Using the Fast Downward System in CPCES

Xiaodi Zhang[1], **Alban Grastien**[1]
[1]Research School of Computing
Australian National University, ACT, Australia, 2601
firstname.lastname@anu.edu.au

Abstract

CPCES is a conformant planning solver that continuously searches for candidate plans through a classical planner and for counter-examples to these plans. CPCES can use any classical planner, but performance with Fast Downward has been so far mediocre. We argue that this is due in part to a sub-optimal translation from PDDL to SAS+. We recognise that the problems that the classical planner is asked to solve have very specific structure, and that they can be split in independent problems to be translated separately. We conduct an empirical study that shows that, when using this translating strategy, the performance of CPCES powered by Fast Downward can be brought to the level of CPCES + Fast Forward.

1. Introduction

Conformant planning refers to a task that generates a plan given unknown initial states and unknown actions' effects, and there is no observation during plan execution. The plan should be valid for all possible initial states. Conformant planning problem is ExpSpace-complete (Haslum and Jonsson 1999). In this paper, we concentrate on *deterministic* conformant planning in which uncertainty is only in the initial state.

CPCES (Grastien and Scala 2017; 2020) is a conformant planner that uses two components. The first component searches for a counter-example to a candidate plan by falsifying one of the (sub)goals of the plan; The second component computes a candidate plan that is valid for the set of counter-examples generated so far; this subproblem is solved by transforming it into a classical planning problem with a multi-interpretation PDDL file, and giving it to a classical planner. CPCES can therefore be used with FF (Fast Forward) (Hoffmann and Nebel 2001), MADAGASCAR (Rintanen 2012), and FD (Fast Downward) (Helmert 2006), but its performance with the later is subpar.

FD is a classical planning system that contains two parts. A pre-processing step translates the input PDDL file to a SAS+ file (multi-valued planning task file). The second step is the planning proper, and it consists of a heuristic search procedure. FD is a general framework that contains many different search procedures and heuristics, and gives us access to a wealth of planning techniques.

The goal of this work is to study how to use FD in CPCES properly. Early experiments have shown us that the translation from PDDL to SAS+ is time consuming and leads to poor SAS+ representations. We recognise however that the classical planning problems have a structure that can be exploited. Indeed, each subproblem at each iteration of CPCES aims to find a plan that is valid for an increasing set of initial states. Practically, the planning problems build on top of a multi-interpretation representation of the environment, each interpretation corresponding to one initial state. Consequently, each problem contains the list of interpretations of the previous iteration plus an extra one. Hence, we show that it is possible to isolate this new interpretation, translate it into SAS+, and add this translation to the existing translation. This is not only faster, as the translation becomes a simple task, but also leads to SAS+ translations that contain fewer variables.

This paper is structured as follows. We quickly review existing conformant planners in the next section. We then give the technical background necessary to understand in this work, defining the conformant planning problem, CPCES, and FD. We then show how to exploit the structure of the classical planning problems in order to accelerate its translation to SAS+. Finally, we show experimentally that this approach is practically effective.

2. Related Works

The problem of conformant planning was defined by Smith and Weld (Smith and Weld 1998) and was recognised as a path-finding problem over the space of belief states (Bonet and Geffner 2000). It was shown to be EXPSPACE-complete (Haslum and Jonsson 1999).

Cimatti, Roveri, and Bertoli (2004) proposed to use Binary Decision Diagrams to compactly represent the belief states. To, Son, and Pontelli (2015) used DNF representations in their solver.

Conformant-FF (Hoffmann and Brafman 2006) extends the classical planner FF, using implicit belief state representation and a SAT solver to solve the problem. The belief state is then represented as a propositional formula on the state variables; to maintain a compact representation, this formula is defined over the past state variables.

The SAT solver is used to determine if the relevant conditions are satisfied in all states of the current belief. The SAT solver is also used to determine the known facts of each belief. `Conformant-FF` adopts planning graph to generate its heuristic function. It uses a 2-CNF projection of the formula that captures the true belief state semantics.

`POND` (Bryce 2006) can solve many problems, such as deterministic conformant planning, non-deterministic planning, probabilistic planning. `POND` uses LUG to represent a set of explicit planning graph, and searches forward in the space of belief states represented by BDDs. `POND` provides various search algorithms, such as A*, AO*, LAO*. The search algorithm is decided by problem and user preferences.

`T1` (Albore, Ramirez, and Geffner 2011) is a translation-based planner. It is sound for problems with width 1,[1] and complete for all problems. `T1` is built on top of the K_S^1 translation that turns a conformant planning problem to a classical one, performing a verification procedure to establish the literals that are positively known or negatively known. `T1` searches a valid plan by forward search in the belief space.

Our work is based on `CPCES` from Grastien and Scala (2020), which we detail in the next section.

3. Background

Conformant Planning

Given a set of facts F, a state s is modeled as the subset of facts $s \subseteq F$ that hold in the state. We write $\mathcal{L}(F)$ the set of propositional formulas defined over the set of propositional variables represented by F. A *deterministic conformant planning* problem $P = \langle F, N, A, I, G\rangle$ is defined as follows:

- F is the finite set of *facts*;
- N is a set of *action names*, also called *actions*;
- $A : N \to \mathcal{L}(F) \times 2^{\mathcal{L}(F)\times 2^F \times 2^F}$ is a function that returns a description of each action such that $A(a) = \langle pre(a), coneff(a)\rangle$ and $coneff(a) = \{\langle c_1, eff_1^+, eff_1^-\rangle, \ldots, \langle c_k, eff_k^+, eff_k^-\rangle\}$;
- $I \in \mathcal{L}(F)$ is the *initial condition*; and
- $G \in \mathcal{L}(F)$ is the *goal condition*.

A state $s \subseteq F$ is *initial* if it satisfies the initial condition: $s \models I$; similarly, a state is a *goal state* if it satisfies the goal condition. Applying action a in state s yields the *effects* $\langle eff^+(s,a), eff^-(s,a)\rangle$ defined by

$$eff^+(s,a) = \bigcup_{\substack{\langle c, eff^+, eff^-\rangle \in coneff(a) \\ s \models c}} eff^+$$

and

$$eff^-(s,a) = \bigcup_{\substack{\langle c, eff^+, eff^-\rangle \in coneff(a) \\ s \models c}} eff^-.$$

[1]The width of a conformant planning problem is a measure of complexity based on how many initially uncertain state variables need to be considered jointly.

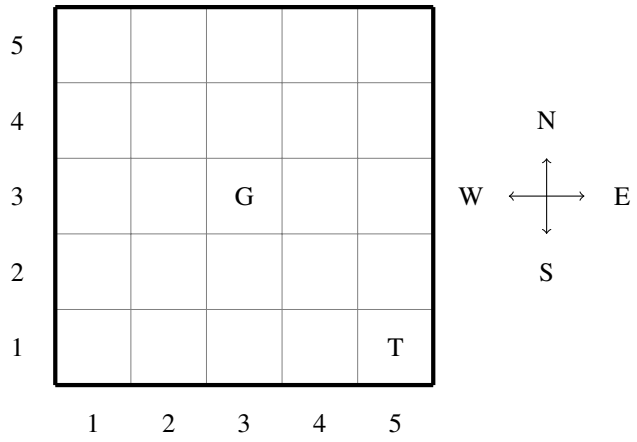

Figure 1: Graphical representation of a conformant planning problem. A robot is standing in this 5×5 grid surrounded by walls. its initial location is unknown. The goal is to move the robot to "G". One valid plan is GO_E $\times$ 4, GO_S $\times$ 4, GO_N $\times$ 2, GO_W $\times$ 2.

The state reached by applying action a in state s is $s[a] = s \setminus eff^-(s,a) \cup eff^+(s,a)$. Action a is *applicable* in state s if the following two conditions holds:

1. state s satisfies the action's precondition: $s \models pre(a)$;
2. the effects of the actions do not conflict: $eff^+(s,a) \cap eff^-(s,a) = \emptyset$.

A *plan* is a sequence of action: $\pi = a1, \ldots, a_k$. *Applying* this plan in state s_0 leads to the sequence of states $s_1, \ldots, s_k$ where $s_i = s_{i-1}[a_i]$ for each i. The plan is *applicable* in state s_0 if each action a_i is applicable in s_{i-1}. The plan is *valid* in state s_0 if it is applicable and s_k is a goal state. It is *valid* in a set of states if it is valid in each state of the set.

A *solution* to the conformant planning problem is a plan that is valid in all initial states. We write $\Pi(P) \subseteq N^*$ the set of solutions of P.

A conformant planning problem is a *classical* planning problem if there is only one initial state.

Figure 1 presents an example of conformant planning problem. A robot is standing in a 5×5 grid, but its exact location is unknown. The grid is surrounded by walls to prevent the robot from moving outside. If the robot hits the wall, it will not move but just stand still. The robot can execute four actions: GO_N, GO_E, GO_W, GO_S, which represent moves to North, East, West, and South, respectively. The robot is required to move to (3,3), denoted as "G". The sequence of actions GO_S $\times$ 4, GO_E $\times$ 4 leads to robots to location $(5, 1)$ regardless of its initial location ("T" in the figure). From there, the sequence GO_N $\times$ 2, GO_W $\times$ 2 leads to the destination.

SAS+

The problem definition given above matches the PDDL representation in which the basic element is a predicate (fact).

Many planners, in particular those using the FD system, reason instead with a multi-valued encoding such as SAS+.

In a multi-valued encoding, facts are replaced by state variables. Each state variable v has a domain D_v, and assignments $(v = \nu)$ where $\nu \in D_v$ is a value from v's domain are equivalent to facts. An atomic effect $(v = \nu)$ then corresponds to a positive effect $(v = \nu)$ and a list of negative effects $(v = \nu')$ for all other value $\nu' \neq \nu$.

Translating a PDDL representation to a SAS+ representation generally requires identifying subsets of pairwise mutually exclusive facts F'. For each subset, a variable v is created with domain $F' \cup \{\bot\}$: $(v = f)$ holds in all reachable SAS+ states where the equivalent PDDL state would include fact f (and not f' for all other facts from $F' \setminus \{f\}$). The symbol $\bot$ is used to represent a situation in which all facts $f \in F'$ are false in the state.

CPCES

CPCES was proposed by Scala and Grastien to solve conformant planning problems (Grastien and Scala 2017). CPCES searches the valid plan by continuously searching for candidate plans and counter-examples to these candidates (Algorithm 1). CPCES stores the counter-examples into a set B called the *sample*. A counter-example is an initial state for which the candidate plan is not valid. In each iteration, CPCES first produces a candidate plan that is valid for all states in the sample (Line 5). In this step, the corresponding conformant planning problem is reduced to classical planning as described later. This translation is only practical because the sample is small; it would not be possible to apply a similar technique for the set of initial states. Then CPCES performs a regression operation to generate an initial state in which the plan is invalid, either because one of the actions is not applicable in the sequence of states induced by the plan or because that plan does not lead to the goal (Line 9). By adding counter-examples to the sample B, CPCES guarantees that the same (invalid) candidate plan will not be produced again in the future.

Currently, CPCES uses FF to produce a candidate plan (Line 5). The goal of this work is to make it practical to use FD instead, as many modern planners are built on top of FD.

Example 1 *We illustrate the execution of CPCES on the problem described in Figure 1. In the first round, sample set B is empty, and a candidate plan is $\pi_0 = \varepsilon$, i.e., the empty sequence. Then, using a SAT solver, CPCES generates the counter-example $(1, 5)$ (from this state, π_0 does not lead to the goal). So $(1, 5)$ is added to the sample set B. In the second round, another candidate plan is produced: $\pi_1 = \text{GO\_E} \times 2, \text{GO\_S} \times 2$. A new counter-example is generated, $(5, 1)$, and added to the sample which now evaluates to $\{(1, 5), (5, 1)\}$. In the third round, a new candidate plan $\pi_2 = \text{GO\_E} \times 4, \text{GO\_S} \times 4, \text{GO\_W} \times 2, \text{GO\_N} \times 2$ is produced that is valid for both $(1, 5)$ and $(5, 1)$. Now, CPCES cannot find any more counter-example, so π_2 is a solution to this problem.*

Reduction from Conformant Planning to Classical Planning We now review how the problem of producing a plan that is valid for a given sample is reduced to classical planning. Given this reduction, CPCES then calls a classical planner to find the candidate plan.

Algorithm 1 The conformant planner CPCES.

1: **input**: conformant planning problem P
2: **output**: a conformant plan, or **no plan**
3: $B := \emptyset$
4: **loop**
5: $\quad \pi := \texttt{produce-candidate-plan}(P, B)$
6: $\quad$ **if** there is no such π **then**
7: $\quad\quad$ **return no plan**
8: $\quad$ **end if**
9: $\quad q := \texttt{generate-counter-example}(P, \pi)$
10: $\quad$ **if** there is no such q **then**
11: $\quad\quad$ **return** π
12: $\quad$ **end if**
13: $\quad B := B \cup \{q\}$
14: **end loop**

Let $B = \{q_1, q_2, \ldots, q_n\}$ be the sample and, for any given state q, let $\varphi(q)$ be the proposition formula that is satisfied only by state q. Finding a plan that is valid for a sample B is equivalent to finding a solution to the conformant planning problem $P_B = \langle F, N, A, I_B, G\rangle$ where $I_B = \bigvee_{i\in\{1,\ldots,n\}} \varphi(q_i)$ is the condition satisfied by all the states in B and only them. The reduction of P_B to classical planning consists of n classical planning problems P_i in parallel where all actions are synchronised. Each problem P_i is defined over the set of facts F_i, a copy of the facts in F such that $F_i \cap F_j = \emptyset$ for all $i \neq j$. For any formula φ over F and any index $i \in \{1, \ldots, n\}$, we write $\varphi_{/i}$ the rewriting of φ where every fact $f \in F$ is replaced by its copy f_i from F_i. The reduction of P_B is then the classical planning problem $P' = \langle F', N', A', I', G'\rangle$ defined by:

- $F' = F_1 \cup \ldots \cup F_n$;
- $N' = N$;
- A' is such that for all $a = \langle pre, coneff\rangle \in N$, $A'(a) = \langle pre', coneff'\rangle$ where $pre' = pre_{/1} \wedge \ldots \wedge pre_{/n}$, and $eff' = \{\langle c_{/i}, eff^+_{/i}, eff^-_{/i}\rangle \mid \langle c, eff^+, eff^-\rangle \in coneff, i \in \{1, \ldots, n\}\}$;
- $I' = \varphi(q_1)_{/1} \wedge \ldots \wedge \varphi(q_n)_{/n}$; and
- $G' = G_{/1} \wedge \ldots \wedge G_{/n}$.

It has been proved that the valid plans of the conformant planning problem are the same as the valid plans of the reduced classical planning problem: $\Pi(P_B) = \Pi(P')$.

Example 2 *Reduction from conformant planning to classical planning. Let us look at an example to understand how the reduction of conformant planning works. In practice, CPCES uses the lifted representation to define the copies F_i of F. This is done via a multiple interpretation, by adding a parameter to each predicate which refers to the interpretation. Hence, a PDDL fact* `pred(p1,...,pk)` *(where* `pred` *is the predicate and* `pi` *are the parameters) is transformed into* `pred(p1,...,pk,i)` *where* `i` *is the interpretation and refers to the state q_i.*

In Example 1, the location of the robot is represented by a tuple (x, y), where $x \in \{x_1, \ldots, x_5\}$ and $y \in \{y_1, ..., y_5\}$. After the second round, CPCES has found two counter-examples, (x_1, y_5) and (x_5, y_1). In the conformant domain file, action GO_E *should be defined as:*

```
(:action GO_E
  :parameters ()
  :precondition ()
  :effect (and
    (when x1 (and x2 (not x1)))
    (when x2 (and x3 (not x2)))
    ...
  )
)
```

After applying reduction, action GO_E *in the interpretation domain file should be:*

```
(:action GO_E
  :parameters ()
  :precondition ()
  :effect
  (forall (?i - interpretation) (and
    (when (x1 ?i)
          (and (x2 ?i) (not (x1 ?i))))
    (when (x2 ?i)
          (and (x3 ?i) (not (x2 ?i))))
    ...
  ))
)
```

The instance file becomes:

```
(:init (and
    (x1 i1) (y5 i1)
    (x5 i2) (y1 i2)
))
```

The candidate plan generated from the interpretation file must be valid for all states of the sample. If no further counter-example exists, this candidate plan is a valid plan for the conformant planning problem.

SUPERB One way to improve CPCES is to generate optimal counter-examples, which is called SUPERB (Zhang, Grastien, and Scala 2020). In Algorithm 1 Line 9, SUPERB does not generate counter-examples randomly, but first computes *contexts* and *tags* and then updates counter-example until it finds one that contains as many new tags as possible. The optimal counter-examples found by SUPERB provide more information, so SUPERB can find a valid plan in fewer iterations and less time for problems with more than one context.

Fast Downward

PDDL uses predicates or propositions to describe planning tasks, while multi-valued planning task represents the tasks in SAS+ file by multi-valued state variables. The advantage of multi-valued encoding is that it compresses variables (Bäckström and Nebel 1995); it also enables richer heuristic functions. Multi-valued encoding is accomplished by computing sets of facts that are mutually exclusive and putting them into one variable.

Example 3 *In the grid world, the robot can only be in one of five columns, i.e. any two facts x and x', where $x, x' \in \{x_1, \ldots, x_5\}$, $x \neq x'$, are mutually exclusive. In multi-valued encoding, these five facts are compacted into one variable since only one of them can be true in any state:*

```
var0:
0: x1
...
4: x5
5: <none of those>
```

This is the same for y variables, where we use var1 to represents $\{y_1, \ldots, y_5\}$. As a result, the problem is compacted from ten variables to two.

The data structure that reflects the value transformation in a multi-valued state variable is the Domain Transition Graph (DTG). To represent the relationship between multi-valued variables, FD has to resort to another data structure: Causal Graph (CG). The third important data structure in FD is Successor Generator (SG) that determines the successor states from a given state and the given operators. SG contains two nodes: sector node and selection node. Sector node has a selection variable $v \in V$ and $|D_v + 1|$ outgoing edges. Each edge is labeled with d where $d \in D_v$ and an additional edge labeled with $\perp$ (don't care). The generator node is the leaf node of the sector node that stores all the applicable actions. The precondition of the action can be obtained by tracing the path from the root node of the SG to the current node. Therefore, it is possible to quickly determine the actions applicable to the current state, and then determine the subsequent state. FD uses multi-valued encoding combined with various search algorithms to find a plan efficiently.

4. Integrating FD Into CPCES

We have seen that CPCES can be used with any complete classical planner able to parse PDDL files and to output a valid plan if it exists, including FD. Prior to this work however, we have had mixed experience with FD. This is illustrated in Table 2. The column CPCES indicates the runtime for several instances from standard benchmarks for an implementation that uses FF; columns CFF and CLM report the runtime for two implementations using FD, one based on eager best-first search with FF heuristic and the other based on LAMA-2011. We see that, with one exception, the implementation that uses FF consistently performs better, and sometimes several orders of magnitude better.

We hypothesised that the poor performance of FD compared to FF was due to its translation to SAS+. Specifically, we see two issues:

1. The FD translator does not handle the forall PDDL construct properly, and produces subpar SAS+ files. With such poor representation, the FD search engine is then unable to fully exploit its capabilities.
2. A translation from PDDL to SAS+ is performed at each iteration of CPCES, which has a non-negligible cost.

We acknowledge here that the sequence of classical planning problems that CPCES send to the classical planner has a peculiar structure that can be exploited. Indeed, each problem is an "increment" of the previous problem in which the set of PDDL predicates is increased while the set of actions, as well as their relation with the existing predicates, remains unchanged. We formalise this intuition now.

Definition 1 (Problem Increment) *Two planning problems P_1 and P_2 are* independent *if their sets of action names are identical and their sets of facts are disjoint: $(N_1 = N_2) \wedge (F_1 \cap F_2 = \emptyset)$.*

Given two planning problems P_1 and P_2 that share the same set of action names, the merge *of these two problems, denoted $P_1 \oplus P_2$, is a problem $P = \langle F, N, A, I, G\rangle$ defined by:*

- $F = F_1 \cup F_2$;
- $N = N_1 = N_2$;
- A *is such that* $A(a) = \langle pre_1(a) \wedge pre_2(a), coneff_1(a) \cup coneff_2(a)\rangle$ *for all* $a \in N$;
- $I = I_1 \wedge I_2$; *and*
- $G = G_1 \wedge G_2$.

When P_1 preexists P_2, we use the notation P for P_1 and P_Δ for P_2, and call P_Δ an increment *to P.*

A problem increment is therefore a refinement of an existing planning problem that defines new predicates relevant to the planning task and specifies how the actions interact with these predicates. The increment also satisfies the property that it does not directly interact with the existing predicates. The definition applies both for PDDL and SAS+ representations.

Example 4 *Consider again the classical planning problem derived from our grid world example of Fig. 1 with a sample containing one state $(1, 5)$. The classical planning problem P_1 built via the reduction from the conformant planning problem contains the facts* `(x1 i1)`, ..., `(x5 i1)` *(as well as the facts pertaining to the vertical position).*

After the new counter-example $(5, 1)$ is added to the sample, we end up with a classical planning problem $P_{1,2}$ that contains the same facts as P_1, as well as the facts `(x1 i2)`, ..., `(x5 i2)`.

Furthermore, in $P_{1,2}$ the sets of facts are independent. This may not be obvious from the description of the actions because of the `forall` *construct, however if we unfold the* `forall`*, we end up with this description for action* GO_E*:*

```
(:action GO_E
  :parameters ()
  :precondition ()
  :effect (and
    (when (x1 i1)
          (and (x2 i1) (not (x1 i1))))
    ...
    (when (x1 i2)
          (and (x2 i2) (not (x1 i2))))
    ...
  ))
```

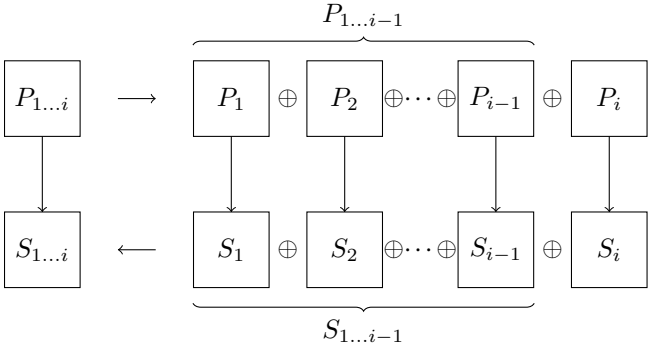

Figure 2: Graphical representation of the decomposition proposed in this paper: the classical planning problem $P_{1...i}$ that CPCES needs solved at the ith iteration can be decomposed into i planning problems P_j. Each of these problems can be translated into its own SAS+ representation, S_j, and these representations merged into a SAS+ representation $S_{1...i}$ equivalent to $P_{1...i}$.

The problem $P_{1,2}$ can therefore be split into two problems P_1 and P_2 where P_1 is exactly the problem from the previous iteration and P_2 is a classical problem defined only over the facts of the second interpretation `i2`.

In practice, because each P_i refers to a single interpretation, we do not even include the interpretation in the PDDL representation of P_i.

The benefit of the notions of merge and increment is that it allows for a simple translation to SAS+ when a translation of the original problem P already exists. This is summarised in the following lemma:

Lemma 1 *Let P be a PDDL planning problem, and let P_Δ be an increment to P. Let S and S_Δ be SAS+ representations of P and P_Δ respectively such that*

- $\Pi(S) = \Pi(P)$;
- $\Pi(S_\Delta) = \Pi(P_\Delta)$; *and*
- *the variables of S and S_Δ do not overlap.*

Then $S \oplus S_\Delta$ is a SAS+ representation of $P \oplus P_\Delta$ such that $\Pi(S \oplus S_\Delta) = \Pi(P \oplus P_\Delta)$.

Proof sketch: The proof relies on the property that when two independent planning problems are merged, be they modelled in PDDL or SAS+, the valid solutions of the resulting problem is the intersection of the solutions of the original problems. Hence:

$$\begin{aligned}\Pi(S \oplus S_\Delta) &= \Pi(S) \cap \Pi(S_\Delta)\\ &= \Pi(P) \cap \Pi(P_\Delta)\\ &= \Pi(P \otimes P_\Delta).\end{aligned}$$

From Lemma 1, we now have a procedure for computing efficiently SAS+ representations of the classical planning problems produced by CPCES. At each iteration, the

Algorithm 2 Incremental generation of candidate plan.

1: **method**: `produce-candidate-plan`(P, B)
2: **input**: conformant planning problem $P = \langle F, N, A, I, G\rangle$
3: **input**: belief B
4: **output**: a candidate plan, or **no plan**
5: **static**: a map $S : 2^F \rightarrow$ SAS_FILES
6: **if** $B = \emptyset$ **then**
7: **return** ε {Empty plan}
8: **end if**
9: **for all** $q \in B$ **do**
10: **if** $S(q)$ is undefined **then**
11: $S(q) :=$ `generate-sas-plus`$(\langle F, N, A, q, G\rangle)$
12: **end if**
13: **end for**
14: $SAS := \bigoplus_{q \in B} S(q)$
15: **return** `FD`(SAS)

classical planning problem $P_{1...i}$ is defined as the merge of all the classical planning problems P_1, $P_2 \ldots$ to P_i corresponding to the i interpretations (or counter-examples) generated so far throughout the procedure. $P_{1...i}$ is also the merge of $P_{1...i-1}$ and P_i. The SAS+ representation of $P_{1...i}$ can therefore be obtained by merging the SAS+ representation of $P_{1...i-1}$ (computed at the previous `CPCES` iteration) with a SAS+ representation of P_i. This is illustrated on Figure 2 where the vertical operation that translates from $P_{1...i}$ to $S_{1...i}$ is replaced with the decomposition (to the right), the vertical translation of each part, and the merge (to the left).

Our approach is summarised in Algo. 2 which implements the method `produce-candidate-plan` (Line 5) from `CPCES` (Algo. 1). This algorithm assumes a static map/dictionary that keeps track of the SAS translation associated with each state that it considered in the belief.[2] When new states are available (i.e., $S(q)$ is undefined), a SAS+ representation of this state is computed. Then, in Line 14, the SAS+ representation for the belief is built as the merge of each state's representation. Finally, `FD` is called with this file.

We see several benefits to this approach. First, it is very fast, as the translation of each problem P_i is very simple. Comparatively, the translation of $P_{1...i}$ can be more difficult because it is not always easy to determine, e.g., how to partition the facts in order to compute the best SAS+ variables. Second, it allows us to avoid the use of the `forall` PDDL construct which `FD` seems to have a hard time dealing with (we understand that the `forall` could be explicitly unfolded, and all the cases that it represents be enumerated, but this solution lacks elegance). Third, because each P_i problem is very small, we could actually spend extra resources to try to optimise the SAS+ encoding; this is something that would be riskier to do for $P_{1...i}$.

[2]This is useful in particular because some variants of `CPCES` remove and re-add counter-examples from the sample (Grastien and Scala 2018).

5. Experiments

We dub our approach as described in Algo. 2 *incremental `CPCES`*.

We compared five algorithms: classical `CPCES` with `FF` (`CPCES`), classical `CPCES` with `FD` in which we choose eager best-first search algorithm, best-first open list, FF heuristic (CFF), incremental `CPCES` in which we choose eager best-first search algorithm, best-first open list, FF heuristic (ICFF) classical `CPCES` with `FD` in which we choose `LAMA-2011` search component (CLM), and incremental `CPCES` in which we choose `LAMA-2011` search component (ICLM). All these are using the `SUPERB` strategy to generate counter-examples (Zhang, Grastien, and Scala 2020). `LAMA` (Richter, Westphal, and Helmert 2011) is a classical planning system based on heuristic forward search. `LAMA-2008` showed best performance among all planners in the sequential satisficing track of the International Planning Competition 2008. `LAMA-2011` greatly outperformed `LAMA-2008` in the competition. Since `LAMA` continuously searches valid plan until finding a best plan (shortest plan length), to improve the efficiency, we stop `LAMA` as soon as it finds the one valid plan, even though this may not be the best plan.

Expectations

Since we have simplified the interpretation instance file before executing multi-valued encoding, the time spent by ICFF and ICLM should perform better than classical CFF and CLM. However, the translation part in current incremental `CPCES` can be further improved, so we do not expect ICFF and ICLM to be better than `CPCES`.

Experimental Setup

To assess the usefulness of our algorithm, we ran experiments over the set of benchmarks from Grastien and Scala (Grastien and Scala 2018). We then implement `CPCES`, CFF, ICFF, CLM, ICLM to measure the amount of time needed to find a valid plan. All these five algorithms are implemented in python.

Our benchmark set contains 8 domains: DISPOSE, ONE-DISPOSE, BLOCKWORLD, LOOK-GRAB, RAOS-KEYS, BOMB, COINS, and UTS. Experiments were run on Ubuntu virtual machine with 8GB memory on Apple M1 chip Macbook Pro. Timeout was set to 600 secs. Each instance was solved three times and the reported time is the median one.

Results and Analysis

Table 2 reports the total time needed to find a valid plan by our five algorithms. First, we notice that except for some complex LOOK-GRAB instances, all the problems that `CPCES` can solve within the specified time can be solved by incremental `CPCES` as well. This result illustrates that incremental `CPCES` is indeed a practical way to using `FD` in `CPCES`.

Second, we find that incremental `CPCES` consistently performs better than simply replacing `FF` with `FD`. The difference is especially obvious in BOMB, COINS, DISPOSE,

Domain	Instance	With forall		Without forall	
		Number of Variables	Number of Iterations	Number of Variables	Number of Iterations
blockworld	p02	115	7	49	8
bomb	p20-5	646	21	120	21
coins	p15	773	15	884	30
dispose	p4-2	916	31	91	31
lookgrab	p4-2-2	442	4	12	4
onedispose	p3-2	591	32	104	27
raoskeys	p2	38	5	24	5
uts	p10	454	22	451	22

Table 1: This table shows two results. First, the number of variables in SAS+ file generated by PDDL file with forall and without forall statements. Second, the number of iterations used to solve corresponding problem.

and ONE-DISPOSE, which suggests incremental CPCES dramatically increases the efficiency of translation. LOOK-GRAB and UTS remain very hard for FD to solve.

Third, for some instances, like BOMB p100-1, BOMB p100-10, COINS p15, COINS p21, ONE-DISPOSE p4-2, incremental CPCES is even faster than classical CPCES. For some other instances, incremental CPCES has almost the same speed as classical CPCES in solving the problem. Such results are encouraging because it shows that sometimes the efficiency of incremental CPCES is close to CPCES.

Table 2 also shows the length of the computed plans. ICFF often computes shorter plans than FF. This is particularly impressive for LOOK-GRAB and UTS, which are the benchmarks that ICFF struggles with. We suspect that there is a correlation here: the reason why FF performs better on these benchmarks is that it is able to compute quickly a poor-quality plan.

Finally we notice a strange phenomenon, that is, the results of LAMA is not as good as expected. The time spends by ICLM is longer than ICFF, and the time spends by CLM is longer than CFF. Understanding this poor performance will require further work.

Table 1 displays the number of iterations used to solve the problem by SAS+ with forall (such as CFF) and without forall (such as ICFF), and the number of variables in SAS+ files generated from PDDL file with forall and without forall statements. If we compare those instances that have the same number of iterations, such as bomb, dispose, lookgrab, raoskeys, and uts, we find by using the forall PDDL, FD needs much more state variables than without forall. For coins, since this problem is solved by 30 rounds in CPCES without forall but solved by 15 rounds with forall, it is reasonable that the number of variables in without forall is larger. This experiments illustrates that without using forall, SAS+ file is more compacted than using forall. The larger number of variables is due to the inability for FD to find good partitions of mutually-exclusive facts, and explains in part why the incremental CPCES is more effective.

6. Conclusion

In this article we provide an algorithm to apply the FD system in CPCES, which is called incremental CPCES. In each round, incremental CPCES translates a small part of interpretation instance file to SAS+ file, avoiding spending long time in translating a complete problem. After translation, incremental CPCES merges the new SAS+ file with the previous SAS+ file, and then FD searches a plan based on that SAS+ file.

Our experiment results show that incremental CPCES with FD is able to reach the level of CPCES with FF. This is a very encouraging result as this gives us access to many different search techniques and heuristics.

We see several avenues for future work. We might be able to further improve the translation to SAS+. For instance, some of the variables that appear in several merged SAS+ files may have identical variables, particularly when some aspects of the planning problem are independent from the initial state. Identifying this symmetry could help further compact the model. Furthermore, similarly to our incremental construction of SAS+, we would like to insert incrementality directly into the plan search, i.e., to use as much as possible the existing search structure rather than replanning from scratch at each new CPCES iteration. Finally, we need to investigate further the search algorithms and heuristics that FD gives us access to; this is particularly interesting since, as we have seen in this paper, the classical planning problems have a specific structure unlike existing benchmarks.

References

Albore, A.; Ramirez, M.; and Geffner, H. 2011. Effective heuristics and belief tracking for planning with incomplete information. In *Twenty-First International Conference on Automated Planning and Scheduling*.

Bäckström, C., and Nebel, B. 1995. Complexity results for sas+ planning. *Computational Intelligence* 11(4):625–655.

Bonet, B., and Geffner, H. 2000. Planning with incomplete information as heuristic search in belief space. In *Fifth International Conference on AI Planning Systems (AIPS-00)*, 52–61.

Bryce, D. 2006. Pond: The partially-observable and nondeterministic planner. *Sixteenth International Conference on Automated Planning and Scheduling* 58.

Cimatti, A.; Roveri, M.; and Bertoli, P. 2004. Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence* 159(1-2):127–206.

Grastien, A., and Scala, E. 2017. Intelligent belief state sampling for conformant planning. In *International Joint Conference on Artificial Intelligence (IJCAI-17)*, 4317–4323.

Grastien, A., and Scala, E. 2018. Sampling strategies for conformant planning. In *Twenty-Eighth International Conference on Automated Planning and Scheduling*.

Grastien, A., and Scala, E. 2020. Cpces: A planning framework to solve conformant planning problems through a counterexample guided refinement. *Artificial Intelligence* 284:103271.

Haslum, P., and Jonsson, P. 1999. Some results on the complexity of planning with incomplete information. In *European Conference on Planning*, 308–318. Springer.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Hoffmann, J., and Brafman, R. I. 2006. Conformant planning via heuristic forward search: A new approach. *Artificial Intelligence* 170(6-7):507–541.

Hoffmann, J., and Nebel, B. 2001. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Richter, S.; Westphal, M.; and Helmert, M. 2011. Lama 2008 and 2011. In *International Planning Competition*, 117–124.

Rintanen, J. 2012. Engineering efficient planners with sat. In *European Conference on Artificial Intelligence (ECAI-2012)*, 684–689.

Smith, D., and Weld, D. 1998. Conformant graphplan. In *Fifteenth Conference on Artificial Intelligence (AAAI-98)*, 889–896.

To, S. T.; Son, T. C.; and Pontelli, E. 2015. A generic approach to planning in the presence of incomplete information: Theory and implementation. *Artificial Intelligence* 227:1–51.

Zhang, X.; Grastien, A.; and Scala, E. 2020. Computing superior counter-examples for conformant planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 10017–10024.

Domain	Instance	Total Time (s)					Plan Length	
		CPCES	CFF	ICFF	CLM	ICLM	CPCES	ICFF
blockworld	p01	0.04	**0.12**	0.16	**0.12**	0.15	5	**4**
blockworld	p02	0.12	**0.45**	0.53	24.12	**0.54**	18	**15**
blockworld	p03	2.05	-	-	-	**2.97**	66	-
blockworld	p04	91.55	-	-	-	**146.43**	141	-
bomb	p20-1	0.46	5.16	**1.57**	-	**1.56**	40	40
bomb	p20-5	0.63	8.02	**1.79**	-	**1.67**	40	40
bomb	p20-10	0.85	13.02	**2.12**	-	**1.83**	40	40
bomb	p20-20	1.29	28.12	**3.21**	-	**2.13**	40	40
bomb	p100-1	34.24	-	**29.89**	-	**55.37**	200	200
bomb	p100-5	71.34	-	**489.46**	-	**111.17**	200	200
bomb	p100-10	119.66	-	-	-	**198.87**	200	-
coins	p10	0.35	**0.62**	0.66	-	**0.52**	34	34
coins	p12	1.51	4.02	**1.96**	-	**4.18**	**69**	78
coins	p15	2.62	6.12	**2.66**	-	**1.08**	81	**78**
coins	p16	1.28	5.43	**2.06**	-	**4.33**	**106**	110
coins	p17	0.78	3.16	**1.29**	-	**3.03**	106	**105**
coins	p18	0.79	4.32	**1.71**	-	**2.15**	110	**96**
coins	p19	0.99	5.4	**2.25**	-	**2.47**	**100**	111
coins	p20	0.8	3.64	**1.38**	-	**2.21**	**109**	114
coins	p21	-	-	-	-	**20.28**	-	-
dispose	p-4-1	0.57	2.92	**1.33**	-	**1.36**	61	**51**
dispose	p-4-2	1.67	9.48	**3.23**	-	**5.29**	**58**	72
dispose	p-4-3	4.28	20.5	**6.3**	-	**25.13**	**82**	94
dispose	p-8-1	35.03	-	**76.48**	-	**76.25**	326	**291**
lookgrab	p-4-1-1	0.22	3.11	**2.07**	33.55	**2.42**	26	**14**
lookgrab	p-4-1-2	0.06	36.91	**27.34**	39.57	**27.28**	4	4
lookgrab	p-4-1-3	0.05	**21.5**	30.46	**22.49**	30.38	4	4
lookgrab	p-4-2-1	0.37	4.17	**3.58**	32.69	**3.68**	30	**14**
lookgrab	p-4-2-2	0.12	75.65	**49.2**	80.07	**49.16**	4	4
lookgrab	p-4-2-3	0.07	**43.97**	54.5	**45.96**	54.54	4	4
lookgrab	p-4-3-1	0.31	**6.26**	13.58	155.01	**26.95**	24	**14**
lookgrab	p-4-3-2	0.13	113.85	**80.47**	122.13	**80.67**	4	4
lookgrab	p-4-3-3	0.11	**65.77**	84.12	**69.19**	84.58	4	4
lookgrab	p-8-1-1	42.02	595.62	**184.63**	-	**222.9**	180	**76**
lookgrab	p-8-1-2	6.7	-	-	-	-	100	-
lookgrab	p-8-1-3	3.59	-	-	-	-	58	-
lookgrab	p-8-2-1	49.34	-	-	-	-	196	-
lookgrab	p-8-2-2	27.48	-	-	-	-	82	-
lookgrab	p-8-2-3	12.55	-	-	-	-	64	-
lookgrab	p-8-3-1	34.2	-	-	-	-	146	-
lookgrab	p-8-3-2	8.41	-	-	-	-	58	-
lookgrab	p-8-3-3	5.58	-	-	-	-	46	-
onedispose	p-2-2	0.21	**0.58**	0.59	103.08	**0.56**	26	**18**
onedispose	p-2-3	0.4	0.97	**0.84**	-	**1.91**	38	**29**
onedispose	p-3-2	2.03	3.81	**2.57**	-	**4.31**	70	**44**
onedispose	p-3-3	60.99	**33.79**	78.21	-	**105.75**	131	**66**
onedispose	p-4-2	-	162.21	**21.47**	-	**266.35**	-	84
raoskeys	p2	0.07	**0.26**	0.41	0.39	**0.37**	17	17
raoskeys	p3	0.86	-	-	-	**244.62**	57	-
uts	p1	0.04	**0.17**	0.27	**0.18**	0.26	4	4
uts	p2	0.08	**0.29**	0.36	0.54	**0.36**	10	10
uts	p3	0.12	**0.44**	0.55	-	**0.7**	18	**16**
uts	p4	0.19	**0.74**	0.82	-	**0.9**	22	22
uts	p5	0.34	1.24	**0.96**	-	**1.21**	28	28
uts	p6	0.49	2.02	**1.4**	-	**2.43**	40	**34**
uts	p7	0.7	3.76	**2.34**	-	**5.98**	40	40
uts	p8	1.12	5.57	**4.48**	-	**5.19**	51	**46**
uts	p9	1.61	9.08	**7.36**	-	**9.59**	60	**52**
uts	p10	2.07	14.32	**13.08**	-	**16.91**	69	**58**
uts	p20	2.12	14.18	**13.11**	-	**14.31**	67	**58**
uts	p30	8.36	**105.49**	186.55	-	**188.83**	88	88
uts	p40	37.1	**509.19**	-	-	-	128	-

Table 2: The total conformant planning time of different algorithms. "CPCES" is classical CPCES with FF; "CLM" is classical CPCES with FD planner using LAMA-2011 search component; "ICLM" is incremental CPCES using LAMA-2011 search component; "CFF" is classical CPCES with FD planner using eager best-first search algorithm, best-first open list, FF heuristic; "ICFF" is incremental CPCES using eager best-first search algorithm, best-first open list, FF heuristic.