Least Commitment in Graphplan

Michel Cayrol Pierre Régnier Vincent Vidal

IRIT
Université Paul Sabatier
118, route de Narbonne
31062 TOULOUSE Cedex 04, FRANCE
{cayrol, regnier, vvidal}@irit.fr

Abstract

Planners of the Graphplan family (Graphplan, IPP, STAN...) are currently considered to be the most efficient ones on numerous planning domains. Their partially ordered plans can be represented as sequences of sets of actions. The sets of actions generated by Graphplan satisfy a strong independence property which allows one to manipulate each set as a whole. We present a detailed formal analysis that demonstrates that the independence criterion can be partially relaxed in order to produce valid plans in the sense of Graphplan. Indeed, two actions at a same level of the planning-graph do not need to be marked as mutually exclusive if there exists a possible ordering between them that respects a criterion of "authorization", less constrained than the criterion of independence. The ordering between the actions can be set up after the plan has been generated, and the extraction of the solution plan needs an extra checking process that guarantees that an ordering can be found for actions considered simultaneously, at each level of the planning-graph. This study lead us to implement a modified Graphplan, LCGP (for "Least Committed GraphPlan"), which is still sound and complete and generally produces plans that have fewer levels than those of Graphplan (the same number in the worst cases). We present an experimental study which demonstrates that, in classical planning domains, LCGP solves more problems than planners from the family of Graphplan (Graphplan, IPP, STAN...). In most cases, LCGP also outperforms the other planners.

1 Introduction

1.1 Generalities

In recent years, the development of a new family of planning systems based on the planner Graphplan [2; 3] has lead to numerous evolutions in planning. Graphplan develops, level after level, a compact search space called a planning-graph. During this construction stage, it does not use all the relations among state variables or actions that are taken into account in other planning techniques like state space search or search in the space of partial plans. In Graphplan, these constraints are only computed and memoized at each level as mutual exclusions, so that the planning-graph can be seen as a Dynamic CSP [10; 11; 18; 23]. The search space is easier to develop, but a second stage, called the extraction stage, is necessary in order to try to extract a valid plan from the planning-graph and the sets of mutual exclusions.

Several techniques have been employed to improve Graphplan: reduction of the search space before the extraction stage [7; 19], improvement of the domain representation language [9; 14; 15; 22; 23], improvement of the extraction stage [8; 10; 11; 13; 16; 24]. In all these works, the structure of the generated plans remains the same whatever the graph construction method is. A plan of Graphplan can thus be represented as a minimal sequence of sets of actions considered simultaneously: each step of the algorithm produces a level of the planning-graph, each level being connected to a set of actions of the plan.

Every set Q of actions that appears in a sequence produced by Graphplan is such that the computation of the final situation E_f, produced by the application of the actions of Q, starting from an initial situation E_i, is independent of the order in which these actions are applied. This is because all the sets of actions kept by Graphplan during the extraction stage verify a property I (Independence), easy to test, that permit them to be excuted in parallel. The final situation E_f is directly computed from the initial situation E_i and from the *global* application of the actions of the set Q.

We have established another property A (Authorization) which is less restrictive than I (I implies A) and easier to verify. This property guarantees the existence of at least one serialization S of the actions of Q (but does not require its computation). The application of this sequence to an initial situation E_i can still be computed as if the sets of actions verified the independence property. The final situation E_f can still be considered to be the result of the *global* application of the actions of Q, but these actions cannot be always executed in parallel: a valid ordering (that can contain parallel actions) must be found.

We have developed a Graphplan-like planner called LCGP (Least Committed Graphplan, see [4; 5]) which works in the same way: it incrementally constructs a stratified graph, then searches it to extract a plan. The graph that Graphplan would have built is a subgraph of LCGP graph (cf. example below) at the same level. So, goals generally appear sooner (at the same time in the worst cases). LCGP then transforms the produced plan into a Graphplan-like plan. The faster computation of a solution has a cost: the plans obtained with LCGP may not be optimal, in the sense that they can have more levels than the ones produced by Graphplan. In practice, LCGP rapidly gives a solution on many classical benchmarks (Logistics, Blocks-world, Ferry...) where Graphplan is unable to produce a plan after a significant running time.

1.2 An example

We introduce below, on a small example, the basic idea of the authorization relation and the changes it implies for Graphplan. Let us first recall informally the basic elements of Graphplan. Objects of the world are represented by ground atoms (called propositions), states of the world are lists of propositions, and actions are triples of lists of propositions: preconditions (propositions that must be present in the state before the execution of the action), add effects (propositions added to the state) and del effects (propositions deleted from the state). The Graphplan algorithm first builds the planning-graph, a stratified graph that interleaves two kinds of nodes: proposition nodes and action nodes. These nodes are collected into levels, that contain one set of proposition nodes and one set of action nodes each. The first level only contains the propositions of the initial state of the world. The second level contains the actions that are applicable in the initial state (actions that have all their preconditions present in the initial state), and the no-ops for every proposition in the initial state. A no-op is an action whose precondition is a proposition and add effect is the same proposition. No-ops permit Graphplan to solve the frame-problem: a proposition not deleted by an action will be present in the next state if the corresponding no-op is applied. The second level also contains the add effects of every action in the second level (including the propositions of the initial state, thanks to the no-ops). At each level, binary mutual exclusions are recorded. Two propositions are mutually exclusive when every pair of actions that produce them are mutually exclusive, and two actions are mutually exclusive when they have mutually exclusive preconditions or when they are not independent. Two actions are independent when neither of them deletes a precondition or an add effect of the other. Actions will not be added in the next level if any of their preconditions are mutually exclusive. This process continues until the following property is verified: either all the propositions of the goal are present in the last level and none of the goal propositions are mutually exclusive, or the problem is proved to be unsolvable with respect to a more complex property (cf. [3] for details). If the problem is not unsolvable, then a solution can be extracted by a backward chaining algorithm that will be briefly described in the example below. If no solution is found, the planning-graph is extended with one more level and the extraction stage starts again until a solution is found or the problem is proved to be unsolvable.

Now comes an example that illustrates the difference between Graphplan and LCGP. The set of propositions is $P = \{a, b, c, d\}$ and the set of actions is $A = \{A, B, C\}$, with:

Preconditions of $A = \{a\}$	Preconditions of $B = \{a\}$	Preconditions of $C = \{b, c\}$
Add effects of $A = \{b\}$	Add effects of $B = \{c\}$	Add effects of $C = \{d\}$
Del effects of $A = \{\}$	Del effects of $B = \{a\}$	Del effects of $C = \{\}$

The initial state of the problem is $I = \{a\}$, and the goal is $G = \{d\}$. The planning-graph of Graphplan is depicted in Figure 1. A full black line from a proposition to an action (from left to right) represents a precondition link, and from an action to a proposition it represents an add effect. Dashed lines represent del effects, and grey lines represent no-ops.

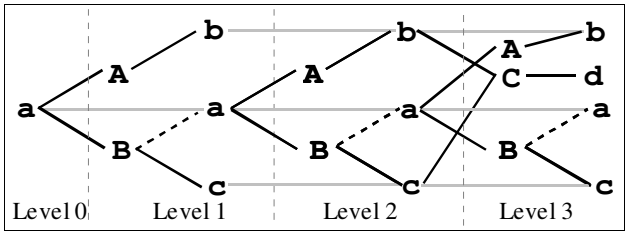

Figure 1: The planning-graph of Graphplan

The actions A and B are always mutually exclusive, because they are not independent: B deletes a which is a precondition of A. At level 1, the pairs of mutually exclusive propositions are $\{a, c\}$ and $\{b, c\}$. So, the action C cannot be used at level 2 to produce the goal. At this level, b and c do not remain mutually exclusive,

because the no-op of b and the action B are not mutually exclusive. At level 3, the action C can be applied and the goal d appears.

The solution can now be extracted by the backward chaining algorithm. A goal list is created that only contains d. The algorithm looks for actions that assert the propositions of the goal list at level 3. The only action that asserts d is C. The goal list is now the union of the preconditions of every considered actions: $\{b, c\}$. The algorithm records that C belongs to the current plan at level 3, and looks for actions at level 2 that assert the proposition of the goal list. Some pairs of actions cannot be chosen, because they are mutually exclusive: $\{A, B\}$, $\{A$, no-op of $c\}$, {no-op of b, no-op of $c\}$. The only possible choice is {no-op of b, $B\}$ which is recorded in the current plan at level 2. The goal list is now $\{b, a\}$, and the only possible actions at level 1 are $\{A$, no-op of $a\}$. The produced plan (without no-ops, which are only useful for the construction of the graph and the extraction stage) is $\langle A, B, C\rangle$.

The authorization relation is a partial relaxation of the independence relation, and is not symmetrical. An action A authorizes an action B when A does not delete a precondition of B and B does not delete an add effect of A, so the action B can be applied after the action A and the resulting state contains the union of the add effects of A and B. Two actions are now mutually exclusive when they have mutually exclusive preconditions or when neither of these two actions authorizes the other. Figure 2 depicts the planning-graph of LCGP.

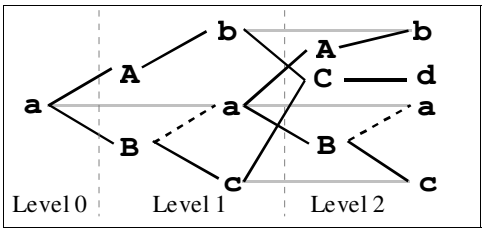

Figure 2: The planning-graph of LCGP

With this new definition, A and B are not mutually exclusive any more, because A authorizes B. Thus, at level 1, the propositions b and c are not mutually exclusive, and the action C can be applied at level 2. The goal d now appears at level 2 and the extraction stage can be performed. The goal list is initialized with $\{d\}$, and the algorithm looks for an action that asserts d: the action C is then recorded in the current plan at level 2. The goal list is now the preconditions of C: $\{b, c\}$. The only actions that assert the propositions of the goal list are now the actions A and B which are not mutually exclusive as before, because A authorizes B. The algorithm must perform an extra test, to ensure that an ordering of these actions can be found. As we have only two actions, the ordering is obvious and the produced plan is the one produced by Graphplan: $\langle A, B, C\rangle$.

A problem can occur when the algorithm considers at least three actions simultaneously. Indeed, if we consider the three actions C, D, E such that C authorizes D but D does not authorize C, D authorizes E but E does not authorize D, and E authorizes C but C does not authorize E, no ordering of $\{C, D, E\}$ can be found: $\langle C, D, E\rangle$ and $\langle C, E, D\rangle$ are impossible because C does not authorize E; and with a circular permutation of these two orderings, all the other orderings are impossible.

An example of a classical benchmark domain in which this problem can occur is the Blocks-world domain with the following actions: MoveFromTable(A, B), MoveFromTable(B, C) and MoveFromTable(C, A). In the initial state, the three blocks A, B and C are on the table, and the goal is {on(A, B), on(B, C), on(C, A)} which is obviously impossible; but that needs to be proved by the planner. The three actions described above produce the propositions of the goal, and there is no pair of mutually exclusive actions. Indeed, MoveFromTable(B, C) authorizes MoveFromTable(A, B): after moving B on C, it is still possible to move A on B. But MoveFromTable(A, B) does not authorizes MoveFromTable(B, C): after moving A on B, B is not clear, so it cannot be taken (a block can be moved only when it is clear). We are exactly in the case described above with the actions C, D, E. It is important to note that the search for an ordering can be performed in polynomial time by a topological sort algorithm.

In Section 2, we present a formal analysis of the independence and authorization relations. In Section 3 we describe the way to modify Graphplan in order to produce authorization based plans, and to transform them into independence based plans. In Section 4 we show experimental results that compare the efficiency of our approach to that of classical Graphplan. Related work is discussed in Section 5, and our conclusions are given in Section 6.

2 Formalization

First, we formalize the structure of the plans of Graphplan (cf. § 2.1). Then, we suggest that Graphplan, using the independence criterion, over-constrains the choice of the actions into the sets of actions considered simultaneously (cf. § 2.2). We then demonstrate that this criterion can be relaxed in order to obtain plans with a different structure. These plans can be easily transformed into plans that Graphplan could have produced (cf § 2.3), and which lead to the same resulting state.

2.1 Semantics and formalization of the plans of Graphplan

The most important element of a plan is an *action*, which is an instance of an *operator*. In Graphplan, operators are Strips-like operators, without negation in their preconditions. We use a first order logic language L, constructed from the vocabularies Vx, Vc, Vp that respectively denote finite disjoint sets of symbols of variables, constants and predicates. We do not use function symbols.

Definition 1 (operator):

An *operator*, denoted by o, is a triple $\langle pr, ad, de\rangle$ where pr, ad and de denote finite sets of atomic formulas of the language L. Prec(o), Add(o) and Del(o) respectively denote the sets pr, ad and de of the operator o. O denotes the finite set of operators.

Definition 2 (state, proposition):

A *state* is a finite set of ground atomic formulas (i.e. without any variable symbols). A ground atomic formula is also called a *proposition*. P denotes the set of all the propositions that can be constructed with the language L.

Definition 3 (action):

An *action* denoted by a is a ground instance $o\theta = \langle pr\theta, ad\theta, de\theta\rangle$ of an operator o which is obtained by applying a substitution θ defined with the language L such that $pr\theta$, $ad\theta$ and $de\theta$ are ground and $ad\theta$ and $de\theta$ are disjoint sets. Prec(a), Add(a), Del(a) respectively denote the sets $pr\theta$, $ad\theta$, $de\theta$ and represent the preconditions, adds and deletes of a. A denotes the finite set of actions, which are all the possible ground instanciations of the operators of O.

The main structure we will use in the following, the *sequence of sets of actions*, will represent the plans of Graphplan and LCGP: it defines the order in which the sets of actions are considered from the point of view of the execution of the actions they contain. All sequences and sets of actions will be finite. A sequence of sets of actions S is noted $\langle Q_i\rangle_n$, with $n \in \mathbb{N}$. If $n = 0$, S is the empty sequence: $S = \langle Q_i\rangle_0 = \langle\rangle$; if $n > 0$, S can be noted $\langle Q_1, Q_2, ..., Q_n\rangle$. If the sets of actions are singletons (i.e. $Q_1 = \{a_1\}$, $Q_2 = \{a_2\}$, ..., $Q_n = \{a_n\}$), the associated sequence of sets of actions is called a *sequence of actions* and will be noted[1] $\langle a_1, a_2, ..., a_n\rangle$. The set of sequences of sets of actions formed from the set of actions A is denoted by $(2^A)^*$. The set of sequences of actions formed using the set of actions A is denoted by A^*.

Definition 4 (first, rest, length):

We define the classical functions *first* and *rest* on non-empty sequences as $\text{first}(\langle Q_1, Q_2, ..., Q_n\rangle) = Q_1$, $\text{rest}(\langle Q_1, Q_2, ..., Q_n\rangle) = \langle Q_2, ..., Q_n\rangle$, and *length* on all sequences as $\text{length}(\langle Q_i\rangle_n) = n$.

Definition 5 (concatenation of sequences of sets of actions):

Let S, $S' \in (2^A)^*$ be two sequences of sets of actions with $S = \langle Q_i\rangle_n$ and $S' = \langle Q'_i\rangle_m$. The *concatenation* (noted $\oplus$) of S and S' is defined by:

$$S \oplus S' = (\text{if } n+m = 0 \text{ then } \langle\rangle \text{ else } \langle R_i\rangle_{n+m}, \text{ with } R_i = (\text{if } 1 \le i \le n \text{ then } Q_i \text{ else } Q'_{i-n})).$$

Definition 6 (linearization):

A *linearization* of a set of actions $Q \in 2^A$ with $Q = \{a_1, ..., a_n\}$ is a permutation of Q. The set of all the linearizations of Q is denoted by Lin(Q).

Notations: If Q is the set of actions $Q = \{a_1, ..., a_n\}$, then:

- the union of the preconditions of the elements of Q is noted Prec(Q): $\text{Prec}(Q) = \text{Prec}(a_1) \cup ... \cup \text{Prec}(a_n)$,
- the union of the adds of the elements of Q is noted Add(Q): $\text{Add}(Q) = \text{Add}(a_1) \cup ... \cup \text{Add}(a_n)$,
- the union of the deletes of the elements of Q is noted Del(Q): $\text{Del}(Q) = \text{Del}(a_1) \cup ... \cup \text{Del}(a_n)$.

We use the same notation for sequences of actions $Q = \langle a_1, ..., a_n\rangle$.

[1] There should be no confusion since it should be clear from the context if we mean a sequence of actions or a sequence of sets of actions.

Like the majority of partial order planners (UCPOP, SNLP...), Graphplan strongly constrains the choice of actions in order to ensure that a parallel and a sequential execution of a plan yield the same resulting state. To achieve this result using a Strips-like description of actions, every action in a set must be independent of the others, i.e. their effects must not be contradictory (no action can delete an add effect of another) and they must not interact (no action can delete a precondition of another).

Definition 7 (independence):

Two actions $a_1 \neq a_2 \in A$ are *independent* iff:

$$(\text{Add}(a_1) \cup \text{Prec}(a_1)) \cap \text{Del}(a_2) = \emptyset \text{ and } (\text{Add}(a_2) \cup \text{Prec}(a_2)) \cap \text{Del}(a_1) = \emptyset.$$

A set of actions $Q \in 2^A$ is an *independent set* iff the actions of this set are pairwise independent, i.e.

$$\forall\ a_1 \neq a_2 \in\ Q,\ (\text{Prec}(a_1) \cup \text{Add}(a_1)) \cap \text{Del}(a_2) = \emptyset.$$

Notice that for two actions to be applicable in parallel, another condition must be true: they must not have incompatible preconditions. Graphplan and LCGP detect and take advantage of these incompatibilities.

A sequence $\langle Q_1, ..., Q_n \rangle$ of sets of actions partially defines the order of execution of the actions. The end of the execution of each action in Q_i must precede the beginning of the execution of each action in Q_{i+1}. This implies that the execution of all the actions in Q_i precedes the execution of all the actions in Q_{i+1}.

Let us formalize a plan of Graphplan, by defining an application to simulate the execution of a sequence of sets of actions from an initial representation of the world. If a sequence of sets of actions cannot be applied to a state, the result will be $\perp$, the impossible state.

Definition 8 (application of a sequence of independent sets of actions):

Let $\Re$: $(2^P \cup \{\perp\}) \times (2^A)^* \rightarrow (2^P \cup \{\perp\})$, defined as:

$E\ \Re\ S =$

If $S = \langle\rangle$ or $E = \perp$

then E

else If first(S) is **independent** and $\text{Prec}(\text{first}(S)) \subseteq E$

then $[(E - \text{Del}(\text{first}(S))) \cup \text{Add}(\text{first}(S))]\ \Re\ \text{rest}(S)$

else $\perp$.

Definition 9 (plan in relation to $\Re$):

A sequence of sets of actions $S \in (2^A)^*$ is a *plan* for a state $E \in (2^P \cup \{\perp\})$, in relation to $\Re$, iff $E\ \Re\ S \neq \perp$.

When $E\ \Re\ S \neq \perp$, we can associate a semantics to S that is connected with the execution of actions in the real world, because we are sure (in a static world) that our prediction of the final state is correct. In this case, we say that S is recognized by $\Re$.

Theorem 1 establishes the essential property of Graphplan: *the actions of a plan of Graphplan that can be executed in parallel give the same result when they are executed sequentially, whatever the order of execution is.*

Theorem 1:

Let $E \in (2^P \cup \{\perp\})$ be a state and $S \in (2^A)^* - \{\langle\rangle\}$ a sequence of sets of actions, with $S = \langle Q_1, ..., Q_n \rangle$. Then:

$$E\ \Re\ S \neq \perp \Rightarrow \forall\ S_1 \in \text{Lin}(Q_1), ..., \forall\ S_n \in \text{Lin}(Q_n),\ E\ \Re\ S = E\ \Re\ (S_1 \oplus ... \oplus S_n).$$

Now, we are going to question this property. We can remark that $E\ \Re\ \langle\{a_1, ..., a_n\}\rangle = E\ \Re\ \langle a_1, ..., a_n \rangle$ when $\forall\ i \in [1, n-1]$, $\text{Del}(a_{i+1}) \cap (\text{Add}(a_1) \cup ... \cup \text{Add}(a_i)) = \emptyset$ and $\forall\ i \in [1, n-1]$, $\text{Prec}(a_{i+1}) \cap (\text{Del}(a_1) \cup ... \cup \text{Del}(a_i)) = \emptyset$. In this case, we can see that $E\ \Re\ \langle a_1, ..., a_n \rangle$ can be computed without knowing the order of the actions of the sequence $\langle a_1, ..., a_n \rangle$.

2.2 Towards a new structure for plans

Graphplan imposes very strong conditions on the plans using the independence property to choose the actions to consider simultaneously. So, it is always possible to execute these actions in parallel. Now, we demonstrate that we can modify this property to relax a part of the constraints on actions of the same set and still produce plans.

When we do this modification, we can no longer be sure that the actions in a set of actions (actions at a same level) can be executed in parallel because they may not be independent. However, the main idea of Graphplan is preserved because each of our new sets of actions can still be used as a whole: we always try to establish all the preconditions of all the actions in a set using the effects of the actions that belong to another set of actions (at the preceding level).

By relaxing a part of the constraints on independent actions, we define a more flexible relation (asymmetrical) between the actions: the *authorization relation*. An action a_1 authorizes an action a_2 if a_2 can be executed at the same time or after a_1 with the same resulting state. In order to achieve this result a property weaker than independence is sufficient: a_1 must not delete a precondition of a_2 (a_2 must still be applicable after a_1 has been executed) and a_2 must not delete a fact added by a_1. This definition implies an order for the execution of two actions: a_1 authorizes a_2 means that if a_1 is executed before a_2, the preconditions of a_2 will not be deleted by the execution of a_1 and the add effects of a_1 will not be deleted by the execution of a_2. On the other hand if a_1 does not authorize a_2 and if we execute a_1 before a_2, either a_2 deletes an add effect of a_1 (so the resulting state cannot be computed), or a precondition of a_2 is deleted by a_1 (so we cannot execute a_2).

Definition 10 (authorization):

An action $a_1 \in A$ *authorizes* an action $a_2 \in A$ (noted $a_1 \angle a_2$) iff (1) $a_1 \neq a_2$ and (2) $\text{Add}(a_1) \cap \text{Del}(a_2) = \emptyset$ and $\text{Prec}(a_2) \cap \text{Del}(a_1) = \emptyset$. An action a_1 *forbids* an action a_2 iff the action a_1 does not authorize a_2, i.e. if $\text{not}(a_1 \angle a_2)$.

This authorization relation leads to a new definition of the sets that can belong to a plan. These sets will no longer be independent sets. For every set of actions, we want to find at least one linearization that could be a plan. Such a linearization introduces a notion of order among actions.

Definition 11 (authorized sequence):

A sequence of actions $\langle a_i \rangle_n \in 2^A$ is *authorized* iff $\forall\ i, j \in [1, n],\ i < j \Rightarrow a_i \angle a_j$, which leads to:

$$\forall\ i \in [1, n-1],\ \text{Del}(a_{i+1}) \cap (\text{Add}(a_1) \cup ... \cup \text{Add}(a_i)) = \emptyset \text{ and } \text{Prec}(a_{i+1}) \cap (\text{Del}(a_1) \cup ... \cup \text{Del}(a_i)) = \emptyset.$$

Definition 12 (authorized set of actions, authorized linearizations):

A set of actions $Q \in (2^A)^*$ is *authorized* iff one can find an authorized linearization $S \in \text{Lin}(Q)$, otherwise it is *forbidden*. We will note $\text{LinA}(Q)$ the set of all the authorized linearizations of Q:

$$\text{LinA}(Q) = \{S \in \text{Lin}(Q) \mid S \text{ is an authorized linearization}\}.$$

So, a set of actions is authorized if one can find an order among the actions of the set such that no action in the set deletes either an add of a preceding action or a precondition of a following action.

Let us define $\mathfrak{R}^*$, a new application of a sequence of sets of actions to a state that uses the authorization relation between actions. Our planner LCGP will be based on $\mathfrak{R}^*$. With this definition, we can demonstrate a new theorem to compute the resulting state (Theorem 2). This theorem does not use all the linearizations of the independent sets of actions but only the linearizations that respect the authorization constraints among actions of the sets (authorized linearizations).

Definition 13 (application of a sequence of authorized sets of actions):

Let $\mathfrak{R}^*: (2^P \cup \{\bot\}) \times (2^A)^* \rightarrow (2^P \cup \{\bot\})$, defined as:

$E\ \mathfrak{R}^*\ S =$

If $S = \langle\rangle$ or $E = \bot$

then E

else If $\text{first}(S)$ is **authorized** and $\text{Prec}(\text{first}(S)) \subseteq E$

then $[(E - \text{Del}(\text{first}(S))) \cup \text{Add}(\text{first}(S))]\ \mathfrak{R}^*\ \text{rest}(S)$

else $\bot$.

Definition 14 (plan in relation to $\mathfrak{R}^*$):

A sequence of sets of actions $S \in (2^A)^*$ is a *plan* for a state $E \in (2^P \cup \{\bot\})$ in relation to $\mathfrak{R}^*$ iff $E\ \mathfrak{R}^*\ S \neq \bot$.

The theorem below says that all applications of the authorized linearizations of the sets of actions of a plan recognized by $\mathfrak{R}^*$ give the same result. It is close to Theorem 1 and has a similar proof.

Theorem 2:

Let $E \in (2^P \cup \{\bot\})$ be a state and $S \in (2^A)^* - \{\langle\rangle\}$ a sequence of sets of actions, with $S = \langle Q_1, ..., Q_n \rangle$. Then:

$$E\ \mathfrak{R}^*\ S \neq \bot \Rightarrow \forall\ S_1 \in \text{LinA}(Q_1), ..., \forall\ S_n \in \text{LinA}(Q_n),\ E\ \mathfrak{R}^*\ S = E\ \mathfrak{R}^*\ (S_1 \oplus ... \oplus S_n).$$

2.3 Relations between the formalisms

The independence and authorization relations are strongly related. The next theorem says that a plan for $\mathfrak{R}$ is a plan for $\mathfrak{R}^*$:

Theorem 3:

Let $E \in (2^P \cup \{\bot\})$ be a state and $S \in (2^A)^*$ a sequence of sets of actions. Then:

$$E\ \mathfrak{R}\ S \neq \bot \Rightarrow E\ \mathfrak{R}^*\ S = E\ \mathfrak{R}\ S.$$

It follows that if a sequence of sets of actions S is not a plan for a situation E in relation to $\mathfrak{R}^*$, it is not a plan for E in relation to $\mathfrak{R}$ either:

Corollary 1:

Let $E \in (2^P \cup \{\bot\})$ be a state and $S \in (2^A)^*$ a sequence of sets of actions. Then:

$$E\ \mathfrak{R}^*\ S = \bot \Rightarrow E\ \mathfrak{R}\ S = \bot.$$

There is another connection between the plans recognized by $\mathfrak{R}$ and the plans recognized by $\mathfrak{R}^*$: all the plans constructed using the authorized linearizations of the sets of actions of a plan recognized by $\mathfrak{R}^*$, are recognized by $\mathfrak{R}$. Moreover, the application of $\mathfrak{R}^*$ to the original plan produces the same resulting state as the application of $\mathfrak{R}$ on every plan constructed using the authorized linearizations of the sets of actions of the plan.

Theorem 4:

Let $E \in (2^P \cup \{\bot\})$ be a state and $S \in (2^A)^* - \{\langle\rangle\}$ a sequence of sets of actions, with $S = \langle Q_1, \ldots Q_n\rangle$. Then:

$$E\ \mathfrak{R}^*\ S \neq \bot \Rightarrow \forall\ S_1 \in \mathrm{LinA}(Q_1), \ldots, \forall\ S_n \in \mathrm{LinA}(Q_n), E\ \mathfrak{R}^*\ S = E\ \mathfrak{R}\ (S_1 \oplus \ldots \oplus S_n).$$

This theorem is essential and gives a meaning to the plans recognized by $\mathfrak{R}^*$: an elementary transformation (the search of an authorized linearization of every set of actions) produces a plan recognized by $\mathfrak{R}$ (and that Graphplan would have produced).

3 Integration of this new structure of plans in Graphplan

Now, we will explain the modifications to Graphplan to implement this new formalism in LCGP. Recall that a planning-graph is a graph consisting of successive levels, each one marked with a positive integer and containing a set of actions and a set of propositions. The level 0 is an exception and only contains propositions representing facts of the initial state.

3.1 Extending the planning-graph

During this stage, the only difference between Graphplan and LCGP involves the computation of the exclusion relations between actions. In Graphplan, two actions a_1 and a_2 are mutually exclusive iff (1) $a_1 \neq a_2$ and (2) they are not independent (i.e. one of them forbids the other: not($a_1 \angle a_2$) **or** not($a_2 \angle a_1$)), or if a precondition of one is mutually exclusive with a precondition of the other. In LCGP, the exclusion relation between actions is thus defined:

Definition 15 (mutual exclusion):

Two actions $a_1, a_2 \in A$ are mutually exclusive iff (1) $a_1 \neq a_2$ and (2) each of them forbids the other: not($a_1 \angle a_2$) **and** not($a_2 \angle a_1$), or if a precondition of one is mutually exclusive with a precondition of the other.

This new definition of the mutual exclusion (**or** in Graphplan, **and** in LCGP), implies that LCGP finds fewer mutually exclusive pairs of actions than Graphplan (the same number in the worst cases). Consequently, a level n of LCGP will include more actions and propositions than a level n of Graphplan (cf. example of § 1) because actions can sometimes be applied earlier in LCGP (given a level n, the graph of Graphplan is a subgraph of the one for LCGP). The graph of LCGP grows faster and contains, for the same number of levels, more potential plans than the graph of Graphplan (the same number in the worst cases). The extension of the graph finishes earlier too because the goals generally appear before being produced by Graphplan (at the same level in the worst cases).

3.2 Searching for a plan

After the construction stage, Graphplan tries to extract a solution from the planning-graph, using a level-by-level approach. It begins with the set of propositions constructed at the last level (that includes the goals) and inspects the different sets of actions that assert the goals. It chooses one of them (backtrack point) and searches again, at the previous level, for the sets of actions that assert the preconditions of these actions. At each level, the actions of the chosen set must be pairwise independent and their preconditions must not be mutually exclusive to be in agreement with the associated semantics (parallel actions, cf. § 2.1). So, Graphplan tests, using the exclusion relations, that there is no pair of mutually exclusive actions.

In LCGP, even when there is no mutual exclusion, it is not guaranteed that a set of actions can be kept for a plan (cf. Example of Blocks−world domain in §1.2). This set must also be authorized (cf. Definition 12), i.e. one must find a sequence of actions (authorized sequence) such that no action deletes a precondition of a following action or an add effect of a previous action of the sequence. This condition can be verified using a

modified topological sort algorithm (linear in the number of arcs and nodes [17]) that tests if the directed graph defined below is acyclic:

Definition 16 (authorization graph):

Let $Q \in 2^A$ be a set of actions, with $Q = \{a_1, ..., a_n\}$. The *authorization graph* AG(N, C) of Q is an oriented graph defined by:

- $N = \{n(a_1), ..., n(a_n)\}$ is the set of nodes containing one node, $n(a_i)$, for each action $a_i \in Q$,
- C is the set of arcs that represent the order constraints among actions: there is an arc from $n(a_i)$ to $n(a_j)$ iff the execution of a_i **must precede** the execution of a_j, i.e. if a_j forbids a_i:

$$\forall\ a_i \neq a_j \in Q,\ (n(a_i), n(a_j)) \in C \Leftrightarrow \text{not}(a_j \angle a_i).$$

Indeed, we can demonstrate that:

Theorem 5:

Let $Q \in 2^A$ be a set of actions and AG(N, C) the authorization graph of Q. Then:

$$\text{AG has no cycle} \Leftrightarrow Q \text{ is authorized.}$$

We use the algorithm **SearchSeq** below to prove that a set of actions is authorized. This algorithm not only returns the answer to the question "is this set of actions Q authorized ?" (cf. Theorem 6, below); it also returns a sequence of independent sets of actions S such that $E\ \Re^*\ \langle Q \rangle = E\ \Re\ S$ (cf. Theorem 7, § 3.3.1). We divided the algorithm into two procedures, because the second one (**Stratify**) will be used later by the algorithm which computes the optimal reordering of a plan.

SearchSeq(Q)
;; Input:
;; – Q: a set of actions
;; Output:
;; – *fail* if Q is not authorized,
;; – else: sequence of sets of actions S such that $E\ \Re^*\ \langle Q \rangle = E\ \Re\ S$.
Begin
  Let AG(N, C) := the authorization graph of Q
  Return Stratify(AG)
End {SearchSeq}

Stratify(G)
;; Input:
;; – G(N, C): a directed graph. N is the set of nodes associated to actions, C is the set of arcs.
;; Output:
;; – *fail* if G is cyclic
;; – else: $\langle Q_1, ..., Q_n \rangle$: sequence of sets of independent actions such that:
;; $\quad Q_1 \cup ... \cup Q_n = \{a_i \mid n(a_i) \in N\}$ with $Q_1 \cap ... \cap Q_n = \emptyset$
  Let *without−pred* := $\emptyset$ **and** *Res* := $\langle\rangle$
  While $N \neq \emptyset$ **do**
    without−pred := $\{n(a) \in N \mid \text{Pred}(a) = \emptyset\}$
    If *without−pred* = $\emptyset$ **then return** *fail* **EndIf**
    Res := *Res* $\oplus \langle \{a \mid n(a) \in$ *without−pred*$\} \rangle$
    $N := N - \{$*without−pred*$\}$
    $C := C - \{(n_1, n_2) \in C \mid n_1 \in$ *without−pred*$\}$
  EndWhile
  Return *Res*
End {Stratify}

Theorem 6:

Let $Q \in 2^A$ be a set of actions. Then:

$$Q \text{ authorized} \Leftrightarrow \textbf{SearchSeq}(Q) \neq \textit{fail}$$

3.3 Returning the plan

The plans that LCGP returns (recognized by $\mathfrak{R}^*$) are not sufficiently ordered to be directly executed. We need to transform them into plans that are recognized by $\mathfrak{R}$. The transformation we use works in two stages:

1. The plan of LCGP is first transformed into a plan recognized by $\mathfrak{R}$ (cf. Theorem 4). In order to do this, the algorithm of § 3.2 (search of a cycle in the authorization graph) is used to return an authorized sequence of sets of actions for each set of actions of the plan.
2. The resulting plan can then be reordered optimally using the polynomial algorithm of [20], revised and formalized by [1] who demonstrates that it finds the optimal reordering in number of levels of the plan (i.e. in number of sets of independent actions).

Details of these two stages are given in the next two sections.

3.3.1 Transformation into Graphplan's semantics

We know that we can use **SearchSeq** to answer the question about the authorization of a set of actions; but we must now prove that the authorized sequences it returns can be used to transform the solution returned by LCGP (recognized by $\mathfrak{R}^*$), into a solution that has the semantics of the plans of Graphplan (recognized by $\mathfrak{R}$).

Theorem 7:

Let $E \in 2^P$ be a state and $S \in (2^A)^* - \{\langle\rangle\}$ a sequence of sets of actions, with $S = \langle Q_1, ..., Q_n \rangle$. Then:

$$E\ \mathfrak{R}^*\ S \neq \bot \Rightarrow E\ \mathfrak{R}^*\ S = E\ \mathfrak{R}\ (\textbf{SearchSeq}(Q_1) \oplus ... \oplus \textbf{SearchSeq}(Q_n))$$

We note that as each set of actions in the plan must be proved to be authorized during search by using the procedure **SearchSeq**, we can "memoize" for every set of actions at each level the result of that test; so when a plan is found, we can avoid the transformation described above and directly compute the optimal reordering of the plan as shown in the next section.

3.3.2 Search of the optimal reordering

As shown previously, the plan returned by using **SearchSeq** is recognized by $\mathfrak{R}$ (which recognizes plans of Graphplan) and solves the problem. We now use the PRF algorithm (cf. [20], revised and formalized by [1, p. 119]) in order to find a reordering that is optimal in the number of levels of the plan (i.e. in the number of independent sets of actions).

This stage will be decomposed in two parts, as we have done for the search of an authorized sequence of a set of actions. Firstly we build a graph that represents the constraints of the plan (i.e. order relations and independence relations among actions). We then use a modified topological sort algorithm on this graph to find the sequence of sets of actions corresponding to the solution-plan.

Definition 17 (partial order graph):

Let $E \in 2^P$ be a state and $S \in (2^A)^*$ a sequence of sets of actions, with $S = \langle Q_1, ..., Q_n \rangle$, such that $E\ \mathfrak{R}\ S \neq \bot$. The *partial order graph* POG(N, C) of S is an oriented graph defined by:

- N is the set of the nodes such that for each action $a \in Q_i$, $\forall\ i \in [1, n]$, there is only one associated node of N noted n(a),
- C is the set of arcs that represent the constraints among actions: there is an arc from n(a_i) to n(a_j) iff the execution of a_i **must precede** the execution of a_j, i.e.:

  $(n(a_i), n(a_j)) \in C \Leftrightarrow (a_i \in Q_k$ and $a_j \in Q_p$ and $1 \leq k < p \leq n$ and (not($a_j \angle a_i$) or $Add(a_i) \cap Prec(a_j) \neq \emptyset$))

The following algorithm simply computes the partial order graph corresponding to a plan, and then uses the **Stratify** algorithm to return the optimal reordering of its input. All proofs can be found in [1].

SearchReordering(S)
;; Input:
;; – S: a sequence of authorized sets of actions
;; Output:
;; – the optimal reordering of S
Begin
  Let POG(N, C) := the partial order graph of S
  Return Stratify(POG)
End {SearchReordering}

Another solution to compute the final plan is to directly transform the plan returned by LCGP (sequence of authorized sets of actions) into a Graphplan-like plan, by adding the following condition to the construction of the partial order graph: we add an arc between an action *a* and an action *b* that belong to the same authorized set if the action *b* does not authorize the action *a*. It is the same condition as for the construction of the authorization graph of § 3.2. This version is presented in [5].

4 Experiments

4.1 Equipment

We have implemented our own version of Graphplan, called GP; and LCGP that corresponds to the modifications of GP described in § 3. The two planners share most of their code and the differences between them are minimal (cf. § 3). The common part includes well-known improvements of Graphplan: EBL/DDB techniques from [10; 11] and a graph construction inspired by [16; 22] (two level circular structure of the planning-graph). GP and LCGP are implemented in Allegro Common Lisp 5.0. All the tests have been performed with a Pentium−II 450Mhz machine with 256Mb of RAM, running Debian GNU/Linux 2.0.

4.2 Comparison between Graphplan-based planners in Logistics domain

Here are the results of the tests we performed on the Logistics domain of the BLACKBOX[2] distribution [13] between LCGP and three planners based on Graphplan: IPP[3] v4.0 [19], STAN[4] v3.0 [7] and GP. IPP and STAN are highly optimized planners implemented in C for IPP, and in C++ for STAN. The heuristic used in the extraction phase of GP and LCGP is the original Graphplan "no-ops first" heuristic: the first action chosen for establishing a proposition is a no-op, and the other actions are left unordered. We present in the next section a more powerful heuristic, which will be used for the other tests.

For the first series of tests we used the 30 problems of the BLACKBOX distribution [13]. The results are shown in Table 1.

- Among the three planners based on Graphplan (which use the independence relation), STAN is the most efficient. Two reasons can explain this result: STAN has the EBL/DDB capacities described in [10; 11], and it preserves only the actions that are relevant for each problem thanks to its pre-planning type analysis tools [7]. Then comes GP, which solves fewer problems than STAN but significantly more than IPP. GP is faster than IPP except on 2 problems. This can be explained by the EBL/DDB capabilities of GP.
- Our planner, LCGP, solves all the problems with extremely good performances compared to the other planners. STAN is however faster than LCGP in 9 problems, but performances of LCGP would likely be better if it had the same features as STAN (C++ implementation and pre-planning analysis tools). In most of the problems, the planning-graph construction takes almost all the time: the search time is then negligible. Only a few problems (*log.c*, *log017*, *log020*, *log023*) take relatively more time due to the hardness of search in the second stage. The improvement is evident: LCGP runs on average 1800 times faster than GP on the problems solved by both planners.

One of the peculiarities of the Logistics domain is that plans can contain a lot of parallel actions. So GP, IPP and STAN find many independent actions, and there are fewer constraints (in relation to the number of actions) than in other domains, like Blocks-world domain with one arm. However, numerous constraints found by GP can be relaxed by LCGP to become authorization constraints. For example, in GP, the two actions "load a package in an airplane at place A" and "fly this airplane from place A to place B" are not independent: one precondition of the first action (the airplane must be at place A) is deleted by the second action. In LCGP, the first action authorizes the second so they can appear simultaneously in an authorized set. So, these results are mainly due to the reduction of the search space in LCGP (cf. § 1, the number of levels needed to solve the problem).

None of these planners produces optimal solutions (in number of actions), but their plans contain approximately the same number of actions. LCGP is not optimal in the sense of Graphplan (in number of levels in relation to the independence relation), it is optimal in number of levels in relation to the authorization relation. However, the plans found by LCGP are not significantly longer: on average for the problems solved by both planners, plans found by GP contain 48.67 actions, while plans found by LCGP

[2] BLACKBOX is downloadable at `http://www.research.att.com/~kautz/blackbox/index.html`
[3] IPP is downloadable at `http://www.informatik.uni-freiburg.de/~koehler/ipp.html`
[4] STAN is downloadable `at http://www.dur.ac.uk/~dcs0www/research/stanstuff/stanpage.html`

contain 49.11 actions. Moreover, LCGP finds the optimal solution on some problems (cf. *log010*, *log013*, *log025...*).

	CPU time (sec.)				Ratio TimeGP/ TimeLCGP	Actions				Levels		
										GP	LCGP	
Problems	**IPP**	**STAN**	**GP**	**LCGP**		**IPP**	**STAN**	**GP**	**LCGP**		**(+)**	**(++)**
log.easy	0.06	0.05	0.41	0.32	1.29	25	25	25	25	9	9	6
rocket.a	23.09	–	16.29	0.49	33.05	30		30	28	7	7	4
rocket.b	34.40	24.34	5.85	0.40	14.62	26	26	26	26	7	7	4
log.a	2,174.07	4.34	164.01	1.23	133.56	54	54	54	54	11	11	7
log.b	5,820.92	5.67	76,402.09	1.78	43,043.43	45	44	45	45	13	13	8
log.c	–	6,135.85	≥86,400	227.69	≥379		52		53	≥11	13	8
log.d	–	22,105.24	–	3.89			68		73		15	9
log.d3	–	≥86,400	≥86,400	4.46	≥19,355				72	≥13	13	8
log.d1	–	–	≥86,400	23.88	≥3,619				68	≥14	17	10
log010	1,861.77	0.59	28.53	3.28	8.70	43	43	42	41	10	11	7
log011	–	218.03	8,635.85	2.18	3,965.04		48	48	49	11	11	7
log012	74.40	0.77	6.61	1.17	5.64	38	38	38	38	8	8	5
log013	–	523.24	4,526.88	3.70	1,222.82		67	67	66	11	11	7
log014	122.65	1.17	6.04	5.00	1.21	70	71	70	75	10	11	7
log015	–	≥86,400	≥86,400	4.26	≥20,267				61	≥11	13	7
log016	–	–	–	4.13					40		16	9
log017	–	–	≥86,400	101.07	≥855				44	≥16	17	10
log018	–	3.04	40.74	5.58	7.31		48	52	50	11	11	7
log019	–	5.77	14.46	2.60	5.56		47	46	50	11	12	7
log020	–	–	≥86,400	578.01	≥149				87	≥14	15	9
log021	–	3,259.27	2,594.30	4.20	617.25		63	63	66	11	12	7
log022	–	–	≥86,400	4.18	≥20,690				74	≥14	15	9
log023	–	232.42	≥86,400	90.50	≥955		61		61	≥13	13	8
log024	–	286.80	3,349.58	3.96	846.71		64	64	67	12	13	8
log025	–	1.46	12.62	3.38	3.74		57	58	56	12	13	8
log026	–	0.64	4.85	3.10	1.56		51	50	50	12	12	8
log027	–	23.15	≥86,400	3.41	≥25,345		71		72	≥13	14	8
log028	–	–	≥86,400	11.61	≥7,445				78	≥14	14	9
log029	3,558.05	1.19	8.27	5.96	1.39	46	49	46	46	10	11	7
log030	–	1.11	49.54	3.06	16.21		52	52	52	13	13	8
Mean (*)	1,518.82	1,563.53	5,325.94	2.85	1,866.28	41.89	52.33	48.67	49.11	10.50	10.89	6.78
Mean (**)	-	-	≥34,280.96	36.95	≥927.82	-	-	-	55.57	≥11.50	12.37	7.53

(+) number of levels of the plan after transformation by SearchReordering (§ 3.3)
(++) number of levels of the plan before transformation by SearchReordering (§ 3.3)
(*) mean of solved problems (white cells). For LCGP : mean of problems solved by Graphplan.
(**) mean of the 30 problems.
grey cell: failure in the resolution of the corresponding problem.
A dash (–) indicates that the corresponding problem could not be solved due to a lack of memory.

Table 1: Comparison between Graphplan-based planners in the Logistics domain

4.3 Heuristics in the search phase of GP and LCGP

In this section, we present the heuristic used in GP and LCGP during the extraction stage in the next series of tests. This heuristic is domain independent and greatly improves the extraction of plans. It combines the "no-ops first" initial heuristic of Graphplan and the "level-based" heuristic proposed for LCGP in [5] and for Graphplan in [12]. By merging these two heuristics, we take advantage of the qualities of both: quality of the solution in number of actions (no-ops first heuristic) and speedup of the search time (level-based heuristic).

As demonstrated in [10; 11], the extraction stage of Graphplan can be seen as a Dynamic Constraint Satisfaction Problem (DCSP) [18]. Indeed, during this process, every proposition p_n at a level n of the planning-graph can be assigned to a variable of a CSP; the set of actions that support every p_n constitutes its domain and the mutual exclusions produced during the construction stage become constraints in the CSP. Graphplan tries to extract a plan from the planning-graph by assigning a value (an action) to every variable (proposition) in order to satisfy the set of constraints (mutual exclusions). The assignment of values to variables is a dynamic process because every assignment at a level n activates other variables in the previous level. During this extraction stage, two orders are involved: the order in which the propositions are considered for assignment (variable ordering heuristic) and the order in which the actions supporting a proposition are employed (value ordering heuristic). To use the classical CSP heuristic "most constrained variable first and least constrained value first", we need a quantitative measure for these constraints.

The "no-ops first" Graphplan heuristic prefers using a no-op to support a proposition, and then the other possible actions. This choice seems reasonable to produce plans that contain fewer actions. However, this heuristic gives no information about the variable or value constraints and is not appropriate from a CSP viewpoint (most constrained variable first and least constrained value first). Our experimental results [5] and those of [12] clearly demonstrate that in numerous domains this strategy leads to a reduction in the search time.

The size of the variable domain is another heuristic employed in [11] to measure how constrained a variable is. Using this criterion, a proposition is said to be more constrained than another if fewer actions support it and the experimental study of [11] shows that, using this heuristic, Graphplan runs as much as 4 times faster.

Neither of these two heuristics is informative enough because they do not really measure the difficulty of the extraction of a solution. Indeed, it is not because several actions support a proposition that this last is easier to obtain. This information only concerns the current level; in order to improve the heuristic, we need a measure for the difficulty to assert a proposition that takes into account the different levels of the planning-graph (from level 0 to the current level).

The level-based heuristic proposed in [5] and [12] uses as a measure the starting level of a proposition (or action), i.e. the number of the level of the planning-graph in which this proposition (or action) appears for the first time. We can reasonably suppose that the higher the starting level of a proposition is, the more difficult it is to obtain it, indeed:

- To be asserted, a proposition with a high starting level needs a plan with a large number of steps. Generally, a plan that needs n steps to assert a proposition p contains more actions (including no-ops) than a plan to assert another proposition p' that appears in a level n', $n' < n$. So, a high starting level generally denotes a proposition that is more difficult to assert than another with a lower starting level.
- The mutual exclusions between propositions (or actions) tend to disappear as the associated propositions (or actions) remain in the higher levels of the planning-graph. So, the propositions with a high starting level are established using recent actions and they have a better chance of being involved in mutual exclusions than older ones.

So, for a proposition, a high starting level denotes two difficulties: the plan to obtain it generally contains a large number of actions (including no-ops), and the actions that support this proposition have a higher probability of being involved in mutual exclusions.

The starting level seems to be a characteristic measure for the degree of constraint of a proposition. So, for the dynamic variable ordering, we will assign first the propositions having the higher starting level. Concerning the static domain ordering, a similar reasoning demonstrates that actions that have lower starting levels are on average easier to produce.

Our experiments prove (cf. Table 2) that it is really interesting to merge this level-based heuristic with the no-ops first heuristic to take advantage of the qualities of both: speedup of the search time (level-based heuristic) and quality of the solution in number of actions (no-ops first heuristic). Consequently, the GP and LCGP heuristic have been implemented using "greater starting level variable first" as variable ordering heuristic and "no-ops first, then least starting level value first" as value ordering heuristic. The bad results in the Tower of Hanoi domain can be explained by the fact that with the standard encoding, the starting level of the propositions of the goal (except one of them, for the biggest disc) is 0: the propositions "disc 1 is on disc 2", "disc 2 is on disc 3", ..., belong both to the initial state and to the goal. A solution would be to encode the domain in a more informative way, by indicating which peg the discs are located on: "disc 1 is on disc 2 on peg A" would belong to the initial state, and "disc 1 is on disc 2 on peg C" would belong to the goal.

Problems	CPU time (sec.)			Ratio No−opsF/		Actions			Levels	
	No−opsF	Level	LevelN	Level	LevelN	No−opsF	Level	LevelN	(+)	(++)
ferry6	3.05	0.30	0.36	10.17	8.47	23	23	23	23	12
ferry8	387.51	2.51	3.06	154.39	126.64	31	31	31	31	16
gripper6	1.45	0.39	0.49	3.74	2.96	17	17	17	11	6
gripper8	165.81	8.02	7.61	20.68	21.79	23	23	23	15	8
bw-large-a	3.42	2.49	2.57	1.37	1.33	12	12	12	12	12
bw-large-b	257.65	19.13	36.05	13.47	7.15	18	18	18	18	18
log020	578.01	8.38	9.20	68.97	62.83	87	93	87	15	9
log023	90.50	3.77	4.30	24.01	21.05	61	65	61	13	8
hanoi5	8.41	10.48	27.30	0.80	0.31	32	32	32	32	21

(+) number of levels of the plan after transformation by SearchReordering (cf. § 3.3)
(++) number of levels of the plan before transformation by SearchReordering (cf. § 3.3)
No−opsF: no-ops first heuristic
Level: level-based heuristic
LevelN: level-based heuristic with no-ops first.

Table 2: Benefits of the heuristic for LCGP

4.4 Comparison GP vs. LCGP in Ferry and Gripper domain

Compared with GP, in Ferry and Gripper domains, LCGP is not as efficient as in the Logistics domain, but always runs faster than GP (see Table 3 and Table 4). The column "Expanded nodes" in the tables represents the number of sets of subgoals that the algorithm tries to assert. It is interesting to see that in the Ferry domain, whose problems have linear solutions, planning-graphs produced by LCGP are almost two times shorter than those of GP. Indeed, in LCGP, the actions "embark a car on side A" and "sail from side A to side B" can belong to the same authorized set; the same holds for "debark a car at side B" and "sail from side B to side A". The same phenomenon occurs in the Gripper domain: the planning-graphs produced by LCGP are also almost two times shorter than those of GP. This domain allows some parallelism between actions: with GP, the robot can take a ball in each of its two grippers at the same time. With LCGP, the action of moving

Subgoals	CPU time (sec.)		Ratio TimeGP/ TimeLCGP	Expanded nodes		Actions		Levels		
								GP	LCGP	
	GP	LCGP		GP	LCGP	GP	LCGP		(+)	(++)
1	0.02	0.01	1.54	4	3	3	3	3	3	2
2	0.03	0.02	1.30	15	5	7	7	7	7	4
3	0.06	0.04	1.58	139	21	11	11	11	11	6
4	0.17	0.06	2.67	646	92	15	15	15	15	8
5	0.56	0.14	4.00	2,310	351	19	19	19	19	10
6	1.96	0.36	5.47	6,998	997	23	23	23	23	12
7	6.29	1.06	5.94	19,125	2,614	27	27	27	27	14
8	18.61	3.06	6.07	48,846	6,657	31	31	31	31	16
9	51.62	7.82	6.60	118,195	14,786	35	35	35	35	18
10	143.54	22.07	6.50	275,921	37,686	39	39	39	39	20
11	385.27	58.16	6.62	626,157	84,930	43	43	43	43	22
12	1,021.97	159.85	6.39	1,392,793	190,266	47	47	47	47	24

(+) number of levels of the plan after transformation by SearchReordering (cf. § 3.3)
(++) number of levels of the plan before transformation by SearchReordering (cf. § 3.3)

Table 3: Comparison in the Ferry domain

Subgoals	CPU time (sec.)		Ratio TimeGP/ TimeLCGP	Expanded nodes		Actions		Levels		
								GP	LCGP	
	GP	LCGP		GP	LCGP	GP	LCGP		(+)	(++)
2	0.04	0.03	1.30	4	3	5	5	3	3	2
4	0.15	0.07	2.17	435	48	11	11	7	7	4
6	3.27	0.49	6.65	9,533	1,272	17	17	11	11	6
8	57.89	7.61	7.61	127,804	15,332	23	23	15	15	8
10	765.05	89.19	8.58	1,233,178	128,664	29	29	19	19	10
12	9,455.39	927.60	10.19	9,176,365	861,096	35	35	23	23	12

(+) number of levels of the plan after transformation by SearchReordering (cf. § 3.3)
(++) number of levels of the plan before transformation by SearchReordering (cf. § 3.3)

Table 4: Comparison in the Gripper domain

from on room to another can be considered simultaneously with taking two balls. But contrarily to the Ferry domain, in which the speedup between the two planners seems to stabilize around 6.5, in the Gripper domain, the speedup grows in relation to the difficulty of the problem.

4.5 Comparison GP vs. LCGP in Blocks-world domain

4.5.1 Prodigy version

In the Prodigy version of this domain, with 6 operators and one arm, there is no parallelism at all to exploit, even for LCGP, and the planning-graphs built by GP and LCGP are exactly the same. So, the search stage is performed in exactly the same way. We could however expect LCGP to be slower than GP, because of the need to recognize the authorized sets (cf. § 3.2). But as there is no parallelism, a set of actions considered during the search contains only one "real" action (all the others are no-ops); and the authorization test must only be performed on the subset of the real actions of a set of actions. Indeed:

- one no-op always authorizes another no-op;
- if an action does not authorize a no-op, then the no-op does not authorize the action, so they are mutually exclusive (and vice versa).

Thus, a set of actions containing at most two real actions is authorized if and only if there is no pair of mutually exclusive actions. This explains why LCGP and GP have very close performances in this domain (see Table 5).

Problems	CPU time (sec.)		Ratio TimeGP/ TimeLCGP	Expanded nodes		Actions		Levels		
								GP	LCGP	
	GP	LCGP		GP	LCGP	GP	LCGP		(+)	(++)
bw−simple	0.015	0.015	1.00	3	3	2	2	2	2	2
bw−sussman	0.059	0.059	1.00	7	7	6	6	6	6	6
bw−reversal4	0.082	0.083	0.99	9	9	8	8	8	8	8
bw−large−a	2.02	2.07	0.98	218	218	12	12	12	12	12
bw−large−b	33.31	34.16	0.98	15,563	15,563	18	18	18	18	18

(+) number of levels of the plan after transformation by SearchReordering (cf. § 3.3)
(++) number of levels of the plan before transformation by SearchReordering (cf. § 3.3)

Table 5: Comparison in the Blocks-world domain (Prodigy version)

4.5.2 Three operators version

We compared GP and LCGP in the version of the Blocks-world with three operators: MoveToTable, MoveFromTable, Move (from one stack to another). The main difference with the Prodigy version of this domain is the possibility to apply actions in parallel. LCGP takes advantage of parallel actions in this domain, and some mutually exclusive actions for GP become authorized for LCGP. For example, if the two blocks A and B are on top of two different stacks, the actions "move block A to table" and "move block B on block A" are mutually exclusive because the second action deletes the fact that block A is clear, which is a precondition of the first action; but the first action authorizes the second one.

In term of CPU time performances (cf. Table 6), LCGP is always faster than GP (on average, 426 times faster). Two problems (*Nineteen* and *P16*) are not solved by GP because it runs out of memory. Indeed, LCGP needs on average 5.38 levels to solve the problems while GP needs 7.13 levels. In term of quality of the solution in number of actions, LCGP generates slightly better solutions than GP (13.13 actions for LCGP against 13.75 for GP), although the size of the plans in number of levels is greater for LCGP (8.63 levels for LCGP against 7.13 for GP). Once again, the loss of optimality in the number of levels seems to have no influence on the quality of the solution in terms of the number of actions.

One remarkable thing that happens in this domain with the problems we tested with LCGP is that the number of developed nodes corresponds exactly to the number of levels of the planning-graph. In other words, once LCGP has found an authorized set of actions that asserts a set of subgoals by checking the mutual exclusions and the authorization of that set, it never backtracks over this choice, even in the hardest problems. This phenomenon does not occur with GP: there is at least one backtrack (problems *Bw−12steps* and *Nine*), and many more in harder problems.

Problems	CPU time (sec.)		Ratio TimeGP/ TimeLCGP	Expanded nodes		Actions		Levels		
								GP	LCGP	
	GP	LCGP		GP	LCGP	GP	LCGP		(+)	(++)
Bw−12step	0.35	0.33	1.06	6	5	6	6	4	6	4
Nine	2.43	0.76	3.20	6	4	10	9	4	5	3
Eleven	12.67	6.04	2.10	8	5	13	14	5	7	4
Fifteen	17,920.31	55.90	320.59	81,402	7	22	18	8	11	6
Nineteen	–	162.80	–	–	7	–	26	–	10	6
P8	3.69	1.51	2.44	15	9	13	15	10	13	8
P10	11.45	4.73	2.42	18	6	14	12	7	7	5
P12	36,180.19	11.82	3,060.41	1,821,844	6	16	15	9	10	5
P14	73.29	46.11	1.59	13	9	16	16	10	10	8
P16	–	216.27	–	–	9	–	26	–	13	8

(+) number of levels of the plan after transformation by SearchReordering (cf. § 3.3)
(++) number of levels of the plan before transformation by SearchReordering (cf. § 3.3)
A dash (−) indicates that the corresponding problem could not be solved due to a lack of memory.

Table 6: Comparison in the Blocks−world domain (3 ops. Version)

4.6 Comparison GP vs. LCGP in Mprime and Mystery domains

We finished the experiments with the Mprime and Mystery domains. These domains are very similar and allow parallel actions. We used the series of problems created for the AIPS'98 planning competition. Contrary to the problems of the other domains we tried, some problems in these series have no solution (cf. Tables 7 and 8). The main difficulty in these domains is to construct the planning-graph; the extraction of the solution is then trivial. The problems not present in the table (*Mprime−X−6*, *Mprime−X−10*, ...) are not solved by the two planners due to a lack of memory during the construction of the planning-graph. These domains are the only ones we found where GP is slightly better than LCGP: in Mprime domain, GP is 1.01 times faster than LCGP, and in Mystery domain, GP is 1.05 times better than LCGP. The reason is that, although they sometimes have fewer levels, the planning-graphs built by LCGP contain in their last levels more actions and propositions than GP's ones. For example, for the problem *Mprime−Y−2* solved in 6.99 seconds by GP and in

Problems	CPU time (sec.)		Ratio TimeGP/ TimeLCGP	Expanded nodes		Actions		Levels		
								GP	LCGP	
	GP	LCGP		GP	LCGP	GP	LCGP		(+)	(++)
Mprime−X−1	1.95	1.17	1.67	6	5	6	8	5	5	4
Mprime−X−2	24.38	29.90	0.82	8	5	9	10	5	5	4
Mprime−X−3	2.48	1.98	1.25	5	4	4	4	4	4	3
Mprime−X−4	1.30	1.15	1.13	8	8	9	10	7	7	6
Mprime−X−5	48.74	53.21	0.92	0	0	–	–	10	–	9
Mprime−X−7	3.41	2.91	1.17	0	0	–	–	10	–	8
Mprime−X−8	9.22	12.02	0.77	6	6	10	10	5	5	5
Mprime−X−9	5.25	5.25	1.00	6	5	8	8	5	5	4
Mprime−X−11	3.51	2.26	1.55	12	6	8	8	7	7	5
Mprime−X−12	5.74	6.22	0.92	6	10	6	9	5	6	5
Mprime−X−16	14.88	15.04	0.99	37	36	10	6	5	5	4
Mprime−X−17	32.57	25.95	1.26	5	4	5	5	4	4	3
Mprime−X−19	101.33	121.91	0.83	7	6	8	8	6	7	5
Mprime−X−21	135.71	131.03	1.04	0	0		–	14	–	12
Mprime−X−25	0.77	0.63	1.24	5	4	4	4	4	4	3
Mprime−X−26	9.01	6.61	1.36	6	5	6	6	5	5	4
Mprime−X−27	13.01	5.68	2.29	7	4	7	7	4	4	3
Mprime−X−28	4.24	2.01	2.11	20	6	9	7	7	7	5
Mprime−X−29	7.02	3.83	1.83	5	4	5	6	4	5	3
Mprime−Y−1	5.23	4.85	1.08	5	4	4	4	4	4	3
Mprime−Y−2	6.99	7.41	0.94	8	7	7	8	7	7	6
Mprime−Y−4	6.66	6.46	1.03	5	4	5	4	4	4	3
Mprime−Y−5	1.12	1.17	0.96	5	5	7	6	4	5	4
Mean	19.33	19.51	0.99	7.48	6.00	6.85	6.90	5.87	5.25	4.83

(+) number of levels of the plan after transformation by SearchReordering (cf. § 3.3)
(++) number of levels of the plan before transformation by SearchReordering (cf. § 3.3)
A dash (−) indicates that the corresponding problem has no solution.

Table 7: Comparison in the Mprime domain

7.41 seconds by LCGP, the last level (7) for GP contains 835 actions against 861 for the last level (6) of LCGP. LCGP also computes more mutual exclusions between actions than GP: 188,820 against 187,386. At a same or inferior level, the planning-graphs developed by LCGP contain more potential plans than the ones built by GP, but they are harder to compute. And although the extraction of the solution is trivial for both planners, LCGP does not take advantage of that.

However we can see that the best performances of GP in these domains have no common measure with the best performances of LCGP in other domains: the maximum speedup in these domains is 1.30 (problems *Mprime−X−8* and *Mysty−X−2*), which is almost nothing compared with a speedup like 39,524 in the logistics domain (problem *log016*). Concerning the quality of the solution in number of actions, GP is better than LCGP in the Mprime domain (on average 0.7% more actions for LCGP) while LCGP is better than GP in the Mystery domain (on average 3.9% more actions for GP).

Problems	CPU time (sec.)		Ratio TimeGP/ TimeLCGP	Expanded nodes		Actions		Levels		
								GP	LCGP	
	GP	LCGP		GP	LCGP	GP	LCGP		(+)	(++)
Mysty−X−1	0.81	0.54	1.48	6	5	5	6	5	6	4
Mysty−X−2	22.31	29.14	0.77	6	8	9	10	5	5	4
Mysty−X−3	2.19	1.75	1.25	5	4	4	4	4	4	3
Mysty−X−4	2.70	2.57	1.05	0	0	–	–	14	–	13
Mysty−X−5	43.91	51.09	0.86	0	0	–	–	10	–	9
Mysty−X−7	3.19	2.80	1.14	0	0	–	–	10	–	8
Mysty−X−8	198.11	193.08	1.03	0	0	–	–	21	–	21
Mysty−X−9	5.33	5.03	1.06	6	5	8	8	5	5	4
Mysty−X−11	1.85	1.15	1.61	12	6	7	7	7	7	5
Mysty−X−12	2.23	2.70	0.82	0	0	–	–	8	–	8
Mysty−X−15	88.37	94.56	0.93	7	6	12	7	6	6	5
Mysty−X−16	11.93	9.78	1.22	0	0	–	–	9	–	7
Mysty−X−17	19.48	16.72	1.16	5	4	4	4	4	4	3
Mysty−X−18	38.58	41.04	0.94	0	0	–	–	18	–	18
Mysty−X−19	23.94	26.65	0.90	7	6	6	7	6	7	5
Mysty−X−20	68.51	63.16	1.08	8	7	13	13	7	7	6
Mysty−X−21	132.89	131.69	1.01	0	0	–	–	14	–	12
Mysty−X−23	129.35	141.87	0.91	0	0	–	–	11	–	10
Mysty−X−24	283.71	318.21	0.89	0	0	–	–	25	–	25
Mysty−X−25	0.81	0.81	1.00	5	4	4	4	4	4	3
Mysty−X−26	4.98	4.39	1.14	7	6	8	6	6	6	5
Mysty−X−27	1.95	1.57	1.24	7	4	7	7	4	4	3
Mysty−X−28	0.97	0.72	1.35	22	6	7	7	7	7	5
Mysty−X−29	1.71	1.62	1.05	5	4	4	4	4	4	3
Mysty−X−30	18.92	18.33	1.03	7	6	10	10	6	6	5
Mean	44.35	46.44	0.95	4.60	3.24	7.20	6.93	8.80	5.47	7.76

(+) number of levels of the plan after transformation by SearchReordering (cf. § 3.3)
(++) number of levels of the plan before transformation by SearchReordering (cf. § 3.3)
A dash (–) indicates that the corresponding problem has no solution.

Table 8: Comparison in the Mystery domain

5 Related work

The idea of allowing more parallelism during the search and adding constraints after a plan has been found was introduced in [6] and used for experiments in [21]. [6] introduces the property of post-serializability of a set of actions: a set of actions A is said to be post-serializable if (a) the union of their preconditions is consistent; (b) the union of their effects is consistent; and (c) the graph A_G is acyclic, where A_G is the graph that contains one node for each action in A, and an edge from an action a to an action b if the preconditions of a are inconsistent with the effects of b. However, our work differs on several points. In [6], the authors used the post-serializability in a different framework: they encode planning problems in nonmonotonic logic programs and search for stable models. We have gone one step further in the relaxation of constraints, because the authorization relation allows inconsistent effects: an action a can delete add effects of an action b if a precedes b in an authorized linearization. In [6] the authors do not perform a test of post-serializability like our test of authorization during search: they need to know before encoding the problem if the domain will support post-serializable actions (or weak post-serializable actions, which consists in breaking cycles by replacing actions in a post-processing phase). We generalized this idea by the fact that we do not need to know anything about particular properties of the domains; our process is guaranteed to be sound and complete

for any STRIPS encoding of a planning problem. Finally, we also introduced the use of a modified version of the PRF algorithm [1] in order to find the reordering optimal in the number of levels of the solution plan.

6 Conclusion

Previous improvements to Graphplan have concentrated on improving the expressiveness of the description language (conditional effects, quantification...), taking into account uncertainty, and improving the implementation of the planning-graph (by using a two level circular structure). In this paper, we question a basic foundation of the research made around planners that produce parallel plans: the independence relation. In Graphplan, the structure of the graph is based on that concept of independence between actions, and allows the generation of plans with parallel actions. We have demonstrated that this condition can advantageously be replaced by a less restrictive one: the authorization between actions. The search space that is then developed by our planner LCGP becomes more compact (fewer levels than Graphplan), which dramatically speeds up the search time in some domains. The loss of optimality in the sense of Graphplan (in number of levels) does not appear to be significant, compared to the gain in efficiency. Furthermore, the optimality in number of actions is not directly related to the optimality in number of levels (when parallelism is possible), so LCGP can give better solutions (in number of actions) than Graphplan.

Acknowledgements

The authors would like to thank the anonymous reviewers, and Jérôme Mengin for his helpful comments and contribution to the writing of the paper.

Appendix A. Proofs

Property 1:

The concatenation of sequences of sets of actions has the following properties:

1. identity element: $\forall\ S \in (2^A)^*,\ S \oplus \langle\rangle = \langle\rangle \oplus S = S$
2. associativity: $\forall\ S_1, S_2, S_3 \in (2^A)^*,\ (S_1 \oplus S_2) \oplus S_3 = S_1 \oplus (S_2 \oplus S_3)$ (it can be noted $S_1 \oplus S_2 \oplus S_3$)
3. stability for A^*: $\forall\ S_1, S_2 \in A^*,\ S_1 \oplus S_2 \in A^*$ and $S_2 \oplus S_1 \in A^*$

Proof: trivial.

Notation: when there is no ambiguity, $((...((E\ \Re\ S_1)\ \Re\ S_2)\ \Re\ ...\ )\Re\ S_n)$ will be noted $E\ \Re\ S_1\ \Re\ S_2\ \Re\ ...\ \Re\ S_n$.

The successive application of $\Re$ to two sequences of sets of actions and to a state gives the same result as the application of the concatenation of these two sequences to the same state.

Property 2:

Let $E \in (2^P \cup \{\bot\})$ be a state and $S_1, S_2 \in (2^A)^*$ two sequences of sets of actions. Then:

$$E\ \Re\ (S_1 \oplus S_2) = E\ \Re\ S_1\ \Re\ S_2.$$

Proof:

Let $E \in (2^P \cup \{\bot\})$ be a state and $S_1, S_2 \in (2^A)^*$ two sequences of sets of actions.

If $E = \bot$:

$E\ \Re\ (S_1 \oplus S_2) = \bot\ \Re\ (S_1 \oplus S_2) = \bot$

$E\ \Re\ S_1\ \Re\ S_2 = (\bot\ \Re\ S_1)\ \Re\ S_2 = \bot\ \Re\ S_2 = \bot$

If $E \neq \bot$, we can make a proof by induction on length(S_1).

- When length(S_1) = 0:
  $E\ \Re\ (S_1 \oplus S_2) = E\ \Re\ (\langle\rangle \oplus S_2) = E\ \Re\ S_2$
  $E\ \Re\ S_1\ \Re\ S_2 = (E\ \Re\ \langle\rangle)\ \Re\ S_2 = E\ \Re\ S_2$
- We assume the following property is true when length(S_1) = n:
  $$E\ \Re\ (S_1 \oplus S_2) = E\ \Re\ S_1\ \Re\ S_2$$
  We demonstrate it when length(S_1) = n+1:
  $E\ \Re\ (S_1 \oplus S_2)$
  $= E\ \Re\ (\langle \text{first}(S_1)\rangle \oplus \text{rest}(S_1) \oplus S_2)$ because length(S_1) > 0
  $= E\ \Re\ (\langle Q_1\rangle \oplus S'_1 \oplus S_2)$ with Q_1 = first(S_1) and S'_1 = rest(S_1)
  Two cases can occur:
  – If Q_1 is not independent or Prec(Q_1) $\not\subseteq$ E:
    $E\ \Re\ (S_1 \oplus S_2) = E\ \Re\ (\langle Q_1\rangle \oplus S'_1 \oplus S_2) = \bot$
    $E\ \Re\ S_1\ \Re\ S_2 = E\ \Re\ (\langle Q_1\rangle \oplus S'_1)\ \Re\ S_2 = \bot\ \Re\ S_2 = \bot$

– If Q_1 is independent and $\text{Prec}(Q_1) \subseteq E$:
$E \mathbin{\Re} (S_1 \oplus S_2)$
$= E \mathbin{\Re} (\langle Q_1 \rangle \oplus S'_1 \oplus S_2)$
$= ((E - \text{Del}(Q_1)) \cup \text{Add}(Q_1)) \mathbin{\Re} (S'_1 \oplus S_2)$
$= E' \mathbin{\Re} (S'_1 \oplus S_2)$ with $E' = (E - \text{Del}(Q_1)) \cup \text{Add}(Q_1)$
$E \mathbin{\Re} S_1 \mathbin{\Re} S_2$
$= E \mathbin{\Re} (\langle Q_1 \rangle \oplus S'_1) \mathbin{\Re} S_2$
$= ((E - \text{Del}(Q_1)) \cup \text{Add}(Q_1)) \mathbin{\Re} S'_1 \mathbin{\Re} S_2$
$= E' \mathbin{\Re} S'_1 \mathbin{\Re} S_2$
$= E' \mathbin{\Re} (S'_1 \oplus S_2)$ (induction hyp.), because $\text{length}(S'_1) = \text{length}(\text{rest}(S_1)) = n$

Theorem 1:
Let $E \in (2^P \cup \{\perp\})$ be a state and $S \in (2^A)^* - \{\langle\rangle\}$ a sequence of sets of actions, with $S = \langle Q_1, ..., Q_n \rangle$. Then:

$$E \mathbin{\Re} S \neq \perp \Rightarrow \forall\, S_1 \in \text{Lin}(Q_1), ..., \forall\, S_n \in \text{Lin}(Q_n),\ E \mathbin{\Re} S = E \mathbin{\Re} (S_1 \oplus ... \oplus S_n).$$

Proof: It is based upon the following three lemmas.

Lemma 1:
Let $A, A_1, ..., A_n, B_1, ..., B_n$ be sets such that $\forall\, i \in [1, n-1]$, $A_{i+1} \cap (B_1 \cup ... \cup B_i) = \emptyset$. Then:

$$(A - (A_1 \cup ... \cup A_n)) \cup (B_1 \cup ... \cup B_n) = ((...((A - A_1) \cup B_1) - ...) - A_n) \cup B_n.$$

Proof:
Let $A, A_1, ..., A_n, B_1, ..., B_n$ be sets such that $\forall\, i \in [1, n-1]$, $A_{i+1} \cap (B_1 \cup ... \cup B_n) = \emptyset$.
We will use the following two properties:

$$A - (B \cup C) = (A - B) - C \qquad (\alpha)$$

and

$$B \cap C = \emptyset \Rightarrow (A - B) \cup C = (A \cup C) - B \qquad (\beta)$$

We can make a proof by induction on n:
- When $n = 1$: trivial
- We assume the following property is true at rank n; let us demonstrate it at rank $n+1$:
$$(A - (A_1 \cup ... \cup A_n)) \cup (B_1 \cup ... \cup B_n) = ((...((A - A_1) \cup B_1) - ...) - A_n) \cup B_n$$
with $\forall\, i \in [1, n-1]$, $A_{i+1} \cap (B_1 \cup ... \cup B_i) = \emptyset$.
Given A_{n+1} and B_{n+1} with $A_{n+1} \cap (B_1 \cup ... \cup B_n) = \emptyset$:
$(A - (A_1 \cup ... \cup A_n \cup A_{n+1})) \cup (B_1 \cup ... \cup B_n \cup B_{n+1})$
$= ((A - (A_1 \cup ... \cup A_n)) - A_{n+1}) \cup (B_1 \cup ... \cup B_n) \cup B_{n+1}$ from (α)
$= (((A - (A_1 \cup ... \cup A_n)) \cup (B_1 \cup ... \cup B_n)) - A_{n+1}) \cup B_{n+1}$ from (β), because $A_{n+1} \cap (B_1 \cup ... \cup B_n) = \emptyset$
$= ((((...((A - A_1) \cup B_1) - ...) - A_n) \cup B_n) - A_{n+1}) \cup B_{n+1}$ (induction hyp.)

The following lemma will be used to calculate the application of a sequence of actions to a state (different from $\perp$) when it contains all the preconditions of every action of the sequence and when an action never deletes the preconditions of another one which succeeds to it (immediately or not). In this particular case, the result is always different from $\perp$.

Lemma 2:
Let $E \in 2^P$ be a state and $S \in A^*$ a sequence of actions, with $S = \langle a_i \rangle_n$, such that: $\text{Prec}(S) \subseteq E$ and $\forall\, i \in [1, n-1]$, $\text{Prec}(a_{i+1}) \cap (\text{Del}(a_1) \cup ... \cup \text{Del}(a_i)) = \emptyset$. Then:

$$E \mathbin{\Re} S = ((...((((E - \text{Del}(a_1)) \cup \text{Add}(a_1)) - \text{Del}(a_2)) \cup \text{Add}(a_2)) - ...) - \text{Del}(a_n)) \cup \text{Add}(a_n).$$

Proof:
Let $E \in 2^P$ be a state and $S \in A^*$ a sequence of actions, with $S = \langle a_i \rangle_n$.
We can make a proof by induction on $\text{length}(S)$:
- When $\text{length}(S) = 0$: $E \mathbin{\Re} S = E \mathbin{\Re} \langle a_i \rangle_0 = E \mathbin{\Re} \langle\rangle = E$.
- We assume the following property is true when $\text{length}(S) = n$:
Given $S = \langle a_i \rangle_n$ with $\text{Prec}(S) \subseteq E$ and $\forall\, i \in [1, n-1]$, $\text{Prec}(a_{i+1}) \cap (\text{Del}(a_1) \cup ... \cup \text{Del}(a_i)) = \emptyset$,
$E \mathbin{\Re} \langle a_1, a_2, ..., a_n \rangle = ((...((((E - \text{Del}(a_1)) \cup \text{Add}(a_1)) - \text{Del}(a_2)) \cup \text{Add}(a_2)) - ...) - \text{Del}(a_n)) \cup \text{Add}(a_n)$.
Let us demonstrate it when $\text{length}(S) = n+1$, with $S = \langle a_i \rangle_{n+1}$, $\text{Prec}(S) \subseteq E$ and $\forall\, i \in [1, n]$, $\text{Prec}(a_{i+1}) \cap (\text{Del}(a_1) \cup ... \cup \text{Del}(a_i)) = \emptyset$:
$E \mathbin{\Re} \langle a_1, ..., a_{n+1} \rangle$
$= E \mathbin{\Re} (\langle a_1, ..., a_n \rangle \oplus \langle a_{n+1} \rangle)$
$= E \mathbin{\Re} \langle a_1, ..., a_n \rangle \mathbin{\Re} \langle a_{n+1} \rangle$ from Property 2

$= ((E \,\mathfrak{R}\, \langle a_1, ..., a_n\rangle) - \mathrm{Del}(a_{n+1})) \cup \mathrm{Add}(a_{n+1})$ because $E \,\mathfrak{R}\, \langle a_1, ..., a_n\rangle \neq \perp$ and $\mathrm{Prec}(a_{n+1}) \subseteq E \,\mathfrak{R}\, \langle a_1, ..., a_n\rangle$

$= ((((...((E - \mathrm{Del}(a_1)) \cup \mathrm{Add}(a_1)) - ...) - \mathrm{Del}(a_n)) \cup \mathrm{Add}(a_n)) - \mathrm{Del}(a_{n+1})) \cup \mathrm{Add}(a_{n+1})$ (induction hyp.)

Lemma 3:

Let $E \in (2^P \cup \{\perp\})$ be a state and $Q \in 2^A$ a set of actions. Then:

$$E \,\mathfrak{R}\, \langle Q\rangle \neq \perp \Rightarrow \forall\, S \in \mathrm{Lin}(Q),\ E \,\mathfrak{R}\, \langle Q\rangle = E \,\mathfrak{R}\, S.$$

Proof:

Let $E \in (2^P \cup \{\perp\})$ be a state and $Q \in 2^A$ a set of actions such that $E \,\mathfrak{R}\, \langle Q\rangle \neq \perp$ (which implies $E \neq \perp$).

As $E \,\mathfrak{R}\, \langle Q\rangle \neq \perp$, Q is an independent set of actions and $\mathrm{Prec}(Q) \subseteq E$ (else we would have $E \,\mathfrak{R}\, \langle Q\rangle = \perp$).

We have:

$$E \,\mathfrak{R}\, \langle Q\rangle = (E - \mathrm{Del}(Q)) \cup \mathrm{Add}(Q)$$

Given $S = \langle a_1, ..., a_n\rangle \in \mathrm{Lin}(Q)$, as $\mathrm{Del}(Q) = \mathrm{Del}(S)$ and $\mathrm{Add}(Q) = \mathrm{Add}(S)$, we have:

$$E \,\mathfrak{R}\, \langle Q\rangle = (E - \mathrm{Del}(S)) \cup \mathrm{Add}(S)$$

As Q is an independent set of actions, $\mathrm{Add}(S) \cap \mathrm{Del}(S) = \emptyset$. We have then:

$$\forall\, i \in [1, n-1],\ \mathrm{Del}(a_{i+1}) \cap (\mathrm{Add}(a_1) \cup ... \cup \mathrm{Add}(a_i)) = \emptyset$$

From Lemma 1, we can deduce that:

$$E \,\mathfrak{R}\, \langle Q\rangle = ((...((((E - \mathrm{Del}(a_1)) \cup \mathrm{Add}(a_1)) - \mathrm{Del}(a_2)) \cup \mathrm{Add}(a_2)) - ...) - \mathrm{Del}(a_n)) \cup \mathrm{Add}(a_n)$$

Even more, as Q is independent: $\forall\, a_1 \neq a_2 \in Q$, $\mathrm{Prec}(a_1) \cap \mathrm{Del}(a_2) = \emptyset$. We have then:

$$\forall\, i \in [1, n-1],\ \mathrm{Prec}(a_{i+1}) \cap (\mathrm{Del}(a_1) \cup ... \cup \mathrm{Del}(a_i)) = \emptyset$$

From Lemma 2, we can deduce that:

$$E \,\mathfrak{R}\, S = ((...((((E - \mathrm{Del}(a_1)) \cup \mathrm{Add}(a_1)) - \mathrm{Del}(a_2)) \cup \mathrm{Add}(a_2)) - ...) - \mathrm{Del}(a_n)) \cup \mathrm{Add}(a_n)$$

We have then $E \,\mathfrak{R}\, \langle Q\rangle = E \,\mathfrak{R}\, S$.

Proof of Theorem 1:

Let $E \in (2^P \cup \{\perp\})$ be a state and $S \in (2^A)^* - \{\langle\rangle\}$ a sequence of sets of actions, such that $E \,\mathfrak{R}\, S \neq \perp$ (which implies $E \neq \perp$).

We can make a proof by induction on length(S).

- When length(S) = 1: from Lemma 3, $\forall\, T \in \mathrm{Lin}(\mathrm{first}(S)),\ E \,\mathfrak{R}\, S = E \,\mathfrak{R}\, T$
- We assume the following property is true when length(S) = n, with $S = \langle Q_1, ..., Q_n\rangle$:

  $$E \,\mathfrak{R}\, S \neq \perp \Rightarrow \forall\, S_1 \in \mathrm{Lin}(Q_1), ..., \forall\, S_n \in \mathrm{Lin}(Q_n),\ E \,\mathfrak{R}\, S = E \,\mathfrak{R}\, (S_1 \oplus ... \oplus S_n)$$

  Let us demonstrate it at rank n+1, with $S = \langle Q_1, ..., Q_n, Q_{n+1}\rangle$ and $E \,\mathfrak{R}\, S \neq \perp$:

  $E \,\mathfrak{R}\, S$

  $= E \,\mathfrak{R}\, \langle Q_1, ..., Q_n, Q_{n+1}\rangle$

  $= E \,\mathfrak{R}\, (\langle Q_1, ..., Q_n\rangle \oplus \langle Q_{n+1}\rangle)$

  $= E \,\mathfrak{R}\, \langle Q_1, ..., Q_n\rangle \,\mathfrak{R}\, \langle Q_{n+1}\rangle$ from Property 2

  $= E \,\mathfrak{R}\, (S_1 \oplus ... \oplus S_n) \,\mathfrak{R}\, \langle Q_{n+1}\rangle$ $\forall\, S_1 \in \mathrm{Lin}(Q_1), ..., \forall\, S_n \in \mathrm{Lin}(Q_n)$, (induction hyp.)

  $= E \,\mathfrak{R}\, (S_1 \oplus ... \oplus S_n) \,\mathfrak{R}\, S_{n+1}$ $\forall\, S_1 \in \mathrm{Lin}(Q_1), ..., \forall\, S_n \in \mathrm{Lin}(Q_n), \forall\, S_{n+1} \in \mathrm{Lin}(Q_{n+1})$, from Lemma 3

  $= E \,\mathfrak{R}\, (S_1 \oplus ... \oplus S_n \oplus S_{n+1})$ $\forall\, S_1 \in \mathrm{Lin}(Q_1), ..., \forall\, S_n \in \mathrm{Lin}(Q_n), \forall\, S_{n+1} \in \mathrm{Lin}(Q_{n+1})$, from Property 2

Property 3:

Let $Q \in 2^A$ be a set of actions. Then:

$$Q \text{ independent} \Rightarrow Q \text{ authorized.}$$

Proof: straightforward from the definitions of the independence and authorization relations.

The successive application of $\mathfrak{R}^*$ to two sequences of sets of actions and to a state gives the same result than the application of the concatenation of these two sequences to the same state:

Property 4:

Let $E \in (2^P \cup \{\perp\})$ be a state and $S_1, S_2 \in (2^A)^*$ two sequences of sets of actions. Then:

$$E \,\mathfrak{R}^*\, (S_1 \oplus S_2) = E \,\mathfrak{R}^*\, S_1 \,\mathfrak{R}^*\, S_2$$

Proof: strictly identical to the proof of Property 2 by replacing $\mathfrak{R}$ by $\mathfrak{R}^*$ and independent by authorized.

Theorem 2:

Let $E \in (2^P \cup \{\perp\})$ be a state and $S \in (2^A)^* - \{\langle\rangle\}$ a sequence of sets of actions, with $S = \langle Q_1, ..., Q_n\rangle$. Then:

$$E \,\mathfrak{R}^*\, S \neq \perp \Rightarrow \forall\, S_1 \in \mathrm{LinA}(Q_1), ..., \forall\, S_n \in \mathrm{LinA}(Q_n),\ E \,\mathfrak{R}^*\, S = E \,\mathfrak{R}^*\, (S_1 \oplus ... \oplus S_n).$$

Proof: it is based upon the following two lemmas.

Lemma 4:

Let $E \in 2^P$ be a state and $S \in A^*$ a sequence of actions, with $S = \langle a_1, a_2, ..., a_n \rangle$, such that: $\text{Prec}(S) \subseteq E$ and $\forall\, i \in [1, n-1]$, $\text{Prec}(a_{i+1}) \cap (\text{Del}(a_1) \cup ... \cup \text{Del}(a_i)) = \varnothing$. Then:

$$E\ \Re^*\ S = ((...((((E - \text{Del}(a_1)) \cup \text{Add}(a_1)) - \text{Del}(a_2)) \cup \text{Add}(a_2)) - ...) - \text{Del}(a_n)) \cup \text{Add}(a_n).$$

Proof: strictly identical to the proof of Lemma 2 by replacing $\Re$ by $\Re^*$.

Lemma 5:

Let $E \in (2^P \cup \{\bot\})$ be a state and $Q \in 2^A$ a set of actions. Then:

$$E\ \Re^*\ \langle Q \rangle \neq \bot \Rightarrow \forall\, S \in \text{LinA}(Q),\ E\ \Re^*\ \langle Q \rangle = E\ \Re^*\ S.$$

Proof: strictly identical to the proof of Lemma 3 by replacing $\Re$ by $\Re^*$, independent by authorized, Lin by LinA and Lemma 2 by Lemma 4.

Proof of Theorem 2: strictly identical to the proof of Theorem 1 by replacing $\Re$ by $\Re^*$, Property 2 by Property 4 and Lemma 3 by Lemma 5.

Theorem 3:

Let $E \in (2^P \cup \{\bot\})$ be a state and $S \in (2^A)^*$ a sequence of sets of actions. Then:

$$E\ \Re\ S \neq \bot \Rightarrow E\ \Re^*\ S = E\ \Re\ S.$$

Proof:

Let $E \in (2^P \cup \{\bot\})$ be a state and $S \in (2^A)^*$ a sequence of sets of actions such that $E\ \Re\ S \neq \bot$. As $E\ \Re\ S \neq \bot$, $E \neq \bot$ and the sets of actions of S are independent. From Property 3, they are authorized. As it is the only difference between the definitions of $\Re$ and $\Re^*$, we have $E\ \Re^*\ S = E\ \Re\ S$.

Corollary 1:

Let $E \in (2^P \cup \{\bot\})$ be a state and $S \in (2^A)^*$ a sequence of sets of actions. Then:

$$E\ \Re^*\ S = \bot \Rightarrow E\ \Re\ S = \bot.$$

Proof:

Let $E \in (2^P \cup \{\bot\})$ be a state and $S \in (2^A)^*$ a sequence of sets of actions such that $E\ \Re^*\ S = \bot$. We suppose that $E\ \Re\ S \neq \bot$. From Theorem 3, we have $E\ \Re^*\ S = E\ \Re\ S$, so $E\ \Re^*\ S \neq \bot$: there is a contradiction.

Theorem 4:

Let $E \in (2^P \cup \{\bot\})$ be a state and $S \in (2^A)^* - \{\langle\rangle\}$ a sequence of sets of actions, with $S = \langle Q_1, ... Q_n \rangle$. Then:

$$E\ \Re^*\ S \neq \bot \Rightarrow \forall\, S_1 \in \text{LinA}(Q_1), ..., \forall\, S_n \in \text{LinA}(Q_n),\ E\ \Re^*\ S = E\ \Re\ (S_1 \oplus ... \oplus S_n).$$

Proof:

Let $E \in (2^P \cup \{\bot\})$ be a state and $S \in (2^A)^* - \{\langle\rangle\}$ a sequence of sets of actions, with $S = \langle Q_1, ... Q_n \rangle$ such that $E\ \Re^*\ S \neq \bot$. From Theorem 2, we have:

$$\forall\, S_1 \in \text{LinA}(Q_1), ..., \forall\, S_n \in \text{LinA}(Q_n),\ E\ \Re^*\ S = E\ \Re^*\ (S_1 \oplus ... \oplus S_n)$$

Let $T \in A^*$ with $T = S_1 \oplus ... \oplus S_n$ and $S_1 \in \text{LinA}(Q_1), ..., S_n \in \text{LinA}(Q_n)$. As $E\ \Re^*\ S \neq \bot$, we have $E\ \Re^*\ T \neq \bot$. Now $T \in A^*$ so each set of actions that compose T are singletons: they are independent sets. We have then $T = \langle a_1, ..., a_m \rangle$ with every $\{a_i\}$ being independent sets of actions. Moreover, as $E\ \Re^*\ T \neq \bot$, after every application of an action a_i from T, the preconditions of a_{i+1} are verified. From these two conditions, we can deduce that $E\ \Re\ T \neq \bot$; from Theorem 3, we can deduce that $E\ \Re^*\ T = E\ \Re\ T$ and so $E\ \Re^*\ S = E\ \Re\ T$.

Theorem 5:

Let $Q \in 2^A$ be a set of actions and AG(N, C) the authorization graph of Q. Then:

$$\text{AG has no cycle} \Leftrightarrow Q \text{ is authorized.}$$

Proof:

Let $Q \in 2^A$ be a set of actions and AG(N, C) the authorization graph of Q. We know that AG has no cycle iff there exists a topological order on the nodes of AG (that induces an order on the actions):

$N = \{\text{n}(a_1), ..., \text{n}(a_m)\}$ with $\forall\, 1 \leq i < j \leq m$, $(\text{n}(a_j), \text{n}(a_i)) \notin C$

$\Leftrightarrow \forall\, 1 \leq i < j \leq m$, $\text{not}(\text{not}(a_i \angle a_j))$

$\Leftrightarrow \forall\, 1 \leq i < j \leq m$, $a_i \angle a_j$

$\Leftrightarrow Q$ is authorized.

Theorem 6:

Let $Q \in 2^A$ be a set of actions. Then:

$$Q \text{ authorized} \Leftrightarrow \textbf{SearchSeq}(Q) \neq \mathit{fail}$$

Proof:

Let $Q \in 2^A$ be a set of actions and AG(N, C) its authorization graph.

We show first that not(Q authorized) $\Rightarrow$ **SearchSeq**(Q) = *fail*.
If Q is forbidden, then from Theorem 5 there exists a cycle in AG. The algorithm will detect it and return *fail*, because during an iteration every node present in the graph will have at least one predecessor. Indeed, during every iteration, the algorithm detects each node without predecessor and removes them; and the nodes of a cycle always have at least one predecessor. So these nodes cannot be removed.

We show then that **SearchSeq**(Q) = *fail* $\Rightarrow$ not(Q authorized).
If the algorithm returns *fail*, we are in the following situation: we are trying to extract the nodes without predecessors of a graph GA'(N', C'), with $N' \subseteq N$, $N' \neq \emptyset$ and $C' \subseteq C$. As $N' \neq \emptyset$, there is nodes in GA'. But these nodes have at least one predecessor, so there is a cycle in GA'. From , the set $Q' = \{a \mid \text{n}(a) \in N'\}$ is not authorized. But as $Q' \subseteq Q$, we can conclude that Q is forbidden.

Theorem 8:

Let $E \in 2^P$ be a state and $S \in (2^A)^* - \{\langle\rangle\}$ a sequence of sets of actions, with $S = \langle Q_1, ..., Q_n\rangle$. Then:

$$E\ \Re^*\ S \neq \bot \Rightarrow E\ \Re^*\ S = E\ \Re\ (\textbf{SearchSeq}(Q_1) \oplus ... \oplus \textbf{SearchSeq}(Q_n)).$$

Proof: it is based upon the following two lemmas.

Lemma 6:

Let $Q \in 2^A$ be a set of actions such that **SearchSeq**(Q) = $\langle Q_1, ..., Q_n\rangle$. Then:

$$\forall\ S_1 \in \text{Lin}(Q_1), ..., \forall\ S_n \in \text{Lin}(Q_n),\ S_1 \oplus ... \oplus S_n \in \text{LinA}(Q)$$

Proof:

Let $Q \in 2^A$ be a set of actions such that **SearchSeq**(Q) = $\langle Q_1, ..., Q_n\rangle$. Let $S_1 \in \text{Lin}(Q_1)$, ..., $S_n \in \text{Lin}(Q_n)$.
The solution returned by the algorithm is such that:

$$Q = Q_1 \cup ... \cup Q_n, \text{ with } Q_1 \cap ... \cap Q_n = \emptyset$$

We can deduce immediately that $S_1 \oplus ... \oplus S_n \in \text{Lin}(Q)$.
We must now prove that $S_1 \oplus ... \oplus S_n$ is an authorized sequence of actions.
We suppose that $S_1 \oplus ... \oplus S_n$ is not authorized. We have then two possibilities:

- Either $\exists\ i \in [1, n]$ such that $S_i = \langle a_1, ..., a_m\rangle$ and $\exists\ a_j, a_k \in S_i$ with $j < k$, such that not($a_j \angle a_k$).
  We have then $a_j \in Q_i$ and $a_k \in Q_i$. By construction, if these two actions are in the same set of actions, it is because their respective nodes, in the authorization graph, have no predecessors in the same iteration. In particular, n(a_k) is not a predecessor of n(a_j). We have then not(not($a_j \angle a_k$)), that is to say $a_j \angle a_k$: there is a contradiction.
- Either $\exists\ i, j \in [1, n]$ such that $i < j$, and $\exists\ a_1 \in S_i$, $\exists\ a_2 \in S_j$ such that not($a_1 \angle a_2$).
  But, if a_1 is in a set of actions that has been found by the algorithm before the one in which is a_2, it is because a_1 had no predecessor in an authorization graph in which a_2 was present. So we had n(a_2) do not precede n(a_1), that is to say not(not($a_1 \angle a_2$)), and then $a_1 \angle a_2$: there is a contradiction.

Lemma 7:

Let $E \in 2^P$ be a state and $Q \in 2^A$ a set of actions. Then:

$$E\ \Re^*\ \langle Q\rangle \neq \bot \Rightarrow E\ \Re^*\ \langle Q\rangle = E\ \Re\ \textbf{SearchSeq}(Q).$$

Proof:

As $E\ \Re^*\ \langle Q\rangle \neq \bot$, by definition of $\Re^*$, Q is authorized. From Theorem 6, **SearchSeq**(Q) $\neq$ *fail*, and **SearchSeq**(Q) = $\langle Q_1, ..., Q_n\rangle$. We have then:

$E\ \Re$ **SearchSeq**(Q)
= $E\ \Re\ \langle Q_1, ..., Q_n\rangle$
= $E\ \Re\ (S_1 \oplus ... \oplus S_n)$ $\quad\forall\ S_1 \in \text{Lin}(Q_1), ..., \forall\ S_n \in \text{Lin}(Q_n)$, from Theorem 1
= $E\ \Re$ S' $\quad$ with S' $\in$ LinA(Q), from Lemma 6

As $E\ \Re^*\ \langle Q\rangle \neq \bot$, we know from Theorem 4 that:

$$\forall\ S \in \text{LinA}(Q),\ E\ \Re^*\ \langle Q\rangle = E\ \Re\ S$$

We can deduce that

$$E\ \mathfrak{R}^*\ \langle Q\rangle = E\ \mathfrak{R}\ S' = E\ \mathfrak{R}\ \textbf{SearchSeq}(Q).$$

Proof of Theorem 7:

Let $E \in 2^P$ be a state and $S \in (2^A)^* - \{\langle\rangle\}$ a sequence of sets of actions, such that $E\ \mathfrak{R}^*\ S \neq \bot$.
We can make a proof by induction on length(S).

- When length(S) = 1: $E\ \mathfrak{R}^*\ S = E\ \mathfrak{R}$ **SearchSeq**(first(S)) from Lemma 7
- We suppose the following property true at rank n, with $S = \langle Q_1, ..., Q_n\rangle$:

$$E\ \mathfrak{R}^*\ S \neq \bot \Rightarrow E\ \mathfrak{R}^*\ S = E\ \mathfrak{R}\ (\textbf{SearchSeq}(Q_1) \oplus ... \oplus \textbf{SearchSeq}(Q_n))$$

We must now prove it at rank n+1, with $S = \langle Q_1, ..., Q_n\rangle$ such that $E\ \mathfrak{R}^*\ \langle Q_1, ..., Q_n, Q_{n+1}\rangle \neq \bot$:

$E\ \mathfrak{R}^*\ \langle Q_1, ..., Q_n, Q_{n+1}\rangle)$
$= E\ \mathfrak{R}^*\ \langle Q_1, ..., Q_n \oplus Q_{n+1}\rangle$
$= E\ \mathfrak{R}^*\ \langle Q_1, ..., Q_n\rangle\ \mathfrak{R}^*\ \langle Q_{n+1}\rangle$ from Property 4
$= E\ \mathfrak{R}\ (\textbf{SearchSeq}(Q_1) \oplus ... \oplus \textbf{SearchSeq}(Q_n))\ \mathfrak{R}^*\ \langle Q_{n+1}\rangle$ (induction hyp.)
$= E\ \mathfrak{R}\ (\textbf{SearchSeq}(Q_1) \oplus ... \oplus \textbf{SearchSeq}(Q_n))\ \mathfrak{R}\ \textbf{SearchSeq}(Q_{n+1})$ from Lemma 7
$= E\ \mathfrak{R}\ (\textbf{SearchSeq}(Q_1) \oplus ... \oplus \textbf{SearchSeq}(Q_n) \oplus \textbf{SearchSeq}(Q_{n+1}))$ from Property 2

References

[1] C. Bäckström, Computational aspects of reordering plans, Journal of Artificial Intelligence Research 9 (1998) 99–137.

[2] A. Blum, M. Furst, Fast planning through planning-graphs analysis, in: Proc. Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95), Montreal, Quebec, 1995, pp. 1636–1642.

[3] A. Blum, M. Furst, Fast planning through planning-graphs analysis, Artificial Intelligence 90 (1997) 281–300.

[4] M. Cayrol, P. Régnier, V. Vidal, LCGP : une amélioration de Graphplan par relâchement de contraintes entre actions simultanées, in: Proc. Douzième Congrès de Reconnaissance des Formes et Intelligence Artificielle (RFIA-2000), Paris, France, 2000, pp. 79–88.

[5] M. Cayrol, P. Régnier, V. Vidal, New results about LCGP, a Least Committed GraphPlan, in: Proc. Fith International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000), Breckenridge, CO, 2000, pp. 273–282.

[6] Y. Dimopoulos, B. Nebel, J. Koehler, Encoding planning problems in nonmonotonic logic programs, in: Proc. Fourth European Conference on Planning (ECP-97), Toulouse, France, 1997, pp. 167–181.

[7] M. Fox, D. Long, The automatic inference of state invariants in TIM, in: Journal of Artificial Intelligence Research 9 (1998) 367–421.

[8] M. Fox, D. Long, The detection and exploitation of symmetry in planning problems, in: Proc. Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99), Stockholm, Sweden, 1999, pp. 956–961.

[9] E. Guéré, R. Alami, A possibilistic planner that deals with non-determinism and contingency, in: Proc. Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99), Stockholm, Sweden, 1999, pp. 996–1001.

[10] S. Kambhampati, Improving Graphplan's search with EBL & DDB techniques, in: Proc. Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99), Stockholm, Sweden, 1999, pp. 982–987.

[11] S. Kambhampati, Planning graph as a (dynamic) CSP: exploiting EBL, DDB and other CSP techniques in Graphplan, Journal of Artificial Intelligence Research 12 (2000) 1–34.

[12] S. Kambhampati, R. S. Nigenda, Distance-based goal-ordering heuristics for Graphplan, in: Proc. Fith International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000), Breckenridge, CO, 2000, pp. 315–322.

[13] H. Kautz, B. Selman, Unifying SAT-based and Graph-based Planning, in: Proc. Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99), Stockholm, Sweden, 1999, pp. 318–325.

[14] J. Koehler, Planning under resources constraints, in: Proc. Thirteenth European Conference on Artificial Intelligence (ECAI-98), Brighton, UK, 1998, pp. 489–493.

[15] J. Koehler, B. Nebel, J. Hoffmann, Y. Dimopoulos, Extending planning-graphs to an ADL subset, in: Proc. Fourth European Conference on Planning (ECP-97), Toulouse, France, 1997, pp. 273–285.

[16] D. Long, M. Fox, The efficient implementation of the plan-graph in STAN, Journal of Artificial Intelligence Research 10 (1999) 87–115.

[17] K. Mehlhorn, Data structures and Algorithms 2: Graph Algorithms and NP–Completness, Springer-Verlag, Berlin, 1984.

[18] S. Mittal, B. Falkenhainer, Dynamic constraint satisfaction problems, in: Proc. Eighth National Conference on Artificial Intelligence (AAAI-90), Boston, MA, 1990, pp. 25–32.

[19] B. Nebel, Y. Dimopoulos, J. Koehler, Ignoring irrelevant facts and operators in plan generation, in: Proc. Fourth European Conference on Planning (ECP-97), Toulouse, France, 1997, pp. 338–350.

[20] P. Régnier, B. Fade, Complete determination of parallel actions and temporal optimization in linear plans of actions, in: Proc. European Workshop on Planning (EWSP-91), Sankt Augustin, Germany, 1991, pp. 100–111.

[21] J. Rintanen, A planning algorithm not based on directionnal search, in: Proc. Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR-98), Trento, Italy , 1998, pp. 617–624.

[22] D. Smith, D. Weld, Temporal Planning with mutual exclusion reasoning, in: Proc. Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99), Stockholm, Sweden, 1999, pp 326–333.

[23] D. Weld, C. Anderson, D. Smith, Extending Graphplan to handle uncertainty and sensing actions, in: Proc. Fifteenth National Conference on Artificial Intelligence (AAAI-98), Madison, WI, 1998, pp. 897–904.

[24] T. Zimmerman, S. Kambhampati, Exploiting symmetry in the planning-graph via explanation-guided search, in: Proc. Sixteenth National Conference on Artificial Intelligence (AAAI-99), Orlando, FL, 1999, pp. 605–611.