LOLLIPOP: aLternative Optimization with causaL Link Injection Partial Ordered Planning

Paper ID#42

Abstract. Abstract goes here

Introduction

For a long time, in the automated planning domain, the dominant approach was Partial-Order Planning (POP). 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, in the recent years, drastic improvements in state search planning was made possible by advanced and efficient heuristics. This allows such planners to scale up much better than plan-space search ones, notably thanks to approaches like GraphPlan [1], fast-forward [2] and LAMA [3].

This search of scalability and performance hides the greatest advantage of 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] that 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. For our system, we prepopulate this plan with the causal graph that was processed during the domain compilation phase. This allows our algorithm 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 plans. This is why we focus the section 3 of our paper on plan quality since we need to have negative refinements on our input plan. 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 causaL Link 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 Definitions

We define our **planning domain** as a tuple $\Delta = \langle T, C, \mathcal{F}, F, O \rangle$ where

- T are the **types**,
- C is the set of **domain constants**,
- $\mathcal{F}$ is the set of **functions** 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**.

We need to add the partial plan as a problem parameter since our approach requires a partial plan as input.

Moreover, in comparison to classical plan-space planning, we decided to make the representation of the partial plan simpler: it is formed by 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.

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 $a_p \xrightarrow{t} a_n$ if and only if*

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

*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 problems 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

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

1.2 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.

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

---

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)
11                            // 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
```

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.

1.3 Limitations

This standard implementation has several limitations. First it can easily make bad choices that will lead to excessive backtracking and poor performances. 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.
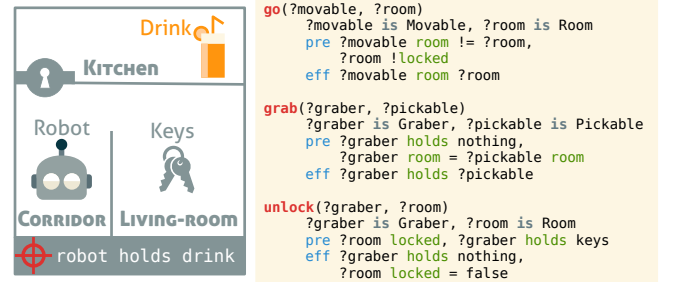

Figure 1. Extract of example domain and problem

To illustrate these limitations, we use the example described in figure 1 where a robot must fetch a drink 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, it 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 an obvious loop in that plan 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 causal graph as partial plan

First of all we need to define what is a domain causal graph.

Definition 4 (Domain causal graph). *The causal graph Γ of a planning dommain Δ is a labelled directed graph that binds two operators $o_1 \xrightarrow{f} o_2$ with $o_1 \neq o_2$ iff it exists at least an unifying fluent $f \in eff(o_1) \cap pre(o_2)$.*

We chose to not include self-loops in the graph since it would actually polute the data for our specific usage since we try to remove loops before using the graph as a partial plan.

3 Negative refinements and plan optimisation

4 Driving goal and flaw selection

5 Found properties of LOLLIPOP

Proof of something. The proof

$\square$

6 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. S. Nau, and P. Traverso, *Automated planning: theory and practice*. Amsterdam ; Boston: Elsevier/Morgan Kaufmann, 2004.