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., onef 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 trade-off 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 prepro- cessing 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 \( \neg p \). \( \ell \) denotes the comple ment of a literal \( \ell \)—i.e., \( \ell = \neg \ell \), where \( \neg p = p \) for \( p \in F \). For every set of literals \( L, L' = \{ \ell | \ell \in L \} \). 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 \( p \in F \), \( \{ p, \neg p \} \subseteq X \) (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. belief state).

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

A state \( s \) satisfies a literal \( \ell \), 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 \( \ell \), 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 \( s \models \text{pre}(a) \). The effect of executing \( a \) in \( s \) is \( e(a, s) = \{ \ell | \exists ! (a : \psi \rightarrow \ell), s \models \psi \} \).

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

We can extend the function \( \Phi \) to define \( \hat{\Phi} \), a transition function which maps sequences of actions and belief states to belief states. \( \hat{\Phi} \) 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 \( \hat{\Phi}([], S) = S \);
- If \( n > 0 \) then
  - \( \hat{\Phi}(\alpha_{n-1}, S) \) is undefined or \( a_n \) is not executable in \( \hat{\Phi}(\alpha_{n-1}, S) \), then \( \hat{\Phi}(\alpha_n, S) \) is undefined;
  - \( \hat{\Phi}(\alpha_{n-1}, S) \) is defined and \( a_n \) is executable in \( \hat{\Phi}(\alpha_{n-1}, S) \) then \( \hat{\Phi}(\alpha_n, S) = \hat{\Phi}(\alpha_{n-1}, S) | s' \in \hat{\Phi}(\alpha_{n-1}, S) \) 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 \( S_f \) 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 \( \hat{\Phi}([a_1, \ldots, a_n], S_f) \) 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 oneof \((\phi_1, \ldots, \phi_k)\) or 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 \( S_f \) 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 \( I = \{ \text{oneof}(f_i, \neg f_i) | i = 1, \ldots, n \} \). This indicates that the initial belief state \( S_f \) 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 \( S_f \) 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 \( \delta \) if \( \delta \subseteq s \). \( \text{ext}(\delta) \), called the extension of \( \delta \), is the set of all completions of \( \delta \). Observe that if \( \delta \) is a partial state then \( \delta \equiv \bigvee \{ s | s \in \text{ext}(\delta) \} \) and \( \delta = \bigcap \{ s | s \in \text{ext}(\delta) \} \).

**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 \( \delta_1 \) and \( \delta_2 \) such that \( \delta_1 \subseteq \delta_2 \).

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

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

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

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

Now consider an action \( b \) with the effect \( b : f \rightarrow g \) and the DNF-state \( \Delta = \{ \neg f \} \). Here, the intuitive result of executing \( b \) in \( \Delta \) would be \( \Delta \) 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 \( \Delta \) will result in the DNF-state \( \{f, h\}, \{\neg f, g\} \). This is because we know that \( \Delta \) 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 \( \Delta \) 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 \( \delta \) as follows.

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

\[
\delta + \gamma = \begin{cases} 
\{ \delta \} & \text{if } \gamma \subseteq \delta \text{ or } \gamma \cap \delta \neq \emptyset \\
\delta \cup \gamma & \text{otherwise}
\end{cases}
\]

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

**Proposition 1.** Let \( \delta \) (resp. \( \Delta \)) be a partial state (resp. DNF-state). If \( \gamma \) is a consistent set of literals, then \( \delta + \gamma \) (resp. \( \Delta + \gamma \)) is a minimal DNF-state which is equivalent to \( \delta \) (resp. \( \Delta \)). If \( \gamma_1 \) and \( \gamma_2 \) 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 \( \delta \) 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 \( \delta \) 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 \( \Delta \) is enabling for \( a \) if every partial state in \( \Delta \) 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 \( b : f \rightarrow \neg g, g \rightarrow f \). One can see that \( \Delta \) is enabling for \( a \) since every partial state in \( \Delta \) either entails \( f \) or \( \neg f \). On the other hand, \( \Delta \) is not enabling for \( b \), 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 \( \Delta \).

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

**Proposition 2.** For every partial state \( \delta \), DNF-state \( \Delta \), and action \( a \):

- \( \exp_a(\delta) \) is a minimal DNF-state which is equivalent to \( \delta \) and enabling for \( a \).

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

To define \( \Phi_{\Delta, a} \) we also need the following notation. For an action \( a \) and a partial state \( \delta \), the effect of \( a \) in \( \delta \), denoted \( e(a, \delta) \), is defined as follows. \( e(a, \delta) = \{ l : \psi \rightarrow \ell \in C_a, \delta \models \psi \} \).
ψ). The result of execution $a$ in $\delta$ is defined by $res(a, \delta) = \delta \setminus e(a, \delta) \cup e(a, \delta)$. We are now ready to define the transition function $\Phi_{DNF}$ which maps pairs of actions DNF-states into DNF-states as follows.

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

**Example 4.** Consider the same DNF-state $\Delta$ and two actions $a$ and $b$ as given in Example 3. Assume that $\text{pre}(a) = \text{pre}(b) = \emptyset$. One can check that: $\exp_a(\Delta) = \emptyset$ and $\Phi_{DNF}(a, \Delta) = \Delta$. On the other hand, $\exp_b(\Delta) = \{\neg f, \neg g\}$, $\exp_a(\neg f, \neg g) = \{\neg f, \neg g\}$, $\exp_b(\Delta) = \{\neg f, \neg g\}$, and $\Phi_{DNF}(b, \Delta) = \{\neg f, \neg g\}$. The next theorem shows that $\Phi_{DNF}$ is equivalent to the complete semantics defined by $\hat{\Phi}$.

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

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

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 $\Phi_{DNF}$ in its search for conformant plans. DNF is a heuristic best-first search, progressions-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 $\Phi_{DNF}$ is implemented in DNF.

**Computing Successor DNF-States**

Given a DNF-state $\Delta$ 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 $\text{pre}(a)$ is not true in $\Delta$ then $\Delta'$ is undefined.

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

1: **Input:** A planning problem $\langle P, O, I, G \rangle$
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 $s = (\Delta, CP)$ be the first element of $Q$
6: **if** $\Delta$ satisfies $G$ **then**
7: **return** $CP$ {the plan reaching $\Delta$}
8: **else**
9: **for** each action $a$ such that $\Delta \models \text{pre}(a)$ **do**
10: Compute $\Delta' = \Phi_{DNF}(a, \Delta)$
11: Insert $(\Delta', CP \circ [a])$ to $Q$
12: {Heuristic of $\Delta'$ as Priority in $Q$}
13: **end for**
14: **end if**
15: **end while**
16: **return** no solution

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

The computation of $\exp_a(\Delta)$ is based on the computation of $\exp_a(\delta)$ for a partial state $\delta$, 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 $\delta$, consistent set of literals $\gamma$
2: **Output:** $\delta + \gamma$
3: if $\gamma \subseteq \delta \lor \pi \land \delta \neq \emptyset$ then
4: **return** $\{\delta\}$
5: **else**
6: Let $X = \{\delta \cup \gamma\}$
7: **for** each $l \in \gamma \setminus \delta$ **do**
8: Set $X = X \cup \{\delta \cup \{l\}\}$
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 $\delta$, action $a$
2: **Output:** $\exp_a(\delta)$
3: Let $C = C_a$, $X = \{\delta\}$
4: **while** $C \neq \emptyset$ **do**
5: Pick an effect $\psi \rightarrow l$ in $C$
6: Let $Y = \emptyset$
7: **for** each $\delta' \in X$ **do**
8: $Y = Y \cup (\delta' + \psi)$
9: **end for**
10: Remove from $C$ all effects with antecedent $\psi$
11: Set $X = Y$
12: **end while**
13: **return** $\text{min}(X)$

The correctness of Algorithm 3 is again easy to see. We are now ready to present the algorithm for computing $\Delta'$. 
due to the use of DNF-states to represent belief states. This
most benchmarks, we observe that
the size of
\(\Delta\)
approaches infinity. We observe that D
changes during its exe-
The main reason for this decision lies in 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 one-of combination, and
the goal splitting strategy. The one-of combination is aimed at reducing the size of the DNF form of the initial state, by
combining separate one-of 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 trans-
lator 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 opti-
mizations aimed at reducing the cost of computing the suc-
cessor DNF-state. These include:
• considering effects of the form \(\psi \rightarrow \varphi\) where \(\varphi\) 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 \(\exp(a, \delta')\) 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
  \(\PhiDNF(a, \Delta)\) (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 \(\psi\) belongs to \(\delta'\). This does not change
  the correctness of \(\PhiDNF\) due to the observation that for
  every effect \(\psi \rightarrow \ell\) in \(C_a\) and partial state \(\delta'\), if \(\ell \in \delta'\)
  then the truth value of \(\psi\) in \(\delta'\) 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 \(S_f\) by converting \(I\) into DNF-state where (i)
  one-of-clause of the form oneof(\(f, \neg f\)) is eliminated; (ii)
  computing the cross product, denoted by \(Z\), of all remain-
  ing oneof-clauses; (iii) creating (a minimal) \(S_f\) 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 t.o.
CPA is an approximation-based planner, and winner of the
NOND track at IPC-08; we employed the most recent re-
lease of this planner. There are several available versions of
POND which, in conjunction with different parameter op-
tions, return very different results. Nevertheless, no single
version/parameter combination appears to dominate the oth-
ers; in the experiments reported in this paper we decided to
select POND version 1.1.1 with the default execution set-
tions. The CFF system was obtained from the public release at members.deri.at/~jjoerg/cff.html. The FF 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-suite 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 trash can 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 plans 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 due to 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, raokey 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

<table>
<thead>
<tr>
<th>Problem</th>
<th>DNF</th>
<th>CPA(C)</th>
<th>T0</th>
<th>CFF</th>
<th>POND</th>
</tr>
</thead>
<tbody>
<tr>
<td>adder-1</td>
<td>0.323/3</td>
<td>11.19/3</td>
<td>TO</td>
<td>NA</td>
<td>TO</td>
</tr>
<tr>
<td>adder-2</td>
<td>OM</td>
<td>OM</td>
<td>TO</td>
<td>NA</td>
<td>TO</td>
</tr>
<tr>
<td>block-1</td>
<td>0.91/7</td>
<td>1.3/7</td>
<td>0.04/5</td>
<td>0.02/6</td>
<td>0.01/6</td>
</tr>
<tr>
<td>block-2</td>
<td>0.94/38</td>
<td>1.55/41</td>
<td>0.2/23</td>
<td>TO</td>
<td>0.06/34</td>
</tr>
<tr>
<td>block-3</td>
<td>210/331</td>
<td>OM</td>
<td>16.78/3</td>
<td>TO</td>
<td>3.96/80</td>
</tr>
<tr>
<td>block-4</td>
<td>OM</td>
<td>OM</td>
<td>F</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>ds-4-3</td>
<td>1.58/185</td>
<td>4.92/88</td>
<td>0.22/122</td>
<td>0.62/73</td>
<td>219.798</td>
</tr>
<tr>
<td>ds-4-4</td>
<td>1.73/187</td>
<td>7.66/421</td>
<td>0.39/120</td>
<td>1.34/90</td>
<td>TO</td>
</tr>
<tr>
<td>ds-4-5</td>
<td>1.89/180</td>
<td>11.42/554</td>
<td>0.77/145</td>
<td>2.55/107</td>
<td>TO</td>
</tr>
<tr>
<td>ds-8-3</td>
<td>36.2/447</td>
<td>782/2462</td>
<td>135/761</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>ds-8-4</td>
<td>51.9/733</td>
<td>1367/5996</td>
<td>750/4835</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>ds-8-5</td>
<td>63.3/878</td>
<td>2152/4730</td>
<td>F</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>ds-10-5</td>
<td>270/1026</td>
<td>TO</td>
<td>F</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>forest-2</td>
<td>3.2/55</td>
<td>31.8/35</td>
<td>0.12/16</td>
<td>0.11/18</td>
<td>1.34/18</td>
</tr>
<tr>
<td>forest-3</td>
<td>1058/16343</td>
<td>TO</td>
<td>0.66/45</td>
<td>TO</td>
<td>386.865</td>
</tr>
<tr>
<td>forest-4</td>
<td>871.9/8207</td>
<td>OM</td>
<td>0.988/78</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>forest-5</td>
<td>OM</td>
<td>OM</td>
<td>2.88/129</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>raokey-2</td>
<td>0.56/39</td>
<td>1/32</td>
<td>0.12/21</td>
<td>0.07/34</td>
<td>F</td>
</tr>
<tr>
<td>raokey-3</td>
<td>1.68/153</td>
<td>2.72/21</td>
<td>F</td>
<td>11.8/102</td>
<td>F</td>
</tr>
<tr>
<td>raokey-4</td>
<td>TO</td>
<td>TO</td>
<td>F</td>
<td>TO</td>
<td>F</td>
</tr>
<tr>
<td>uts-c-3</td>
<td>1.18/3</td>
<td>2.31/3</td>
<td>0.12/3</td>
<td>NA</td>
<td>no sol.</td>
</tr>
<tr>
<td>uts-c-4</td>
<td>1.29/6</td>
<td>17.67/6</td>
<td>0.42/7</td>
<td>NA</td>
<td>no sol.</td>
</tr>
<tr>
<td>uts-c-5</td>
<td>1.37/10</td>
<td>OM</td>
<td>1.92/10</td>
<td>NA</td>
<td>no sol.</td>
</tr>
<tr>
<td>uts-c-6</td>
<td>1.43/17</td>
<td>OM</td>
<td>9.46/17</td>
<td>NA</td>
<td>no sol.</td>
</tr>
<tr>
<td>uts-c-7</td>
<td>2.82/32</td>
<td>OM</td>
<td>62.8/26</td>
<td>NA</td>
<td>no sol.</td>
</tr>
<tr>
<td>uts-c-8</td>
<td>8.86/41</td>
<td>OM</td>
<td>F</td>
<td>NA</td>
<td>no sol.</td>
</tr>
<tr>
<td>uts-fc-4</td>
<td>OM</td>
<td>OM</td>
<td>F</td>
<td>NA</td>
<td>no sol.</td>
</tr>
</tbody>
</table>

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-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-full-connected domain.

t0 outperforms DNF for two smallest problems in the uts-cycle domain, but it does not scale as well as DNF for larger instances (in block, ds, raokey, and uts-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-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 one or clauses.

Domains from previous IPCs: Table 2 reports the exper-
Table 2: Execution times from previous IPCs.

<table>
<thead>
<tr>
<th>Problem</th>
<th>DNF</th>
<th>CPA(C)</th>
<th>T0</th>
<th>CFF</th>
<th>POND</th>
</tr>
</thead>
<tbody>
<tr>
<td>coins-10</td>
<td>0.91/27</td>
<td>1.79/67</td>
<td>0.036/26</td>
<td>0.12/38</td>
<td>0.52/146</td>
</tr>
<tr>
<td>coins-15</td>
<td>1.41/67</td>
<td>7/362</td>
<td>0.12/79</td>
<td>2.64/89</td>
<td>10.2/124</td>
</tr>
<tr>
<td>coins-20</td>
<td>4.17/99</td>
<td>17.8/472</td>
<td>0.14/107</td>
<td>16.3/143</td>
<td>104/153</td>
</tr>
<tr>
<td>coins-21</td>
<td>OM</td>
<td>OM</td>
<td>F</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>comm-15</td>
<td>3.57/125</td>
<td>5.93/95</td>
<td>0.11/110</td>
<td>0.29/52</td>
<td>11.5/98</td>
</tr>
<tr>
<td>comm-20</td>
<td>117/296</td>
<td>17/429</td>
<td>0.51/278</td>
<td>4.99/239</td>
<td>OM</td>
</tr>
<tr>
<td>comm-25</td>
<td>1264/501</td>
<td>1767/389</td>
<td>1.53/453</td>
<td>38.9/389</td>
<td>OM</td>
</tr>
<tr>
<td>lg-2-2</td>
<td>1.92/41</td>
<td>2.3/41</td>
<td>0.036/19</td>
<td>0.01/19</td>
<td>0.26/21</td>
</tr>
<tr>
<td>lg-2-4</td>
<td>2.81/123</td>
<td>3.39/81</td>
<td>0.05/40</td>
<td>0.06/40</td>
<td>1.86/45</td>
</tr>
<tr>
<td>lg-3-2</td>
<td>3.72/70</td>
<td>3.8/39</td>
<td>0.06/24</td>
<td>0.03/23</td>
<td>2.02/26</td>
</tr>
<tr>
<td>lg-3-3</td>
<td>5.68/160</td>
<td>7.92/21</td>
<td>0.08/35</td>
<td>0.04/37</td>
<td>22.5/37</td>
</tr>
<tr>
<td>ring-2</td>
<td>0.85/7</td>
<td>1.41/8</td>
<td>0.01/5</td>
<td>0.01/7</td>
<td>0.06</td>
</tr>
<tr>
<td>ring-3</td>
<td>0.99/11</td>
<td>1.63/12</td>
<td>0.01/58</td>
<td>0.11/15</td>
<td>0.02/13</td>
</tr>
<tr>
<td>ring-4</td>
<td>1.02/15</td>
<td>1.77/17</td>
<td>0.02/13</td>
<td>1.45/26</td>
<td>0.13/16</td>
</tr>
<tr>
<td>ring-5</td>
<td>1.68/19</td>
<td>2.46/21</td>
<td>0.024/17</td>
<td>25.4/45</td>
<td>3.06/20</td>
</tr>
<tr>
<td>safe-10</td>
<td>0.87/10</td>
<td>0.88/10</td>
<td>0.016/10</td>
<td>0.02/10</td>
<td>no sol.</td>
</tr>
<tr>
<td>safe-30</td>
<td>1.05/30</td>
<td>2.85/30</td>
<td>0.072/30</td>
<td>1.24/30</td>
<td>no sol.</td>
</tr>
<tr>
<td>safe-50</td>
<td>1.49/50</td>
<td>20.6/56</td>
<td>0.232/50</td>
<td>27.17/50</td>
<td>no sol.</td>
</tr>
<tr>
<td>safe-70</td>
<td>2.48/70</td>
<td>90.8/70</td>
<td>0.488/70</td>
<td>113.6/70</td>
<td>no sol.</td>
</tr>
<tr>
<td>sort-05</td>
<td>0.93/15</td>
<td>0.94/12</td>
<td>0.196/15</td>
<td>NA</td>
<td>0/12</td>
</tr>
<tr>
<td>sort-10</td>
<td>1.85/54</td>
<td>3.18/39</td>
<td>OM</td>
<td>NA</td>
<td>0.03/38</td>
</tr>
<tr>
<td>sort-13</td>
<td>8.92/90</td>
<td>39.7/41</td>
<td>F</td>
<td>NA</td>
<td>0.08/55</td>
</tr>
<tr>
<td>sort-15</td>
<td>35.8/118</td>
<td>243/65</td>
<td>F</td>
<td>NA</td>
<td>0.14/65</td>
</tr>
<tr>
<td>uts-k-10</td>
<td>2.51/66</td>
<td>20.1/80</td>
<td>0.89/59</td>
<td>14.5/58</td>
<td>18.6/68</td>
</tr>
<tr>
<td>uts-k-20</td>
<td>22.4/136</td>
<td>142/197</td>
<td>11.6/119</td>
<td>1558/118</td>
<td>OM</td>
</tr>
<tr>
<td>uts-k-30</td>
<td>114/206</td>
<td>TO</td>
<td>59.8/179</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>uts-k-40</td>
<td>344/276</td>
<td>TO</td>
<td>224/239</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>uts-k-50</td>
<td>846/546</td>
<td>TO</td>
<td>592/299</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>uts-k-55</td>
<td>1265/381</td>
<td>TO</td>
<td>F</td>
<td>TO</td>
<td>OM</td>
</tr>
</tbody>
</table>

Table 3: Execution times for challenging domains found by DNF are usually shorter than the lengths of plans obtained by other planners.

<table>
<thead>
<tr>
<th>Problem</th>
<th>DNF</th>
<th>CPA(C)</th>
<th>T0</th>
<th>CFF</th>
<th>POND</th>
</tr>
</thead>
<tbody>
<tr>
<td>id-2-5</td>
<td>1.57/14</td>
<td>1.45/12</td>
<td>F</td>
<td>TO</td>
<td>46/14</td>
</tr>
<tr>
<td>id-2-7</td>
<td>19.4/14</td>
<td>14.3/12</td>
<td>F</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>id-2-9</td>
<td>451/14</td>
<td>OM</td>
<td>F</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>id-5-1</td>
<td>1.85/98</td>
<td>2.29/98</td>
<td>F</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>id-5-3</td>
<td>127/122</td>
<td>264/102</td>
<td>F</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>id-5-4</td>
<td>1732/536</td>
<td>OM</td>
<td>F</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>id-10-1</td>
<td>151/406</td>
<td>172/568</td>
<td>F</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>id-10-2</td>
<td>568/536</td>
<td>OM</td>
<td>F</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>ling-5-1</td>
<td>2.14/35</td>
<td>2.19/32</td>
<td>0.91/31</td>
<td>414/133</td>
<td>OM</td>
</tr>
<tr>
<td>ling-5-2</td>
<td>2.77/30</td>
<td>3.52/43</td>
<td>F</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>ling-5-3</td>
<td>27.4/99</td>
<td>28.7/94</td>
<td>114/115</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>ling-8-2</td>
<td>51.4/73</td>
<td>60.5/98</td>
<td>F</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>ling-8-3</td>
<td>449/32</td>
<td>OM</td>
<td>F</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>ling-10-1</td>
<td>157/157</td>
<td>160/204</td>
<td>F</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>ling-10-2</td>
<td>301/225</td>
<td>OM</td>
<td>F</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>push-5-1</td>
<td>1.89/64</td>
<td>2.35/113</td>
<td>0.49/107</td>
<td>13.8/82</td>
<td>TO</td>
</tr>
<tr>
<td>push-5-3</td>
<td>6.6/523</td>
<td>2546/3282</td>
<td>161/251</td>
<td>513/105</td>
<td>TO</td>
</tr>
<tr>
<td>push-5-5</td>
<td>14/63</td>
<td>TO</td>
<td>451/211</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>push-9-1</td>
<td>68/205</td>
<td>75/253</td>
<td>590/652</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>push-9-2</td>
<td>90/998</td>
<td>OM</td>
<td>F</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>push-9-3</td>
<td>157/2225</td>
<td>OM</td>
<td>F</td>
<td>TO</td>
<td>OM</td>
</tr>
<tr>
<td>push-10-4</td>
<td>640/4387</td>
<td>OM</td>
<td>F</td>
<td>TO</td>
<td>OM</td>
</tr>
</tbody>
</table>

Table 4: Execution times for small or easy instances.

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

**Considerations**

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 ∆ = {∅} results
Table 4: DNF search time/translation time vs. \( t_0 \) exec. time in \( \Delta \) in CPA, while it results in \( \{\{f, h\}, \{\neg f, g\}\} \) in DNF.

Another difference lies in the presentation 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 main representation of formula in this planner. 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 \( g \rightarrow f \). \( \Delta \) is enabling for \( a \). If the prime implicants form is used, \( \Delta \) 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 formulae. 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 \textit{uts-full-connected} from the IPC-08 competition—which leads to extremely large disjunctive normal form formulae—and the \textit{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 \( t_0 \) for providing us access to their systems. The research has been supported by NSF grants 0812267 and 0420407.

**References**


<table>
<thead>
<tr>
<th>Problem</th>
<th>DNF</th>
<th>TO</th>
<th>Problem</th>
<th>DNF</th>
<th>TO</th>
</tr>
</thead>
<tbody>
<tr>
<td>block-1</td>
<td>0.006/0.904</td>
<td>0.04</td>
<td>lg-4-2-4</td>
<td>0.41/2.39</td>
<td>0.05</td>
</tr>
<tr>
<td>block-2</td>
<td>0.033/0.907</td>
<td>0.2</td>
<td>ring-2</td>
<td>0.006/0.844</td>
<td>0.01</td>
</tr>
<tr>
<td>uts-c-3</td>
<td>0.009/1.171</td>
<td>0.12</td>
<td>ring-3</td>
<td>0.032/0.958</td>
<td>0.015</td>
</tr>
<tr>
<td>uts-c-4</td>
<td>0.031/1.259</td>
<td>0.42</td>
<td>safe-10</td>
<td>0.009/0.861</td>
<td>0.016</td>
</tr>
<tr>
<td>coms-10</td>
<td>0.018/0.952</td>
<td>0.036</td>
<td>safe-30</td>
<td>0.125/0.925</td>
<td>0.072</td>
</tr>
<tr>
<td>coms-15</td>
<td>0.08/1.33</td>
<td>0.12</td>
<td>sort-05</td>
<td>0.09/0.84</td>
<td>0.196</td>
</tr>
<tr>
<td>comm-15</td>
<td>0.11/1.3</td>
<td>0.11</td>
<td>lng-5-1-1</td>
<td>0.11/0.23</td>
<td>0.91</td>
</tr>
<tr>
<td>lg-4-2-2</td>
<td>0.06/1.86</td>
<td>0.036</td>
<td>push-5-1</td>
<td>0.079/1.811</td>
<td>0.11</td>
</tr>
</tbody>
</table>