Logical Circuit Filtering

Dafna Shahaf and Eyal Amir
Computer Science Department
University of Illinois, Urbana-Champaign
Urbana, IL 61801, USA
{dshahaf2,eyal}@uiuc.edu

Abstract

Logical Filtering is the problem of tracking the possible states of a world (belief state) after a sequence of actions and observations. It is fundamental to applications in partially observable dynamic domains. This paper presents the first exact logical filtering algorithm that is tractable for *all* deterministic domains. Our tractability result is interesting because it contrasts sharply with intractability results for structured stochastic domains. The key to this advance lies in using *logical circuits* to represent belief states. We prove that both filtering time and representation size are linear in the sequence length and the input size. They are independent of the domain size if the actions have compact representations. The number of variables in the resulting formula is at most the number of state features. We also report on a reasoning algorithm (answering propositional questions) for our circuits, which can handle questions about past time steps (*smoothing*). We evaluate our algorithms extensively on AI-planning domains. Our method outperforms competing methods, sometimes by orders of magnitude.

1 Introduction

Much work in AI applies system models whose state changes over time. Applications use these dynamic-system models to diagnose past observations, predict future behavior, and make decisions. Those applications must consider multiple possible states when the initial state of the system is not known, and when the state is not observed fully at every time step. One fundamental reasoning task in such domains is *Logical Filtering* [Amir and Russell, 2003]. It is the task of finding the set of states possible (belief state) after a sequence of observations and actions, starting from an initial belief state.

Logical Filtering in large *deterministic* domains is important and difficult. Planning, monitoring, diagnosis, and others in partially observable deterministic domains estimate the belief state (e.g., [Biere *et al.*, 1999; Cimatti and Roveri, 2000; Bertoli *et al.*, 2001; Petrick and Bacchus, 2004]) as part of performing other computations. This estimation is difficult because the number of states in a belief state is exponential in the number of propositional features defining the domain.

Several approaches were developed that represent belief states compactly in logic (e.g., BDDs [Bryant, 1992], Logical Filter, and database progression [Winslett, 1990; Lin and Reiter, 1997]) and update this representation. However, none of them guarantees compact representation, even for simple domains. [Amir and Russell, 2003], for instance, guarantees compactness and tractability only for sequences of STRIPS actions whose preconditions are known to hold. Most importantly, the straightforward approach to Logical Filtering (deciding if a clause should be in the belief state representation of time $t+1$, based on the belief state of time t) was shown to be coNP-complete [Liberatore, 1997].

In this paper we show that solving the update problem in its entirety is easier (done in poly-time) than creating the new belief state piecemeal. We present *C-Filter*– the first exact, tractable Logical Filtering algorithm that can handle *any* deterministic domain. Importantly, both time (to compute a belief state) and space (to represent it) do not depend on the domain size. Furthermore, the number of variables in the resulting formula is at most n, the number of state features (compare this with $n \cdot t$, the number of variables used in Bounded Model Checking [Clarke *et al.*, 2001]).

We extend C-Filter to NNF Circuits (no internal negation nodes), and show that similar space and time bounds hold for this more restricted representation. We further show how to reason with an output circuit, including *smoothing* (queries about the past). Our results are also useful from the perspective of representation-space complexity; they sidestep previous negative results about belief-state representation (Section 5.4) and intractability results for stochastic domains.

The key to our advance lies in using *logical circuits* to represent belief states instead of traditional formulas. We show that updating a logical circuit formula amounts to adding a few internal connectives to the original formula. We take advantage of determinism: a feature is true after an action iff the action made it true or it was already true and the action did not change that. Interestingly, our empirical examination suggests that other graphical representations (e.g., BDDs) do not maintain compact representation with such updates.

C-Filter applies to many problems that require belief-state update, such as Bounded Model Checking and planning with partial observability. The attractive nature of this approach is that we can apply standard planning techniques without fear of reaching belief states that are too large to represent.

2 Logical Filtering

We now describe the problem of Logical Filtering (tracking the state of the world), hereby referred to as Filtering. Imagine an assembly robot that can put items together in order to construct some machine. The parts are randomly oriented, and they must be brought to goal orientations for assembly. At the beginning, the parts are located on a conveyor belt. Each part drifts until it hits a fence perpendicular to the belt, and then rotates so one of its edges is aligned against the fence (see Figure 1). A sensor measures that edge, providing partial information about the part's orientation (partial, because the part can have some edges of equal length, and the sensor might be noisy). The robot can then rotate a part (by a certain, discrete amount) and place it back on the belt, or pick it up and try assembling it. We now define the problem formally.

Definition 2.1 (Deterministic Transition System) *A* transition system *is a tuple* $\langle P, S, A, R \rangle$. P *is a finite set of propositional fluents,* $S \subseteq Pow(P)$ *is the set of world states. A state contains exactly the fluents that are true in it.* A *is a finite set of actions, and* $R : S \times A \to S$ *is the transition function.*

Executing action a in state s results in state $R(s, a)$. R may be partial. In our example, $P = \{$ *OnBelt(part1), PartOfAssembly(part1), Touch(part1-e1), Touch(part1-e2), ...* $\}$, $A = \{$ *PickUp(part1), Assemble(part1), Rotate(part1,90), ...* $\}$
In the sequel we assume an implicit transition system.

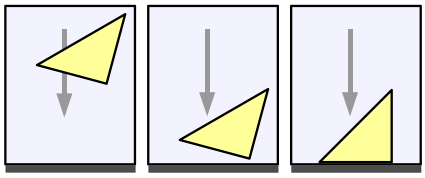


Figure 1: A conveyor belt: the triangle drifts down, hits the fence and rotates. The edge touching the fence is then measured.

Our robot tries to keep track of the state of the world, but it cannot observe it completely. One possible solution is to maintain a *belief state*– a set of possible world states. Every $\rho \subseteq S$ is a belief state. The robot updates its belief state as the result of performing actions and receiving observations. We now define the semantics of Filtering.

Definition 2.2 (Filtering Semantics [Amir and Russell, 2003]) $\rho \subseteq S$*, the states that the robot considers possible,* $a_i \in A$*. We assume that observations* o_i *are logical sentences over* P*.*

1. $Filter[\epsilon](\rho) = \rho$ *(ϵ: an empty sequence)*
2. $Filter[a](\rho) = \{s' \mid s' = R(s, a), s \in \rho\}$
3. $Filter[o](\rho) = \{s \in \rho \mid o \text{ is true in } s\}$
4. $Filter[\langle a_j, o_j \rangle_{i \le j \le t}](\rho) =$
   $Filter[\langle a_j, o_j \rangle_{i < j \le t}]\ (Filter[o_i](Filter[a_i](\rho)))$

We call step 2 progression with a *and step 3 filtering with* o*.*

Every state s in ρ becomes $s' = R(s, a)$ after performing action a. After receiving an observation, we eliminate every state that is not consistent with it.

Figure 2 illustrates a belief-state update. Imagine that the robot has an isosceles right triangle (edges of size 1,1,$\sqrt{2}$), and one of the 1-edges is currently touching the fence. There are two possible orientations ((a) and (b), left part). After rotating the triangle 90 degrees, each possible state is updated (middle part). If the world state was (a), we should see a 1-edge again. Otherwise, we expect to see the $\sqrt{2}$-edge. After observing a 1-edge (right part), the robot eliminates (b) from his belief state, leaving him only with (a). That is, the robot knows the orientation of the triangle.

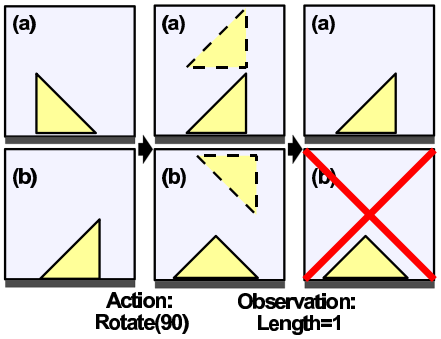


Figure 2: A belief-state update with a 1,1,$\sqrt{2}$ triangle. Left: Possible initial states. Middle: Progressing with *Rotate(90)*– rotating the triangle by 90° and putting it on the belt again. Right: After observing length=1, state (b) is eliminated.

3 Circuit Filtering

Filtering is a hard problem. There are $2^{2^{|P|}}$ belief states, so naïve methods (such as enumeration) are intractable for large domains. Following [Amir and Russell, 2003], we represent belief states in logic. Their solution guarantees an exact and compact representation only for a few classes of models (e.g. permuting domains, belief-states in a canonical form). We use *Logical Circuits* (not flat formulas) in order to extend their results to all deterministic domains. In this section we describe our representation and explain how to update it with an action-observation sequence, and how to reason with the result.

3.1 Representation

A belief-state ρ can be represented as a logical formula φ over some $P' \supseteq P$: a state s is in ρ iff it satisfies φ ($s \wedge \varphi$ is *satisfiable*). We call φ a *belief-state formula*. We represent our belief state formulas as circuits.

Definition 3.1 (Logical Circuits) Logical Circuits *are directed acyclic graphs. The leaves represent variables, and the internal nodes are assigned a logical connective. Each node represents a formula– the one that we get by applying the connective to the node's children.*

We allow the connectives $\wedge, \vee, \neg$. $\neg$ nodes should have exactly one child, while $\wedge, \vee$ can have many. In Corollary 5.6 we explain how to avoid internal $\neg$ nodes (for NNF).

We use logic to represent R, too: a *domain description* is a finite set of *effect rules* of the form "a **causes** F **if** G", for a an action, F and G propositional formulas over P. W.l.g., F is a conjunction of literals. The semantics of these rules is as follows: after performing a in state s, iterate through its rules. If the rule's precondition G holds in s, its effect F

will hold in $R(s,a)$. If this leads to a contradiction, a is not possible to execute. The rest of the fluents stay the same; if no preconditions hold, the state does not change (we can also make action failure lead to a *sink state*).

Consider the triangle in Figure 2. If the triangle is on the belt, action a = *Rotate(90)* will rotate it, so the touching edge will change: *e1* to *e2*, *e2* to *e3*, *e3* to *e1*. a's effect rules are:

"a **causes** *Touch(e2)* $\wedge \neg$*Touch(e1)* **if** *OnBelt()* $\wedge$ *Touch(e1)*"
"a **causes** *Touch(e3)* $\wedge \neg$*Touch(e2)* **if** *OnBelt()* $\wedge$ *Touch(e2)*"
"a **causes** *Touch(e1)* $\wedge \neg$*Touch(e3)* **if** *OnBelt()* $\wedge$ *Touch(e3)*"

3.2 C-Filter

We described how domains and belief-states are represented; we can now present our Circuit-Filtering algorithm, *C-Filter*.

PROCEDURE ***C-Filter***$(\langle a_i, o_i\rangle_{0<i\leq t}, \varphi, D)$
{a_i actions, o_i observations, φ an initial belief state over P, D domain description}
Preprocess D, φ
1: ***ProcessDomain***$(D, \langle a_i\rangle_{0<i\leq t})$
2: **for** $f \in P$ **do** $expl_f$:= a new proposition f_0
3: cb := ***Time0***(φ)
Process sequence
4: **for** i = 1 **to** t **do**
5: ***ProgressAction***(a_i)
6: ***FilterObservation***(o_i)
7: **return** $cb \wedge \bigwedge_{f\in P}(f \leftrightarrow expl_f)$

PROCEDURE ***ProgressAction***(a)
{a an action}
Update *cb*: a executed, thus was possible.
Get f's next-value explanation:
a caused f, or f held and a did not cause $\neg f$
1: **for** $f \in$ *Effects(a)* **do**
2: $cb := cb \wedge$ ***Time0***(*Poss(a,f)*)
3: $expl'_f$:= ***Time0***(*NextVal(a,f)*)
4: **for** $f \in$ *Effects(a)* **do** $expl_f := expl'_f$

PROCEDURE ***FilterObservation***(o)
{o an observation over P}
1: $cb := cb \wedge$ ***Time0***(o)

PROCEDURE ***Time0***(ψ)
{ψ a formula}
Return an equivalent circuit over time 0
1: **for** $f \in P$ in ψ **do** replace f with the node pointed by $expl_f$
2: **return** ψ

PROCEDURE ***ProcessDomain***$(D, \langle a_i\rangle_{0<i\leq t})$
{D a domain description, a_i actions}
Extract "Next Value" and "Possible" Formulas
1: **for** $f \in P, a \in \langle a_i\rangle$ **do**
2: *NextVal(a,f)* := *Cause(a,f)* $\vee$ ($f \wedge \neg$*Cause(a,¬f)*) [1]
3: *Poss(a,f)* := $\neg$(*Cause(a,f)* $\wedge$ *Cause(a,¬f)*)
4: (optional: Simplify Formulas)

Figure 3: *C-Filter* Algorithm

Algorithm Overview

C-Filter is presented in Figure 3 and demonstrated in Section 4. It receives an action-observation sequence, an initial belief state formula, φ, over P, and a domain description D. It outputs the filtered belief state as a logical circuit.

The algorithm maintains a circuit data structure, and pointers to some of its nodes. A pointer to a node represents the formula which is rooted in that node (and they will be used interchangeably). We maintain pointers to the following formulas: (1) A formula cb (constraint base) – the knowledge obtained so far from the sequence (receiving observations and knowing that actions were possible to execute constrains our belief state). (2) For every fluent $f \in P$, a formula $expl_f$; this formula explains why f should be true now (in Figure 4, the node marked *e(Tch1)* is the explanation formula of *Touch(e1)* at time t, and the root node is the explanation at time $t+1$).

We keep the number of variables in our representation small ($|P|$) by allowing those formulas to involve *only* fluents of time 0. In a way, this is similar to regression: **$expl_f$ expresses the value of fluent f as a function of the initial world state, and *cb* gives the constraints on the initial state.**

The belief state is always $cb \wedge \bigwedge_{f\in P}(f \leftrightarrow expl_f)$. In other words, a possible model should satisfy *cb*, and each fluent f can be replaced with the formula $expl_f$.

In the preprocessing phase, we extract data from the domain description (Procedure *C-Filter*, line 1). We then create a node for each fluent, and initialize the $expl_f$ pointers to them. We also create a circuit for the initial belief-state, φ (using the $expl_f$ nodes), and set the *cb* pointer to it (lines 2-3). Then we iterate through the sequence, update the circuit and the pointers with every time step (lines 4-6, see below), and finally return the updated belief state.

A Closer Look

The circuit is constructed as follows. In the preprocessing stage, we extract some useful formulas from the domain description. Let *Effects(a)* be the set of fluents that action a might affect. For each f in this set, we need to know how a can affect f. Let *Cause(a,f)* be a formula describing when a causes f to be true. It is simply the precondition of the rule of a causing f to hold (if there are several, take the disjunction; if there are none, set it to *FALSE*). That is, if $s \models$ *Cause(a,f)* and a is possible to execute, f will hold after it. *Cause(a,¬f)* is defined similarly.

For example, take a = *Rotate(90)* (Section 3.1). *Effects(a)*={*Touch(e1)*, *Touch(e2)*, *Touch(e3)*} , and

Cause(a,Touch(e1)) = *OnBelt()* $\wedge$ *Touch(e3)*
Cause(a,¬ Touch(e1)) = *OnBelt()* $\wedge$ *Touch(e1)* (*)

Procedure *ProgressDomain* then constructs the formula *NextVal(a,f)*, which evaluates to *TRUE* iff f holds after a (given the previous state). Intuitively, either a caused it to hold, or it already held and a did not affect it. Similarly, the formula *Poss(a,f)* states that a was possible to execute regarding f, i.e. did not cause it to be true and false at the same time.

After preprocessing, we iterate through the sequence. Procedure *ProgressAction* uses those formulas to update the belief state: First, it constructs a circuit asserting the action was

[1] *Cause(a,f)* represents the conditions for a to cause f, extracted from the domain description (See (*))

possible (corresponding to the *Poss* formula) and adds it to *cb* (line 2). Then, it builds a circuit for the *NextVal* formula. Procedure *Time0* ensures the circuit uses only time-0 fluents. When we construct a new *Poss* or *NextVal* circuit, its leafs represent fluents of the previous time step; *Time0* replaces them by their *equivalent* explanation nodes. Our circuit implementation is crucial for the efficiency of this replacement. Instead of copying the whole formula, we only need to update edges in the graph (using the pointers). This way, we can share formulas recursively, and maintain compactness.

After all the new circuits were built, the explanation pointers are updated (line 4); the new explanation is the root of the corresponding *NextVal* circuit, built earlier (line 3; see also Section 5.1). Then we deal with the observation (Procedure *FilterObservation*): similarly, we use *Time0* to get a time-0 formula, and simply add it to *cb*.

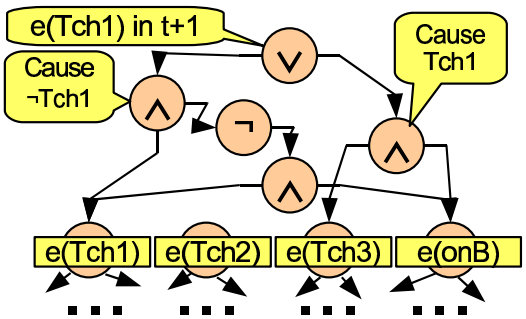


Figure 4: Updating the explanation of *Touch(e1)* after *Rotate(90)*

Example: Figure 4 shows an update of the explanation of *Touch(e1)* after the action *Rotate(90)*. Yellow rectangles (on the bottom nodes) represent the explanation pointers of time t (before the action). The circuit in the image is the *NextVal* formula, after Procedure *Time0* replaced its fluents by the corresponding explanation nodes.

The $\vee$ node is the root of the graph representing state of *Touch(e1)* after the action: the right branch describes the case that the action caused it to hold, and the left branch is the case that it held, and the action did not falsify it. In the next iteration, the pointer of *Touch(e1)* will point at this node.

Note the re-use of some time-t explanation nodes; they are internal nodes, possibly representing large subformulas.

3.3 Query Answering with the End Formula

C-Filter returns an updated belief state φ^t, represented as a logical circuit. We are interested in satisfiability queries ($\varphi^t \wedge \psi$ satisfiable) and entailment queries ($\varphi^t \models \psi$, or $\varphi^t \wedge \neg\psi$ unsatisfiable). In the following, we construct a circuit corresponding to the query and run inference on it.

Query Circuits

Let ψ be an arbitrary propositional query formula; we want to check whether $\varphi^t \wedge \psi$ is satisfiable. Very similarly to an observation, we add ψ to *cb*, and replace the fluents for their explanations. The new *cb* is our query circuit. Queries are usually about time t, but the circuit structure allows more interesting queries, in particular *smoothing*– queries that refer to the past (e.g., did f change its value in the last 5 steps? Could the initial value of g be TRUE?). Note that every fluent in every time step has a corresponding node. If we keep track of those nodes, we can replace fluents from any time step by their explanations. If the queries are given in advance, this does not change the complexity. Otherwise, finding a past-explanation node might take $O(\log t)$ time. Note that the same mechanism (tracking previous explanations) has many interesting applications, such as filtering in non-Markovian domains.

SAT for Circuits

After building a query circuit, we check satisfiability. Traditional approaches check circuit-SAT by converting the circuit into a CNF formula. The approaches for doing so either grow the representation exponentially (duplicating shared subformulas) or grow the number of variables significantly.

Instead, we run inference on the circuit itself. A number of works show that the structural information lost in a CNF encoding can be used to give SAT procedures a significant performance improvement. Using circuit SAT solvers, we can solve the problem more efficiently and effectively in its original non-clausal encoding. Several such algorithms have been proposed recently, taking advantage of the circuit structure [Ganai *et al.*, 2002; Thiffault *et al.*, 2004]. We use those, and a simple algorithm of our own, *C-DPLL*.

C-DPLL is a generalization of DPLL. Every iteration, an uninstantiated variable f is chosen, and set to *TRUE*. The truth value is then propagated as far as possible, resulting in a smaller circuit (for example, if f had an OR parent, it will be set to *TRUE* as well). Then, *C-DPLL* is called recursively. If no satisfying assignment was found, it backtracks and tries f=*FALSE*. If no assignment is found again, return UNSAT. *C-DPLL* takes $O(|E| \cdot 2^l)$ time and $O(|E|)$ space for a circuit with $|E|$ edges and l leaves.

4 Extended Example

We will now give a detailed example of the whole process. Interestingly, this example shows that our circuit implementation can compactly represent a belief state that *cannot* remain compact using CNF formulas over the same variables.

Our domain includes fluents $\{p_1, ..., p_n, odd\}$. We will come up with a sequence of actions that will make *odd* equal to $p_1 \oplus p_2 \oplus ...p_n$, the parity of the other fluents. Our actions $a_1, ..., a_{n-1}$ are defined such that a_1 sets $odd := p_1 \oplus p_2$, and any other a_i sets $odd := odd \oplus p_{i+1}$. More formally:

"a_1 **causes** *odd* **if** $(p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge p_2)$"
"a_1 **causes** $\neg$*odd* **if** $\neg[(p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge p_2)]$"
"a_i **causes** *odd* **if** $(odd \wedge \neg p_{i+1}) \vee (\neg odd \wedge p_{i+1})$"
"a_i **causes** $\neg$*odd* **if** $\neg[(odd \wedge \neg p_{i+1}) \vee (\neg odd \wedge p_{i+1})]$"

Applying the sequence $a_1, ..., a_{n-1}$ sets $odd = p_1 \oplus ...p_n$. We now show how our algorithm maintains the belief state throughout the sequence.

Preprocessing the Domain:

In this phase we extract the *Poss* and *NextVal* formulas. We examine the action specifications: the only fluent which is affected is *odd*. When is it possible to execute a_1? When executing it does not cause both *odd*, $\neg$*odd*.

Cause(a_1,*odd*) $= (p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge p_2)$

$Cause(a_1, \neg odd) = \neg[(p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge p_2)]$
$Poss(a_1, odd) = \neg[Cause(a_1, odd) \wedge Cause(a_1, \neg odd)]$

It is easy to see that both cannot hold simultaneously, and the formula can be simplified to *TRUE*: indeed, a_1 is always executable. Similarly, all of our actions are always possible to execute, so the *Poss* formulas are all equal *TRUE*.

Now, the *NextVal* formulas. After executing a_1, *odd* will be set to $Cause(a_1, odd) \vee [odd \wedge \neg Cause(a_1, \neg odd)]$.

This is equivalent to $Cause(a_1, odd)$. In other words, *odd* will be set to $p_1 \oplus p_2$. Similarly, after action a_i *odd* will be set to $p_{i+1} \oplus odd$. Note, simplifying the formulas is not mandatory; the representation will be compact without it, too.

Executing the Actions:
Imagine that we receive the (arbitrary) sequence that $a_1, a_2, ..., a_{n-1}, odd \wedge \neg p_n$ (performing n actions and receiving an observation). Figure 5 describes how we update the belief-state according to this sequence. At time 0 (5a) we create a node for every fluent, and another for *TRUE*. The nodes represent the value of the fluent at time 0. We set a pointer (the yellow rectangles) for each formula that we want to maintain: the formula for *cb* is set to *TRUE*, because we do not have any initial knowledge. The explanation formula of each fluent is set to the corresponding node.

We then execute a_1, arriving at time 1 (5b). No constraint was added to *cb*, since the action is always executable. No explanation formula of p_i changed, since a_1 does not affect them. The only thing that changed is the state of *odd*: its new explanation is $p_1 \oplus p_2$. We construct the graph for this formula, and update the explanation pointer to its root node.
NOTE: the image shows xor gates just for the sake of clarity. In fact, each of them should be replaced by five gates, as depicted in 5b.

Executing a_2 is similar (time 2, 5c). We construct the graph for *odd*'s new value, $odd \oplus p_3$. Note that we substitute the fluents in this formula (odd, p_3) by their explanations in time 1, i.e. the pointers of the previous time step.

We execute $a_3, ..., a_{n-1}$, and then observe $odd \wedge \neg p_n$ (5d). This is just an example observation; alternatively, you can think of it as querying whether it is possible that $odd \wedge \neg p_n$ holds now. First, we handle the actions, updating the explanation of *odd*. We then add $odd \wedge \neg p_n$ to our knowledge, constructing our new *cb* circuit and updating the pointer. Finally, we return the circuit in 5d, along with the pointers. This is our updated belief state.

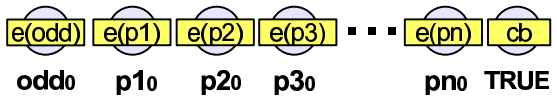
Answering Queries:
In 5e we show an example of truth-value propagation: if we assume that at time 0 p_1=*TRUE* and the rest are set to *FALSE*, those values are propagated up and result in *cb*=*TRUE*. That is, this assignment is consistent with our sequence.

5 Analysis and Complexity

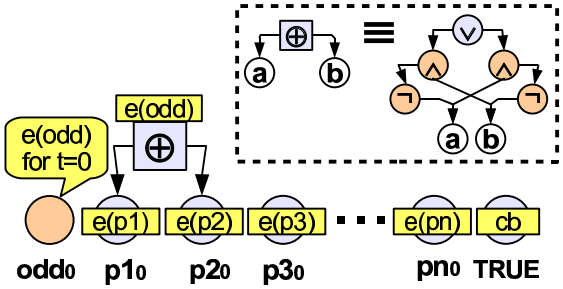
5.1 Correctness

Theorem 5.1 *C-Filter is correct. For any formula φ and a sequence of actions and observations $\langle a_i, o_i \rangle_{0<i\leq t}$,*

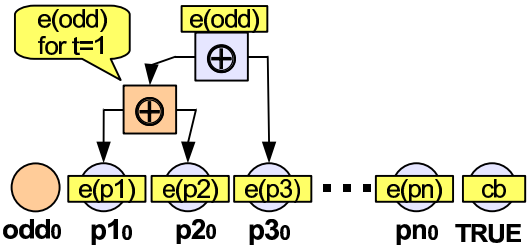
$$\{ s \in S \text{ that satisfy C-Filter}(\langle a_i, o_i \rangle_{0<i\leq t}, \varphi)\} = Filter[\langle a_i, o_i \rangle_{0<i\leq t}](\{s \in S \text{ that satisfy } \varphi\}).$$



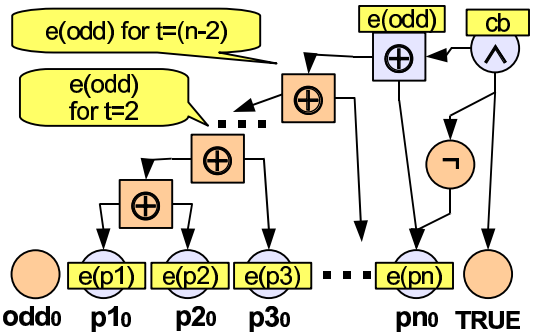
(a) At time t=0: initial belief state $\varphi = TRUE$



(b) Time t=1: after performing a_1



(c) Time t=2: after performing a_1, a_2



(d) Time t=(n-1): after performing $a_1, .., a_{n-1}$ and observing $(odd \wedge \neg p_n)$

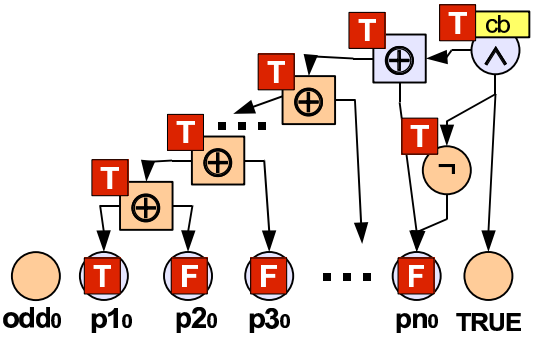


Figure 5: (e) Propagating truth-values

Recall that a state s satisfies formula φ if $s \wedge \varphi$ is *satisfiable* (Section 3.1). s is used as a formula and as a state.

PROOF SKETCH We present an effect model, and show how to update a belief-state (flat) formula with this model. We show that the Filtering definition in Section 2 can be reduced to consequence finding (in a restricted language) with this formula. Then, we show that *C-Filter* computes exactly those consequences.

Definition 5.2 ***Effect Model:***
For an action a, define the ***effect model*** *of a at time t to be:*

$$T_{\mathit{eff}}(a,t) = a_t \rightarrow \bigwedge_{f \in P} \mathit{Poss}(a,f,t) \ \wedge \ (f_{t+1} \leftrightarrow \mathit{NextVal}(a,f,t))$$

$$\mathit{Poss}(a,f,t) = \neg(\mathit{Cause(a,f)}_t \wedge \mathit{Cause(a,\neg f)}_t)$$

$$\mathit{NextVal}(a,f,t) = \mathit{Cause(a,f)}_t \vee (f_t \wedge \neg \mathit{Cause(a,\neg f)}_t)$$

a_t asserts that action a occurred at time t, and f_{t+1} means that f after performing a. ψ_t is the result of adding a subscript $_t$ to every fluent in formula ψ (see Section 3.2 for definition of *Cause(a,f)*). The effect model corresponds to effect axioms and explanation closure axioms from Situation Calculus [Reiter, 1991]. If the robot can recognize an impossible action, we can drop the assumption that actions are possible, and adopt a slightly different effect model.

Recall that *Filter*$[](\cdot)$ was defined over a set of states. We now define its analogue *L-Filter*, which handles belief-state logical formulas.

Definition 5.3 (L-Filter) *Let φ a belief-state formula.*
- *L-Filter*$[a](\varphi) = Cn^{L_{t+1}}(\varphi_t \wedge a_t \wedge T_{\mathit{eff}}(a,t))$
- *L-Filter*$[o](\varphi) = \varphi \wedge o$

where $Cn^L(\psi)$ are the consequences of ψ in vocabulary L. $L_{t+1} = (L(\varphi_t) \cup P_{t+1}) \setminus P_t$, for $P_t = \{f_t \mid f \in P\}$ and $L(\varphi_t)$ the language of φ_t; i.e., L_{t+1} does not allow fluents with subscript t.

Lemma 5.4 *The result of applying L-Filter$[a]$ for $a \in A$ is a formula representing exactly the set of states Filter$[a]$.*
More formally, let φ be a belief state formula.

$$\mathit{Filter}[a](\{s \in S \mid s \text{ satisfies } \varphi\}) = \{s \in S \mid s \text{ satisfies } \mathit{L\text{-}Filter}[a](\varphi)\}$$

That is, both definitions are equivalent (for lack of space, we do not present the proof here). As a result, we can compute *Filter* using a consequence finder in a restricted language. However, this does not guarantee tractability. Instead of using a consequence-finder, we show that *C-Filter* computes exactly those consequences.

Let $\psi := \varphi_t \wedge a_t \wedge T_{\mathit{eff}}(a,t)$. According to our definition, *L-Filter*$[a](\varphi) = Cn^{L_{t+1}}(\psi)$. We now observe that consequence finding is easy if we keep φ_t in the following form:

$$\varphi_t = \mathit{cb} \wedge \bigwedge_{f \in P}(f_t \leftrightarrow \mathit{expl}_f)$$

cb and expl_f do not involve any fluent of time t.

We now show how to compute the consequences of such formulas. Furthermore, we show that the resulting formula maintains this form, so we only need to check the form of the initial belief-state. Luckily, this is not a problem; every initial belief-state can be converted to this form (in linear time) using new proposition symbols.

Let ψ be a formula in this form. ψ states that $(f_t \leftrightarrow \mathit{expl}_f)$ for every $f_t \in P_t$: we construct an equivalent formula, ψ', by replacing every $f_t \in P_t$ in $T_{\mathit{eff}}(a,t)$ with the formula expl_f. Notation: $\psi' := \varphi_t \ \wedge a_t \ \wedge \ T_{\mathit{eff}}(a,t)[\mathit{expl}_f / f_t]$.

$$\psi \equiv \psi' \Rightarrow Cn^{L_{t+1}}(\psi) \equiv Cn^{L_{t+1}}(\psi')$$

Therefore, we can find the consequences of ψ' instead. Note that consequence finding in L_{t+1} is the same as using the Resolution algorithm to resolve fluents of P_t. We use this to compute $Cn^{L_{t+1}}(\psi')$:

$$Cn^{L_{t+1}}(\psi') \equiv \mathit{cb} \wedge \bigwedge_{f \in P}(\mathit{Poss}(a,t,f)[\mathit{expl}_g / g_t]) \ \wedge \bigwedge_{f \in P}(f_{t+1} \leftrightarrow \mathit{NextVal}(a,t,f)[\mathit{expl}_g / g_t])$$

Let $\mathit{cb}' := \mathit{cb} \wedge \bigwedge_{f \in P}(\mathit{Poss}(a,t,f)[\mathit{expl}_g / g_t])$
$\mathit{expl}'_f := \mathit{NextVal}(a,t,f)[\mathit{expl}_g / g_t])$.

The last formula can be re-written as

$$\mathit{cb}' \wedge \bigwedge_{f \in P}(f_{t+1} \leftrightarrow \mathit{expl}'_f)$$

Now note that *C-Filter* maintains the belief-state formula exactly in that easy-to-compute form, namely $\mathit{cb} \wedge \bigwedge_{f \in P}(f_t \leftrightarrow \mathit{expl}_f)$, *cb* and expl_f involve only special propositions, representing time-0 (to avoid confusion, you might think of the new propositions in line 2 as f_{init}, not f_0).

Also, $\mathit{cb}', \mathit{expl}'_f$ are exactly the constraint-base and explanation formulas after *C-Filter*'s *ProgressAction*. That is, *C-Filter* correctly progresses the belief-state with actions. The proof for handling observations is similar. ■

5.2 C-Filter Complexity

Let φ^0 be the initial belief state, t the length of the action-observation sequence, and $|\mathit{Obs}|$ the total length of the observations in the sequence. Let *ActDesc* be the longest description of an action $a \in A$ (preconditions + effects).

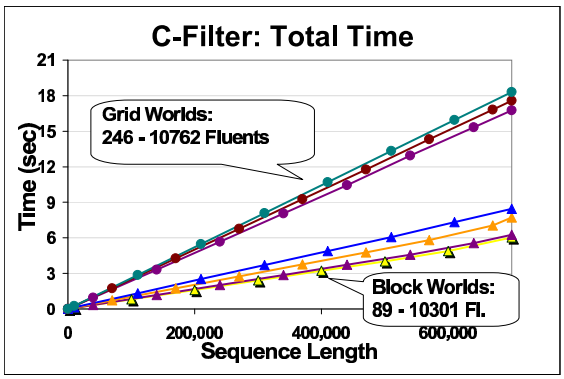
Theorem 5.5 *Allowing preprocessing of $O(|P|)$ time and space (or, alternatively, using a hash table), C-Filter takes time $O(|\varphi^0| + |\mathit{Obs}| + t \cdot \mathit{ActDesc})$. Its output is a circuit of the same size.*

If the observations are always conjunctions of literals, we can drop $|\mathit{Obs}|$ from space complexity. If there are no preconditions, we can maintain a flat formula instead. Note that this *does not* depend on the domain size, $|P|$. *ActDesc* is usually small– especially if the actions in the domain affect a small number of fluents, and have simple preconditions.

PROOF SKETCH: Initializing *cb* takes $O(|\varphi^0|)$ time. Handling each action adds at most $O(\mathit{ActDesc})$ nodes and edges to the graph, and takes the same time: in the worst case, assuming no simplifications were done, we need to construct a graph for each of the action's *Causes* formulas. Finally, each time we receive an observation o we add at most $O(|o|)$ nodes and edges, resulting in total $O(|\mathit{Obs}|)$.

Corollary 5.6 *We can maintain an NNF-Circuit (no negation nodes) with the same complexity.*

PROOF SKETCH The circuit's leaves represent *literals* (instead of propositions). We maintain explanation formulas for them. Since $(f \leftrightarrow \mathit{expl}_f) \Rightarrow (\neg f \leftrightarrow \neg \mathit{expl}_f)$, we can define $\mathit{expl}_{\neg f} := \neg \mathit{expl}_f$. We take the NNF-form of every



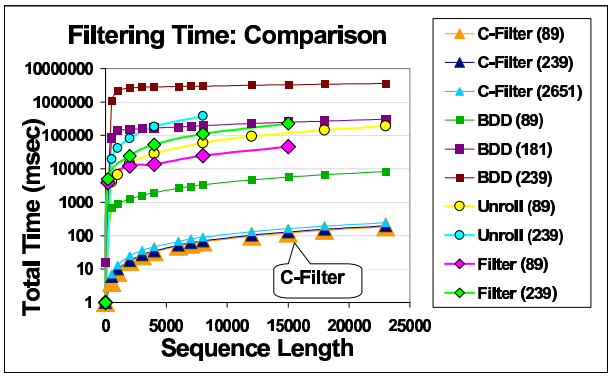


Figure 6: Left: Filtering time (sec) for C-Filter, applied to Block-World and Grid domains of different sizes. The time is linear, and does not depend on the domain size (slight differences in the graph are due to hash-table implementation). Right: Comparison of Filtering time (msec) for several methods (numbers represent domain size). Note that this is *log-scale*.

formula we use (observations, explanations, etc.), and replace every literal by its explanation. Converting to NNF takes time linear in the formula's size; therefore, time and space complexities will not change (modulo a small constant). ■

5.3 Projection

Projection is the problem of checking that a property holds after t action steps of an action system, starting from a belief state. Generalizations allow additional observations.

Our results from previous sections show that projection is doable by applying *C-Filter* (generating the belief state at time t, φ^t, adding the query and running our *C-DPLL* solver).

Let m be maximal length of a single observation plus *ActDesc*. Usually $m = O(1)$. Since φ^t includes $O(n)$ variables, and has overall size $O(n + mt)$, checking if a variable assignment is a model of φ^t takes time $O(n+mt)$. Thus, testing satisfiability is NP-Complete in n, instead of $n + mt$ (the size of the formula) or $n \cdot t$ (the number of propositional variables that appear in an unrolling of the system over t steps; used in Bounded Model Checking). We need to guess n variable assignments, and then apply a linear algorithm to check it. The following result refines earlier complexity results.

Theorem 5.7 (Projection) *Let D be a domain with deterministic actions. The problem of answering whether all the states in $Filter[\pi](\varphi)$ satisfy Q, for belief state formula φ, sequence of actions π and query Q, is coNP-complete in the number of state variables, n. We assume π, φ, ActDesc and Q are polynomial in n.*

5.4 Representation Complexity

Our results have implications for the theory of representation-space complexity. A simple combinatorial argument shows that there are belief states of n fluents that cannot be described using logical circuit of size $o(2^n)$, i.e., strictly smaller than some linear function of 2^n. Nevertheless, our results for *C-Filter* show that those belief states that are reachable within a polynomial number of actions from an initial belief state are of size linear in t and the input size (initial belief state size, and longest action description).

Also, [Amir and Russell, 2003] showed that for every general-purpose representation of belief states there is a (possibly nondeterministic) domain, an initial belief state, and a sequence of actions after which our belief state representation is exponential in the initial belief state size. Our results show that this does not hold for deterministic systems.

5.5 A Note on Non-Determinism

C-Filter can handle non-determinism in observations, but not in transition models. However, many real-life environments are inherently non-deterministic. One natural solution is to treat each non-deterministic action as a choice between several deterministic ones. For example, coin-flipping:

$$flip_t \rightarrow [ExactlyOneCase \wedge (case_1 \rightarrow heads_{t+1}) \wedge (case_2 \rightarrow \neg heads_{t+1})]$$

where *ExactlyOneCase* specifies that exactly one of $case_1$, $case_2$ holds (we can also use a binary encoding). Unfortunately, the number of propositions grows linearly with t.

We can handle another (very limited) non-deterministic class without adding propositions (specifically, "a **causes** $p \vee q$ if G", G deterministic). The idea is to maintain a belief state in a different form, with $(l \rightarrow expl_l)$ for every literal l .

6 Experimental Evaluation

Our Filtering algorithm was implemented in C++. Our implementation could also handle parametrized domain descriptions, such as STRIPS. We tested it on AI-Planning domains (Figure 7 lists several) of various sizes and observation models. We generated long action-observation sequences with a random sequence generator (implemented in Lisp), and ran inference on the results using our own *C-DPLL* and *NoClause* ([Thiffault *et al.*, 2004]). circuit SAT solver.

Blocks: 108/124	Ferry: 163/17	Grid: 251/53
Gripper: 110/6	Hanoi: 259/16	Logistics: 176/16
Movie: 47/13	Tsp: 98/15	

Figure 7: Overview of *C-Filter* experiments: AI-Planning domains (2000+ fluents, 10000 steps). Results presented as Filtering time/ Model finding time (both in msec).

Figures 6, 7, 8 present some of the results. Figure 6 (left) shows that *C-Filter* is linear in the sequence length; note that

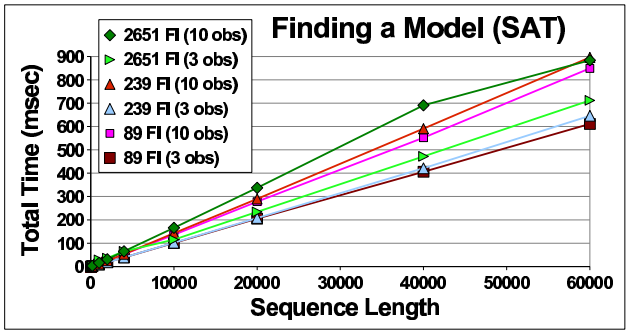


Figure 8: Total time for finding a model (msec), for Block-Worlds of different size and number of observations per step.

time depends on the domain but *not* on the domain size. In both Block-World and Grid-World, filtering time almost does not change, even when the number of fluents grows 100 times larger (slight difference in the graph is due to hash-table implementation, instead of an array; the circuit size does not depend on this implementation and was the same).

The right part of Figure 6 shows a comparison to other filtering methods. We compared our algorithm to (1) *Filter* [Amir and Russell, 2003] (in Lisp), (2) Filtering by unrolling the system over t steps (using $|P|t$ propositions), (3) BDD-based Filtering, based on the BuDDy package [Lind-Nielsen, 1999]. *C-Filter* outperformed them all, sometimes by orders of magnitude; note that the graph is *log-scale*.

Comparison Analysis: BDD sizes depend highly on variable ordering. Even for some very simple circuits, the representation can have either linear or exponential size depending on the order (finding an optimal ordering is known to be NP-complete). The long processing time at the beginning is due to heuristic methods that try to achieve a good ordering; after a while, a good ordering was reached, making processing faster. Even with those heuristics, we could not process large (> 300) domains. *Filter* and *Unroll* were also slower, and could not process long sequences or large domains (also, *Filter* can handle only a limited class of domains). *Unroll* suffers from the frame problem, i.e. needs to explicitly state the fluents that do not change ($f_t \leftrightarrow f_{t+1}$), and therefore depends on the domain size. *C-Filter*, however, managed to handle large domains (hundreds of thousands of fluents), taking a few milliseconds per step.

Figure 8 shows the time to find a model for the resulting circuits using a modified version of *NoClause*. Significantly, reasoning time grows only linearly with t. This allows practical logical filtering over temporal sequences of unbounded length. Note that the more observations an agent gets, the more constrained his belief state is. Therefore, it takes longer to find a satisfying model (also, the formula is larger).

7 Conclusions

A straightforward approach to filtering is to create all the prime implicates (or all consequences) at time $t+1$ from the belief state representation of time t. Previous work (e.g. [Liberatore, 1997]) showed that deciding if a clause belongs to the new belief state is coNP-complete, even for deterministic domains. This discouraged further research on the problem.

Nevertheless, in this work we presented an exact and tractable filtering algorithm for all deterministic domains. Our result is surprising because it shows that creating a representation of *all* of the consequences at time $t+1$ is easier (poly-time) than creating the new belief state piecemeal.

Several approaches were developed in the past to represent belief states in logic (e.g., BDDs, [Amir and Russell, 2003]), but none of them guaranteed compactness. The key to our advance was our *logical circuits* representation. We also showed how to maintain NNF-Circuits.

The results obtained here have implications in many important AI-related fields. We expect our algorithms to apply to planning, monitoring and controlling, and perhaps stochastic filtering. We plan to explore these directions in the future.

Acknowledgements This work was supported by the National Science Foundation CAREER award grant. We thank Rodrigo de Salvo Braz and the anonymous reviewers for their helpful comments.

References

[Amir and Russell, 2003] E. Amir and S. Russell. Logical filtering. In *IJCAI '03*. MK, 2003.

[Bertoli *et al.*, 2001] P. Bertoli, A. Cimatti, and M. Roveri. Heuristic search + symbolic model checking = efficient conformant planning. In *IJCAI '01*. MK, 2001.

[Biere *et al.*, 1999] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *DAC'99*, 1999.

[Bryant, 1992] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 1992.

[Cimatti and Roveri, 2000] A. Cimatti and M. Roveri. Conformant planning via symbolic model checking. *JAIR*, 2000.

[Clarke *et al.*, 2001] E.M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 2001.

[Ganai *et al.*, 2002] M. K. Ganai, P. Ashar, A. Gupta, L. Zhang, and S. Malik. Combining strengths of circuit-based and cnf-based algorithms for a high-performance sat solver. In *DAC*, 2002.

[Liberatore, 1997] P. Liberatore. The complexity of the language A. *ETAI*, 1997.

[Lin and Reiter, 1997] F. Lin and R. Reiter. How to progress a database. *AIJ*, 1997.

[Lind-Nielsen, 1999] J. Lind-Nielsen. Buddy - a binary decision diagram package. Technical report, Institute of Information Technology, Technical University of Denmark, 1999.

[Petrick and Bacchus, 2004] R.P.A. Petrick and F. Bacchus. Extending the knowledge-based approach to planning with incomplete information and sensing. In *ICAPS-04*. AAAI Press, 2004.

[Reiter, 1991] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In *Artificial Intelligence and Mathematical Theory of Computation*. Academic Press, 1991.

[Thiffault *et al.*, 2004] C. Thiffault, F. Bacchus, and T. Walsh. Solving non-clausal formulas with dpll search. In *Principles and Practice of Constraint Programming*, 2004.

[Winslett, 1990] M.A. Winslett. *Updating Logical Databases*. Cambridge U. Press, 1990.