A New Approach to Conformant Planning via Classical Planners

Khoi Nguyen and **Vien Tran** and **Tran Cao Son** and **Enrico Pontelli**
Computer Science Department
New Mexico State University
Email: {knguyen,vtran,tson,epontell}@cs.nmsu.edu

Abstract

In this paper, we introduce a new approach to conformant planning via classical planners. We view a conformant planning problem as a set of classical planning problems, called *sub-problems*, and solve it using a generate-and-complete algorithm. Key to this algorithm is a procedure which takes a solution of a sub-problem and generates a solution for other sub-problems. We implement this algorithm in a new planner, called CPCL and evaluate it empirically against state-of-the-art conformant planners using various benchmarks. The experimental results show that CPCL is superior to other planners in most benchmarks, both in performance and in scalability.

Introduction

Conformant planning is the problem of computing a sequence of actions that achieves a goal in presence of incomplete information about the initial state (Smith and Weld 1998). By definition, conformant planning searches for the plan in the belief state space. Due to the incomplete information, the belief state usually has large size which leads to difficulty in searching for the solution. Thus one way to address the problem is to translate the conformant planning problem to a classical planning problem which has been done by `t0` (Palacios and Geffner 2006).

The idea of using a classical planning system to solve a non-classical planning problem has been applied to other types of planning problems such as probabilistic planning. FF-Replan (Yoon *et al.* 2007), the winner of the 2004 IPC, which solves a probabilistic planning problem by (*i*) translating the problem into a classical planning problem, (*ii*) computing a solution using a classical planner (FF), and (*iii*) replanning whenever necessary.

It is interesting to contrast the approaches adopted in `t0` and FF-Replan. While the translation employed by `t0` could produce a new problem whose size is exponential in the size of the original one (if completeness is required), and thus making the problem more difficult, the *determinizing* process of FF-Replan simplifies the original problem by removing all information related to non-determinicity. This raises the interesting question of whether an alternative approach to `t0`, perhaps in a similar spirit to that of FF-Replan, could produce similar results in conformant planning. It is clear that the algorithm of FF-Replan cannot be applied to conformant planning, since conformant planning does not interleave planning and execution.

In this paper, we develop a new approach to conformant planning using classical planners. We implement the idea in a system, called CPCL, and evaluate it against state-of-the-art conformant planners using several benchmarks. The experimental results show that the new planner performs exceptionally well in almost all domains and scales up better than other planners.

Conformant Planning Problem

A conformant planning problem P is specified by a tuple $\langle F, O, I, G\rangle$, where F is a set of propositions, O a set of action descriptions, I a set of formulae describing the initial state of the world, and G a formula describing the goal.

A *literal* is a proposition $p \in F$ or its negation $\neg p$. $\bar{\ell}$ denotes the complement of the literal ℓ, and it is defined as $\bar{\ell} = \neg\ell$, where $\neg\neg p = p$ for $p \in F$. For a set of literals L, $\overline{L} = \{\bar{\ell} \mid \ell \in L\}$; and L is often used to represent $\wedge_{\ell\in L}\ell$.

A set of literals X is *consistent* if there exists no $p \in F$ such that $\{p, \neg p\} \subseteq X$. A *state* s is a consistent and *complete* set of literals, i.e., s is consistent, and for each $p \in F$, either $p \in s$ or $\neg p \in s$. A *belief state* is a set of states. A set of literals X satisfies a literal ℓ (resp. a set of literals Y) iff $\ell \in X$ (resp. $Y \subseteq X$).

Each action a in O is associated with a precondition, denoted by $pre(a)$, and a set of conditional effects of the form $\psi \rightarrow \ell$ (denoted by $a : \psi{\rightarrow}\ell$), where $pre(a)$ and ψ are sets of literals and ℓ is a literal. We often write $a : \psi \rightarrow \ell_1, \ldots, \ell_k$ as a shorthand for the set $\{a : \psi \rightarrow \ell_1, \ldots, a : \psi \rightarrow \ell_k\}$.

The initial state I is a collection of literals, `one-of` clauses (each of the form $\texttt{one-of}(\psi_1, \ldots, \psi_n)$), and `or` clauses (each of the form $\texttt{or}(\psi_1, \ldots, \psi_m)$) where each ψ_i is a set of literals.

A set of literals X satisfies the `one-of` clause $\texttt{one-of}(\psi_1, \ldots, \psi_n)$ if there exists some i, $1 \leq i \leq n$, such that $\psi_i \subseteq X$ and for every $j \neq i$, $1 \leq j \leq n$, $\overline{\psi_j} \cap X \neq \emptyset$. X satisfies the `or` clause $\texttt{or}(\psi_1, \ldots, \psi_m)$ if there exists some $1 \leq i \leq m$ such that $\psi_i \subseteq X$.

By $ext(I)$ we denote the set of all states satisfying every literal in I, every `one-of` clause in I, and every `or` clause in I (e.g., if F=$\{g, f, h\}$ and I=$\{\texttt{or}(g, h), \texttt{one-of}(f, h)\}$ then $ext(I) = \{\{g, h, \neg f\}, \{g, \neg h, f\}, \{\neg g, h, \neg f\}\}$).

The goal G is a collection of literals and `or` clauses.

Given a state s and an action a, a is executable in s if $pre(a) \subseteq s$. A conditional effect $a : \psi \rightarrow l$ is applicable in s if $\psi \subseteq s$. The set of effects of a in s, denoted by $e_a(s)$, is defined as: $e_a(s) = \{l \mid a : \psi \rightarrow l \in O \text{ is applicable in } s\}$. The execution of a in a state s results in a successor state $succ(a, s)$, where $succ(a, s) = (s \cup e_a(s)) \setminus \overline{e_a(s)}$ if a is executable in s, and $succ(a, s) = \mathbf{failed}$, otherwise. Using this function, we define $\widehat{succ}$ for computing the state resulting from the execution of a sequence of actions $\alpha = [a_1, \ldots, a_n]$: $\widehat{succ}(\alpha, s) = s$ if $n = 0$; and $\widehat{succ}(\alpha, s) = succ(a_n, \widehat{succ}(\beta, s))$ if $n > 0$ where $\beta = [a_1, \ldots, a_{n-1}]$ and $\widehat{succ}(\gamma, \mathbf{failed}) \stackrel{def}{=} \mathbf{failed}$ for any sequence of actions γ. For a belief state S and action sequence α, let $\widehat{succ}^*(\alpha, S) = \{\widehat{succ}(\alpha, s) \mid s \in S\}$ if $\widehat{succ}(\alpha, s) \neq \mathbf{failed}$ for every $s \in S$; and $\widehat{succ}^*(\alpha, S) = \mathbf{failed}$, otherwise. α is a *solution* of P iff $\widehat{succ}^*(\alpha, ext(I)) \neq \mathbf{failed}$ and G is satisfied in every state belonging to $\widehat{succ}^*(\alpha, ext(I))$.

Conformant Planning using a Classical Planner—An Intuition

In this section, we present our idea of how to use a classical planner to solve conformant planning problems. Let us illustrate our idea in an example.

Example 1. Let us consider a small instance (denoted by P_1) of the coin problem from the IPC 2008 (Bryce and Buffet 2008). In this problem, we have one elevator e_0 which can move between floors f_0 and f_1 if one of the actions go_up or go_down is performed, depending on the location of the elevator. Each floor has two positions p_0 and p_1. An agent can enter (or exit) the elevator by using the action *step_in* (or *step_out*). The agent can also move between positions on the same floor by using the actions *move_left* and *move_right*. If the agent is at the same position as a coin, he can collect the coin by using the action *collect*.

There is one coin, denoted by c_0, whose initial location is only partially known: the coin is on the floor f_1 but it is not known whether it is at the position p_0 or p_1. Furthermore, the elevator's location is initially unknown: it can be at f_0 or f_1. Initially, the agent is at the position p_0 of floor f_0.

The goal is of the problem is to collect the coin c_0—denoted by the fluent $have(c_0)$.

Let us explore the encoding $P_1 = \langle F, O, I, G\rangle$. In this domain, the set of propositions F contains the following propositions:[1]

- $at(f, p)$: the agent is at position p of floor f,
- $in(e, f)$: the elevator is at the floor f,
- *coin_at*(c, f, p): the coin c is at position p of the floor f,
- $inside(e)$: the agent is inside the elevator e,
- $have(c)$: the agent has the coin.

where $f \in \{f_0, f_1\}$, $p \in \{p_0, p_1\}$, $e = e_0$, and $c = c_0$. The set of actions O with their conditional effects is given next:

[1]For simplicity, we omit the predicate $shaft(e, p)$, $dec\_f(f, f')$ and $dec\_p(p, p')$ that denotes the spatial relation between elevators, positions and floors as they are static and will be compiled away by the preprocessor of most planners.

$go\_up(e, f_0, f_1) : in(e, f_0) \rightarrow in(e, f_1), \neg in(e, f_0)$
$go\_down(e, f_1, f_0) : in(e, f_1) \rightarrow in(e, f_0), \neg in(e, f_1)$
$step\_in(e, f, p) : in(e, f) \rightarrow inside(e), \neg at(f, p)$
$step\_out(e, f, p) : in(e, f) \rightarrow at(f, p), \neg inside(e)$
$move\_left(f, p_1, p_0) : true \rightarrow at(f, p_0), \neg at(f, p_1)$
$move\_right(f, p_0, p_1) : true \rightarrow at(f, p_1), \neg at(f, p_0)$
$collect(c_0, f, p) : coin\_at(c_0, f, p) \rightarrow have(c_0), \neg coin\_at(c_0, f, p)$

where $f \in \{f_0, f_1\}$, and $p \in \{p_0, p_1\}$. In addition,

$$\begin{array}{ll} pre(go\_up(e, f_0, f_1)) & = \{\} \\ pre(go\_down(e, f_1, f_0)) & = \{\} \\ pre(step\_in(e, f, p)) & = \{at(f, p)\} \\ pre(step\_out(e, f, p)) & = \{inside(e)\} \\ pre(move\_left(f, p, p')) & = \{at(f, p)\} \\ pre(move\_right(f, p, p')) & = \{at(f, p)\} \\ pre(collect(c_0, f, p)) & = \{at(f, p)\} \end{array}$$

The initial state of the problem can be given by $I = I^d \cup I^o$ where $I^d = \{at(f_0, p_0)\}$ and

$$I^o = \{\texttt{one-of}(coin\_at(c_0, f_1, p_0), coin\_at(c_0, f_1, p_1)), \texttt{one-of}(in(e_0, f_0), in(e_0, f_1))\}.$$

Finally, the goal of the problem is given by $G = \{have(c_0)\}$. Let

$$\begin{array}{l} u_0 = \{at(f_0, p_0), coin\_at(c_0, f_1, p_0), in(e_0, f_0)\} \\ u_1 = \{at(f_0, p_0), coin\_at(c_0, f_1, p_1), in(e_0, f_0)\} \\ u_2 = \{at(f_0, p_0), coin\_at(c_0, f_1, p_0), in(e_0, f_1)\} \\ u_3 = \{at(f_0, p_0), coin\_at(c_0, f_1, p_1), in(e_0, f_1)\} \end{array}$$

Define $s_i = comp(u_i)$. We have that $ext(I) = \{s_0, s_1, s_2, s_3\}$. One of the solutions to this problem is:

$$\alpha = \left[\begin{array}{l} go\_down(e_0, f_1, f_0), step\_in(e_0, f_0, p_0), \\ go\_up(e_0, f_0, f_1), step\_out(e_0, f_1, p_0), \\ collect(c_0, f_1, p_0), move\_right(f_1, p_0, p_1), \\ collect(c_0, f_1, p_1) \end{array}\right]$$

□

Let us introduce the notion of sub-problem.

Definition 1. *Let $P = \langle F, O, I, G\rangle$ be a conformant planning problem. For every $s \in ext(I)$, the planning problem $P(s) = \langle F, O, s, G\rangle$ is called a* sub-problem *of P.*

Clearly, for every $s \in ext(I)$, $P(s)$ is a classical planning problem. It is obvious that solution of P can be founded by selecting (randomly) a sub-problem $P(s)$ of P and repeatedly (**i**) computing a solution α of $P(s)$; and (**ii**) testing if α is a solution of P until a solution of P is found. Even though this process is theoretically sound, such a brute-force computation may not be practical for different reasons. First, the set of solutions of $P(s)$ is generally infinite and thus generating all solutions is impractical. Second, for efficiency and space reasons, most state-of-the-art planners use heuristics and remove some parts of the search space (non-optimal planners avoid exploring the same state twice while optimal planners ignore paths which violate some criteria, e.g., cost of current path is greater than an established threshold). Third, the process ignores the relationships among the sub-problems which are often useful in solving the problem. Inspired by FF-Replan, we develop an algorithm for conformant planning using a modification of the steps (**i**)-(**ii**).

Although every solution α of P is a solution of $P(s)$, it is often the case that we can find a subsequence[2] α_s of α such that α_s is a solution of $P(s)$. For example, for the problem P_1 in Example 1, the following sub-sequences α_{s_i} of α, are solutions of $P(s_i)$:

$$\begin{aligned}
\alpha_{s_0} &= [step_in(e_0, f_0, p_0), go_up(e_0, f_0, f_1), \\
&\qquad step_out(e_0, f_1, p_0), collect(c_0, f_1, p_0)] \\
\alpha_{s_1} &= [step_in(e_0, f_0, p_0), go_up(e_0, f_0, f_1), \\
&\qquad step_out(e_0, f_1, p_0), move_right(f_1, p_0, p_1), \\
&\qquad collect(c_0, f_1, p_1)] \\
\alpha_{s_2} &= [go_down(e_0, f_1, f_0), step_in(e_0, f_0, p_0), \\
&\qquad go_up(e_0, f_0, f_1), step_out(e_0, f_1, p_0), \\
&\qquad collect(c_0, f_1, p_0)] \\
\alpha_{s_3} &= [go_down(e_0, f_1, f_0), step_in(e_0, f_0, p_0), \\
&\qquad go_up(e_0, f_0, f_1), step_out(e_0, f_1, p_0), \\
&\qquad move_right(f_1, p_0, p_1), collect(c_0, f_1, p_1)]
\end{aligned}$$

Thus, one way of solving conformant problem is to modify a solution α_s of a sub-problem $P(s)$ of P—by adding actions—so that it becomes a solution of P, as shown in the next example.

Example 2. Let us consider the problem P_1 from Example 1. Let us assume that the classical planner selects s_0 from $ext(I)$ and generates $\alpha_{s_0} = [step\_in(e_0, f_0, p_0), go\_up(e_0, f_0, f_1), step\_out(e_0, f_1, p_0), collect(c_0, f_1, p_0)]$ as its first solution.

α_{s_0} is not a solution of $P_1(s_1)$. However, α_{s_0} and α_{s_1} share the first three actions and α_{s_0} is executable in s_1. Furthermore, for $s'_1 = \widehat{succ}(s_1, \alpha_{s_0})$, we have that[3] α_{01}=$\alpha_{s_0} \circ \beta$, where $\beta = [move\_right(f_1, p_0, p_1), collect(c_0, f_1, p_1)]$, is a solution of $P_1(s_1)$.

Observe that α_{01} is also a solution of $P_1(s_0)$. Thus, α_{01} is a solution of the planning problem $\langle P, O, \{\mathtt{or}(s_0, s_1)\}, G\rangle$.

Let us consider $P_1(s_2)$. Checking to see whether α_{01} is a solution of $P_1(s_2)$ reveals that its first action, $step\_in(e_0, f_0, p_0)$, is executable in s_2; however, one of the effects of this action, $\neg at(f_0, p_0)$, is not contained in $succ(step\_in(e_0, f_0, p_0), s_2)$ because the precondition $in(e_0, f_0)$ of the conditional effect $step\_in(e_0, f_0, p_0) : in(e_0, f_0) \rightarrow \neg at(f_0, p_0)$ is not satisfied in s_2. In order to achieve this condition for $step\_in(e_0, f_0, p_0)$ from s_2, we should execute first the action $go\_down(e_0, f_1, f_0)$. Let $\alpha_{012} = [go\_down(e_0, f_1, f_0)] \circ \alpha_{01}$. We can verify that α_{012} is a solution of $P_1(s_2)$. Moreover, we can also see that α_{012} is a solution of $P_1(s_0)$, $P_1(s_1)$, and $P_1(s_3)$. In other words, α_{012} is a solution of P_1. Observe that α_{012} is identical to the solution given in Example 1. □

The above example shows that it is possible to use a classical planner and search in the original state space of a conformant planning problem for a solution by repeating the following two steps until a solution is found:

- Compute a solution α_s of a sub-problem $P(s)$ of P, and
- Incrementally repair α_s to meet the needs of other sub-problems of P.

[2]We say that α is a subsequence of β is α is obtained by removing any number of elements from β.

[3]$\circ$ denotes concatenation of two lists.

CPCL—A New Conformant Planner

We now describe a novel algorithm, called CPCL, which solves a conformant planning problem by solving several classical planning problems.

Algorithm

Alg. 1 shows the main search algorithm of the planner CPCL. $plan(X)$ plays the role of a classical planner that returns a set of solutions of X. We assume that $plan(X)$ returns one solution at a time, **nil** if there is no more solution, or **failed** if X does not have a solution. $is\_solution(\beta, P)$ checks whether or not β is a solution of the problem P.

Algorithm 1 CPCL(P)

```
1: Input: A planning problem P = ⟨F, O, I, G⟩
2: Output: A solution for P
3: Let Σ = [s_0, ..., s_n] = ext(I)        {Compute ext(I)}
4: α_{s_0} = plan(P(s_0))                  {Get a solution of P(s_0)}
5: if α_{s_0} = failed then return failed
6: while α_{s_0} ≠ nil do
7:    if is_solution(α_{s_0}, P) then return α_{s_0}
8:    else β = completion(α_{s_0}, P, Σ, 1)
9:       if is_solution(β, P) then return β
10:   α_{s_0} = plan(P(s_0))
11: end while
12: return unknown
```

$completion(\alpha, P, \Sigma, j)$ $(0 \leq j \leq n)$ takes a problem P, whose initial belief state is $[s_0, \ldots, s_n]$, and a solution $\alpha_{s_{j-1}}$ of $P(s_{j-1})$ (where $s_{-1} = s_n$), and attempts to create solutions α_{s_i} for $P(s_i)$, $i = j+1, \ldots, n$.

The procedure constructs a solution of the sub-problem $P(s_i)$ from the solution $\alpha_{s_{i-1}}$ of $P(s_{i-1})$, by inserting actions into $\alpha_{s_{i-1}}$. To achieve this, the procedure starts with the state s_i and an empty plan, considers each action a in $\alpha_{s_{i-1}}$, and executes the following tasks:

- **Task 1**: inserts a sequence of actions before a, so that *(1)* a is executable and *(2)* the execution of a maintains some effects of a. If this fails then the algorithm stops and declares that the original plan cannot be extended to a plan for s_i.
- **Task 2**: makes sure that the final sequence of actions α_{s_i} achieves the goal of $P(s_i)$ and this may require adding extra actions at the end of α_{s_i}.

Implementation and Evaluation

We develop CPCL using the source code of LAMA, the winner of the deterministic track of IPC 2008 because of its exceptional performance and its object-oriented implementation which allows for an easy instantiation a new planning module with different initial state and goal. In order to test CPCL on the wide range of collected conformant planning problems and achieve good performance, we have made the following modifications to LAMA:

- The parser has been modified to consider various types of actions that were rejected by LAMA;
- The parser has also been modified to enable the computation of the initial belief state of the planning problem;

- Algorithms 1 have been integrated to LAMA. To generate more than one solution of a problem, we disable the A* search feature of LAMA by keeping the open list (queue of unexplored nodes) after the first solution is found and continue the search for the next solution if needed.

We compare CPCL with other state-of-the-art planners—i.e., CPA (Tran *et al.* 2009), DNF (To *et al.* 2009), and `t0` (Palacios and Geffner 2009)—on problems from the literature and previous planning competitions. The experiment have been performed on a Core 2 2.66GHz machine, with 4Gb memory, with a run-time cutoff of 30 minutes.

The benchmark set contains 731 instances of 18 domains from the recent IPCs (2006 and 2008) and from the distribution of CFF and `t0`. CPCL is able to solve 684 instances while other planners can only solve up to 417 instances(333, 360 and 417 for CPA(H), `t0` and DNF respectively).

Instance	CPA(H)	`t0`	DNF	CPCL
blw-03	20.4/205	48.51/80	307/325	**1.3**/266
blw-04	AB	AB	AB	**29.5**/1384
coins-10	**0.03**/48	0.04/26	0.20/27	0.037/36
coins-30	AB	AB	AB	**1.0**/1107
comm-15	2.29//95	**0.092**/110	3.43/125	0.1/97
comm-25	1222/389	1.55/453	1797/501	**0.8**/294
sortnet-5	**0.02**/13	0.18/15	0.03/15	0.05/15
sortnet-15	240/74	AB	**35**/118	63.9/120
sortnum-5	AB	1.9/10	1.67/10	**0.81**/10
sortnum-20	AB	AB	AB	**12.3**/190
uts-30	4.9/74	0.79/67	1.39/73	**0.17**/64
uts-cycle-03	**0.01**/3	0.14/3	0.01/3	0.04/3
uts-cycle-15	AB	AB	AB	**1314**/272
raos-keys-02	0.26/32	**0.02**/21	0.09/39	0.05/38
raos-keys-04	AB	AB	AB	**16.78**/163
forest-03	AB	0.62/45	TO	**0.46**/167
forest-09	AB	AB	AB	**183.8**/963

Table 1: Results for IPC 2006/08 Domains (Time in *seconds*)

Domains from IPCs: Table 1 contains the results of our experiments with domains from the IPCs 2006 and 2008—in terms of the time and length of the first solution reported by each planner. Boldface indicates the fastest planner. AB denotes an execution aborted by the planner due to"out of memory," and TO denotes time-out. CPCL performs exceptionally well, both in term of efficiency and scalability. CPCL consistently outperforms other planners in large instances. For space reason, we omit the result on domains from the distribution of CFF and `t0`.

Conclusion

In this paper, we proposed a novel approach to conformant planning using classical planner. We implemented the new algorithm using the source code of the classical planner LAMA, and evaluated the new planner, CPCL, against state-of-the-art conformant planners. CPCL outperforms other planners in both performance and scalability, indicating that the proposed approach is a strong alternative to current state-of-the-art approaches.

It is well-known that the scalability and performance of a conformant planner depend on two factors: the ability to deal with the potential large size of the initial belief state and the ability to guide the search. CPCL deals with these two problems by taking advantage of the best from the research in conformant and in classical planning.

CPCL copes with the huge size of the initial belief state by considering each possible initial state *separately*, which reduces the memory requirements—which is often the problem for other planners. This also allows CPCL to easily take advantage of techniques that have been developed in conformant planning research for reducing the size of the initial belief state. By converting the problem to a classical problem, CPCL can exploit the best heuristic classical planners in computing a solution. Furthermore, the generate-and-complete algorithm allows CPCL to generate a solution by solving multiple smaller problems.

We observe that, even though CPCL can solve a wide range of benchmarks from various sources, which seem to be difficult for other state-of-the-art conformant planners, there are still domains in which CPCL does not work well. Among them, the `adder` domain seems to be the most difficult one. This domain is special in that the size of the initial belief state is very small, but the number of actions which can be executed in a state is very large. Furthermore, the conditional effects of the actions are much more complex than those in other domains. We hypothesize that these two factors make this domain difficult for conformant planners.

References

D. Bryce and O. Buffet. The uncertainty part of the 6th IPC, 2008.

R. Brafman and J. Hoffmann. Conformant planning via heuristic forward search: A new approach. In *ICAPS 2004*, pages 355–364, 2004.

H. Palacios and H. Geffner. Compiling Uncertainty Away: Solving Conformant Planning Problems Using a Classical Planner (Sometimes). In *AAAI*, 2006.

H. Palacios and H. Geffner. Compiling Uncertainty Away in Conformant Planning Problems with Bounded Width. *JAIR*, 35:623–675, 2009.

S. Richter and M. Westphal. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *JAIR*, 39:127–177, 2010.

D.E. Smith and D.S. Weld. Conformant graphplan. In *AAAI*, pages 889–896, 1998.

S. T. To, E. Pontelli, and T. C. Son. A conformant planner with explicit disjunctive representation of belief states. In *ICAPS 2009*, AAAI, 2009.

D. V. Tran, H. K. Nguyen, E. Pontelli, and T. C. Son. Improving performance of conformant planners: Static analysis of declarative planning domain specifications. In *PADL 2009*, pages 239–253. Springer, 2009.

S.W. Yoon, A. Fern, and R. Givan. FF-Replan: A baseline for probabilistic planning. *ICAPS*, 352–259. AAAI. 2007.