LOLLIPOP: Generating and using proper plan and negative refinements for online partial order planning

Paper ID#42

Abstract. In recent years, automated planning domain has mainly focused on performances and advances in state space planning to improve scalability. That orientation shadows other less efficient ways of doing like Partial Ordering Planning (POP) that has the advantage to be much more flexible. This flexibility allows these planners to take incomplete plans as input and to refine them into an optimized solution for the problem. This paper presents a set of mechanisms, called aLternative Optimization with partiaL pLan Injection Partial Ordered Planning (LOLLIPOP), that adapts the classical POP algorithm. The aim is to obtain a high plan quality while allowing fast online re-planning. Our algorithm builds a proper plan from the domain compilation that gives information on the possible behavior of operators. A safe proper plan is generated at initialization and used as LOLLIPOP's input to reduce the overhead of the initial subgoal backchaining. We prove that the use of these new mechanisms keeps POP sound and complete.

Introduction

Until the end of the 90s, Plan-Space Planning (PSP) was generally preferred by the automated planning community. However, more recently, drastic improvements in state search planning (SSP) were made possible by advanced and efficient heuristics. This allowed those planners to scale up more efficiently than plan-space search ones, notably thanks to approaches like GraphPlan [1], fast-forward [2], LAMA [3] and Fast Downward Stone Soup [4]. This evolution led to a preference for performances upon other aspects of the problem of automated planning like expressiveness and flexibility, which are clear advantages of PSP over SSP. For instance, Partial Order Planning (POP), also known as Partial Order Causal Link (POCL), can take advantage of its least commitment strategy [5]. It allows to describe a partial plan with only the necessary order of the actions and not a fixed sequence of steps. Thus, POP has greater flexibility at plan execution time [6]. It has also been proven to be well suited for multi-agent planning [7] and temporal planning [8] . These advantages made UCPOP [9] one of the preferred POP planners of its time with works made to port some of its characteristics into state-based planning [10].

Related works already tried to explore new ideas to make POP an attractive alternative to regular state-based planners like the appropriately named “Reviving partial order planning” [11] and VHPOP [12]. More recent efforts [13], [14] are trying to adapt the powerful heuristics from state-based planning to POP's approach. An interesting approach of these last efforts is found in [15] with meta-heuristics based on offline training on the domain. However, we clearly note that only a few papers lay the emphasis upon plan quality using POP [16], [17].

The present paper lays the base for a project that aims to assist dependent persons to accomplish tasks and so, to infer the pursued goals of the persons. One way of achieving goal recognition is to use inverted planning for plan inference, as for instance in [18]. This requires computing online planning with a feed of observational data. In a real world context, we also need to handle data derived from limited or noisy sensors. In this context, using inverted planning requires that the planner should be resilient to misleading input plans. From that perspective, we need a planner able to meet the following criteria:

- repair and optimize existing plans,
- perform online planning efficiently
- retain performances on complex but medium sized problems

In this article, we propose to improve POP with better refining techniques and resolver selection for online partial order planning. Classical POP algorithms don't meet most of these criteria but can be enhanced to fit the first one to fit plan reparation, as for instance in [19]. Usually, POP algorithms take a problem as an input and use a loop or a recursive function to refine the plan into a solution. We can't simply use the refining recursive function in order to be able to use our existing partial plan. This causes multiples side effects if the input plan is suboptimal. *Our view* on the problem diverges from other works: PSP is a very different approach compared to SSP. It is usually more computationally expensive than modern state space planners but brings several advantages. We want to make the most of them instead of trying to change POP's fundamental nature. That view is at the core of our model: we use the refining and least commitment abilities of POP in order to improve online performances and quality. In order to achieve this, we start by computing a *proper plan* that is computed offline with the input of the domain. Proper plan definition and generation is given in section 2.1. Using existing partial plans as POP's input leads to several issues, mainly because of new flaw types that aren't treated in classical POP. This is why we focus the section 2.2 of our paper on plan quality and negative refinements. We, therefore, introduce new negative flaws and resolvers that aim to fix and optimize the plan. Side effects of negative flaws and resolvers can lead to conflicts. In order to avoid them and to enhance performances and quality, the algorithm needs resolver and flaw selection mechanisms that are explained in the section 2.3. All these mechanisms are part of our aLternative Optimization with partiaL pLan Injection Partial Ordered Planning (LOLLIPOP) algorithm presented in details in section 2.4. We prove that the use of these new mechanisms keeps LOLLIPOP sound and complete in section 3. Experimental results and benchmarks are presented in the section 4 of this paper.

1 Partial Order Planning Preliminaries

1.1 Notation

In this paper, we use the notation defined in table 1. We use the symbol $\pm$ to signify that there is a notation for the positive and negative symbols but the current formula works regardless the sign. All related notions will be defined later.

Table 1. Most used symbols in the paper.

Symbol	Description
$pre(o), eff(o)$	Preconditions and effects of the operator o
Δ	Planning domain
Π	Planning problem
$T.x$	Access element x of tuple T
$l_{\rightarrow}, l_{\leftarrow}$	Source and target of the causal link l
$o_1 \succ o_2$	Precedence operator (o_1 precedes o_2)
O^P	Proper plan of the set of operators O
$p.d^{\pm}(o)$	Outgoing and incoming degrees of o
$o.d^{\pm}$	Proper degrees of o ($\|pre(o)\|$ and $\|eff(o)\|$)
$p.L^{\pm}(o)$	Outgoing and incoming causal links of o
$C(p)$	Set of cycles in partial plan p
$C_p(o)$	Set of cycles in p which o is part of
$SC_p(o)$	$\{o\}$ if o has a self cycle in p, $\emptyset$ otherwise
$F^{\pm}(p)$	Set of flaws in p
$r(f)$	Resolvers of the flaw f
$f.n$	Needer of the flaw f
$f(p)$	Application of the flaw f on plan p
$fs(p)$	Full support of p
$p \models \Pi$	The partial plan p is a valid solution of Π

1.2 Basic Definitions

Planning systems need a representation of its fluents and operators. Our framework is based on a classical domain definition. This is defined in the following definition from [20].
Definition 1 (Domain)**.** *We define our planning domain as a tuple $\Delta = \langle T, C, P, F, O\rangle$ where*

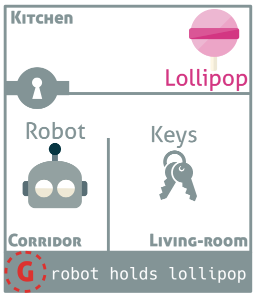
- *T are the **types**,*
- *C is the set of **domain constants**,*
- *P is the set of **properties** with their arities and typing signatures,*
- *F represents the set of **fluents** defined as potential equations over the terms of the domain,*
- *O is the set of optionally parameterized **operators** with preconditions and effects.*

Along with a domain, every planner needs a problem representation. We use the classical problem representation with some special additions.
Definition 2 (Problem)**.** *The planning problem is defined as a tuple $\Pi = \langle \Delta, C_\Pi, I, G, p\rangle$ where*

- *Δ is a planning domain,*
- *C_Π is the set of **problem constants** disjoint from the domain constants C,*
- *I is the **initial state**,*
- *G is the **goal**,*
- *p is a given **partial plan**.*

The framework uses the *closed world assumption* in which all predicates and properties that aren't defined in the initial step are assumed false or don't have a value.

Figure 1 portrays an example of a planning domain and problem that we use as a guideline throughout the article.



```
go(?movable, ?room)
     ?movable is Movable, ?room is Room
     pre ?movable room != ?room,
         ?room locked false
     eff ?movable room ?room

grab(?graber, ?pickable)
     ?graber is Graber, ?pickable is Pickable
     pre ?graber holds nothing,
         ?graber room = ?pickable room
     eff ?graber holds ?pickable

unlock(?graber, ?room)
     ?graber is Graber, ?room is Room
     pre ?room locked, ?graber holds keys
     eff ?graber holds nothing,
         ?room locked false
```

Figure 1. Example domain and problem featuring a robot that aims to fetch a lollipop in a locked kitchen. The operator `go` is used for movable objects (such as the robot) to move to another room. The `grab` operator is used by grabbers to hold objects and the `unlock` operator is used to open a door when the robot holds the key.

In order to simplify this framework, we need to introduce some differences from the classical partial plan representation. First, the partial plan is a part of the problem tuple as it is a needed input of the LOLLIPOP algorithm.
Definition 3 (Partial Plan)**.** *We define a partial plan as a tuple $\langle S, L, B\rangle$ with S the set of **steps** (semi or fully instantiated operators also called* actions*), L the set of **causal links**, and B the set of **binding constraints**.*

Second we factorize the set of *ordering constraints*, used in classical representations, as being part of the causal links. Indeed, causal links are always supported by an ordering constraint. The only case where bare ordering constraints are needed is in threats. We represent them with **bare causal links**. These are stored as causal links without bearing any fluents. Causal links can be represented by their bore fluents called *causes*. We note $f \in l$ the fact that a causal link l bears the fluent f (bare causal links are noted $l = \emptyset$). That allows us to introduce the **precedence operator** noted $a_i \succ a_j$ with $a_i, a_j \in S$ iff there is a path of causal links that connects a_i to a_j. We call a_i *anterior* to a_j.

A specificity of Partial Order Planning is that it fixes flaws in a partial plan in order to refine it into a valid plan that is a solution to the given problem. Next, we define the classical flaws in our framework.
Definition 4 (Subgoal)**.** *A flaw in a partial plan, called subgoal, is a missing causal link required to satisfy a precondition of a step. We can note a subgoal as: $a_p \xrightarrow{s} a_n \notin L \mid \{a_p, a_n\} \subseteq S$ with a_n called the **needer** and a_p an eventual **provider** of the fluent s. This fluent is called* open condition *or **proper fluent** of the subgoal.*
Definition 5 (Threat)**.** *A flaw in a partial plan called threat consists of having an effect of a step that can be inserted between two actions with a causal link that is threatened by the said effect. We say that a step a_b is threatening a causal link $a_p \xrightarrow{t} a_n$ iff $a_b \neq a_p \neq a_n \wedge \neg t \in eff(a_b) \wedge a_p \succ a_b \succ a_n$ with a_b being the **breaker**, a_n the* needer *and a_p a* provider *of the* proper fluent *t.*

Flaws are fixed via the application of a resolver to the plan. A flaw can have several resolvers that match its needs.
Definition 6 (Resolvers)**.** *A resolver is a potential causal link defined as a tuple $r = \langle a_s, a_t, f\rangle$ with:*

- *$a_s, a_t \in S$ being the source and the target of the resolver,*

- f *being the considered fluent.*

For classical flaws, the resolvers are simple to find. For a *subgoal* the resolvers are sets of the potential causal links between a possible provider of the proper fluent and the needer. To solve a *threat* there are mainly two resolvers: a causal link between the needer and the breaker called **demotion** or a causal link between the breaker and the provider called **promotion**.

Once the resolver is applied, another important step is needed in order to be able to keep refining. The algorithm needs to take into account the **side effects** the application of the resolver had on the partial plan. Side effects are searched by type.
Definition 7 (Side effects)**.** *Flaws that arise because of the application of a resolver on the partial plan are called causal side effects or* related flaws. *They are caused by an action* a_t *called the* ***trouble maker*** *of a resolver. This action is the* source *of the resolver applied onto the plan.*

We can derive this definition for subgoals and threats:

- **Related Subgoals** are all the open conditions inserted with the *trouble maker*. The subgoals are often searched using the preconditions of the trouble maker and added when no causal links satisfy them.
- **Related Threats** are the causal links threatened by the insertion of the *trouble maker*. They are added when there is no causal path that prevents the action to interfere with the threatened causal link.

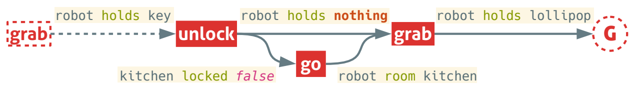


Figure 2. Example of a partial plan occurring during the computation of POP on the previous example domain illustrated by figure 1. Full arrows are existing causal links. The dotted arrow and operator depict the current resolver.

In the partial plan presented in figure 2, we consider that a resolver providing the fluent `robot holds key` is considered. This resolver will introduce the open conditions `robot holds nothing, key room _room, robot room _room` since it just introduced this instantiation of the `grab` operator in the partial plan. This resolver triggers related subgoals with new `grab` operator as needer. The potentially related threat of this resolver is that the effect `robot holds key` might threaten the link between the existing `unlock` and `grab` steps, but won't be considered since there are no way the new step can be inserted after `unlock`.

1.3 Classical POP Algorithm

The classical POP algorithm starts with a simple partial plan that contains only the initial and goal steps and refines its *flaws* until they are all resolved to make the found plan a solution of the problem.

The algorithm 1 is inspired by [21]. This POP implementation uses an agenda of flaws that is efficiently updated after each refinement of the plan. At each iteration, a flaw is selected and removed from the agenda (line 7). A resolver for this flaw is then selected and applied (line 10). If all resolvers cause failures, the algorithm backtracks to the last resolver selection to try another one. The algorithm terminates

Algorithm 1 Classical Partial Order Planning

```
1  function POP(Queue of Flaws $a$, Problem $\Pi$)
2      POPULATE($a$, $\Pi$)                         Populate agenda only on first call
3      if $\Pi.G = \emptyset$ then          Goal is empty, default solution is provided
4          $\Pi.p.L \leftarrow (I \rightarrow G)$
5      if $a = \emptyset$ then
6          return Success                                        Stop all recursion
7      Flaw $f \leftarrow$ CHOOSE($a$)                          Non deterministic choice
8      Resolvers $R \leftarrow$ RESOLVERS($f$, $\Pi$)
9      for all $r \in R$ do                      Non deterministic choice operator
10         APPLY($r$, $\Pi.p$)                          Apply resolver to partial plan
11         SideEffects $s \leftarrow$ SIDEEFFECT($r$)
12         APPLY($s$, $a$)                               Side effects of the resolver
13         if POP($a$, $\Pi$) = Success then                     Refining recursively
14             return Success
15         REVERT($r$, $\Pi.p$)                       Failure, undo resolver insertion
16         REVERT($s$, $a$)                     Failure, undo side effects application
17     return Failure        Revert to last non deterministic choice of resolver
```

when no more resolver fits a flaw (`Failure`) or when all flaws have been fixed (`Success`).

This standard implementation has several limitations. First, it can easily make poor choices that will lead to excessive backtracking. It also can't undo redundant or non-optimal links if they don't lead to backtracking. To explain these limitations, we use the example described in figure 1 where a robot must fetch a lollipop in a locked room. This problem is solvable by regular POP algorithms. However, we can have some cases where small changes in POP's inputs can cause a lot of unnecessary backtrackings. For example, if we add a new action called `dig_through_wall` that has as effect to be in the desired room but that requires a *jackhammer*, the algorithm will simply need more backtracking. The effects could be worse if obtaining a jackhammer would require numerous steps (for example building it). This problem can be solved most of the time using simple flaw selection mechanisms. The other limitation arises when the plan has been modified. This can occur in the context of dynamic environments of online planning applications. When POP is modified to use input plans [22], these partial plans are carefully designed to avoid violating the properties required by POP to operate. In our case, the plan can contain misleading information and this can cause a variety of new problems that can only be fixed using new refinements methods.

2 LOLLIPOP's Approach

In this section, proper plan generation used by LOLLIPOP to ease the initial backchaining of POP is presented. Then we introduced new negative flaws for online planning refinements. LOLLIPOP algorithm and its resolvers and flaws selection based on the proper plan are finally detailed.

2.1 Proper Plan Generation

One of the main contributions of the present paper is our use of the concept of *proper plan*. First of all, we define this notion.
Definition 8 (Proper Plan)**.** *A proper plan* O^P *of a set of operators* O *is a labeled directed graph that binds two operators with the causal link* $o_1 \xrightarrow{f} o_2$ *iff it exists at least a unifying fluent* $f \in eff(o_1) \cap pre(o_2)$.

This definition was inspired by the notion of domain causal graph as explained in [20] and originally used as heuristic in [23]. Causal graphs have fluents as their nodes and operators as their edges. Proper plans are the opposite: an *operator dependency graph* for a set of actions. A similar structure was used in [24] that builds the operator dependency graph of goals and uses precondition nodes instead of labels. This structure is very useful for getting information on the *shape of a problem*. This shape leads to an intuition based on the potential usefulness or hurtfulness of operators. Cycles in this graph denote the dependencies of operators. We call *co-dependent* operators that form a cycle. If the cycle is made of only one operator (self-loop), then it is called *auto-dependent*.

While building this proper plan, we need a **providing map** that indicates, for each fluent, the list of operators that can provide it. This is a simpler version of the causal graphs that is reduced to an associative table easier to update. The list of providers can be sorted in order to drive resolver selection (as detailed in section 2.3). A **needing map** is also built but is only used for proper plan generation. We note Δ^P the proper plan built with the set of operators in the domain Δ. In the figure 3, we illustrate the application of this mechanism on our example from figure 1. Continuous lines correspond to the *domain proper plan* computed during domain compilation time.

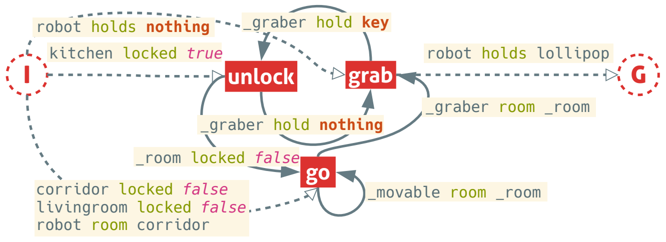


Figure 3. Diagram of the proper plan of example domain. Full arrows represents the domain proper plan and dotted arrows the dependencies added to inject the initial and goal steps.

The generation of the proper plan is detailed in algorithm 2. It explores the operators space and builds a providing and needing map that gives the provided and needed fluents for each operator. Once done it iterates on every precondition and searches for a satisfying cause in order to add the causal links to the proper plan.

Algorithm 2 Proper plan generation and update algorithm

```
function ADDVERTEX(Operator $o$)
    CACHE($o$)                          Update of the providing and needing map
    if binding then    boolean that indicates if the binding was requested
        BIND($o$)
function CACHE(Operator $o$)
    for all $eff \in eff(o)$ do          Adds $o$ to the list of providers of $eff$
        ADD($providing, eff, o$)
    ...                            Same operation with needing and preconditions
function BIND(Operator $o$)
    for all $pre \in pre(o)$ do
        if $pre \in providing$ then
            for all $p \in$ GET($providing, pre$) do
                Link $l \leftarrow$ GETEDGE($p, o$)           Create the link if needed
                $l \leftarrow l \cup \{pre\}$                   Add the fluent as a cause
    ...                                  Same operation with needing and effects
```

To apply the notion of proper plan to planning problems, we just need to add the initial and goal steps to the proper plan. In figure 3, we depict this insertion with our previous example using dotted lines. However, since proper plans may have cycles, they can't be used directly as input to POP algorithms to ease the initial backchaining. Moreover, the process of refining a proper plan into a usable one could be more computationally expensive than POP itself.

In order to give a head start to the LOLLIPOP algorithm, we propose to build proper plans differently with the algorithm detailed in algorithm 3. A similar notion was already presented as "basic plans" in [22]. These "basic" partial plans use a more complete but slower solution for generation that ensures that each selected steps is *necessary* for the solution. In our case, we built a simpler solution that can solve some basic planning problems but that also make early assumption (since our algorithm can handle them). It does a simple and fast backward construction of a partial plan driven by the providing map. Therefore, it can be tweaked with the powerful heuristics of state search planning.

Algorithm 3 Safe proper plan generation algorithm

```
1  function SAFE(Problem $\Pi$)
2      Stack $open \leftarrow [\Pi.G]$
3      Stack $closed \leftarrow \emptyset$
4      while $open \neq \emptyset$ do
5          Operator $o \leftarrow$ POP($open$)                  Remove $o$ from $open$
6          PUSH($closed, o$)
7          for all $pre \in pre(o)$ do
8              Operators $p \leftarrow$ GETPROVIDING($\pi, pre$) Sorted by usefulness
9              if $p = \emptyset$ then                          (see section 2.3)
10                 $\Pi.p.S \leftarrow \Pi.p.S \setminus \{p\}$
11                 continue
12             Operator $o' \leftarrow$ GETFIRST($p$)
13             if $o' \in closed$ then
14                 continue
15             if $o' \notin \Pi.p.S$ then
16                 PUSH($open, o'$)
17             $\Pi.p.S \leftarrow \Pi.p.S \cup \{o'\}$
18             Link $l \leftarrow$ GETEDGE($o', o$)             Create the link if needed
19             $l \leftarrow l \cup \{pre\}$                     Add the fluent as a cause
```

This algorithm is very useful since it is specifically used on goals. The result is a valid partial plan that can be used as input to POP algorithms.

2.2 Negative Refinements and Plan Optimization

Classical POP algorithm works upon a principle of positive plan refinements. The two standard flaws (subgoals and threats) are fixed by *adding* steps, causal links, or variable binding constraints to the partial plan. Online planning needs to be able to *remove* parts of the plan that are not necessary for the solution. Since we assume that the input partial plan is quite complete, we need to define new flaws to optimize and fix this plan. These flaws are called *negative* as their resolvers apply subtractive refinements on partial plans.

Definition 9 (Alternative)**.** *An alternative is a negative flaw that occurs when it exists a better provider choice for a given link. An alternative to a causal link* $a_p \xrightarrow{f} a_n$ *is a provider* a_b *that has a better* utility value *than* a_p*.*

The **utility value** of an operator is a measure of usefulness at the heart of our ranking mechanism detailed in section 2.3. It uses the incoming and outgoing degrees of the operator in the domain proper plan to measure its usefulness.

Finding an alternative to an operator is computationally expensive. It requires searching a better provider for every fluent needed by a step. In order to simplify that search, we select only the best provider for a given fluent and check if the one used is the same. If not, we add the alternative as a flaw. This search is done only on updated steps for online planning. Indeed, the safe proper plan mechanism is guaranteed to only choose the best provider (algorithm 3 at line 12). Furthermore, subgoals won't introduce new fixable alternative as they are guaranteed to select the best possible provider.

Definition 10 (Orphan)**.** *An orphan is a negative flaw that means that a step in the partial plan (other than the initial or goal step) is not participating in the plan. Formally, a_o is an orphan iff $a_o \neq I \land a_o \neq G \land \left(p.d^+(a_o) = 0\right) \lor \forall l \in p.L^+(a_o), l = \emptyset$.*

With $p.d^+(a_o)$ being the *outgoing degree* of a_o in the directed graph formed by p and $p.L^+(a_o)$ being the set of *outgoing causal links* of a_o in p. This last condition checks for *hanging orphans* that are linked to the goal with only bare causal links (introduced by threat resolution).

The introduction of negative flaws requires modifying the resolver definition (cf. definition 6).
Definition 11 (Signed Resolvers)**.** *A signed resolver is a resolver with a notion of sign. We add to the resolver tuple the sign of the resolver noted $s \in \{+, -\}$.*

The solution to an alternative is a negative refinement that simply removes the targeted causal link. This causes a new subgoal as a side effect, that will prioritize its resolver by its rank (explained in section 2.3) and then pick the first provider (the most useful one). The resolver for orphans is the negative refinement that is meant to remove a step and its incoming causal link while tagging its providers as potential orphans.

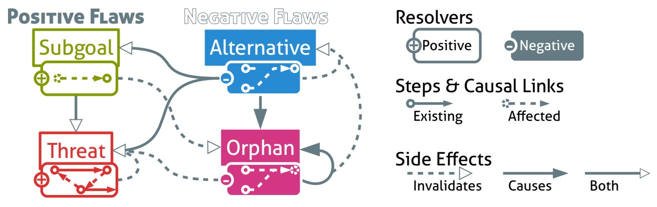


Figure 4. Schema representing flaws with their signs, resolvers and side effects relative to each other

The side effect mechanism also needs an upgrade since the new kind of flaws can interfere with one another. This is why we extend the side effect definition (cf. definition 7) with a notion of sign.
Definition 12 (Signed Side Effects)**.** *A signed side effect is either a regular* causal side effect *or an* invalidating side effect. *The sign of a side effect indicates if the related flaw needs to be added or removed from the agenda.*

The figure 4 exposes the extended notion of signed resolvers and side effects. When treating positive resolvers, nothing needs to change from the classical method. When dealing with negative resolvers, we need to search for additional subgoals and threats. Deletion of causal links and steps can cause orphan flaws that need to be identified for removal.

In the method described in [25], a **invalidating side effect** is explained under the name of *DEnd* strategy. In classical POP, it has been noticed that threats can disappear in some cases if subgoals or other threats were applied before them. For our mechanisms, we decide to gather under this notion every side effects that removes the need to consider a flaw. For example, orphans can be invalidated if a subgoal selects the considered step. Alternatives can remove the need to compute further subgoal of an orphan step as orphans simply remove the need to fix any flaws that concern the selected step.

These interactions between flaws are decisive for the validity and efficiency of the whole model, that is why we aim to drive flaw selection in a very rigorous manner.

2.3 Driving Flaws and Resolvers Selection

Resolvers and flaws selection are the keys to improving performances. Choosing a good resolver helps to reduce the branching factor that accounts for most of the time spent on running POP algorithms [26]. Flaw selection is also very important for efficiency, especially when considering negative flaws which can conflict with other flaws.

Conflicts between flaws occur when two flaws of opposite sign target the same element of the partial plan. This can happen, for example, if an orphan flaw needs to remove a step needed by a subgoal or when a threat resolver tries to add a promoting link against an alternative. The use of side effects will prevent most of these occurrences in the agenda but a base ordering will increase the general efficiency of the algorithm.

Based on the figure 4, we define a base ordering of flaws by type. This order takes into account the number of flaw types affected by causal side effects.

1. **Alternatives** will cut causal links that have a better provider. It is necessary to identify them early since they will add at least another subgoal to be fixed as a related flaw.
2. **Subgoals** are the flaws that cause most of the branching factor in POP algorithms. This is why we need to make sure that all open conditions are fixed before proceeding on finer refinements.
3. **Orphans** remove unneeded branches of the plan. However, these branches can be found out to be necessary for the plan in order to meet a subgoal. Since a branch can contain numerous actions, it is preferable to let the orphan in the plan until they are no longer needed. Also, threats concerning orphans are invalidated if the orphan is resolved first.
4. **Threats** occur quite often in the computation. Searching and solving them is computationally expensive since they need to check if there are no paths that fix the flaw already. Numerous threats are generated without the need of resolver application [25]. That is why we prioritize all related subgoals and orphans before threats because they can add causal links or remove threatening actions that will fix the threat.

Resolvers need to be ordered as well, especially for the subgoal flaws. Ordering resolvers for a subgoal is the same operation as choosing a provider. Therefore, the problem becomes "how to rank operators?". The most relevant information on an operator is its usefulness and hurtfulness. These indicate how much an operator will help and how much it may cause branching after selection.
Definition 13 (Degree of an operator)**.** *Degrees are a measurement of the usefulness of an operator. Such notion is derived from the incoming and outgoing degrees of a node in a graph.*

We note $p.d^+(o)$ and $p.d^-(o)$ respectively the outgoing and incoming degrees of an operator in a plan p. These represent the number of causal links that goes out or toward the operator. We call proper degree of an operator $o.d^+ = |eff(o)|$ and $o.d^- = |pre(o)|$ the number of preconditions and effects that reflect its intrinsic usefulness.

There are several ways to use the degrees as indicators. *Utility value* increases with every d^+, since this reflects a positive participation in the plan. It decreases with every d^- since actions with higher incoming degrees are harder to satisfy. The utility value bounds are useful when selecting special operators. For example, a user-specified constraint could be laid upon an operator to ensure it is only selected as a last resort. This operator will be affected with the minimum utility value possible. More commonly, the maximum value is used for initial and goal steps to ensure their selection.

Our ranking mechanism includes several computation steps. The first step is the computation of the **base scores** noted $Z_0(o) = \langle o^+, o^- \rangle$. It is a tuple that contains two components: a positive score that acts as a participation measurement and a negative score that represents the dependencies of the operator. For each component of the score we consider a *subscores array* noted $S_z(o^\pm)$. We define them as follows:

- $S_z(o^+)$ containing only $\Delta^P.d^+(o)$, the positive degree of o in the domain proper plan. This will give a measurement of the predicted usefulness of the operator.
- $S_z(o^-)$ containing the following subscores:
  1. $o.d^-$ the proper negative degree of o. Having more preconditions can lead to a potentially higher need for subgoals.
  2. $\sum_{c \in C_{\Delta^P}(o)} |c|$ with $C_{\Delta^P}(o)$ being the set of cycles where o participates in the domain proper plan. If an action is co-dependent (cf. section 2.1) it may lead to a deadend as its addition will cause the formation of a cycle.
  3. $|SC(o)|$ is the number of self-cycle (0 or 1) o participates in. This is usually symptomatic of a *toxic operator* (cf. definition 15). Having an operator behaving this way can lead to backtracking because of operator instantiation.
  4. $\left|pre(o) \setminus \Delta^P.L^-(o)\right|$ with $\Delta^P.L^-(o)$ being the set of incoming edges of o in the domain proper plan. This represents the number of open conditions. This is symptomatic of action that can't be satisfied without a compliant initial step.

A parameter is associated to each subscore. It is noted $P_n^\pm$ with n being the index of the subgoal in the list. The final formula for the score is then defined as: $o^\pm = \sum_{n=1}^{|S_z(o^\pm)|} P_n^\pm S_z^n(o^\pm)$.

Once this score is computed, the ranking mechanism starts the second phase, which computes the **realization scores**. These scores are potential bonuses given once the problem is known. It first searches the *inapplicable operators* that are all operators in the domain proper plan that have a precondition that isn't satisfied with a causal link. Then it searches the *eager operators* that provide fluents with no corresponding causal link (as they are not needed). These operators are stored in relation with their inapplicable or eager fluents.

The third phase starts with the beginning of the solving algorithm, once the problem has been provided. It computes the *effective realization scores* based on the initial and goal steps. It will add P_1^+ to o^+ for each realized eager links (if the goal contains the related fluent) and subtracts P_4^- from o^- for each inapplicable preconditions realized by the initial step.

Last, the **final score** of each operator o, noted r_o, is computed from positive and negative scores using the following formula: $r_o = o^+ \alpha^{-o^-}$. This respects the criteria of having a bound for the *utility value* as it ensures that it remains positive with 0 as a minimum bound and $+\infty$ for a maximum. The initial and goal steps have their utility values set to the upper bound in order to ensure their selections over other steps.

Choosing to compute the resolver selection at operator level has some positive consequences on the performances. Indeed, this computation is much lighter than approaches with heuristics on plan space [15] as it reduces the overhead caused by real time computation of heuristics on complex data. In order to reduce this overhead more, the algorithm sorts the providing associative array in order to easily retrieve the best operator for each fluent. This means that the evaluation of the heuristic is done only once for each operator. This reduces the overhead and allows for faster results on smaller plans.

2.4 LOLLIPOP Algorithm

The LOLLIPOP algorithm uses the same refinement algorithm as described in algorithm 1. The differences reside in the changes made on the behaviour of resolvers and side effects. In line 10 of algorithm 1, LOLLIPOP algorithm applies negative resolvers if the selected flaw is negative. In line 11, it searches for both signs of side effects. Another change resides in the initialization of the solving mechanism and the domain as detailed in algorithm 4. This algorithm contains several parts. First, the DOMAININIT function corresponds to the code computed during the domain compilation time. It will prepare the rankings, the proper plan and its caching mechanisms. It will also use strongly connected component detection algorithm to detect cycles. These cycles are used during the base score computation (line 10). We add a detection of illegal fluents and operators in our domain initialization (line 5). Illegal operators are either inconsistent or toxic.

Algorithm 4 LOLLIPOP initialization mechanisms

```
 1 function DOMAININIT(Operators $O$)
 2     ProperPlan $P$
 3     Ranking $R$
 4     for all Operator $o \in O$ do
 5         if ISILLEGAL($o$) then                    Remove toxic and useless fluents
 6             $O \leftarrow O \setminus \{o\}$                     If entirely toxic or useless
 7             continue
 8         $P$.ADDVERTEX($o$)                       Add, cache and bind all operators
 9     Cycles $C \leftarrow$ STRONGLYCONNECTEDCOMPONENT($P$) Using DFS
10     $R.Z \leftarrow$ BASESCORES($O$, $P$)
11     $R.I \leftarrow$ INAPPLICABLES($P$)
12     $R.E \leftarrow$ EAGERS($P$)
13 function LOLLIPOPINIT(Problem $\Pi$)
14     REALIZE($\Pi.\Delta.R$, $\Pi$)                              Realize the scores
15     CACHE($\Pi.\Delta.P$, $\Pi.I$)                Cache initial step in providing ...
16     CACHE($\Pi.\Delta.P$, $\Pi.G$)                         ... as well as goal step
17     SORT($\Pi.\Delta.P.providing$, $\Pi.\Delta.R$)              Sort the providing map
18     if $\Pi.p.L = \emptyset$ then
19         SAFE($\Pi$)     Computing the safe proper plan if the plan is empty
20     POPULATE($a$, $\Pi$)                        populate agenda with first flaws
21 function POPULATE(Agenda $a$, Problem $\Pi$)
22     for all Update $u \in \Pi.U$ do               Updates due to online planning
23         Fluents $F \leftarrow eff(u.new) \setminus eff(u.old)$                Added effects
24         for all Fluent $f \in F$ do
25             for all Operator $o \in$ BETTER($\Pi.\Delta.P.providing$, $f$, $o$) do
26                 for all Link $l \in \Pi.P.L^+(o)$ do
27                     if $f \in l$ then                    With $l_\leftarrow$ the target of $l$
28                         ADDALTERNATIVE($a$, $f$, $o$, $l_\leftarrow$, $\Pi$)
29         $F \leftarrow eff(u.old) \setminus eff(u.new)$                 Removed effects
30         for all Fluent $f \in F$ do
31             for all Link $l \in \Pi.P.L^+(u.new)$ do
32                 if ISLIAR($l$) then
33                     $\Pi.L \leftarrow \Pi.L \setminus \{l\}$
34                     ADDORPHANS($a$, $u.new$, $\Pi$)
35         ...  Same with removed preconditions and incomming liar links
36     for all Operator $o \in \Pi.p.S$ do
37         ADDSUBGOALS($a$, $o$, $\Pi$)
38         ADDTHREATS($a$, $o$, $\Pi$)
```

Definition 14 (Inconsistent operators)**.** *An operator a is contradictory iff* $\exists f \{f, \neg f\} \in eff(o) \vee \{f, \neg f\} \in pre(o)$

Definition 15 (Toxic operators)**.** *Toxic operators have effects that are already in their preconditions or empty effects. An operator o is toxic iff* $pre(o) \cap eff(o) \neq \emptyset \vee eff(o) = \emptyset$

Toxic actions can damage a plan as well as make the execution of POP algorithm longer than necessary. This is fixed by removing the toxic fluents ($pre(a) \not\subseteq eff(a)$) and by updating the effects with

$eff(a) = eff(a) \setminus pre(a)$. If the effects become empty, the operator is removed from the domain.

The LOLLIPOPINIT function is executed during the initialization of the solving algorithm. We start by realizing the scores, then we add the initial and goal steps in the providing map by caching them. Once the ranking mechanism is ready, we sort the providing map. With the ordered providing map, the algorithm runs the fast generation of the safe proper plan for the problem's goal.

The last part of this initialization (line 20) is the agenda population that is detailed in the POPULATE function. During this step, we perform a search of alternatives based on the list of updated fluents. Online updates can make the plan outdated relative to the domain. This forms liar links :

Definition 16 (Liar links)**.** *A liar link is a link that doesn't hold a fluent in the preconditions or effect of its source and target. We note:*

$$a_i \xrightarrow{f} a_j | f \notin eff(a_i) \cap pre(a_j)$$

A liar link can be created by the removal of an effect or preconditions during online updates (with the causal link still remaining).

We call lies the fluents that are held by links without being in the connected operators. To resolve the problem, we remove all lies. We delete the link altogether if it doesn't bear any fluent as a result of this operation. This removal triggers the addition of orphan flaws as side effects.

While the list of updated operators is crucial for solving online planning problems, a complementary mechanism is used in order to ensure that LOLLIPOP is complete. This mechanism is explained in lemma 4.

3 Theoretical Analysis

As proven in [9], the classical POP algorithm is *sound* and *complete*.

First, we define some additional properties of partial plans. The following properties are taken from the original proof. We present them again for convenience.

Definition 17 (Full Support)**.** *A partial plan p is fully supported if each of its steps $o \in p.S$ is fully supported. A step is fully supported if each of its preconditions $f \in pre(o)$ is supported. A precondition is fully supported if it exists a causal link l that provides it. We note:*

$$fs(p) \equiv \begin{array}{l} \forall o \in p.S\ \forall f \in pre(o)\ \exists l \in p.L^-(o) : \\ (f \in l \wedge \nexists t \in p.S(l_{\rightarrow} \succ t \succ o \wedge \neg f \in eff(t))) \end{array}$$

with $p.L^-(o)$ being the incoming causal links of o in p and $l_{\rightarrow}$ being the source of the link.

Definition 18 (Partial Plan Validity)**.** *A partial plan is a* ***valid solution*** *of a problem Π iff it is* fully supported *and* contains no cycles. *The validity of p regarding a problem Π is noted $p \models \Pi \equiv fs(p) \wedge (C(p) = \emptyset)$ with $C(p)$ being the set of cycles in p.*

3.1 Proof of Soundness

In order to prove that this property apply to LOLLIPOP, we need to introduce some hypothesis:

- operators updated by online planning are known.
- user provided steps are known.
- user provided plans don't contain illegal artifacts. This includes toxic or inconsistent actions, lying links and cycles.

Based on the definition 18 we state that:

$$\left(\begin{array}{l} \forall pre \in pre(\Pi.G) : \\ fs(pre) \wedge \begin{array}{l} \forall o \in \Pi.L^-_{\rightarrow}(\Pi.G)\ \forall pre' \in pre(o) : \\ (fs(pre') \wedge C_p(o) = \emptyset) \end{array} \end{array} \right) \implies p \models \Pi \tag{1}$$

where $\Pi.L^-_{\rightarrow}(\Pi.G)$ is the set of direct antecedents of $\Pi.G$ and $C_p(o)$ is the set of fluents containing o in p.

This means that p is a solution if all preconditions of G are satisfied. We can satisfy these preconditions using operators iff their preconditions are all satisfied and if there is no other operator that threatens their supporting links.

First, we need to prove that equation (1) holds on LOLLIPOP initialization. We use our hypothesis to rule out the case when the input plan is invalid. The algorithm 3 will only solve open conditions in the same way subgoals do it. Therefore, safe proper plans are valid input plans.

Since the soundness is proven for regular refinements and flaw selection, we need to consider the effects of the added mechanisms of LOLLIPOP. The newly introduced refinements are negative, they don't add new links:

$$\forall f \in F(p)\ \forall r \in r(f) : C_p(f.n) = C_{f(p)}(f.n) \tag{2}$$

with $F(p)$ being the set of flaws in p, $r(f)$ being the set of resolvers of f, $f.n$ being the needer of the flaw and $f(p)$ being the resulting partial plan after application of the flaw. Said otherwise, an iteration of LOLLIPOP won't add cycles inside a partial plan.

The orphan flaw targets steps that have no path to the goal and therefore can't add new open conditions or threats. The alternative targets existing causal links. Removing a causal link in a plan breaks the full support of the target step. This is why an alternative will always insert a subgoal in the agenda corresponding to the target of the removed causal link. Invalidating side effects also don't affect the soundness of the algorithm since the removed flaws are already solved. This makes:

$$\forall f \in F^-(p) : fs(p) \implies fs(f(p)) \tag{3}$$

with $F^-(p)$ being the set of negative flaws in the plan p. This means that negative flaws don't compromise the full support of the plan.

Equations (2) and (3) lead to equation (1) being valid after the execution of LOLLIPOP. The algorithm is, therefore, sound.

3.2 Proof of Completeness

The soundness proof shows that LOLLIPOP's refinements don't affect the support of plans in term of validity. It was proven that POP is complete. There are several cases to explore in order to transpose the property to LOLLIPOP:

Lemma 1 (Conservation of Validity)**.** *If the input plan is a valid solution, LOLLIPOP returns a valid solution.*

Proof. With equations (2) and (3) and the proof of soundness, the conservation of validity is already proven. □

Lemma 2 (Reaching Validity with incomplete partial plans)**.** *If the input plan is incomplete, LOLLIPOP returns a valid solution.*

Proof. Since POP is complete and the equation (3) proves the conservation of support by LOLLIPOP, then the algorithm will return a valid solution if the provided plan is an incomplete plan and the problem is solvable. □

Lemma 3 (Reaching Validity with empty partial plans)**.** *If the input plan is empty, LOLLIPOP returns a valid solution.*

Proof. This is proven using lemma 2 and POP's completeness. However, we want to add a trivial case to the proof: $pre(G) = \emptyset$. In this case the line 4 of the algorithm 1 will return a valid plan. □

Lemma 4 (Reaching Validity with a dead-end partial plan)**.** *If the input plan is in a dead-end, LOLLIPOP returns a valid solution.*

Proof. Using input plans that can be in a undermined state is not covered by the original proof. The problem lies in the existing steps in the input plan. However, using our hypothesis we can add a failure mechanism that makes LOLLIPOP complete. On failure, the needer of the last flaw is deleted if it wasn't added by LOLLIPOP. User defined steps are deleted until the input plan acts like an empty plan. Each deletion will cause corresponding subgoals to be added to the agenda. In this case, the backtracking is preserved and all possibilities are explored as in POP. □

As all cases are covered, these proofs show that LOLLIPOP is complete.

4 Experimental Results

The experimental results were focused around the properties of the algorithm for online planning. Since classical POP is unable to perform online planning, we tested our algorithm in relation to the time taken for solving the problem for the first time. We profiled the algorithm on a benchmark problem containing each of the possible issues described previously.

Table 2. Average times of 1.000 executions on the problem. First column is for a simple run on the problem. Second and third columns are experiments with one and two changes in the plan after a first solving.

Experiment	*Single*	*Online 1*	*Online 2*
Time (ms)	0.86937	0.38754	0.48123

The measurements exposed in table X was made with an Intel® Core™ i7-4720HQ with a 2.60GHz clock. Only one core was used for the solving. The same experiment done only with the measurement code gave a result of $70ns$ of error. We can see an increase of performance in the online runs because of the way they are conducted by LOLLIPOP.

In figure 5, we expose the planning domain used for the experiments. During the domain initialization the action u and v are eliminated from the domain since they serve no purpose in the solving process. The action x is stripped of its negative effect -2 because it is inconsistent with the effect 2. First the domain compilation will

```
[DEBUG]  A(=[4]) is useless ! Getting rid of it.
[DEBUG]  A(=[4]+[4]) is entirely toxic ! Getting
    rid of it.
```

#TODO Finalize

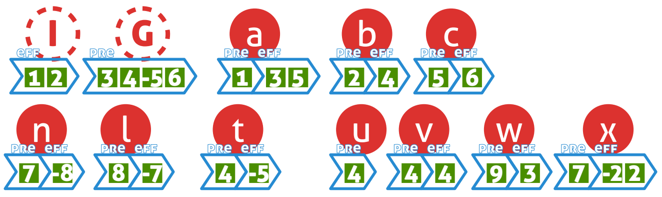


Figure 5. Domain used to compute the results. First line is the initial and goal step along with the useful actions. Second line contains a threatening action t, two co-dependent actions n and l, a useless action u, a toxic action v, a deadend action w and an inconsistent action x

Conclusion

In this paper, we present a set of new mechanisms to perform online partial order planning. These mechanisms allow negative refinements to be made upon existing plans in order to improve their quality. This allows LOLLIPOP to use pre-calculated plans in order to improve the resulting plan. This input can be used in online planning and make LOLLIPOP suitable for use in dynamic environments often encountered in robotic applications. We proved that LOLLIPOP is sound and complete like POP.

For future works we plan to improve the performance and expressiveness of our algorithm in order to represent complex real life problems encountered by dependent persons. We also aim to develop a soft solving mechanism that aims to give the best probable plan in case where the problem is unsolvable. Then this planner could be used for plan recognition using inverted planning.

References

[1] A. L. Blum and M. L. Furst, "Fast planning through planning graph analysis," *Artificial intelligence*, vol. 90, no. 1, pp. 281–300, 1997.

[2] J. Hoffmann, "FF: The fast-forward planning system," *AI magazine*, vol. 22, no. 3, p. 57, 2001.

[3] S. Richter, M. Westphal, and M. Helmert, "LAMA 2008 and 2011," in *International Planning Competition*, 2011, pp. 117–124.

[4] G. Röger, F. Pommerening, and J. Seipp, "Fast Downward Stone Soup," 2014.

[5] T. L. McCluskey and J. M. Porteous, "Engineering and compiling planning domain models to promote validity and efficiency," *Artificial Intelligence*, vol. 95, no. 1, pp. 1–65, 1997.

[6] C. Muise, S. A. McIlraith, and J. C. Beck, "Monitoring the execution of partial-order plans via regression," in *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, 2011, vol. 22, p. 1975.

[7] J. Kvarnström, "Planning for Loosely Coupled Agents Using Partial Order Forward-Chaining." in *ICAPS*, 2011.

[8] J. Benton, A. J. Coles, and A. Coles, "Temporal Planning with Preferences and Time-Dependent Continuous Costs." in *ICAPS*, 2012,

vol. 77, p. 78.

[9] J. S. Penberthy, D. S. Weld, and others, “UCPOP: A Sound, Complete, Partial Order Planner for ADL.” *Kr*, vol. 92, pp. 103–114, 1992.

[10] B. C. Gazen and C. A. Knoblock, “Combining the expressivity of UCPOP with the efficiency of Graphplan,” in *Recent Advances in AI Planning*, Springer, 1997, pp. 221–233.

[11] X. Nguyen and S. Kambhampati, “Reviving partial order planning,” in *IJCAI*, 2001, vol. 1, pp. 459–464.

[12] H. akan L. Younes and R. G. Simmons, “VHPOP: Versatile heuristic partial order planner,” *Journal of Artificial Intelligence Research*, pp. 405–430, 2003.

[13] A. J. Coles, A. Coles, M. Fox, and D. Long, “Forward-Chaining Partial-Order Planning.” in *ICAPS*, 2010, pp. 42–49.

[14] O. Sapena, E. Onaindia, and A. Torreno, “Combining heuristics to accelerate forward partial-order planning,” *Constraint Satisfaction Techniques for Planning and Scheduling*, p. 25, 2014.

[15] S. Shekhar and D. Khemani, “Learning and Tuning Meta-heuristics in Plan Space Planning,” *arXiv preprint arXiv:1601.07483*, 2016.

[16] J. L. Ambite and C. A. Knoblock, “Planning by Rewriting: Efficiently Generating High-Quality Plans.” DTIC Document, 1997.

[17] T. A. Estlin and R. J. Mooney, “Learning to improve both efficicency and quality of planning,” in *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 1997, pp. 1227–1233.

[18] M. Ramirez and H. Geffner, “Plan recognition as planning,” in *Proceedings of the 21st international joint conference on Artifical intelligence. Morgan Kaufmann Publishers Inc*, 2009, pp. 1778–1783.

[19] R. Van Der Krogt and M. De Weerdt, “Plan Repair as an Extension of Planning.” in *ICAPS*, 2005, vol. 5, pp. 161–170.

[20] M. Göbelbecker, T. Keller, P. Eyerich, M. Brenner, and B. Nebel, “Coming Up With Good Excuses: What to do When no Plan Can be Found.” in *ICAPS*, 2010, pp. 81–88.

[21] M. Ghallab, D. Nau, and P. Traverso, *Automated planning: theory & practice*. Elsevier, 2004.

[22] L. Sebastia, E. Onaindia, and E. Marzal, “A Graph-based Approach for POCL Planning,” in *ECAI*, 2000, pp. 531–535.

[23] M. Helmert, “The Fast Downward Planning System.” *J. Artif. Intell. Res.(JAIR)*, vol. 26, pp. 191–246, 2006.

[24] D. E. Smith and M. A. Peot, “Postponing threats in partial-order planning,” in *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 1993, pp. 500–506.

[25] M. A. Peot and D. E. Smith, “Threat-removal strategies for partial-order planning,” in *AAAI*, 1993, vol. 93, pp. 492–499.

[26] S. Kambhampati, “Design Tradeoffs in Partial Order (Plan space) Planning.” in *AIPS*, 1994, pp. 92–97.