LOLLIPOP: aLternative Optimization with partiaL pLan Injection Partial Ordered Planning

Paper ID#42

Abstract. Abstract goes here

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 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.

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 The 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 and problem

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 $\doteq$).

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.

1.2 Classical flaws and resolvers

Also, since we will introduce a new type of flaws, we need to rewrite the existing ones to fit the new generic resolvers.

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*

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

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

These flaws are fixed via the application of a resolver to the plan. A flaw can have several resolvers that match its needs. Since we will have negative flaws we need the resolver to be able to handle both types of flaws.

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

- *a_s and a_t being the source and target of the resolver,*
- *f being the considered fluent,*
- *s being the sign of the resolver in $\{+, -\}$.*

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

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

1.3 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.

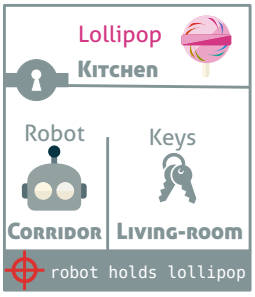
Algorithm 1 Classical Partial Order Planning

```
1  function POP(Queue of Flaws agenda, Problem Π)
2      POPULATE(agenda, Π)                          // Only on first call
3      if agenda = ∅ then
4          return Success                           // Stop all recursion
5      Flaw f ← agenda.popFromQueue                 // First element of the queue
6      Resolvers R ← RESOLVERS(f, Π)                // Ordered resolvers to try
7      for all r ∈ R do                             // Non deterministic choice operator
8          APPLY(r, Π.p)                            // Apply resolver to partial plan
9          if CONSISTENT(Π.p) then                  // Π.p is the partial plan
10             POP(agenda∪ RELATEDFLAWS(f), Π)
                                                    // Finding new flaws introduced by the resolver
12         else
13             REVERT(r, Π.p)                       // Undo resolver insertion
14     return Failure  // Revert to last non deterministic choice of resolver
```

The algorithm 1 was inspired by [13], [14]. This POP implementation uses an agenda of flaws that is efficiently updated after each refinement of the plan. At each recursion, it selects the flaw at the top of the agenda pile and 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.

1.4 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 the partial plan similar to $I \rightarrow go(robot, livingroom) \rightarrow grab(robot, keys) \rightarrow go(robot, corridor) \rightarrow go(robot, livingroom) \ldots$. 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 Using domain proper plan as initial partial plan

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 [15]. This plan is computed during the domain compilation time and therefore gives an advantage to the POP algorithm. This proper plan is saved for usage related to the utility function (cf. section 4.1).

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 5)

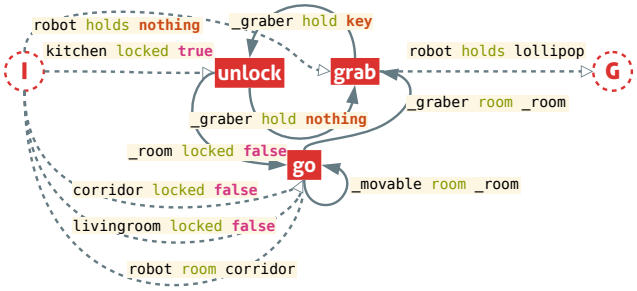


Figure 2. Proper plan of example domain with dotted initial and goal step insertion.

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.

!Alt

1. These cycles are problematic. The first way to deal with them is to remove them preemptively. In this case, the definition is modified to disallow self-loops. The algorithm is also set to break cycles by removing arbitrary edges belonging to each cycle. This solution could be improved based on earlier cycle detection when first building the proper plan or with the removal of the minimum feedback arc set.
2. 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 alternative flaws. If it exists a path from an action to the goal and the cycle breaking phase does not find an alternative the problem is unsolvable.

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 optimisation

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.

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.

3.1 Negative flaws

Definition 5 (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 4.1)*

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. Since the search for an alternative is very expensive it is only executed once during the agenda population phase.

Definition 6 (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 $a_o \neq I \wedge a_o \neq G \wedge P.d^+(a_o) = 0$.*

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.

!Alt

Definition 7 (Cycle)**.** *A cycle is a flaw corresponding to a set of causal links that forms a loop.*

A simple way to deal with cycles is to compute the minimum feedback arc set of the partial plan graph at initialisation. This set contains the best candidates to removal in the set of causal links. Each causal link of this set will cause a cycle flaw to be added to the agenda.

Even if these flaws improve the plan quality, they can also interfere with the standard flaws of POP. Indeed, we can have a goal and an alternative looping with each other if nothing prevents it.

This is the reason why we need a new goal and flaw selection mechanism.

4 Driving resolver and flaw selection

Resolver and flaws selection are the key to improvements in performances. Chosing a good resolver helps reducing the branching factor that account for most of the time spent on running POP algorithms.

4.1 Utility

An utility function is an heuristic that allows to rank operators. This is at the heart of the algorithm since it will try to make the best choice during 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 operation more aware of the utility of the considered data. A good explanation of the machanics behind this can be found in [16].

The **utility value** of an operator o is defined using the following formula :

$$h(o) = \frac{P.d^+(o) \times \alpha^{-P.d^-(o)}}{\Delta^P.d^+(o) \times \alpha^{-\Delta^P.d^-(o)}}$$

with d^- and d^+ being the incoming and outgoing degree of the node representing the given operator in the given graph and α being a numerical constant. Another property of this utility function is that it will return $+\infty$ is the parameter is the initial step.

!Alt Other heuristics :

$$h(o) = \frac{P.d^+(o) + |eff(o)|}{\alpha^{P.d^-(o)+|pre(o)|}}$$

$$h(o) = \frac{P.d^+(o) + \Delta^P.d^+(o)}{\alpha^{P.d^-(o)+\Delta^P.d^-(o)}}$$

The goal behind that heuristic is that it was meant to encourage participating actions against needy ones. So this function will increase with outgoing degree of the opertor in both graphs (domain proper plan and current partial plan).

4.2 Resolver selection

Some flaws have a large amount of resolvers to try. Knowing that chosing the wrong one can cause extensive backtracking its selection if crutial to the performance of the application. From there the selection is the most important for subgoals. This kind of flaws have resolvers with the most impact on the plan since they add a causal link and a new action. A subgoal resolver represent the choice of a provider for the open condition. Since we have a heuristic to rank them we use it to order the set of resolvers during the search. In order to improve even more on the performance, we can keep an ordered list of all available steps and operators updated.

4.3 Flaw selection

Another problem lies in the selection of flaws. Indeed the order that they are fixed can cause big problems. For example if all the subgoals of an alternative are computed before it will cause a big amount of unecessary computation. That is why we chose to compute the flaws that reduce ther search space the most at first. Our order is the following :

1. **!Alt Cycles** that comes from the original domain proper plan are an indication that some operators are co-dependant and that is often where problems arise. This step is also the most succeptible 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 causes the most branching factor for POP algorithms. This is why we need to make sure that all open conditions are fixed before proceding 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 heavy. That is why we prioritise all related subgoals before threats because they can actually add cuasal links that will fix the threat without needing to do anything.
5. **Orhpans** are a fine optimisation of plans. They remove uneeded 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.

5 Complete LOLLIPOP algorithms

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

Algorithm 2 All the yummy sweet lollipop

```
1 function LOLLIPOP(Queue of Flaws agenda, Problem Π)
2 |   SOLVEALLWORLDSPROBLEMS(agenda, Π)      // Only on first call
```

5.1 Domain compilation phase

The domain compilation phase is the first to quick in. It needs to parse and interprate 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 those compilation includes 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 durring runtime. We also keep a copy of the proper plan in memory and start ordering operators to allow fast goal selection.

5.2 Runtime initialization phase

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 unfulfiled 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 in the agenda.

5.3 Runtime main loop phase

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 needs 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 curent subgoals have been fixed. Then all orphans that are kept last may only cause other orphans .

6 Found properties of LOLLIPOP

Proof of something. The proof

□

7 Experiments

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, "Planification et décision," *Hermes*, 2001.

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

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

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