LOLLIPOP: aLternative Optimization with partiaL pLan Injection Partial Ordered Planning

Paper ID#42

Abstract. The abstract! Infinite source of drama, confusion and *shameless* paper promotion. Stay tuned for more!

 # Everything that starts with a # is purely commentary and WILL be removed

Introduction

PSP

Until the end of the 90s, plan space planning was generally preferred by the automated planning community. Its early commitment, expressivity, and flexibility were clear advantages over state space planning. SSP? However, more recently, drastic improvements in state search planning was made possible by advanced and efficient heuristics. This allowed those planners to scale up more ~~easily~~ 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. Some of these aspects can be more easily addressed in Partial Order Planning (POP). For example POP, has can take advantage of least commitment [5] that offers more flexibility with a final plan that describes only the necessary order of the actions considered without forcing a particular sequence. POP has also been proven to be well suited for multi-agent planning [6] and temporal planning [7] . These advantages made UCPOP [8] one of the best POP planner of its time with works made to port some of its characteristics into state-based planning [9].

Related works already tried to explore new ideas to make POP into an attractive alternative to regular state-based planners like the appropriately named "Reviving partial order planning" [10] and VHPOP [11]. More recent efforts [12], [13] 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 [14] 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 [15], [16].

This work is the base of our project For ~~of~~ an intelligent robotic system that can use plan and goal inference to help dependent persons to accomplish tasks. This project is based on the works of Ramirez et al. [17] on inverted planning for plan inference. That is what we need to improve POP with better refining techniques and utility driven heuristics. Since we need to handle data derived from limited robotic sensors, we need a way for the planner to be able to be resilient to basic corruption on its input. Another aspect of this work lies in the fact that the final system will need to compute online planning with a feed of observational data. In order to achieve this we need a ~~base~~ planner that can:

- refine existing partial plans for online planning,
- be able to optimize a plan by removing unnecessary steps,
- and be able to select the best-suited action for providing each subgoal. Retain some performance on complex but medium sized problems.

The classical POP algorithm doesn't fit these criteria but can be enhanced to fit the first criteria. 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 simply expose the recursive function in order to be able to use our existing partial plan. This, however, causes multiples side effects if the plan is suboptimal.

Our view on the problem diverges from other works: Plan-Space Planning (PSP) is a very different approach that state space planning. It is usually more computationally expensive than modern state space planners but brings several advantages. We want to make the most of the ~~differences~~ of POP instead of trying to change its 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 ~~*domain*~~ *proper plan* that is computed offline with the input of the domain. The explanation of the way this notion is defined and used can be found in section 2.1 of the present paper.

Using existing partial plans as input leads to several issues, mainly with 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 optimise the plan: the alternative and the orphan flaws.

A side effect of negative flaws and resolvers is that they can interfere with each others and need guiding to cooperate into making use of least commitment and to participate in a better solution quality. That is the reason behind the section 2.3 of our work: goal and flaw selection that aims to reduce the branching factor of our algorithm.

#TODO To prove

new

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 leads to fewer iterations, a reduced branching factor and better quality than standard POP in section 3. Experimental results and benchmarks are presented and explained in the section 4 of this paper.

Before explaining our solution we need to detail the existing POP and its limitations.

1 Classical Partial Order Planning Framework

While needing expressivity and simplicity in our domain definition we also need speed and flexibility for online planning on robots. Our framework is inspired by existing multi-purpose semantic tools such as RDF Turtle [18] and has an expressivity similar to PDDL 3.1 with object-fluents support [19]. This particular type of domain description was chosen because we intend to extend works on soft solving in order to handle corrupted data better in future papers. The next definitions are based on the ones exposed in [20].

1.1 Basic Definitions

Every planning paradigm needs a way to represent its fluents and operators. Our planner is based on a rather classical domain definition with lifted operators and representing the fluents as propositional statements.

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 description in order to work. For this, 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 constant*** *disjoint from* C*,*
- I *is the* ***initial state****,*
- G *is the* ***goal****,*
- p *is a given* ***partial plan****.*

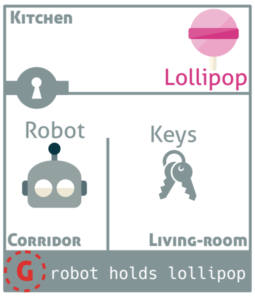
The framework uses the *closed world assumption* in which all predicates and properties that aren't defined in the initial step are assumed false or don't have a value.

We want to introduce a problem in figure 1 that we will use to exemplify the presented notion.

In order to simplify this framework we need to introduce some differences from the classical representation. For example, 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****.*

In classical representations, a set of *ordering constraints* is also added. We propose to factorize this notion as being part of the causal links which are always supported by an ordering constraint. The only case where bare ordering constraints are needed is in threats. We decided to represent them with "bare causal links". These are stored as causal links without bearing any fluents. This also eases implementation with the definition of the causal link giving only one graph of steps with a




```
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.

(possibly empty) list of fluents as a label as our main definition for a partial plan. 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 with a_j with a_i being *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. In this section, 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 intolerant to said effect. We say that a step* a_b *is threatening a causal link* $a_p \xrightarrow{t} a_n$ *iff* $\neg t \in eff(a_b) \wedge a_p \succ a_b \succ a_n \models L$ *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 target of the resolver,*
- f *being the considered fluent.*

For standard flaws, the resolvers are simple to find. For a *subgoal* the resolvers are a set of the potential causal links between a possible provider of the proper fluent and the needer. To solve a *threat* there is 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. It searches the related flaws of the elected resolver. These related flaws 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 effect 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 path of causal links that prevent the action to interfere with the threatened causal link.

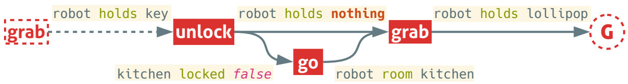


Figure 2. Example partial plan occurring during the computation of POP on our example domain of figure 1. The `grab` operator at the left is part of the currently considered resolver in POP's execution.

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. Each of these will trigger a related subgoal that will have this new `grab` operator as their 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`.

There is no need to search for ~~related flaws~~ when fixing a threat or when simply adding a causal link between existing steps.

1.2 Classical POP Algorithm

The classical POP algorithm is pretty straight forward: it starts with a simple partial plan 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 Π)
2    POPULATE(a, Π)                    ▷ Populate agenda only on first call
3    if Π.G = ∅ then                   ▷ Goal is empty, default solution is provided
4      Π.p.L ← (I → G)
5    if a = ∅ then
6      return Success                  ▷ Stop all recursion
7    Flaw f ← CHOOSE(a)                ▷ Non deterministic choice
8    Resolvers R ← RESOLVERS(f, Π)
9    for all r ∈ R do                  ▷ Non deterministic choice operator
10     APPLY(r, Π.p)                   ▷ Apply resolver to partial plan
11     SideEffects s ← SIDEEFFECT(r)
12     APPLY(s, a)                     ▷ Side effects of the resolver
13     if POP(a, Π) = Success then     ▷ Refining recursively
14       return Success
15     REVERT(r, Π.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
```

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 nonoptimal links if they don't fail.

To illustrate these limitations, we use the example described in figure 1 where a robot must fetch a lollipop in a locked room. This problem is quite easily solved by regular POP algorithms.

However, we can have some cases where small changes in POP's inputs can cause a lot of unnecessary back-trackings. For example, if we add a new action called `dig_throught_wall` that has as effect to be in the desired room but that requires a hammer, the algorithm will simply need more backtracking. The effects could be worse if obtaining the hammer would require numerous steps (for example needing to build it). This problem can be solved most of the time using simple flaw selection mechanisms. However, this was never applied in the context of POP. The other limitation arises when the plan has been modified. This can arise ~~since users are free to do arbitrary modifications~~ to ~~of the domain or the plan~~ in the context of ~~the~~ dynamical environments of online planning's application. Regular POP algorithms do not consider this issue as they do not take a partial plan as input. This can cause a variety of new problems that are related to planning corruption.

In order to address these issues, we present a set of new mechanisms.

2 Methods

2.1 Proper Plan Generation and Injection

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

This definition was inspired by the notion of domain causal graph as explained in [20] and originally used as heuristic in [22]. A variation of this notion was used in [23] that builds the operator dependency graph of goals and uses precondition nodes instead of labels. A proper plan is, therefore, an *operator dependency graph* for a set of actions. This structure is very useful for getting information on the *shape of a problem*. This shape ~~gives~~ leads to an intuition based on the potential usefulness or hurtfulness of operators. Cycles in this graph ~~contain~~ denotes information regarding the dependencies of operators. We call *co-dependent* several operators that form a cycle. If the cycle is made of only one operator (self-loops), 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 associative table is easy to update and the list of provider can be ~~arbitrarily~~ sorted in order to drive resolver selection. A **needing map** is also built but isn't used in further mechanisms.

In the figure 3 we illustrate the application of this mechanism on our previous example. Continuous lines are the base *domain proper plan*.

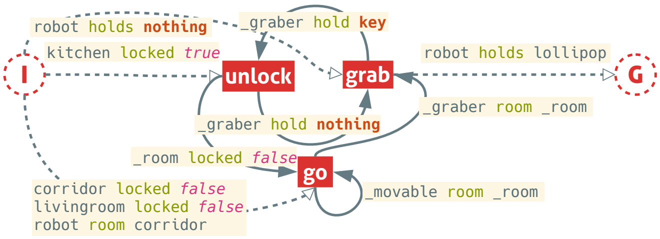


Figure 3. Diagram of the proper plan of example domain. In full arrows the proper plan as computed during domain compilation time and in dotted arrows the dependencies added to inject the initial and goal steps.

This particular proper plan, noted Δ^P, is built with the information present in the domain.

The generation of the proper plan is based upon the previous definition: It will explore the operators space and build a providing and needing map that gives the provided and needed fluents for each operator. Once done it will iterate on every precondition and search for a satisfying cause in order to add the causal link to the proper plan. The algorithm 2 details this procedure.

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 ∈ eff(o) do
        if eff ∈ providing then           ▷ providing is the providing map
            ADD(providing, eff, o)
    ...                  ▷ Same operation with needing and preconditions
function BIND(Operator o)
    for all pre ∈ pre(o) do
        if pre ∈ providing then
            for all p ∈ GET(providing, pre) do
                Link l ← GETEDGE(p, o)           ▷ Create the link if needed
                l ← l ∪ {pre}                   ▷ Add the fluent as a cause
    ...                       ▷ Same operation with needing and effects
```

Applying the notion of proper plan for problems only needs the initial and goal steps added in the proper plan. In figure 3 we illustrate this insertion with our previous example using dotted lines.

A derived notion can be built from proper plan. We can make a "safe" version of this algorithm. The aim is to do a simple and fast backward chaining algorithm that can build a proper plan that does not contain cycles.

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 of POP algorithms. The focus of this is that it is a simple backward search that is driven by the providing map and can, therefore, be tweaked with the powerful heuristics of state search plan.

+ Remove most subgoal

2.2 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. In our case, it is important to be able to *remove* part of the plan that isn't necessary for the solution.

Algorithm 3 Safe proper plan generation algorithm

```
1  function SAFE(Problem Π)
2      Stack open ← [Π.G]
3      Stack closed ← ∅
4      while open ≠ ∅ do
5          Operator o ← POP(open)                  ▷ Remove o from open
6          PUSH(closed, o)
7          for all pre ∈ pre(o) do
8              Operators p ← GETPROVIDING(π, pre)
9              if p = ∅ then
10                 Π.p.S ← Π.p.S \ {p}
11                 continue
12             Operator o′ ← GETFIRST(p)
13             if o′ ∈ closed then
14                 continue
15             if o′ ∉ Π.p.S then
16                 PUSH(open, o′)
17             Π.p.S ← Π.p.S ∪ {o′}
18             Link l ← GETEDGE(o′, o)         ▷ Create the link if needed
19             l ← l ∪ {pre}                  ▷ Add the fluent as a cause
```

Since we are given a partial plan that is quite complete, we need to add new flaws to optimize and fix this plan. These flaws are called *negative* since their resolvers differ from classical ones from their effects on the plan.

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 have a better* utility value *than* a_p.

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

⊖ (Rewriti

Finding an alternative to an operator is computationally expensive. The search needs to search 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 chose the best provider (algorithm 3 at line 12). + + as one subg.

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* $a_o \neq I \wedge a_o \neq G \wedge \left(p.d^+(a_o) = 0\right) \vee \forall l \in p.L^+(a_o), l = \emptyset$.

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

+ hanging

With the introduction of negative flaws comes the modification of resolvers to handle negative refinements.

We add onto the definition 6 :

Definition 11 (Signed Resolvers)**.** *A signed resolver is a resolver with a notion of sign. We add to the resolver tuple* s *as the sign of the resolver in* $\{+, -\}$.

An alternative notation for the signed resolver is inspired by the causal link notation with simply the sign underneath :

$$r = a_s \xrightarrow[+/-]{f} a_t$$

The previously defined negative flaws have all their associated negative resolvers.

The solution to an alternative is a negative refinement that simply remove the targeted causal link. We count on the fact that this will create a new subgoal that will prioritize its resolver by usefulness and then pick the most useful provider. is a self

The resolver for orphans is the negative refinement that is ~~only~~ meant to remove the targeted action and its incoming causal link while tagging its providers as potential orphans.

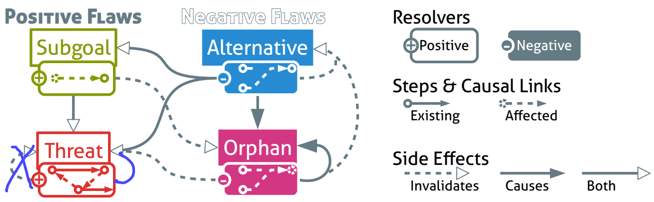


Figure 4. Schema representing flaws with their signs, resolvers and side effects relative to each other

The standard mechanism of side effects needs an upgrade since the new kind of flaws can easily interfere with one another. This is why we extend the 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 illustrate the extended notion of signed side effects of resolver application. 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. In fact, negative refinements will most likely cause an increase in subgoals or threats since they remove causal links or steps. This deletion of causal links and steps can cause orphan flaws that need to be identified for removal.

An *invalidating side effect* can be found in other works [24] with other names. In classical POP, it has been noticed that threats can disappear in some cases if subgoals or other threats were applied before them. In our formalism, we decide to gather under this notion every side effects that removes the need to consider a flaw. For example, orphans can be "saved" if a subgoal selects the orphan step. Alternatives can remove the need to compute further subgoal of a now orphan step as orphans simply remove the need to fix any flaws that concern the selected step.

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

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. Flaw selection is also very important for efficiency, especially when considering negative flaws which can enter into conflict with other flaws.

Flaws conflicts happen when two ~~negative flaws or~~ flaws of opposite sign target the same element of the partial plan. This can happen for example if an orphan removes a step needed by a subgoal or when a threat 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 create a base ordering of flaws by type. This order takes into account the number of flaw types affected by causal side effects. This will increase the number of side effects early on and, therefore, taking the most important decisions early on to reduce the branching factor.

Following this reasoning, we order the flaw types by number of different side effects in the figure 4 ('Both' links count double, self-side-effects count half) :

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. *(Score 5.5)*
2. **Subgoals** are the flaws that cause ~~the~~ most branching factor for POP algorithms. This is why we need to make sure that all open conditions are fixed before proceeding on finer refinements. *(Score 3.0)*
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. *(Score 2.5)* + removed threats
4. **Threats** occur quite often in the computation. They cost a lot of processing power since they need to check ~~is~~ there are no paths that fix the flaw already. Numerous threats are generated without need of intervention. 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. *(Score 0.5)*

Resolvers also need to be ordered, especially for the subgoal flaw. This ordering is the same as ranking operators and steps against one another. 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. Now the problem is how to measure these factors.

Definition 13 (Degree of an operator)**.** *Degrees are a measurement of the usefulness of an operator. The notion is derived from the incoming and outgoing degree of a node in a graph.*

We note $g.d^+(o)$ and $g.d^-(o)$ respectively the outgoing and incoming degree of an operator in a plan. 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 ~~potential usefulness of an operator based on its number~~ of preconditions and effects. that indicate its usefulness

There are several ways to use ~~the~~ degrees as indicators. The goal is to increase the *utility value* with every d^+, since this reflects a positive participation in the plan, and decreases it with every d^- since actions with higher incoming degrees are harder to satisfy. Utility values should belong to a bounded domain with a minimum and a maximum. The minimum value is used for actions that should be selected only as a last resort and the maximum value should ensure selection. Another criteria for the utility value is that its computation shouldn't be too computationally expansive.

Our ranking mechanism includes several computation phases. The first step is the computation of the **base scores** noted $Z_o = \langle o^+, o^- \rangle$. For each component of the score we consider a *sub score array* noted $S_z(o^\pm)$. We define them as follows :

+ explain[illegible]

- $S_z(o^+)$ containing only $\Delta^P.d^+(o)$, the positive degree of o in the domain proper plan.
- $S_z(o^-)$ containing the following subscores :
  1. $o.d^-$ the proper negative degree of o,
  2. $\sum_{c \in C(o)} |c|$ with $C(o)$ being the set of cycles o participates in,
  3. $|SC(o)|$ with $SC(o)$ is the set of self-cycle o participates in and
  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 proper plan of the domain.

#TODO simplify + examples

The computation affects a parameter for each subscore 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. This phase aims to compute the **realization scores** that are potential scores that are held until the third phase. 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 provides fluents with no corresponding causal link (as they are not needed). These operators are stored in associative arrays 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 step. It will add P_1^+ to o^+ for each realized eager operators (if the goal contains the related fluent) and subtract P_4^- from o^- for each inapplicable operators realized by the initial step.

From there, we have the **final scores** that are used in the ultimate phase. In this phase, the scores are combined into a single number in order to rank the operators. In order to respect the criteria of having a bounded value for the *utility value* we use the following formula : $r_o = o^+ \alpha^{-o^-}$. This ensures that the value is positive with 0 as a minimum bound and $+\infty$ for a maximum. The initial and goal steps have their utility value set to the upper bound in order to ensure their selection in priority.

Choosing to compute the resolver selection at operator level has some interesting consequences on the performances. Indeed, this computation is much lighter than approaches with heuristics on plan space [14] as it reduce 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.

+ not computed each time
+ [illegible] head on sort [illegible]

2.4 LOLLIPOP Algorithm

The LOLLIPOP algorithm is not too different from the regular POP. In fact, the recursion algorithm is exactly the same as described in algorithm 1. The main differences reside in its initialization and agenda as described in algorithm 4.

#TODO better redaction and flow

We use strongly connected component detection algorithm to detect cycles. These cycles will be used during the base score computation.

We added a detection of illegal fluents and operators in our domain initialization. Illegal operators are either inconsistent or toxic.

Definition 14 (Inconsistent operators)**.** *An operator a is contradictory if and only if*

$$\exists f \{f, \neg f\} \in eff(o) \lor \{f, \neg f\} \in pre(o)$$

Definition 15 (Toxic operators)**.** *Toxic operators have effects that are already in their preconditions or empty effects. They are defined as:*

$$o | pre(o) \cap eff(o) \neq \emptyset \lor eff(o) = \emptyset$$

Toxic actions can damage a plan as well as make the execution of POP algorithm much longer than necessary. This is fixed by removing the toxic fluents ($pre(a) \nsubseteq eff(a)$) 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.

During the building of the agenda, a search of liar links is performed. This search triggers the addition of orphan flaws if links are removed.
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 can 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, 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.

Algorithm 4 LOLLIPOP initialisation mechanisms

```
function DOMAININIT(Operators O)
    ProperPlan P
    Ranking R
    for all Operator o ∈ O do
        if ISILLEGAL(o) then                         ▷ Remove toxic and useless fluents
            O ← O \ {o}                              ▷ If entirely toxic or useless
            continue
        P.ADDVERTEX(o)                               ▷ Add, cache and bind all operators
    Cycles C ← STRONGLYCONNECTEDCOMPONENT(P)        ▷ Using DFS
    R.Z ← BASESCORES(O, P)
    R.I ← INAPPLIABLES(P)
    R.E ← EAGERS(P)
function LOLLIPOPINIT(Problem Π)
    REALIZE(Π.Δ.R, Π)                                ▷ Realize the scores
    CACHE(Π.Δ.P, Π.I)                                ▷ Cache initial step in providing ...
    CACHE(Π.Δ.P, Π.G)                                ▷ ... as well as goal step
    SORT(Π.Δ.P.providing, Π.Δ.R)                     ▷ Sort the providing map
    if Π.p.L = ∅ then
        SAFE(Π)                    ▷ Computing the safe proper plan if the plan is empty
    POPULATE(a, Π)                                   ▷ populate agenda with first flaws
function POPULATE(Agenda a, Problem Π)
    for all Update u ∈ Π.U do                        ▷ Updates due to online planning
        Fluents F ← eff(u.new) \ eff(u.old)          ▷ Added effects
        for all Fluent f ∈ F do
            for all Operator o ∈ BETTER(Π.Δ.P.providing, f, o) do
                for all Link l ∈ Π.P.L⁺(o) do
                    if f ∈ l then
                        ADDALTERNATIVE(a, f, o, →(l), Π)
                                                     ▷ With →(l) the target of l
        F ← eff(u.old) \ eff(u.new)                  ▷ Removed effects
        for all Fluent f ∈ F do
            for all Link l ∈ Π.P.L⁺(u.new) do
                if ISLIAR(l) then
                    Π.L ← Π.L \ {l}
                    ADDORPHANS(a, u.new, Π)
        ...          ▷ Same with removed preconditions and incomming liar links
    for all Operator o ∈ Π.p.S do
        ADDSUBGOALS(a, o, Π)
        ADDTHREATS(a, o, Π)
```

3 Analysis

#TODO List of properties:

- Lollipop is complete and sound (which is quite important)
- Lollipop will always output plans at least as good as POP (define a measure of quality first)
- Lollipop won't need to compute more standard flaw than POP

Proof of something. The proof

□

4 Experimental Results

#TODO Things we want to know:

- Is Lollipop faster than POP ? In which cases?
- Is the lollipop competitive in small problems?
- which heuristic are the best? How to improve? Metaheuristic ?
- Measure the increase in quality
- Plot the selection time and every other indicator are taken in [25]

Conclusion

#TODO Things we want to discuss:

- Discussion of results and properties
- Summary of improvements
- Introducing soft solving and online planning.
- Online planning
- plan recognition and constrained planning

Conclusion

#TODO

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

[7] J. Benton, A. J. Coles, and A. Coles, "Temporal Planning with Preferences and Time-Dependent Continuous Costs." in *ICAPS*, 2012, vol. 77, p. 78.

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

[9] 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.

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

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

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

[13] 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.

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

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

[16] 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.

[17] 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.

[18] W3C, "RDF 1.1 Turtle: Terse RDF Triple Language." https://www.w3.org/TR/turtle/, Jan-2014.

[19] D. L. Kovacs, "BNF definition of PDDL 3.1," *Unpublished manuscript from the IPC-2011 website*, 2011.

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

[23] 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.

[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.