SODA POP : Robust online Partial Ordering Planning for real life applications

Paper ID#232

Abstract

As of recent years, automated planning domain has mainly focused on performances and advances in state space planning to improve scalability. That orientation shadows other less efficient ways of doing like Partial Ordering Planning (POP) that has the advantage to be much more flexible. This approach *generates* plans that can be refined easily and can be used to achieve greater resilience, repairing capabilities and soft resolution that applications *based on* online plan recognition or decision-making can benefit from. *This paper presents* a set of algorithms, named Soft Ordering and Defect Aware Partial Ordering Planning (SODA POP), that aims at targeting these objectives and maintain a high plan quality. Our algorithms create proper offline plans for goals and use an effective defect fixing *algorithm* to repair input plans even if these plans are corrupted. SODA POP can also use healer actions and links in order to always return a valid plan even in cases where the problem is impossible to solve using problem derivation. Some relevant properties of these algorithms are analyzed in this article and experimental results show interesting performances for online planning and repairing.

Introduction

For some time Partial Order Planning (POP) has been the most popular approach to planning resolution. This kind of algorithms are based on *least commitment strategy* on plan step ordering that can allow actions to be flexibly interleaved during execution [1]. Thus, the way the search is made using flexible partial plan as a search space allowed for more versatility for a wide variety of uses. As of more recent years, new state space search models and heuristics [2] have been demonstrated to be more efficient than POP planners due to the simplicity and relevance of states representation opposed to partial plans [3]. This have made the search of performance the main axis of research for planning in the community .

While this goal is justified, it shadows other problems that some practical applications cause. For example, the flexibility of Plan Space Planning (PSP) is an obvious advantage for applications needing online planning: plans can be repaired on the fly as new informations and objectives enter the system. The idea of using POP for online planning and repairing plans instead of replanning everything is not new [4], but has never been paired with the resilience that some other cognitive applications may need, especially when dealing with sensors noise in input data.

This resilience makes fixing errors easier than with an external algorithm as the plan logic allows for context driven decision on the way to repair the issues. For example if an action becomes unrelevant or incoherent, the flaw in the partial plan makes the issue explicit and therefore easier to fix. Softwares might sometimes provide plans that can contain errors and imperfections that will decrease significantly the efficiency of the computation and the quality of the output plan.

Adding to that, these plans may become totally unsolvable. This problem is to our knowledge not treated in planning of all forms (state planning, PSP, and even constraint satisfaction planning) as usually the aim is to find a solution relative to the original plan (which makes sense). But as we proceed a mechanism of *problem derivation* may be required. This will allow soft solving of any problem regardless of its resolvability.

One of the applications that needs these improvements is plan recognition with the particular use of off-the-shelf planners to infer the pursued goal of an agent where online planning and resilience is particularly important [5]. This method adds dummy actions that need to be satisfied by modifying the goal to ensure that the observed plan is selected by the planner. A system that is able to repair plan in real time will ease such an application. Another application is decision-making in dynamical environments. Indeed, having a plan that details all steps and explicits the ones that are not possible, can help decide of the line of action to take.

These problems call for new ways to improve the answer of a planner. These improvements must provide relevant plan information pointing out exactly what needs to be done in order to make a planning problem solvable, even in the case of obviously inconsistent input data. This paper aims to solve this while preserving the advantages of PSP algorithms (flexibility, easy fixing of plans, soundness and completeness). Our Soft Ordering and Defect Aware Partial Ordering

Planning (SODA POP) algorithm will target those issues.

Our new set of auxiliary algorithms allows making POP algorithm more resilient, efficient and relevant. This is done by pre-emptively computing proper plans for goals, by solving new kinds of defects that input plans may exhibit, and by healing compromised plan by deriving the initial problem with forged actions to allow resolution.

To explain this work we first introduce a few notions, notations and specificities of existing POP algorithms. Then we present and illustrate their limitations, categorizing the different defects arising from the resilience problem and explaining our auxiliary algorithms, their uses and properties. To finish we compare the performance, resilience and quality of POP and our solution.

Definitions

First we introduce some definitions and notations for mathematical representation to present our model.

Classical planning

Definition 1 (Fluents)**.** *A fluent is a property of the world. We note $\neg f$ the complementary fluent of f meaning that f is true when $\neg f$ is false and vice-versa.*

It is often represented by first order logical propositions but in this paper we choose to focus on the algorithm and to represent fluents as simple literals (fully instantiated) using $\mathbb{Q}^*$, the set of relative integers without 0, as the fluent domain. We use negative integers to represent opposite fluents.

Definition 2 (State)**.** *A state is defined as a set of fluents. States can be additively combined. We note $s_1 + s_2 = (s_1 \cup s_2) - \{f | f \in s_1 \wedge \neg f \in s_2\}$ such operation. It is the union of the fluents with an erasure of the complementary ones.*

Definition 3 (Action)**.** *An action is a state operator. It is represented as a tuple $a = \langle pre, eff\rangle$ with pre and eff being sets of fluents, respectively the preconditions and the effects of a. An action can be used only in a state that verifies its preconditions. We note $s \models a \Leftrightarrow pre(a) \subset s$ the verification of an action a by a state s.*

An action a can be functionally applied to a state s as follows :

$$a := {}_{a(s) \mapsto s + eff(a)}^{\{s \models a | s \in S\} \to S}$$

with S being the set of all states. This means that an action adds all its effect to the state of the world on application (by removing complementary fluents if needed)

We distinguish between two specific kinds of actions : actions with no preconditions are synonymous to a state and those with empty effect are called a goal.

Plan Space Planning

Problem We define a partial plan satisfaction problem as a tuple noted $P = \langle A, I, G, p\rangle$ with :

- I and G being the pseudo actions representing respectively the initial state and the goal.
- p being a partial plan to refine.
- A the set of all actions.

Definition 4 (Partial Plan)**.** *A partial plan is a tuple $p = \langle A_p, L\rangle$ where A_p is a set of steps (actions) and L is a set of causal links of the form $a_i \xrightarrow{f} a_j$, such that $\{a_i, a_j\} \subset A_p \wedge f \in eff(a_i) \cap pre(a_j)$ or said otherwise, this causal link means that the fluent f is provided by an effect of a_i to a precondition of a_j. We include the ordering constraints of PSP in the causal links. An ordering constraint is noted $a_i \to a_j$ and means that the plan means a_i as a step that is prior to a_j without specific cause (usually because of threat resolution). We note the order relation over A for an action a_i that is prior to the action a_j following all the causal links and ordering constraints $a_i \succ a_j$.*

The figure 1 details how the elements of partial plans are represented in the rest of this paper.

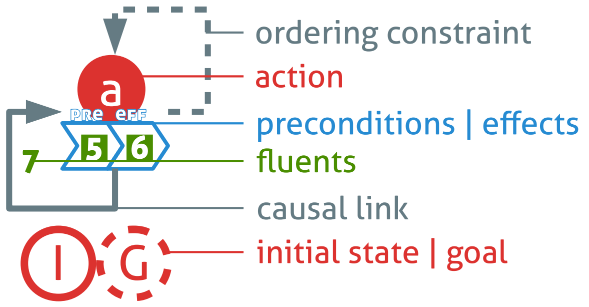

Figure 1: Global legend for how partial plans are represented in this paper

In this figure the fluent 5 is a precondition of the action a and the fluent 6 an effect. The causal link is providing the fluent 7 (only as a representation example or it would be a liar link). The dotted line is an ordering constraint that would mean that a should be placed before a (for representation still or it would be a cycle). I and G are respectively the initial and goal step.

Flaws When refining a partial plan, we need to fix flaws. Those could be present in the input or created by the refining process. Flaws can either be unsatisfied subgoals or threats to causal links.

Definition 5 (Subgoal)**.** *A subgoal s is a precondition of an action $a_s \in A_p$ with $s \in pre(a_s)$ that isn't satisfied by any causal link. We can note a subgoal as :*

$$a_i \xrightarrow{s} a_s \notin L \mid \{a_i, a_s\} \subset A_p$$

Definition 6 (Threat)**.** *A step a_t is said to threaten a causal link $a_i \xrightarrow{t} a_j$ if and only if*

$$\neg t \in eff(a_t) \wedge a_i \succ a_t \succ a_j \models L$$

In other words, the action has a possible complementary effect that can be inserted between two actions needing this fluent while being consistent with the ordering constraint in L.

Resolvers Resolvers are a set of actions and causal links $r = \langle A_r, L_r \rangle$ that fixes flaws. A resolver for a subgoal is an action $a_r \in A$ that has s as an effect $s \in eff(a_r)$ inserted along with a causal link noted $a_r \xrightarrow{s} a_s$. The usual resolvers for a threat are either $a_t \to a_i$ or $a_j \to a_t$ which are called respectively promotion and demotion links. Another resolver is called a white knight that is an action a_k that reestablishes t after a_t along with an ordering constraint $a_t \to a_k$ and a causal link $a_k \xrightarrow{t} a_j$.

Solution The solution of a PSP problem is a valid partial plan that respect the specification of said problem (only using actions in A and having the correct initial and goal step).

Definition 7 (Consistency)**.** *A partial plan is consistent if it contains no ordering cycles. That means that the directed graph formed by step as vertices and causal links as edges isn't cyclical. This is important to guarantee the soundness of the algorithm.*

Definition 8 (Flat Plan)**.** *We can instantiate one or several flat plans from a partial plan. A flat plan is an ordered sequence of actions $\pi = [a_1, a_2 \dots a_n]$ that acts like a pseudo action $\pi = \langle pre_\pi, eff_\pi \rangle$ and can be applied to a state s using functional composition operation $\pi := \bigcirc_{i=1}^{n} a_n$.*

We call a flat plan valid if and only if it can be functionally applied on an empty state. We note that this is different from classic state planning because in our case the initial state is the first action that is already included in the plan.

Definition 9 (Validity)**.** *A partial plan is valid if and only if it is consistent and if all flat plans that can be generated are valid.*

Classical POP

Partial Order Planning (POP) is a popular implementation of the general PSP algorithm. It refines a partial plan by trying to fix its flaws and is proven to be sound and complete [6].

Description

Algorithm 1 presents the base algorithm for a planer in the plan space. POP implementation uses an agenda of flaws that is efficiently updated after each refinement of the plan. A flaw is selected for resolution and we use a non deterministic choice operator to pick a resolver for the flaw. The resolver is inserted in the plan and we recursively call the algorithm on the new plan. On failure we return to the last non deterministic choice to pick another resolver. The algorithm ends when the agenda is empty or when there is no more resolver to pick for a given flaw.

Algorithm 1 Classical Partial Order Planning

```
function POP(Queue of Flaws $agenda$, Problem $P$)
    POPULATE($agenda$, $P$)                        ▷ Only on first call
    if $agenda = \emptyset$ then
        return Success                              ▷ Stop all recursion
    Flaw $f \leftarrow agenda.pop$        ▷ First element of the queue
    Resolvers $R \leftarrow$ RESOLVERS($f$, $P$) ▷ Ordered list of resolvers to try
    for all $r \in R$ do                           ▷ Choice operator
        APPLY($r$, $P.p$)
        if CONSISTENT($P.p$) then
            POP($agenda \cup$ RELATEDFLAWS($f$), $P$)
        else
            REVERT($r$, $P.p$)
    return Failure              ▷ Return to last choice of resolver
```

Limitations

This standard way of doing have seen multiple improvements over expressiveness like with UCPOP [7], hierarchical task network to add more user control over sub-plans [8], cognition with defeasible reasoning [9], or speed with multiple ways to implement the popular fast forward method from state planning [10]. However, all these variants do not treat the problem of online planning, resilience and soft solving.

Some other closer works like [4] treats the problem of online planning by removing big chunks of the partial plan by identifying incorrect trees in the plan. This along with heuristic choice of unrefinemment strategies. This causes a heavy replanning of the problem even if only one action needs removal. This also lead to an exponential number of plan to consider. This is a big problem when trying to adapt a plan with minimal changes due to replanning.

Indeed, all these problems can affect POP's performance and quality as they can interfere with POP's inner working when the algorithm is able to give an answer at all.

Before continuing, we present a simple example of classical POP execution with the problem represented in figure 2. We did not represent empty preconditions or effects to improve readability. Here we have an initial state $I = \langle \emptyset, \{1, 2\} \rangle$ and a goal $G = \langle \{3, 4, -5, 6\}, \emptyset \rangle$ encoded as dummy steps. We also introduce actions that are not steps yet but that are provided by A. The actions a, b and c are normal actions that are useful to achieve the goal. The action t is meant to be threatening to the plan's integrity and will generate threats. We introduce u and v as toxic actions, w as a dead-end action and x as a contradictory action. These notions will be defined later.

This example has been crafted to illustrate problems with standard POP implementations. We give a possible resulting plan of standard POP in figure 3. We can see some issues as for how the plan has been built. The action v is being used even if it is useless since b already provided fluent 4. We can also notice that despite being contradictory the action x raised no alarm. As for ordering constraints, we can clearly see that the link $a \to t$ is redundant with the path $a \xrightarrow{5} c \to t$

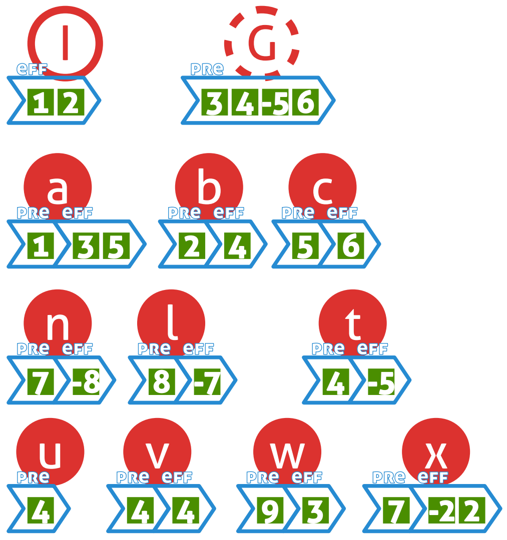

Figure 2: A simple problem that will be used throughout this paper

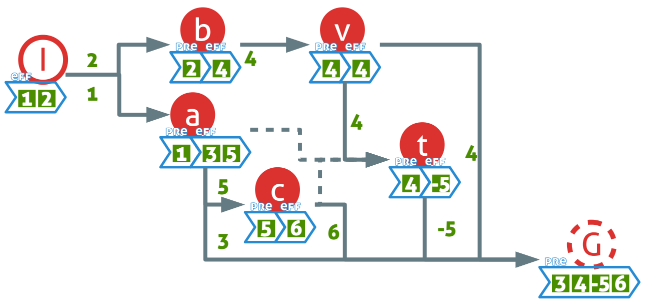

Figure 3: Standard POP result to the problem

that already puts that constraint by transitivity. Also, some problems arise during execution with the selection of w that causes a dead-end.

Of course the flaw selection mechanism of certain variant can prevent that to happen in that case. But often flaw selection mechanisms are more speed oriented and will do little if a toxic action seems to fit better than a more coherent but complex one .

All these issues are caused by what we call *defects* as they are not regular PSP flaws but they still cause problems with execution and results. We will address these defects and propose a way to fix them in the next section.

Auxiliary algorithms to POP

In order to improve POP algorithms' resilience, online performance and plan quality, we propose a set of auxiliary algorithms that provides POP with a clean and efficiently populated initial plan. The complete algorithm will be presented in the next section as a combination of all auxiliary algorithms and classical POP.

Proper plan generation

Algorithm 2 Proper plan generation for a given goal g

```
function PROPERPLAN(Goal $g$, Actions $A$)
    Partial Plan $p \leftarrow \emptyset$
    Actions $relevants \leftarrow$ SATISFY($g$, $A$, $p$)    ▷ Satisfy given goal with all necessary actions and causal links
    Queue of Actions $open \leftarrow relevants$
    while $open \neq \emptyset$ do
        Action $a \leftarrow open.pop$
        Actions $candidates \leftarrow$ SATISFY($a$, $A$, $p$)
        for all $candidate \in candidates$ do
            if $candidate \notin relevants$ then
                $open.push(candidate)$
```

As in online planning goals can be known in advance, we propose a new mechanism that generates proper plans for goals. We take advantage of the fact that this step can be done offline to improve the performance of online planning. This offline execution prevents us to access the details of the initial state of the world as it will be defined at runtime. We define for that the concept of *participating action*. An action $a \in A$ participates in a goal G if and only if a has an effect f that is needed to accomplish G or that is needed to accomplish another participating action's preconditions. A proper plan is a partial plan that contains all participating actions as steps and causal links that bind them with the step they are participating in. This proper plan is independent from the initial step because we might not have the initial step at the time of the proper plan generation.

We define the concept of *participating action*. An action $a \in A$ participates in a goal G if and only if a has an effect f that is needed to accomplish G or to accomplish another participating action's preconditions. A proper plan is a partial

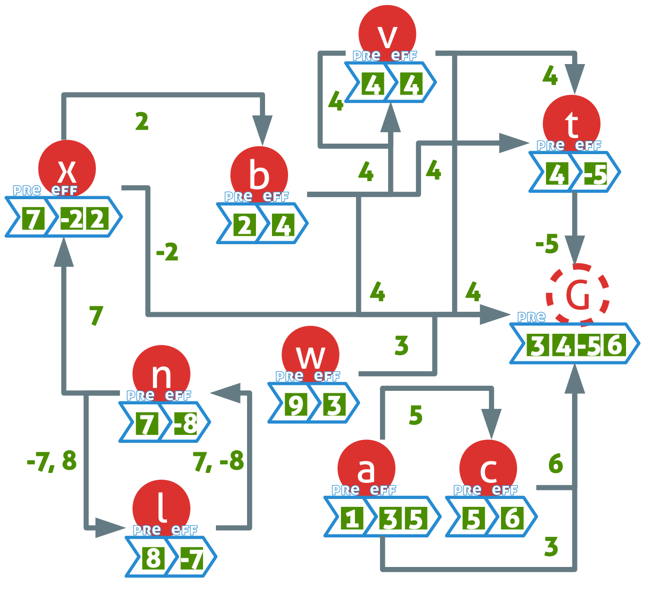

Figure 4: Proper plan of the example goal

plan that contains all participating actions as steps and causal links that bind them with the step they are participating in. Proper plan generation, detailed in Algorithm 2, recursively chooses participating actions. It starts to populate the proper plan with a quick and incomplete backward chaining.

On our example it illustrates really well the need for resilience. The algorithm simply recursively chose the actions and end up with the partial plan represented in figure 4. The partial plan doesn't have an initial state (because of its offline nature). This plan also shows several cycles and obvious problems. Moreover, the plan has all the steps of the correct final plan and therefore remove the subgoal search in POP if it uses this plan has input (after it has been cleaned obviously).

This auxiliary algorithm is therefore used as a caching mechanism for online planning. The algorithm starts to populate the proper plan with a quick and incomplete backward chaining.

Once applied on the example of figure 2, proper plan generation returns the partial plan presented in figure 4. This partial plan doesn't have initial state because of its offline nature. It also shows several cycles and obvious problems. However it has all the steps of the correct final plan.

Defect resolution

When the POP algorithm is used to refine a given plan (that was not generated with POP or that was altered), a set of new defects can be present in it interfering in the resolution and sometimes making it impossible to solve. We emphasize that these defects are not regular POP flaws but new problems that classical POP can't solve. The aim of this auxiliary algorithm is to clean the plans from such defects in order to improve

Algorithm 3 Defect resolution

```
function CLEAN(Problem $P$)
    ASSERT($pre(P.I) = \emptyset$)
    ASSERT($eff(P.G) = \emptyset$)
    $P.p.A_p \leftarrow P.p.A_p \cup \{P.I, P.G\}$
    $defects \leftarrow$ FINDDEFECTS($P$)
    FIX($defects$, $P$)
function FIX(Defects $defects$, Problem $P$)
    for all Defect $d \in defects$ do
        $defect$.FIX        ▷ Fixing is proper to the defect
function FINDDEFECTS(Problem $P$)
    return ILLEGAL($P$) + INTERFERING($P$)
```

computational time, resilience and plan quality. It should be noted that in some cases cleaning plans will increase the number of flaws in the plan but will always improve the overall quality of it.

Definition 10 (Plan Quality)**.** *Plan quality is an indicator that is measured by the number of defects in a partial plan. There are two specific indicators for a plan quality : the action quality and the link quality. We define the link and action quality as respectively the number of link or action related defects divided by the number of links or actions in the resulting plan.*

$$quality_{action}(p) = 1 - \left(\frac{\#ActionDefects(p)}{\#p.A_p}\right)$$

From there it is obvious that plan quality will improve over POP since the present algorithm is guaranteed to remove all defects in a plan.

There are two kinds of defects: the illegal defects that violate base hypothesis of PSP and the interference defects that can lead to excessive computational time and to poor plan quality.

Illegal defects These defects are usually hypothesized out by regular models. They are illegal use of partial plan representation and shouldn't happen under regular circumstances. They may appear if the input is generated by an approximate cognitive model that doesn't ensure consistency of the output or by unchecked corrupted data. These defects will most of the time simply break regular POP algorithms (from existing cycles) or at least make the performances decrease significantly (from toxic actions). Illegal defects and solutions to fix them are presented in the following.

Definition 11 (Cycles)**.** *A plan cannot contain cycles as it makes it impossible to complete. Cycles are usually detected as they are inserted in a plan but poor input can potentially contain them and breaks the POP algorithm as it cannot undo cycles.*

We use strongly connected component detection algorithm to detect cycles. Upon cycle detection the algorithm can remove arbitrarily a link in the cycle to break it. The most effective solution is to remove the link that is the farthest from the goal traveling backward as it would be that link that would have been last added in the regular POP algorithm. But finding this is extremely time consuming and most of the time the cycles are of one or two actions so we choose to keep that algorithm simpler and faster.

In a plan some actions can be illegal for POP. Those are the actions that are contradictory.

Definition 12 (Inconsistent actions)**.** *An action a is contradictory if and only if*

$$\exists f \{f, \neg f\} \in eff(a) \vee \{f, \neg f\} \in pre(a)$$

We remove only one of those effect or preconditions based on the usage of the action in the rest of the plan. If none of those are used, we choose to remove both. In our example in figure 2 the action x is one of these inconsistent actions with fluent 2 and -2 in its effects.

Definition 13 (Toxic actions)**.** *Toxic actions are those that have effects that are already in their preconditions or empty effects. This can damage a plan as well as make the execution of POP algorithm much longer than necessary. They are defined as :*

$$a | pre(a) \cap eff(a) \neq \emptyset \vee eff(a) = \emptyset$$

They 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 ($pre(a) \not\subseteq eff(a)$) following $eff(a) = eff(a) - pre(a)$. If the effects becomes empty, the action is removed alltogether from plan and A. For example in figure 2 the actions u and v are toxic as the fluent 4 is in the effects and preconditions of v, and u has empty effects. In this case these actions will be removed by the algorithm as they don't have any other effects.

The defects can be related to incorrect links. The first of which are liar links.

Definition 14 (Liar links)**.** *A liar link is a link that doesn't reflect 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 either already present in the data or created by the removal of an effect of an inconsistent or toxic action (with the causal link still remaining).

We call lies fluents that participates in the lies. To resolve the problem we remove f and add all savior fluents, i.e. a fluent in $eff(a_i) \cap pre(a_j)$ that isn't already provided. We delete the link all together if it doesn't link any fluents.

Interference defects This kind of defects is not as toxic as the illegal ones: they won't make the plan unsolvable but they can still cause performance drops in POP execution. These can appear more often in regular POP results as they are not targeted by standard implementations. Interference defects and solutions to fix them are presented in the following.

Definition 15 (Redundant links)**.** *A redundant link have a transitive equivalent of longer length. It means that a link $a_i \rightarrow a_j$ is redundant if and only if it exists another path from a_i to a_j of length greater to* 1.

Since POP relies on those additional links, this part focus on removing the ones that were generated for threat removal purpose to simplify the plan. For example in figure 3 we can see that the ordering constraint from a to t is redundant with the path $a \xrightarrow{5} c \rightarrow t$ in that it isn't needed to resolve the threat. Therefore, the algorithm would remove it. This reduces the number of edges in the plan and therefore simplifies it.

Causal links can be found to compete with one another.

Definition 16 (Competing causal links)**.** *A competing link $a_i \xrightarrow{f} a_k$ competes with another link $a_j \xrightarrow{f} a_k$ if it provides the same fluent to the same action.*

This cannot happen in classical POP algorithm so it is not handled by it. Multiple links providing the same fluent to the same action are making the plan more complex and can also hide useless actions. This is why they must be removed in a way that reveals useless actions so as to reduce the need for extra unecessary actions in the plan. Otherwise this would cause more flaws to handle.

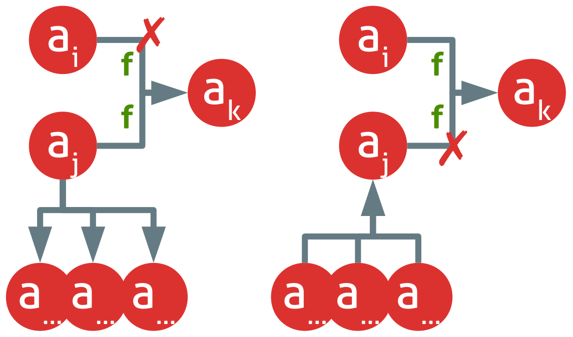

Figure 5: Example of choice for cutting competing links based on usefulness of actions

In order to remove the least interesting link in competing causal links, we need to compare their respective providing action to prune the least useful ones.

Definition 17 (Usefullness of an action)**.** *The usefullness of an action is the number of participating links (outgoing causal links) divided by the number of needing links (incoming causal links). If there are no needing links, or the action is either an initial step or a goal the usefulness is $+\infty$. The usefulness of saviour action is* 0.

We choose the link with the source action having the highest usefulness. This replaces unnecessary saviour actions and allows to reduce the usefulness of the least action in the hope it will become orphan and pruned from the plan .

In the example figure 5 the action a_j on the left participates much more than the action a_i and therefore the link to be removed would be $a_i \xrightarrow{f} a_k$. On the right example the actions doesn't have different outgoing links but the action a_j is here much needier than its competitor and therefore the link to be removed is the link $a_j \xrightarrow{f} a_k$.

On the left example of figure 5, the action a_j participates much more than the action a_i and therefore the link to be removed would be $a_i \xrightarrow{f} a_k$. On the right example the actions don't have different outgoing links but the action a_j is here much needier than its competitor. Therefore the link to be removed is the link $a_j \xrightarrow{f} a_k$.

Actions can sometimes have no use in a plan as they don't contribute to it.

Definition 18 (Orphan actions)**.** *Orphans actions are actions without any links or action with no outgoing path to the goal (meaning that it doesn't participate in the plan). That also concerns action that would become orphan if another orphan action is removed.*

In order to fix this we derive the idea of the backward removal tree of [4]. The algorithm, detailed in Algorithm 4, removes an action if all its outgoing causal links are leading to useless actions. It iterates over all actions until no new useless action is discovered. All useless actions are then removed from the plan.

Algorithm 4 Orphan actions finding

```
function $OrphanAction$.FIND(Problem $P$)
    Actions $orphan \leftarrow \emptyset$
    int $oldSize$
    repeat
        $oldSize \leftarrow \#orphan$
        for all Action $step \in P.p.A_p$ do
            if $step \in orphan$ then
                continue
            int $towardOrphan \leftarrow 0$
            Causal Links $outgoing \leftarrow$ OUTGOINGEDGESOF($step$, $P.p$)
            for all Causal Link $link \in outgoing$ do
                if $link.target \in orphan$ then
                    $towardOrphan{+}{+}$
            if $\#outgoing = towardOrphan \wedge step \notin \{P.I, P.G\}$ then
                $orphan \leftarrow orphan \cup \{step\}$
    until $oldSize \neq \#orphan$
    return $orphan$
```

Definition 19 (Competing actions)**.** *In the same way links can be competing toward a common needer, there are sometimes in the plan actions that are more suited to achieve a goal. These actions are taken into account if they are more useful than the source of the link, if they wouldn't cause a cycle and if they have effects that are carried by the link.*

In such a case the link is removed and another one is formed by the better suited action.

Soft resolution

This auxiliary algorithm is meant to deal with failure. It will heal the plan to make the failure recoverable for the next iteration of POP. Of course it can't fix the plan by keeping the problem as it is. This obviously breaks some properties as the algorithm no longer adheres to the specification of the input, but in exchange it will always issue a valid plan whatever happens. For more information on this property go take a look at the appropriate section bellow.

Algorithm 5 Soft resolution healing

```
function HEAL(Problem $P$)
    int $minViolation \leftarrow \infty$
    Plan $best \leftarrow P.p$
    Flaw $annoyer$
    for all $\langle flaw, plan\rangle \in P.partialSolutions$ do
        int $currentViolation \leftarrow$ VIOLATION($plan$, $P.G$)
        if $currentViolation < minViolation$ then
            $best \leftarrow plan$
            $annoyer \leftarrow flaw$
            $minViolation \leftarrow currentViolation$
    $P.p \leftarrow best$
    $P.partialSolutions \leftarrow \emptyset$
    for all Resolver $resolver \in$ HEALERS($annoyer$) do
        APPLY($resolver$, $P.p$)
```

Soft failure is useful when the precision and validity of the output is not the most important criteria we look for. In some cases (like in recognition processes) it is more useful to have an output even if it is not exact than no output at all. That is why we propose a soft failing mechanism for POP algorithms.

Definitions We define first some new notions, then we will explain the healing algorithm.

Definition 20 (Needer)**.** *A needer is an action that needs a resolution related to a flaw. We define different types of needer according to the type of the flaw.*

- *Subgoal needer For a subgoal $a_n \xrightarrow{s} a_s$ the needer is the action a_n that has an unsatisfied precondition in the current partial plan.*

- *Threat needer The needer of a threat a_t of a link $a_p \xrightarrow{t} a_n$ is the target a_n of the threatened causal link.*

Definition 21 (Proper fluents)**.** *A proper fluent of a flaw is the one that caused the flaw. For a subgoal $a_n \xrightarrow{s} a_s$ it is the unsatisfied precondition s. For a threat a_t of a causal link $a_p \xrightarrow{t} a_n$ it is the fluent t held by the threatened causal link.*

Definition 22 (Savior)**.** *The savior of a flaw is the forged action $a_s = \langle\emptyset, \{p\}\rangle | a_s \notin A$ with p being the proper fluent of the flaw.*

The concept of healer is made to target rogue flaws that caused total failure.

Definition 23 (Healers)**.** *A healer is a resolver that is built around the savior of the flaw in order to provide the missing fluents to it. The general formula of a healer is the following :*

$$a_s \xrightarrow{p} a_n$$

with a_s being the savior of the flaw.

For threats we need an additional healer specified as an ordering constraint from the threatening action to the saviour $a_t \to a_s$ to ensure that the saviour acts after the threat and therefore provides the proper fluent for the needer.

Definition 24 (Violation degree)**.** *The violation degree $v(p)$ of a plan p is an indicator of the health of a partial plan. It is the sum of the number of flaws and the number of saviors in the plan.*

Healing process The healing method is to keep track of reversions in the algorithm by storing the partial plan and the unsatisfiable flaw each time a non deterministic choice fails. We note the set of these failed plans F. As the POP algorithm encounters a final failure, this auxiliary algorithm get invoked. The aim is to evaluate each backtracking partial plan to choose the best one.

Therefore, we add an order relation for F noted

$$\prec : p \prec q \iff v(p) < v(q) | \{p, q\} \subset F$$

Once the POP algorithm fails completely, the soft failing algorithm can be invoked to heal the plan. It chooses the best plan $b | \forall p \in F, b \prec p$ to heal it. If two plans have the same violation degree, the algorithm chooses one arbitrarily .

The healing process is similar to how POP works : we apply the healer of the flaw that caused the failure of the partial plan we chosen. We empty the set F to allow POP to iterate further since the flaw is resolved. The healing process can be done for each unsolved flaws as POP fails repeatedly. This ensures some interesting properties explained in the following section.

SODA POP and its properties

The combination of classical POP and all the auxiliary algorithms presented previously gives SODA POP algorithm detailed in Algorithm 6. In this section we will focus on the properties of SODA POP.

Algorithm 6 SODA POP

```
function SODA(Problem $P$)
    $P.p \leftarrow$ PROPERPLAN($P.G$, $P.A$) ▷ Offline execution then this will be replaced by the previous online plan adjusted with the environment.
    CLEAN($P$)
    bool $valid \leftarrow false$
    while $\neg valid$ do
        $valid \leftarrow$ POP($P$) $= Success$
        if $valid$ then
            CLEAN($P$)
            return $Success$
        HEAL($P$)
```

Convergence of POP

As to our knowledge, no proof of the convergence of POP has been done we want to explicitly formulate one.

The classic planning problem is already proven to be decidable without functions in the fluents [3]. Therefore, we can categorize the termination cases.

Convergence of POP over solvable problemes. In the case of a solvable problem, POP is proven to be complete. This ensures that it will find a solution for the problem and therefore terminate.

□

Now for the more complex case of unsolvable problems we need to refer to the way POP works.

Convergence of POP over unsolvable problemes. POP algorithm will seek to solve flaws. Be $flaws(p)$ the set of flaws of a given partial plan.

The number of flaws is the number of subgoals plus the number of threats

$$flaws(p) = subgoals(p) \cup threats(p)$$

We consider the number of actions $\#A$ as being finite. Therefore, the number of steps in the plan is at worse $\#A_p = \#A$. We also assume that actions has a finite number of preconditions and effects (since we don't use functions over fluents). This leads to :

$$\#subgoals(p) < \sum_{a \in A_p} \#pre(a) < \infty$$

$$\#threats(p) < \#L \leq \sum_{a \in A_p} \#pre(a) < \infty$$

This means the number of all possible flaws is finite. As POP resolves these flaws it will decrease their numbers and iterate over resolvers. The number of resolver is $\#subgoals(p) * \#A + \#threats(p) * 2$ and is also finite. This means that the iteration will in the worst case be of the number of resolvers before failing and this proves that the algorithm terminates.

□

Hypersoundness

Now that we proved that regular POP converges we can introduce the next property : hyper soundness.

Definition 25 (Hypersoundness)**.** *An algorithm is said to be hypersound when it gives a valid solution for all problems regardless of their resolvability.*

We note that this property isn't compatible with consistency regarding the original problem and then doesn't fit the classical idea of soundness that implicitly states that the validity of a solution is relative to the problem. In the case of hypersoundness the problem is valid and a solution to a problem $P' = \langle A', I, G, p \rangle$. We note the new set $A' = A \cup S$

with S being the set of saviors for all flaws that made the POP algorithm fails during the execution of SODA.

The hypersoundness of our combined algorithm is proven using the convergence of POP and the way the Soft solving behaves.

Hypersoundness of SODA POP. The proper plan and defect fixing algorithms are obviously convergent. The proper plan algorithm cannot iterate more than the number of actions (since duplicates are forbidden) and the defect resolution will fix the finite numbers of defects present in the finite partial plan issued by the proper plan algorithm.

POP is sound and converges. Therefore, if SODA POP converges it will return a valid solution for P'. In the same way we proved for POP, the new healing process converges because it reduces the number of flaws in the partial plan and since this number is finite, the algorithm converges.

$\square$

Enhancement for online planning

SODA POP algorithm can be used in dynamic online environments to allow a robust way to replan an existing obsolete plan. The first thing to execute prior to the runtime is the generation of the proper plans for all the goals that will be considered.

For the first execution of the online planning the initial state is added to the problem and the defect detection is applied with the rest of the SODA POP algorithm. This will give the first plan.

For every change of the environment the previous plan is modified accordingly and then fed to the defect detection and SODA algorithm again. This must be done each time the plan is modified in order to actualize the plan. The new plan will have all defects fixed. For example a savior action can be removed if the initial step or a new action becomes competing with it and actions with liar link to a removed fluent in the goal will become probably an orphan.

In practice this will generate little to no iteration of the algorithm.

Experimental results

In order to test the validity of these algorithms we implemented a prototype in Java. This prototype is exactly derived from the definitions and pseudo code presented in this paper. We used basic actions with integer fluents to focus on the way the algorithm behaves.

For the simulation runs we used a computer with an Intel® Core™ i7-4720HQ CPU clocking at 2.60GHz for 8 cores and 8GB of memory. The speed was measured in nano seconds before and after the call to the solve function. The process has a warming up phase that compute random plans for a few seconds before starting the benchmarks. The computer was left idle during the test to remove any related noise to the measurements.

Quality of classical POP

The first metric we decided to measure was the performance of our reference POP algorithm. We measured the solving time for valid problems and the quality of the resulting plans. In figure 6 we can see a plot of these measures.

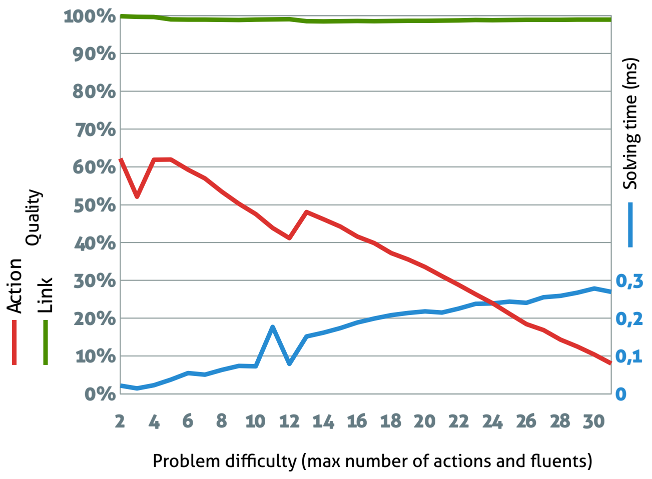

Figure 6: Action and link quality and solving time of POP over 10000 valid problems for each difficulty

We tested the algorithm with randomly generated valid problems. We generated those by randomly creating plans based on a difficulty setting that was the upper bound for the number of actions and half the possible number of fluents. This difficulty setting also made the initial and goal step larger accordingly. The actions were cleaned of any toxicity and was built using a forward chaining algorithm. We added also random unrelated actions. Each problem was verified by POP before being generated to ensure resolvability.

As we clearly see in the figure 6, POP linearly increase its solving time depending on the problem difficulty. That obviously follows the fact that POP has more flaws to fix and therefore more iterations.

As we can observe, POP has a rather high link quality with almost no problems issued with causal link regardless of the difficulty of the problem. This can be linked to either the way we generated our valid plans or to a real tendency of POP to not create that kind of defects on its own.

The action quality on the other hand drops significantly as the difficulty gets higher. This is linked to the number of competing and useless actions that can make the plan simpler but are kept by the flaw selection mechanism of POP.

Performance of SODA POP

As SODA POP and regular POP doesn't have the same range of capabilities we can't compare them hand to hand. Indeed, our algorithm will always output plans with 100% quality since the defect detection system aims to remove them all and since our algorithms is hypersound it will give valid plans for unsolvable problems by derivation. In these case POP won't be able to return a result and will terminate much quicker as the plan is found unsolvable.

Therefore, we measured the performances of our set of algorithms on completely random problems. Each time the algorithm outputted a solution with the lowest possible violation despite the complete invalidity of the input. In figure 7 we can see the way SODA POP scales up on larger problems.

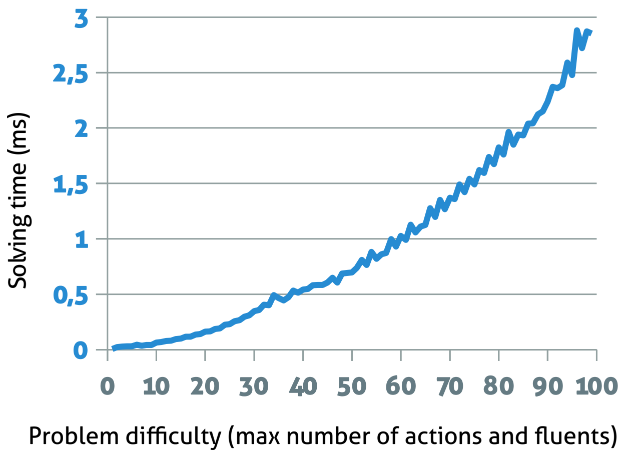

Figure 7: Average solving time of SODA POP over 10000 random problems for each difficulty

The performances display logarithmic solving time. This is explained by the fact that POP has to be called recursively on the healed problems. The defect detection participates too in the increase of solving time especially the competing action detection that needs to iterate over all actions and edges and calculate usefulness for two actions each time.

Conclusion

We defined our SODA POP algorithm and demonstrated its properties. The set of algorithms has extended capabilities of plan repairing and soft solving as shown in examples and simulations. While slower than POP, the resilience of SODA POP can be used in various applications. We aim to use it on the particular case of online plan recognition and decision making as the capabilities to derive plans and repair them will prove useful when comparing potential goals with observed plans.

The algorithm can be improved with a better defect detection algorithm that can rank them and start fixing the most offensive defects before targeting others. Also, if the competing action detection can be improved, it can be used as heuristic for resolver selection to improve the performances.

Another way the present work can be improved is by extending it to the use of more expressive fluents and port all notions to variable choices. One possible goal would be to combine this work with ontologies or an argumentation language in order to significantly boost expressiveness.

We hope that this work will be useful and that it will be extended and used in various applications in the future and that is why we plan to release the software with an open source license soon.

References

[1] D. S. Weld, "An introduction to least commitment planning," *AI magazine*, vol. 15, no. 4, p. 27, 1994.

[2] S. Richter, M. Westphal, and M. Helmert, "LAMA 2008 and 2011," in *International Planning Competition*, 2011, pp. 117–124.

[3] M. Ghallab, D. S. Nau, and P. Traverso, *Automated planning: theory and practice*. Amsterdam ; Boston: Elsevier/Morgan Kaufmann, 2004.

[4] R. Van Der Krogt and M. De Weerdt, "Plan Repair as an Extension of Planning." in *ICAPS*, 2005, vol. 5, pp. 161–170.

[5] M. Ramırez 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.

[6] K. Erol, J. A. Hendler, and D. S. Nau, "UMCP: A Sound and Complete Procedure for Hierarchical Task-network Planning." in *AIPS*, 1994, vol. 94, pp. 249–254.

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

[8] P. Bechon, M. Barbier, G. Infantes, C. Lesire, and V. Vidal, "HiPOP: Hierarchical Partial-Order Planning," in *STAIRS 2014: Proceedings of the 7th European Starting AI Researcher Symposium*, 2014, vol. 264, p. 51.

[9] D. R. García, A. J. García, and G. R. Simari, "Defeasible reasoning and partial order planning," in *Foundations of Information and Knowledge Systems*, Springer, 2008, pp. 311–328.

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