a reformuler pas clair, a enlever ou autre

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].

REDITE

a major/one of the main

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. pas clair ? 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].

justifier, citation, a vérifier ?

motivation de l'article ? non

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.

c'est quoi proper domain plan ?

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.

negativ et positiv flaw ?

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

ne faudrait pas juste dire que tu utilises un planning framework basé sur RDF et PDDL 3.1 ?

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.

plutôt à mettre dans l'intro générale ? Dire en une phrase l'article de Godelbecker

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**.

wHich

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.

present fwk: POP ou celui que tu présentes ?

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.

pour quoi ?

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. new generic resolvers ? le tien ?

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.*

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.

c quoi les negative flaw ? et les positif —> cf. §3

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

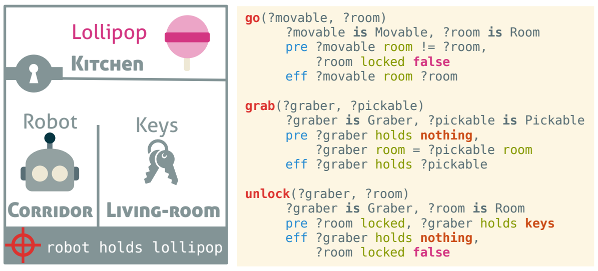
```
function POP(Queue of Flaws $agenda$, Problem $\Pi$)
    POPULATE($agenda$, $\Pi$)                                  // Only on first call
    if $agenda = \emptyset$ then
        return Success                                        // Stop all recursion
    Flaw $f \leftarrow agenda.popFromQueue$                  // First element of the queue
    Resolvers $R \leftarrow$ RESOLVERS($f$, $\Pi$)            // Ordered resolvers to try
    for all $r \in R$ do                                      // Non deterministic choice operator
        APPLY($r$, $\Pi.p$)                                   // Apply resolver to partial plan
        if CONSISTENT($\Pi.p$) then                           // $\Pi.p$ is the partial plan
            POP($agenda \cup$ RELATEDFLAWS($f$), $\Pi$)
                                     // Finding new flaws introduced by the resolver
        else
            REVERT($r$, $\Pi.p$)                              // Undo resolver insertion
    return Failure     // Revert to last non deterministic choice of resolver
```

pourquoi ? il faut justifier

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.