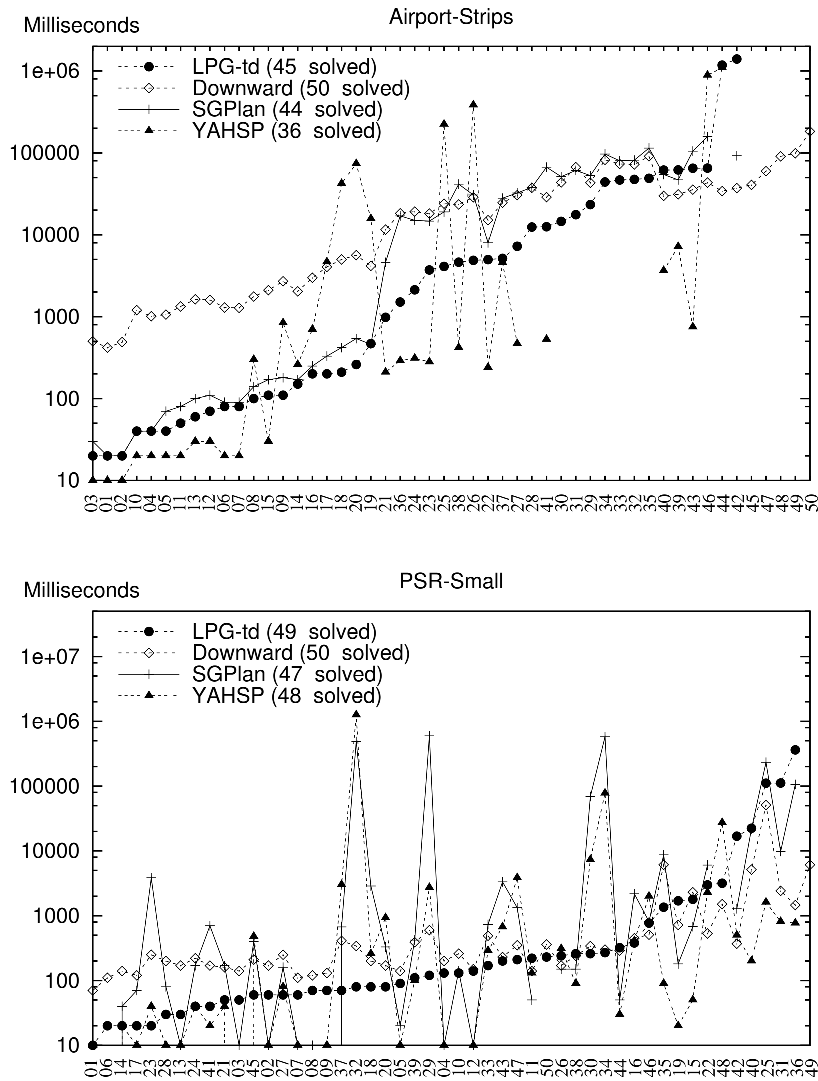


Figure 8: Performance of LPG-td and the other awarded IPC-4 planners in the STRIPS variant of two benchmark domains. On the x-axis we have the problem names (abbreviated by numbers). On the y-axis, we have CPU-time (log scale).

several domains, and each domain has some variants involving different features of PDDL2.1 or PDDL2.2.[14]

Table 1 gives summary results for all the domain variants of IPC-4. LPG-td and SGPLAN are the only planners supporting all the major features of PDDL2.1 and PDDL2.2. Both planners have a good success ratio (close to 80%). DOWNWARD and YAHSP have a success ratio better than LPG-td and SGPLAN, but they handle only propositional domains (the first with derived predicates).

SGPLAN attempted more problems than LPG-td because it was tested also on the "compiled version" of the variants with derived predicates and "timed initial literals" (the new features of PDDL2.2).[15] Moreover, LPG-td did not attempt the numerical variant of the two versions of the `Promela` domain and the ADL variant of `PSR-large`, because they use equality in some numerical preconditions or conditional effects, which currently our planner does not support.

Since our main focus in this paper is planning with derived predicates, we compare LPG-td with the other IPC-4 planners using some variants of the benchmark domains containing derived predicates: `PSR-Middle`, and the two versions of `Promela` (`Philosophers` and `Optical-Telegraph`). We consider only the planners that used the same formalization of the domains, because we believe that comparing the performance when different formalizations are used can be misleading.[16]

[14] For a description and formalization of the IPC-4 benchmark problems and domains, see `http://ls5-www.cs.uni-dortmund.de/~edelkamp/ipc-4/index.html`.

[15] Such versions were generated for planners that do not support these features of PDDL2.2, which are supported by LPG-td.

[16] Most domain variants had both a STRIPS version and an ADL version, which the competitors were free to choose for their tests. In

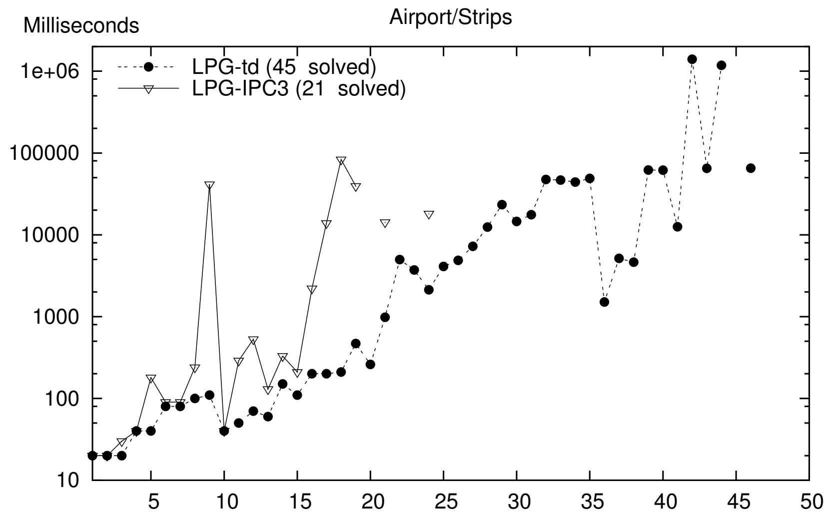


Figure 9: Performance of LPG-td and LPG-IPC3 in the IPC-4 domain `Airport` (STRIPS version). On the x-axis we have the problem names (abbreviated by numbers), while on the y-axis, we have the CPU-time (log scale).

Figure 7 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.

Regarding `Optical-Telegraph` with derived predicates, LPG-td did not perform as well as DOWNWARD, which solved more problems and was generally faster than LPG-td. We believe that the main reason of this is related to the particular structure of these benchmark problems (and of their search space), in which LPG-td's heuristics are less effective, more than to the method used for representing and managing derived predicates.

Figure 8 gives sample results from IPC-4 in the STRIPS variant of two test domains (`Airport` and `PSR-Small`). In these plots the problems on the x-axis are ordered according to their increasing difficulty for LPG-td. Overall, compared to the IPC-4 top-performer STRIPS-planner DOWNWARD, LPG-td is faster in solving most of the problems. However, DOWNWARD performs better in most of the largest `Airport` problems, and in some of the largest PSR problems. In `Airport`, SGPLAN performs slightly worse than LPG-td, while YAHSP performs very well in several problems, but poorly in some others, and overall solves fewer problems than LPG-td. In PSR, overall SGPLAN and YAHSP perform similarly to LPG-td.

Concerning plan quality, the results of an analysis of the official results of IPC-4 show that the quality of the plans produced by LPG-td is generally better than the plan quality of the other planners, in terms of both the number of plan actions and the plan metric in the problem specification. (These results are available in the web page of LPG `http://lpg.ing.unibs.it`.)

The summary results of Table 1 show that, the success ratio of LPG-IPC3 (the version of our planner that was awarded at the previous competition), is significantly lower than the one of LPG-td. The main reasons of this improvement are the revised search and neighborhood restriction heuristics that we have presented in this paper.

Figure 9 compares the CPU-time of LPG-IPC3 and LPG-td for the STRIPS variant of AIRPORT. LPG-td is up to two orders of magnitude faster than the previous version of the planner, and solves many more problems. Finally, we compared LPG-td and LPG-IPC3 also using the benchmark domains of IPC-3, and overall the new version of the planner was significantly faster (for more details, see the web page of LPG).

6 Conclusions

We have presented some new techniques for planning in domains involving derived predicates, an important feature supported by the recent PDDL2.2 language.

Our methods extend the "planning through action graphs and local search" approach previously developed and implemented in the LPG planner by (i) including a rule graph to support a simple form of reasoning about derived predicates in the search states produced by the plan actions; (ii) augmenting the action graph representation with

`Philosophers`, LPG-td attempted the ADL version, SGPLAN the STRIPS version; in PSR, LPG-td attempted the STRIPS version, DOWNWARD and MARVIN the ADL version.

additional nodes and arcs representing (automatically triggered) domain rules; (iii) defining a new search space formed by such rule-augmented action graphs, (iv) designing new heuristics based on relaxed plans to guide a local search process, and to restrict the search neighborhood for speeding up the search.

Our techniques are implemented in a new version of LPG, which showed good performance in many benchmark problems from IPC-4 and IPC-3. Current and future work includes a more detailed analysis of the empirical results of IPC-4, the study of further heuristics to improve the search neighborhood evaluation, and more effective techniques for selecting the best activation set of a derived precondition.

References

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

[2] 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*, The University of Washington, Computer Science Department.

[3] Chen, Y., Hsu, C,. and Wha, W. 2004. SGPlan: Subgoal Partitioning and Resolution in Planning *In Abstract Booklet of the competing planners of ICAPS-04*.

[4] Coles, A., and Smith, A. 2004. Marvin: Macro Actions from Reduced Versions of the Instance *In Abstract Booklet of the competing planners of ICAPS-04*.

[5] 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.

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

[7] Gerevini, A., and Serina, I. 2002. LPG: A planner based on local search for planning graphs with action costs. In *Proc. of AIPS-02*.

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

[9] Gerevini, A., Saetti, A., and Serina, I. 2004. An Empirical Analysis of Some Heuristic Features for Local Search in LPG. *Proc. of ICAPS-04*.

[10] Helmert, M. 2004. A Planning Heuristic Based on Causal Graph Analysis. *Proc. of ICAPS-04*.

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

[12] Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Velosa, M., Weld, D., and Wilkins, D. 1998. PDDL - The Planning Domain Definition Language *T.R. CVC TR98-003/DCS TR-1165*, Yale Center for Computational Vision and Control

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

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

[15] Pollack, M.E. and Joslin, D. and Paolucci, M. Flaw Selection Strategies for Partial-Order Planning. *JAIR* 6:223–262. 1997.

[16] Selman, B., Kautz, H., and Cohen, B. 1994. Noise strategies for improving local search. In *Proc. of AAAI-94*.

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