A Conformant Planner with Explicit Disjunctive Representation of Belief States

Son Thanh To and **Enrico Pontelli** and **Tran Cao Son**
New Mexico State University
Dept. Computer Science
sto@cs.nmsu.edu, epontell@cs.nmsu.edu, tson@cs.nmsu.edu

Abstract

This paper describes a novel and competitive complete conformant planner. Key to the enhanced performance is an efficient encoding of belief states as disjunctive normal form formulae and an efficient procedure for computing the successor belief state. We provide experimental comparative evaluation on a large pool of benchmarks. The novel design provides great efficiency and enhanced scalability, along with the intuitive structure of disjunctive normal form representations.

Introduction

Conformant planning is the problem of finding a sequence of actions that achieves a goal from every possible initial state of the world. Typically, conformant planning arises in presence of incomplete knowledge about the initial state of the world. Since its introduction in (Smith and Weld 1998), conformant planning has attracted the attention of several researchers. A number of efficient and sophisticated conformant planners have been developed, and a conformant track has been a part of the bi-annual international planning competition for several years.

The development of a best-first search and progression-based planner begins with the selection of a representation language and the definition of a progression function that, given a state and an action, computes the next state of the world. To deal with incomplete knowledge about the initial state of the world, the notion of *belief state* has been introduced—defined as a set of states. The progression function is accordingly extended to define a transition function between belief states.

The majority of progression-based planners use PDDL (Ghallab et al. 1998) as the input representation language, extended with features for representing incomplete information about the initial state (e.g., `oneof` clauses). The advantage of using PDDL is twofold. First, there are several conformant planning benchmarks encoded in PDDL, providing a wide range of use cases for testing. Second, it makes the comparison between systems easier.

A direct implementation of a best-first search and progression-based planner begins with the critical decision of selecting a *representation for the belief states*. The second important question is related to the development of heuristics functions to guide the search process. Before the introduction of CPA (Tran et al. 2009; Son et al. 2005), two main approaches had been commonly employed. The first approach relies on *ordered binary decision diagrams (OBDD)* (Bryant 1992). The advantage of this approach is the availability of efficient libraries for creating and manipulating OBDDs. This allows the developer to focus on the second issue, i.e., the development of heuristic. The system POND (Bryce et al. 2006) is a good example of a planner that relies on OBDD technology. The main disadvantage of this approach is that the size of the OBDD can be very large—the structure of the OBDD is sensitive to the ordering of variables and the manipulation of the OBDD might require intermediate OBDDs of exponential size. The second approach, used in CFF(Brafman and Hoffmann 2004), does not employ an explicit representation of belief states during the computation. Instead, the belief state at any point in the search process is implicitly described by the sequence of actions used to reach that belief state in the search. In this approach, checking entailment in a belief state requires encoding the initial state and the effects of the sequence of actions as a propositional theory and using a SAT solver. The advantage of this approach is that it is less demanding in terms of memory w.r.t. planners using OBDDs. The tradeoff is represented by the need to call a SAT solver. Systems like CFF introduce techniques to minimize the number of calls made to the SAT solver.

The negative aspects encountered in both methods are not surprising, considering that the problem of checking whether a proposition holds after the execution of an action in presence of incomplete information is a co-NP complete problem (Baral et al. 2000).

An alternative, indirect, approach to best-first search and progression-based planning consists of transforming a conformant planning problem into a *classical planning problem*, and then using an off-the-shelf classical planner to search for solutions. The planner `t0` (Palacios and Geffner 2007), and its predecessor `cf2cs(ff)`, employ this method.

The introduction of CPA (Son et al. 2005) brought a different perspective to the development of conformant planners. Instead of using the complete transition function in the search, CPA uses an *approximation*, first described in (Son and Baral 2001). A belief state is approximated by the in-

tersection of the states it contains. The advantage of this approach lies in the low-complexity of the approximation: the successor (approximated) belief state can be computed in polynomial time. This approach proves to be adequate in a number of benchmarks. The approximation is, however, incomplete. Consequently, planners employing the approximation are incomplete. To address this issue, (Son and Tu 2006) identifies conditions for the completeness of CPA and develops techniques to make CPA complete. These techniques require the system to deal with *sets* of approximated states, which—in the worst case—are the same as belief states. CPA represents this information as a formula in disjunctive normal form. The advantage of this approach is that the computation of the next state is still very simple. On the other hand, there are planning problems where the size of the disjunction can become exponential, preventing the planner from realistically start its search. To address this issue, the authors of the recent CPA developed preprocessing techniques which help reduce the size of the initial formula (Tran et al. 2009). These techniques enabled the planner to perform very well in several benchmarks and won the conformant planning category in the IPC-08 (`http://ippc-2008.loria.fr/wiki/index.php/Main_Page`).

The objective of this paper is to answer the question of whether disjunctive normal form representations, used to provide a *complete* representation of the belief state, can be competitively used in a best-first and progression-based search planner. The intuition is that disjunctive normal form representation can provide easy computation of the successor belief state, and they can be less demanding in terms of memory consumption. To this end, we develop an implementation of a *complete* transition function over belief states encoded using disjunctive normal form formulae, and empirically compare this alternative to the state-of-the-art in conformant planning. The experimental outcomes validate this option: the newly developed planner outperforms existing systems on a large variety of benchmarks.

Background: Conformant Planning

A *planning problem* is a tuple $P = \langle F, O, I, G\rangle$, where F is a set of propositions, O is a set of actions, I describes the initial state, and G describes the goal. A *literal* is either a proposition $p \in F$ or its negation $\neg p$. $\bar{\ell}$ denotes the complement of a literal ℓ—i.e., $\bar{\ell} = \neg\ell$, where $\neg\neg p{=}p$ for $p{\in}F$. For a set of literals L, $\overline{L} = \{\bar{\ell} \mid \ell \in L\}$. A conjunction of literals is often represented as the set of its conjuncts.

A set of literals X is *consistent* (resp. *complete*) if for every $p{\in}F$, $\{p, \neg p\}{\not\subseteq}X$ (resp. $\{p, \neg p\}{\cap}X{\neq}\emptyset$). A *state* is a consistent and complete set of literals. A *belief state* is a set of states. We will often use lowercase (resp. uppercase) letter to represent a state (resp. a belief state).

Each action a in O is associated with a precondition ϕ (denoted by $pre(a)$) and a set of conditional effects C_a of the form $\psi \rightarrow \ell$ (also written as $a : \psi \rightarrow \ell$), where ϕ and ψ are sets of literals and ℓ is a literal.

A state s satisfies a literal ℓ, denoted by $s \models \ell$, if $\ell \in s$. s satisfies a conjunction of literals X, denoted by $s \models X$, if it satisfies every literal belonging to X. The satisfaction of a formula in a state is defined in the usual way. Likewise, a belief state S satisfies a literal ℓ, denoted by $S \models \ell$, if $s \models \ell$ for every $s \in S$. S satisfies a conjunction of literals X, denoted by $S \models X$, if $s \models X$ for every $s \in S$.

Given a state s, an action a is *executable* in s if $s \models pre(a)$. The effect of executing a in s is

$$e(a, s) = \{\ell \mid \exists(a : \psi \rightarrow \ell).\ s \models \psi\}$$

The transition function, denoted by Φ, in the planning domain of P is defined by $\Phi(a, s) = s \setminus \overline{e(a, s)} \cup e(a, s)$ if $s \models pre(a)$; and $\Phi(a, s) = undefined$, otherwise.

We can extend the function Φ to define $\widehat{\Phi}$, a transition function which maps sequences of actions and belief states to belief states. $\widehat{\Phi}$ is used to reason about the effects of plans. Let S be a belief state. We say that an action a is executable in a belief state S if it is executable in every state belonging to S. Let $\alpha_n = [a_1, \ldots, a_n]$ be a sequence of actions:

- If $n = 0$ then $\widehat{\Phi}([\,], S) = S$;
- If $n > 0$ then
  - if $\widehat{\Phi}(\alpha_n, S)$ is undefined or a_n is not executable in $\widehat{\Phi}(\alpha_{n-1}, S)$, then $\widehat{\Phi}(\alpha_n, S)$ is undefined;
  - if $\widehat{\Phi}(\alpha_{n-1}, S)$ is defined and a_n is executable in $\widehat{\Phi}(\alpha_{n-1}, S)$ then $\widehat{\Phi}(\alpha_n, S){=}\{\Phi(a_n, s') | s' {\in} \widehat{\Phi}(\alpha_{n-1}, S)\}$ where $\alpha_{n-1} = [a_1, \ldots, a_{n-1}]$.

The initial state of the world I is a belief state and is represented by a formula. By S_I we denote the set of all states satisfying I. Typically, the goal description G can contain literals and `or`-statements (see below).

A sequence of actions $[a_1, \ldots, a_n]$ is a solution of P if $\widehat{\Phi}([a_1, \ldots, a_n], S_I)$ satisfies the goal G.

DNF Representation of Belief States

In this section we develop the theoretical underpinning of using DNF formulae to represent belief states. We will describe a progression function for computing the successor belief state given the current belief state is represented as a formula in disjunctive normal form. Let us begin with some motivations of the approach.

The specification of an initial state I is often given as a conjunction of literals C_1 and a set of statements C_2 of the form $\texttt{oneof}(\phi_1, \ldots, \phi_k)$ or $\texttt{or}(\phi_1, \ldots, \phi_k)$, where $\phi_1, \ldots, \phi_k$ are conjunctions of literals, and `oneof` and `or` is the exclusive-or and the logical-or operator, respectively.

For a specification I, the belief state S_I is uniquely determined and can be computed by (*i*) generating all states satisfying C_1; and (*ii*) selecting among those the states satisfying all statements in C_2. As we have observed in the introduction, the size of the initial belief state (in terms of its cardinality) can be exponential in the number of propositions of the domain. Worst, the process of creating the initial state can prevent the system to start the search. For example, given a domain with n propositions $f_1, \ldots, f_k$ and the specification $I = \{\texttt{oneof}(f_i, \neg f_i) \mid i = 1, \ldots, n\}$. This indicates that the initial belief state S_I consists of all possible states of the world and has the cardinality of 2^n. On the other hand, it is easy to see that I is logically equivalent to the empty set of states. As such, one might wonder whether S_I can be replaced by $\emptyset$. This view has been investigated in (Son et al. 2005), where approximation reasoning is defined

and a planner is discussed. The main disadvantage of this method is the incompleteness of the approximation. For a detailed discussion on this issue and a possible way to address it, the reader is referred to (Son and Tu 2006).

In this paper, we would like to investigate a middle ground of the two aforementioned extremes. On the one hand, we would like to represent the above initial state by $\emptyset$. On the other hand, we would like to maintain the completeness of the planner utilizing this representation. In order to achieve this goal, we define a notion of a *DNF-state* and define the progression function for computing successor DNF-states. We need the following terminologies and definitions.

A *partial state* is a consistent set of literals. A state s is a *completion* of a partial state δ if $\delta \subseteq s$. $ext(\delta)$, called the *extension* of δ, is the set of all completions of δ. Observe that if δ is a partial state then $\delta \equiv \bigvee\{s|s \in ext(\delta)\}$ and $\delta = \bigcap\{s|s \in ext(\delta)\}$.

Definition 1. *A* DNF-state *is a set of partial states. A DNF-state is* minimal *if it does not contain a pair of different partial states δ_1 and δ_2 such that $\delta_1 \subset \delta_2$.*

Let Δ be a DNF-state. We define $\min(\Delta) = \Delta \setminus \{\delta \mid \exists \delta' \in \Delta . \delta' \subset \delta\}$. It is easy to see that $\min(\Delta)$ is a minimal DNF-state equivalent to Δ. We call $ext(\Delta) = \bigcup\{ext(\delta)|\delta \in \Delta\}$ the completion of Δ. Obviously, the completion of Δ is a belief state which is equivalent to Δ.

Given a partial state δ, a literal ℓ is true (resp. false) in δ if $\ell \in \delta$ (resp. $\bar{\ell} \in \delta$). A literal ℓ is known in δ if it is true or false in δ. Otherwise, it is unknown. Abusing the notation, we will write $\delta \models \ell$ to indicate that ℓ is true in δ. For a set of literals γ, $\delta \models \gamma$ if $\delta \models \ell$ for every $\ell \in \gamma$. For a DNF-state Δ and a set of literals γ, $\Delta \models \gamma$ if $\delta \models \gamma$ for every $\delta \in \Delta$.

We will now define the progression function over DNF-states, which will be denoted by Φ_{DNF}. Given a DNF-state Δ and an action a, we need to define $\Phi_{DNF}(a, \Delta)$. Let us consider some examples that provide the motivation behinds the definition of Φ_{DNF}.

Example 1. Consider a domain with three propositions f, g, and h and the action a with one effect $a : \top \rightarrow f$ and the DNF-state $\Delta = \emptyset$. Intuitively, we would expect that the execution of a in Δ will yield the DNF-state $\{\{f\}\}$ as the execution of a in any state belonging to $ext(\Delta)$ will yield f while g and h retain their value. So, $\{\{f\}\}$ is equivalent to the belief state $\Phi(a, ext(\Delta))$.

Now consider an action b with the effect $b : f \rightarrow g$ and the DNF-state $\Delta = \{\{\neg f\}\}$. Here, the intuitive result of execution b in Δ would be Δ as the condition of the effect cannot be satisfied, i.e., executing b will not change anything.

Example 2. Consider a domain with three propositions f, g, and h and the action a with two effects $a : f \rightarrow h$ and $a : \neg f \rightarrow g$ and the DNF-state $\Delta = \emptyset$. Intuitively, we would expect that the execution of a in Δ will result in the DNF-state $\{\{f, h\}, \{\neg f, g\}\}$. This is because we know that Δ represents all possible states of the world and in any state, either f of $\neg f$ is true. As such, executing a will cause either $h \wedge f$ or $g \wedge \neg f$ to be true.

Now consider the same action a with the DNF-state $\Delta = \{\{f, \neg h\}, \{g\}\}$. Executing a in Δ should result in $\Delta' = \{\{f, h\}, \{g, f, h\}, \{g, \neg f\}\}$ which is equivalent to $\Delta' = \{\{f, h\}, \{g, \neg f\}\}$.

The examples show the two cases that we need to consider. In the first case, the conditional effects of the action is either satisfied or not by the DNF-state (Example 1). The progression function should proceed as if we have complete information. In the second case, the actions might have some effects whose condition is unknown in the given DNF-state (Example 2). In this case, the DNF-state might have to be unfolded into an equivalent DNF-state in which the conditions are known. To this end, we define the notion of partial extension of a partial state δ as follows.

Definition 2. *Let δ be a partial state and γ a consistent set of literals. The* partial extension of δ w.r.t. γ, *denoted by $\delta + \gamma$, is a set of partial states and is defined as*

$$\delta + \gamma = \begin{cases} \{\delta\} & \text{if } \gamma \subseteq \delta \text{ or } \overline{\gamma} \cap \delta \neq \emptyset \\ \{\delta \cup \gamma\} \cup \{\delta \cup \{\bar{l}\} | l \in \gamma \backslash \delta\} & \text{otherwise} \end{cases} \quad (1)$$

Intuitively, $\delta + \gamma$ is a DNF-state satisfying δ in which γ is known. In the first and second cases, γ is true or false in δ respectively. In the third case, γ is unknown and therefore, we need to extend δ with literals in δ so that the truth value of γ can be determined in every partial state in $\delta + \gamma$. For a DNF-state Δ, let $\Delta + \gamma = \min(\bigcup_{\delta \in \Delta}(\delta + \gamma))$. It holds that

Proposition 1. *Let δ (resp. Δ) be a partial state (resp. DNF-state). If γ is a consistent set of literals, then $\delta + \gamma$ (resp. $\Delta + \gamma$) is a minimal DNF-state which is equivalent to δ (resp. Δ). If γ_1 and γ_2 are two consistent sets of literals then $(\delta + \gamma_1) + \gamma_2 = (\delta + \gamma_2) + \gamma_1$.*

This proposition indicates that the order of extensions of a partial state δ with different sets of literals is irrelevant.

Definition 3. *Let a be an action with the set of conditional effects C_a. A partial state δ is called* enabling *for a if for every conditional effect $\psi \rightarrow \ell$ in C_a, either $\delta \models \psi$ or $\delta \models \neg\psi$ holds. A DNF-state Δ is* enabling *for a if every partial state in Δ is enabling for a.*

Example 3. Consider the DNF-state $\Delta = \{\{\neg f\}, \{f, \neg g\}\}$, the action a with the effect $a : f \rightarrow \neg g$, and the action b with two effects $b : f \rightarrow \neg f, g$ and $b : g \rightarrow f$. One can see that Δ is enabling for a since every partial state in Δ either entails f or $\neg f$. On the other hand, Δ is not enablingfor b, due to the fact that the truth value of g, the antecedent of the second effect of b, is unknown in the partial state $\{\neg f\}$ in Δ.

For an action a and a partial state δ, let $exp_a(\delta) = ((\delta + \psi_1) + \ldots) + \psi_k$ where $C_a = \{\psi_1 \rightarrow \ell_1, \ldots, \psi_k \rightarrow \ell_k\}$. For a DNF-state Δ, $exp_a(\Delta) = \bigcup_{\delta \in \Delta} exp_a(\delta)$.

Proposition 2. *For every partial state δ, DNF-state Δ, and action a:*

- *$exp_a(\delta)$ is a minimal DNF-state which is equivalent to δ and enabling for a.*
- *If Δ is a minimal DNF-state so $\exp_a(\Delta)$ is a minimal DNF-state which is equivalent to Δ and enabling for a.*

To define Φ_{DNF} we also need the following notation. For an action a and a partial state δ, the effect of a in δ, denoted $e(a, \delta)$, is defined as follows. $e(a, \delta) = \{l | \psi \rightarrow l \in C_a, \delta \models$

$\psi\}$. The result of execution a in δ is defined by $res(a,\delta) = \delta \backslash \overline{e(a,\delta)} \cup e(a,\delta)$. We are now ready to define the transition function Φ_{DNF} which maps pairs of actions DNF-states into DNF-states as follows.

Definition 4. *Let Δ be a DNF-state and a an action. The execution of a in Δ results in a DNF-state, denoted by $\Phi_{DNF}(a,\Delta)$, is defined as follows: $\Phi_{DNF}(a,\Delta) = \min(\{res(a,\delta') \mid \delta' \in exp_a(\Delta)\})$ if $\Delta \models pre(a)$ and $\Phi_{DNF}(a,\Delta) = undefined$ otherwise.*

Example 4. Consider the same DNF-state Δ and two actions a and b as given in Example 3. Assume that $pre(a) = pre(b) = \top$. One can check that: $\exp_a(\Delta) = \Delta$ and $\Phi_{DNF}(a,\Delta) = \Delta$. On the other hand, $\exp_b(\{\neg f\}) = \{\{\neg f, g\}, \{\neg f, \neg g\}\}$, $\exp_b(\{f, \neg g\}) = \{\{f, \neg g\}\}$, $\exp_b(\Delta) = \{\{\neg f, g\}, \{\neg f, \neg g\}, \{f, \neg g\}\}$, and $\Phi_{DNF}(b,\Delta) = \{\{f, g\}, \{\neg f, \neg g\}\, \{\neg f, g\}\}$.

By definition, for every action a and DNF-state, $\Phi_{DNF}(a,\Delta)$ is a minimal DNF-state. The function Φ_{DNF} is extended to allow for reasoning about the effects of an action sequence as follows. Let Δ be a DNF-state and $\alpha_n = [a_1, \ldots, a_n]$ an action sequence. We define

- $\widehat{\Phi_{DNF}}(\alpha_n,\Delta) = \Delta$ if $n = 0$;
- $\widehat{\Phi_{DNF}}(\alpha_n,\Delta) = \Phi_{DNF}(a_n, \widehat{\Phi_{DNF}}(\alpha_{n-1},\Delta))$, if $n > 0$, where $\alpha_{n-1} = [a_1, \ldots, a_{n-1}]$ and $\Phi_{DNF}(a, undefined) = undefined$ for every action a.

The next theorem shows that $\widehat{\Phi_{DNF}}$ is equivalent to the complete semantics defined by $\widehat{\Phi}$.

Theorem 1. *Let Δ be a DNF-state and $[a_1, \ldots, a_n]$ be an action sequence. Then,*

$$\widehat{\Phi_{DNF}}([a_1, \ldots, a_n], \Delta) \equiv \widehat{\Phi}([a_1, \ldots, a_n], ext(\Delta)).$$

This theorem allows us to compute solutions of conformant planning problems without the need to explicitly enumerate all possible states of the world when it is not necessary. In the next section, we describe the implementation of a planner employing this progression function.

Implementation

In this section, we detail the implementation of the conformant planner, called DNF, which uses the function Φ_{DNF} in its search for conformant plans. DNF is a heuristic best-first search, progression-based planner. As such, the overall structure of DNF is similar to that of any direct implementation of a heuristic best-first search, progression-based planner such as CFF, CPA, POND, etc. For completeness of the paper, we include the overall search algorithm implemented in DNF (Algorithm 1). From now on, for a planning problem $\langle F, O, I, G\rangle$, by S_I we denote a DNF-state satisfying I (the computation of S_I will be presented later). We will now provide the detail of how Φ_{DNF} is implemented in DNF.

Computing Successor DNF-States

Given a DNF-state Δ and an action a with the set of conditional effects C_a, we need to compute $\Delta' = \Phi_{DNF}(a,\Delta)$. By Definition 4, we have the following steps:

1. If $pre(a)$ is not true in Δ then Δ' is undefined.

Algorithm 1 Search(P, O, I, G)

```
1: Input: A planning problem ⟨P, O, I, G⟩
2: Output: Solution if exists; No solution otherwise
3: Create Priority Queue Q and initialize it with (S_I, [])
   {Initializing Search}
4: while Q is not empty do
5:    Let s = (Δ, CP) be the first element of Q
6:    if Δ satisfies G then
7:       return CP {the plan reaching Δ}
8:    else
9:       for each action a such that Δ ⊨ pre(a) do
10:         Compute Δ' = Φ_DNF(a, Δ)
11:         Insert (Δ', CP ∘ [a]) to Q
12:         {Heuristic of Δ' as Priority in Q}
13:      end for
14:   end if
15: end while
16: return no solution
```

2. If $pre(a)$ is true in Δ we need to compute the DNF-state $exp_a(\Delta)$ and compute the DNF-state $\min\{res(a,\delta') | \delta' \in exp_a(\Delta)\}$.

The computation of $exp_a(\Delta)$ is based on the computation of $exp_a(\delta)$ for a partial state δ, which in turn is based on the computation of $\delta + \gamma$. Algorithm 2 shows how $\delta + \gamma$ is computed.

Algorithm 2 Computing $\delta + \gamma$

```
1: Input: partial state δ, consistent set of literals γ
2: Output: δ + γ
3: if γ ⊆ δ ∨ γ̄ ∩ δ ≠ ∅ then
4:    return {δ}
5: else
6:    Let X = {δ ∪ γ}
7:    for each l ∈ γ \ δ do
8:       Set X = X ∪ {δ ∪ {l̄}}
9:    end for
10:   return X
11: end if
```

It is easy to see that Algorithm 2 computes $\delta + \gamma$, according to Def. 2. This is used in computing $exp_a(\delta)$ as follows.

Algorithm 3 Computing $\exp_a(\delta)$

```
1: Input: partial state δ, action a
2: Output: exp_a(δ)
3: Let C = C_a, X = {δ}
4: while C ≠ ∅ do
5:    Pick an effect ψ → ℓ in C
6:    Let Y = ∅
7:    for each δ' ∈ X do
8:       Y = Y ∪ (δ' + ψ)
9:    end for
10:   Remove from C all effects with antecedent ψ
11:   Set X = Y
12: end while
13: return min(X)
```

The correctness of Algorithm 3 is again easy to see. We are now ready to present the algorithm for computing Δ'.

Algorithm 4 Computing $\Phi_{DNF}(a, \Delta)$

1: **Input**: DNF-state Δ, action a
2: **Output**: $\Phi_{DNF}(a, \Delta)$
3: **if** $\Delta \not\models pre(a)$ **then**
4: **return** undefined
5: **else**
6: Compute $exp_a(\Delta)$ {Using Algorithm 3}
7: Compute $X = \{res(a, \delta) \mid \delta \in exp_a(\Delta)\}$
8: **end if**
9: **return** $\min(X)$

As with other algorithms, the correctness of Algorithm 4 is easy to verify. We will next discuss some properties related to the run-time of each of these algorithms. Let us assume that the domain has n propositions and the action a has k effects ($|C_a| = k$). We have that

Proposition 3. *Let δ be a partial state, Δ a DNF-state, γ a consistent set of literals, and a an action. We have that*

- *Computing $\delta + \gamma$ is $O(|\gamma| * log(n))$.*
- *Computing $\exp_a(\delta)$ is at most $O(\Pi_{\psi \to \ell \in C_a}(|\psi| * log(n)))$.*
- *Computing $\min(\Delta)$ can be done in $O(n * |\Delta|^2)$.*
- *For a DNF-state Δ which is enabling for a, computing $\Phi_{DNF}(a, \Delta)$ can be archived in $O(k * n * |\Delta|)$.*

These results show that computing $\Phi_{DNF}(a, \Delta)$ depends on the size of Δ and the cost of computing $exp_a(\Delta)$. Observe that the cost of computing $exp_a(\Delta)$ is the extra cost incurred due to the use of DNF-states to represent belief states. This extra cost, theoretically, depends on n, the size of C_a and its elements. It can therefore be quite significant when C_a contains a large number of effects with distinct antecedents. Fortunately, this situation does not often arise in practice. In most benchmarks, we observe that k is often less than 5.

Heuristics

Given a DNF-state Δ, DNF uses a heuristic that is a combination of the following three values:

- $h_{goal}(\Delta)$: the number of goals satisfied by Δ.
- $h_{card}(\Delta)$: the cardinality of Δ.
- $h_{dis}(\Delta)$: the *square distance* of Δ to the goal, defined by:
$$h_{dis}(\Delta) = \Sigma_{\delta \in \Delta}(|G| - h_{goal}(\delta))^2$$
where G is the goal of the problem.

The heuristic function used in DNF is defined by triples of the form $\langle h_{goal}(\Delta), h_{card}(\Delta), h_{dis}(\Delta)\rangle$ with the lexicographical order. While the first two values are inspired by CPA and other planners, the third value aimed at promoting DNF-states in which the satisfaction of goals among its partial states is uniform. We observe that DNF favors the number of satisfied goals value over the cardinality, while CPA prefers cardinality over the number of satisfied goals. The main reason for this decision lies in the fact that Φ_{DNF} maintains the minimality of the DNF-states during its execution.

Implementation Considerations

The implementation of the DNF system was developed by modifying the source code of the CPA system. This choice was motivated by the fact that both CPA and DNF rely on a representation of belief states using disjunctive normal form formulae. This also allows DNF to take advantages of the preprocessing techniques developed for the most recent CPA system (Tran et al. 2009). Those techniques include the forward/backward simplification, the `oneof` combination, and the *goal splitting* strategy. The `oneof` combination is aimed at reducing the size of the DNF form of the initial state, by combining separate `oneof` clauses. The *goal splitting* transformation is aimed at partitioning a planning problem into subproblems, each having only a subset of the final goal. For completeness, the DNF system applies only the first two of these transformations but not the last one (*goal splitting*)—which is, in general, incomplete. Those techniques are built in prolog inside the translator of the input theory. This translator reads the input in PDDL and write the output to a file in a particular format that CPA or DNF can read. Beside the advantage of reducing the size of the DNF form of the initial state, this process usually generates overhead of the run-time and it is significant for small or easy problems.

To improve DNF's performance, we implement some optimizations aimed at reducing the cost of computing the successor DNF-state. These include:

- considering effects of the form $\psi \to \varphi$ where φ is a set of literals. This is useful for two reasons: (*i*): it is commonly in PDDL used in the benchmarks; (*ii*) it removes the cost caused by the computation in `Line 10`, Algorithm 3.
- computing $e(a, \delta')$ for each $\delta' \in exp_a(\delta)$ during the computation of $exp_a(\delta)$. This is possible due to Definition 2. This is useful because it reduces the cost of computing $\Phi_{DNF}(a, \Delta)$ (`Line 7`, Algorithm 4).
- replacing "$Y = Y \cup (\delta' + \psi)$" (`Line 8`, Algorithm 3) with $Y = Y \cup \{\delta'\}$ if the head of all effects with the same antecedent ψ belongs to δ'. This does not change the correctness of Φ_{DNF} due to the observation that for every effect $\psi \to \ell$ in C_a and partial state δ', if $\ell \in \delta'$ then the truth value of ψ in δ' is unimportant. This is useful since it eliminates the computation of $\delta' + \psi$ when it is not needed.
- The computation of X (Algorithm 4) can be modified to obtain $\min(X)$ directly.
- computing S_I by converting I into DNF-state where (*i*) `oneof`-clause of the form $\texttt{oneof}(f, \neg f)$ is eliminated; (*ii*) computing the cross product, denoted by Z, of all remaining `oneof`-clauses; (*iii*) creating (a minimal) S_I from the set of facts, the set of `or`-clauses, and Z.

Experimental Evaluation

Planners

We compare the performance of DNF with four state-of-the-art conformant planners: CPA, CFF, POND, and `t0`. CPA is an approximation-based planner, and winner of the NOND track at IPC-08; we employed the most recent release of this planner.There are several available versions of POND which, in conjunction with different parameter options, return very different results. Nevertheless, no single version/parameter combination appears to dominate the others; in the experiments reported in this paper we decided to select POND version 1.1.1 with the default execution set-

tings. The CFF system was obtained from the public release at `members.deri.at/~joergh/cff.html`. The `t0` system has been provided by its authors—the version we used was obtained in April 2009.

The planners have been executed using the default settings. All the experiments have been performed using a dedicated Linux Intel Core 2 Dual 9400 2.66GHz workstation with 4GB of memory. The execution times reported here represent the average of two runs with minimal variance. In each experiment, we set a time-out limit of two hours.

Benchmarks

We have collected a number of benchmark problems from different domains. For ease of evaluation, the results are divided into benchmark suites, based on the source of the domains: IPC-08 domains, domains from previous IPCs, and challenging domains.

IPC-08 Domains: There are six domains used in IPC-08. The $adder$ domain is the synthesis of an adder boolean circuit. $dispose$ (denoted by *ds-*n*-*m in the table), is the problem of retrieving objects, whose initial locations are unknown, and placing them in a trash can; n represents the number of locations and m the number of objects. $forest$, $raokey$, and *uts-cycle* are variants of the grid problems.

Domains from Previous IPCs: This test-suit consists of seven domains. The domains $coins$, $comm$, $sortnet$, and *uts_k* are selected from IPC-06. The remaining domains, $logistics$, $ring$, and $safe$, come from the distribution of CFF and `t0`. Note that $logistics$ is the "incomplete version" of the well-known logistics domain.

Challenging Domains: The authors of `t0` proposed some challenging problems which are variants of the grid problem. Some of them have been used in IPC-08, e.g., $dispose$. 1_$dispose$, is a variant of $dispose$. $push$ is another variant of $dispose$, where objects can be picked up only at two designated locations, to which all objects have to be pushed to. $look\_n\_grab$ is about picking up objects that are sufficiently close, and dumping them in the trashcan before continuing.

Experiments and Results

In our experimental study, we are interested in comparing different planners, according to *performance*—i.e., speed of computation—and *scalability*—ability to solve increasingly larger problem instances. The execution times reported in this section are expressed in seconds. For each experiment, we report in the tables the execution time and the length of the plan found. The execution time reported for each experiment represents the total execution time; for DNF, CPA, and `t0` the execution time includes both the translation time as well as the time to search for a plan. In the result tables, **OM** denotes out-of-memory, **NA** denotes non-applicable benchmark, **F** means failure during execution of `t0` probably due to out-of-memory, **TO** denotes time-out.

IPC-08 Domains: The results for the benchmarks from IPC-08 are reported in Table 1; `ds` represents the dispose problem, `raok` represents the Rao's key problem, `uts-c` represents the uts-cycle problem, and `uts-fc` represents the uts-full-connected problem. The results in Table 1 show

Problem	DNF	CPA(C)	T0	CFF	POND
adder-1	**0.323/3**	11.19/3	TO	NA	TO
adder-2	OM	OM	TO	NA	TO
block-1	0.91/7	1.3/7	0.04/5	0.02/6	**0.01/6**
block-2	0.94/38	1.55/41	0.2/23	TO	**0.06/34**
block-3	210/331	OM	16.7/83	TO	**3.96/80**
block-4	OM	OM	F	TO	TO
ds-4-3	1.58/185	4.9/288	**0.22/122**	0.62/73	219.7/98
ds-4-4	1.73/187	7.66/421	**0.39/120**	1.342/90	TO
ds-4-5	1.89/180	11.42/554	**0.77/145**	2.55/107	TO
ds-8-3	**36.2/447**	782/2462	135/761	TO	TO
ds-8-4	**51.9/733**	1367/3596	750.4/835	TO	TO
ds-8-5	**63.3/878**	2152/4730	F	TO	TO
ds-10-5	**270/1286**	TO	F	TO	TO
forest-2	3.2/55	31.87/35	0.12/16	**0.11/18**	1.34/18
forest-3	1058/16343	TO	**0.66/45**	TO	386.8/65
forest-4	871.9/8207	OM	**0.988/78**	TO	OM
forest-5	OM	OM	**2.88/129**	TO	OM
raok-2	0.56/39	1/32	0.12/21	**0.07/34**	F
raok-3	**1.68/153**	2.72/21	F	11.8/102	F
raok-4	TO	TO	F	TO	F
uts-c-3	1.18/3	2.31/3	**0.12/3**	NA	no sol.
uts-c-4	1.29/6	17.67/6	**0.42/7**	NA	no sol.
uts-c-5	**1.37/10**	OM	1.92/10	NA	no sol.
uts-c-6	**1.43/17**	OM	9.46/17	NA	no sol.
uts-c-7	**2.82/32**	OM	62.8/26	NA	no sol.
uts-c-8	**8.86/41**	OM	F	NA	no sol
uts-fc-4	OM	OM	F	NA	no sol.

Table 1: Execution times and plan lengths, IPC-08

that DNF outperforms the other planners in four out of six domains ($adder$, ds, $raokey$, and $uts\text{-}cycle$). Furthermore, the results show that DNF has a higher level of scalability—being able to solve several instances that are not solvable by the other planners.

Observe that, CPA outperforms `t0` for $adder$, $dispose$, and $raokey$ while `t0` is better than CPA in the other three domains. Observe also that no planner is able to solve even the smallest instance of the $uts\text{-}full\text{-}connected$ domain.

`t0` outperforms DNF for two smallest problems in the $uts\text{-}cycle$ domain, but it does not scale as well as DNF for larger instances (in $block$, ds, $raokey$, and $uts\text{-}cycle$). `t0` is the best planner in the $forest$ domain, and the lengths of the plans found are significantly shorter than those found by DNF. We believe that this is due to the effectiveness of the heuristic function used by FF in this domain. In contrast, the heuristic function used by DNF, which relies heavily on the number of satisfied goals, is not useful in this domain, since the number of goals in each problem in this domain is only two.

The experiments for POND show an inconsistent behavior. The system largely outperforms the other planners in the $block$ domain, but it leads to time out or out of memory situations for the majority of the other domains. The plans returned by POND for $raokey$ appear to be incorrect (they are plans of length 1)—this has been denoted by F in the table. For the $uts\text{-}cycle$ domain, POND incorrectly reports that there are no solutions (while solutions actually exist).

CFF is unable to handle problems that have disjunctive goal clauses or conjunctions within `oneof` clauses.

Domains from previous IPCs: Table 2 reports the exper-

Problem	DNF	CPA(C)	T0	CFF	POND
coins-10	0.91/27	1.79/67	**0.036/26**	0.12/38	0.52/146
coins-15	1.41/67	7/362	**0.12/79**	2.64/89	10.2/124
coins-20	4.17/99	17.8/472	**0.14/107**	16.3/143	104/153
coins-21	OM	OM	F	TO	TO
comm-15	3.57/125	5.93/95	**0.11/110**	0.2/95	11.5/98
comm-20	117/296	174/239	**0.51/278**	4.99/239	OM
comm-25	1246/501	1767/389	**1.53/453**	39.8/389	OM
lg-4-2-2	1.92/41	2.3/41	0.036/19	**0.01/19**	0.26/21
lg-4-2-4	2.8/123	3.39/81	**0.05/40**	0.06/40	1.86/45
lg-4-3-2	3.72/70	3.8/39	0.06/24	**0.03/23**	2.02/26
lg-4-3-3	5.68/160	7.9/221	0.08/35	**0.04/37**	22.5/37
ring-2	0.85/7	1.41/8	**0.01/5**	0.01/7	0/6
ring-3	0.99/11	1.63/12	**0.015/8**	0.11/15	0.02/13
ring-4	1.02/15	1.77/17	**0.02/13**	1.45/26	0.13/16
ring-5	1.68/19	2.46/21	**0.024/17**	25.4/45	3.06/20
safe-10	0.87/10	0.88/10	**0.016/10**	0.02/10	no sol.
safe-30	1.05/30	2.85/30	**0.072/30**	1.24/30	no sol.
safe-50	1.49/50	20.6/50	**0.232/50**	27.17/50	no sol.
safe-70	2.48/70	90.8/70	**0.488/70**	113.6/70	no sol.
sort-05	0.93/15	0.94/12	0.196/15	NA	**0/12**
sort-10	1.85/54	3.18/39	OM	NA	**0.03/38**
sort-13	8.92/90	39.7/41	F	NA	**0.08/55**
sort-15	35.8/118	243/65	F	NA	**0.14/65**
uts-k-10	2.51/66	20.1/80	**0.89/59**	14.55/58	18.6/68
uts-k-20	22.4/136	1423/197	**11.6/119**	1558/118	OM
uts-k-30	114/206	TO	**59.8/179**	TO	OM
uts-k-40	344/276	TO	**224/239**	TO	OM
uts-k-50	846/346	TO	**592/299**	TO	OM
uts-k-55	**1265/381**	TO	F	TO	OM

Table 2: Execution times from previous IPCs

imental results from several domains from previous IPCs. lg represents the logistics domain and $sort$ is the $sortnet$ domain. The table indicates that these domains seem to be easy for most of the conformant planners. In this test-suite, `t0` is the best at five out of seven domains. On the other hand, `t0` does not succeed in $sortnet$; DNF provides greatest scalability on the *uts-k* domain. Although DNF is not the best in many of these domains, it is the only one which can solve all the problems on the table with a reasonable execution time. The exception is the $comm$ domain where DNF does not scale well. We observe that in most problems of this domain, the time spent by the pre-processor of DNF is significant more than the time spent by DNF to search for solution. For example, DNF requires only 0.96 seconds and 3.378 seconds to find a plan for *comm-20* and *comm-25* after completion of the translation.

Challenging Domains: Table 3 presents the execution times from the challenging domain; in the table, $1d$ denotes the 1-dispose domain, lng denotes the look-and-grab domain. `t0` can only solve a few of these problem instances, mostly the smaller size ones. Note that the set of actions in these domains is so complicated. We suspect that a heuristic heavily dependent on the distance in the graph plan does not work well in these domains. As such, `t0`, CFF, and POND do not perform well in this test suite. CPA can solve most of the problems, but is significantly slower than DNF. DNF proves its absolute advantage in these difficult domains where the translation time is negligible compared to the search time. We also observe that, in these domains, the lengths of plans

Problem	DNF	CPA(C)	T0	CFF	POND
1d-2-5	1.57/14	**1.45/12**	F	TO	46/14
1d-2-7	19.4/14	**14.3/12**	F	TO	OM
1d-2-9	**451/14**	OM	F	TO	OM
1d-5-1	**1.85/98**	2.29/98	F	TO	OM
1d-5-3	**127/122**	264/102	F	TO	OM
1d-5-4	**1732/536**	OM	F	TO	OM
1d-10-1	**151/406**	172/568	F	TO	OM
1d-10-2	**568/536**	OM	F	TO	OM
lng-5-1-1	2.14/35	2.19/32	**0.91/31**	414/133	OM
lng-5-1-2	**2.77/30**	3.52/43	F	TO	OM
lng-8-1-1	**27.4/99**	28.7/94	114/415	TO	OM
lng-8-2-2	**51.4/73**	60.5/98	F	TO	OM
lng-8-3-3	**449/32**	OM	F	TO	OM
lng-10-1-1	**157/157**	160/204	F	TO	OM
lng-10-1-2	**301/225**	OM	F	TO	OM
push-5-1	1.89/64	2.35/113	**0.49/107**	13.8/82	TO
push-5-3	**6.6/523**	2546/3282	161/251	513/105	TO
push-5-5	**14/631**	TO	451/213	TO	TO
push-9-1	**68/205**	75/253	590/652	TO	TO
push-9-2	**90/998**	OM	F	TO	TO
push-9-3	**157/2225**	OM	F	TO	TO
push-10-4	**640/4387**	OM	F	TO	TO

Table 3: Execution times for challenging domains

found by DNF are usually shorter than the lengths of plans obtained by other planners.

Considerations

We will now discuss in more details the difference between DNF and `t0` and between DNF and CPA.

DNF **vs.** `t0`**:** `t0` often outperforms DNF in small or easy instances, especially those from Table 2. On the other hand, DNF provides better scalability to large or challenging instances. Both planners apply some preprocessing techniques to the original problem specification before starting the search for a plan. In `t0`, a conformant problem is translated to an equivalent classical problem and FF is used to compute a solution of this problem, which will be finally converted to a solution of the original problem. DNF, on the other hand, employs the preprocessing techniques of CPA to simplify the problem before the search starts. As it turns out, the preprocessing techniques do not always pay off. In some instances, the preprocessing overhead is significantly larger than the time used by DNF to search for a solution. This problem does not seem to occur in `t0`. Table 4 illustrates this observation in selected small instances, where DNF's preprocessing time is definitely larger than the time needed to find a solution.

On the other hand, for larger or challenging instances, the overhead does not seem to play a significant role in the total time of DNF while the searching time still remains small. This explains the fact that DNF can scale up better than `t0`.

DNF **vs.** CPA**:** DNF was developed using the source code of CPA. Nevertheless, the two systems are radically different. CPA relies on an approximation for progression while DNF uses complete reasoning. Even though both use sets of partial states to represent belief states. The computation of the successor belief state of these planners is different. For example, executing action a in Example 2 in $\Delta = \{\emptyset\}$ results

Problem	DNF	T0	Problem	DNF	T0
block-1	0.006/0.904	0.04	lg-4-2-4	0.41/2.39	0.05
block-2	0.033/0.907	0.2	ring-2	0.006/0.844	0.01
uts-c-3	0.009/1.171	0.12	ring-3	0.032/0.958	0.015
uts-c-4	0.031/1.259	0.42	safe-10	0.009/0.861	0.016
coins-10	0.018/0.952	0.036	safe-30	0.125/0.925	0.072
coins-15	0.08/1.33	0.12	sort-05	0.09/0.84	0.196
comm-15	0.11/1.3	0.11	lng-5-1-1	0.11/2.03	0.91
lg-4-2-2	0.06/1.86	0.036	push-5-1	0.079/1.811	0.49

Table 4: DNF search time/translation time vs. `t0` exec. time in Δ in CPA, while it results in $\{\{f, h\}, \{\neg f, g\}\}$ in DNF.

Another difference lies in the representation of the initial state employed by these planners. Given an incomplete representation I of the initial state, CPA computes an approximation I' of I which guarantees its completeness, and uses I' as the initial state for its search. This may cause, in some problems, the size of I' to grow significantly w.r.t. the size of I, e.g., 65536 vs. 0 for the $sortnet\_15$ problem. Observe that the computation of I' from I can be costly, especially when the size of I and the problem size (i.e., the number of literals in the problem and number of conditional effects). These observations are confirmed by the experimental results presented above. In addition, DNF can operate with any DNF representation which is logically equivalent to I (and still preserve soundness and completeness). This opens the doors to the use of preprocessing to simplify the initial DNF states and encode it using a compact DNF representation.

Since DNF can start searching with any equivalent DNF form of the initial state, one may consider using *prime implicants* as the representation of formula in this case. Nevertheless, we chose the minimal DNF representation for two reasons: (*i*) its computation is simple and the difference in size of both forms is not significant given the de-facto representation of initial state in the considered benchmarks; (*ii*) the computation of the successor belief state appears to be faster. The conversion of a DNF formula to prime implicants form is expensive and it may generate overhead of computation of the progression function. For example, let us consider an initial DNF-state $\Delta = \{\{f, g\}, \{\neg f, g\}, \{\neg f, \neg g\}\}$ and an action a with the effect $a : g \rightarrow f$. Observe that Δ is enabling for a. If the prime implicants form is used, Δ is first converted to $\{\{f, g\}, \{\neg f\}\}$, which is not enabling for a. As such, some overhead might be needed (e.g., computing partial extensions) if prime implicant form is used.

Conclusion and Future Work

In this paper, we presented the design and implementation of a best-first search, progression-based conformant planner, which relies on a complete representation of belief states using disjunctive normal form formulate. We proposed a formalization of the representation and of the corresponding progression function, and demonstrated its integration in a best-first search planner. The planner has been experimentally evaluated on several sets of benchmarks—with performance results that are superior to those provided by other state-of-the-art conformant planners. These results indicate that DNF representations have the potential of providing fast computation of successor belief states, without producing excessive memory consumption—this leads to faster computations and enhanced scalability.

Although the DNF representation appears to provide efficiency and some level of scalability, there are still a number of domains that are beyond the capabilities of our system (as well as several of the other conformant planners). This is the case, for example, of domains like *uts-full-connected* from the IPC-08 competition—which leads to extremely large disjunctive normal form formulae—and the $adder$—which contains an extremely large number of actions. Our future work will focus on how to modify our approach to effectively cope with these domains.

Acknowledgments

We would like to thank the authors of the planners CPA and `t0` for providing us access to their systems. The research has been supported by NSF grants 0812267 and 0420407.

References

C. Baral, V. Kreinovich, and R. Trejo. Computational complexity of planning and approximate planning in the presence of incompleteness. *AIJ*, 122:241–267, 2000.

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

R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

D. Bryce et al. Planning Graph Heuristics for Belief Space Search. *JAIR*, 26:35–99, 2006.

A. Cimatti et al. Conformant Planning via Symbolic Model Checking and Heuristic Search. *AIJ*, 159:127–206, 2004.

M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. The Planning Domain Definition Language. Version 1.2. Technical Report TR98003/DCS TR1165, Yale, 1998.

H. Palacios and H. Geffner. From Conformant into Classical Planning: Efficient Translations that may be Complete Too. In *ICAPS*, 2007.

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

T.C. Son and C. Baral. Formalizing sensing actions — a transition function based approach. *Artificial Intelligence*, 125(1-2):19–91, January 2001.

T.C. Son and P.H. Tu. On the Completeness of Approximation Based Reasoning and Planning in Action Theories with Incomplete Information. In *KRR*, 481–491, 2006.

T.C. Son, P.H. Tu, M. Gelfond, and R. Morales. Conformant Planning for Domains with Constraints—New Approach. *AAAI*, pp. 1211–1216, 2005.

D-V. Tran, H-K. Nguyen, E. Pontelli, and T.C. Son. Improving performance of conformant planners. In *PADL*, pp. 239–253. Springer, 2009.