Handling Non-local Dead-ends in Agent Planning Programs

Lukáš Chrpa
Charles University in Prague &
Czech Technical University
Prague, Czech Republic
chrpaluk@fel.cvut.cz

Nir Lipovetzky
University of Melbourne
Melbourne, Australia
nir.lipovetzky@unimelb.edu.au

Sebastian Sardina
RMIT University
Melbourne, Australia
sebastian.sardina@rmit.edu.au

Abstract

We propose an approach to reason about *agent planning programs* with global information. Agent planning programs can be understood as a *network* of planning problems, accommodating long-term goals, non-terminating behaviors, and interactive execution. We provide a technique that relies on reasoning about "global" dead-ends and that can be incorporated to any planning-based approach to agent planning programs. In doing so, we also introduce the notion of *online* execution of such planning structures. We provide experimental evidence suggesting the technique yields significant benefits.

1 Introduction

Conforming with an actor's view of a planning system [Ghallab *et al.*, 2014], several *extended* planning-based frameworks have been proposed in the literature, such as Dal Lago *et al.* [2002]'s EAGLE language, De Giacomo *et al.* [2010; 2016]'s Agent Planning Programs, and Shivashankar *et al.* [2012]'s Hierarchical Goal Networks (HGN). Loosely speaking, such frameworks propose, in some way or another, "programming with goals" approaches for specifying and synthesizing behavior that better caters for integration with a real-world executor and is more amenable for knowledge engineering. Here, we contribute to this line of work with a simple but effective technique to help solve (and execute) agent planning programs.

Agent Planning Programs (APPs) are a novel approach to represent and synthesize complex behaviors based on *sequences of goals*, *user decisions*, and *repetition of tasks* for continuous long-term operation. Roughly speaking, APPs are directed graphs, where nodes represent the states of the program and edges (transitions) stand for planning problems. Importantly, which path in such a graph is to be taken is external to the APP: it is determined by its user at run-time. The framework was firstly proposed by De Giacomo *et al.* [2010], together with a computational technique via synthesis of a carefully designed GR(1) specification [Bloem *et al.*, 2012], a fragment of LTL with significant lower complexity than full LTL synthesis [Pnueli and Rosner, 1989]. Then an alternative approach using automated planners "off-the-shelf" was proposed and evaluated experimentally [Gerevini *et al.*, 2011; De Giacomo *et al.*, 2016]

The experiments carried out in the above contributions confirm that, while the planning-based approach outperforms the synthesis one, solving APPs remains difficult. The problem of the state-of-the-art approach is that it relies heavily on backtracking at the "meta-planning" level, searching for alternative plans to realize a transition in the APP whenever a subsequent transition in the APP is found not to admit any solution. As already recognized by De Giacomo *et al.* [2016], such (expensive) backtracking is often the result of reasoning too "locally," by solving each transition planning problem in isolation, without considering the rest of the planning problem in the APP. The challenge is that planning engines are designed to solve *single* planning problems (i.e., with only one goal), whereas an APP requires solving *several interdependent* planning problems.

So, to address this, we propose a principled approach to incorporate some "global reasoning" features when solving each (local) planning problem in an APP. The idea is to *extract* global knowledge—by leveraging on recent work on reasoning about local *dead-ends* [Lipovetzky *et al.*, 2016]—to *prune* those world states that are known to be in "conflict" with any potential "future" planning problem when the APP is to be executed. In addition, we present an *online* alternative execution model for APPs (as opposed to the offline models from previous contributions). When solving an APP online, extracting and using global information about the whole APP becomes even more crucial for finding successful runs.

To evaluate our proposal, we designed experiments using six planning domains from the International Planning Competition. The results provide evidence that reasoning about "global" dead-ends is able to, often significantly, improve the performance, in terms of the number of solved APPs, of the state-of-the-art offline approach [Gerevini *et al.*, 2011; De Giacomo *et al.*, 2016] as well as of the naive, baseline, online approach, which does not reason about "global" dead-ends. Importantly, the proposed technique is "modular," in that it can be incorporated into any planning system that can (be adapted to) make use of information about dead-ends.

2 Agent Planning Programs

Automated Planning seeks to find a plan to achieve a given goal from an initial world state relative to a model of the en-

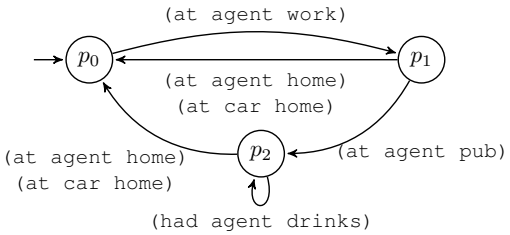

Figure 1: An APP for the daily routine of an academic, meant to be solved relative to a given planning domain (not shown).

vironment [Ghallab *et al.*, 2004]. Classical planning, in particular, assumes a deterministic and fully observable environment; a solution plan amounts to a sequence of actions. Technically, a ***planning domain*** is a tuple $\mathcal{D} = \langle L, O \rangle$, where L is the set of propositional atoms used to describe the world state (set of propositions from L that are true), and O is the set of operators (or actions). An ***operator*** is a tuple $o = (\mathit{name}(o), \mathit{pre}(o), \mathit{del}(o), \mathit{add}(o))$, where $\mathit{name}(o)$ is a unique operator name, and $\mathit{pre}(o), \mathit{del}(o)$, and $\mathit{add}(o)$ are sets of atoms from L representing o's precondition, delete, and add effects, respectively. An operator o is *applicable* (or executable) in a world state s if and only if $\mathit{pre}(a) \subseteq s$. Application of operator o in state s (if possible) yields the successor world state $(s \setminus \mathit{del}(a)) \cup \mathit{add}(a)$. A ***planning problem*** is a tuple $\mathcal{P} = \langle \mathcal{D}, I, G \rangle$, where $\mathcal{D}$ is a planning domain, I is the initial world state, and G is the goal condition, generally in the form of a set of propositions. A ***solution plan*** (for a planning problem $\mathcal{P}$) is a sequence of operators such that their consecutive execution starting in the initial world state I results in a world state s, where $s \supseteq G$. Let $o_1, \ldots, o_n$ be a sequence of operators and $s_0, s_1, \ldots, s_n$ be a sequence of world states such that for all $1 \leq i \leq n$, o_i is applicable in s_{i-1} and s_i is the resulting world state of application of o_i in s_{i-1}. Then, we say that $s_0, s_1, \ldots, s_n$ is a ***state trajectory*** of $o_1, \ldots, o_n$.

A Network of Goals and Planning Problems While powerful, planning problems by themselves have limitations when it comes to capturing many complex situated behaviors. Firstly, a complex behavior cannot generally be expressed in terms of achieving a *single* goal. In a cyber-physical emergency management domain, for example, the support system may first aim to deploy an emergency team to the disaster zone, and once that is achieved, plan to assess the situation, and then to rescue the victims. Thus, an agent will typically *go through a sequence of goals*. Secondly, which goal comes next in such a sequence may depend on factors external to the system. The decision whether, once at the location, the emergency support system should plan for transporting the victims, saving property, or just mitigating danger may come from a human domain expert (in the deployed team). That is, we want behavior that is "interactive." Finally, complex systems are often meant to run continuously, and not just terminate after goal achievement.

Agent Planning Programs (APPs) aim to accommodate these features while remaining in the realms of automated planning [De Giacomo *et al.*, 2010; 2016]. Essentially, APPs are high-level representations of complex behavior in a given domain. Technically, they are transition systems, with nodes representing decision points and transitions representing possible goals via triples consisting of a guard, a maintenance goal, and an achievement goal.

Definition 1. An ***agent planning program*** over a planning domain $\mathcal{D} = \langle L, O \rangle$ is a tuple $\mathcal{A} = \langle L, V, v_0, s_0, \delta \rangle$, where:

- V is the finite set of *program states*;
- $v_0 \in V$ is the *initial program state* of $\mathcal{A}$, and $s_0 \in 2^L$ is the initial world state; and
- δ is the *transition relation* of $\mathcal{A}$, where $\langle v, \langle \gamma, \psi, \phi \rangle, v' \rangle \in \delta$, written $v \xrightarrow{\gamma:\psi,\phi} v'$ in $\mathcal{A}$, stating that whenever the *guard* condition γ over L holds, $\mathcal{A}$ may legally move from its state v to state v' by "*achieving* ϕ *while maintaining* ψ" (in the domain).

Thus, a transition in an APP corresponds to a planning problem over the same shared domain $\mathcal{D}$ such that a guard (γ) must hold in the current world state – the initial state, an achievement goal (ϕ) is the goal. Solution plans for such a planning problem must comply with a maintenance goal (ψ). That is, ψ holds in each world state in the state trajectory of a "complying" solution plan. Figure 1 depicts a simplified APP for an academic everyday-life routine with only achievement goal transitions (example taken from [De Giacomo *et al.*, 2010]. Note that the APP only mentions (goal) conditions—no actions—over propositions in L. What actions are available and how they affect those conditions will be given by some planning domain $\mathcal{D}$ (not shown here). Also observe the difference between program states and world states; the former corresponds to the "program counter" of the APP, while the latter is the usual state of the environment (specified by domain $\mathcal{D}$) the APP is running over.

At any point in time, the system is in a certain program state of the APP, and the *user* (e.g., the academic) issues a transition goal request, among the possible ones in such a state. In our example, the academic can at the start only aim to be at work (transition from p_0 to p_1). The system then *deploys a plan* that achieves the requested goal, and the APP then evolves to its next program state; in our case node p_1. After that, the user is able to issue the next goal request. Now, from program state p_1, the academic may request to return back home (transition p_1 to p_0) *or* alternatively aim to go to the pub with colleagues (transition p_1 to p_2). Once again, depending on the goal requested, the system is meant to deploy an adequate plan, always. As one can see, each goal decision choice is *decided at run-time*, by an external "user" of the APP (e.g., a human expert or another software). Importantly, the system ought to be prepared to meet *any* legal sequence of goals. Observe also that these sequences are unbounded as the APP is cyclic and encodes repetition of tasks, thus accounting for continuous behavior. In the example, the academic always comes back to home, and if the car has been used, it also must be back at home by the end of the day.

Solving Agent Planning Programs A solution to an APP—called a *realization* of the APP—is a strategy guaranteeing successful plans as goal-transitions are requested by

the user [De Giacomo *et al.*, 2016]. More precisely, a ***realization*** is a ***policy*** $\sigma(v, s, g)$ that maps a program state v of the APP, a world state s, and a transition with an achievement goal g to a plan π that will achieve goal g from s (relative to the shared planning domain $\mathcal{D}$), and will leave the agent ready to realize, once again, any possible next goal in the APP. Intuitively, plans in a realization solution ought to be "synchronized". I.e., the world state resulting from the execution of a plan for a transition τ in the APP should be a legal initial world state for (all) the planning problems corresponding to the directly outgoing transitions (from the program state τ moves into). This implies that, unfortunately, solutions do not simply amount to assigning plans to APP's transitions, as a transition may require different plans for different initial world states. As a matter of fact, checking realizability of an APP is hard: it is EXPTIME-complete even under deterministic domains [De Giacomo *et al.*, 2016].

Two approaches have been developed to check (and compute) existence of realizations. The first approach was proposed in [De Giacomo *et al.*, 2010] and resorts to reactive synthesis techniques for GR(1) specification [Bloem *et al.*, 2012]. While very powerful in that it outputs *universal* solutions containing all possible realizations, it can be very demanding computationally. The second approach was first proposed by Gerevini *et al.* [2011] and leverages state-of-the-art planning systems to find a realization, thus being able to take advantage of recent and future advances in automated planning. While it does not produce universal solutions, it has been shown to scale up better [De Giacomo *et al.*, 2016]. Moreover, the technique is conceptually simple and leaves room for further optimizations. Because of this we are interested, in this paper, in pushing such technique further.

Roughly speaking, the idea behind the planning approach to APPs is as follows. Consider an APP $\mathcal{A} = \langle L, V, v_0, s_0, \delta \rangle$ over a planning domain $\mathcal{D}$. Assume that $\mathcal{A}$ is maintenance free,[1] so its transitions are of the form $v \xrightarrow{\gamma:\texttt{true},\phi} v'$, which we shall write as $v \xrightarrow{\gamma:\phi} v'$. So, at each step, a planner is used, off-the-shelf, to find a solution plan for the planning problem corresponding to a particular transition in $\mathcal{A}$. At the start, for every initial transition (i.e., for every γ_0, G_0 and v_1) of the form $v_0 \xrightarrow{\gamma_0:G_0} v_1$ in $\mathcal{A}$ for which condition γ_0 holds true in world state s_0, a corresponding planning problem $\langle \mathcal{D}, s_0, G_0 \rangle$ is solved. Suppose that π is the solution plan found for one of such initial transitions, and that s_1 is the world state resulting from executing π from s_0. The next step involves synthesizing successful plans for all the planning problems corresponding to (applicable from s_1) transitions arising from state v_1 in $\mathcal{A}$ and the initial world state s_1. So, for every transition (i.e., for every γ_1, G_1 and v_2) of the form $v_1 \xrightarrow{\gamma_1:G_1} v_2$ in $\mathcal{A}$ for which γ_1 holds in world state s_1, the planning problem $\langle \mathcal{D}, s_1, G_1 \rangle$ ought to be solved. This process is repeated until all transitions in $\mathcal{A}$ have been "covered" (for every world state in which such transition may apply). If, at any stage, no plan solution is found for any of such problems, then *backtracking* is triggered to the previous transition, for an alternative plan that leaves the world in another, hopefully better, state.

[1] Maintenance can be compiled away, e.g., [Edelkamp, 2006].

To make this iterative process more efficient, Gerevini *et al.* [2011] exploit plan generation techniques with *preferred end-states* and *tabu end-states*. The authors used the LPG planning system [Gerevini *et al.*, 2003], which supports both features. The idea is to bias each planning instance to a world state from where solution plans are known (for the subsequent goals in the successor program state), and to forbid world states in which we know no solutions can be found for future goals. For example, suppose that the solver is required, at some stage, to find a plan for planning task $\langle \mathcal{D}, s, G \rangle$ in order to fulfill a transition $v \xrightarrow{\gamma:G} v'$ in $\mathcal{A}$. Suppose further that plans have already been constructed, in previous iterations, for all planning tasks corresponding to transitions arising from $\mathcal{A}$'s state v' and world states in some subset $S_{v'}$. Then, the idea is to bias the planner solving goal G from world state s towards a solution plan that will eventually leave the world into one of the states in $S_{v'}$. If that happens, no planning is required for transitions arising from APP state v', they have already been constructed (and stored). In turn, if the solver has previously failed to find a successful plan for some transitions arising from v' in some world state s', then s' is treated as a tabu state when solving $\langle \mathcal{D}, s, G \rangle$ for APP state v. Notice, thought, that it may take several (expensive) meta-level backtracking steps (i.e., backtracking at the APP level) for the solver to be aware of such "bad" states.

Below, we show how to further enhance the just described planning-based approach by using dead-end detection formulas as *global* information about future possible planning problems when solving each "local" problem, so that meta-level backtracking is mitigated.

3 Handling Non-local Dead-ends in APPs

Intuitively, dead-ends are branches of the search tree that do not lead to any goal state. So, world states from which the goal cannot be achieved are *dead-end states* for that goal.

We call a dead-end state "local" if the current goal is unreachable. In the context of APPs, we are interested in avoiding non-local dead-ends instead, i.e., world states where future possible goals belonging to other planning problems are unreachable. Indeed, our intention is to compile global information to solve planning problems that avoid future unsolvable problems. Relevant to our aim, Lipovetzky *et al.* (2016) proposed a new approach to characterize dead-end states as a k-DNF *trap*, namely, a formula in disjunctive normal form whose terms have at most k literals. A trap is a conditional invariant of the planning problem, i.e., once a state is reached that satisfies the trap formula, the trap is also satisfied by all its reachable states. A trap that is mutually exclusive with the goal characterizes a dead-end trap formula, as it is true only in dead-end states. Intuitively, a dead-end trap is a disjunction of partial states that capture a region of the search space from which the goal is unattainable. The algorithm to compute k-DNF dead-end traps is exponential on the k parameter (i.e., $\mathcal{O}(n^k)$, where n is the number of atoms describing the environment), and can be used as a preprocessing stage (prior to planning). For a sufficiently large k, all dead-ends are captured. The resulting traps can be used with any search algorithm by pruning any state that satisfies the formula [Lipovet-

zky *et al.*, 2016].

As hinted above, global information such as dead-end traps can be crucial for solving an APP, as it requires much more than solving each local planning problem in it (or otherwise its complexity would be PSPACE). The fact is that a solution plan σ_1 for the planning problem $\mathcal{P}_1$ encoded in given transition of the APP may not be part of a complete solution for the APP, because it precludes, for example, solutions for a subsequent planning problem $\mathcal{P}_2$ in the network. The insight here is that for that to happen the world state produced by executing plan σ_1 has to be a dead-end state for the goal of $\mathcal{P}_2$. Let us show this with an example.

Example 1. Suppose that, in our academic example, the agent successfully achieves the goals of being at work, then at the pub by driving his/her car, and having some drinks there. Doing so, however, will later preclude his/her meeting the transition from program state p_2 to program state p_0: the agent is law-abiding and it is prohibited to drive intoxicated, so driving home from the pub is not acceptable and the agent will not be able to take the car home as required. This means that the agent should be smart enough to either go to work by public transport (and leave the car home), or drive the car back home in case s/he decides to go to the pub for a couple of drinks. □

In this example, if the agent goes to work and then to the pub by car and drinks, the agent becomes "trapped" in a dead-end state with respect to a subsequent goal, namely, being at home with the car at home. In other words, the agent *must* avoid "dead-end traps," i.e., being intoxicated while not having his/her car at home, to have any chance to solve the APP.

Theorem 1. *Let $\mathcal{A} = \langle L, V, v_0, s_0, \delta\rangle$ be an APP over a planning domain $\mathcal{D}$ and σ be a partial policy for $\mathcal{A}$ such that:*

- *there exists a sequence of transitions $v_0 \xrightarrow{g_1} \cdots \xrightarrow{g_k} v_k \xrightarrow{g_{k+1}} \cdots \xrightarrow{g_{n-1}} v_{n-1} \xrightarrow{g_n} v_n$ in $\mathcal{A}$ for some $n \geq 1$;*
- *there exists a sequence of world states $s_0, \ldots, s_k$ ($1 \leq k \leq n$) such that for all $i \in \{0, \ldots, k-1\}$:* (i) *g_{i+1}'s guard condition holds in state s_i; and* (ii) *s_{i+1} is a possible resulting world state after execution of plan $\sigma(v_i, s_i, g_{i+1})$; and*
- *there exists a world state s^* in the state trajectory of $\sigma(v_{k-1}, s_{k-1}, g_k)$ that is a dead-end state for goal g_n.*

Then, σ cannot be part of any realization for $\mathcal{A}$ in $\mathcal{D}$.

Proof (Sketch). Consider a sequence of world states $s_k, s_{k+1}, \ldots, s_{n-1}$ such that for all $i \in \{k, \ldots, n-1\}$: *(i)* guard condition of g_{i+1} holds true in s_i; and *(ii)* s_{i+1} is a possible resulting world state after execution of plan $\sigma(v_i, s_i, g_{i+1})$. Now, because world state s^* is a dead-end state for g_n, and $s_k, s_{k+1}, \ldots, s_{n-1}$ represents a legal evolution of $\mathcal{A}$ (for some sequence of legally executed operators), it is not hard to see that s_i is a dead-end state for g_n, for all $i \in \{k, \ldots, n-1\}$. So s_{n-1} is a dead-end state for g_n, and hence a planning problem $\mathcal{P}'$, where s_{n-1} and g_n are the initial and goal world states, admits no solution. This implies that for $\sigma(v_{n-1}, s_{n-1}, g_n)$ there does not exist any plan achieving goal g_n, from where it can be proved that the partial policy σ cannot be part of a realization for $\mathcal{A}$. □

Algorithm 1 Online APP realization with dead-end filtering.

Input: APP $\mathcal{A}$=$\langle P, V, v_0, \delta\rangle$, max steps M, initial world state s_0

```
1: v ← v0; s ← s0; steps ← 0
2: while ⟨v, ⟨γ, ψ, φ⟩, v'⟩∈δ such that s ⊨ γ and steps < M do
3:     select t = ⟨v, ⟨γ, ψ, φ⟩, v'⟩ such that γ holds in s
4:     ψ ← ψ∪ ¬Traps(A, v', s, {})
5:     π ← Plan(s, φ, ψ)
6:     if undefined π then
7:         return failure
8:     end if
9:     s ← apply(s, π); v ← v'; steps ← steps + 1
10: end while
11: return success

12: function TRAPS(A, v, s, visited)
13:     traps ← {}
14:     for each t = ⟨v, ⟨γ, ψ, φ⟩, v'⟩ & t ∉ visited do
15:         visited ← visited ∪ {t}
16:         traps ← traps∪FindTraps(s,φ)∪Traps(A, v', s, visited)
17:     end for
18:     return traps
19: end function
```

So, if a given strategy "falls" into a dead-end s^* of *any* potential possible goal g_n, then such a strategy cannot be a solution for the APP of concern. However, the reasoner is aware of the agent possible future goals—the APP is known a priori—so one should be able to prevent falling into such traps. In our example, knowing that his/her car cannot be in other location than home before the agent starts having drinks forces the agent to make "smart" plans, for example, driving his/her car back home before going to pub for drinks. In other words, dead-end states contain atoms $\langle$`(had agent drinks)` and `(at car work)`$\rangle$, or $\langle$`(had agent drinks)` and `(at car pub)`$\rangle$. Technically speaking, *prior to planning towards the currently selected goal*, the agent has to determine dead-end states with respect to his/her possible future goals. These dead-end states have to be avoided during the planning process.

Ideally, if the domain model involved in an APP have no operators with irreversible effects then, in principle, every solution plan for a transition could be part of a realization. In contrast, and as expected, it is precisely on instances with dead-ends that the planning-based approach proposed by Gerevini *et al.* [2011] has major difficulties, as demonstrated empirically by De Giacomo *et al.* [2016]. The reason is that the APP solver will experience higher backtracking (at the APP level) in the presence of dead-ends, as it discovers that a plan found for a previous goal G_1 precludes solutions for a future goal G_2. This suggests that it could be highly beneficial to be able to account for dead-end states of G_2 *while achieving goal* G_1. By doing so, we aim to reason about future goals in each local planning problem, hence encoding global constraints as local ones. This is indeed the core idea of our proposal.

3.1 Online Realization of APPs

In contrast to the aforementioned (off-line) approach that precomputes plans for all transitions in APPs, the *online approach* alternates planning for the next goal (transition) and

execution of the plan. That is, starting in the initial program state and the initial world state, the online approach selects one applicable transition, plans towards the transition goal, and finally executes such a plan and moves to the next program state.

Definition 2. An ***online realization*** of an APP $\mathcal{A} = \langle P, V, v_0, s_0, \delta \rangle$ over a planning domain $\mathcal{D}$ is a sequence $(s^0, v^0)t^1\pi^1(s^1, v^1)t^2\pi^2 \cdots (s^{n-1}, v^{n-1})t^n\pi^n(s^n, v^n)$, for $n \geq 0$, such that:

- $s^0 = s_0$ – the execution begins in the initial world state;
- for each $i \leq n$, $t^i = v^{i-1} \xrightarrow{\gamma^i : \psi^i, \phi^i} v^i$ in $\mathcal{A}$ such that $s^{i-1} \models \gamma^i$ (i.e., the transition is applicable), and $v^0 = v_0$
- π^i is a solution plan for problem $\langle \mathcal{D}, s^{i-1}, \phi^i \rangle$ such that application of π^i in s^{i-1} results in the world state s^i (s.t. $s^i \models \phi^i$) while ψ^i holds hold for every world state in the state trajectory of π^i.

Observe that only *local* requirements are imposed on the action sequence in each plan π^i: it should bring about the goals of the *current* transition t^i. This implies that lookahead reasoning may be necessary at each program transition, since the plan deployed may preclude the realization of future APP transitions. Thus, unlike offline realizations, an online realization may get *stuck*. In our example, if the agent gets intoxicated after going to the pub by car, then s/he cannot bring his/her car back home after that.

Robustness of online realizations of APPs can be (considerably) improved by incorporating a "global" dead-end reasoning technique that prior to the planning phase identifies dead-end traps for all possible future goals and encodes them as maintenance formulas. Algorithm 1 shows how the "online realizing" loop (Lines 2–10) is enhanced by identifying dead-end traps for potentially future goals. Notice that selection of t (Line 3) can be done externally (by a user). The function *Traps* identifies dead-end traps for all potential future goals from a given program state v. Each transition $t = \langle v, \langle \gamma, \psi, \phi \rangle, v' \rangle$ that has not yet been analyzed is a subject for i) identifying dead-end traps with respect to the goal ϕ from state s (the FindTraps function) and ii) identifying dead-end traps for future goals that can be reached from the program state v' (a recursive call of the Traps function). These dead-end traps are then combined (Line 16). The k-DNF formula representing the dead-end traps for all future goals is negated and added to the maintenance formula ψ (Line 4), so the corresponding dead-end states are avoided during the planning process (Line 5). Clearly, if the planning phase fails to find a plan, then the realization of $\mathcal{A}$ fails. On the other hand, if a "leaf" program state is reached or the maximum number of iterations has been performed, then the realization of $\mathcal{A}$ is successful. It should be noted that specifying the maximum number of iterations M is for "escaping" loops in APPs, i.e., the online realization terminates after M planning episodes, otherwise the online realization of APPs might be infinite. Alternatively, the terminating condition can be specified differently (e.g., the agent in Example 1 has to return home 5x).

		Floortile	Logistics	Glued-Bw	Matching-Bw	Airport	Woodworking	Total
Online	Naive (LPG)	4	0	0	0	15	0	19
	Naive (DFS+)	8	1	1	1	17	3	31
	Naive (LmCut)	8	16	0	0	17	2	43
	Traps (DFS+)	16	4	20	20	13	20	93
	Traps (LmCut)	18	19	20	18	13	20	108
Offline	Plain (LPG)	10	7	1	3	16	14	51
	Plain (DFS+)	8	1	1	1	17	8	36
	Plain (LmCut)	9	15	1	2	16	11	54
	Traps (DFS+)	17	4	20	12	13	13	79
	Traps (LmCut)	19	19	20	11	13	9	91

Table 1: The number of solved APPs of the naive and "dead-end traps" enhanced online approaches as well as plain and "dead-end traps" enhanced offline approaches.

3.2 Unsolvability of APPs

Although for online realization we need only a partial policy that corresponds with the selected path in an APP, we assume that an APP is solvable if and only if there exists a realization for all possible paths in the APP.

A (classical) planning problem is unsolvable if and only if the initial world state is a dead-end state with respect to the goal. For APPs, if the initial world state is a dead-end state for at least one future goal, then the APP is unsolvable (as formalized in the following proposition).

Proposition 1. *Let $\mathcal{A}$ be an APP, v_0 be the initial program state, and s_0 be the initial world state. If s_0 is a dead-end state with respect to ϕ for some transition $\langle v, \langle \gamma, \psi, \phi \rangle, v' \rangle$ such that v is reachable from v_0, then $\mathcal{A}$ is unsolvable.*

The proposition provides an indication that our approach can in some cases determine unsolvability of APPs before any planning episode (the Plan function as in Line 5 will trivially fail to find a plan if the initial world state belongs to a trap).

Notice that the opposite implication of the proposition does not hold. Dead-end states might also occur with respect to sequences of goals. For example, driving to certain destinations requires some amount of fuel, for each destination we obviously cannot have less fuel than required; however, if we need to visit these destinations in sequence we need to have enough fuel for the whole trip.

4 Experimental Evaluation

The purpose of the experimental evaluation is to demonstrate that reasoning about non-local (or global) dead-ends is important and improve performance (in terms of solved APPs) of both offline and online approaches for realizing APPs.

4.1 Domains

In the *Airport* domain, a planner has to control the ground traffic at the airport, i.e., navigating planes to gates or runways. Here, the APPs consist of sequences of goals, where

each of them is either to park a plane at a certain gate, or navigate a plane to take off. It might happen that some parked plane blocks another plane going to the runway.

In the *Floortile* domain, a set of robots paints floor tiles to either black or white. Robots can move only on non-painted tiles. Here, we consider two types of APPs. The first type consists of sequences of goals, where each goal is to paint a line of tiles (in the grid) with selected colors (per tile). In the second type, tiles are painted consecutively, where each tile is painted with a color selected just before painting it. Painting the tiles in a wrong order, or "trapping" robots in a "surrounded" area (by painted tiles) might cause dead-ends.

"Glued" BlocksWorld is a variant of the well known BlocksWorld domain [Slaney and Thiébaux, 2001], where after a block is stacked on another block, it cannot be unstacked. Here, the APPs consist of sequences of goals that represent the elementary steps to be taken to build a stack of blocks. We consider two types of APPs, where elementary goals are ordered correctly (from bottom to top), or randomly. Notice that if blocks are "wrongly" stacked, i.e., if the stack is being built from the "middle" some blocks will become inaccessible (lie beneath the "stacked" blocks).

Matching BlocksWorld is another variant of BlocksWorld, where blocks have either positive or negative polarity and there are two robotic hands, one with positive and one with negative polarity. When a robotic hand with an opposite polarity is used to carry a block, the block will become damaged and no other block can be stacked on top of it. Here, the APPs consist of loops of various configurations of blocks.

In the *Logistics* domain, packages have to be transported by planes between cities and by trucks within cities. The APPs consider three cities (A, B and C) such that each city is reachable from A and B but no city is reachable from C (i.e., planes are "trapped" in C). We have two types of APPs. In the first type, we initially require some packages to be delivered to city C, then we have a loop where we deliver the remaining packages between cities A and B. In the second type, we consider a sequence of goals where each goal is to deliver several packages into random cities but never from C to either A or B. Planes are initially in A or B.

The *Woodworking* domain simulates the works in a woodworking workshop, where wood parts have to be cut, treated and colored as required. Here, the APPs represent a manufacturing process of wood pieces such that pieces are manufactured one by one while having three options, each consisting of two "requirement" goals, of how a piece can be manufactured. Noticeably, some "requirements" might preclude each other and for fulfilling one requirement we might need to take decisions influencing other possible requirements.

4.2 Settings

For each domain, we have generated 20 APPs as we believe that this number is representative to determine relative performance of the techniques in particular domains. Also, for all APPs, guards are always true, so any transition can be chosen from the current program state (in our experiments, we do so randomly in order to simulate external user's decisions). We have chosen LPG [Gerevini *et al.*, 2003], Lm-cut [Helmert and Domshlak, 2009] and DFS+ [Lipovetzky and Geffner, 2014] as benchmark planners, since they incorporate different high performance planning techniques. Moreover, Lm-cut guarantees optimal solutions (minimum plan length or total action cost). Lm-cut and DFS+ are available with the "trapper" tool since both are integrated in the LAPKT toolkit [Ramirez *et al.*, 2015] in which "trapper" is implemented (notice that LPG is not implemented in the LAPKT toolkit). Furthermore, LPG was already used to showcase the high performance of classical planners in offline APPs realization [De Giacomo *et al.*, 2016]. We compute 2-DNF dead-end traps since they provide reasonable "performance/cost" ratio, that is, the number of identified traps vs time spent on finding them, as observed by Lipovetzky *et al.* [2016]. The $maxsteps$ parameter was set to "infinity" for acyclic APPs (the realization is successful if a leaf program state is reached), i.e, for Airport, Floortile, "Glued"-Bw and Woodworking, and to 10 for APPs with cycles, i.e., Logistics and Matching-Bw, in order to assure that every cycle has been performed at least twice.

Since LPG is a randomized planner and thus with different runs solution plans might differ even for the same planning problem, we performed 20 runs for each APP. In Woodworking, where the structure of APPs allows different realizations (depending on chosen branches), 20 runs for each APP were performed for all configurations (including LPG). Notice that we consider an APP as solved if all 20 runs were successful – this provides a reasonable confidence for determining a successful online realization of an APP.

With regards to the offline method, we have used a "hybrid" approach, that is, using LPG when planning to preferred states (these are required, for example, for "closing" loops in cyclic APPs). The reason is that the Lm-cut and DFS+ planners, integrated in the LAPKT toolkit, do not fully support reasoning with preferred states. Consequently, no global dead-end reasoning is applied for such planning episodes, since LPG is not implemented in the LAPKT toolkit in which the "trapper" tool is implemented.

All the experiments were run on i7-7700 3.6 Ghz CPU with 16GB of RAM. Notice that a time limit of three hours (10800 seconds) was applied.

4.3 Results

Table 1 shows the number of solved APPs (out of 20 instances) per each domain and planner by i) the naive online approach, and ii) the "dead-end traps" enhanced online approach, iii) the state-of-the art offline approach [De Giacomo *et al.*, 2016], and iv) the "dead-end traps" enhanced offline approach. The results clearly indicate that reasoning about non-local dead-ends increases the overall robustness, i.e., the number of solved APPs is higher. Remarkably, despite no theoretical guarantee of identifying all dead-end states the results indicate a high success rate of the non-local dead-end reasoning enhanced approaches.

In the online approach, the impact of non-local dead-end reasoning could be better isolated. In other words, better coverage (i.e., more solved APPs) is due to avoiding non-local dead-ends, since the naive online approach fails when a current planning problem is unsolvable (i.e., while solving any of the previous planning problem a dead-end state for

the current problem has been reached). However, non-local dead-end reasoning comes with a "tax" of additional computational time. Although finding 2-DNF dead-end traps is of quadratic complexity with respect to the number of atoms describing the world states, when the number of atoms is very high the "tax" of non-local dead-end reasoning can be very high too. That was manifested in the Airport domain, where more complex APPs had 100000s atoms, resulting in failing to solve them in the given time limit (while these APPs were solved by the naive approach). Noticeably. the time limit was also exceeded for two APPs in Matching-bw for Lm-cut since some of the planning problems were too complex for the Lm-cut planner.

The "plain" offline approach [De Giacomo *et al.*, 2016] can benefit from backtracking, that is, generating a different plan for a previous planning problem if the current problem has shown to be unsolvable. When compared to the naive online approach, the performance of the plain offline approach improves in the Woodworking domain and/or when the LPG planner is used. In Woodworking, we do not have to backtrack "far" and, therefore, the chance for recovering from non-local dead-ends is higher than in the other domains. LPG usually generates plans faster than Lm-cut and DFS+ and thus it has better chance to eventually escape non-local dead-ends. Non-local dead-end reasoning, unsurprisingly, improves the performance of the offline approach. On the other hand, the total number of solved APPs is less than for the "traps" online approach because of relatively poor performance of the "traps" offline approach in Matching-Bw and Woodworking. In those domains, the offline approach has to reason with preferred states and since we use the "hybrid" approach (as mentioned above) we cannot fully exploit the "trapper" tool. Also, in Woodworking, the offline approach has to compute a policy for every possible path in an APP, which, clearly, is much more time consuming than in the online case where just one path has to be explored.

Considering runtime, i.e., how long it takes for a particular technique to realize an APP, the naive (or plain) approach usually outperforms the non-local dead-end reasoning enhanced approach in APPs where both approaches are successful. This is, however, not very surprising, since reasoning about non-local dead-ends in such cases just costs time as we elaborated earlier. The offline approach is slower than the online approach in APPs with branching (e.g., Woodworking) since it has to consider all alternatives.

Particular planning problems vary from very easy, that is, plans consist of a few actions, to more complex, that is, plans consist of tens of actions. Since Lm-cut is an optimal planner, plans generated by it were often shorter than those produced by DFS+ (although in terms of runtime, DFS+ unsurprisingly outperformed Lm-cut). It should be noted that Lm-cut can guarantee optimality of plans with respect to non-local dead-ends – these plans might be longer than in a "standard" case in order to avoid dead-end states for possible future goals. For example, in Glued-Bw we might want to achieve goals `(on a b)` and `(on b c)` in this order from the initial world state, where all blocks are on the table. In the naive approach, the block `a` will just be stacked on the block `b` making the subsequent goal `(on b c)` unachievable. With non-local dead-end reasoning (e.g. identifying dead-end traps) it can be found out that for keeping `(on b c)` achievable nothing has to be stacked on `b` unless it is already on `c`. Hence, the "traps" approach when achieving `(on a b)` would stack `b` on `c` and then `a` to `b` (even if an optimal planner is used). Then `(on b c)` is trivially achieved by an empty plan.

5 Conclusions

The overarching contribution of this work lies on bringing research from classical planning on local dead-ends into a higher formalism of APPs to handle global (or non-local) dead-ends. Concretely, we devised a way to handle *non-local* dead-ends, which are the main cause behind the poor performance of current techniques [De Giacomo *et al.*, 2016]. To that end, we leveraged on recent work on reasoning about dead-ends in planning [Lipovetzky *et al.*, 2016] and encoded global dead-ends into each local planning steps. We showed (Theorem 1) that doing so is always correct. In addition, we proposed an alternative *online* execution model for APPs (Algorithm 1) that is arguably more realistic, and for which avoiding non-local dead-ends is crucial, as the executor cannot resort to meta-level backtracking. The experiments reported, which involve a variety of well-known planning domains from the International Planning Competition, demonstrate that reasoning about global (or non-local) dead-ends has, in terms of the number of solved APPs, a considerably positive impact on both *(i)* the state-of-the-art offline approach [De Giacomo *et al.*, 2016] as well as *(ii)* the online approach. Hence, the results demonstrated that identifying non-local dead-ends prior to the planning episode can, often considerably, increase robustness of the approach for online realization of APPs, which is naturally prone to getting "trapped" in non-local dead-ends.

Agent Planning Programs appear as a promising principled elaboration of planning that accounts for features required in many real-world control scenarios, such as sequential goals, external decisions, and repeating tasks. Also, APPs can serve as a source of challenging benchmarks for the planning community, while the agents community can benefit from the topic of dead-end detection. For future work, we would like to investigate *theoretical guarantees* of k-DNF traps with respect to given domains to improve the robustness of the online approach for realizing APPs. We are also interested in exploiting APPs with global dead-end reasoning to solve (classical) planning problems, by decomposing them in sub-goals, as done by Vernhes *et al.* [2013], but mitigating backtracking when some subsequent goals become not achievable.

Acknowledgements

We thank the reviewers for their suggestions and acknowledge the support of the Czech Science Foundation (project no. 17-17125Y), the Marie Curie project "Semdata", the Australian Research Council (DP120100332) and DST.

References

[Bloem *et al.*, 2012] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis

of reactive(1) designs. *Journal of Computer and System Sciences*, 78(3):911–938, 2012.

[Dal Lago *et al.*, 2002] Ugo Dal Lago, Marco Pistore, and Paolo Traverso. Planning with a language for extended goals. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 447–454, 2002.

[De Giacomo *et al.*, 2010] Giuseppe De Giacomo, Fabio Patrizi, and Sebastian Sardina. Agent programming via planning programs. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 491–498, 2010.

[De Giacomo *et al.*, 2016] Giuseppe De Giacomo, Alfonso Gerevini, Fabio Patrizi, Alessandro Saetti, and Sebastian Sardina. Agent planning programs. *Artificial Intelligence*, 231:64–106, 2016.

[Edelkamp, 2006] Stefan Edelkamp. On the compilation of plan constraints and preferences. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 374–377, 2006.

[Gerevini *et al.*, 2003] Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. Planning through stochastic local search and temporal action graphs. *Journal of Artificial Intelligence Research (JAIR)*, 20:239 – 290, 2003.

[Gerevini *et al.*, 2011] Alfonso Gerevini, Fabio Patrizi, and Alessandro Saetti. An effective approach to realizing planning programs. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 323–326, 2011.

[Ghallab *et al.*, 2004] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning, theory and practice*. Morgan Kaufmann, 2004.

[Ghallab *et al.*, 2014] Malik Ghallab, Dana S. Nau, and Paolo Traverso. The actor's view of automated planning and acting: A position paper. *Artificial Intelligence*, 208:1–17, 2014.

[Helmert and Domshlak, 2009] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 162–129, 2009.

[Lipovetzky and Geffner, 2014] Nir Lipovetzky and Héctor Geffner. Width-based algorithms for classical planning: New results. In *Proceedings of the European Conference in Artificial Intelligence (ECAI)*, pages 88–90, 2014.

[Lipovetzky *et al.*, 2016] Nir Lipovetzky, Christian J. Muise, and Hector Geffner. Traps, invariants, and dead-ends. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 211–215, 2016.

[Pnueli and Rosner, 1989] Amir Pnueli and Roni Rosner. On the synthesis of an asynchronous reactive module. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, pages 652–671, 1989.

[Ramirez *et al.*, 2015] Miquel Ramirez, Nir Lipovetzky, and Christian Muise. Lightweight Automated Planning ToolKiT. http://lapkt.org/, 2015. Accessed: 2016-11-1.

[Shivashankar *et al.*, 2012] Vikas Shivashankar, Ugur Kuter, Dana S. Nau, and Ronald Alford. A hierarchical goal-based formalism and algorithm for single-agent planning. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 981–988, 2012.

[Slaney and Thiébaux, 2001] John K. Slaney and Sylvie Thiébaux. Blocks world revisited. *Artificial Intelligence*, 125(1-2):119–153, 2001.

[Vernhes *et al.*, 2013] Simon Vernhes, Guillaume Infantes, and Vincent Vidal. Problem splitting using heuristic search in landmark orderings. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2401–2407, 2013.