LOLLIPOP: Online partial order planning using an operator graph and negative refinements

Antoine Gréa and Samir Aknine and Laëtitia Matignon
Université de Lyon, CNRS, Université Claude Bernard Lyon 1 , LIRIS, UMR5205, F-69622, France
'firstname.familyname@univ-lyon1.fr'

***Abstract*—In recent years, the field of automatic planning has mainly focused on performances and advances in state space planning to improve scalability. That orientation shadows other less efficient ways of planning like Partial Order 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 an operator graph from the domain compilation that gives information on the possible behavior of operators. A safe operator graph 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. But, 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 as explained in [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 import 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. Yet, we clearly note that only a few papers lay the emphasis upon plan quality using POP [16].

This work was supported by the ARC2 (Academic Research Comunity) of the region Rhone-Alpes.

The present paper lays the base for a project that aims to assist dependent persons in accomplishing 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 [17]. This requires performing 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.

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 repairing, as for instance in [18]. Usually, POP algorithms take a problem as an input and use a loop or a recursive function to refine the plan into a solution. Using existing partial plans as input causes multiples side effects if said plan is suboptimal. This problem was already explored as of LGP-adapt [19] that explains how re-using a partial plan often implies replanning parts of the plan. We use the refining and least commitment abilities of POP to improve online performances and quality. In order to achieve this, we start by computing an *operator graph* that is computed offline with the input of the domain. Operator graph definition and generation are given in section II-A. 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 II-B of our paper on plan quality and negative refinements. We 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 II-C. All these mechanisms are part of our aLternative Optimization with partiaL pLan Injection Partial Ordered Planning (LOLLIPOP) algorithm presented in detail in section II-D. We prove that the use of these new mechanisms keeps LOLLIPOP sound and complete in section III. Experimental results and benchmarks are presented in the section IV of this paper.

I. Partial Order Planning Preliminaries

A. Basic Definitions

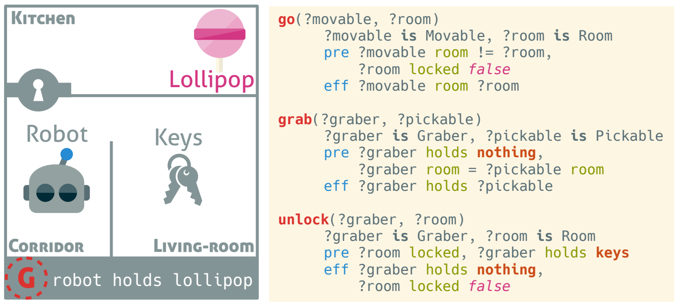
Planning systems need a representation of its domains and problems. Our framework is based on a classical domain definition inspired by [20].

Definition 1 (Problem)**.** *The planning problem is defined as a tuple $\mathcal{P} = \langle \mathcal{D}, I, G, \pi \rangle$ where*

- *$\mathcal{D}$ is a planning domain that is a set of **operators** with preconditions and effects,*
- *I is the **initial state**,*
- *G is the **goal**,*
- *π is a given **partial plan**.*

The framework uses the *closed world assumption* in which all predicates and relations 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 2 (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**.*

A specificity of our definition is that the ordering constraints are part of causal links. When an ordering constraint doesn't have a cause we use a **bare causal links** (mostly used for threats). 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.

Partial Order Planning fixes flaws in a partial plan 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 3 (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 a possible **provider** of the fluent s. This fluent is called* open condition *or **proper fluent** of the subgoal.*

Definition 4 (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 5 (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, the algorithm needs to take into account **side effects** of that application.

Definition 6 (Side effects)**.** *Flaws that arise because of the application of a resolver on the partial plan are called causal side effects or* related flaws.

We can derive this definition for subgoals and threats:

- **Related Subgoals** are all the new open conditions inserted by new steps.
- **Related Threats** are the causal links threatened by the insertion of a new step or deletion of a causal link.

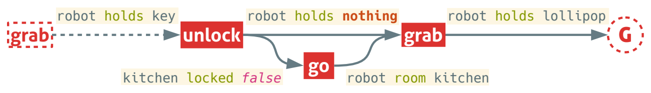


Figure 2. Example of a partial plan occurring during the computation of POP on the previous example domain illustrated by figure 1. Full arrows are existing causal links. The dotted arrow and operator depict the current resolver.

In the partial plan presented in figure 2, we consider a resolver providing the fluent `robot holds key`. It introduces the open conditions `robot holds nothing`, `key room _room`, `robot room _room` since it just added an instantiation of the `grab` operator in the partial plan. This triggers related subgoals to satisfy the new preconditions. 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`.

B. 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.

Algorithm 1 Classical Partial Order Planning

```
1  function POP(Queue of Flaws $a$, Problem $\mathcal{P}$)
2      POPULATE($a$, $\mathcal{P}$)                      Populate agenda only on first call
3      if $G = \emptyset$ then                 Goal is empty, default solution is provided
4          $L \leftarrow (I \rightarrow G)$
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($f$, $\mathcal{P}$)
9      for all $r \in R$ do                    Non deterministic choice operator
10         APPLY($r$, $\pi$)                           Apply resolver to partial plan
11         SideEffects $s \leftarrow$ SIDEEFFECT($r$)
12         APPLY($s$, $a$)                             Side effects of the resolver
13         if POP($a$, $\mathcal{P}$) = Success then           Refining recursively
14             return Success
15         REVERT($r$, $\pi$)                     Failure, undo resolver insertion
16         REVERT($s$, $a$)               Failure, undo side effects application
17     REVERT($a$, $f$)                            Add back the unresolved flaw
18     return Failure         Revert to last non deterministic choice of resolver
```

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 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. To explain these limitations, we use the example described in figure 1 where a robot must fetch a lollipop in a locked room. Even if this problem is solvable, 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 `kitchen` but that requires a *jackhammer*, the algorithm will need to meet more preconditions and will produce a partial plan of poor quality. The effects could be worse if obtaining a jackhammer would require many steps (for example building it). The other limitation arises when the plan has been modified. This can occur with online planning applications. When POP is modified to use input plans [22], these partial plans are carefully designed to fit 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.

II. LOLLIPOP's Approach

In this section, operator graph 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 operator graph are finally detailed.

A. Operator Graph Generation

One of the main contributions of the present paper is our use of the concept of *operator graph*. First of all, we define this notion.

Definition 7 (Operator Graph)**.** *An operator graph O^{Π} 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 there exists at least one 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 a heuristic in [4]. Causal graphs have fluents as their nodes and operators as their edges. Operator graphs are the opposite: an *operator dependency graph* for a set of actions. A similar structure was used in [23] that builds the operator dependency graph of goals and uses precondition nodes instead of labels. 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 operator graph, 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 to drive resolver selection (as detailed in section II-C). A **needing map** is also built but is only used for operator graph generation. We note $\mathcal{D}^{\Pi}$ the operator graph built with the set of operators in the domain $\mathcal{D}$. In the figure 3, we illustrate the application of this mechanism on our example from figure 1. Continuous lines correspond to the *domain operator graph* computed during domain compilation time.

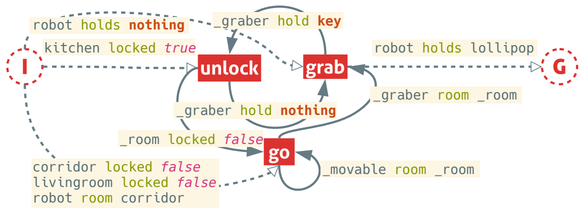


Figure 3. Diagram of the operator graph of example domain. Full arrows represent the domain operator graph and dotted arrows the dependencies added to inject the initial and goal steps.

The generation of the operator graph is detailed in algorithm 2. It explores the operators space and builds a providing and a 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 to add the causal links to the operator graph.

To apply the notion of operator graphs to planning problems, we just need to add the initial and goal steps to the operator graph. In figure 3, we depict this insertion with our previous example using dotted lines. However, since operator graphs may have cycles, they can't be used directly as input to POP algorithms

Algorithm 2 Operator graph generation and update algorithm

```
function BIND(Operator $o$)
    for all $pre \in pre(o)$ do
        if $pre \in providing$ then
            for all $\pi \in$ GET($providing$, $pre$) do
                Link $l \leftarrow$ GETEDGE($\pi$, $o$)        Create the link if needed
                $l \leftarrow l \cup \{pre\}$        Add the fluent as a cause
    ...        Same operation with needing and effects
```

to ease the initial backchaining. Moreover, the process of refining an operator graph 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 operator graphs differently. A similar notion was already presented as “basic plans” in [22]. These “basic” partial plans use a more complete but slower solution for the generation that ensures that each selected steps are *necessary* for the solution. In our case, we built a simpler solution that can solve some basic planning problems but that also make early assumptions (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. This algorithm is useful since it is specifically used on goals. The result is a valid partial plan that can be used as input to POP algorithms.

B. Negative Refinements and Plan Optimization

The 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 8 (Alternative)**.** *An alternative is a negative flaw that occurs when there is 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 II-C. It uses the incoming and outgoing degrees of the operator in the domain operator graph 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. 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 operator graph mechanism is guaranteed to only choose the best provider . Furthermore, subgoals won’t introduce new fixable alternative as they are guaranteed to select the best possible provider.

Definition 9 (Orphan)**.** *An orphan is a negative flaw that occurs when 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 $a_o \neq I \wedge a_o \neq G \wedge (d^+_\pi(a_o) = 0) \vee \forall l \in L^+_\pi(a_o), l = \emptyset$.*

With $d^+_\pi(a_o)$ being the *outgoing degree* of a_o in the directed graph formed by π and $L^+_\pi(a_o)$ being the set of *outgoing causal links* of a_o in π. 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 5).

Definition 10 (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 focus on its resolver by its rank (explained in section II-C) 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.

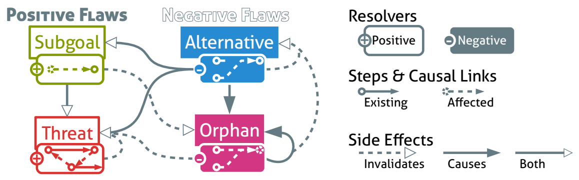


Figure 4. Schema representing flaws with their signs, resolvers and side effects relative to each other

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 6) with a notion of sign.

Definition 11 (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 extra 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 [24], a **invalidating side effect** is explained under the name of *DEnd* strategy. In classical POP, threats can disappear in some cases if other flaws were applied before them. For our mechanisms, we decide to gather under this notion every side effect 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 rigorous manner.

C. 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 [25]. Flaw selection is also 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 detection of side effects will prevent this problem but a base ordering will also 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.
2. **Subgoals** are the flaws that cause most of the branching factor in POP algorithms.
3. **Orphans** remove unneeded branches of the plan. Yet, these branches can be found out to be necessary for the plan to meet a subgoal.
4. **Threats** occur quite often in the computation. Many threats are generated without the need of resolver application [24].

Resolvers need to be ordered as well, especially for the subgoal flaws. The problem can be translated to "how to rank operators by provided fluents?". The most relevant information on an operator is its usefulness and hurtfulness. These two notions show how much an operator will help and how much it may cause branching after selection.

Definition 12 (Degree of an operator)**.** *Degrees are a measurement of the usefulness of an operator. Such a notion is derived from the incoming and outgoing degrees of a node in a graph.*

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

There are several ways to use the degrees as indicators. d^+ reflects a positive participation in the plan and d^- means that the operator is harder to satisfy. The higher limit of utility values is useful for initial and goal steps to ensure their selection over less critical operators.

Our ranking mechanism is based on scores noted $s^\pm(o)$. A score contains two components: a positive and negative subscore array that represent its usefulness and hurtfulness.

The first step is the computation of the **base scores**. They are computed according to the following:

- $s^+(o)$ contains only $d^\pm_{\mathcal{D}^\Pi}(o)$, the positive degree of o in the domain operator graph. This will give a measurement of the predicted usefulness of the operator.
- $s^-(o)$ containing the following subscores:
  1. $d^-(o)$ the proper negative degree of o. Having more preconditions can lead to a potentially higher need for subgoals.
  2. $\sum_{c \in C_o(\mathcal{D}^\Pi)} |c|$ with $C_o(\mathcal{D}^\Pi)$ being the set of cycles where o participates in the domain operator graph. If an action is co-dependent (cf. section II-A) it may lead to a dead-end as its addition will cause the formation of a cycle.
  3. $|C^s_o(\mathcal{D}^\Pi)|$ is the number of self-cycle (0 or 1) o participates in. This is usually symptomatic of a *toxic operator* (cf. definition 14). Having an operator behaving this way can lead to backtracking because of operator instantiation.
  4. $\left|pre(o) \setminus L^-_{\mathcal{D}^\Pi}(o)\right|$ with $L^-_{\mathcal{D}^\Pi}(o)$ being the set of incoming causal links of o in the domain operator graph. This represents the number of open conditions. This is symptomatic of action that can't be satisfied without a compliant initial step.

Once these subscores are 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 operator graph 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 increment $s^+_1(o)$ for each realized eager links (if the goal contains the related fluent) and decrement $s^-_4(o)$ for each inapplicable preconditions realized by the initial step.

Last, the **final score** of each operator o, noted $h(o)$, is computed from positive and negative scores using the following formula:

$$h(o) = \sum_{n=1}^{|s^\pm(o)|} \pm p^\pm_n s^\pm_n(o)$$

A parameterized coefficient is associated to each subscore. It is noted $p^\pm_n$ with n being the index of the subscore in the array $s^\pm$. 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 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 handles simpler data structures. In order to reduce this overhead more, the algorithm sorts the providing associative array to easily retrieve the best operator for each fluent.

D. The LOLLIPOP Algorithm

The LOLLIPOP algorithm uses the same refinement algorithm as described in algorithm 1. The differences reside in the changes made on the behavior 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 3. 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 operator graph, 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 11). We add a detection of illegal fluents and operators in our domain initialization (line 5). Illegal operators are either inconsistent or toxic.

Algorithm 3 LOLLIPOP initialization mechanisms

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

Definition 13 (Inconsistent operators)**.** *An operator a is contradictory iff $\exists f \{f, \neg f\} \in eff(o) \vee \{f, \neg f\} \in pre(o)$*

Definition 14 (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 operator graph for the problem's goal.

The last part of this initialization (line 21) 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 15 (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 to ensure that LOLLIPOP is complete. User provided plans have their steps tagged. If the failure has backtracked to a user-provided step, then it is removed and replaced by subgoals that represent each of its participation in the plan. This mechanism loops until every user provided steps have been removed.

III. Theoretical Analysis

As proven in [9], some POP algorithms can be *sound* and *complete*.

The following properties are inspired by the proof for UCPOP. We present them again for convenience.

Definition 16 (Full Support)**.** *A partial plan π is fully supported if each of its steps $o \in S$ 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 there exists a causal link l that provides it. We note:*

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

with $L^-_\pi(o)$ being the incoming causal links of o in π and $l_\rightarrow$ being the source of the link.

Definition 17 (Partial Plan Validity)**.** *A partial plan is a* ***valid solution*** *of a problem $\mathcal{P}$ iff it is* fully supported, contains no threats *and* contains no cycles. *The validity of π regarding a problem $\mathcal{P}$ is noted* $(\pi \models \mathcal{P}) \equiv [\Downarrow \pi \wedge (\mathcal{F}_t(\pi) = \emptyset) \wedge (C(\pi) = \emptyset)]$ *with $\mathcal{F}_t(\pi)$ being the set of threats present in π and $C(\pi)$ being the set of cycles in π.*

A. *Proof of Soundness*

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

1. operators updated by online planning are known.
2. user provided steps are known.
3. user provided plans don't contain illegal artifacts. This includes toxic or inconsistent actions, lying links and cycles (or the input is cleaned).

From definition 16 we deduce that $\Downarrow \pi \implies \mathcal{F}_s(\pi) = \emptyset$ with $\mathcal{F}_s(\pi)$ being the set of subgoals present in π. From definition 3 we can say that $\mathcal{F}_s(\pi) = \emptyset \implies \Downarrow \pi$. This means that full support of a plan is equivalent to the absence of subgoals : $\Downarrow \pi \equiv \mathcal{F}_s(\pi) = \emptyset$

Based on this and the fact that $\mathcal{F}^+(\pi) = \mathcal{F}_s(\pi) \cup \mathcal{F}_t(\pi)$ we can transform definition 17 into : $(\pi \models \mathcal{P}) \equiv [(\mathcal{F}^+(\pi) = \emptyset) \wedge (C(\pi) = \emptyset)]$

POP is guaranteed to return a result plan iff its flaw agenda is empty. Since the algorithm add all newly formed flaws in the agenda :

$$\forall \mathcal{P} : \mathcal{F}(lollipop(\mathcal{P})) = \emptyset \tag{1}$$

We only need to consider the effects of the added mechanisms of LOLLIPOP on cycles. The newly introduced refinements are negative, they don't add new links: $\forall f \in \mathcal{F}(\pi)\ \forall r \in r(f) : C_\pi(f.n) = C_{f(\pi)}(f.n)$ with $\mathcal{F}(\pi)$ being the set of flaws in π, $r(f)$ being the set of resolvers of f, $f.n$ being the needer of the flaw and $f(\pi)$ being the resulting partial plan after application of the flaw. Said otherwise, an iteration of LOLLIPOP doesn't introduce cycles inside a partial plan. Hypothesis 3 leads to :

$$\forall \mathcal{P} : C_{lollipop(\mathcal{P})} = \emptyset \tag{2}$$

Equations (1) and (2) lead to the property of soundness for LOLLIPOP.

B. *Proof of Completeness*

The soundness proof shows that LOLLIPOP's refinements don't affect the support of plans in term of validity. There are several cases to explore to prove completeness of LOLLIPOP.

1) Simple problem: Let's start with the simple case where there are no input partial plan. As previously shown in equation (1), LOLLIPOP will out plans that are flawless. In the loop in algorithm 1 at line 9 all resolvers are considered. Therefore, the worst complexity of LOLLIPOP is $\mathcal{O}(lollipop(a, \mathcal{P})) = |resolvers(f)| \times \mathcal{O}(lollipop(a \setminus \{f\}, \mathcal{P}))$ with f being a flaw chosen in the agenda. This leads to $\forall \mathcal{P} : \mathcal{O}(lollipop(a, \mathcal{P})) = \prod_{f \in a} |resolvers(f)|$.

We can state that $|resolvers(f)| \leq |S| + |\mathcal{D}| + 1$ since the number of resolver is at most the number of steps plus the number of operators plus the initial state. We can also say that $|a| = |\mathcal{F}(\pi)| \leq \sum_{o \in \mathcal{D}} d^-(o) + 3|S|$. This means that the size of the agenda is also finite since the number of subgoals is at most the number of preconditions in the plan, the number of threats, alternative and orphan is the number of steps. All this leads to :

$\forall \mathcal{P}$:
$\mathcal{O}(lollipop(a, \mathcal{P})) \leq (|S| + |\mathcal{D}| + 1) \left(\sum_{o \in \mathcal{D}} d^-(o) + 3|S|\right)$
This means that the complexity of the LOLLIPOP algorithm is strictly finite and therefore *LOLLIPOP converges*.

The previous analysis also shows that all resolvers get considered. This means that if a flaw is solvable, then all its solutions will be considered. Also since all possible flaws are treated that means that *if a flawless plan exists it will be considered*.

That isn't sufficient to prove completeness because of negative flaws. One can think that their presence will cause a destructive interference with POP's normal behavior. However, we can show that negative refinements have only an optimization effect since they can't fail and always generate their equivalence in positive flaws to be added to the agenda. Orphans remove only dead trees from the plan and alternative always add their counterpart subgoal forcing the verification of all alternate providers.

This proves completeness of LOLLIPOP in that case.

2) Online refinement and user input: Using the hypothesis 1 and 2 we can make sure that we only target updated steps. The problem lies in the existing steps in the input plan. Still, using our hypothesis, we 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.

Since all cases are covered, this proves the property of completeness for LOLLIPOP.

IV. Experimental Results

The experimental results focused on the properties of LOLLIPOP for online planning. We profiled the algorithm on a benchmark problem containing each of the possible issues described earlier.

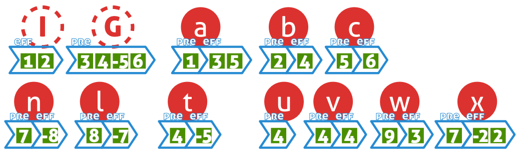


Figure 5. Domain used to compute the results. First line is the initial and goal step along with the useful actions. Second line contains a threatening action t, two co-dependent actions n and l, a useless action u, a toxic action v, a deadend action w and an inconsistent action x

In figure 5, we expose the planning domain used for the experiments. During the domain initialization, the actions u

Table I
AVERAGE TIMES OF 1.000 EXECUTIONS OF PREVIOUS TIME MARKERS.

Experiment	*1*	*2*	*3*
Time (ms)	0.86937	0.38754	0.48123

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 because it is inconsistent with the effect 2.

To start, LOLLIPOP computes a safe operator graph (full lines in figure 6). Since this plan is nearly complete the main refining function (time markers **1**) receives an agenda with only a few flaws remaining. LOLLIPOP selects a causal link from a to satisfy the open condition of the goal step.

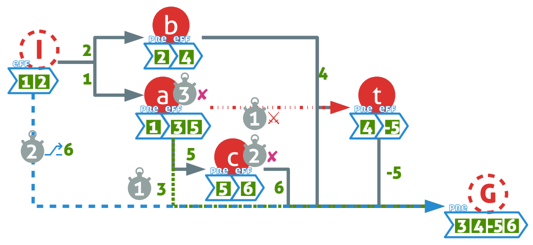


Figure 6. In full lines the initial safe operator graph. In thin, sparse and irregularly dotted lines a subgoal, alternative and threat caused causal link.

Once the threat between a and t is resolved the second threat is invalidated. On a second execution (time markers **2**), the domain changes for online planning with 6 added to the initial step. This adds as flaw an alternative on the link from c to the goal step. A subgoal is added that links the initial and goal step for this fluent. An orphan flaw is also added that removes c from the plan. Another iteration takes place as 3 is removed from the goal (time markers **3**). This causes the link from a to be cut since it became a liar link. This adds a as an orphan that gets removed from the plan even if it was hanging by the bare link to t.

The measurements exposed in table I were made with one core of an Intel® Core™ i7-4720HQ with a 2.60GHz clock. We can see an increase of performance in the online runs because of the way they are conducted by LOLLIPOP.

CONCLUSION

In this paper, we have presented a set of new mechanisms to perform online partial order planning. These mechanisms allow negative refinements to be made upon existing plans to improve their quality. This allows LOLLIPOP to use pre-calculated plans 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.

For future works, we plan to improve the performance and expressiveness of our algorithm 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 incomplete plan in the cases where the problem is unsolvable. Then this planner could be used for plan recognition using inverted planning. For future works, we plan to improve the performance and expressiveness of our algorithm 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 incomplete plan in the cases 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] M. Helmert, G. Röger, and E. Karpas, "Fast downward stone soup: A baseline for building planner portfolios," in *ICAPS 2011 Workshop on Planning and Learning*, 2011, pp. 28–35.

[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," *JAIR*, pp. 405–430, 2003.

[13] A. Coles, A. Coles, M. Fox, and D. Long, "popf2 : a forward-chaining partial order planner," *IPC*, p. 65, 2011.

[14] O. Sapena, E. Onaindia, and A. Torreno, "Combining heuristics to accelerate forward partial-order planning," *CSTPS*, 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] M. Ramirez and H. Geffner, "Plan recognition as planning," in *IJCAI*, 2009, pp. 1778–1783.

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

[19] D. Borrajo, "Multi-agent planning by plan reuse," in *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, 2013, pp. 1141–1142.

[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] D. E. Smith and M. A. Peot, "Postponing threats in partial-order planning," in *NCAI*, 1993, pp. 500–506.

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

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