Artificial Intelligence 284 (2020) 103271

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

CPCES: A planning framework to solve conformant planning problems through a counterexample guided refinement ☆

Alban Grastien [a,*], Enrico Scala [b,a]

[a] The Australian National University, Canberra, Australia
[b] University of Brescia, Brescia, Italy

ARTICLE INFO

Article history:
Received 8 April 2019
Received in revised form 18 December 2019
Accepted 21 March 2020
Available online 26 March 2020

Keywords:
Conformant planning
Classical planning
Propositional satisfiability

ABSTRACT

We introduce CPCES, a novel planner for the problem of deterministic conformant planning. CPCES solves the problem by producing candidate plans based on a sample of the initial belief state, searching for counter-examples to these plans, and assigning these counter-examples to the sample, until a valid plan has been produced or the problem has been proved unfeasible. On top of providing a means to compute a conformant plan, the sample can also be understood as a justification for the plan being found, or relevant reasons why a plan cannot be found. We study the theoretical properties that CPCES enjoys—correctness, completeness, and optimality—and how the several variants of CPCES we describe differ in behaviour. Moreover, we establish a theoretical connection between the CPCES framework and well-known concepts from the literature such as tags and width. With this connection we prove the worst case complexity for some variants of CPCES.

Finally, we show how CPCES can be used in a more incremental fashion by learning sequencing of actions from the previous plan being found. Such a technique mimics the use of macro-operators, widely used in automated planning to speedup resolution.

Our theoretical analysis is accompanied with a thorough experimental evaluation of the (many) possible incarnations of CPCES. This not only proves our theoretical findings from an empirical perspective, but also shows that CPCES is able to handle problems that have been traditionally hard to solve by the existing conformant planners, whilst remaining competitive over "easier" conformant planning problems. Importantly, CPCES is able to prove many unsolvable conformant planning problems as such, extending substantially the reach of conformant planners.

Contents

☆ This work was performed in part while A. Grastien was working for Data61 and E. Scala for FBK.

* Corresponding author.
E-mail addresses: alban.grastien@anu.edu.au (A. Grastien), enricos83@gmail.com (E. Scala).

https://doi.org/10.1016/j.artint.2020.103271
0004-3702/

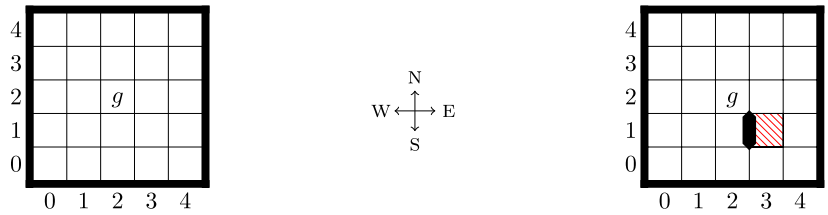
1. Introduction

Conformant Planning is the problem of finding a sequence of actions that leads an agent to some goal with the following two assumptions: the plan needs to be robust to uncertainty; the plan execution is not observable. We restrict ourselves to the problem of *deterministic* conformant planning, where the uncertainty is only on the initial state of the agent and not on the effect of the actions. Conformant planning is useful in contexts where the agent's state or its environment is partially unknown, and the agent is either blind or unable to process its sensors input. Conformant planning problems are the targeted computational problem of more powerful planning formulations too, such as probabilistic conformant planning [43], and finite-state controllers computation [8]. Therefore, much as classical planning has benefited conformant planning, we see any advance to conformant planning as a direct improvement to yet more complex planning formulations.

1.1. Example

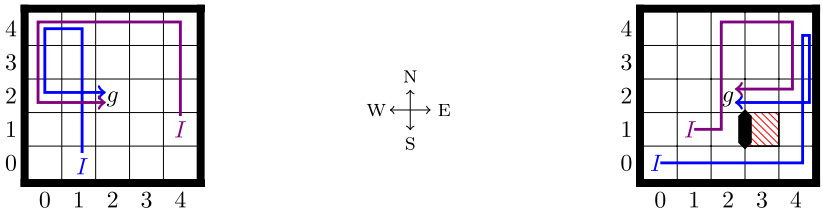
We introduce an example, called Grid World, that we use heavily in the rest of the paper. We consider two variants of the same problem, both presented in Fig. 1. The problem involves a robot with an unknown initial position. The robot needs to travel to the center of a grid (represented with g) and can move in all four cardinal directions: N, S, W, and E. When the robot hits a wall, it stays in its current location. The red stripes represent a swamp, a location that the robot cannot leave. It is assumed that the robot does not start in the swamp. Notice that in this example the goal state is unique, but this is not the case in general.

One solution to the problem of Fig. 1.a consists in moving four times north, four times west, twice south, and twice east (N*4 W*4 S*2 E*2). After the first eight moves, the robot is known to be in the top left corner, from where it can easily move to the center of the map. The plan execution is illustrated on two initial states in Fig. 2.a.



a. Without a swamp. b. With a swamp.

Fig. 1. Two deterministic conformant planning problems: the initial location of the robot is unknown; it can move in four directions and is stopped by walls; it cannot escape the swamp (red stripes); it needs to end in the center location.



a. Without a swamp: N*4 W*4 S*2 E*2. b. With a swamp: E*4 N*4 E*2 S*2 W*2.

Fig. 2. Examples of executions of plans for the problems of Fig. 1: for some of the initial states (represented by an I), the lines show the path taken by the agent when applying the plan from this state.

The problem of Fig. 1.b is slightly more difficult because the swamp needs to be avoided. More fundamentally, this example includes two features that make this problem harder than that of Fig. 1.a. First, the swamp introduces a connection between the horizontal and vertical coordinates of the robot (for instance, the robot can move north from position $(2, 1)$ but will fail to do so from position $(3, 1)$ as it is then stuck in the swamp). As we will see, this complexity finds a mathematical explanation in the notion of the *width* [34]. Roughly, the width is the number of initially unknown variables whose values are interconnected; the width evaluates to one in Fig. 1.a and two in Fig. 1.b. We discuss in Section 4 how this measure can be captured in general. Second, this example contains dead-ends, i.e., applicable partial plans that cannot be extended into valid plans. The presence of dead-ends notoriously makes planning problems much harder because the agent cannot greedily apply the actions at hand [30]; repairing the plan becomes much more difficult. One of the shortest valid plans for Fig. 1.b is E*4 N*4 E*2 S*2 W*2. The plan execution is illustrated on two initial states in Fig. 2.b.

1.2. Issues with conformant planning

Computationally speaking, conformant planning is harder than *classical planning*, the variant in which there is no uncertainty. It is also much more difficult to solve in practice. Indeed, much as classical planning, conformant planning can be viewed as a path finding problem [6]: each vertex of the search space corresponds to a set of states that represents the possible states that the system could be in (aka a *belief state*) and each edge represents the effect on the belief of applying an action. For a plan to be valid, the belief upon applying the plan should entail the goal condition. There are complicating factors to this formulation however. The number of belief states is exponential in the number of system states, which implies a much larger search space and, potentially, much longer plans. Each belief state is also generally expensive to maintain as it includes exponentially many states. Finally, as conformant plans have often to be found in very narrow solution spaces (a conformant plan needs to work for *all* initial states), heuristics approaches that work well in classical planning problems often struggle in such a class of problem.

In order to handle the last two issues mentioned above we propose to not consider the whole initial belief state but, instead, to compute a plan that is valid only for a small *sample* of this belief state, not unlike the use of *basis* [2]. This decision addresses the problem of representing the belief state since it is now limited in size. Furthermore, since the sample is small (in the hundreds) the problem of finding a plan valid for this sample can be practically reformulated into classical planning [34], which gives us access to powerful heuristics.

There remains the problem of finding the right sample. If we compute a plan based on an improper sample, this plan will likely be invalid for the complete belief state. We use all the flawed candidate plans to choose the sample: by adding to the sample and for each plan at least one initial state for which this plan is invalid (aka, a *counter-example*), we guarantee that the sample will become better in the sense that the same invalid plan will not be generated.

This procedure is formalised in CPCES, the family of conformant planners that we introduce in this paper. CPCES involves two procedures that are intertwined. First, given a sample of initial states, a candidate plan is computed; if no such plan is found, the planner concludes that there is no conformant plan. Notice that the ability of determining that a conformant planning problem is unsolvable is a rare feature for conformant planners. Second, the sample is updated so that it includes

a counter-example to each plan; if no such sample is found, the planner concludes that the last plan is valid. The algorithm iterates over these two procedures until a decision was made.

This work relies on the assumption that small samples can be found that cover appropriately the initial belief state. Our empirical validation shows that, for the existing benchmarks, this assumption is mostly valid. As a side benefit, CPCES is able to return this sample, which can be used as a justification for the conformant plan: indeed every invalid plan that was considered by the planner at some point is rejected by at least one state of this sample. Therefore, it is likely that this sample mentions all the important contingencies of this planning problem. If an external operator (say, a human decision maker) proposes a different, more efficient plan, the sample probably includes a counter-example to this proposed plan. Our work also produces a counter-example generator which can be used to test the validity of a plan.

1.3. Contributions

This paper describes the CPCES family and discusses its properties. We demonstrate that CPCES is sound and complete. We present several variants of CPCES that differ on how the sample is generated and how the candidate plan is produced. We also show how CPCES is connected to the notions of *width* and *tag* [34]. We provide experimental results that show that CPCES scales much better than existing planners on problems with non-trivial width or dead-ends.

The present paper is based on our previous work published in IJCAI-17 [19] and ICAPS-18 [20]. We here provide a more comprehensive theoretical analysis of CPCES (including complete proofs). As a spin-off of the CPCES framework, we also introduce the learning mechanism that accelerates the production of new candidate plans. Our experimental evaluation is deeper than the ones presented in previous works. First, we evaluate the learning process of CPCES with the various configurations previous studied. Then we extend our comparative analysis from T1 [2] to two other excellent conformant planners, namely CG-LAMA [33] and DNF [44]. Finally, our experiments also consider and study CPCES paired with MADAGASCAR [39] as an alternative classical planner.

1.4. Outline

This paper is organised as follows. Section 2 formally defines the Conformant Planning Problem. Section 3 presents the existing approaches from the scientific literature. Section 4 discusses the structural properties of conformant planning problems, and in particular the notion of tags [34] and bases [2]; these notions are very important to explain and understand the behaviour of our planner. In Section 5 we show how planning problems can be reformulated so that the conformant planning problem is transformed into classical planning; the discussion of this section is also useful to understand how our planner works. Section 6 presents CPCES along with the variants we considered and the theoretical properties that they enjoy. Section 7 and Section 8 then explain the two subroutines that CPCES builds on, namely the production of candidate plans and the generation of counter-examples. Section 9 shows how CPCES can learn macros to facilitate the production of candidate plans. Finally, Section 10 presents our experimental evaluation.

2. Problem definition

In this section we provide a formal definition of the problem of deterministic conformant planning and its semantics. We use the example of the previous section to illustrate the concepts introduced here.

Given a state variable $v \in \mathcal{V}$, we write D_v its *domain*. An assignment of v is a pair $\langle v, \ell \rangle$ where ℓ is a value of its domain. Given a set of variables $V \subseteq \mathcal{V}$ with a domain D_v for each variable $v \in V$, we write $Asgmt(V)$ the set of assignments of variables in V. Assignments are to be interpreted as propositional variables, and we write $Forms(V)$ the set of propositional formulas over the set $Asgmt(V)$. A state is the representation of a given situation for the agent.

Definition 1. Given the set $\mathcal{V}$ of state variables, a *state* is a function s that assigns each variable v with a value $s(v) \in D_v$ from its domain. The state is also interpreted as a conjunction of assignments, or as a set of assignments. We write S the set of states of $\mathcal{V}$.

When the system's state is not known precisely, the minimal set of states that we are sure contains the system state is called the *belief state*. A formula $\phi \in Forms(\mathcal{V})$ over the assignments of $\mathcal{V}$ implicitly represents the set of states $states(\phi)$ defined as the states that satisfy the formula: $states(\phi) = \{s \in S \mid s \models \phi\}$. Equivalent formulas represent the same set of states. Nevertheless, with slight abuse of notation, we write $\Phi_{S'}$ to denote one of the formulas that represent S', i.e., $\Phi_{S'}$ is defined such that $states(\Phi_{S'}) = S'$. In particular Φ_s, the formula associated with state s, is the conjunction of assignments $\bigwedge_{v \in \mathcal{V}} (v = s(v))$. In order to simplify the discussion we sometimes use sets of states or their corresponding formula interchangeably.

Example. In the example of Fig. 1.a, a state is a function s such that $s(x) = i$ and $s(y) = j$ where $i \in \{0, \ldots, 4\}$ and $j \in \{0, \ldots, 4\}$; for this example, we denote with $s_{i,j}$ the state that associates variable x with i and variable y with j. $\diamond$

A state changes as an effect of actions; an action is a state transition which is under the control of the agent. The state transition amounts at a modification of the state variables of the state in which it is applied. Actions can be executed and implement some of their effects only when some condition is satisfied. Formally:

Definition 2. Given the set of state variables $\mathcal{V}$ and the set of *actions* A, a *precondition function* $prec : A \to Forms(\mathcal{V})$ is a function that associates each action with a formula. A *conditional effect* is a function $e : Asgmt(\mathcal{V}) \to Forms(\mathcal{V})$ that associates every assignment $\alpha = \langle v, \ell \rangle$ (aka *effect*) with a *condition* $e(\alpha)$ that indicates when the assignment becomes satisfied. The conditional effects of two distinct assignments of the same variable are inconsistent:

$$\forall v \in \mathcal{V}.\ \forall \{\ell, \ell'\} \in D_v.\quad \ell \neq \ell' \quad \Rightarrow \quad e(\langle v, \ell \rangle) \wedge e(\langle v, \ell' \rangle) \models \bot.$$

An *effect function* $eff : A \to (Asgmt(\mathcal{V}) \to Forms(\mathcal{V}))$ is a function that associates each action with a conditional effect. We call *trivial* a conditional effect whose condition is $\bot$.

Our notation of conditional effect is different from more classical ones. Rintanen [36], for instance, defines a conditional effect as a formula that includes elements such as $\varphi \rhd \alpha$ where φ is a condition and α is an effect on the state variables. This notation and ours are equivalent (it is possible to reformulate one into the other, at least for deterministic effects), but we chose Definition 2 because it makes the condition that assigned a variable to some value is easy to retrieve (it is simply $eff(\langle v, \ell \rangle)$).

We use the notation $a = \langle p, e \rangle$ as a shortcut for $p = prec(a)$ and $e = eff(a)$. Traditionally the semantics of an action (its precondition and its conditional effect) is defined as part of the action rather than in separate functions as we do here. This is because, in this paper, we discuss reformulations of planning problems in which the semantics of the actions are redefined. We want to be able to compare (in particular through the subset relation) the set of solutions of these problems. Therefore, we need the problems to be defined over the same actions, which explains the separation between the actions and their semantics.

Definition 3 *(Action Semantics).* An action $a = \langle p, e \rangle$ is *applicable* in state s if this state satisfies its precondition: $s \models p$. The *application* of action a in state s leads to the state $s[a]_{eff}$ that, for all state variables $v \in \mathcal{V}$, satisfies $s[a]_{eff}(v) = \ell$ iff

- either $s \models e(\langle v, \ell \rangle)$: the assignment of variable v is modified because the condition of an effect associated with v is satisfied;
- or $s(v) = \ell \ \wedge\ \forall \ell' \in D_v.\ s \not\models e(\langle v, \ell' \rangle)$: the assignment of variable v is not modified because the conditions of none of the effects associated with v are satisfied.

A *plan* is a sequence of actions. The *application* of plan $\pi = a_1 \dots a_k$ is the state $s[\pi]_{eff}$ defined by the iterative application of the actions in the plan: $s[\pi]_{eff} = s[a_1]_{eff} \dots [a_k]_{eff}$. The plan $\pi = a_1 \dots a_k$ is *applicable* in state s if every action is applicable:

$$\forall i \in \{1, \dots, k\}.\ s[a_1]_{eff} \dots [a_{i-1}]_{eff} \models p_i \quad \text{where} \quad a_i = \langle p_i, e_i \rangle.$$

For simplicity we assume that the application of a plan, and the sequence of states it leads to, is defined even if the plan is not applicable. When obvious from the context, the effect function *eff* is removed from $s[a]_{eff}$.

Example. In the example of Fig. 1.a, action N is defined as:

- $prec(\text{N}) = \top$
- $eff(\text{N}) = \begin{cases} \langle x, 0 \rangle \mapsto \bot, & \langle y, 0 \rangle \mapsto \bot \\ \langle x, 1 \rangle \mapsto \bot, & \langle y, 1 \rangle \mapsto y = 0 \\ \langle x, 2 \rangle \mapsto \bot, & \langle y, 2 \rangle \mapsto y = 1 \\ \langle x, 3 \rangle \mapsto \bot, & \langle y, 3 \rangle \mapsto y = 2 \\ \langle x, 4 \rangle \mapsto \bot, & \langle y, 4 \rangle \mapsto y = 3 \end{cases}$, which can be interpreted as:
  - N has no effect on x (the condition that would assign x to i if $\bot$);
  - N never assigns y to 0;
  - N assigns y to $i + 1$ whenever y previously evaluated to i.

In the example of Fig. 1.b, action N is defined as:

- $prec(\text{N}) = \top$
- $eff(\text{N}) = \begin{cases} \langle x, 0 \rangle \mapsto \bot, & \langle y, 0 \rangle \mapsto \bot \\ \langle x, 1 \rangle \mapsto \bot, & \langle y, 1 \rangle \mapsto y = 0 \\ \langle x, 2 \rangle \mapsto \bot, & \langle y, 2 \rangle \mapsto y = 1 \wedge \neg(x = 3) \\ \langle x, 3 \rangle \mapsto \bot, & \langle y, 3 \rangle \mapsto y = 2 \\ \langle x, 4 \rangle \mapsto \bot, & \langle y, 4 \rangle \mapsto y = 3. \end{cases}$

Coming back to the example of Fig. 1.a, consider the state $s = s_{1,1}$. Because $prec(\text{N})$ is $\top$, the action is applicable in s. For all $i \in D_x$, the condition $eff(\text{N})(\langle x, i\rangle)$ is unsatisfied in the state s; therefore the assignment of x is unchanged by N: $s[\text{N}](x) = s(x) = 1$. We notice however that $j = 2$ is such that $s \models eff(\text{N})(\langle y, j\rangle)$ (by Definition 2, there cannot be two such j values). Therefore $s[\text{N}](y) = j = 2$, and $s_{1,1}[\text{N}] = s_{1,2}$. ◇

Definition 4. A *(deterministic) conformant planning problem instance* is a tuple $P = \langle \mathcal{V}, A, prec, eff, \Phi_I, \Phi_G\rangle$ where

- $\mathcal{V}$ is the set of state variables;
- A is the set of actions;
- $prec : A \to Forms(V)$ is the precondition function;
- $eff : A \to Asgmt(\mathcal{V}) \to Forms(\mathcal{V})$ is the effect function;
- $\Phi_I \in Forms(\mathcal{V})$ is the *initial formula* that represents the initial belief; and
- $\Phi_G \in Forms(\mathcal{V})$ is the *goal formula*.

The conformant planning problem is then defined as the problem of finding a plan that is applicable in every initial state of the instance.

Definition 5 *(Validity of Plans).* A plan π is *valid* in state s if it is applicable and its application leads to a goal state: $s[\pi] \models \Phi_G$. A plan π is *valid* in a set of states if it is valid in each of its states. A plan π is *valid* for the planning problem instance $P = \langle \mathcal{V}, A, prec, eff, \Phi_I, \Phi_G\rangle$ if it is valid for its set of initial states. A valid plan is also called a *solution* of the planning problem. We write $\Pi(P)$ the set of valid plans for P. We say that two planning problems are *equivalent*, denoted by $P \equiv P'$, if their solution sets are equal: $\Pi(P) = \Pi(P')$.

Example. In the example of Fig. 1.a, the set I of possible initial states contains all the states S, which is represented by the initial formula $\Phi_I = \top$. The goal set (the states that we want the robot to end in) is the singleton $G = \{s_{2,2}\}$, and the goal is therefore $\Phi_G = (x = 2) \wedge (y = 2)$. Consider the plan $\pi_1 = \text{NE}$. This plan is valid for initial state $s_{1,1}$ but is invalid for any other initial state; therefore it is not a solution to the conformant problem. The plan $\pi_2 = \text{N}^*4\ \text{W}^*4\ \text{S}^*2\ \text{E}^*2$ is valid for all initial states, and is therefore a solution to the conformant problem. ◇

A conformant planning problem that includes only one initial state ($|states(\Phi_I)| = 1$) is called a *classical* planning problem. Classical planning problems have been extensively studied. They are fundamentally easier to solve (PSPACE-COMPLETE vs EXPSPACE-COMPLETE for conformant planning problems [22,5]). One of the most popular approaches to classical planning is heuristics search [7]; the motivation for its success is the existence of powerful relaxation-based heuristics through which planners can quickly explore enormous state spaces. Unfortunately, heuristics are much more difficult to define for conformant planning. Similarly to Palacios and Geffner [34], part of our motivation for this work was to reformulate conformant planning into classical planning in order to leverage from powerful heuristics that have been developed in that context.

3. Literature review

The problem of conformant planning was defined by Smith and Weld [42], and studied computationally as a shortest path problem over the set of belief states only a few years later by Bonet and Geffner [6]. This latter formulation allows us to better understand the underlying computational hardness. In fact, compared to a classical planning problem, which is often solved by search as well, the conformant planning case is much more involved, if only because the search space is much larger. For instance, if the number of states is n and if the number of initial states is m, then the search space for the classical planning problem is $O(n)$ against $O(n^m)$ for the conformant planning problem. From a computational complexity standpoint, in general, the task of deciding whether a solution exists is proved to be EXPSPACE-COMPLETE [22,5]. Despite this discouraging worst-case result, researchers have attacked conformant planning from different perspectives, obtaining systems that can practically work for problems involving a very large number of initial states, i.e., m is often in the thousands or more. In the following we survey what we believe are the main lines of research that have emerged, and have influenced our work.

3.1. Compact representation of the belief

One issue with searching over the belief space is the representation size of the belief states along any kind of search. In general, their size makes impractical an explicit enumeration. Instead, it is much more practical to represent them symbolically, i.e., with a propositional formula that evaluates to true for precisely all states that belong to the set. The challenge is then to understand which representation is the most suitable one: a good representation should be compact, easy to manipulate, and easy to query (for instance, it should be easy to verify whether all the states of the belief state satisfy a given action precondition), cf. the work from Darwiche et al. [16,25]. Cimatti and Roveri [13,14] use Binary Decision Diagrams [10] for this purpose. More recently, the work by To et al. [44] has proposed different solutions to address this issue:

Disjunctive Normal Form (DNF), Conjunctive Normal Form (CNF), and Prime Implicates (PI). The authors, in particular, have focused on the formulation of compact transition functions aimed at preserving some notion of minimality of the resulting formula, and on the definition of a best-first search schema that navigates the beliefs space independently from the given formulation. The most effective encoding, according to the results presented by To et al. [44], i.e., DNF, is tested against our approach in the experimental evaluation.

Hoffman and Brafman [23] also represent the belief state implicitly in their planner Conformant-FF (CFF), but leave the search along the plan prefixes explicit. Any relevant condition (either the goal or an action precondition) that needs to be verified in the current belief state is checked via a SAT query not unlike our procedure to compute a counter-example (Section 8). In words, this procedure consists in finding an initial state such that applying the current plan prefix leads to a state where the condition is not satisfied (the existence of this state is a necessary and sufficient condition for the goal/precondition to not be satisfied in the current set of states). In order to improve computation they also determine what state assignments are known to hold or to not hold at some point during the execution (i.e., the "backbone" of the belief state). CFF uses a conformant planning extension of relaxed planning graph heuristics [24] to guide the search. Other, more sophisticated, heuristics based on relaxed planning graph have been studied by Bryce et al. [11] too, but for a class of problems that is more general than conformant planning. These heuristics are extrapolated from what the authors call the Labeled Uncertainty Graph (LUG), which is a compact representation of several relaxed planning graphs, each built from a different initial state. The LUG differs from the relaxed planning graph constructed by Hoffmann and Brafman [23] because i) it takes mutex relationships ([4]) between different initial states into account, and ii) it is used with a different relaxed plan extraction procedure. Bryce et al. [11] present two planners based on these heuristics, CAltAlt and POND; these planners, as CFF, differ substantially from our work because they rely on a fixed relaxation of the problem. Our approach can also be understood as a way to exploit a problem relaxation (the classical planning problem that we generate). However, our relaxation is not fixed but gets tighter and tighter as more counterexamples are generated.

The work by Rintanen [35] solves conformant planning problems with Quantified Boolean Formula; this can be seen as an alternative compact representation for the belief state, that is however better integrated with the search for a valid plan. The problem is defined as an $\exists\forall\exists$ formula, i.e., "there exists a plan such that for all initial states there exists an execution that reaches the goal." A new reduction was later proposed [37] of the form $\forall\exists$ which is interpreted as "for all plan and execution, either the execution is not the result of applying the plan or the execution fails to reach the goal", the negation of which yields a conformant plan. This was used by Kronegger et al. [28,18] This approach struggles at scaling up because it fails to exploit good heuristics.

3.2. From conformant to classical planning and width

To overcome some of the limits of the previous approaches, and with the idea of exploiting the impressive advancement of classical planners, some authors have proposed the compilation of the whole conformant planning problem into classical planning.

The first compilation of conformant planning to classical planning [34,1] reasons about which assignments are known to hold. In the Grid World domain of Fig. 1 for instance, they would introduce $K(y = 0)$, the literal that indicates that the y position of the agent is known to be 0; in other words, this literal will evaluate to *true* only when it has been proved that all the states s the system could be in share the property $s(y) = 0$. The planning problem is then modified accordingly: we have seen that the condition associated with effect $y = 1$ of action N is $\mathit{eff}_{\mathrm{N}}(\langle y, 1\rangle) = (y = 0)$ (i.e., the y position of the agent becomes 1 if its current position is 0); this action will now have the effect "$K(y = 1)$ becomes true" whenever the condition $K(y = 0)$ is true in the current state. Similarly, the goal is rephrased so as to become $(K(x = 2) = \mathit{true}) \wedge (K(y = 2) = \mathit{true})$. Tu et al. [46] proposed a similar approach that uses state *approximations*. In this context, an approximate (or partial) state specifies literals that are known to hold or that may hold. The approximation leads to a deterministic model, and known techniques used in classical planning (here answer set programming) can be used to find a conformant plan.

This reformulation is not sufficient however to guarantee completeness, i.e., the set of plans valid for the classical problem may be a strict subset (sometimes empty) of that of the conformant problem. This should be obvious from the different complexity classes of conformant and classical planning. Essentially, even when a fact such as $(y = 1)$ is provably true after executing a plan, the state reached by applying the plan in this reformulated problem may not yield $K(y = 1)$. Palacios and Geffner [34] then identified the concept of *tags*, which we use in this paper and we present more formally in the next section. A tag t is a conjunction of assignments that is satisfied in some initial states, for instance $t = (y = 3)$. Additional literals are then injected into the planning problem: for instance, $K(y = 1)_{/t}$ should be interpreted as "if t held in the initial state (i.e., here, the y position of the agent was initially 3) then the y variable is known to evaluate to 1 in the current state." Again the actions need to appropriately reformulated: for all tag t, action N will make $K(y = 1)_{/t}$ true whenever $K(y = 0)_{/t}$ is currently true.

In this context, a *merge* is then a collection of tags such that all initial states satisfy at least one tag. In the Grid World example, a merge would be $\{t_0, \ldots, t_4\}$ where for all $i \in \{0, \ldots, 4\}$, t_i is $(y = i)$ (each possible initial y position of the agent is initially between 0 and 4). If $K(v = \ell)_{/t}$ evaluates to true for every tag t in the merge, it is possible to infer that v evaluates to ℓ in all current states: $K(v = \ell)$ holds. This is done explicitly through a "merge" action that the authors add to the planning problem: its precondition is $\bigwedge_t K(v = \ell)_{/t}$ over all the tags of the merge, and its effect is $K(v = \ell)$. It is possible to compute all tags and merges that make this reduction complete.

A careful study of this reduction shows the relevance of the *width* of the conformant planning problem. Essentially the width is the maximum number of variables mentioned in a single tag. The number of tags relevant to each variable is exponential in the width. Consequently this reduction is practical only if the width is one.

The main issue with this approach is that the tags need to be computed upfront based on the structure of the problem. When these tags are numerous, performance is strongly impacted. In some instances, for instance in the Grid-Wall or the One-Dispose domains described in the experimental section, not all the tags are required because they are "subsummed", either completely or for the most part, by other tags. Our approach is able, to some degree, to determine what subset of tags is actually necessary.

Following up on translation-based approaches, Albore et al. [2] introduce the notion of *basis*, which is a subset of initial states that exhibit the same set of valid plans as the original problem. This is useful for the following reason: it is possible to reduce conformant planning to classical planning, as we also demonstrate in Section 5, but the reduction is linear in the number of initial states. This complexity makes it impractical for any problem instance with a non-trivial number of initial states, and replacing the initial belief with a (small) basis can make the reduction practical. Albore et al. use the concept of a basis to build an extended version of Hoffmann and Brafman' planner where the heuristic is obtained using the relaxation induced by the complete yet unsound conformant planning reduction proposed. This led to the implementation of the T1 planner which supersedes the CFF planner. T1 is tested in our experimental evaluation.

Tran et al. [45] also use the structure of the conformant planning problem to replace the initial belief state with a basis. Considering the Grid World of Fig. 1.a for instance, they identify conditions that allow one to replace the initial belief state

$$\texttt{one-of}(x=0, x=1, x=2, x=3, x=4) \wedge \texttt{one-of}(y=0, y=1, y=2, y=3, y=4)$$

with

$$\texttt{one-of}((x=0 \wedge y=0), (x=1 \wedge y=1), (x=2 \wedge y=2), (x=3 \wedge y=3), (x=4 \wedge y=4)).$$

The second belief state is indeed a subset of the first one and corresponds to locations in the grid in the diagonal. Compared to this work and the work from Albore et al., CPCES is able to identify automatically which states are relevant and should be included in the initial belief state.

3.3. Sampling based methods

Other approaches sample only a portion of the belief state and try to find a plan that works for it. The fragment planner [29] tries to find a conformant plan by combining plans resulting from solving each initial state independently; such initial states are randomly sampled. The authors investigate different ways of performing this search, yet none guarantees a systematic reduction of the initial states that is able to exploit the structure of the problem; as noted by Nguyen et al. [33], the fragment planner does not scale well. With the aim of overcoming the fragment planner limitations, Nguyen et al. [33] proposed the so called generate-and-complete approach. The idea is to find a plan for a sub-problem $P(s_0)$ of the conformant planning problem using a classical planner, try to repair it to account for other initial states, and if that does not work explore another classical planning solution for $P(s_0)$. The system implementing this idea is called CG-LAMA; its performance are also evaluated experimentally in Section 10. SDR [9], which stands for *Sample, Determinize, Replan*, behaves in a similar manner[1]; it generates a plan for a single initial and then executes it; if the plan failed, a new plan is computed and executed. Compared to our approach in which we recompute a plan if there exists a counter-example, SDR computes a new plan after the first one was executed. This implies that SDR already committed to that plan. Hence, this approach is incomplete in general: in the example of Fig. 1.b, if the robot gets into the swamp, it is stuck there forever. We will see that the sampling adopted in our framework explores only those states that are able to contradict some plan solution; this contrasts previous work in that CPCES does not explore several classical planning solutions for the same initial state but iterates over an increasing set of initial states.

Finally we want to discuss the notion of *regression* of a plan. Rintanen [38], and Brafman and Shani [9] showed how to compute the set of initial states for which a given plan is valid. One way to understand how we check whether a plan is valid is to use regression. That is, computing a formula that encompasses the conditions under which a given plan is valid and evaluating whether the initial belief entails such a formula. As noted by Rintanen [38], however, this is not practical as the formula grows exponentially with the length of the plan.

To solve this issue, we use a SAT-based encoding in a way that is very similar to SAT-Based planning [26,40]. There is a fundamental difference between these existing works and our approach however. Traditional approaches assume that the initial state is known, and they use SAT to compute a sequence of actions that leads to the goal. Comparatively, we assume that the plan is already provided, and we use SAT to compute an initial state for which the plan is invalid.

[1] We note that SDR was developed for a different problem, namely planning with non-deterministic effects and partial observations, but we consider it interesting to compare how it differs to CPCES on how it handles invalid plans.

4. Structure of conformant planning problems

In this section we discuss the structural properties of conformant planning problems, and in particular the notion of tag and basis introduced by Palacios and Geffner [34]. Even though these notions have been used elsewhere to establish conditions for sound and complete translation from conformant planning to classical planning, this discussion is relevant for us because it provides the theoretical grounding to define the properties of our system, such as the maximum number of iterations CPCES requires. An other reason for presenting these notions here is that we slightly modify the definitions to handle any type of condition in condition effects, as opposed to only conjunctions.

4.1. Tags

We start with a brief and intuitive description of the notions presented in the next definitions. A "subgoal" is any formula that may need to be satisfied at some point during the plan execution, for the plan to be valid. The "context" of a subgoal is then the set of variables whose value in the initial state influences the satisfaction of the subgoal during the plan execution; to decide whether a particular subgoal is satisfied during the execution, it is sufficient to reason only about the variables in the context. Next, a "tag" is one of the possible initial assignments of the variables in a context. Tags are useful in the sense that they represent the contingencies that the planner must address. The formal definitions are now presented and later illustrated on the running example as well as a more elaborate example.

Definition 6. Given the conformant planning problem $P = \langle \mathcal{V}, A, prec, eff, \Phi_I, \Phi_G \rangle$, the following notions are defined:

- A *validity condition* is either the precondition of an action or the goal formula.
- A *subgoal* is a conjunct of a validity condition. In other words, all validity conditions are conjunctions of subgoals. We write $SubGoals(P)$ the list of subgoals of P.
- Given a formula $\varphi \in Forms(\mathcal{V})$, φ *depends on* variable v if the variable v is mentioned in φ.
- Given a subgoal φ, the *context* of φ is the minimal set of variables $ctx(\varphi) \subseteq \mathcal{V}$ that satisfies:
  - if φ depends on variable v, then $v \in ctx(\varphi)$ holds; and
  - if $v \in ctx(\varphi)$ holds and there exists an action a and a value $\ell \in D_v$ such that $eff(a)(\langle v, \ell \rangle)$ (the condition under which action a assigns v to ℓ) depends on $v' \in \mathcal{V}$, then $v' \in ctx(\varphi)$ holds.
- Given a subgoal φ and its context C, the *width* of the context is the number of variables whose initial value is uncertain in the initial belief of the conformant planning problem. The *width* of the problem is the maximum width of any subgoal of the problem.
- A *tag* is an assignment consistent with the initial belief of all variables in the context $ctx(\varphi)$ of φ, where $\varphi \in SubGoals(P)$ is one of the problem subgoals. We write $Tags(P)$ the set of tags of P.

Example. We illustrate these definitions on the Grid World example of Fig. 1.a and Fig. 1.b.

In both cases, the validity conditions are the goal $(x = 2) \wedge (y = 2)$ and the preconditions of all actions $\top$. The preconditions contain no conjunct, but the goal yields two subgoals: $(x = 2)$ and $(y = 2)$, represented respectively with φ_x and φ_y.

In Fig. 1.a, consider the conditions $eff(a)(\langle x, \ell \rangle)$ for any action a and any value ℓ. The only variable that these conditions mention is x, i.e., the robot's position in x is only ever determined by its former position in x and never by its position in y. Therefore, the subgoal φ_x has context $ctx(\varphi_x) = \{x\}$. Similarly $ctx(\varphi_y) = \{y\}$ holds, and the width of the problem is one.

In the problem of Fig. 1.b, we have seen that the condition $eff(\text{N})(\langle y, 2 \rangle)$ is $y = 1 \wedge \neg(x = 3)$, which means that the subgoal $(y = 2)$ depends on both y and x. Therefore in this problem, $ctx(\varphi_y)$ evaluates to $\{x, y\}$. Similarly $ctx(\varphi_x)$ equates $\{x, y\}$, and the width of the problem is two.

Given these contexts, there are ten tags in the problem of Fig. 1.a, which are the five tags $\{x \mapsto i\}$ for all $i \in D_x$ and the five tags $\{y \mapsto j\}$ for all $j \in D_y$. On the other hand, there are 24 tags in the problem of Fig. 1.b, which are $\{x \mapsto i, y \mapsto j\}$ for all $i \in D_x$ and $j \in D_y$ (except for $\{x \mapsto 3, y \mapsto 1\}$ which is not possible in the initial state). $\diamond$

We now discuss why these notions are relevant. Given an initial state s_0 and a plan $\pi = a_1, \ldots, a_k$, the execution of π from s_0 gives us the sequence $s_0, \ldots, s_k$ of states (remember that we assume that this sequence is defined even when some of the actions are not applicable). The plan π is valid from state s_0 iff

- every subgoal associated with every action a_i is satisfied in state s_{i-1}; and
- every subgoal associated with the goal condition is satisfied in state s_k.

Furthermore, the plan is valid from the initial belief state iff it is valid in every one of its initial states. Therefore, the plan validity relies on a conjunction of conditions based on the subgoals (the specific set of subgoals varies depending on the plan). In addition, the contexts of each subgoal describe the list of variables whose initial value can influence the satisfaction of the subgoal at any point during the execution, and the tags tell us what the possible values of these variables can be. Therefore the plan is valid iff the subgoals are satisfied for all tags the initial belief yields.

Because some complexity results rely on the number of tags, it is convenient to define them so that their number is minimised. For instance, the example of Fig. 1.a yields only 10 tags against the 24 tags of Fig. 1.b, which, other things being equals, makes the first example easier than the second one. In particular, the definition allows the "splitting" of conjunctive validity conditions into subgoals. We now introduce a more sophisticated example to illustrate i) why the subgoals are conjuncts (and not disjuncts for instance) and, consequently, ii) how the formulation of the conformant planning problem affects the set of tags.

Example. Consider a problem in which an action a is applicable if and only if the following two conditions are satisfied: the agent must have received proper authorisation (modeled as $\mathit{has\_authorisation} = \mathit{true}$) and the agent must have some form of identification (a passport $\mathit{has\_pp} = \mathit{true}$ or a driving license $\mathit{has\_dl} = \mathit{true}$), i.e., the precondition is:

$$\mathit{has\_authorisation} = \mathit{true} \ \wedge \ (\mathit{has\_pp} = \mathit{true} \ \vee \ \mathit{has\_dl} = \mathit{true})$$

According to our definitions, this action produces two subgoals, respectively $\varphi_1 = (\mathit{has\_authorisation} = \mathit{true})$ and $\varphi_2 = (\mathit{has\_pp} = \mathit{true} \ \vee \ \mathit{has\_dl} = \mathit{true})$.

Why is it possible to split this precondition into two subgoals? Intuitively, the idea is that it is possible to reformulate the planning problem so that the following properties are satisfied:

- action a is replaced with two actions a_1 and a_2 whose preconditions are respectively φ_1 and φ_2; and
- small changes are included so that a_1 must always be followed by a_2, and a_2 always preceded by a_1.

Plan π is valid in the original problem iff π', the copy of π where any occurrence of a is replaced with a_1 followed by a_2, is valid in the reformulated problem. Therefore the semantics of these two problems is equivalent (up to a rewriting of the plans). This explains why φ_1 and φ_2 can be studied independently.

Let us now consider disjunctions, and in particular φ_2. Let us replace action a_2 with two new actions a_2' and a_2'' with preconditions $\mathit{has\_pp} = \mathit{true}$ and $\mathit{has\_dl} = \mathit{true}$ respectively. Then the semantics of this new problem is fundamentally different from the original one. Indeed, the action a_2' requires the agent to be sure that they have a passport; and a_2'' that they have a driving license. Assume, however, that a previous action consists in picking up the top document of the pile, which could be either the passport or the driving license depending on the initial state; then neither a_2' nor a_2'' is applicable while a_2 is. This shows that the disjunction φ_2 ought not be split. Another way to look at this question is to recognise that, while $\forall x.\ (f(x) \wedge g(x))$ and $\forall x.\ f(x) \wedge \forall x.\ g(x)$ are logically equivalent, $\forall x.\ (f(x) \vee g(x))$ and $\forall x.\ f(x) \vee \forall x.\ g(x)$ are not.

Importantly, the formulation of the planning problem will affect the definition of tags. For instance, if the precondition of action a was instead the equivalent reformulation

$$(\mathit{has\_authorisation} = \mathit{true} \wedge \mathit{has\_pp} = \mathit{true}) \ \vee \ (\mathit{has\_authorisation} = \mathit{true} \wedge \mathit{has\_dl} = \mathit{true}),$$

then there would be a single subgoal that involves all three variables. CPCES is not directly affected by the problem formulation (although the internal classical planner it uses may be), but this discussion is nevertheless relevant when discussing CPCES' properties. ◇

The original definition of tag [34], assumes that the domain of the tag includes only variables that are initially uncertain, whereas we consider that it spans over all the variables of the context. This choice is questionable, and there are situations where two tags that look similar could be merged into a single one (i.e., when the only difference is that their domains disagree on variables that are initially certain). Our motivation is to be able to compare the set of tags for the same problem with different initial belief states. In particular with our definition, the conformant planning problems exhibit the following monotonic property:

$$\mathcal{B}(\Phi_1) \subseteq \mathcal{B}(\Phi_2) \ \Rightarrow \ \mathit{Tags}(P^{\Phi_1}) \subseteq \mathit{Tags}(P^{\Phi_2}),$$

where P^{Φ} denotes a copy of P in which the initial formula is replaced with Φ. In other words, adding states to the initial belief increases the set of tags of the planning problem monotonically.

4.2. Basis

We now present the notion of *basis* where, as above, P^{Φ} denotes a copy of P in which the initial formula is replaced with Φ.

Definition 7. Given the conformant planning problem $P = \langle \mathcal{V}, A, \mathit{prec}, \mathit{eff}, \Phi_I, \Phi_G \rangle$ a *basis* is a subset $\mathcal{B} \subseteq \mathit{states}(\Phi_I)$ of initial states such that $\Pi(P)$ equals $\Pi(P^{\Phi_{\mathcal{B}}})$.

We say that a basis is *minimal* if no strict subset of the basis is itself a basis. Small bases are useful because the hardness of conformant planning derives from the size of the initial belief states. If $\mathcal{B}$ is a basis of fixed size and $s \in \mathcal{B}$ is one of its

elements, it is possible to show that deciding whether $\mathcal{B}' = \mathcal{B} \setminus \{s\}$ is a basis is PSPACE-COMPLETE: $\mathcal{B}'$ is not a basis if there exists a plan valid for $\mathcal{B}'$ that is not valid s. If the size of $\mathcal{B}$ is not bounded, then the problem is EXPSPACE-COMPLETE. CPCES approximates the computation of a basis by detecting useful states although there is no guarantee that the final sample will be a basis.

Example. As we shall see in the next section, any subset of states whose set of tags equals the set of tags of the conformant problem is a basis. For instance in the example of Fig. 1.a, the set of states $\{s_{0,0}, s_{1,1}, s_{2,2}, s_{3,3}, s_{4,4}\}$ exhibits the 10 tags of the conformant planning problem, and is therefore a basis. However in some problems, there exist smaller bases. In particular in the example of Fig. 1.a, a minimal basis is $\{s_{0,0}, s_{4,4}\}$.

Computing a minimal basis is more tricky for the example of Fig. 1.b. Here is one such example: $\{s_{3,0}, s_{2,0}, s_{1,0}, s_{0,0}, s_{3,2}, s_{2,2}, s_{1,2}, s_{0,2}, s_{2,1}, s_{4,1}, s_{3,3}, s_{3,4}\}$. $\diamond$

5. Rewriting planning problems: specialisation, projection, and synchronisation

In this section, we discuss theoretical aspects of the problem of conformant planning. This work is the theoretical ground that CPCES relies on.

The ideas exposed here are very much inspired by the work from Palacios and Geffner [34] and Albore et al. [2], in that they exploit classical planning reformulations. With respect to previous work, our discussion provides additional insights about the conformant planning problem, and also allows us to define two new reformulations to classical planning which are more amenable for how we intend to use them (Section 6). We show exactly how these reformulations differ from the reformulations proposed by these authors, namely $K_{T,M}$ and K_S; in particular they do not require the definition of new actions, and they handle disjunctive subgoals more elegantly.

We start with a quick overview (Fig. 3) of the three operations that we define: Specialisation, Synchronisation and Projection:

1. **Specialisation.** The specialisation of P defines a classical problem P^s where the set of initial states is replaced with the single initial state s. We show that the intersection of the solution sets of all specialisations is precisely the solution set of P. This operation is very similar to the operation $P[s]$ from [34] that was left unnamed.
2. **Synchronisation.** In order to compute a plan that is valid for a collection of classical planning problems, the synchronisation operation transforms this collection into a single planning problem. We show that the set of solutions of the synchronisation is precisely the intersection of the solution sets of this collection.
   Specialisation and synchronisation give us a reformulation from conformant to classical planning: the synchronisation of all specialisations of the conformant problem exhibits the same solution set as said conformant problem. This is not very practical however as the size of the synchronised problem is linear in the number of initial states, which in turn is generally exponential in the number of state variables.
3. **Projection.** For this purpose, we define the projection operation that focuses on a single subgoal of a classical planning problem. Again, we show that the intersection of the solution sets of all projections of a classical problem is precisely the solution set of this problem. Therefore the synchronisation of the projections of the specialisations of P is a classical problem $P_\oplus$ whose solution set is the same as that of P. The benefit of this method is that, as illustrated in Fig. 3, the projection generates many identical problems, the copies of which can be discarded. As a consequence, the size of $P_\oplus$ is only linear in the number of tags of P.

5.1. Specialisation

Specialisation is an operation that transforms a conformant planning problem into a classical planning problem by focussing on a single initial state.

Definition 8. Given a conformant problem $P = \langle \mathcal{V}, A, prec, eff, \Phi_I, \Phi_G \rangle$, given an initial state s (i.e., $\Phi_s \models \Phi_I$), the *specialisation* of s for P is the classical planning problem $P^s = \langle \mathcal{V}, A, prec, eff, \Phi_s, \Phi_G \rangle$.

The specialisation of s for P produces the classical planning problem where the set of initial states Φ_I is replaced with a single initial state s. The sets of solutions of the specialised problems and the original conformant problem are linked by the following lemma:

Lemma 1. *Let $P = \langle \mathcal{V}, A, prec, eff, \Phi_I, \Phi_G \rangle$ be a conformant planning problem. The following statement holds:*

$$\Pi(P) = \bigcap_{s \in states(\Phi_I)} \Pi(P^s).$$

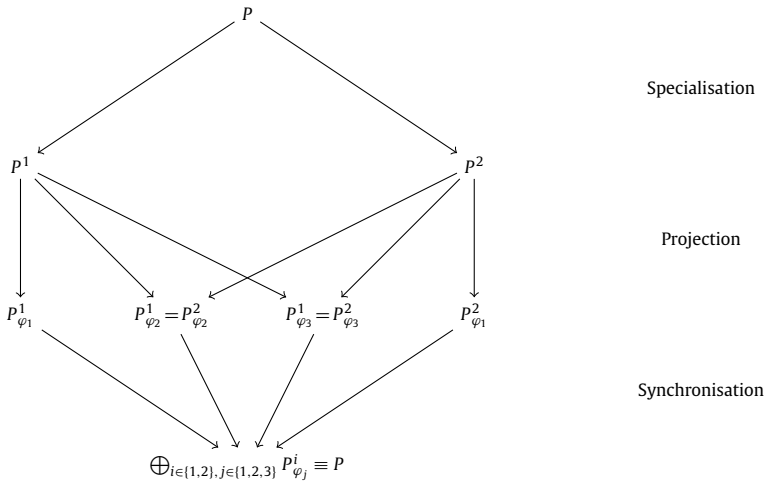


Fig. 3. Example of manipulation of a conformant planning problem. Here because there are two initial states, P is specialised in two problems P^1 and P^2. Each P^i is projected on three subgoals; some of the projections lead to equivalent problems. Finally all projections are synchronised into the single classical planning problem $P_\oplus$ that is equivalent to P.

Proof of Lemma 1. By definition of plan validity, a plan is valid for P iff it is valid for each of its initial states. Therefore, if a plan is valid for P then it is valid for P^s (so, $\Pi(P) \subseteq \bigcap_{s\in states(\Phi_I)} \Pi(P^s)$). Furthermore if the plan is valid for all of its initial states, then it is valid for P (so, $\Pi(P) \supseteq \bigcap_{s\in states(\Phi_I)} \Pi(P^s)$). Therefore the two sets are equivalent. □

5.2. Projection

Projection is the operation that focuses on one aspect of the planning problem, i.e., a subset of variables; specifically, the context of a subgoal. This operation could be defined on conformant planning problems but we limit ourselves to classical planning problems as this is sufficient for our purpose.

Given a conjunction $\phi = \phi_1 \wedge \cdots \wedge \phi_k \in Forms(\mathcal{V})$ (where k may evaluate to 1 or even 0), given a subset of variables $V \subseteq \mathcal{V}$, we write $\phi \downarrow V$ the formula $\phi' = \phi'_1 \wedge \cdots \wedge \phi'_k$ where for each i,

$$\phi'_i = \begin{cases} \phi_i & \text{if all the variables mentioned by } \phi_i \text{ belong to } V \\ \top & \text{otherwise.} \end{cases}$$

So for instance, if $\phi = ((v_1 = \ell_1) \vee (v_2 = \ell_2)) \wedge ((v_1 = \ell_1) \vee (v_3 = \ell_3))$, then $\phi \downarrow \{v_1, v_2, v_4\}$ is defined as $((v_1 = \ell_1) \vee (v_2 = \ell_2)) \wedge \top$, which reduces to $(v_1 = \ell_1) \vee (v_2 = \ell_2)$.

Given a complete assignment s of the variables $\mathcal{V}$ and a subset of variables $V \subseteq \mathcal{V}$, we write $s \downarrow V$ the *projection* of s defined as the restriction of s to the variables in V. Notice that, by definition, if $s \models \phi$, then $(s \downarrow V) \models (\phi \downarrow V)$.

Definition 9. Given a classical problem $P = \langle \mathcal{V}, A, prec, eff, \Phi_I, \Phi_G \rangle$, given a subgoal φ, the *projection* of P on φ is the classical planning problem $P_\varphi = \langle \mathcal{V}', A', prec', eff', \Phi'_I, \Phi'_G \rangle$ defined as follows:

- $\mathcal{V}' = ctx(\varphi)$;
- $A' = A$;
- for all action $a \in A'$, $prec'(a) = prec(a) \downarrow \mathcal{V}'$;
- for all action $a \in A'$, for all variable $v \in \mathcal{V}'$, and for all value $\ell \in D_v$, $eff'(a)(\langle v, \ell \rangle) = \begin{cases} eff(a)(\langle v, \ell \rangle) & \text{iff } v \in \mathcal{V}' \\ \text{is undefined} & \text{otherwise} \end{cases}$, i.e., $eff'(a)$ is the restriction of $eff(a)$ to the assignments of $\mathcal{V}'$;
- $\Phi'_I = \Phi_I \downarrow \mathcal{V}'$; and
- $\Phi'_G = \Phi_G \downarrow \mathcal{V}'$.

Example. Consider the GridWorld example from Fig. 1.a with only the initial state $\{x \to 1, y \to 2\}$. The subgoal $\varphi = (x = 2)$ yields the context $ctx(\varphi) = \{x\}$. The projection of this problem on φ is a classical planning problem that mentions the x variable and ignores the y variable. Actions such as N or S are still defined, but since they only affect variable y in the original problem, they have no effect on the projected problem. Furthermore, since the goal of the original problem is $(x = 2) \wedge (y = 2)$, the projected goal is simply $(x = 2)$. ◇

Notice that the problem P_φ is well defined for the subgoal. Indeed, the initial and goal formulas only mention variables of $\mathcal{V}'$ thanks to the $\downarrow$-operator, and so do the actions' preconditions; furthermore, from the definition of the context of a subgoal, the conditions of the conditional effects in P_φ are also defined over $\mathcal{V}'$.

Also notice that if φ and φ' depend on the same variables, then P_φ is identical to $P_{\varphi'}$. In other words, the number of projections of P is bounded by the number of contexts of its subgoals.

Lemma 2. *Let $P = \langle \mathcal{V}, A, prec, eff, \Phi_I, \Phi_G \rangle$ be a classical planning problem, and let φ be a subgoal of P. The following statement holds:*

$$\Pi(P) \subseteq \Pi(P_\varphi).$$

Proof of Lemma 2. We write $P = \langle \mathcal{V}, A, prec, eff, \Phi_I, \Phi_G \rangle$ and $P_\varphi = P' = \langle \mathcal{V}', A', prec', eff', \Phi'_I, \Phi'_G \rangle$. Let $\pi = a_1, \ldots, a_k$ be a valid plan of P. We shall prove that π is a valid plan of P'.

Let s_0 be the initial state of P (i.e., $\{s_0\} = states(\Phi_I)$), and $s_i = s_0[a_1, \ldots, a_i]_{eff}$ be the state reached from applying the first i actions of the plan. Let s'_0 be the initial state of P' (i.e., $\{s'_0\} = states(\Phi'_I)$) and $s'_i = s'_0[a_1, \ldots, a_i]_{eff'}$ be the state reached from applying the first i actions of the plan.

The proof has three parts. First, we demonstrate $s'_i = s_i \downarrow \mathcal{V}'$. Second, we use this result to demonstrate that each action a_i is applicable from s'_{i-1}. Third, we demonstrate that the goal is satisfied in s'_k, which finishes the proof.

1. We prove $s'_i = s_i \downarrow \mathcal{V}'$ by induction on i.
   - Base case:
     By definition of the $\downarrow$-operator, $s_0 \downarrow \mathcal{V}'$ is the only state of $\mathcal{V}'$ that satisfies $\Phi'_I = \Phi_I \downarrow \mathcal{V}'$.
   - Induction step:
     Assume that $s'_{i-1} = s_{i-1} \downarrow \mathcal{V}'$ holds. Let $\langle v, \ell \rangle$ be an assignment of a variable $v \in \mathcal{V}'$. Then $eff(a_i)(\langle v, \ell \rangle) = eff'(a_i)(\langle v, \ell \rangle)$, which implies that the value associated with v changes to ℓ between s_{i-1} and s_i iff it also changes to the same value between s'_{i-1} and s'_i. Therefore $s'_i = s_i \downarrow \mathcal{V}'$.
2. Since a_i is applicable in s_{i-1}, the precondition $prec(a_i)$ is satisfied in s_{i-1}. Therefore $(s_{i-1} \downarrow \mathcal{V}') \models (prec(a_i) \downarrow \mathcal{V}')$ holds. Since s'_{i-1} is $(s_{i-1} \downarrow \mathcal{V}')$ and $prec'(a_i)$ is $(prec(a_i) \downarrow \mathcal{V}')$, $s'_{i-1} \models prec'(a_i)$ holds and the action a_i is applicable in s'_{i-1}.
3. Similarly the fact that Φ_G is satisfied in s_k and the fact that $s'_k = s_k \downarrow \mathcal{V}'$ prove that $\Phi_G \downarrow \mathcal{V}'$ is satisfied in s'_k. □

Lemma 3. *Let P be a classical planning problem. The following statement holds:*

$$\Pi(P) \supseteq \bigcap_{\varphi \in SubGoals(P)} \Pi(P_\varphi).$$

Proof of Lemma 3. Consider a plan $\pi = a_1, \ldots, a_k$ that is not valid for P. We shall prove that this plan is invalid for at least one of its projections.

Since π is not valid, then either at least one of its action is not applicable or the goal is not satisfied in the final state. We focus on the action preconditions, as the case with the goal can be treated similarly.

We denote φ the subgoal that is unsatisfied by π in P and show that π is not valid for $P' = P_\varphi$. We reuse the notation from the previous proof and remember that $s'_i = s_i \downarrow \mathcal{V}'$. We denote j the index at which the subgoal is incorrectly unsatisfied.

Then since $prec'(a_j) = prec(a_j) \downarrow \mathcal{V}'$ and since $s'_j = s_j \downarrow \mathcal{V}'$ (from the proof of Lemma 2), it follows that $prec'(a_j)$ is not satisfied in s'_{j-1}. Therefore π is not valid for P'. □

We can combine Lemmas 2 and 3 to prove the following corollary:

Corollary 1. *Let P be a classical planning problem. The following statement holds:*

$$\Pi(P) = \bigcap_{\varphi \in SubGoals(P)} \Pi(P_\varphi).$$

5.3. Synchronisation

Synchronisation is an operation that transforms two planning problems with the same set of actions into a single problem where each action affects both problems in a synchronous fashion.

Definition 10. Given two classical planning problems $P^1 = \langle \mathcal{V}^1, A^1, prec^1, eff^1, \Phi^1_I, \Phi^1_G \rangle$ and $P^2 = \langle \mathcal{V}^2, A^2, prec^2, eff^2, \Phi^2_I, \Phi^2_G \rangle$ defined over the same set of actions ($A^1 = A^2$ and is also denoted A) but disjoint sets of state variables ($\mathcal{V}^1 \cap \mathcal{V}^2 = \emptyset$), the *synchronisation* of P^1 and P^2 is the problem $P^1 \oplus P^2 = \langle \mathcal{V}, A, prec, eff, \Phi_I, \Phi_G \rangle$ defined by:

- $\mathcal{V} = \mathcal{V}^1 \cup \mathcal{V}^2$;
- for all action $a \in A$, $prec(a) = prec^1(a) \wedge prec^2(a)$;
- for all action $a \in A$, for all variable $v \in \mathcal{V}$, and for all value $\ell \in D_v$, $eff(a)(\langle v, \ell \rangle) = \begin{cases} eff^1(a)(\langle v, \ell \rangle) & \text{if } v \in \mathcal{V}^1 \\ eff^2(a)(\langle v, \ell \rangle) & \text{if } v \in \mathcal{V}^2; \end{cases}$
- $\Phi_I = \Phi_I^1 \wedge \Phi_I^2$; and
- $\Phi_G = \Phi_G^1 \wedge \Phi_G^2$.

It is easy to see that $\oplus$ is commutative and associative under the equivalence relation $\equiv$ over the planning problems. We extend the definition to the case where the variables intersect ($\mathcal{V}^1 \cap \mathcal{V}^2 \neq \emptyset$) by assuming that the variables are renamed so that they do not intersect.

Example. In the GridWorld example of Fig. 1, consider two problem instances, one in which the initial state is $\{x \to 0, y \to 0\}$ and one in which it is $\{x \to 4, y \to 4\}$. Because the two instances' variable sets intersect, we rename them as follows: we append 1 to the first instance's variables (so $x1$ and $y1$) while we append 2 to the other ones. In the resulting synchronised problem, the initial state is therefore $\{x1 \to 0, y1 \to 0, x2 \to 4, y2 \to 4\}$. The N action will then affect both variables $y1$ and $y2$. $\diamond$

Notice that when, in Definition 10, we write, e.g., $\Phi_I = \Phi_I^1 \wedge \Phi_I^2$, we mean in practice $\Phi_I = \varphi_1^1 \wedge \cdots \wedge \varphi_{k_1}^1 \wedge \varphi_1^2 \wedge \cdots \wedge \varphi_{k_2}^2$, where $\Phi_I^i = \varphi_1^i \wedge \cdots \wedge \varphi_{k_i}^i$ for both $i \in \{1, 2\}$; the reason why this is relevant is that the conjuncts of Φ_I are the union of the conjuncts of Φ_I^1 and Φ_I^2.

Lemma 4. *Let P^1 and P^2 be two classical planning problems defined on the same set of actions. The following statement holds:*

$$\Pi(P^1 \oplus P^2) = \Pi(P^1) \cap \Pi(P^2).$$

Proof of Lemma 4. We use Corollary 1 to prove this result, and we write $P = P^1 \oplus P^2$.

By definition, the subgoals of P are precisely the subgoals of P^1 plus those of P^2. Let φ be a subgoal of P^i for $i \in \{1, 2\}$. Recall that P_φ (resp. P_φ^i) is the projection of P (resp. P^i) over the context of φ; by definition, and because the variables of P^1 and the variables of P^2 are disjoint, the contexts of P_φ and P_φ^i are the same, and the projections are also the same ($P_\varphi^i = P_\varphi$). Therefore,

$$\begin{aligned}
\Pi(P^1 \oplus P^2) &= \Pi(P) \\
&= \textstyle\bigcap_{\varphi \in SubGoals(P^1 \oplus P^2)} \Pi(P_\varphi) && \text{(Corollary 1)} \\
&= \textstyle\bigcap_{\varphi \in SubGoals(P^1) \cup SubGoals(P^2)} \Pi(P_\varphi) \\
&= \textstyle\bigcap_{\varphi \in SubGoals(P^1)} \Pi(P_\varphi) \cap \bigcap_{\varphi \in SubGoals(P^2)} \Pi(P_\varphi) \\
&= \textstyle\bigcap_{\varphi \in SubGoals(P^1)} \Pi(P_\varphi^1) \cap \bigcap_{\varphi \in SubGoals(P^2)} \Pi(P_\varphi^2) \\
&= \Pi(P^1) \cap \Pi(P^2). && \text{(Corollary 1)} \quad \square
\end{aligned}$$

5.4. Summarising the results

It is possible to combine the results of Lemma 1, Corollary 1, and Lemma 4 to propose a reformulation of conformant planning in a classical planning problem.

Consider a tag $t \in Tags(P)$. By definition there exist at least one state $s \in states(\Phi_I)$ and one subgoal φ such that $t = s \downarrow ctx(\varphi)$. Then we define P^t as P_φ^s, i.e., the result of a specialisation on s and then a projection on φ. Notice that P^t remains the same regardless which state s and which condition φ was used. Notice also that P^t is a classical planning problem.

Corollary 2. *Let P be a conformant planning problem. The following statement holds:*

$$\Pi(P) = \bigcap_{t \in Tags(P)} \Pi(P^t).$$

This is a consequence of Lemma 1 and Corollary 1. Adding the synchronisation operation, we obtain the following result:

$$P \equiv \bigoplus \{ P^t \mid t \in Tags(P) \}.$$

It is therefore possible to reformulate the conformant planning problem into a classical planning problem with a number of state variables that is linear in the number of tags. We denote this reformulation SPS (specialisation + projection + synchronisation), while we call SS the simpler reformulation consisting in specialisation followed by synchronisation.

We notice that if two sets of states exhibit the same sets of tags ($Tags(\mathcal{B}_1) = Tags(\mathcal{B}_2)$) then their set of solutions is identical: $\Phi(P^{\mathcal{B}_1}) = \Phi(P^{\mathcal{B}_2})$. In particular if $Tags(\mathcal{B})$ equals the set of tags of the conformant problem, then $\mathcal{B}$ is a basis, as already acknowledged by Albore et al. [2]; notice however that this condition is not necessary.

Our two reformulations (SS and SPS) are very similar to the two reformulations from Albore et al. [2] and Palacios and Geffner [34], namely K_{S0} and $K_{T,M}$. The main difference is that they introduce new "merge" actions, whose purpose is to verify the value of some state variables. So, consider for instance an action whose precondition is $(v = \ell)$. In our reformulation SPS, this action's precondition will be reformulated as:

$$(v_1 = \ell) \wedge \cdots \wedge (v_k = \ell)$$

where k is the number of tags for the subgoal $(v = \ell)$. In $K_{T,M}$ instead, the action would have precondition $K(v = \ell) = true$. This assignment could be obtained as an effect of the existing actions, but $K_{T,M}$ provides a merge action to achieve this precondition, as this action precondition is:

$$(K(v = \ell)_1 = true) \wedge \cdots \wedge (K(v = \ell)_k = true)$$

where k is, again, the number of tags. We see that the two reformulations very similar, the main difference being that $K_{T,M}$ requires the introduction of new actions, which can affect the optimality of the plan, complicates the process of validating when a solution is valid, and can introduce unnecessary branching points.

Another difference between our reformulations and the existing ones is the fact that our approach handles sophisticated preconditions in a more elegant way. If a precondition is a disjunction, such as $(v = \ell) \vee (v' = \ell')$, our reformulations can be used as is. The existing reformulations, on the other hand, will need some additional work in order to be able to express this type of subgoals, as they are not directly supported.

In the next section we show the system that is grounded the principles reported here. As we will see, we do not actually anticipate a full transformation to classical planning (which would be linear in the number of tags), but build it incrementally by learning (via sampling) only those states which are necessary; from a practical standpoint this lazy approach is more effective because it does not need to anticipate all the contingencies upfront. Moreover, and more interestingly, our best sampling iterates a number of times that is at most the number of tags of the problem (Theorem 3).

6. CPCES: a general conformant planner

In previous works [34,2] and in the previous section, sound and complete reformulations of conformant planning problems into classical planning problems have been proposed. These reformulations share the same shortcoming however, which is that the size of the resulting problem is linear in the number of tags or states of the initial belief, which in turn can be exponential in the number of state variables. This implies that this approach is essentially limited to domains with a small width, typically one.

In this section we present CPCES, a novel family of conformant planners. While CPCES is very similar to the approach of the previous section (and, indeed, most of the proofs in connection with CPCES are trivial implications of the theorems and lemmas of Section 5), CPCES is completely oblivious to the notion of tags. CPCES maintains a subset of initial states, the *sample*, that is a tentative basis of the conformant planning problem, and uses this sample to compute a candidate plan. The sample is populated by *counter-examples*, initial states that make earlier candidate plans invalid. Different variants of CPCES disagree on how this sample should be updated. Using this procedure, CPCES is often able to identify small samples that are sufficient to discover a valid plan: in the previous section, the validity of the plan was only guaranteed by having the basis exhibit all tags. In CPCES, the irrelevant tags are naturally ignored by the planner.

CPCES can be seen as an implementation of CEGAR (counter-example guided abstraction refinement) [15], a popular method to address the complexity of hard problems. Under the CEGAR schema, a hard problem P is solved as follows: an abstraction P' of P is built, which is easy to solve; the solution of P' is computed; the validity of this solution for P is tested; if the solution is invalid, P' is refined (i.e., made less abstracted) so that this solution is invalid for the new problem.

In this section, we first present CPCES as a general conformant planning algorithm. We discuss how this algorithm returns not only a conformant plan but also, potentially, a justification for this plan. We then look at two important properties that we want the CPCES inner loop to satisfy, namely that the counter-example set (formally defined next) is dominance-preserving and that the counter-example set is minimal. Because these two properties are somewhat conflicting we also propose another implementation of CPCES that uses a heuristic to refine the counter-example set. Finally we discuss a variant of CPCES for optimal planning.

The CPCES family, i.e., the different variants with their relations, is summarised in Fig. 4.

6.1. The CPCES algorithm

We present the general CPCES framework. Notice that this framework differs from the CPCES that we presented in our 2017 ICAPS paper [19], which is the gCPCES implementation of CPCES (cf. Subsection 6.3).

CPCES solves conformant planning problems by proposing candidate plans and then testing their validity. We use the notation $\mathfrak{V}$ to represent the set of invalid plans that were discovered so far. The new candidate plan is produced based on a

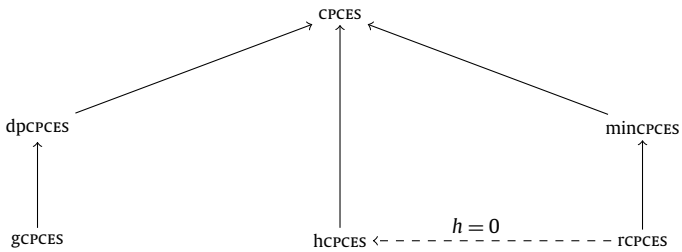


Fig. 4. Summary of the CPCES family: CPCES is a very general framework; dpCPCES and minCPCES are subclasses of CPCES implementations defined by properties over the sample; gCPCES, hCPCES, and rCPCES are implementations of CPCES; rCPCES can be interpreted as a special case of hCPCES when the heuristic used is always 0; dpCPCES and its (here, unique) subvariants have a bounded number of iterations equals to $|Tags(P)| + 1$ (Theorem 3), which is tighter than the upper bound $2^{|Tags(P)|}$ for the general CPCES (Theorem 2).

"counter-example set" of $\mho$, which is a collection of initial states that are sufficient to prove that all plans in $\mho$ are invalid. We also call the counter-example set a *sample* of the initial states.

Definition 11. Given a set $\mho$ of invalid plans, a *counter-example set* $\mathcal{B}$ of $\mho$ is a subset of initial states such that any plan in $\mho$ is invalid for at least one element of $\mathcal{B}$:

$$\forall \pi \in \mho.\ \exists s \in \mathcal{B}.\ \pi \notin \Pi(P^s).$$

Example. With reference to Fig. 1.a, we consider the following invalid plans:

- $\pi_1 = \varepsilon$ (no action);
- $\pi_2 =$ N*2 E*2;
- $\pi_3 =$ W N*2 E*2.

A counter-example set of $\mho_1 = \{\pi_1\}$ is $\mathcal{B}_1 = \{s_{0,0}\}$. $\mathcal{B}_1$ is not a counter-example set of $\mho_2 = \{\pi_1, \pi_2\}$, but $\mathcal{B}_2 = \{s_{0,0}, s_{1,0}\}$ and $\mathcal{B}_3 = \{s_{1,0}\}$ are. Finally neither $\mathcal{B}_2$ nor $\mathcal{B}_3$ are counter-example sets of $\mho_3 = \{\pi_1, \pi_2, \pi_3\}$ but $\mathcal{B}_4 = \{s_{1,0}, s_{2,2}\}$ is. ◇

CPCES is presented on Algorithm 1. The algorithm relies on two subroutines that are only briefly described here. We provide an actual implementation for these subroutines in the next two sections.

1. `produce-candidate-plan` is a procedure that computes a plan that is a valid for a sample. In effect it is a conformant planner. However, compared to the conformant planning problem that CPCES is trying to solve, the sample contains a small number of states (in the hundreds at most), which `produce-candidate-plan` can exploit. In practice our implementation of `produce-candidate-plan` reduces the conformant planning problem to classical planning.
2. `generate-counter-example-set` is a procedure that, given a collection of candidate plans, searches for a counter-example set of this collection. Our implementation relies on computing only one counter-example to the last plan inserted into the collection and adds it to the existing counter-example set. The counter-example is found using SAT.

Starting with an empty set of invalid plans $\mho$ and a counter-example set $\mathcal{B}$ of $\mho$ (in practice we use an empty sample), the following iteration is performed. First CPCES searches for a candidate plan π that is valid for all states in the sample. If no such plan is found, CPCES concludes that there is no plan. Otherwise, CPCES searches for a counter-example set of $\mho \cup \{\pi\}$. If no such counter-example set is found, CPCES concludes that π is a valid plan. Otherwise, π is added permanently to the set of invalid plans, and a new iteration is performed.

Example. We illustrate CPCES on the example of Fig. 1.a with the execution presented in Table 1. This example uses gCPCES (a specific implementation of CPCES described in more details in 6.3) that grows the counter-example set monotonically; this monotonic behaviour is not an artifact of all implementations of CPCES, as illustrated later, e.g., in Table 2. In this execution, CPCES starts with an empty sample and produces an empty candidate plan. The set $\{s_{0,0}\}$ is a counter-example of this candidate plan. CPCES produces a second candidate plan that is valid for $s_{0,0}$, namely E*2 N*2. CPCES generates the counter-example set $\{s_{0,0}, s_{1,3}\}$ for the two candidate plans. A new candidate plan is produced, which is valid for the two initial states of $\mathcal{B}_3$. Then the counter-example set $\{s_{0,0}, s_{1,3}, s_{4,4}\}$ is generated. The new candidate plan is then W*4 S*4 E*2 N*2. Finally no counter-example set is found for this last plan, which is returned by CPCES. ◇

Theorem 1. *If* `produce-candidate-plan` *and* `generate-counter-example-set` *are sound and complete, so is* CPCES. *That is, it finds a valid conformant plan if such a plan exists. Also, it eventually terminates if no conformant plan can be found.*

```
input: conformant planning problem P
output: a conformant plan, or no plan
𝔙 := ∅
ℬ := generate-counter-example-set(P, 𝔙)        // In general ℬ := ∅
loop
   π := produce-candidate-plan(P, ℬ)
   if there is no such π then
      return no plan
   end if
   ℬ := generate-counter-example-set(P, 𝔙 ∪ {π})
   if there is no such ℬ then
      return π
   end if
   𝔙 := 𝔙 ∪ {π}
end loop
```

Algorithm 1: The conformant planner CPCES.

Table 1
Example of execution of GCPCES on the problem of Fig. 1.a.

Iteration	Counter-example set	Candidate plan
1	$\mathcal{B}_1 = \{\ \}$	ε
2	$\mathcal{B}_2 = \{s_{0,0}\}$	E*2 N*2
3	$\mathcal{B}_3 = \{s_{0,0}, s_{1,3}\}$	S N*2 W*3 E*2
4	$\mathcal{B}_4 = \{s_{0,0}, s_{1,3}, s_{4,4}\}$	W*4 S*4 E*2 N*2
5	None	N/A

Proof of Theorem 1. Assume that CPCES returns **no plan**. This implies that there is no valid plan for the current sample $\mathcal{B}$. Now, $\mathcal{B}$ is either the empty set or a counter-example set for $\mathfrak{V}$, i.e., $\mathcal{B}$ is a subset of initial states. Therefore, since there is no valid plan for $\mathcal{B}$, there is no valid plan for the set of initial states and, indeed, for P.

Assume instead that CPCES returns a plan π. Then there is no counter-example set for $\mathfrak{V} \cup \{\pi\}$; in particular the set of initial states is not such a counter-example set. Notice however that there is a counter-example set for $\mathfrak{V}$. Therefore, from the definition of a counter-example set, π is valid for all initial states and, indeed, for P.

Termination is addressed in the next theorem. □

Theorem 2. *If* `produce-candidate-plan` *and* `generate-counter-example-set` *are sound and complete, then* CPCES *terminates after at most* $2^{|Tags(P)|}$ *iterations.*

Proof of Theorem 2. Let $\mathcal{B}_i$ be the counter-example set ($\mathcal{B}$ in the algorithm) at the ith iteration and π_i the candidate plan at this time step. Notice that π_i is valid for $\mathcal{B}_i$, while π_i is invalid for all sets $\mathcal{B}_j$ where $j > i$ since $\mathcal{B}_j$ is a counter-example set of a set $\mathfrak{V}_j$ that includes π_i. Therefore, $\Pi(P^{\mathcal{B}_i}) \neq \Pi(P^{\mathcal{B}_j})$ holds, i.e., the set of valid plans for the two samples $\mathcal{B}_i$ and $\mathcal{B}_j$ are different.

From Corollary 2 however, we know $\Pi(P^{\mathcal{B}}) = \bigcap_{t \in Tags(\mathcal{B})} \Pi(P^t)$ where $Tags(\mathcal{B})$ are the tags of $\mathcal{B}$, i.e., the set of valid plans for $\mathcal{B}$ is defined entirely by the tags of $\mathcal{B}$. In other words, since $\Pi(P^{\mathcal{B}_i}) \neq \Pi(P^{\mathcal{B}_j})$ holds, then $Tags(\mathcal{B}_i) \neq Tags(\mathcal{B}_j)$ holds too, for any $i \neq j$. Finally the set of tags of a sample is subset of the total set $Tags(P)$. The maximum number of iterations is therefore limited to the number of sets of tags, i.e., $2^{|Tags(P)|}$. □

6.2. Justification of conformant planning

At this stage, we want to stress a benefit of CPCES that will be useful in comparing the behaviour of different variants of CPCES.

A basis, as defined in Section 4, embeds all the important information about the initial state. Indeed, the conformant planning problem yields the same solution set with the original set of initial states or with the basis. In practice, this means that if a plan is suggested (e.g., by a human operator) it is fairly easy to verify its validity by simulating the plan from all states of the basis. Assuming the basis is small, this verification can be performed manually by the human.

In particular the following observation holds:

Observation 1. *If the counter-example set computed by* CPCES *is a basis, then the next candidate plan produced is valid.*

This is a simple consequence of the fact that the candidate plan has to be valid for the current sample. The benefit of this observation is that if the counter-example set is computed carefully (i.e., without adding too many states), then a small basis can be identified.

For many problems however, there is no basis of small cardinality. Furthermore, it is hard to find a subset-minimal basis. Indeed, while it is possible to construct a basis by selecting a set of initial states whose set of tags is precisely that of the problem, such bases may not be minimal: some tags may be "subsummed" by other tags. This is the case in the GridWorld example of Fig. 1.a, in which the tag $x = 1$ is subsummed by the combination of two tags $x = 0$ and $x = 4$.

The sample computed by CPCES can act as a "near-basis": the sample contains counter-examples for all plans of $\mho$. Presumably all plans that a human expert would come up with naturally exhibit similar shortcomings. Consequently the counter-example set acts as a good justification of the plan, providing a good, small, sample of important or representative initial states. We illustrate this point with a new example.

Example. Consider ONE-DISPOSE, a variant of DISPOSE where the agent can hold only one item at a time. In this problem, an agent must pick up each of I items (represented by $i \in \{1, \dots, I\}$) from one of the L locations (represented by $\ell \in \{1, \dots, L\}$), and then drop them at a special location. The agent can carry only one item at a time, and the initial location of the items is unknown. Actions are fairly straightforward (go to, pick up, drop), have no precondition, and just fail if they cannot be performed (for instance, the pick-up action fails if the agent is already holding an item).

One of the valid plans can be described by the following procedure:

for all item $i \in \{1, \dots, I\}$ **do**
  for all location $\ell \in \{1, \dots, L\}$ **do**
    go to location ℓ
    pick up item i
  end for
  go to the drop location
  drop item i
end for

This plan is valid because, for each item, the agent visits all locations and it will eventually succeed in picking up the item.

We now show that the basis of this problem includes all initial states. To this end, we pick a random initial state, and provide a plan that is valid for all but this state. Let q be the initial state where each item $i \in \{1, \dots, I\}$ starts in location ℓ_i, and consider the following plan π_q:

// First part
for all item $i \in \{1, \dots, I\}$ **do**
  for all location $\ell \in \{1, \dots, L\} \setminus \{\ell_i\}$ **do**
    go to location ℓ
    pick up item i
  end for
  go to drop location
  drop item i
end for
// Second part
for all item $i \in \{1, \dots, I-1\}$ **do**
  go to location ℓ_i
  pick up item i
  go to location ℓ_I
  pick up item I
  go to drop location
  drop item i
  drop item I
end for

Notice how, in the first part of π_q, the item i is not picked up if and only if it started in location ℓ_i. Also, in the second part of π_q, the item I cannot be picked up from location ℓ_I at iteration i if and only if the item i started in location ℓ_i. As a consequence, this plan is invalid for q because the agent will never be able to pick up item I. Furthermore, q is the only counter-example to π_q. We conclude that state q must be included in any basis, and that the basis is precisely all the initial states. $\diamond$

This observation seems concerning, because it implies that CPCES may require an exponential number of iterations of solve ONE-DISPOSE. Indeed, let $\{q_1, q_2, \dots, q_k\}$ be the set of initial states; if CPCES produces the plans $\pi_{q_1}, \dots, \pi_{q_k}$ in this order, it will also generate the counter-examples $q_1, \dots, q_k$ in the same order and it will only terminate after all initial states have been found.

In many examples however, and in this one in particular, such plans π_q are very specific (so, unlikely to be found by the planner) and suboptimal (so, unattractive for the planner). In practice, a valid plan is produced before π_q, which means that not all initial states are generated.

6.3. *dp*CPCES *and g*CPCES

In CPCES, the sample $\mathcal{B}$ implicitly represents the set of plans $\Pi(\mathcal{B}) = \bigcap_{s \in \mathcal{B}} \Pi(s)$ such that $\Pi(\mathcal{B}) \supseteq \Pi(P)$. Ideally, all other things being equal, we want the set $\Pi(\mathcal{B})$ to be as tight as possible, so that the next candidate plan is more likely to be valid for the conformant planning problem. (In particular, if $\mathcal{B}$ is a basis, i.e., $\Pi(\mathcal{B}) = \Pi(P)$, the next candidate plan is guaranteed to be valid.) A relevant property is therefore the fact that the set of valid plans for the sample decreases monotonically at each iteration of CPCES.

Definition 12. Belief state $\mathcal{B}$ *dominates* belief state $\mathcal{B}'$ if its valid plans form a subset of those of $\mathcal{B}'$: $\Pi(P^{\mathcal{B}}) \subseteq \Pi(P^{\mathcal{B}'})$.

A sampling strategy is *dominance-preserving* if at each iteration it keeps a counter-example set that dominates all the previous samples (or equivalently, that dominates the previous sample). We write dpCPCES the class of implementations of CPCES that use a dominance-preserving strategy.

gCPCES (*greedy* CPCES) is an instantiation of CPCES where the counter-example set is computed (Line 10) in a greedy fashion by adding a single initial state:

10.1: $s := \texttt{generate-counter-example}(P, \pi)$
10.2: $\mathcal{B} := \mathcal{B} \cup \{s\}$

Notice that this algorithm is correct, in the sense that if candidate plan π is invalid and if $\mathcal{B}$ is a counter-example set of $\mho$, then there is at least one initial state s for which π is invalid and for any such state $\mathcal{B} \cup \{s\}$ is a counter-example set of $\mho \cup \{\pi\}$.

We also notice that gCPCES is a dominance-preserving implementation of CPCES. Indeed in gCPCES the sample is a super set of the sample at the previous iteration; its set of solutions is therefore a subset of the set of solutions at the previous iteration.

Compared to CPCES we can prove a tighter bound on the number of iterations of dpCPCES.

Theorem 3. *The number of iterations of* dpCPCES *is at most* $|Tags(P)| + 1$.

Proof of Theorem 3. We use the notation $\mathcal{B}_i$ to denote the sample at the ith iteration and T_i the set of tags of $\mathcal{B}_i$. Because dpCPCES is dominant-preserving, we know that $\Pi(P^{\mathcal{B}_{i+1}}) \subset \Pi(P^{\mathcal{B}_i}) \subset \cdots \subset \Pi(P^{\mathcal{B}_0})$. Furthermore we know that for all i, $\Pi(P^{\mathcal{B}_i}) = \bigcap_{t \in T_i} \Pi(P^t)$. Notice that T_i is not necessarily a superset of T_{i-1}, in particular if some tags of T_{i-1} are useless.

For all i, we define $T'_i = T_i \cup T'_{i-1}$. This time, it holds that T'_{i-1} is a strict subset of T'_i. The relation is even strict because the set of solutions of $P^{\mathcal{B}_i}$ differs from that of $P^{\mathcal{B}_{i-1}}$. The sequence $T'_0, T'_1, \ldots$ has a length bounded by $|Tags(P)| + 1$, which bounds the number of iterations of dpCPCES. □

The result of Theorem 3 is very important in practice as we show in the experiment section that gCPCES converges faster compared to other implementations of CPCES.

6.4. *min*CPCES *and r*CPCES

We mentioned that the role of the sample is to reduce the number of valid solutions. A different ambition is to reduce the size of the sample, either for computational purposes (the production of a candidate plan is simpler if the size of the sample is smaller) or to provide a smaller justification.

Definition 13. A state $s \in \mathcal{B}$ is *redundant* for $\mho$ if all plans of $\mho$ that are invalid for s are also invalid for some other state of $\mathcal{B}$. A counter-example set $\mathcal{B}$ is minimal iff none of its states is redundant, since one such state could be removed from $\mathcal{B}$ without affecting its set of tags.

We call minCPCES an implementation of CPCES that computes a minimal counter-example set in Line 10.

We present rCPCES, an implementation of minCPCES. As gCPCES, rCPCES only differs from CPCES by instantiating its Line 10:

10.1: $s := \texttt{generate-counter-example}(P, \pi)$
10.2: $\mathcal{B} := \mathcal{B} \cup \{s\}$
10.3: $\mathcal{B} := \texttt{minimise\_sample}(P, \mathcal{B}, \mho)$

The method of Line 10.3 computes a subset of the current sample $\mathcal{B}$ that forms a minimal counter-example set of $\mho$. This procedure iteratively verifies whether the sample includes a redundant state. The minimisation procedure is presented in Algorithm 2. Notice that the procedure requires to check whether a given plan is valid from a given state, which is very simple to verify.

rCPCES guarantees that the sample is always minimal; it is therefore an implementation of minCPCES. Compared to a dpCPCES implementation, rCPCES proposes a very aggressive strategy to reduce the sample. This strategy can be counter-productive, as the counter-examples that are discarded may need to be re-added later.

```
procedure minimise_sample(P,B,Ʊ)
for all s ∈ B do
    if state_is_redundant(s,B \ {s},Ʊ) then
        B := B \ {s}
    end if
end for
return B

procedure state_is_redundant(s,S,Ʊ)
for all π ∈ Ʊ do
    if π is invalid for s then
        if ¬ has_counter_example(π, S) then
            return false {s is the only counter-example to π}
        end if
    end if
end for
return true

procedure has_counter_example(π,S)
for all s′ ∈ S do
    if π is invalid for s′ then
        return true
    end if
end for
return false
```

Algorithm 2: Minimising the sample.

Table 2
Example of execution of *r*CPCES on the problem of Fig. 1.a.

Iteration i	New counter-example	Counter-example set $\mathcal{B}_i$	Candidate plan π_i
1	N/A	$\mathcal{B}_1 = \{\ \}$	ε
2	$s_{0,0}$	$\mathcal{B}_2 = \{s_{0,0}\}$	E*2 N*2
3	$s_{1,0}$	$\mathcal{B}_3 = \{s_{1,0}\}$	E N*2
4	$s_{0,0}$	$\mathcal{B}_4 = \{s_{0,0}, s_{1,0}\}$	W E*2 N*2
5	$s_{2,0}$	$\mathcal{B}_5 = \{s_{2,0}\}$	N*2
6	$s_{0,0}$	$\mathcal{B}_6 = \{s_{0,0}, s_{2,0}\}$	W*2 E*2 N*2
7	$s_{3,0}$	$\mathcal{B}_7 = \{s_{3,0}\}$	W N*2
…	…	…	…

Example. We illustrate the execution of rCPCES on the example of Fig. 1.a as presented in Table 2. Starting with an empty sample and a null plan $\pi_1 = \varepsilon$, the counter-example $s_{0,0}$ is generated and added to the sample. The new plan π_2 is produced and the counter-example $s_{1,0}$ is generated. Notice that $s_{1,0}$ is a counter-example of π_1 too, so that the first counter-example, $s_{0,0}$ can be removed from the sample. A new plan π_3 is generated, for which $s_{0,0}$ is a counter-example. We notice therefore that $s_{0,0}$ needs to be reintroduced.

The execution continues, and we want to highlight the situation at iteration 6. We notice that the sample is $\{s_{0,0}, s_{2,0}\}$. This means that at this stage *r*CPCES was able to identify that $s_{1,0}$ was redundant and could be removed. ◇

This example illustrates two behaviours that we want to emphasise. First rCPCES can find counter-examples, ignore them, and then reintroduce them. We saw that the number of iterations of gCPCES is bounded by $|Tags(P)|+1$, which is bounded by the number n of initial states. On the other hand for some problem instances, rCPCES may require up to $2^{|Tags(P)|}$ iterations.

The second behaviour that we want to emphasise is the ability of rCPCES, and minCPCES in general, to disregard redundant states.

6.5. *h*CPCES

We argued in the previous subsections that dominance-preserving strategies require fewer iterations than minCPCES because a counter-example will never be added twice to the sample. We also argued that reducing the size of the sample is beneficial. It is possible to define a strategy that minimises the sample while preserving dominance. Computing such sample is however hard so we settle for a heuristic approach instead.

We say that a counter-example s is *useful* for $\mathcal{B}$ iff $\mathcal{B} \cup \{s\}$ restricts the set of solutions, i.e., the resulting set dominates the starting one: $\Pi(P_{\mathcal{B}\cup\{s\}}) \subset \Pi(P_{\mathcal{B}})$. If only non-useful states are removed from the sample, then these states will never be reintroduced into the sample since these states cannot be counter-examples to the plan produced from the resulting sample. However, deciding whether a state is useful in a sample is computationally hard as it can be reduced to finding a plan that is invalid for this state and valid in all other states.

```
procedure heuristic_refinement(P,ℬ,𝒱)
for all s ∈ ℬ do
  if state_is_redundant(s,ℬ \ {s},𝒱) then
    if h⁺(ℬ \ {s}) = h⁺(ℬ) then
      ℬ := ℬ \ {s}
    end if
  end if
end for
```

Algorithm 3: Heuristic Refinement.

We use the delete-free relaxation based heuristic to approximate the property of a useful counter-example. Specifically, given the sample $\mathcal{B}$ and the conformant planning problem $P_{\mathcal{B}}$, we first reduce $P_{\mathcal{B}}$ to classical planning as described in Section 7, and then perform a delete-free relaxation on the classical problem. We denote $h^+(\mathcal{B})$ the cost of its optimal solution. We say that s is *probably-useful* for $\mathcal{B}$ if $h^+(\mathcal{B})$ is strictly lower than $h^+(\mathcal{B} \cup \{s\})$. This gives us guarantees that the addition of s is at least contradicting all the plans of length i) strictly less than $h^+(\mathcal{B} \cup \{s\})$ and ii) longer than $h^+(\mathcal{B})$. These plans were less obviously contradicted before. The idea is to use this heuristic notion of usefulness to reconsider a reasonable subset of counter-examples discarded by rCPCES.

The sampling strategy that we use can therefore be explained as follows: instead of discarding all redundant counter-examples as does rCPCES, hCPCES first verifies their usefulness and keeps them if they are at least probably-useful. hCPCES is presented in Algo. 3.

Nicely, rCPCES can be seen as a special case of hCPCES: it suffices to substitute the h^+ function with a trivial admissible heuristic returning 0 for each input.

On the negative side however, hCPCES only approximates a dominance preserving sampling strategy: There could be counter-examples that the heuristic does not consider as useful, but that in principle would lead to the actual dominant counter-example set for that iteration.

Finally, for alleviating the computational burden within the sampling, we approximate h^+ using the h_{FF} heuristic.[2]

Example. We illustrate hCPCES on the example of Fig. 1.a. Consider that the set of invalid plans is $\mathcal{V} = \{\pi_1, \pi_2\}$ where $\pi_1 = \varepsilon$ and $\pi_2 = \text{N}^*2\ \text{E}^*2$. Assume that the counter-example found for π_1 is $s_{0,0}$; the counter-example found for π_2 is $s_{1,0}$. The second counter-example is also a counter-example to $s_{0,0}$, so hCPCES will consider whether $s_{0,0}$ can be removed.

Using delete-relaxation heuristics on $\mathcal{B} = \{s_{0,0}, s_{1,0}\}$, it is easy to determine that the following actions are necessary: 2 applications of N and 2 applications of E. A heuristic value for $\mathcal{B}$ is therefore 4. Considering sample $\mathcal{B}' = \{s_{1,0}\}$, the necessary actions are 2 applications of N and 1 application of E; the heuristic value for $\mathcal{B}'$ is therefore only 3. Consequently, hCPCES would keep the state $s_{0,0}$ in the sample. ◇

6.6. Optimal conformant planning

The planning community is often interested in finding optimal plans. We assume here that the cost function is state independent, i.e., regardless of the initial state, the cost of performing the conformant plan is identical. We show here that CPCES allows to compute the optimal plan.

The following result holds:

Theorem 4. *If the procedure* `produce-candidate-plan` *is optimal, then* CPCES *is optimal.*

Proof of Theorem 4. Assume that the plan π is returned by CPCES, and that there exists a cheaper valid plan π^*. This plan is optimal for the current sample $\mathcal{B}$, i.e., there is no plan π' that is both valid for $\mathcal{B}$ and cheaper than π. Therefore, by definition, π^* is invalid for $\mathcal{B}$ and is thus invalid for the conformant planning problem, which invalidates the assumption. □

7. Producing candidate plans

The first subroutine that CPCES requires is the procedure `produce-candidate-plan`, which computes a plan that is valid for a sample. This problem is indeed a conformant planning one, and any existing conformant planner could be used to solve it. However we propose to reduce this problem to classical planning which is much simpler to solve and offers us useful heuristics. The practicality of this reduction relies on the following property: the initial belief state of the problem (i.e., the sample) includes a very small number of states.

This property is supported by the following two observations. The first observation is that there often exists a small sample that represents the complete problem well enough that the candidate plan will be conformant. We refer to the example on ONE-DISPOSE from subsection 6.2.

[2] Let us recall that solving the delete-free relaxation problem optimally is NP-Hard [12].

The second observation is that the CPCES implementations that we consider add states one at a time. The procedure `produce-candidate-plan` is therefore called at least as many times as the size of the sample. Consequently, we cannot handle situations where the number of states in the sample is high. In practice, we assume that the cardinality of the sample is at most in the hundreds.

We solve the problem of computing a candidate plan by reformulating it into a classical planning problem, using the results of Section 5. Given the conformant planning problem $P = \langle \mathcal{V}, A, prec, eff, \mathcal{B}, \Phi_G \rangle$ where $states(\mathcal{B})$ equals $\{s_1, \ldots, s_k\}$ and k is small, we first perform the specialisation operation, which creates k copies P^{s_i} of P. Then we synchronise all these copies into a single classical planning problem $P_\otimes$. Notice that we do not perform the projection operation. This is because the resulting problem $P_\otimes$ is fairly small (since k is small) and the projection is therefore not strictly required. Furthermore, as we show in Section 10, the implementation in PDDL of the specialisation and merge is very simple.

The reformulation can be understood as follows: for every state variable v and every initial state $s_i \in states(\mathcal{B})$ we create a state variable $v_{/s_i}$ with the same domain as v, similarly to [34]. The set of variables $\{v_{/s_i} \mid v \in \mathcal{V}\}$ refers to the ith copy of P. The problem $P_\otimes$ is so defined that any plan is performed in parallel on each copy. Indeed, if action a requires the precondition $prec(a)$ to be satisfied in P the precondition $prec_\otimes(a)$ in $P_\otimes$ requires $prec(a)$ to be satisfied in each copy, i.e., $prec_\otimes(a) = prec_{/s_1}(a) \wedge \cdots \wedge prec_{/s_k}(a)$ where $prec_{/s_i}(a)$ is a rewriting of $prec(a)$ with each variable v replaced with $v_{/s_i}$. Similarly, if action a sets variable v to ℓ when condition $eff(\langle v, \ell \rangle)$ is satisfied, then $v_{/s_i}$ is set to ℓ whenever condition $eff(\langle v, \ell \rangle)_{/s_i}$ is satisfied.

Example. Consider the example of Fig. 1.a. Assume that the current sample includes the states $s_1 = s_{0,0}$ and $s_2 = s_{1,3}$. Then the reformulation includes four variables: $x_{/s_1}$, $y_{/s_1}$, $x_{/s_2}$, and $y_{/s_2}$. The interpretation of variable $x_{/s_1}$, for instance, is that it tracks the x position of the robot if it was initially in state s_1. The initial state of the reformulated planning problem is

$$(x_{/s_1} = 0) \wedge (y_{/s_1} = 0) \wedge (x_{/s_2} = 1) \wedge (y_{/s_2} = 3).$$

The action N has precondition $\top$ and, among others, the following conditional effects:

- $y_{/s_1}$ is set to 4 if $y_{/s_1} = 3$,
- $y_{/s_1}$ is set to 3 if $y_{/s_1} = 2$, etc.

The goal formula of the reformulated problem is

$$(x_{/s_1} = 2) \wedge (y_{/s_1} = 2) \wedge (x_{/s_2} = 2) \wedge (y_{/s_2} = 2). \diamond$$

We notice that the type of planning problems that we end up with have a very interesting structure. This is because the domain of the classical planning problem consists literally of k copies of the same set of state variables which are identical except for their initial values. It sounds reasonable to assume that specific enhancements exploiting this peculiar structure could be incorporated into existing classical planners so that they perform better for this particular class of problems.

8. Generating counter-example sets

The second subroutine required by CPCES is a procedure called `generate-counter-example-set` that computes a counter-example set to a set of invalid plans; if at least one of the plans is valid then the subroutine should fail to find the counter-example set. It is indeed possible to define such a procedure; we mentioned, however, that the implementations of CPCES we are interested in, namely GCPCES, RCPCES, and HCPCES, implement `generate-counter-example-set` on top of the subroutine `generate-counter-example`, which we discuss in this section.

Given a plan, `generate-counter-example` returns a counter-example to this plan, i.e., an initial state for which this plan is invalid, if it exists; if no such counter-example exists, i.e., if the plan is in fact valid, the procedure should fail to find a counter-example.

We show that the problem of determining whether a counter-example exists is NP-COMPLETE [21]. Consequently, we propose to use another NP-COMPLETE problem, namely propositional Boolean satisfiability (SAT), to solve it. Notice that this implies that verifying the validity of a plan is CO-NP-COMPLETE.

8.1. Complexity of finding a counter-example

Definition 14. Given a conformant planning problem P and a plan π, the *validity problem* is the problem of deciding whether π is valid for P. The *counter-example existence problem* is the problem of deciding whether there exists a counter-example to π.

We notice that the validity problem is the complement of the counter-example existence problem, because the plan π is valid iff there is no counter-example to this plan.

In order to show the complexity of validity problem, we first present a reduction from SAT to the counter-example existence problem. We use the following notation for the SAT problem:

- V represents the set of Boolean variables;
- a literal is either v or $\neg v$ where $v \in V$ is a variable;
- a clause c is a subset of literals;
- the SAT problem is represented by a set C of clauses.

The SAT problem C is *satisfiable* iff there exists an assignment $\alpha : V \to \{\bot, \top\}$ such that for each clause $c \in C$, there exists a literal $l \in c$ where either $l = v$ and $\alpha(v) = \top$ or $l = \neg v$ and $\alpha(v) = \bot$. SAT is the prototypical NP-COMPLETE problem.

Definition 15. Given a SAT problem $C = \{c_1, \ldots, c_m\}$ over the set of variables V, the *reduction* from C to a conformant planning problem is the pair $\langle P, \pi \rangle$ where $P = \langle \mathcal{V}, A, prec, eff, \Phi_I, \Phi_G \rangle$ is the conformant planning problem defined by

- $\mathcal{V} = \{f_v \mid v \in V\} \cup \{f_c \mid c \in C\} \cup \{f_g\}$,
  - the domain of each state variable f_v associated with a proposition variable v is $D_{f_v} = \{\bot, \top\}$;
  - and the domain of each other state variable f is $D_f = \{u, \bot, \top\}$;
- $A = \{a_c \mid c \in C\} \cup \{a_g\}$;
- $prec(a) = \top$ for all action $a \in A$ (all actions are always applicable);
- the non-trivial effects of a_c are:
  - $eff(a_{c_i})(\langle f_{c_i}, \top \rangle) = \bigvee_{v \in c_i} (f_v = \top) \ \vee \ \bigvee_{\neg v \in c_i} (f_v = \bot)$,
  - $eff(a_{c_i})(\langle f_{c_i}, \bot \rangle) = \neg eff(a_{c_i})(\langle f_{c_i}, \top \rangle)$,[3]

  and those of a_g are
  - $eff(a_g)(\langle f_g, \top \rangle) = \bigwedge_{c \in C} (f_c = \top)$,
  - $eff(a_g)(\langle f_g, \bot \rangle) = \neg eff(a_g)(\langle f_g, \top \rangle)$;
- $\Phi_I = \bigwedge_{c \in C} (f_c = u) \ \wedge \ (f_g = u)$;
- $\Phi_G = (f_g = \bot)$;

and $\pi \in A^\star$ is the plan

- $\pi = a_{c_1} \ldots a_{c_m} a_g$.

The reduction essentially works as follows.

- The state variable f_v stores the value of SAT variable v and never changes. We say that the state satisfies a clause if the value of the variables hence stored in the state satisfies the clause.
- The variable f_c is initially u (for unknown) and is set to $\top$ by action f_c if the clause is satisfied in the current state; otherwise it is set to $\bot$.
- The goal variable f_g is initially u (for unknown) and is set to $\top$ by action f_g if all variables f_c evaluate to $\top$; otherwise it is set of $\bot$.

The goal of the plan is to make sure that the SAT problem is not satisfied in the initial state. There is a bijection between the initial states and the possible assignments of the proposition variables.

Lemma 5. *The SAT problem C over the set of variables V is satisfiable iff its reduction $\langle P, \pi \rangle$ yields an invalid plan π to the conformant problem P.*

Proof of Lemma 5. Assume that the SAT problem is satisfiable. Then there exists a satisfying assignment α to C. Consider the initial state corresponding to α; for all clause c, after applying actions a_c, then f_c evaluates to $\top$. Therefore the application of a_g does not make f_g evaluate to $\bot$ and the plan is invalid.

Conversely assume that the SAT problem is unsatisfiable. Let s be any initial state and let α be the corresponding non-satisfactory assignment. Since α is unsatisfiable, there is at least one clause c that is not satisfied. Therefore the variable f_c will be assigned to $\bot$ and the action a_g will set f_g to $\bot$. Hence, the plan is valid for s, and for all states. Therefore the plan is valid for the conformant planning problem. □

Theorem 5. *The counter-example existence problem is* NP-COMPLETE *even if the formulas used in the problem description (precondition, conditions of effects, initial formula, goal formula) are simply conjunctions or disjunctions of assignments.*

Proof of Theorem 5. From Lemma 5 we know that there is a reduction from SAT to the problem of verifying that there exists a counter-example to a conformant planning problem; therefore this problem is NP-HARD.

[3] Notice that this can be written as a conjunction of assignments using De Morgan's laws.

Given the conformant planning problem and a plan, one can guess an initial state and verify in polytime whether the plan is valid. Therefore the problem is in NP. □

8.2. Finding a counter-example

The consequence of Theorem 5 is that SAT is an appropriate tool to compute a counter-example, which is why we used such an approach. The reduction to SAT is rather straightforward but we provide it here for completeness.

A counter-example of plan $\pi = a_1 \dots a_k$ is an initial state $s_0 \in I$ such that there exists a sequence of states $s_1, \dots, s_k$ where each s_i is defined as the result of applying action a_i on state s_{i-1} and either some action is non-applicable ($s_{i-1} \not\models prec(a_i)$) or the goal is not satisfied in the final state ($s_k \not\models \Phi_G$).

This is precisely how we define the SAT problem $\Phi_{P,\pi}$. For all timestep $i \in \{0, \dots, k\}$ we define the propositional variables

$$V_i = \{(v = \ell)_i \mid v \in \mathcal{V} \wedge \ell \in D_v\}.$$

Each SAT variable $(v = \ell)_i$ evaluates to true in the satisfying assignment iff the state variable v evaluates to ℓ in the state s_i. The variables of the SAT problem are the union of all these variables $V = \bigcup_{i \in \{0,\dots,k\}} V_i$.

We define a formula ϕ_a that represents how the state before applying action a relates to the state after applying a (without considering the precondition of a). We use the notation v' to represent the variable of the state after the action. Then the formula is defined as follows:

$$\phi_a = \bigwedge_{v \in \mathcal{V}, \ell \in D_v} (v' = \ell) \leftrightarrow \left(\mathit{eff}(a)(\langle v, \ell \rangle) \vee ((v = \ell) \wedge \bigwedge_{\ell' \in D_v} \neg \mathit{eff}(a)(\langle v, \ell' \rangle)) \right),$$

i.e., the variable v' evaluates to ℓ iff either the condition of effect is satisfied or v evaluated to ℓ and none of the conditions associated with an effect on v is satisfied.

Given a formula ϕ defined over assignments we use the notation $\phi[V_i]$ to represent the replacement of the assignments $(v = \ell)$ with $(v = \ell)_i$. Furthermore $\phi[V_i, V_j]$ replaces $v = \ell$ with $(v, \ell)_i$ and $v' = \ell$ with $(v, \ell)_j$.

The SAT problem is defined as

$$\Phi_{P,\pi} = \Phi_I[V_0] \wedge \bigwedge_{i \in \{1,\dots,k\}} \phi_{a_i}[V_{i-1}, V_i] \wedge \left(\bigvee_{i \in \{1,\dots,k\}} \neg prec(a_i)[V_{i-1}] \vee \neg \Phi_G[V_k] \right)$$

It should be clear that $\Phi_{P,\pi}$ represents precisely all the sequences of states $s_0, \dots, s_k$ defined as above and that

1. there exists a counter-example iff $\Phi_{P,\pi}$ is satisfiable;
2. the variables V_0 of any satisfying assignment represent one such counter-example.

Notice that there are several differences to SAT-based planning. First the plan is already known, which implies that the length k of the sequence of states is known from the start.

Second in SAT-based planning the initial state is known and the plan is unknown, which is the opposite of our instance. Consequently the problem is much simpler since the SAT problem is much more compact (the effect of all actions do not need to be embedded in the SAT problem). Furthermore, the backdoor of the SAT problem (i.e., the set of variables that it is sufficient to instantiate correctly so that finding the value of the other variables is polytime) is small: here the backdoor is the set of variables V_0 while in SAT-based planning it is the list of variables that represents the actions performed at each timestep.

Notice that our counter-example generator serves a second purpose: it can also act as a validator to conformant planning. This is quite important as we could not find any existing software that is efficient enough to verify the validity of the plans CPCES produced.

9. Learning information from previous iteration in form of macros

Consider the execution of any configuration of CPCES at some iteration, and assume the current (invalid) plan is π and the current sample is $\mathcal{B}$. In this section, we show how it is possible to encode the conformant planning for the current sample of states as a classical planning problem in a way that not only guarantees to find a plan conformant for that sample, but also exploits information from the previous iteration, thereby achieving some incrementality and thus speed-up in finding the new plan.

We observed in fact that, in some domains presented for the planning competition, a large part of the plan is preserved from one step to the other. This is due to the fact that the interdependence between the variables is often loose, so much that, sometimes, what has been done to achieve some subgoal does not need to be undone to achieve another subgoal. It is therefore possible that a substantial portion of the previous plan is still a valid and relevant trajectory, which can be

Table 3
Example of execution of gcpces with the lazy strategy on a smaller version of example of Fig. 1.a where the size of the grid is 3×3 and the goal is still in the middle. The follow shows the progress of gcpces by underlying the actions taken (over the course of the iterations) from the solution plan of the previous step.

Iteration	Counter-example set	Candidate plan
1	$\mathcal{B}_1 = \{\ \}$	ε
2	$\mathcal{B}_2 = \{s_{0,0}\}$	E N
3	$\mathcal{B}_3 = \{s_{0,0}, s_{1,0}\}$	W $\underline{\text{E}}$ $\underline{\text{N}}$
4	$\mathcal{B}_4 = \{s_{0,0}, s_{1,0}, s_{2,0}\}$	W $\underline{\text{W}}$ $\underline{\text{E}}$ $\underline{\text{N}}$
5	$\mathcal{B}_5 = \{s_{0,0}, s_{1,0}, s_{2,0}, s_{2,1}\}$	S $\underline{\text{W}}$ $\underline{\text{W}}$ $\underline{\text{E}}$ $\underline{\text{N}}$
6	$\mathcal{B}_6 = \{s_{0,0}, s_{1,0}, s_{2,0}, s_{2,1}, s_{2,2}\}$	S $\underline{\text{S}}$ $\underline{\text{W}}$ $\underline{\text{W}}$ $\underline{\text{E}}$ $\underline{\text{N}}$

exploited to speed up the reasoning in later iterations. This has been tried for other problems in the past through the use of macro-actions [41,27]. The questions we try to answer in this section are: is it possible to encode such a piece of knowledge for conformant planning? Can it be done while still using a classical planner to produce a candidate plan? Can it be done in a way that this information is also practical [32]?

In the following, given an invalid plan π, we report on an encoding of a classical planning problem that allows the classical planner to exploit fragments of π during the search. The way such pieces of plan are going to be used resembles the use of macro-actions. In our mechanism, however, we do not use macros as atomic actions because this would cause an exponential blow up when regressing conditional effects [38]. Instead we force/suggest the planner to follow specific patterns of action that have occurred in previous plans. More precisely, the planner is allowed to alternate between fragments of π and original actions. Intuitively, the more the planner chooses fragments of the previous plan, the more it will be reusing decisions made in a previous iteration.

Operationally, let $\pi = a_1, a_2, \ldots, a_n$ be a (invalid) plan; the encoding affects both the model of the actions and the initial state of the problem. First we add fresh variables and actions with the aim of capturing the *suggested* sequencing to the planner. The first added variable is p, whose domain is $D_p = \{0, |\pi|\}$. Intuitively, this variable is used to capture how many actions from the previous plan have been accumulated in the plan prefix (p is a short for *progress*). Then, we create, for each $a_i \in \pi$ a copy of it, namely a'_i, that can be executed only when the previous action in π, i.e., a'_{i-1} (if any) has been executed. This is achieved by preconditioning each $i-th$ action to $(p = i-1)$, and by setting $(p = i)$ as an effect. The resulting set of actions encompasses both new actions (the primed actions) and the original actions of the domain (the non-primed actions).

Then, we modify the initial state of the incumbent classical planning problem to enable the suggested sequence. Let φ_I be the state of the classical planning problem obtained by projecting the conformant planning onto the current sample, we conjoin φ_I with $(p = 0)$; i.e., $\varphi'_I = (\varphi_I \wedge (p = 0))$.

To force the classical planner to favor fragments from the previous plan, we also modify the actions in the domain by adding an extra action that has to be performed before any non-primed action. This action is called `pay`. The role of this action is to discourage the planner from applying actions that are not part of the suggested actions' sequence. Applying the action requires the new literal *toll*. The *toll* domain is $\{paid, not\_paid\}$. Initially, *toll* is set to *not_paid*. As an effect, `pay` sets such a literal to *paid*. Then, we substitute each action b_i that is not in π with a copy of it that includes as an additional precondition the literal $(toll = paid)$, and as an additional effect its negation $(toll = not\_paid)$.

In order to force the planner to follow on the sequence of the actions relevant for the previous planning problem, we devise three policies varying on the level of commitment the planner has on its previous decisions. In the less committed policy, namely lazy, we let the planner freely choose to select the original action or the action from the plan; of course any original action requires the `pay` action to be executed just before. This configuration is achieved by leaving the description presented above intact. Then we devise a more committed policy, namely cautious; cautious decreases the flexibility of the planner. It does so by disallowing the execution of original actions whilst the sequence of actions belonging to the plan has started. The third policy we devise, namely eager, is a fully committed strategy; eager forces the planner to eagerly initiate the previous plan of actions and, only after, try to deal with the current problem by appending suffixes of actions to it. Note that at least one action should be added to the plan. (See Table 3.)

The *Cautious* policy is obtained by forcing the `pay` not to be executable whilst the macro is under execution. To capture this we conjoin the precondition of the action `pay` with a disjunction, i.e., $(p = 0) \vee (p = |\pi|)$.

The third policy is obtained by removing the `pay` action, and forcing any original action to be executed only when $(p = |\pi|)$ is satisfied. The three strategies are summarized in Table 4.

Note that while *Cautious* and *Lazy* are complete policies (i.e., both return a solution if there is any), the latter is not. As matter of facts, the commitments done on the previous sequence of actions may result in a dead-end situation, which is motivated by the presence of some contingency of the initial state that was not handled.

Once the classical planner returns a plan (if any), this plan needs to be decoded. In particular, every `pay` action is dropped, as its occurrence is only used for computational purposes; then each primed-actions a'_i is substituted with the non-primed actions that a'_i has been generated from.

Table 4
Schema for macro-based classical planning translation using the EAGER, the CAUTIOUS or the LAZY commitment policy. The initial state is left intact but for the presence of the literal $(p = 0)$.

Domain Actions

$A' = \{\}$
for all $a \in A$ **do**
  $a' = a$
  if strategy is not *eager* **then**
    $prec(a') = prec(a) \wedge (toll = paid)$
    $eff(a') = eff(a) \cup \{(toll = not\_paid)\}$
  else
    $prec(a') = prec(a) \wedge (p = |\pi|)$
  end if
  $A' = A' \cup \{a'\}$
end for

Plan-Based Actions

for $i \leftarrow 1$ to $|\pi|$ **do**
  $b' = b_i$
  $prec(b') = prec(b) \wedge (p = i - 1)$
  $eff(b') = eff(b) \cup \{(p = i)\}$
  $A' = A' \cup \{b'\}$
end for
if strategy is not EAGER **then**
  pay = *new* Action()
  if strategy is CAUTIOUS **then**
    $prec(pay) = (toll = not\_paid) \wedge ((p = 0) \vee (p = \pi))$
  else
    $prec(pay) = (toll = not\_paid)$
  end if
  $eff(pay) = \{(toll = paid)\}$
  $A' = A' \cup \{pay\}$
end if

Note that the decision of using an artificial extra action to discourage the planner from using original actions is only pragmatic. On a more general level, one would in fact ascribe a positive cost to only the actions taken from the original domain, and perhaps give a zero cost to all the actions that are part of the incumbent plan. Unfortunately, our experiments indicate that classical planners using action costs and conditional effects in a sensitive manner were substantially slower, and therefore left our experimental analysis with as explained above. It is nevertheless possible that a better support for action cost will enable a more scalable and effective compilation.

10. Experimental evaluation

In this section, we report on our experimental analysis done to study the practical implications of the CPCES framework. The analysis has been carried out for answering the following questions: can there be a system that can effectively handle the classical planning problems we generate and the validation problem together? Does CPCES scale up to large conformant planning problems, as those presented in the International Planning Competition[4]? Is the upper bound on the number of iterations on the various configurations of CPCES an actual critical aspect from a practical perspective, or is it only a very worst-case complexity result? How does CPCES compare with recent conformant planners? In order to answer these questions we developed a planner called, unsurprisingly CPCES. CPCES orchestrates two subsystems, namely a SAT-Solver and a Classical Planner. In the next section we provide details on this implementation, and then we study the behaviour of CPCES in different settings over a set of benchmarks instances, and against other conformant planners.

10.1. Implementation

The implementation of CPCES is fairly simple, yet it requires some machinery to handle the two main problems it needs to deal with; that is (i) the classical planning problem arising from the conformant to classical planning rewriting explained in Section 7, and (ii) the counter-example finding procedure explained in Section 8.

Our planning problem is specified using the PDDL language [31], of which we make heavy use of the so called ADL constructs. In particular, the first-order constructs (i.e., universal quantification and action schema) and conditional effects. Although our theory builds on state variables, we decided to use PDDL for the great availability of efficient classical planners. In the PDDL specification we create a new type, called `interpretation`. Each `interpretation` object refers to a possible interpretation of the initial states. Starting from a sample $\mathcal{B} = \{q_1, \ldots, q_k\}$, we generate k interpretation values `{i1,...,ik}`, one for each state in $\mathcal{B}$. Then, we reformulate each lifted predicate `pred(?p1,...,?pj)` in `pred(?p1,...,?pj,?i)` where `?i` is an interpretation. Intuitively, we add a dimension to the planning problem, which is used to model every state in the sample. Action preconditions are reformulated using the `forall` construct over the interpretations, so as to enforce that each precondition is satisfied regardless from the given initial condition or, analogously, from the given interpretation. Similarly, the effects are defined using a `forall` operator. Universal quantification here guarantees that each conditional effect that can arise from the given interpretation of the initial state is considered in the actual action effects. By keeping the interpretation as a variable, our domain description is perpetual across the CPCES iterations. What we only need to change from one iteration to the other is the number of interpretation objects at hand, and their implications on the initial state. Fig. 5 presents an action schema (called multi-interpretation action to highlight the multiplicity of initial states it has to deal with) that is generated as for the description above.

[4] http://icaps-conference.org/ipc2008/probabilistic/wiki/index.html.

```
(:action drop
 :parameters (?o  - obj ?p  - pos )
 :precondition
   (and (located ?p) (trash_at ?p)))
 :effect (and
   (when (holding ?o)
     (and (not (holding ?o))
          (disposed ?o))))))
```

(a) Original action.

```
(:action drop
 :parameters (?o  - obj ?p  - pos )
 :precondition (forall (?i - interpretation)
   (and (located ?p ?i) (trash_at ?p ?i)))
 :effect (and (forall (?i - interpretation)
   (when (holding ?o ?i)
     (and (not (holding ?o ?i))
       (disposed ?o ?i))))))
```

(b) Multi-Interpretation action.

Fig. 5. Illustrating our reformulation from conformant planning to classical planning with the action `drop(?o,?p)` from the DISPOSE domain. In the reformulated version, the precondition and the effects are distributed over all interpretations through the use of the `forall` PDDL construct.

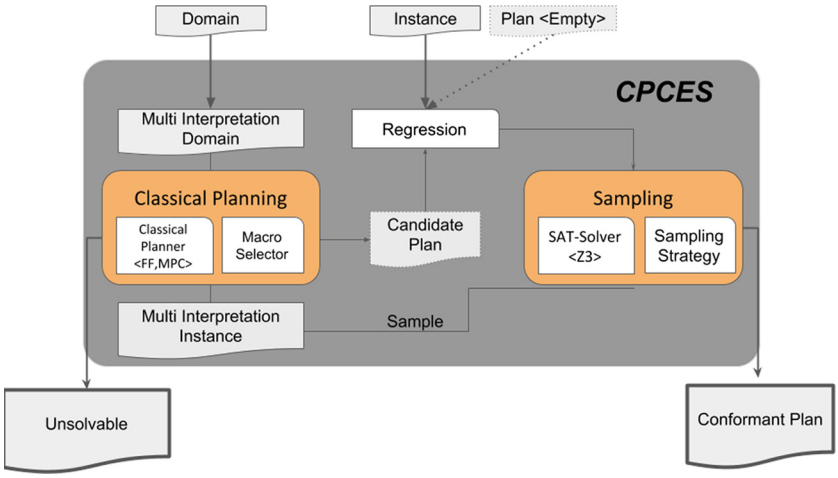


Fig. 6. This is the overall software architecture implementing the various aspects of CPCES, in all its incarnations.

For the validity testing, we use a SAT encoding expressed with the SMTLIB language [3]. SMTLIB's constants are used to represent the boolean variables of our problems. Those variables encode (much as it is done in SAT-based planning [39]) both the actions executed at each step (note here that we already know which actions are going to be executed), and the variables for each step of the execution. The actual unknowns of the problem (the ones the SAT problem has to find a model for), are the variables representing the initial state.

Similarly to PDDL, using a standard language for representing the validity problem allows us to be independent on the specific solver used. For our experiments, we used FF [24] and MPC [39] as classical planners, and Z3 [17] as a SAT-Solver.[5] Despite the large availability of classical planners from the International Planning Competitions, we had to restrict our attention on FF and MPC. They were in fact the only planners providing effective support for problems exhibiting a large number of universal quantifiers and objects combined with conditional effects. More recent classical planners are all based on the FAST-DOWNWARD Python preprocessor, which turned out to be very inefficient in dealing with our classical planning problems. The majority of our evaluation is performed using FF because it is generally faster than MPC on our problems. To complete the evaluation, we also provide findings using MPC as classical planner. Note that, within the CPCES framework, this can be obtained by simply changing the executable target of the PDDL classical planner.

The various incarnations of CPCES (that is GCPCES, HCPCES, and RCPCES) are implemented in JAVA, and the external systems (e.g., the classical planner) are called as black-boxes. Fig. 6 summarises the overall orchestration.[6]

10.2. Setting of the experiments

The experiments were run on an Intel(R) Xeon(R) CPU E3-1240 v3 @ 3.40GHz, running with 16GB of RAM on a Linux-based distribution. Every problem instance has been allocated 4800 seconds; we have not used any memory cut, so the system potentially uses all the 16GB at disposal.

[5] Note that Z3 is not a pure SAT-Solver, but is a SMT-Solver, so it can be used for more expressive problems as well, as for instance problems containing numeric state variables, numeric constraints and so forth.

[6] The code of our implementation is publicly available at https://bitbucket.org/enricode/cpces/.

10.3. Benchmarks

The benchmark domains considered in our experimental evaluation are taken from the International Planning Competition IPPC-2008, which was the last official competition with a track specific for conformant planning, from some of the benchmarks used in the T1's suite, and the two grid-based domains we used as examples. These domains have been selected with the aim of having a good distribution of cases with low and high width. We expect CPCES to perform nicely on problems with low width; yet, since CPCES is not constructed to eagerly exploit it, we expect CPCES to behave decently over also domains with higher width. To these domains, we also added some of their reformulations that have been presented for the DNF planner [44]. In the following we provide a brief description of the domains and instances used:

- DISPOSE. This domain models an agent that can move in a grid and needs to collect and then dispose of a set of items into a predefined location. The uncertainty in this domain refers to the initial position of the objects, which can be in any cell in the grid. Instances here scale with the size of the grid, and the number of objects to be disposed of. The size of the grid in the instances spans from 4×4 to 16×16, while the number of objects from 1 to 3. We have a total of 11 instances.
- ONE-DISPOSE. This is basically the previous domain, with the exception that the agent can carry at most one object per time. This small variation makes the width of the problem larger than one, therefore harder for conformant planners that specifically exploit this decomposition. The size of the grid in the instances spans from 4×4 to 6×6, while the number of objects from 1 to 3. We have a total of 10 instances.
- BLOCKSWORLD. This is the uncertainty version of the famous blocks-world problem, extensively studied in the planning community. Uncertainty in this problem comes from the configurations of the block, and instances scale on the number of blocks. The suite includes from 2 to 5 blocks instances. Note that the original benchmark suite contains an unsolvable problem (the instance with 5 blocks). We keep both the unsolvable and a solvable version of the same, so as to test the capability of our system in handling unsolvable instances.
- BOMB. This domain models an agent that is in charge of disarming a number of bombs, whose initial state is unknown (i.e., the agent does not know whether the bomb is armed or not). In order to disarm these bombs, the agent needs to dunk them in one of the available toilets, which needs to be unclogged first if it has been used. Instances here scale with the number of bombs, and the number of toilets at disposal. Our benchmark suite instances contains from 20 to 100 bombs, and from 1 to 100 toilets, for a total of 10 instances.
- LOOK-AND-GRAB. This domain is similar to DISPOSE; also in this case the agent needs to displace objects from an initial unknown position to a final position within a grid. Differently from DISPOSE, LOOK-AND-GRAB makes it possible for the agent to collect more objects per time. At each step of the plan execution, this requires a more sophisticated reasoning over the belief since it is not known whether the agent has some other objects in the hand or not. Because of this more profound interaction among the variables representing the position of the objects, this domain is known to lead to instances with a have a width larger than one. Instances scale on the number of cells in the grid (we have instances with 4×4 and 8×8 cells), the extension of the agent in taking objects which are not in her position but in the contiguous cells next to her (from 1 to 3), and the number of objects to be collected (from 1 to 3). We have a total of 18 instances.
- COINS. This domain models an agent that has to pick coins in different locations; each location can be accessed using elevators or moving right and/or left on a floor. Initially, the agent does not know whether there is a coin in a position, and does not know the location of the elevators. Instances scale on the number of elevators (from 2 to 5), number of floors (from 4 to 10), number of positions (from 4 to 11), and number of coins (from 4 to 11). The total number of instances is 9.
- RAO'S KEYS. This domain models the problem of collecting a number of keys whose initially unknown position is under some particular light. Each light is associated to a location and behind a gate. To open a gate the agent has to use a key. However, also whether a given key opens a given gate is unknown; therefore the agent has to try all keys before being sure that a gate is opened. Instances scale with the number of lights, from two to four. For this problem we have two solvable instances, and an unsolvable one.
- UTS. This domain describes the problem of visiting a set of nodes of a graph, without knowing the initial position of the agent. Instances scale on the number of nodes, from 2 to 60. In total, 15 instances are collected.
- GRID-EMPTY and GRID-WALL. These domains are extension of the grid-based example with the swamp presented in the paper. Instances here scale on the number of cells in the grid. We have grid of size 3×3 up to 20×20 for a total of 18 instances for each variant. In GRID-WALL, we arrange always a wall next to the sink, so as to make the problem solvable. Instead, in GRID-EMPTY, instances can be both solvable or not depending on whether the sink is located next to the border (in this case the instance is solvable) of the grid or not. For these domains we also offer a problem instance generator.

To these domains we also add the conformant planning problems proposed by the authors of DNF [44]. These domains have been built to further challenge the state of the art conformant planners; some of them are slight reformulations of previously presented domains. They were considered to assess how they affect CPCES performance. In particular, following DNF's suite nomenclature, we have: NEW-DISPOSE, NEW-RING, NEW-UTS and NEW-PUSH. Differently from its original version, NEW-DISPOSE features disjunctions in the goal set, which results in the problem accepting more plans. NEW-RING is the

Table 5
Coverage Analysis. The number of each cell corresponds to the number of instances solved for a given domain. Solved means that the planner proved the problem solvable or not solvable.

		$\|I\|$	Width	gCPCES	hCPCES	rCPCES
SAT	Dispose	10	1	4	4	0
	BlocksWorld	4	>1	4	3	3
	Bomb	9	1	5	5	2
	UTS	15	1	18	18	17
	Look-and-Grab	18	>1	7	4	0
	Coins	9	1	13	13	6
	One-Dispose	10	>1	8	8	1
	Rao's Keys	2	>1	2	2	1
	Grid-Empty	4	>1	4	4	4
	Grid-Wall	18	>1	18	14	16
	New-Push	10	>1	6	6	4
	New-Dispose	6	1	0	0	0
	New-Ring	11	1	11	11	11
	New-UTS	10	1	5	5	2
UNSAT	Grid-Empty	14	>1	14	14	14
	Rao's Keys	1	>1	1	1	1
	BlocksWorld	1	>1	1	1	1

problem of closing and locking a number of windows. The new version introduces uncertainty in the locker status, and accepts plans that ensure each window is closed, and only in case its locker is not damaged, also lock it. New-UTS is a constrained version of UTS. Each node but the root node can be visited at most once. New-Push models an agent whose goal is to pickup a number of objects whose locations are initially unknown. The agent can move only when the destination position is free of objects, and in order to empty it, she needs to push objects around.

Using these domains, depending on the evaluation carried on (see below), we study different dimensions:

- Coverage: the number of instances solved by a system on a specific domain. This applies to all systems tested.
- Planning time: the overall run-time of the planner in providing a plan or the guarantee that such a plan does not exist. This applies to all systems tested; note that only CPCES returns a meaningful answer for this latter situation, i.e., the sample which certifies from which states in the belief there cannot be a valid conformant plan.
- Iterations: the number of iterations that CPCES had to perform before termination.
- Samples: the size of the sample (i.e., the number of initial states) for the last call of CPCES before termination.
- Plan-Length: the number of actions of the found conformant plan. This applies to all systems tested.

Planning time, Iterations, Samples and Plan-Length will be often given as averages over instances solved by all systems under comparison. Care will be taken to evaluate this piece of information only when there is a reasonable set of instances.

10.4. Results on CPCES*: assessing the sampling strategies*

In this evaluation we focus our attention on coverage, planning time, plan length, iterations and the size of the last sample for the entire set of domains. Intuitively, the planning time and plan length are the obvious performance indicators that give us a measure of the speed of the system in providing a solution, and of the quality of the solution being found, respectively. The last sample measures the number of initial states considered in the last planning iteration (for solvable instances); for unsolvable instances, this is the number of initial states that has sufficed to prove the unsolvability of the problem. This number indicates two aspects: the size of the classical planner problems that CPCES has to deal with, and the succinctness of the justification to prove unsolvability. That is, the smaller the better. The number of iterations measures the number of cycles that CPCES has to execute before ending the process.

Table 5 shows a high level overview of the coverage obtained using the three sampling configurations of CPCES over all the domains under investigation. The table splits the case that are known to be solvable (SAT in the table), from the ones that are known to be unsolvable (UNSAT in the table). As it is possible to observe, gCPCES dominates the other sampling strategies despite its simplicity. This provides evidence on the theoretical usefulness of the dominance preserving property exhibited by gCPCES. Note that, gCPCES solves all instances that are solved by hCPCES and rCPCES. In other words, there is no instance that is solved by hCPCES but not by gCPCES, and there is no instance that is solved by rCPCES and not by CPCES. The survival plot of Fig. 7 shows gCPCES solving instances much faster than the other systems overall; convergence for gCPCES is quicker and, although the size of classical planning formulations are generally larger than those generated by the other sampling strategies, the number of iterations of this configuration is in general lower, and this translates in solving instances quicker. Note how gCPCES tails off quite quickly, i.e., after 1000 seconds over a budget of 4800 seconds. The other strategies took a bit longer to converge, but tailed off anyway. This supports the hypothesis for which is unlikely that both hCPCES and rCPCES (but mostly rCPCES) will reach the coverage we obtain in CPCES from a practical standpoint, even if a much larger amount of time is given to the planner.

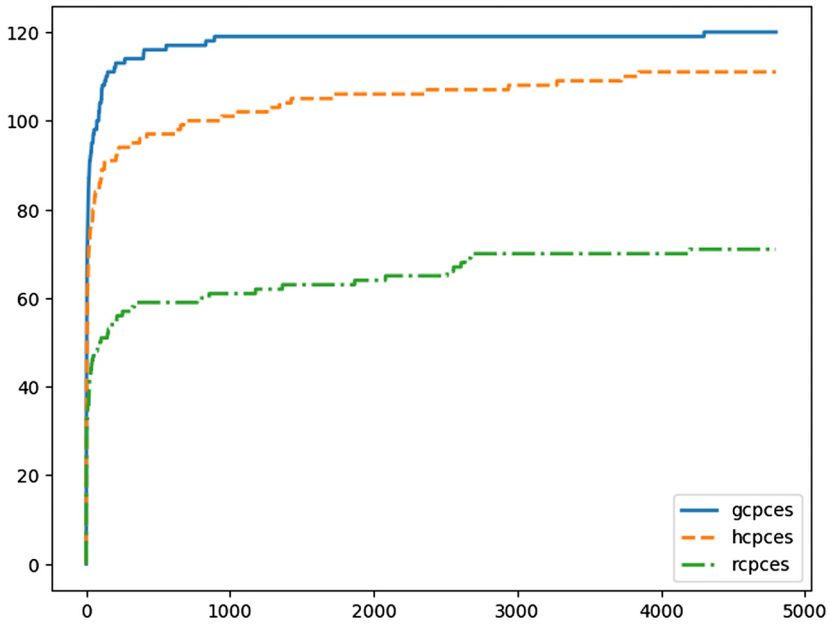


Fig. 7. Survival plot for the three sampling strategies of CPCES. x-axis is the number of seconds given to the planner, y-axis is the number of instances solved. Higher is better.

Table 6
Planner and SAT solving time, expressed in seconds, spent by GCPCES during the solving of the conformant planning problem. Third column reports the ratio between classical planner time and SAT solving time. Results are average over all solved instance for each domain. The remaining running time is due to the machinery built to handle all the interactions between the two phases. A similar trend can be observed for HCPCES and RCPCES.

Domain	Planning Time	Classical Planner Time	SAT Solving Time	CP/ST
DISPOSE	47.01	31.43	5.00	6.28
BLOCKSWORLD	184.25	178.86	0.98	182.151
ONE-DISPOSE	117.78	112.14	1.43	78.42
LOOK-AND-GRAB	29.21	25.44	0.52	48.92
BOMB	125.34	93.53	9.40	9.95
UTS	16.86	10.70	1.80	5.94
COINS	4.58	1.89	0.57	3.32
RAO'S KEYS	2.15	0.84	0.20	4.20
GRID-EMPTY	0.88	0.24	0.05	4.80
GRID-WALL	38.48	34.36	0.68	50.53
NEW-PUSH	127.66	123.36	0.81	152.30
NEW-RING	0.99	0.14	0.03	4.67
NEW-UTS	17.74	15.41	0.48	32.10

Comparing the strategies, it is worth noticing that, while GCPCES is clearly superior over solvable instances, its performance on unsolvable instances are comparable, and sometimes even substantially inferior to that of the other samplings. For instance, in BLOCKSWORLD, RCPCES is the fastest configuration, gaining one order of magnitude of speed up. We will see this aspect better in the sub-sections below. None of the sampling strategies have worked with the NEW-DISPOSE domain; NEW-DISPOSE features disjunctive goals, and FF is not good at handling them. All runs stopped after a couple of iterations. To explore this issue better, we reformulate the domain by compiling all the disjunctive goals away by using artificial actions, each of which meant to enable the desired disjunct in the goal. This new version scales better with FF. We believe, therefore, that this issue is not a problem with CPCES method, and is likely avoidable with a more careful problem formulation, or through the adoption of some next generation classical planner.

This high level picture terminates with Table 6. In this table we show the computational time spent by the classical planner, and that spent by the SAT solver to provide the counterexamples. The results are averaged over all instances solved by GCPCES, and the table shows the results domain per domain. The conclusion is quite interesting. As matter of facts, almost the entire computational time is spent by the classical planner, and the effect of the SAT solver is almost negligible in the entire process. As the classical planner can, therefore, be the bottleneck of our approach, our experimental analysis will also investigate the implications of using different classical planners; we focus on FF and MPC.

Table 7
Focus on the comparison between gcpces and hcpces over instances solved by both configurations.

		Planning Time		Samples		Iterations		Plan Length	
	Domain	gcpces	hcpces	gcpces	hcpces	gcpces	hcpces	gcpces	hcpces
SAT	Dispose	47.01	725.23	37	32	38.00	37.50	167.25	165.50
	BlocksWorld	3.47	61.76	10.67	8.33	11.67	11.33	29.00	29.67
	One-Dispose	117.78	713.19	29.4	14.8	30.40	141.00	81.40	71.00
	Look-and-Grab	29.21	232.11	9.61	6.61	10.61	16.67	49.67	44.67
	Bomb	34.17	915.96	42.75	42.75	43.75	173.75	77.75	77.75
	UTS	16.86	247.82	15.38	15.38	16.38	16.38	49.69	49.08
	Coins	4.58	37.99	11	9.25	12.00	12.38	88.00	87.88
	Rao's Keys	2.15	9.78	13	11.5	14.00	15.50	35.50	33.50
	Grid-Empty	0.88	2.89	5	2.75	6.00	8.00	17.50	17.50
	Grid-Wall	13.61	695.78	25	17.5	26.00	344.57	55.29	55.29
	New-Push	127.66	646.04	15.5	10.67	16.50	28.00	56.67	53.83
	New-Ring	0.99	0.67	1	1	2.00	2.00	29.00	29.00
	New-UTS	17.74	25.69	12.8	12.8	13.80	13.80	37.40	37.40
UNSAT	BlocksWorld	95.12	27.06	52	27	52	69	NA	NA
	Rao's Keys	0.3	0.3	1.0	1.0	1.0	1.0	NA	NA
	Grid-Empty	6.5	4.0	20.64	6.36	20.64	18	NA	NA

More details on the behaviour of cpces using the different sampling strategies are reported on Table 7 and 8 where we intersect the instances solved by gcpces and hcpces, and then by gcpces and rcpces. A pairwise comparison follows.

gcpces vs hcpces. Runtime wise, gcpces largely outperforms hcpces. To be noted though is the quality of the plans obtained by the two configurations. On a couple of domains, namely One-Dispose and Look-and-Grab, hcpces computes much shorter plans. A slight improvement can also be observed in other domains (i.e., New-Push, UTS and Coins). It seems there is a correlation between the average number of samples maintained and the resulting plan length. The larger is the size of the sample, and the longer is the plan computed by the classical planner. This is due to the sub-optimality of the planner being used (if it was optimal we would not observe any difference between the two strategies). Even though it is hard to extract a general conclusion for this, we conjecture this behaviour is due to the goal agenda mechanism used by FF in its main search engine. The goal agenda mechanism implemented in FF consists in assigning an order for the achievement of the goals. While this makes the planning generation somehow faster, it introduces some degree of suboptimality because it does not optimize situations for which a given sub-plan can achieve multiple goals. The other details are almost self-explanatory; the number of iterations for hcpces is consistently larger (with the only exception of SAT instances of Grid-Empty; here in two instances out of 4, hcpces required one iteration less than gcpces), and this is mostly the reason why the performances of hcpces are inferior to those of gcpces.

gcpces vs rcpces. Also in this case we observe a substantial degradation of the runtime performance. Similarly to hcpces, rcpces can retract some state from the sample set; while this makes the classical planning problem smaller, it breaks the dominance preserving property of gcpces. By looking at the experimental data, as for the comparison between gcpces and hcpces, we observe a correlation between smaller sample and plan size, but in a less evident way. Differently from the other cases, with rcpces we observe that also the number of iterations, and therefore the number of modifications that are performed on the plan along the entire functioning of cpces, seems to affect the final plan size. For instance, in New-Push, although rcpces average sample size is substantially smaller than that of gcpces, the average size of the plans is higher in rcpces. As for the previous comparison, it is hard to extract a general conclusion from this, and for this very aspect we do not have a strong conjecture. The classical planner used (FF) is not informed on the various planning problems it has to solve, yet there is some hidden relation between the search space under investigation and the evolution of the sample that leads the classical planning to favor more expensive (in terms of plan length) trajectories.

On another dimension, a very interesting aspect to be highlighted here is the substantial speed up that we obtain over the unsolvable instances under evaluation. Smaller samples produce smaller classical planning problems, and therefore rcpces turns out to be much faster in proving their unsolvability. Note that, beside Rao's Keys, both BlocksWorld and Grid-Wall are far from being simple to prove unsolvable; this is somehow highlighted by the number of iterations that had to be taken into account in the tested instances. Yet, FF behaved quite well, although deciding unsolvability is often very hard for satisficing classical planners.

We can conclude this first analysis with a clear take-home message. If one has to choose from a sampling strategy she should know that over solvable instances gcpces is probably the more stable system, as well as, on the other hand, rcpces is the choice to make to get better performances in showing unsatisfiability. Yet, if the planning problem is not too big, it would make sense to use rcpces and hcpces so as to ameliorate the quality of the resulting plans, or better, running several algorithms in parallel as to exploit complementarity at its best.

10.5. Results on cpces*: assessing the use of macro-based policies*

In this section we study the effect of the macro policies devised in Section 9. As the selection of the macro is independent on the sampling used, the next subsections study the behaviour of the system for each sampling strategy. Our analysis

Table 8
Focus on comparison between gcpces and rcpces over instances solved by both configurations. Dispose and Bomb have been dropped from the comparison as rcpces did not solve any instance in those domains.

		Planning Time		Samples		Iterations		Plan Length	
	Domain	gcpces	rcpces	gcpces	rcpces	gcpces	rcpces	gcpces	rcpces
SAT	BlocksWorld	3.47	1537.59	10.67	6	11.67	4164.67	29.00	27.33
	One-Dispose	1.72	41.94	12.00	5	13.00	470.50	34.50	30.00
	Look-and-Grab	20.40	246.96	8.35	3.59	9.35	189.12	40.94	35.18
	UTS	1.26	707.20	7.00	7	8.00	910.00	20.00	20.67
	Coins	1.90	34.87	7.00	4	8.00	384.00	31.00	28.00
	Rao's Keys	0.80	1.67	4.00	4	5.00	16.00	16.00	16.00
	Grid-Empty	0.88	1.28	5.00	2	6.00	12.50	17.50	17.50
	Grid-Wall	22.97	588.82	28.31	20.56	29.31	654.75	62.25	62.25
	New-Push	4.71	16.60	11.25	6	12.25	80.25	35.50	38.00
	New-Ring	0.99	0.73	1.00	1	2.00	2.00	29.00	29.00
	New-UTS	2.30	87.30	7.00	7	8.00	520.00	20.00	20.00
UNSAT	BlocksWorld	95.12	7.85	52	1	52	164	NA	NA
	Rao's Keys	0.3	0.3	1.0	1	1.0	1.0	NA	NA
	Grid-Empty	6.5	6.22	20.64	4	20.64	41.21	NA	NA

Table 9
Coverage analysis of gcpces with the three macro policies.

	gcpces	gcpces		
Domain	no-macro	lazy	cautious	eager
Dispose	4	7	5	4
BlocksWorld	4	3	4	0
One-Dispose	5	5	9	9
Look-and-Grab	18	18	18	18
Bomb	7	7	6	1
UTS	13	13	13	0
Coins	8	8	8	0
Rao's Keys	2	2	2	2
Grid-Empty	4	4	4	0
Grid-Wall	18	10	18	0
New-Push	6	5	5	1
New-Dispose	0	0	0	0
New-Ring	11	11	11	11
New-UTS	5	6	5	0

Table 10
gcpces with different macro policies. Instances considered are those solved by every system under this experimental evaluation. Eager macro policy excluded for lack of coverage.

gcpces	Planning Time			Samples			Iterations			Plan Length		
Domain	NM	L	C	NM	L	C	NM	L	C	NM	L	C
Dispose	47.01	42.00	142.40	37	38.75	35.75	38.00	39.75	36.75	167.25	93.25	274.50
BlocksWorld	3.47	12.55	9.74	10.67	12.00	7.33	11.67	13.00	8.33	29.00	32.33	74.67
One-Dispose	117.78	817.70	16.89	29.4	40.00	26.20	30.40	41.00	27.20	81.40	94.80	134.00
Look-and-Grab	29.21	170.81	24.83	9.61	10.56	9.11	10.61	11.56	10.11	49.67	46.67	67.22
Bomb	66.64	105.66	265.19	52	52.00	52.00	53.00	53.00	53.00	90.83	90.83	90.83
UTS	16.86	31.63	16.44	15.38	15.38	15.38	16.38	16.38	16.38	49.69	44.15	44.15
Coins	4.58	10.40	13.58	11	11.75	10.25	12.00	12.75	11.25	88.00	93.00	182.50
Rao's Keys	2.15	6.09	3.06	13	9.50	11.00	14.00	10.50	12.00	35.50	33.50	54.00
Grid-Empty	0.88	1.55	0.97	5	4.75	4.75	6.00	5.75	5.75	17.50	21.75	20.75
Grid-Wall	5.31	60.70	35.26	18.1	18.10	13.90	19.10	19.10	14.90	41.00	56.30	57.70
New-Push	142.81	388.34	35.08	14.6	13.80	14.00	15.60	14.80	15.00	48.80	48.20	45.60
New-Ring	0.99	0.68	1.04	1	1.00	1.00	2.00	2.00	2.00	29.00	29.00	29.00

focuses on solvable instances only. Unsolvability is clearly harder to prove in the larger search space generated by the Lazy and Cautious, which are the only two policies that preserve completeness.

10.5.1. Greedy sampling and macros

Table 9 and 10 reports experimental results for the gcpces using the three different macro policies, that is *Lazy* (short used L), *Cautious* (short used C), and *Eager*. Thanks to the use of the *Cautious* policy cpces has been able to exploit information coming from previous iterations for some domain, but this did not always pay off. Extremely good are for instance

Table 11
Coverage analysis of hcpces with the three macro policies.

	hcpces	hcpces		
Domain	no-macro	lazy	cautious	eager
Dispose	4	4	4	4
BlocksWorld	3	3	3	0
One-Dispose	5	5	10	9
Look-and-Grab	18	18	18	18
Bomb	4	4	4	1
UTS	13	13	13	0
Coins	8	8	8	0
Rao's Keys	2	2	2	2
Grid-Empty	4	4	4	0
Grid-Wall	14	10	18	0
New-Push	6	6	7	1
New-Dispose	0	0	0	0
New-Ring	11	11	11	11
New-UTS	5	5	5	0

Table 12
hcpces with different macro policies. Instances considered are those solved by every. Eager macro policy excluded for lack of coverage.

hcpces	Planning Time			Samples			Iterations			Plan Length		
Domain	NM	L	C	NM	L	C	NM	L	C	NM	L	C
Dispose	725.23	659.21	664.98	32	34	32.50	37.50	39.75	36.25	165.50	102.25	214.00
BlocksWorld	61.76	24.99	6.58	8.33	9.33	7.00	11.33	14.33	10.67	29.67	30.33	76.33
One-Dispose	713.19	259.55	15.03	14.8	13.2	11.60	141.00	120.80	21.40	71.00	72.60	141.80
Look-and-Grab	232.11	128.84	48.51	6.61	6.67	6.33	16.67	13.50	10.56	44.67	49.33	54.44
Bomb	915.96	1479.73	1469.06	42.75	42.75	42.75	173.75	173.75	173.75	77.75	77.75	77.75
UTS	247.82	193.29	227.66	15.38	15.38	15.38	16.38	16.38	16.38	49.08	44.15	44.15
Coins	37.99	6.17	10.44	9.25	9	9.12	12.38	13.12	12.12	87.88	94.50	173.62
Rao's Keys	9.78	3.04	8.05	11.5	9.5	9.00	15.50	11.00	11.50	33.50	33.50	56.00
Grid-Empty	2.89	1.25	1.79	2.75	2.5	2.25	8.00	5.75	5.75	17.50	21.75	20.75
Grid-Wall	144.26	46.45	20.30	12.8	7.8	7.00	157.10	30.80	19.80	41.00	47.00	50.40
New-Push	646.04	94.14	60.54	10.67	9.33	10.50	28.00	22.67	21.50	53.83	54.00	59.67
New-Ring	0.67	5.21	0.74	1	1	1.00	2.00	2.00	2.00	29.00	29.00	29.00
New-UTS	25.69	25.30	47.51	12.8	12.8	12.80	13.80	13.80	13.80	37.40	37.40	39.60

the results of cpces with the *Cautious* policy on the One-Dispose. Here the speed-up is substantial and goes up to 1 order of magnitude on average: hence the *Cautious* policy let gcpces solve 4 instances that could not be solved before. Good speed-up can be observed on Look-and-Grab and New-Push, yet the performance degrades on more difficult domains such as Grid-Wall. As expected, the greedier nature of the planner when using macros makes the plans some time much longer. This is particularly evident in Coins, BlocksWorld and Dispose.

10.5.2. Heuristic sampling and macros

Tables 11 and 12 move the focus on the heuristic-based sampling. The situation here is quite different. As a matter of facts, the best performing macro policy (i.e., *Cautious*) boosts run-time performance in the majority of the instances tried. As for gcpces the best result is for One-Dispose, but as it is possible to observe in Table 12, there are substantial speed-up also elsewhere, for instance in Bomb. Similarly to gcpces, also in this case the adoption of macros results in a severe performance degradation in Grid-Wall.

10.5.3. Refined sampling with macros

Tables 13 and 14 report information about the behaviour of the refined-based sampling. The trend here is somewhat more pronounced. Apart from Grid-Wall, the speed-up is homogeneous in each tested domain. Interestingly, we can observe a pronounced reduction of the average number of iterations needed to get to a solution. This result was at first look unexpected, but a careful analysis reveals that, especially in non-greedy samplings, the macro embodying the plan previously computed not only incorporates pieces of knowledge useful to speed-up the search, but also some memory of the states handled in some previous iteration. In other words, by keeping part of the previous plan intact, also initial states which are not in the current sample but that were previously encountered will be handled. This opens an interesting angle of research that we did not have the time to explore.

Overall, the addition of macro operator seems promising but, as we can see from the results, we have just scratched the surface of this technique. In fact, we conjecture a more careful selection of macros aimed at exploring the dependencies between actions (much as it is done using deordered plans by Scala and Torasso [41]), and/or a more integrated way to inject them into the planning system should lead to much more improvement. For instance, more recent planning formalisms support action-cost. In our case the plan-actions could be given 0-cost so as to encourage the planner to use them instead

Table 13
Coverage analysis of rcpces with the three macro policies.

	rcpces	rcpces		
Domain	no-macro	lazy	cautious	eager
Dispose	0	0	1	4
BlocksWorld	3	2	3	0
One-Dispose	2	2	4	9
Look-and-Grab	17	17	18	18
Bomb	0	0	0	1
UTS	6	6	12	0
Coins	1	1	8	0
Rao's Keys	1	2	2	2
Grid-Empty	4	4	4	0
Grid-Wall	16	10	18	0
New-Push	4	4	4	1
New-Dispose	0	0	0	2
New-Ring	11	11	11	11
New-UTS	2	2	2	0

Table 14
rcpces with different macro policies. Instances considered are those solved by every. Eager macro policy excluded for lack of coverage.

rcpces	Planning Time			Samples			Iterations			Plan Length		
Domain	NM	L	C	NM	L	C	NM	L	C	NM	L	C
BlocksWorld	1.84	1.16	1.09	2.50	2	1.50	18.00	8.50	7.00	11.00	10.00	13.00
One-Dispose	41.94	59.88	5.02	5.00	5	2.50	470.50	537.00	44.50	30.00	32.00	39.50
Look-and-Grab	246.96	435.00	156.97	3.59	3.29	2.71	189.12	227.88	74.12	35.18	36.35	42.59
UTS	707.20	354.93	20.22	7.00	7	2.83	910.00	910.00	52.50	20.67	19.00	23.17
Coins	34.87	120.88	9.34	4.00	4	2.00	384.00	813.00	86.00	28.00	34.00	54.00
Rao's Keys	1.67	1.67	12.24	4.00	4	2.00	16.00	16.00	9.00	16.00	16.00	17.00
Grid-Empty	1.28	1.34	3.38	2.00	2	1.75	12.50	11.00	10.00	17.50	19.50	24.25
Grid-Wall	87.25	43.14	29.77	12.90	8.3	7.00	292.90	77.00	35.80	41.00	48.80	57.00
New-Push	16.60	38.37	28.64	6.00	6.75	4.75	80.25	94.50	80.00	38.00	37.00	37.25
New-Ring	0.73	0.81	1.01	1.00	1	1.00	2.00	2.00	2.00	29.00	29.00	29.00
New-UTS	87.30	188.83	52.76	7.00	6.5	3.50	520.00	471.50	182.00	20.00	20.00	20.50

of original actions (which should be left at their original cost). This should make the planner exploit this information directly, instead of enlarging the search space with useless actions.

10.6. Different classical planners

As we have seen in the introductory part of this experimental analysis section, one of the bottlenecks of our approach is the classical planner. In this section then, we conclude our in-depth analysis by studying the impact of the classical planner on the performance of gcpces (which is the most stable sampling strategy). As hinted at in the introduction we support FF and MpC, so the analysis is performed using these two systems; Table 15 shows an overview of the collected data. MpC turned out generally slower than FF, and this is due partially to a slightly larger start-up time. We call the planner as a black-box so every time a new classical planning instance needs to be built and parsed by the system. FF provides magnificent performance in this regard. However, although FF is generally faster, there are slight differences in the performance for some specific domains. For instance, in New-Dispose, MpC managed to solve one instance where FF solves none. We observed that the length of the resulting plan can also be quite different. In particular, in Look-and-Grab, MpC produces plans which are almost half the size of those found by FF. For a substantial set of domains MpC produces plans which are better than those found by FF. We did of course not test the performance over unsolvable instances: MpC uses an incremental bounded SAT problem which is practically incomplete (unless one admits the unrolling of formulae for plans that can have exponential length), so did not give any answer for our unsolvable problems. On the other hand, MpC handles quite well more complex goals, such as the disjunction used in the One-Dispose domain. It is likely that, for planning problems exhibiting more complex construct, MpC could be preferred over FF. Finally, in all instances, the configuration with FF solves all instances that are solved by MpC, but New-Dispose.

Another important aspect to highlight here refers to the extreme ductility and modularity of cpces, which makes it a very good asset if new classical planners are put in place. cpces makes in fact no assumption on the classical planner being used, besides of course requiring the planner to support the pddl language as an input.

Table 15
Comparison between gCPCES using FF and MPC as classical planners. Planning-Time, Samples, Iterations and Plan-length are averages computed over instances solved by both configurations.

	Coverage		Planning Time		Samples		Iterations		Plan Length	
Domain	FF	MPC	FF	MPC	FF	MPC	FF	MPC	FF	MPC
DISPOSE	4	3	10.49	24.13	28	27.67	29.00	28.67	83.33	52.33
BLOCKSWORLD	4	2	1.15	0.89	3.5	4.50	4.50	5.50	12.50	11.50
ONE-DISPOSE	5	3	2.12	11.70	13.67	28.67	14.67	29.67	37.67	36.00
LOOK-AND-GRAB	18	18	29.21	665.81	9.61	7.39	10.61	8.39	49.67	25.17
BOMB	7	4	8.63	4.91	27.5	21.25	28.50	22.25	37.75	58.25
UTS	13	11	3.58	834.91	11.82	8.64	12.82	9.64	40.00	531.55
COINS	8	8	4.58	4.76	11	10.50	12.00	11.50	88.00	87.00
RAO'S KEYS	2	2	2.15	2.12	13	13.50	14.00	14.50	35.50	33.50
GRID-EMPTY	4	4	0.88	1.00	5	5.25	6.00	6.25	17.50	20.00
GRID-WALL	18	6	1.88	329.29	10	9.50	11.00	10.50	25.33	26.83
NEW-PUSH	6	6	127.66	855.80	15.5	17.00	16.50	18.00	56.67	48.17
NEW-DISPOSE	0	1	NA	NA	NA	NA	NA	NA	NA	NA
NEW-RING	11	11	0.99	1.20	1	1.00	2.00	2.00	29.00	37.91
NEW-UTS	5	2	2.30	69.50	7	7.00	8.00	8.00	20.00	399.50

10.7. Comparative analysis

10.7.1. Competitors details

In this section we study gCPCES, which is the fastest and more reliable version of CPCES, against T1 [2], CG-LAMA [33], and DNF [44]. We have chosen these planners as they cover quite well the relevant literature in conformant planning. T1 is a conformant planner that well synthesized the work in conformant planning via heuristic search (in particular those exploiting heuristics obtained from proper relaxations of the problem) and translation from conformant to classical planning. T1 is built on top of CONFORMANT-FF [23]. It shares its main idea, that is doing heuristic search over a belief state space. At each step of the search T1 needs to check action preconditions for entailment; this is to guarantee that the action is applicable regardless of the belief state. States are explored using best first search[7] over a sophisticated mechanism with multiple priority queue serving the purpose of allowing to switch between regular and helpful actions. The heuristic (and therefore the helpful actions that are extracted from it) is the relaxed plan size, obtained from solving the relaxation of the problem. Such a relaxation assumes the width of the conformant planning problem being equal to one. Therefore, the planner is expected to perform well especially on those problems with low width, as the heuristic should provide great guidance. The other competitor, CG-LAMA, is a planner based on the idea of constructing a plan incrementally, check for its consistency and then try to repair it with additional actions. This planner is a good representative of problems adopting a sampling mechanism. In this regard, CG-LAMA is similar in the spirit to CPCES; but since it does not ensure a systematic exploration of the initial belief, is incomplete and can be trapped into dead-ends. However, it showed magnificent performance over the IPC benchmarks. Our last competitor is DNF, a planner that does symbolic search using a sophisticated compact encoding of the belief, incrementally updated with the incumbent plan of actions explored using a best-first search approach. DNF is a good representative of a class of system that puts emphasis on the construction of powerful and compact representation of the belief [44]. DNF guides its search with a goal-counting heuristic, in a greedy fashion; this can result quite ineffective for more complex problems, such as those involving dead-ends.

We took the last version of all planners from the websites of their respective authors, and worked hard to ensure all planners manage to correctly interpret the suite of problems on which we performed the experimental analysis. Note, in fact, that each planner makes its own assumption on how the belief has to be represented in the conformant planning extension of PDDL, and sometime uses directly the compilation to other intermediate languages (i.e., DNF uses Prolog).

10.7.2. Results

Table 16 reports an overview of our experimental results obtained by analyzing the performance of gCPCES and CG-LAMA. Over the majority of the tried IPC solvable instances (6 domains out of 9), CG-LAMA runs much faster; yet performs quite poorly in terms of plan length. In some domains CG-LAMA yields plans in some case till fourth times longer than those computed by gCPCES. We have noticed in fact that, although the classical planner used by gCPCES is a suboptimal one (i.e., FF runs enforced hill climbing as a first search which is neither optimal nor complete), CPCES's plans are much shorter than those found by CG-LAMA. Even in the greedy setting, CPCES's outer loop keeps the number of initial contingencies to be dealt quite low; therefore, the final plan remains often short in that only necessary actions tend to be considered. As we have seen in the previous section, this is not true when macro-actions are considered.

Interestingly, there are domains in which not only gCPCES yields shorter plans, but also achieve higher coverage than CG-LAMA. We noticed in fact that, as soon as we move our comparison over domains involving more complicated reasoning over the uncertainty aspects, for instance for the presence of dead-ends (our grid-based domains) CG-LAMA's performance

[7] Note that T1 can be run in several modalities; we used the default setting.

Table 16
Results for CG-LAMA and GCPCES.

		Coverage		Planning Time		Plan Length	
		GCPCES	CG-LAMA	GCPCES	CG-LAMA	GCPCES	CG-LAMA
SAT	BlocksWorld	4	4	3.2	0.2	29.0	67.0
	One-Dispose	5	10	117.4	0.1	81.4	121.4
	Look-and-Grab	18	18	28.8	52.8	49.7	228.8
	Bomb	7	9	124.8	0.1	106.0	101.1
	UTS	13	15	16.5	0.2	49.7	45.2
	Coins	8	8	4.3	1843.1	88.0	168.5
	Dispose	4	11	46.6	0.1	167.2	370.0
	Rao's Keys	2	2	1.9	0.1	35.5	75.5
	Grid-Empty	4	0	NA	NA	NA	NA
	Grid-Wall	18	0	NA	NA	NA	NA
	New-Push	6	0	NA	NA	NA	NA
	New-Dispose	0	2	NA	NA	NA	NA
	New-Ring	11	7	0.8	313.6	18.7	92.6
	New-UTS	5	0	NA	NA	NA	NA
UNSAT	Grid-Empty	14	0	NA	NA	NA	NA
	Rao's Keys	1	0	NA	NA	NA	NA
	BlocksWorld	1	0	NA	NA	NA	NA

Table 17
Results for T1 and GCPCES.

		Coverage		Planning Time		Plan Length	
		GCPCES	T1	GCPCES	T1	GCPCES	T1
SAT	BlocksWorld ($w > 1$)	4	2	0.9	0.1	12.5	13.0
	One-Dispose ($w > 1$)	5	4	33.2	165.0	61.8	79.2
	Look-and-Grab ($w > 1$)	18	12	21.9	1.9	38.5	30.3
	Bomb	7	9	124.8	0.1	106.0	101.1
	UTS	13	11	2.9	0.2	36.4	40.6
	Coins	8	9	4.3	0.5	88.0	148.6
	Dispose	4	9	46.6	4.1	167.2	177.2
	Rao's Keys ($w > 1$)	2	1	0.6	0.4	16.0	21.0
	Grid-Empty ($w > 1$)	4	4	0.7	0.0	17.5	17.5
	Grid-Wall ($w > 1$)	18	4	0.7	0.0	17.5	17.5
	New-Push ($w > 1$)	6	8	13.8	7.8	47.6	55.4
	New-Dispose	0	2	NA	NA	NA	NA
	New-Ring	11	0	NA	NA	NA	NA
	New-UTS	5	7	17.9	0.1	35.0	35.0
UNSAT	Grid-Empty ($w > 1$)	14	9	2.2	0.0	NA	NA
	Rao's Keys ($w > 1$)	1	0	NA	NA	NA	NA
	BlocksWorld ($w > 1$)	1	0	NA	NA	NA	NA

degrades quite heavily. For instance, in both grid-based domains CG-LAMA timed out or consumed the entire memory (we gave 16GB of RAM) on all the instances tried, even the smaller ones. CG-LAMA is also not able to prove unsolvability. None of the instances of the grid and the two unsolvable instances from the competition were solved by CG-LAMA. From the experiments, it therefore clearly arises that for problems with no-dead ends, where the length of the solution is not an issue, CG-LAMA is probably the best system to employ. On the other hand, for more involved problems, and in particular for unsolvable problems, CPCES is probably the best option.

Table 17 reports data concerning our comparative analysis between GCPCES and T1. As T1 is a planner aimed at exploiting structural width-based decomposition of conformant planning problems, we highlight for convenience all those domains having width larger than one in the table. As expected, T1 runs faster on problems with low width, but exhibits some weaknesses (as CG-LAMA) as far as it is concerned by the quality of the produced plans; this happens both over instances with low and high width. Over unsolvable instances, T1 is less performant than GCPCES in terms of coverage, but, differently from GCPCES, it quickly managed to prove the unsolvability of some of the instances of Grid-Empty (9 out of 14 instances). In problems with larger width, the superiority of GCPCES becomes prominent both in terms of instances solved, and in average run-time. Interestingly, in Look-and-Grab, T1 produces higher quality plans.

As for CG-LAMA's comparison analysis, GCPCES is complementary to T1. In particular if one is interested in problems which do not exhibit clear width-based decomposition or are even unsolvable, she should prefer CPCES over T1. On the other hand, if the problem is known to have low width, then T1 is probably the best choice.

Table 18 shows the results comparing CPCES with DNF. DNF proves very competitive across several domains, but shows its limits particularly over unsolvable instances and most of the instances where there is an important width. The only exception is Dispose, in all its incarnations. It seems that DNF is particularly effective in this domain. Overall, CPCES seems

Table 18
Results for CPCES and DNF.

		Coverage		Planning Time		Plan Length	
		gCPCES	DNF	gCPCES	DNF	gCPCES	DNF
SAT	BlocksWorld	4	3	3.2	0.8	29.0	56.3
	One-Dispose	5	10	117.4	1.0	81.4	67.4
	Look-and-Grab	18	8	0.8	357.9	9.5	7.8
	Bomb	7	9	124.8	1.6	106.0	101.1
	UTS	13	15	16.5	1.6	49.7	50.0
	Coins	8	8	4.3	0.8	88.0	83.1
	Dispose	4	11	46.6	1.5	167.2	114.5
	Rao's Keys	2	2	1.9	0.7	35.5	62.0
	Grid-Empty	4	4	0.7	62.4	17.5	18.5
	Grid-Wall	18	4	0.7	62.6	17.5	18.5
	New-Push	6	10	127.2	178.6	56.7	49.0
	New-Dispose	0	6	NA	NA	NA	NA
	New-Ring	11	11	0.7	47.3	29.0	28.1
	New-UTS	5	12	17.4	1.1	37.4	37.4
UNSAT	Grid-Empty	14	0	NA	NA	NA	NA
	Rao's Keys	1	0	NA	NA	NA	NA
	BlocksWorld	1	0	NA	NA	NA	NA

Table 19
Analysis on reformulation of Coins and New-Push, with *oneof* predicates substitute with plain disjunctions.

	Time		Plan Length	
Instance	gCPCES	DNF	gCPCES	DNF
OR-Coins				
or-10	1.02	1.40	31	44
or-12	2.36	3105.95	68	178
or-15	3.68	TO	81	TO
or-16	3.23	TO	108	TO
or-17	2.40	TO	97	TO
or-18	2.26	TO	98	TO
or-19	2.45	TO	109	TO
or-20	2.37	TO	112	TO
or-21	TO	TO	TO	TO
OR-NewPush				
or-4_2	2.46	1325.91	34	40
or-4_3	1.85	TO	36	TO
or-4_4	4.50	TO	36	TO
or-4_5	4.58	TO	36	TO
or-6_2	719.90	TO	102	TO
or-6_3	35.64	TO	96	TO
or-8_1	TO	TO	TO	TO
or-8_2	TO	231.63	TO	824
or-9_1	TO	TO	TO	TO
or-9_2	TO	TO	TO	TO

generally more robust than DNF also against the harder domains proposed by DNF's authors. That is, CPCES is not severely affected by the complexity of these problems, showing even better results than DNF in New-Ring. Nevertheless, DNF's performance can be substantially better than CPCES, not only in Dispose, but also in other domains such as UTS. To further understand the difference of performance between the two systems, we also ran, for these two systems, experiments on reformulated version of Coins and New-Push, where we replace the *oneof* predicate with a plain disjunction. This modification follows the experiment done in the DNF's paper [44] to challenge belief tracking in DNF. We want to understand whether this aspect is challenging for gCPCES as well. The reformulated domains are called OR-Coins and OR-NewPush. Table 19 reports an instance by instance analysis. gCPCES dominates DNF, but for a single instance of OR-NewPush. This is not unexpected though. The size of the belief does not affect gCPCES performance because such a belief is only explored incrementally. The performance of gCPCES does not vary in any meaningful way because gCPCES captures the fact that many of the newly introduced initial states are actually irrelevant for the kinds of plan that it generates. Similarly to the other comparisons, there is a complementary behaviour of performance between DNF and CPCES; DNF is very fast but there are problems, such as unsolvable problems and problems with large beliefs, where gCPCES seems to be a substantial better option than DNF.

11. Conclusion

In this paper we presented CPCES, a novel conformant planner, that uses classical planners to propose candidate conformant plans based on a sample of the initial belief state, and updates this sample by adding counter-examples of invalid candidate plans. We show that CPCES is sound and complete, and in particular proves effective in determining when there is no solution to the planning problem. CPCES can also be made optimal if the classical planner is optimal. We study the theoretical properties of different variants of CPCES with respect to the known notions of tag and width. Our comprehensive empirical evaluation shows that CPCES performs very well, particularly for the hardest problems (i.e., those with a non-trivial width) compared to the existing solvers. Among other options in the literature, CPCES is the only system that provides theoretical guarantees (soundness, completeness and optimality) proving to be practically effective too in a unified framework.

CPCES is a generic and modular approach to planning under uncertainty. There are many possible extensions of this work, which can easily be explored by improving on the various modules CPCES consists of. We already presented one such avenue: because the conformant planning problems are very similar from one sample to the next we showed in Section 9 that it is possible to learn from the previous candidate plan in order to accelerate candidate plan production.

In the future we want to improve the different procedures involved in CPCES. For instance we want the plan validator to return better counter-examples, where this notion of "better" needs to be properly defined. We also believe that the learning aspect can be improved. Interestingly in gCPCES the classical planning problems at any iteration is just a refinement of the previous problem, i.e., it is defined over the same set of actions and the main differences are the facts that there are more state variables and that the preconditions and conditional effects are updated in a monotonic fashion. It should be possible to define an incremental classical planner that allows state variables to be added after a first plan was computed. More generally it would be interesting to have a classical planner able to carry over information from one iteration to the next; MPC for instance was not competitive precisely because it performed expensive preprocessing on every classical problem instance.

We also expect the general framework of CPCES to be adaptable to more general problems beyond just deterministic conformant planning. We want to address problems with uncertainty on the action effects, partial observability, continuous variables, probabilities, etc. It is not clear at this stage which problems a CPCES-inspired approach will be sound and complete for, or competitive.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] A. Albore, H. Palacios, H. Geffner, Compiling uncertainty away in non-deterministic conformant planning, in: Nineteenth European Conference on Artificial Intelligence (ECAI-10), 2010, pp. 465–470.
[2] A. Albore, M. Ramírez, H. Geffner, Effective heuristics and belief tracking for planning with incomplete information, in: 21st International Conference on Automated Planning and Scheduling (ICAPS-11), 2011.
[3] C. Barrett, A. Stump, C. Tinelli, The smt-lib standard version 2.0, Technical report, 2010.
[4] Avrim Blum, Merrick L. Furst, Fast planning through planning graph analysis, Artif. Intell. 90 (1–2) (1997) 281–300.
[5] B. Bonet, Conformant plans and beyond: principles and complexity, Artif. Intell. 174 (2010) 245–269.
[6] B. Bonet, H. Geffner, Planning with incomplete information as heuristic search in belief space, in: Fifth International Conference on AI Planning Systems (AIPS-00), 2000, pp. 52–61.
[7] B. Bonet, H. Geffner, Planning as heuristic search, Artif. Intell. 129 (1–2) (2001) 5–33.
[8] B. Bonet, H. Palacios, H. Geffner, Automatic derivation of memoryless policies and finite-state controllers using classical planners, in: ICAPS, 2009.
[9] R. Brafman, G. Shani, Replanning in domains with partial information and sensing actions, J. Artif. Intell. Res. 45 (2012) 565–600.
[10] R. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Trans. Comput. 35 (8) (1986) 677–691.
[11] Daniel Bryce, Subbarao Kambhampati, David E. Smith, Planning graph heuristics for belief space search, J. Artif. Intell. Res. 26 (2006) 35–99.
[12] T. Bylander, The computational complexity of propositional STRIPS planning, Artif. Intell. 69 (1–2) (1994) 165–204, https://doi.org/10.1016/0004-3702(94)90081-7.
[13] A. Cimatti, M. Roveri, Conformant planning via symbolic model checking, J. Artif. Intell. Res. 13 (2000) 305–338.
[14] A. Cimatti, M. Roveri, P. Bertoli, Conformant planning via symbolic model checking and heuristic search, Artif. Intell. 159 (2004) 127–206.
[15] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement, in: Twelfth International Conference on Computer-Aided Verification (CAV-00), 2000, pp. 154–169.
[16] A. Darwiche, P. Marquis, A knowledge compilation map, J. Artif. Intell. Res. 17 (2002) 229–264.
[17] L. De Moura, N. Bjørner, Z3: an efficient smt solver, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2008, pp. 337–340.
[18] U. Egly, M. Kronegger, F. Lonsing, A. Pfandler, Conformant planning as a case study of incremental QBF solving, Ann. Math. Artif. Intell. 80 (2017) 21–45.
[19] A. Grastien, E. Scala, Intelligent belief state sampling for conformant planning, in: 26th International Joint Conference on Artificial Intelligence (IJCAI-17), 2017, pp. 4317–4323.
[20] A. Grastien, E. Scala, Sampling strategies for conformant planning, in: 28th International Conference on Automated Planning and Scheduling (ICAPS-18), 2018, pp. 97–105.
[21] A. Grastien, E. Scala, Verifying the validity of a conformant plan is co-NP-complete, viXra:1705.0340, 2018.

[22] P. Haslum, P. Jonsson, Some results on the complexity of planning with incomplete information, in: Fifth European Conference on Planning (ECP-99), 1999, pp. 308–318.
[23] J. Hoffmann, R. Brafman, Conformant planning via heuristic forward search: a new approach, Artif. Intell. 170 (2006) 507–541.
[24] J. Hoffmann, B. Nebel, The ff planning system: fast plan generation through heuristic search, J. Artif. Intell. Res. 14 (2001) 253–302.
[25] J. Huang, A. Darwiche, The language of search, J. Artif. Intell. Res. 29 (2007) 191–219.
[26] H. Kautz, B. Selman, Pushing the envelope: planning, propositional logic, and stochastic search, in: Thirteenth Conference on Artificial Intelligence (AAAI-96), 1996, pp. 1194–1201.
[27] R.E. Korf, Macro-operators: a weak method for learning, Artif. Intell. 26 (1) (1985) 35–77.
[28] M. Kronegger, A. Pfandler, R. Pichler, Conformant planning as a benchmark for QBF solvers, in: International Workshop on Quantified Boolean Formulas (QBF-13), 2013, pp. 1–5.
[29] J. Kurien, P. Nayak, D. Smith, Fragment-based conformant planning, in: Sixth International Conference on AI Planning Systems (AIPS-02), 2002, pp. 153–162.
[30] I. Little, S. Thiébaux, Probabilistic planning vs. replanning, in: ICAPS Workshop on IPC: Past, Present and Future, 2007.
[31] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, D. Wilkins, PDDL - the Planning Domain Definition Language, Technical report, 1998, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.212.
[32] S. Minton, Quantitative results concerning the utility of explanation-based learning, Artif. Intell. 42 (2) (1990) 363–391.
[33] K. Nguyen, V. Tran, T.C. Son, E. Pontelli, On computing conformant plans using classical planners: a generate-and-complete approach, in: 22nd International Conference on Automated Planning and Scheduling (ICAPS-12), 2012, pp. 190–198.
[34] H. Palacios, H. Geffner, Compiling uncertainty away in conformant planning problems with bounded width, J. Artif. Intell. Res. 35 (2009) 623–675.
[35] J. Rintanen, Constructing conditional plans by a theorem-prover, J. Artif. Intell. Res. 10 (1999) 323–352.
[36] J. Rintanen, Expressive equivalence of formalisms for planning with sensing, in: Thirteenth International Conference on Automated Planning and Scheduling (ICAPS-03), 2003, pp. 185–194.
[37] J. Rintanen, Asymptotically optimal encodings of conformant planning in QBF, in: 22nd Conference on Artificial Intelligence (AAAI-07), 2007, pp. 1045–1050.
[38] J. Rintanen, Regression for classical and nondeterministic planning, in: Eighteenth European Conference on Artificial Intelligence (ECAI-08), 2008, pp. 568–572.
[39] J. Rintanen Madagascar, Scalable planning with sat, in: Proceedings of the 8th International Planning Competition (IPC-2014), 2014, p. 21.
[40] J. Rintanen, K. Heljanko, I. Niemelä, Planning as satisfiability: parallel plans and algorithms for plan search, Artif. Intell. 170 (12–13) (2006) 1031–1080.
[41] E. Scala, P. Torasso, Deordering and numeric macro actions for plan repair, in: 24th International Joint Conference on Artificial Intelligence (IJCAI-15), 2015, pp. 1673–1681.
[42] D. Smith, D. Weld, Conformant graphplan, in: Fifteenth Conference on Artificial Intelligence (AAAI-98), 1998, pp. 889–896.
[43] R. Taig, R. Brafman, Compiling conformant probabilistic planning problems into classical planning, in: ICAPS, 2013, pp. 197–205.
[44] S.T. To, T. Cao Son, E. Pontelli, A generic approach to planning in the presence of incomplete information: theory and implementation, Artif. Intell. 227 (2015) 1–51.
[45] V. Tran, K. Nguyen, T.C. Son, E. Pontelli, A conformant planner based on approximation: CpA(H), ACM Trans. Intell. Syst. Technol. 4 (2) (2013) 36:1–36:38.
[46] P.H. Tu, T.C. Son, M. Gelfond, A.R. Morales, Approximation of action theories and its application to conformant planning, Artif. Intell. 175 (2011) 79–119.