LOLLIPOP: aLternative Optimization with partiaL pLan Injection Partial Ordered Planning

Paper ID#42

Abstract. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque quis arcu ultricies, feugiat magna convallis, vulputate ex. Nam ullamcorper, velit vel fringilla laoreet, leo felis pellentesque massa, quis volutpat quam erat sed leo. Sed sagittis hendrerit ante, sit amet bibendum mi ornare a. Pellentesque eget convallis purus. Nunc aliquam nisl sit amet cursus scelerisque. Vivamus lacinia, tortor quis placerat mollis, tellus est dignissim nulla, et mattis nisl mi at neque. In tincidunt maximus tellus vel consequat. Nam malesuada nec dolor et tempus.

Introduction

Automated planning aims to synthesize a set of actions into an ordered plan to achieve the user's goals. Most academic research on general purpose planning was, until the end of the 90s, oriented toward plan space planning. It was praised for its least commitment orientation that makes it more efficient than classical planning and also more of an elegant solution for its flexibility. 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] and LAMA [3].

This search of scalability and performance shadows the greatest advantage of Partial-Order Planning (POP): flexibility. This advantage allows POP to efficiently build plans with the parallel use of actions. It also means that it can refine broken plans and optimize them further than a regular state-based planner. This was shown clearly during the golden age of POP with UCPOP [4] . It was a reference in terms of quality and expressiveness for years, to such extends that other works tried to inject state-space planners with its advances [5].

In this paper, we explore new ideas to revive POP as an attractive alternative to other totally ordered approach. Some papers like the appropriately named "Reviving partial order planning" [6] and VHPOP [7] already tried to extend POP into such an interesting alternative. More recent efforts [8], [9] are trying to adapt the powerful heuristics from state-based planning to POP's approach.

Our approach is different: we want to build a robust and quality oriented planner without losing speed. In order to achieve this, the planner is implemented in its recursive form and takes a partial plan as an input. Our system prepopulates this plan with a proper domain plan that was processed during the domain compilation phase. This allows it to have a significant head-start in relation to most planners as the plan is already almost complete in most cases. This is described in section 2 of the present paper.

The problem with this is that this partial plan contains problems that can break the POP algorithm (such as cycles) or that can drive it toward nonoptimal results. This is why we focus the section 3 of our paper on plan quality and negative refinements. This leads naturally toward the introduction of negative flaws that aims to optimise the plan: the cycle, the alternative and the orphan flaws.

This causes another problem since POP wasn't made to select these flaws as they can interfere with positive flaws by deconstructing their work. That is the reason behind the section 4 of our work: goal and flaw selection that aims to reduce the branching factor of our algorithm. This will allow greater speed and better plan quality.

This leads to a complete and coherent planning algorithm that gives optimised plans rather efficiently. The algorithm with its phases and properties is explained in the section 5 with the experimental result presented in detail in section 6.

In order to present our aLternative Optimization with partiaL pLan Injection Partial Ordered Planning (LOLLIPOP) system, we need to explain the classical POP framework and its limits.

1 Classical Partial Order Planning Framework

In this paper, we decided to build our own planning framework based on PDDL's concepts. This new framework is called WORLD as it is inspired by more generalistic world description tools such as RDF Turtle [10]. It is about equivalent in expressiveness to PDDL 3.1 with object-fluents support [11].

We chose this type of domain description because we plan to extend on the work of Göbelbecker et al. [12] in order to make this planner able to do soft resolution in future works. The next definitions are based on the ones exposed in this paper.

1.1 Domain

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

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.

The symbol system is completed with a notion of **term** (either a constant, a variable parameter or a property) and a few relations. We provide types with a relation of **subsumption** noted $t_1 \prec t_2$ with $t_1, t_2 \in T$ meaning that all instances of t_1 are also instances of t_2. On terms, we add two relations: the **assignation** (noted $\leftarrow$) and the **potential equality** (noted $\doteq$).

1.2 Problem

Along with a domain, every planner needs a problem description in order to work. For this, we use a simplified version of the classical problem representation with some special additions.

From there we add the definition of a planning problem as the 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 wich all undefined predicates are false in the initial state and undefined properties doesn't have a value.

Even if the present framework is based upon classical plan space planning it introduces some key differences. For example, we need to add the partial plan as a problem parameter since our approach requires it. We define a partial plan as a tuple $\langle S, L, B \rangle$ with S the set of **steps** (instantiated operators also called actions), L the set of **causal links**, and B the set of **binding constraints**. In classic representations, we also add ordering constraints that were voluntarily omitted here. Since the causal links always are supported by an ordering constraint and since 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. The old ordering constraint can still be achieved using the transitive form of the causal links. That allows us to introduce the **precedence operator** noted $a_i \succ a_j$ iff there is a path of causal links that connects a_i with a_j with a_i being anterior to a_j.

1.3 Flaws

A specificity of Partial Order Planning is that it must fix 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.

Definition 1 (Subgoal)**.** *What we call subgoal is the type of flaw that consists into a missing causal link 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 **proper fluent** of the subgoal.*

Definition 2 (Threat)**.** *We call a threat a type of flaws that consist 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 can say that a step a_b is said to threaten a causal link $a_p \xrightarrow{t} a_n$ iff*

$$\neg t \in eff(a_b) \land a_p \succ a_b \succ a_n \models L$$

*In this case we call the action a_b the **breaker**, a_n the needer and a_p provider of the proper fluent t.*

1.4 Resolvers

These flaws are fixed via the application of a resolver to the plan. A flaw can have several resolvers that match its needs.

Definition 3 (Resolvers)**.** *A resolver is a potential causal link. We can note it as a tuple $r = \langle a_s, a_t, f \rangle$ with :*

- *a_s and a_t 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**.

1.5 Related Flaws

Once the resolver is applied, another important step is needed in order to be able to keep refining. The algorithm needs to find the related flaws of the elected resolver. These related flaws are searched by type.

The algorithm for searching related flaws is a function that needs the *trouble maker* and the problem. A **trouble maker** is the *source* of the resolver if it was inserted into the plan. There is no need to search for related flaws when fixing a threat or when simply adding a causal link.

1.6 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.

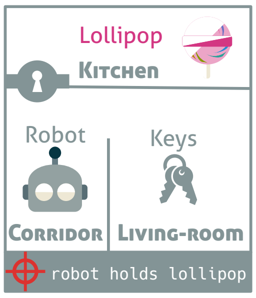
Algorithm 1 Classical Partial Order Planning

```
function POP(Queue of Flaws $a$, Problem $\Pi$)
    POPULATE($a$, $\Pi$)                    // Populate agenda only on first call
    if $a = \emptyset$ then
        return Success                       // Stop all recursion
    Flaw $f \leftarrow$ CHOOSE($a$)          // Non deterministic choice
    Resolvers $R \leftarrow$ RESOLVERS($f$, $\Pi$)
    for all $r \in R$ do                     // Non deterministic choice operator
        APPLY($r$, $\Pi.p$)                  // Apply resolver to partial plan
        $a \leftarrow a \cup$ RELATED($r$)   // Related flaws introduced by the resolver
        if POP($a$, $\Pi$) = Sucess then     // Refining recursively
            return Sucess
        REVERT($r$, $\Pi.p$)                 // Failure, undo resolver insertion
    return Failure    // Revert to last non deterministic choice of resolver
```

The algorithm 1 was inspired by [13]. This POP implementation uses an agenda of flaws that is efficiently updated after each refinement of the plan. At each recursion, it selects a flaw in the agenda to remove it for processing. It then selects a resolver and tries to apply it. If it fails to apply all resolvers the algorithm backtracks to last choice to try another one. The algorithm terminates when no more resolvers fit a flaw or when all flaws have been fixed.

1.7 Limitations

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. The last part is that if we input a partial plan that has problems into the algorithm it can break and either not returning a solution to a solvable problem or give an invalid solution.



```
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. Extract of example domain and problem

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 observe that if we inserted an operator that has a similar effect as *go* but that has impossible precondition (e.g. $false$), then the algorithm might select it and will need backtracking to solve the plan. This problem is solved via using simple goal selection methods. However, to our knowledge, this was never applied in the context of POP.

Another limitation is met when the input plan contains a loop or a contradiction. We consider a partial plan similar to $I \rightarrow go(robot, livingroom) \rightarrow grab(robot, keys) \rightarrow go(robot, corridor) \rightarrow go(robot, livingroom) \dots$. There is a loop in that plan (between the two *go* actions) that standard POP algorithms won't fix. In the literature this is not considered since classical POP doesn't take a partial plan as input or hypothesise out directly such inconsistencies.

2 Proper Plan Generation and Injection

One of the main contributions of the present paper is our use of what we call a *domain proper plan* in order to quickly derive a partial plan from it.

2.1 Definition

First of all we need to define what is a domain proper plan.
Definition 4 (Domain proper plan)**.** *The proper plan Δ^P of a planning dommain Δ is a labelled directed graph that binds two operators $o_1 \xrightarrow{f} o_2$ iff it exists at least an 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 [12] and originally used as heuristic in [14]. A variation of this notion was used in [15] that directly binds the goal and uses precondition nodes instead of labels. That makes the proper plan a kind of operator dependency graph of the domain. With this information, we can know how potentially useful an operator can be in any plan and also where to look for providers of a fluent.

2.2 Generation

This plan is computed during the domain compilation time and, therefore, gives an advantage to the POP algorithm. For more explanation of phases see section 5.

Algorithm 2 Domain proper plan generation algorithm

```
function LOLLIPOP(Queue of Flaws $agenda$, Problem $\Pi$)
    SOLVEALLWORLDSPROBLEMS($agenda$, $\Pi$)        // Only on first call
```

The algorithm 2 is based upon the previous definition. It will explore the operators space and build a causal map to know wich operator causes what. 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.

This proper plan is saved for usage related to heuristics (cf. section 4.1).

2.3 Initial and Goal Step Injection

The next step is to derive a viable partial plan from the proper plan. The main problem with this is the lack of initial or goal step in it. Since it is made during domain compilation time the algorithm doesn't have access to the problem's data. That is why during the problem processing phase, an algorithm will inject the initial and goal step into the plan. This uses the same algorithm used to build the proper plan in the first place. It will bind the initial step to the operators that can be used in the initial world state and the goal step to the operators that can fulfil its preconditions. We decide to leave the operators uninstantiated for our cycle breaking mechanism to work. This mechanism is based on the alternative negative flaw (cf. definition 6)

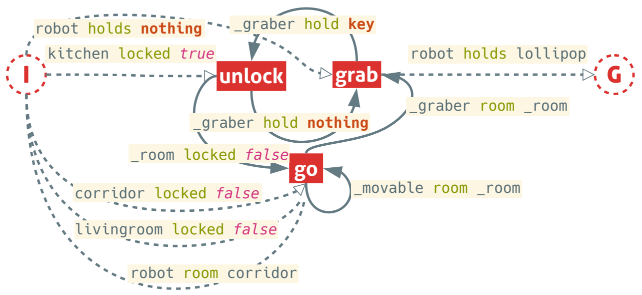


Figure 2. Proper plan of example domain with dotted initial and goal step insertion.

In figure 2 we illustrate the proper plan mechanics with our previous example. The most notable feature of this graph is its coverage. It contains cycles wich can be proven problematic for POP.

However, cycles contain 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*. This information help detects early inconsistencies in the plan when instantiating the operators in relation to the initial and goal step. The solution is then to simply remove inconsistent causal links while saving them as potentially problematic. Then the initialization algorithm only has to break the remaining loops using cycle flaws. This flaw is described in section 3.

The last of these problems is that even if the proper plan can be coherent and even solve the problem, it may contain many unnecessary steps. This is the main reason why we introduce *negative refinements* in the next section.

3 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 or even dangerous to the solution.

3.1 Negative Flaws

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

Definition 5 (Cycle)**.** *A cycle is a flaw corresponding to a set of causal links that forms a loop. A causal link $a_i \rightarrow a_j$ belongs to a cycle iff it exists a path from a_j to a_i. This path can be of lenght 0 in whitch case $a_i = a_j$ and the cycle is a self-loop.*

The complete path of a cycle is called a **closed walk**. In order to improve runtime efficiency, the cycles are detected as the proper plan is built: during domain compilation phase (see more about phases section 5).

Definition 6 (Alternative)**.** *An alternative is a negative flaw that occurs when it exists a better provider choice for a given link. We can say that 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 (cf. section* **??***)*

The alternatives are expensive to find. For this reason, they are found only once during the initialization phase (cf. section 5).

Definition 7 (Orphan)**.** *An orphan is a negative flaw that means that a step in the partial plan (that is not the initial or goal step) is not participating in the plan. In other words a_o is an orphan iff $a_o \neq I \wedge a_o \neq G \wedge p.d^+(a_o) = 0$.*

With $p.d^+(a_o)$ being the outgoing degree of a_o in the dirrected graph formed by p.

3.2 Negative Refinements

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

We add onto the definition 3 :

Definition 8 (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.

A *cycle* will have as negative resolvers each causal link belonging to its closed walk. This way the algorithm will know that there are no solution if there aren't any way to remove any causal links of the cycle. An improvement upon this would be to sort the links with the utility value of their target.

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 choose the better alternative.

The way to fix an orphan is quite simple. The negative refinement is only meant to remove the targeted action and its incoming causal link while tagging the sources of them as potential orphans.

3.3 Related Flaws For Negative Resolvers

The standard mechanism of related flaws needs an upgrade since the new kind of flaws can easily cause side effects and even interfere with one another. All of this is explained in the figure 4. This figure contains the notion of **side effects** of flaws. This notion is important to understand how to properly search for related flaws. The first thing to notice in the system is that when treating positive resolvers nothing need to change from the classical method. When dealing with negative resolver we need to search for additional subgoals and threats. In fact negative refinements will most likely cause an increase in subgoals or threats. Another side effects of negative refinements is on the generation of possible orphan flaws.

4 Driving Flaw and Resolver Selection

Resolver and flaws selection are the keys to improvements in performances. Choosing a good resolver helps to reduce the branching factor that accounts for most of the time spent on running POP algorithms.

4.1 Operator Oriented Heuristics

A heuristic is a function that allows to rank operators. This is at the heart of the algorithm since it will try to make the best choice for goal selection. Driving choice was already shown as a performance improvement mechanism in [7] as several heuristics were combined to improve POP's efficiency.

In our case, we chose to have one main heuristic that aims to lower branching factor by trying to make the base operations more aware of the utility of the considered data. A good explanation of the mechanics behind this can be found in [16].

The aim is to create heuristics that have a sense of the usefulness of an operator or step. In order to do that, we need to create a function that will give a higher value the more the action participates potentially or effectively in a plan and a lower value the more it is needy.

We tried different function that could accomplish this and compared them together. Since we rely heavily on the notion of positive and negative degree, we need to define them first:

Definition 9 (Degrees). *Degrees are meant to measure 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.

There are several ways to use the degrees as an indicator. The goal is to increase the *utility value* with every d^+, since this reflects a positive participation in the plan, and decrease it with every d^- since actions with a higher incoming degree means being harder to satisfy. With this idea in mind, we chose to try several approaches to compute the utility value. In order to unify the notation we decide to transform the data into tuples with degrees ordered from the most specific to the most general : $d^\pm(o) = \langle P.d^\pm(o), \Delta.d^\pm(o), o.d^\pm \rangle$

4.1.1 Simple Degree Product

One of the first ways that come to mind in order to achieve this is to simply subtract the negative degrees from the positive ones and to do the product of all the results:

$$h_{simple} = \alpha_{simple} \prod_{i=1}^{3} d_i^+(o) - d_i^-(o)$$

with α_{simple} being the unification constant. It is used to adjust the value of the heuristics to similar levels as to other heuristics for comparison. We can see two problems with this approach: it gets to zero if any data misses or if any degree cancels each other. This is problematic since it means that the result gets less entropy from the data than it should.

4.1.2 Simple Ratio

Another way to do this is to do a ration of the degrees. The problem is that it often occurs that any divisor will turn out to be zero, wich is problematic. A simple workaround is to simply add a constant to the divisor :

$$h_{ratio} = \prod_{i=1}^{3} \frac{d_i^+(o)}{d_i^-(o) + \alpha_{ratio}}$$

4.1.3 Logarithmic Ratio

Taking the previous approach and trying another way to deal with null divisor gave this solution. We use powers to actually transform the quotient into a product that allows zero as a value :

$$h_{logarithmic} = \left(\sum_{i=1}^{3} d_i^+(o) \right) \times \alpha_{logarithmic}^{-\sum_{i=1}^{3} d_i^-(o)}$$

This has the advantage to be able to be more finely tune the way the heuristic behaves, we can make incoming edges more or less damaging to the utility of an action.

4.1.4 Scalar Product

Another way to combine the data is to see it as vectors. A classical operation that can be done between two vectors is to get the dot product of the two. The formula become :

$$h_{scalar} = \alpha_{scalar} \times \left(d^+(o) \cdot -d^-(o) \right) = \alpha_{scalar} \sum_{i=1}^{3} d_i^+(o) \times \left(-d_i^-(o) \right)$$

Since most of the time we get big negative values we chose to often make $0 < \alpha_{scalar} < 1$.

4.1.5 Normative Vectorial Product

Another operation that can be done is to take the norm of the product of the two vector. This gives us :

$$h_{vectorial} = \alpha_{vectorial} \times \left| d^+(o) \times -d^-(o) \right|$$

These heuristics have clearly different behaviours. To illustrate this we plotted the values given by each of them in our example in figure 3.

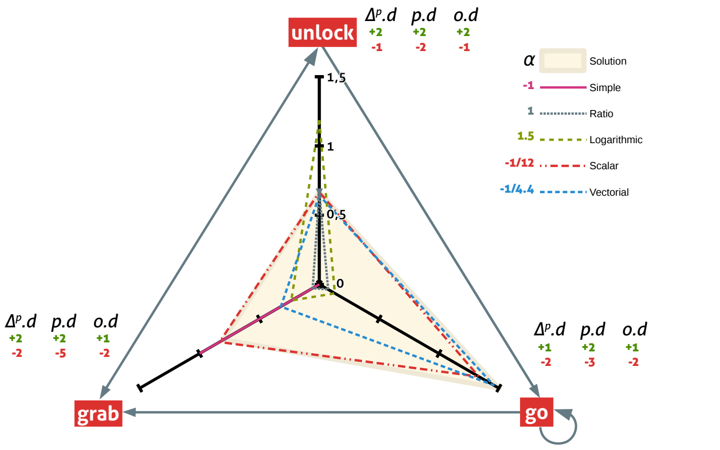


Figure 3. Repesentation of the application of the different heuristics on the example. The solution is computed using a variation of the *simple ration* heuristic : $\left(p.d^+(o)/p.d^-(o) + 1\right) i(o)$ with $i(o)$ being the number of instances of the operator in the solution plan p

We chose the best-suited alpha values in order to fit the solution.

What appears as the most appropriate heuristic is the scalar product. It gives good results on this example. However we see empirical results show that **????**

4.2 Flaws Interference and Priority

Another problem lies in the selection of flaws. Indeed, the order that they are fixed can cause big problems. This kind of problems is called **flaw interference**.

These interferences include alternatives and cycles where there is no actual need to find an alternative to a causal link belonging to a closed walk. Another problem arise with the competition between orphans and subgoals: a subgoal might need an orphan and, therefore, remove the need for removal. Removing orphans before being sure that they won't be needed can be counter productive. The last interference is quite known in the domain. It is the interaction between subgoals and threats as subgoals can remove some threats without the need for intervention.

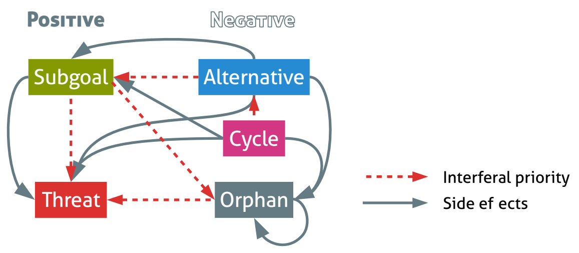


Figure 4. Schema of interferencial priorities and causal side effects

All the possible interferences are listed in the figure 4. This schema shows the dirrection of the interference denoting the priotity of each flaws on the others. This leads us to the following ordered list of priority amongst flaws.

1. **Cycles** that comes from the original domain proper plan are an indication that some operators are co-dependent and that is often where problems arise. This step is also the most susceptible to cause an early failure wich is very beneficial for the speed of the algorithm.
2. **Alternatives** will cut causal links that have a better provider. It is necessary to do that first since they will add at least another subgoal to be fixed as a related flaw.
3. **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.
4. **Threats** occurs quite often in the computation and requires lots of computation to check if they are applicable and are one of the most side-effect heavies. That is why we prioritise all related subgoals before threats because they can actually add causal links that will fix the threat without needing to do anything.
5. **Orphans** are a fine optimisation of plans. They 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 we are sure that they won't be needed.

4.3 Dynamic Flaw Selection

5 LOLLIPOP Algorithm

With all these mechanisms defined we can now present the complete algorithm of LOLLIPOP :

Algorithm 3 All the yummy sweet lollipop

```
function LOLLIPOP(Queue of Flaws $agenda$, Problem $\Pi$)
    SOLVEALLWORLDSPROBLEMS($agenda$, $\Pi$)        // Only on first call
```

5.1 Domain Phase

The domain compilation phase is the first to quick in. It needs to parse and interpret a domain description input and make it usable by POP algorithms. This phase includes a cleaning phase that rules out illegal defects present in the initial description in order to be resilient to basic logic errors. Most of that compilation include a translation from the expressive complexity of the input to a much simpler representation used in algorithms for performance. In this step, we also add our proper plan creation algorithm along with cycle detection on the result. This is used to cache cycle flaws in order to make the agenda population easier during runtime. We also keep a copy of the proper plan in memory and start ordering operators to allow fast goal selection.

5.2 Initialization

This phase comes as the problem is provided. It contains multiple steps.

1. Initial and Goal insertion into the proper plan to construct the initial partial plan.
2. Binding variables and inconsistencies detection.
3. Add the cached cycle flaws to the top of the agenda.
4. Search and add alternative flaws that aren't in cycles.
5. Search and add all obvious subgoals left unfulfilled in the plan.
6. Search for early threats in the partial plan and add them to the bottom of the agenda.
7. Search for all orphans and add them last to the agenda.

5.3 Main Loop

That part is similar to regular POP algorithms as it will almost only take flaws from the agenda and try to fix them one by one. On of the differences is that each time a flaw is fixed, it calls a special function that provides related flaws. In the case of cycles or alternative, we can have potential orphans and new subgoals that need to be fixed. In the case of subgoals, it can lead to new subgoals or threats. Threats are a bit special in that regard. They can't cause other flaws but are treated differently. The algorithm will always delay them, after all, current subgoals have been fixed. Then all orphans that are kept last may only cause other orphans .

5.4 Properties and Proofs

???? 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

□

6 Experimental Results

???? 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 ? Meta heuristic ?
- Measure the increase in quality
- Plot the selection time and every other indicator taken in [16]

Conclusion

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

[5] 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.

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

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

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

[9] 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.

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

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

[12] 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.

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

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

[15] M. A. Peot and D. E. Smith, "Postponing Threats in Partial-Order Planning," 1994.

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