Artificial Intelligence 216 (2014) 206–232

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Plan aggregation for strong cyclic planning in nondeterministic domains

Ron Alford [a,*], Ugur Kuter [c], Dana Nau [b], Robert P. Goldman [c]

[a] ASEE/NRL Postdoctoral Fellow, 4555 Overlook Ave., SW Washington, DC 20375, United States
[b] Department of Computer Science, Institute of Systems Research, and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, United States
[c] Smart Information Flow Technologies (d/b/a SIFT, LLC), 211 N. First St., Suite 300, Minneapolis, MN 55401, United States

ARTICLE INFO

Article history:
Received 5 December 2011
Received in revised form 3 July 2014
Accepted 17 July 2014
Available online 23 July 2014

Keywords:
Automated planning
Nondeterministic actions
Strong cyclic solutions

ABSTRACT

We describe a planning algorithm, NDP2, that finds strong-cyclic solutions to nondeterministic planning problems by using a classical planner to solve a sequence of classical planning problems. NDP2 is provably correct, and fixes several problems with prior work.
We also describe two preprocessing algorithms that can provide a restricted version of the symbolic abstraction capabilities of the well-known MBP planner. The preprocessing algorithms accomplish this by rewriting the planning problems, hence do not require any modifications to NDP2 or its classical planner.
In our experimental comparisons of NDP2 (using FF as the classical planner) to MBP in six different planning domains, each planner outperformed the other in some domains but not others. Which planner did better depended on three things: the amount of nondeterminism in the planning domain, domain characteristics that affected how well the abstraction techniques worked, and whether the domain contained unsolvable states.

1. Introduction

This paper is about a way to use classical planners to solve nondeterministic planning problems. Given a nondeterministic planning problem P and any classical planner CP, our NDP2 algorithm calls CP on a sequence of classical planning problems derived from P, and uses CP's solutions to construct a strong-cyclic solution for P. NDP2 is based on the NDP algorithm [30], but overcomes several problems with that prior work. Our contributions are as follows:

- NDP2 corrects two problems that NDP had with unsolvable states. Although NDP's pseudocode included a way to deal with unsolvable states by making modifications to the planning domain, its authors did not implement this part of NDP, and did not realize that it has two significant problems: (1) when it encounters unsolvable states, NDP modifies the domain model in a way that can make it exponentially larger, and (2) there are cases in which unsolvable states will cause NDP to generate incorrect solutions.
  NDP2 overcomes both of these problems. If NDP2 is used with a classical planner CP that is sound, complete, and guaranteed to terminate on classical planning problems, then NDP2 will be sound, complete, and guaranteed to terminate on nondeterministic planning problems.

* Corresponding author.
E-mail addresses: ronald.alford.ctr@nrl.navy.mil (R. Alford), ukuter@sift.net (U. Kuter), nau@cs.umd.edu (D. Nau), rpgoldman@sift.net (R.P. Goldman).

http://dx.doi.org/10.1016/j.artint.2014.07.007
0004-3702/

- Planners such as MBP [9], POND [7], and Yoyo [28] do not represent states in the usual classical fashion, but instead use binary decision diagrams (BDDs) [8] to represent sets of states that have common properties. This provides substantial efficiency gains, by enabling those planners to plan for large sets of states at once. To provide a limited form of MBP's state-abstraction capability within an ordinary classical state representation, the NDP paper [30] described a technique called "conjunctive abstraction" that involved modifying the planning domain to include "abstract states" that represent sets of ordinary states. Although [30] provided examples of such abstractions, it did not include an algorithm to produce them.
  We provide preprocessing algorithms to make two kinds of planning-domain modifications similar to conjunctive abstraction. When used as preprocessors to NDP2, these algorithms can sometimes provide state-abstraction abilities analogous to MBP's, and they preserve NDP2's ability to be used with any classical planner.
- We provide the results of experimental comparisons of NDP2 (using FF [21] as the classical planner) with MBP, on more than 4800 planning problems in six nondeterministic planning domains. Unlike [30], in which the experimental tests were limited to planning domains in which all states were solvable, three of our six domains include unsolvable states. Our experiments showed NDP2 outperforming MBP in some planning domains, and MBP outperforming NDP2 in others. Which algorithm performed better depended mainly on (1) the amount of nondeterminism in the search space, (2) how well the nondeterminism could be abstracted out (either by using our abstraction algorithms with NDP2, or by MBP using its BDDs), and (3) whether some of the nondeterministic outcomes could lead to unsolvable states.

This paper is organized as follows. Section 2 provides definitions and notation. Section 3 gives an algorithm for the case where all states are solvable, Section 4 extends the algorithm to deal with unsolvable states, and Section 5 motivates and describes our abstraction formalisms and algorithms. Section 6 provides the results of the experimental evaluations, Section 7 is a discussion of related work, and Section 8 is the conclusion. Appendix A contains the correctness proofs for NDP2, Appendix B describes techniques for translating solution policies from abstract to non-abstract domains, and Appendix C describes a case where NDP [30] is unsound.

2. Basic definitions and notation

Below, Sections 2.1 and 2.2 give definitions and notation for nondeterministic planning domains and classical planning, and Section 2.3 defines determinizations of nondeterministic domains.

2.1. Nondeterministic planning domains

A *nondeterministic planning domain* is one in which each action may have more than one possible outcome. Formally, it is a pair $D = (\mathcal{L}, O)$, where $\mathcal{L}$ is a function-free first-order language with finitely many constant symbols (hence finitely many ground atoms), and O is a finite set of *nondeterministic planning operators* as defined below.

We will represent states in the usual classical fashion: if $F = \{\text{all ground atoms of } \mathcal{L}\}$, then a *state* is a subset of F, and the set of all possible states is $\mathcal{S} = 2^F$. A literal l is true in s if l is a non-negated atom and $l \in s$, or if l is a negated atom $\neg\alpha$ and $\alpha \notin s$; otherwise l is false in s.

Each operator $o \in O$ is a pair

$$o = \big(\mathsf{pre}(o), \mathsf{effects}(o)\big),$$

where $\mathsf{pre}(o)$ is a conjunction of literals called o's *preconditions*, and $\mathsf{effects}(o)$ is a set of conjunctions of literals called o's *possible effects*. Intuitively, $\mathsf{pre}(o)$ describes what must be true in order to use o, and each conjunction in $\mathsf{effects}(o)$ describes one of the possible outcomes of using o. We sometimes will refer to o as $o(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are the variable symbols in o in some canonical order.

An *action* a is a ground instance of an operator o, and $\mathsf{pre}(a)$ and $\mathsf{effects}(a)$ are the corresponding ground instances of $\mathsf{pre}(o)$ and $\mathsf{effects}(o)$. If a is the action produced by replacing the variables in $o(x_1, \ldots, x_n)$ with constants $c_1, \ldots, c_n$, then we will sometimes refer to a as $o(c_1, \ldots, c_n)$. We will use $\mathcal{A}$ to denote the finite set of all possible actions, i.e., all possible ground instances of the operators in O. An action a is *executable* in any state that satisfies $\mathsf{pre}(a)$. For each state s, $A(s) \subseteq \mathcal{A}$ is the set of all actions that are executable in s.

Let $a \in A(s)$, and let $e_1, \ldots, e_n$ be the conjunctions in $\mathsf{effects}(a)$. For $i = 1, \ldots, n$, let $\gamma(s, e_i) = (s - e_i^-) \cup e_i^+$, where e_i^+ and e_i^- are the sets of atoms that appear positively and negatively in e_i. Then the *result* of executing a in s is the following set of states[1]:

$$\gamma(s, a) = \big\{\gamma(s, e_i)\big\}_{i=1}^{n} = \big\{\big(s - e_i^-\big) \cup e_i^+\big\}_{i=1}^{n}.$$

A *policy* is a function π that maps some of the states into actions, i.e., $\pi : S \to \mathcal{A}$ for some set of states $S \subseteq \mathcal{S}$. For each state-action pair $(s, a) \in \pi$, the intended meaning is that a is the action to perform in s. A *hyperpolicy* is a function π^* that

[1] When necessary to avoid ambiguity, we will write γ_D to refer to the value of γ in the planning domain D.

maps *sets* of states into actions, i.e., $\pi^* : \mathbf{S} \to \mathcal{A}$, for some set $\mathbf{S} \subseteq 2^{\mathcal{S}}$. For each pair $(S, a) \in \pi^*$, the intended meaning is that a is the action to perform in every state $s \in S$ (hence there is ambiguity about what action to perform if s is in more than one $S \in \mathbf{S}$). In the published literature on planning in nondeterministic environments, the solutions to planning problems are defined to be policies—but for purposes of computational efficiency, most of the better-known planning algorithms (e.g., [37,9,28,7]) reason instead about hyperpolicies, using Binary Decision Diagrams (BDDs) to represent the sets in $\mathbf{S}$.

The *π-descendants* of a state s are defined recursively as follows:

- s is a π-descendant of itself.
- If s' is a π-descendant of s and $\pi(s')$ is defined, then every $s'' \in \gamma(s', \pi(s'))$ is also a π-descendant of s.

A *π-result* of s is any π-descendant s' of s for which $\pi(s')$ is not defined (the intuition is that if we execute π starting at s and end up at s', then execution will cease). Thus we can define $\gamma(s, \pi) = \{s' \mid s' \text{ is a } \pi\text{-result of } s\}$. Note that as a special case, if $\pi = \emptyset$ then $\gamma(s, \pi) = \{s\}$. By extension, a π-result of a set of states S is any state that is a π-result of at least one of the states in S.

A *nondeterministic planning problem* is a triple $P = (D, S_0, G)$, where $D = (\mathcal{L}, O)$ is a nondeterministic planning domain. $S_0 \subseteq \mathcal{S}$ is a set of initial states, and $G \subseteq \mathcal{S}$ is a set of *goal states*. P may have different kinds of solutions [9,16]:

- A *weak solution* must provide a possibility of reaching a goal state, but doesn't need to guarantee that a goal state will always be reached. More specifically, a policy π is a weak solution if for every $s \in S_0$, some goal state $s_g \in G$ is a π-result of s.
- A *strong cyclic solution* is a policy π that has the following property: for every state s that is a π-descendant of S_0, there is a goal state $s_g \in G$ that is a π-result of s. Such a policy is guaranteed to reach a goal state in every *fair* execution, i.e., every execution that doesn't remain in a cycle forever if there's a possibility of leaving the cycle.
- P may also have *strong* solutions [9,16], but we will not need that definition in this paper.

A state $s \in S$ is *weakly solvable* if the planning problem $(D, \{s\}, G)$ has at least one weak solution, and *strong cyclically solvable* if $(D, \{s\}, G)$ has at least one strong cyclic solution. Otherwise s is *unsolvable*.

If every state that is reachable from S_0 is weakly solvable, then P is *everywhere weakly solvable*. Similarly, if every state that is reachable from S_0 is strong-cyclically solvable, then P is *everywhere strong-cyclically solvable*. The following lemma (the proof is in Appendix A) shows that the two terms are equivalent, so we will just say *everywhere solvable* instead.

Lemma 1. *A nondeterministic planning problem $P = (D, S_0, G)$ is everywhere weakly solvable iff it is everywhere strong cyclically solvable.*

2.2. Classical planning domains

An operator or action o is *classical* (or *deterministic*) if $\mathsf{effects}(o)$ contains just one conjunction of literals. A planning domain $D = (\mathcal{L}, O)$ is classical if every operator in O is classical. A planning problem $P = (D, S_0, G)$ is classical if D is classical and there is just one initial state, i.e., $S_0 = \{s_0\}$ for some $s_0 \in S$. In this case we will dispense with S_0 and write $P = (D, s_0, G)$.

For classical planning problems, solutions are conventionally defined to be sequential plans rather than policies. Formally, a *plan* is a sequence $p = \langle a_1, \ldots, a_k \rangle$ of classical actions. Given a state s_0, if there are states $s_1, \ldots, s_k$, such that for $1 \le i \le k$, $\gamma(s_{i-1}, a_i) = \{s_i\}$, then p is *executable* in s_0 and $\gamma(s_0, p) = s_k$. Given a planning problem $P = (D, s_0, G)$, a state s is *solvable* if there is a plan p such that $\gamma(s, p) \in G$. If s_0 is solvable then we say that P itself is solvable. If every state that is reachable from s_0 is solvable, then P is *everywhere-solvable*.

If a plan $p = \langle a_1, \ldots, a_k \rangle$ is executable at a state s_0, then p is *acyclic* at s_0 if each state $s_0, \ldots, s_k$ produced by executing p is unique (the plan does not traverse the same state twice). In this case, p corresponds to a unique policy $\pi = \{(s_0, a_1), (s_1, a_2), \ldots, (s_{k-1}, a_k)\}$ that we will call p's *policy image* at s_0.

2.3. Determinizations of nondeterministic domains

If $o = (\mathsf{pre}(o), \mathsf{effects}(o))$ is a nondeterministic operator and $\mathsf{effects}(o) = \{e_1, \ldots, e_n\}$, then the *determinization* of o is a set $\bar{o}$ of deterministic operators, one for each of o's possible effects:

$$\bar{o} = \big\{\big(\mathsf{pre}(o), e_1\big), \big(\mathsf{pre}(o), e_2\big), \ldots, \big(\mathsf{pre}(o), e_n\big)\big\}.$$

If an action a is a ground instance of o, then its determinization $\bar{a} = \{a_1, \ldots, a_n\}$ is defined similarly. The determinization of a nondeterministic planning domain $D = (\mathcal{L}, O)$ is a classical planning domain $\overline{D} = (\mathcal{L}, \overline{O})$, where $\overline{O} = \bigcup_{o \in O} \bar{o}$. The determinization of a nondeterministic planning problem $P = (D, \{s_0\}, G)$ is a classical planning problem $\overline{P} = (\overline{D}, s_0, G)$.

Lemma 2. *For every state s in a nondeterministic planning problem P, s is weakly solvable in P if and only if it is solvable in $\overline{P}$.*

Algorithm 1: NDPR, a planner for nondeterministic planning problems that are everywhere-solvable. (D, S_0, G) is the planning problem, and *CP* is the classical planner.

```
1  Procedure NDPR(D, S_0, G, CP)
2      π ← ∅; D̄ ← a determinization of D
3      loop
4          S ← {all non-goal π-results of S_0}
5          if S = ∅ then
6              return π
7          arbitrarily select a state s ∈ S
8          if CP(D̄, s, G) returns a solution plan ⟨ā_1, ..., ā_k⟩ then
9              Let a_1, ..., a_k be the nondeterministic versions of ā_1, ..., ā_k
10             for i = 1, ..., k do
11                 if π(s) is defined then remove (s, π(s)) from π
12                 insert (s, a_i) into π
13                 s ← γ_D̄(s, ā_i)
14                 if a goal state is a π-descendant of s then break
15         else
               // CP didn't find a solution, so either CP is incomplete or the planning problem is not
                  everywhere-solvable.
16             return Failure
```

The lemma is proved in Appendix A. From the lemma, it follows immediately that if a nondeterministic planning problem $P = (D, s_0, G)$ is everywhere-solvable, then its determinization $\overline{P} = (\overline{D}, \{s_0\}, G)$ also is everywhere-solvable.

3. Algorithm for everywhere-solvable planning problems

The clearest way to describe NDP2 is to start with an algorithm for a special case: planning problems that are everywhere-solvable. This section presents that algorithm; and in Section 4 we will extend the algorithm to deal correctly with unsolvable states.

Algorithm 1, NDPR, takes as input a nondeterministic planning problem $P = (D, S_0, G)$ and a classical planner *CP*. NDPR works by calling *CP* on problems of the form $(\overline{D}, s, G)$, and combining *CP*'s solutions into a solution for P. It is nearly identical to the NDP algorithm in [30], except that it omits NDP's faulty pseudocode for unsolvable states and it specifies exactly how to incorporate a plan into the policy (NDP left it unstated).

NDPR first initializes π to be the empty policy, and generates the determinization $\overline{D}$ of D (Line 2). Line 3 begins the main planning loop. If every π-result of S_0 is a goal state, then π is a strong cyclic solution, so NDPR returns it (Line 6). Otherwise, NDPR selects a π-result s of S_0 that is not a goal state, and uses *CP* to search for a plan that solves s in $\overline{D}$. If *CP* is incomplete or if P is not everywhere-solvable, then *CP* may fail to find a solution plan for $(\overline{D}, s, G)$; and in this case NDPR returns failure (Line 16). But if *CP* returns a solution plan p, then NDPR incorporates the actions of p into π (Lines 10–14) one at a time, stopping if it finds a state that π already weakly solves. Note that if *CP* is guaranteed to return acyclic solutions, then Line 11 can be omitted and the condition in Line 14 can be replaced with a check to see if $\pi(s)$ is defined.

Example. To illustrate how NDPR works, let D and $\overline{D}$ be the nondeterministic domain and its determinization as shown in Fig. 1 and Fig. 2. Consider the nondeterministic planning problem $P = (D, \{s_0\}, \{s_2\})$, in which the set of initial states is $\{s_0\}$ and there is a single goal state, s_2. In NDPR's first iteration, NDPR calls the classical planner on the problem $(\overline{D}, s_0, \{s_2\})$. Suppose that the classical planner returns the plan $\langle a_{12} \rangle$. NDPR will incorporate this plan into the currently empty policy π (Lines 10–14). As a result, s_1 is now a non-goal π-result of $\{s_0\}$.

In NDPR's second iteration, it will call the classical planner on $(\overline{D}, s_1, \{s_2\})$. Suppose the planner returns the plan $\langle a_2, a_{12} \rangle$. NDPR will incorporate the first action a_2, but then stop incorporating the plan at Line 14 since s_0 already has a weak solution. There are now no non-goal π-result of $\{s_0\}$ (the intuition is that all of the π-results of $\{s_0\}$ are goal states), and NDPR will exit on the next iteration (Line 6). □

4. Dealing with unsolvable states

Kuter et al. [30] described a way for NDP to deal with unsolvable states by removing state-action pairs from the domain. If *CP* returned failure on a state s, the idea was to take every state s' and action a such that $\pi(s') = a$ and $s \in \gamma(s', a)$ and modify the definition of a to make it inapplicable in s'. This requires modifying a's precondition to exclude s' without excluding any other states. Such a precondition will be a large disjunction that includes a positive or negative literal for every ground atom in the planning domain, and the number of ground atoms is often exponential in the size of the domain description. Thus NDP's way of dealing with unsolvable states often increases the size of the domain description—and the computational overhead of evaluating action preconditions—by an exponential amount.

In Section 4.1 we present ConstrainProblem, a procedure for modifying a classical planning problem $P = (\overline{D}, s, G)$ to make some of the actions inapplicable at the first step of any solution to P. Unlike removing state-action pairs,

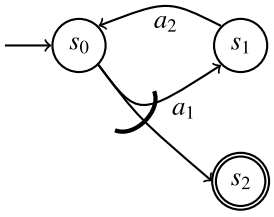

Fig. 1. Graphic depiction of a nondeterministic planning domain. Circles represent states, hyperedges represent actions.

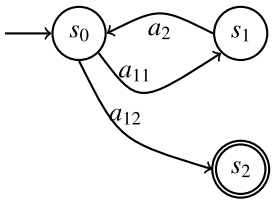

Fig. 2. Determinization of the planning domain in Fig. 1. The determinization of a_1 is $\{a_{11}, a_{12}\}$.

Algorithm 2: ConstrainProblem takes a planning problem $(\overline{D}, s, G)$ and a set of actions A, and returns a new planning problem P that has the same solutions minus the set of plans that start with an action in A.

```
1 Procedure ConstrainProblem(D̄, s, G, A)
2     D̄′ ← D̄; s′ ← s
3     foreach action o(c_1, ..., c_n) ∈ A do
4         s′ ← {disallowed_o(c_1, ..., c_n)} ∪ s′
5     foreach operator o(x_1, ..., x_n) ∈ D̄′ do
6         pre(o) ← (¬disallowed_o(x_1, ..., x_n)) ∧ pre(o)
7         foreach action a(c_1, ..., c_n) ∈ A do
8             effect(o) ← {¬disallowed_a(c_1, ..., c_n)} ∪ effect(o)
9     return (D̄′, s′, G)
```

ConstrainProblem only incurs a quadratic increase in the size of the domain description per constrained action. In Section 4.2 we present Find-Acceptable-Plan, a procedure that uses ConstrainProblem to search for an acyclic plan whose policy image avoids known unsolvable states. In Section 4.3 we present NDP2, a modified version of NDPR (see Section 3) that uses Find-Acceptable-Plan to avoid with known unsolvable states.

4.1. Restricting which actions are available

Algorithm 2 is the ConstrainProblem procedure, which takes a classical planning problem $(\overline{D}, s, G)$ and a set A of actions, and returns a new planning problem $(\overline{D}', s', G)$ for which a solution is any solution to $(\overline{D}, s, G)$ that does not start with an action in A.

For each operator $o(x_1, \ldots, x_n) \in O$, we introduce a new predicate $disallowed_o(t_1, \ldots, t_n)$. ConstrainProblem begins by creating a new state s' that is identical to s except that for each action $o(c_1, \ldots, c_n) \in A$, s' contains a new atom of the form $\mathsf{disallowed}_o(c_1, \ldots, c_n)$. Then, for each operator $o\overline{D}'$ with variables $x_1, \ldots, x_n$, ConstrainProblem adds $\neg\mathsf{disallowed}_o(x_1, \ldots, x_n)$ to o's preconditions (Line 6). This prevents any grounding of o with the constants $c_1, \ldots, c_n$ from being applicable whenever $\mathsf{disallowed}_o(c_1, \ldots, c_n)$ is true.

Finally, to the effects of each action, ConstrainProblem adds the negation of the disallowed predicates that it added to the initial state (Line 8). This ensures that $\neg\mathsf{disallowed}_a(\ldots)$ always holds after applying any action to the initial state.

4.2. Avoiding known-unsolvable states

We use ConstrainProblem in Find-Acceptable-Plan (Algorithm 3), which is used to construct acyclic plans whose nondeterministic images avoid known unsolvable states. Find-Acceptable-Plan's parameters consist of a nondeterministic planning domain D, its determinization $\overline{D}$, an initial state s_0, a set of goal states G, a classical planner CP, and a set U of states to avoid.

In Line 2, Find-Acceptable-Plan initializes five variables that it will maintain throughout its search: p is the current plan, S is a list of states associated with p, s is the last state in S, and B is a mapping from states to sets of actions known to lead to cycles or unsolvable states, and K is a set of states which can't be part of any solution.

Algorithm 3: Find-Acceptable-Plan takes a classical planning problem, classical planner, and a set U of states to avoid. It returns a plan for which no action's nondeterministic version can go to a state in U.

```
1  Procedure Find-Acceptable-Plan(D, D̄, s_0, G, CP, U)
2    s ← s_0; p ← ⟨⟩; S ← ⟨s_0⟩; B ← ∅; K ← U
3    loop
4      if s ∈ G then return p
5      A ← {action b(c_1, ..., c_n) such that (s, b(c_1, ..., c_n)) ∈ B}
6      P' ← ConstrainProblem(D̄, s, G, A)
7      call CP on the planning problem P'
8      if CP returns a solution plan ⟨ā_1, ..., ā_k⟩ then
9        foreach i = 1, ..., k do
10         a_i ← the nondeterministic action in D that corresponds to ā_i
11         s' ← γ_D̄(s, ā_i)
12         if s' ∈ S ∪ K or γ_D(s, a_i) ∩ U ≠ ∅ then
13           B ← B ∪ {(s, ā_i)}
14           break
15         append ā_i to p and append s' to S; s ← s'
16     else CP returns Failure
17       a' ← last element of p
18       remove last elements of p and S
19       if S = ⟨⟩ then return Failure
20       s' ← last element of S
21       B ← B ∪ {(s', a')}; K ← K ∪ {s}
22       s ← s'
```

In Lines 3–22, Find-Acceptable-Plan repeatedly calls *CP* to try and extend p towards a goal state without causing a cycle or choosing an action whose nondeterministic version leads to a state in U. In Line 4, Find-Acceptable-Plan checks if s, the last state of p, is a goal state and returns p if it is.

Otherwise, Find-Acceptable-Plan calls *CP* to generate a plan from s to a goal state. This requires overcoming two potential problems: (1) if the plan p generated by *CP* contains a cycle, then p cannot be translated into a policy because it will require two different actions at one of its states, and (2) if p goes through a state in U, then it cannot be translated into a policy that solves the nondeterministic problem $(D, \{s\}, G)$, since the states in U are known to be unsolvable. Find-Acceptable-Plan makes progress by ensuring that *CP* never returns a plan that starts with an action it has seen before. First, it finds the set of actions in B associated with the current state that are known to cause loops or lead to states in U (Line 5). Find-Acceptable-Plan then takes the classical problem $P = (\overline{D}, s, G)$, and uses ConstrainProblem to create a new planning problem P' for which these actions cannot appear in the first step of a solution (Line 6). Find-Acceptable-Plan then calls the classical planner *CP* on this modified problem (Line 7). If *CP* is sound and complete, there are two cases:

Case 1: *CP* returns a plan $q = \langle \overline{a_1}, \ldots, \overline{a_k} \rangle$ that leads to a goal state. Then in Lines 9 through 15, Find-Acceptable-Plan iterates over the actions, adding them to the current plan p and updating the current state s. If $\overline{a_i}$ leads to a state already seen in p or to a known-unsolvable state in K, or if its nondeterministic counterpart a_i leads to a state in U, then Find-Acceptable-Plan inserts $(s, \overline{a_i})$ into B and stops integrating q (without removing already integrated actions). Planning will restart at the state just before a_i. If the plan is fully integrated, then, assuming *CP* is sound, s is a goal state, and Find-Acceptable-Plan will return p on the next iteration of the main loop.

Case 2: *CP* cannot find a plan, and returns failure. Then Find-Acceptable-Plan backtracks to the previous state s' in S and adds the state-action (s', a) pair leading to s to B, and adds s to the set of known-unsolvable states K (Line 21). If there is no previous state, then this means that there is no plan starting from s_0 and reaching a goal state whose policy image doesn't lead to U, so Find-Acceptable-Plan returns Failure (Line 19).

Example. Let D and $\overline{D}$ be as in Fig. 1 and Fig. 2, and consider the call to Find-Acceptable-Plan$(D, \overline{D}, s_0, \{s_1\}, CP, \{s_2\})$. The current state s will be set to s_0, the list of states S set to $\langle s_0 \rangle$.

With B initially empty, the call to ConstrainProblem$(\overline{D}, s_0, \{s_1\})$ will return an identical classical problem $P' = (\overline{D}, s_0, \{s_1\})$ (Fig. 3). When Find-Acceptable-Plan calls *CP* on P', it will return the plan: $\langle a_{11} \rangle$. Since the nondeterministic action corresponding to a_{11} leads to s_2 (which is in U), Find-Acceptable-Plan will break before incorporating the first action of the plan, and add the pair (s_0, a_{11}) to B.

On the next iteration of Find-Acceptable-Plan, a_{11} will be in the set A of actions to constrain. ConstrainProblem$(\overline{D}, s_0, \{s_1\}, \{a_{11}\})$ will return a classical problem $(\overline{D}', s_0', \{s_1\})$, with a new initial state s_0' that has the same set of applicable actions as s_0, except for a_{11} (Fig. 4). Since this problem is unsolvable, *CP* returns failure. Hence Find-Acceptable-Plan also returns failure. □

Theorem 1. *Let CP be a sound and complete classical planner, U be a set of states, D be a nondeterministic planning domain, and $\overline{D} = (\mathcal{L}, \overline{O})$ be the determinization of D. Let S be the set of all states in $\mathcal{L}$ (i.e., $S = 2^F$), $F = \{$all ground atoms over $\mathcal{L}\}$, and $\mathcal{A}$ be the set of all possible actions (i.e., all possible ground instantiations of the planning operators in O).*

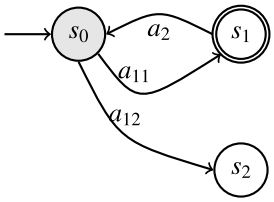

Fig. 3. ConstrainProblem($\overline{D}, s_0, s_1, \emptyset$), identical to the determinization shown in Fig. 2.

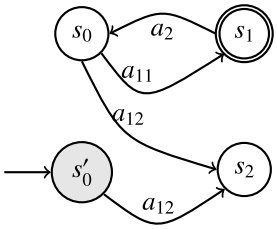

Fig. 4. ConstrainProblem($\overline{D}, s_0, s_1, \{a_{11}\}$), with a new initial state s'_0.

Algorithm 4: NDP2 is a modified version of NDPR that works correctly in all nondeterministic planning domains. The call to *CP* in Line 8 is replaced with a call to Algorithm 3, which uses *CP* to look for plans that do not include nodes that NDP2 has identified as unsolvable.

```
1  Procedure NDP2(D, S_0, G, CP)
2    π ← ∅
3    D̄ ← a determinization of D ; U ← ∅
4    loop
5      S ← {all non-goal π-results of S_0}
6      if S = ∅ then return π
7      arbitrarily select a state s ∈ S
       // Find-Acceptable-Plan searches for an acyclic plan in D̄ that avoids the states in U
8      call Find-Acceptable-Plan(D, D̄, s, G, CP, U)
9      if Find-Acceptable-Plan returns a solution plan ⟨ā_1, ..., ā_k⟩ then
10       Let ⟨a_1, ..., a_k⟩ be the nondeterministic actions corresponding to ⟨ā_1, ..., ā_k⟩
11       for i = 1, ..., k do
12         if π(s) is defined then remove (s, π(s)) from π
13         insert (s, a_i) into π
14         s ← γ_D̄(s, ā_i)
15         if a goal state is a π-descendant of s then break
16     else
         // Find-Acceptable-Plan returned Failure
17       if s ∈ S_0 then return Failure
18       U ← U ∪ {s}
19       foreach s' such that s ∈ γ(s', π(s')) do
20         π ← π \ {(s', π(s'))}
```

Then Find-Acceptable-Plan($D, \overline{D}, s_0, G, CP, U$) *makes at most* $|S| \cdot |A| + 1$ *calls to CP, and returns an acyclic plan, if such a plan exists, whose policy image in D avoids the states in U.*

For the proof, see Appendix A.

4.3. NDP2 *planning algorithm*

Algorithm 4 is the NDP2 algorithm, a modified version of NDPR that can deal with unsolvable states. The key differences between NDP2 and NDPR are:

- When NDP2 encounters an unsolvable state, it removes all actions that lead to it from the policy and adds the state to a set of known-unsolvable states U (Line 18).
- NDP2 does not call the classical planner directly, but instead calls Find-Acceptable-Plan, which generates solutions that avoid the states in U.

There are also two key differences between NDP2 and NDP:

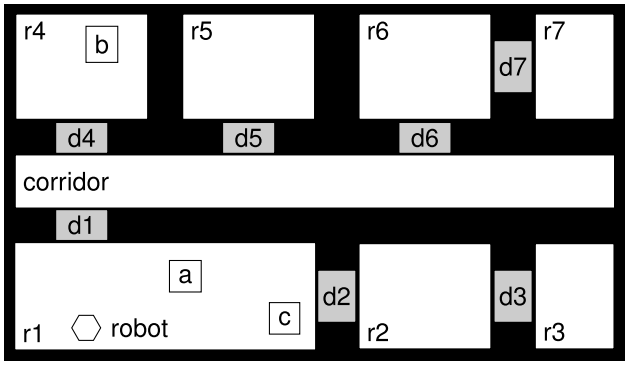

Fig. 5. A state in the Robot Navigation domain.

- NDP removed state-action pairs directly from the domain instead of using Find-Acceptable-Plan. There are potentially doubly-exponentially many states in the size of the domain [12], meaning possibly doubly-exponential increase in the size of the determinization of the nondeterministic planning domain. Even in the case that a single state is removed from the domain, identifying a state out of a set S must take on average $\log|S|$ space, increasing the size of the determinized domain by an exponential amount in the size of the domain.
- As discussed in Appendix C, NDP used a plan integration routine which is unsound on problems with unsolvable states. NDP2 does not have this problem.

Theorem 2. *Let CP be a sound and complete classical planner and $P = (D, S_0, G)$ be a nondeterministic planning problem with $D = (\mathcal{L}, O)$. Let S be the set of all states in $\mathcal{L}$ (i.e., $S = 2^F$), $F = \{$all ground atoms over $\mathcal{L}\}$, and $\mathcal{A}$ be the set of all possible actions (i.e., all possible ground instantiations of the planning operators in O).*

Then NDP2(D, S_0, G, CP) *is sound and complete, and returns at most in $|S|^2$ calls to* Find-Acceptable-Plan.

For the proof, see Appendix A.

5. Abstractions and compound abstractions

In nondeterministic planning domains, some major representation and reasoning problems can occur if each action has a very large number of possible outcomes. Probably the best-known example of this is the Robot Navigation domain [25,37,9,28,29], which is illustrated in Fig. 5. In this domain, there is a building with several rooms, and a robot that needs to go among these rooms to deliver packages. To go into or out of a room, the robot may need to open a door, and there is a child (the "kid") who can interfere with this by running around very quickly, nondeterministically opening and closing some of the doors. This problem can be translated into a single-agent nondeterministic planning problem by representing the kid's actions as nondeterministic outcomes of the robot's actions [25,9].

In a Robot Navigation domain with k "kid doors" (i.e., doors that the kid can open and close), each of the robot's actions can have 2^k possible outcomes: one for each possible combination of open and closed kid doors. If a planner has to plan for each of these outcomes separately, then this causes an exponential blowup in the amount of space needed to represent the plan, and the amount of time needed to compute it.

Planners such as MBP [9], POND [7], and Yoyo [28] tackle this problem by using BDDs [8] to represent and reason about *sets* of states rather than individual states. For example, consider the problem of finding a policy π that will move the robot through door d1 in Fig. 5, regardless of which kid doors are open and which ones are closed. This policy will need to contain 2^k state-action pairs: one for each possible combination of open and closed kid doors. But in each of the 2^k states, the only thing that matters is whether d1 is open or closed. A planner that uses a BDD-based state representation can generate a much smaller *hyperpolicy* (see Section 2) such as

$$\pi^* = \{(S_1, a_1), (S_2, a_2)\}, \tag{1}$$

where $S_1 =$ {all states in which the robot is in r1 and the door is open}, $S_2 =$ {all states in which the robot is in r1 and the door is closed}, a_1 is the action of moving the robot from r1 to the corridor, and a_2 is the action of opening the door.

It is not feasible for NDP2 to use a similar BDD representation. That would require extensive modifications to NDP2's classical planner *CP*, which conflicts with the objective of allowing *CP* to be *any* classical planner. However, we sometimes can get some of the same benefits, without having to modify *CP* at all, by preprocessing the planning domain D to produce an *abstracted* planning domain D^* in which some of the states represent sets of states in D. Once this has been done, NDP2 can be called on D^* rather than D.

For example, if D^* contains two "abstract states" that represent the sets S_1 and S_2 above, then in D^*, NDP2 can plan how to go through d1 with only two calls to *CP*. In this case, the solution to the planning problem is the same hyperpolicy π^* as in Eq. (1), but with S_1 and S_2 represented by abstract states rather than BDDs.

The *conjunctive abstraction* techniques in [30] were an initial version of that approach. However, that work did not include a formal definition of conjunctive abstraction, and all of the modifications to the states and planning operators were done manually. This left it unclear how or whether the approach could be generalized, and whether it could be done automatically. In the following subsections, we develop an approach similar to conjunctive abstraction; but we define it formally and provide pseudocode for the translations.

5.1. Language and states

Let $D = (\mathcal{L}, O)$ be a nondeterministic planning domain, and let $\mathcal{L}^*$ be an augmented version of $\mathcal{L}$ such that for every predicate symbol p of $\mathcal{L}$, $\mathcal{L}^*$ includes both p and a new predicate symbol p^* of the same arity as p. If $\alpha = p(c_1, \ldots, c_n)$ is any ground atom of $\mathcal{L}$, then $\alpha^* = p^*(c_1, \ldots, c_n)$ is the atom produced from α by replacing p with p^*. We will call α^* an *abstraction* of α and $\neg\alpha$, because its purpose is for use in representing states in which α may be either true or false.

If s is a state and $\alpha \notin s$, then according to the usual classical planning semantics, α is false in s and α is true in the state $s \cup \{\alpha\}$. If we let $s' = s \cup \{\alpha^*\}$, then s' is an *abstract state* that is intended to represent both of the states s and $s \cup \{\alpha\}$. More generally:

Definition 1. If is a state and $A = \{\alpha_1, \ldots, \alpha_k\}$ is a set of ground atoms that are not in s, then $s' \cup \{\alpha_1^*, \ldots, \alpha_k^*\}$ is an *abstract state*, and the set of states that are *represented* by s' is $\{s \cup A' \mid A' \subseteq A\}$. We let $[s']$ denote this set of states.

There is an important difference between abstract states and the belief states used in partially observable planning problems. If an abstract state s' contains the atom α^*, it does not mean that α's truth value will be unknown at execution time. Instead, s' represents a set of fully observable states in which α may be either true or false, so that we can plan for these states simultaneously.

Example. In the Robot Navigation domain, consider all states in which the robot and the packages are at the locations shown in Fig. 5, and all doors are closed except that d6 and d7 may each be either open or closed. There are four such states:

$$s_1 = \{\mathsf{in(r1)}, \mathsf{loc(a, r1)}, \mathsf{loc(b, r4)}, \mathsf{loc(c, r1)}, \mathsf{open(d6)}, \mathsf{open(d7)}\}; \tag{2}$$

$$s_2 = \{\mathsf{in(r1)}, \mathsf{loc(a, r1)}, \mathsf{loc(b, r4)}, \mathsf{loc(c, r1)}, \mathsf{open(d6)}\}; \tag{3}$$

$$s_3 = \{\mathsf{in(r1)}, \mathsf{loc(a, r1)}, \mathsf{loc(b, r4)}, \mathsf{loc(c, r1)}, \mathsf{open(d7)}\}; \tag{4}$$

$$s_4 = \{\mathsf{in(r1)}, \mathsf{loc(a, r1)}, \mathsf{loc(b, r4)}, \mathsf{loc(c, r1)}\}. \tag{5}$$

Let

$$s^* = \{\mathsf{in(r1)}, \mathsf{loc(a, r1)}, \mathsf{loc(b, r4)}, \mathsf{loc(c, r1)}, \mathsf{open^*(d6)}, \mathsf{open^*(d7)}\}. \tag{6}$$

Then the set of all states represented by s^* is $\{s_1, s_2, s_3, s_4\}$. □

5.2. Operators with abstract effects

We now will describe a way to rewrite planning operators to produce abstract states.

Let o be any operator in D, and let $E = \mathsf{effects}(o)$. Suppose that two of the conjunctions in E are $e_1 = e \wedge \alpha$ and $e_2 = e \wedge \neg\alpha$, where e is a (possibly empty) conjunction of literals and α is a literal not in e. In other words, one possible effect of o is to make both e and α true, and another possible effect of o is to make e true and α false. Then we can define the *abstraction of E over* $\{e_1, e_2\}$ to be the set of conjunctions

$$E' = \big(E - \{e_1, e_2\}\big) \cup \{e \wedge \alpha^* \wedge \neg\alpha\}.$$

The reason for including $\neg\alpha$ in this equation is because we will want to use E' for the effects of an abstract operator, and we need to ensure that such an operator will work correctly when executed in a state s that contains α. Recall that the intended meaning of α^* is to assert that we are in an abstract state in which α may be either true or false, hence it would be inconsistent for the abstract state to also contain an assertion that α is true.

We can perform the abstraction process iteratively, abstracting E' over a pair of conjunctions to get E'', abstracting E'' over another pair of conjunctions to get E''', and so forth until we get an abstraction E^* of E that is *maximal* (i.e., E^* cannot be abstracted any further).

In general, there may be more than one maximal abstraction of E. Algorithm 5 is a simple greedy algorithm to compute one of them (we do not care which one). After the following example, we will define an *abstract operator* whose effects are E^*.

Algorithm 5: Compute a maximal abstraction of a set of conjunctions.

```
1 procedure Create-Abstract-Conjunction(E)
2   while there is an abstractable pair of conjunctions e1, e2 ∈ E
3     E ← the abstraction of E over {e1, e2}
4   return E
```

Example. Consider a Robot Navigation problem in which there are two kid doors, d6 and d7. Here is a nondeterministic action a to open d1 when the robot is in room r1:

$$\mathsf{pre}(a) = \mathsf{in}(\mathsf{r1}) \wedge \neg\mathsf{open}(\mathsf{d1}), \tag{7}$$

$$\mathsf{effects}(a) = \{e_1, e_2, e_3, e_4\}, \tag{8}$$

where

$$e_1 = \mathsf{open}(\mathsf{d1}) \wedge \neg\mathsf{open}(\mathsf{d6}) \wedge \neg\mathsf{open}(\mathsf{d7});$$

$$e_2 = \mathsf{open}(\mathsf{d1}) \wedge \neg\mathsf{open}(\mathsf{d6}) \wedge \mathsf{open}(\mathsf{d7});$$

$$e_3 = \mathsf{open}(\mathsf{d1}) \wedge \mathsf{open}(\mathsf{d6}) \wedge \neg\mathsf{open}(\mathsf{d7});$$

$$e_4 = \mathsf{open}(\mathsf{d1}) \wedge \mathsf{open}(\mathsf{d6}) \wedge \mathsf{open}(\mathsf{d7}).$$

If we let $E = \mathsf{effects}(a)$, then E can be abstracted three times. The first abstraction is to replace e_1 and e_2 with

$$e_5 = \mathsf{open}(\mathsf{d1}) \wedge \neg\mathsf{open}(\mathsf{d6}) \wedge \mathsf{open}^*(\mathsf{d7}) \wedge \neg\mathsf{open}(\mathsf{d7}),$$

the second one is to replace e_3 and e_4 with

$$e_6 = \mathsf{open}(\mathsf{d1}), \mathsf{open}(\mathsf{d6}) \wedge \mathsf{open}^*(\mathsf{d7}) \wedge \neg\mathsf{open}(\mathsf{d7}),$$

and the third one is to replace e_5 and e_6 with

$$e_7 = \mathsf{open}(\mathsf{d1}) \wedge \mathsf{open}^*(\mathsf{d6}) \wedge \neg\mathsf{open}(\mathsf{d6}) \wedge \mathsf{open}^*(\mathsf{d7}) \wedge \neg\mathsf{open}(\mathsf{d7}).$$

This produces the maximal abstraction

$$E^* = \{\mathsf{open}(\mathsf{d1}) \wedge \mathsf{open}^*(\mathsf{d6}) \wedge \neg\mathsf{open}(\mathsf{d6}) \wedge \mathsf{open}^*(\mathsf{d7}) \wedge \neg\mathsf{open}(\mathsf{d7})\}. \tag{9}$$

Definition 2. If is a planning operator (or an action), then an *abstraction* of o is an operator (or action) o^* such that

1. $\mathsf{pre}(o^*)$ is the result of modifying $\mathsf{pre}(o)$ by replacing each negative literal $\neg\alpha$ with the conjunction $\neg\alpha \wedge \neg\alpha^*$;
2. $\mathsf{effects}(o^*)$ is a maximal abstraction of $\mathsf{effects}(o)$.

The reason for replacing negative literals with conjunctions in $\mathsf{pre}(o^*)$ is to prevent o^* from being applied in cases where applying it would be unsound. It is not necessary to replace positive literals with conjunctions, because no state will ever contain both α and α^*.

Example. Let s_1, s_2, s_3, s_4, s^* be as in Eqs. (2)–(6), and a be as in Eqs. (7)–(8). Then the following action a^* is an abstraction of a:

$$\mathsf{pre}(a^*) = \mathsf{in}(\mathsf{r1}) \wedge \neg\mathsf{open}(\mathsf{d1}) \wedge \neg\mathsf{open}^*(\mathsf{d1})$$

$$\mathsf{effects}(a^*) = \{\mathsf{open}(\mathsf{d1}) \wedge \mathsf{open}^*(\mathsf{d6}) \wedge \neg\mathsf{open}(\mathsf{d6}) \wedge \mathsf{open}^*(\mathsf{d7}) \wedge \neg\mathsf{open}(\mathsf{d7})\}. \tag{10}$$

Thus $\gamma(s_1, a^*) = \{s^*\}$.

In a^*'s precondition, the literal $\neg\mathsf{open}^*(\mathsf{d1})$ prevents a^* from being applied to abstract states where applying it would be unsound, such as the following state:

$$s^{**} = \{\mathsf{in}(\mathsf{r1}), \mathsf{loc}(\mathsf{a}, \mathsf{r1}), \mathsf{loc}(\mathsf{b}, \mathsf{r4}), \mathsf{loc}(\mathsf{c}, \mathsf{r1}), \mathsf{open}^*(\mathsf{d1}), \mathsf{open}^*(\mathsf{d6})\}.$$

Before a^* can be applied, $[s^{**}]$ must first be split into two subsets: the states that satisfy $\mathsf{open}(\mathsf{d1})$ and the states that don't. a^* will be applicable to the second subset but not the first one. □

To provide a means for splitting abstract states into subsets, we will define, for each predicate symbol p of D, a *splitting* operator split-p such that

$$\text{pre}(\text{split-}p) = \{p^*(x_1, \ldots, x_n)\};$$
$$\text{effects}(\text{split-}p) = \{\neg p^*(x_1, \ldots, x_n) \wedge p(x_1, \ldots, x_n),\ \neg p^*(x_1, \ldots, x_n)\};$$

where n is the arity of p.

Example. Continuing the previous example, the operator split-open(x) has

$$\text{pre}(\text{split-open}) = \{\text{open}^*(x)\};$$
$$\text{effects}(\text{split-open}) = \{\neg\text{open}^*(x) \wedge \text{open}(x),\ \neg\text{open}^*(x)\}. \tag{11}$$

Thus, split-open(d1) will split s^{**} into a pair of abstract states: one in which d1 is open, and one in which it is closed:

$$s_1^* = \{\text{in(r1)}, \text{loc(a, r1)}, \text{loc(b, r4)}, \text{loc(c, r1)}, \text{open(d1)}, \text{open}^*\text{(d6)}\};$$
$$s_2^* = \{\text{in(r1)}, \text{loc(a, r1)}, \text{loc(b, r4)}, \text{loc(c, r1)}, \text{open}^*\text{(d6)}\}. \quad \square$$

Note that although splitting operators resemble nondeterministic planning operators syntactically, their semantics are quite different: they do not correspond to actions in D, and their possible outcomes do not model nondeterminism in D. Instead, they simply perform bookkeeping operations for the purpose of translating sets of states (represented as abstract states) back into ordinary states, and they do not appear in the solution policies returned by NDP2.

Definition 3. An abstraction of a nondeterministic planning domain D is a planning domain D^* in which the set of operators is $O^* \cup \Sigma$, where

- O^* contains an abstraction of each planning operator o in D;
- Σ contains a splitting operator split-p for every predicate p in $\mathcal{L}$ such that p^* appears in the effects of at least one operator $o \in O^*$.

By extension, if $P = (D, S_0, G)$ is a planning problem in D, then we will call $P^* = (D^*, S_0, G)$ an *abstraction* of P.

Since a solution π^* to an abstracted problem represents a hyperpolicy, it is possible to extract an ordinary policy π from it. Algorithm 7 in Appendix B is an algorithm for doing this. The basic idea is quite similar to a policy-extraction algorithm that is provided with the MBP planner—and just as with MBP's policy-extraction algorithm, which is almost never used, there is no real need for Algorithm 7. Given any state s, finding the action to perform in s is no harder to do with π^* than with π, and in domains such as the Robot Navigation domain, π^* is much easier to use since π is exponentially larger.

5.3. Compound abstractions

In order to create abstract planning problems, we modified the planning operators' effects to produce abstractions of pairs of literals. But the preconditions of each planning operator still referred to the original literals rather than the abstract ones, making it necessary to use splitting operators to map some of the abstract literals back to the original literals before applying the planning operator. When certain conditions are satisfied, it is possible to modify some of the planning operators' preconditions to refer directly to the abstract literals, removing the need for the splitting operators. We provide an algorithm to do this.

Let $P^* = (D^*, S_0, G)$ be an abstraction of a planning problem $P = (D, S_0, G)$. Let Σ and O^* be the sets of splitting operators and planning operators in P^*. A splitting operator split-$p \in \Sigma$ is *compoundable* with a planning operator $o \in O^*$ if the following conditions hold:

- p occurs exactly once in pre(o), in a non-negated atom α;
- Each conjunction $e \in$ effects(o) contains at most one of α, $\neg\alpha$, and α^*, and no other atom in effects(o) is unifiable with α or α^*;
- p does not appear in G.

For each $o \in O^*$, we let Σ_o be the set of all splitting operators that are compoundable with o. For each splitting operator split-$p \in \Sigma_o$, we let the *compound* operator split-$p \cdot o$ be an operator whose precondition is pre(o) with α replaced by α^*, and whose effects are effects(o) with the following modifications:

Algorithm 6: Compute a compound abstraction of a planning problem. Σ is the set of splitting operators in D^*, and O^* is the set of non-splitting planning operators in D^*.

```
1 Procedure Create-Compound-Operators(O*, Σ)
2   O** ← O*
3   foreach planning operator o ∈ O* do
4     B ← {all abstract predicates in pre(o)}
5     foreach set of abstract predicates B' ⊆ 2^B do
6       o** ← split-p_1 · split-p_2 · ... · split-p_|B'| · o, where |B'| is the size of B' and p_1, p_2, ..., p_|B'| ∈ B'
7       insert o** into O**
8   return O**
```

- for each effect $e \in \text{effects}(o)$ that does not contain α^* or $\neg\alpha$, replace e with $e \wedge \neg\alpha^* \wedge \alpha$;
- for each effect $e \in \text{effects}(o)$ that contains $\neg\alpha$, replace $\neg\alpha$ with $\neg\alpha^*$;
- add an additional effect $\neg\alpha \wedge \neg\alpha^*$, to represent the case where split-p's nondeterministic outcome is $\neg\alpha \wedge \neg\alpha^*$ (whence o is inapplicable).

Example. Let a^* be as in (10), and split-open be as in (11). Then split-open(d1) and a^* are not compoundable, because pre(a^*) contains ¬open(d1) rather than open(d1). But consider the following action b^*, which is an abstraction of an action for exiting room r1 through door d1:

$$\text{pre}(b^*) = \text{in(r1)} \wedge \text{open(d1)};$$
$$\text{effects}(b^*) = \{\neg\text{in(r1)} \wedge \text{in(hall)} \wedge \text{open}^*\text{(d6)} \wedge \text{open}^*\text{(d7)}\}.$$

In most Robot Navigation problems, the goal G consists entirely of package locations, so that the open predicate does not occur in G, whence split-open(d1) is compoundable with b^*. The compound operator split-open(d1) $\cdot\, b^*$ has

$$\text{pre}(\text{split-open(d1)} \cdot b^*) = \text{in(r1)} \wedge \text{open}^*\text{(d1)};$$
$$\text{effects}(\text{split-open(d1)} \cdot b^*) = \{\neg\text{in(r1)} \wedge \text{in(hall)} \wedge \text{open}^*\text{(d6)} \wedge \text{open}^*\text{(d7)} \wedge \neg\text{open}^*\text{(d1)} \wedge \text{open(d1)},$$
$$\text{in(r1)} \wedge \neg\text{open(d1)} \wedge \neg\text{open}^*\text{(d1)}\}. \qquad \square$$

If two splitting operators split-p and split-q are both compoundable with o, then it is not hard to show that split-p is compoundable with split-$q \cdot o$.

If $o \in O^*$, and if $\Sigma' = \{\text{split-}p_1, \ldots, \text{split-}p_k\}$ is an ordered set of splitting operators that are compoundable with o, then we will define

$$\Sigma' \cdot o = \text{split-}p_1 \cdot \text{split-}p_2 \cdot \ldots \cdot \text{split-}p_k \cdot o.$$

We will define

$$O^{**} = \{\Sigma' \cdot o \mid o \in O^* \text{ and } \Sigma' \subseteq \Sigma_o\},$$

where $\Sigma_o = \{\text{all splitting operators in } \Sigma \text{ that are compoundable with } o\}$, and where we assume an arbitrary sequential order on the operators in each subset Σ' of Σ_o. Thus O^{**} includes all of the compound operators and all of the operators in O^*. Let Σ_{NC} be the set of all splitting operators that are *non-compoundable*, i.e., each split-$p \in \Sigma_{NC}$ either p appears in the goal G or p is in the precondition of some operator o but is not compoundable with o. Then the planning problem $P^{**} = (\mathcal{L}, O^{**} \cup \Sigma_{NC}, G)$ is a *compound-abstract* version of P^*.

Algorithm 6 is a high-level description of our procedure to automatically create compound abstractions of planning operators given a set of abstract predicates and the splitting operators for those predicates in the planning domain. For each planning operator o in the planning domain, Algorithm 6 first generates all of the abstract predicates that appear in the preconditions of o in the set B (Line 4). For each subset B' of B, the algorithm then creates a compound operator o^{**} from the predicates in that subset and the planning operator o. The rationale behind considering every subset is for the sake of completeness: the compound abstraction must create a planning operator for each possible case where some of the literals in o's precondition are abstracted and the rest are not.

For each abstract predicate p in B', Algorithm 6 first finds the splitting operator for p and then creates a compound abstraction of o with that splitting operator (Line 6). The compound operator o^{**} is then inserted into the set of operators to be returned by the algorithm (Line 7).

Algorithm 8 in Appendix B is an algorithm to translate a compound-abstract solution π^{**} for P^{**} into an abstract policy π^*. The basic idea is quite simple; for each action in π^{**} that is compound, the algorithm separates it into its two component pieces (a splitting action and an ordinary abstract action). By first running Algorithm 8 and then running Algorithm 7, one could extract an ordinary policy π. However, as we pointed out at the end of Section 5.2, there is no real need to do this since the abstract policy π^* is easier to work with.

6. Experimental evaluation

We implemented NDP2 in Common Lisp, and compared it experimentally with MBP [9] in six fully-observable nondeterministic planning domains that were chosen to present a variety of different issues for the planners to deal with. As NDP2's classical planner in these experiments, we used FF [21], since it was the classical planner that had worked best with NDP [30].

For each planning domain, we tested the planners on a large suite of randomly-generated test problems of multiple sizes, for a total of about 4500 planning problems. We ran both NDP2 and MBP on Intel Xeon 2.33 GHz processors running Red Hat Enterprise Linux 5.5. We gave both planners 2 hours and 2 GB of RAM to complete each planning problem.

We attempted to broaden our comparison to include POND [7] and GAMER [11], but were unable to do so. POND does not support planning problems that require cyclic solutions. In GAMER we ran into several implementation issues that prevented it from creating proper ground versions of our problems. Thus, despite very helpful discussions with the authors of these planners, we were not able to run them on the problems in our test suite.

Three of the planning domains are everywhere-solvable, and all three of them are well-known from previous experimental studies:

- In the Robot Navigation domain with 7 kid doors (see Section 5), each action has 2^7 possible outcomes. Thus in order to avoid a huge combinatorial explosion in the search space, it is essential for the planner to partition the states into a small number of classes and plan over those classes, rather than reason about each state individually. MBP's BDD representation enables it do such reasoning quite well in this domain [37,27], and we wanted to see whether our abstraction techniques would work well enough to make NDP2 competitive with MBP.
- In the Hunter–Prey domain [2,10], each action has roughly 5^n outcomes, where n is the number of prey. Thus, although the number of locations are polynomial, the amount of nondeterminism for the hunter after each of its move increases combinatorially with the number of prey in the domain. Our abstraction techniques do not work in this domain, and we wanted to see how this would affect NDP2's performance.
- In the Nondeterministic Blocks World [26], reasoning over sets of states is not particularly useful, but there is a large number of goal interactions (e.g., deleted-condition interactions) to deal with. Many classical planners are good at reasoning about such interactions, and we wanted to see if NDP2 could take advantage of this.

In everywhere-solvable planning domains, NDP2 calls its classical planner at most once per reachable state, because the classical planner (assuming it is complete) will never return failure. But in planning domains that contain unsolvable states, NDP2 may need to call the classical planner many times per state. To see how this would affect NDP2's performance, we compared NDP2 with MBP on three planning domains that contained many unsolvable states:

- The Exploding Blocks World has been used in several of the International Planning Competitions, e.g., [44,6]. For most planning problems in this domain, the solution must include actions that would be redundant in any solution to the determinized version of the problem; and since the classical planner is unlikely to generate plans that include those redundant actions, the classical planner will usually return plans that lead to unsolvable states in the original problem P. But this difficulty is mitigated by the small branching factor of the nondeterminism: unlike the competition version of this domain, we only had one explosive block.
- The Tire World has also been used in several of the International Planning Competitions, e.g., [44,6]. Like the Exploding Blocks World, it requires solutions whose actions are redundant in the determinization. On one hand, Tire World has fewer available actions per state than the Exploding Blocks World; but on the other hand, the size of the smallest solution to Tire World problems grows exponentially with the number of locations in the domain.
- Lost in Space is a new domain that we developed to test NDP2's subroutines for avoiding unsolvable states. For planning problems in this domain, the shortest solution for the determinized planning problem almost always leads to unsolvable states in nondeterministic planning problem. Thus NDP2 must repeatedly modify its determinization of the planning domain, in order to force the classical planner to avoid using any of the problematic actions.

In the Robot Navigation domain, we tested the planners on the problems in our test suite, and also on abstract and compound-abstract versions of the same problems. We used the translation algorithms (Algorithms 5 and 6) to generate these versions of the problems. We did not bother to develop computer implementations of those algorithms, but instead ran them by hand.

For the other planning domains in our experiments, we did not run separate experiments on abstract and compound-abstract versions of the problems, because those versions of the problems are identical to the original versions. The abstraction and compound-abstraction techniques modify a planning operator only when some of the operator's nondeterministic outcomes differ by a single literal—and in those domains, every pair of nondeterministic outcomes differ by more than one literal.

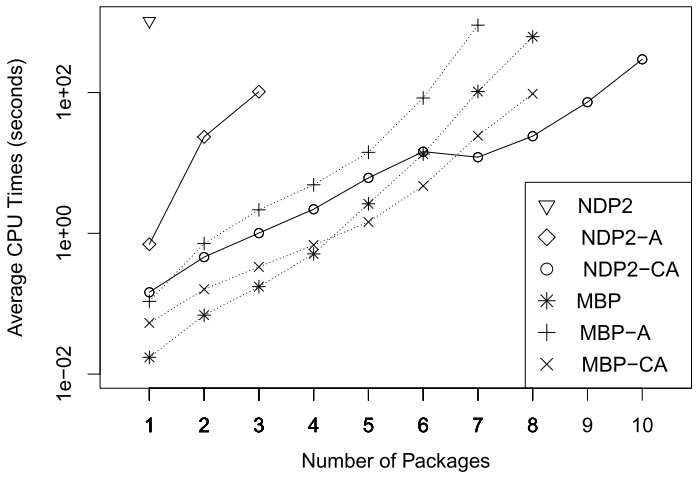

Fig. 6. Average CPU times on Robot Navigation problems with 7 kid doors, as a function of the number of packages.

6.1. Experiments with everywhere-solvable domains

Robot Navigation [9] The first set of experiments were in the Robot Navigation domain described previously, with $k = 7$ (i.e., all 7 doors were kid doors). We varied the number of packages n from 1 to 10. For each value of n, we measured each planner's average CPU time on 100 randomly-generated problems. As in [38], MBP's CPU times include both its preprocessing and search times. Omitting the former would not have significantly affected the results, because the preprocessing times were never more than a few seconds, and usually below one second.

In addition to testing the algorithms on Robot Navigation problems, we also tested them on abstract and compound-abstract versions of the problems. We used the translation algorithms (Algorithms 5 and 6) to generate these versions, performing these algorithms manually rather than running them on the computer. Those algorithms are easy to perform by hand; and furthermore, the Robot Navigation domain was the only one of our experimental domains in which we needed to use them. In all of the other planning domains in our experiments, the abstract and compound-abstract versions are identical to the original domain.

Fig. 6 shows the average CPU times for all cases where a planner solved all 100 problems (a meaningful average is impossible if the planner solves some of the problems but not all of them). The labels MBP and NDP2 are for the original planning problems, MBP-A and NDP2-A are for the abstract versions of the problems, and MBP-CA and NDP2-CA are for the compound-abstract versions of the problems. We discuss the results below.

MBP did worse on the abstract versions of the problems than on the original problems, because the splitting operators increased the branching factor of MBP's search space by creating branches in MBP's BDD structure in places where MBP would not ordinarily have created branches. MBP did better on the compound-abstract problems than the abstract ones, because the compound operators alleviated the search-space blowup caused by the splitting operators.

Surprisingly, MBP did better on the compound-abstract versions of problems with 5 or more packages than on the original versions of those problems. This puzzles us, but we suspect the compound-abstraction helped MBP to focus its search on parts of the search space that were relevant for finding a solution.

On the original planning problems, where NDP2 had to reason about each of the 2^7 outcomes of each action, its performance was quite poor. It solved all of the 1-package problems, and some of the 2- and 3-package problems, but no problems larger than that. As we had hoped, NDP2 did better on the abstracted versions of the problems: it solved all of the problems with 3 or fewer packages, and some problems with 4 to 7 packages. But this was still much worse than MBP's performance, and we believe it is because FF's hill-climbing algorithm returned plans with extraneous split actions that produced needless branches in the policy.

In the compound-abstract planning problems, NDP2 did dramatically better: it completed problems with up to 10 packages, and it outperformed MBP on problems with 7 or more packages. In the original problems, NDP2 had to call FF roughly 2^7 times for every step of the initial weak solution—but in the compound-abstract problems, NDP2's number of calls to FF was less than twice the number of steps in the initial weak solution.

Hunter–Prey [2,10] In this domain, the world is an $n \times n$ grid in which a hunter wants to catch one or more prey. The hunter has five possible actions; move north, south, east, or west, and catch (the latter is applicable only when the hunter and prey are in the same location). Each prey has also five actions: the four movement actions plus a stay-still action. Like the kid in the Robot Navigation domain, the prey are not represented as separate agents: instead, their possible actions are encoded as nondeterministic outcomes of the hunter's actions.

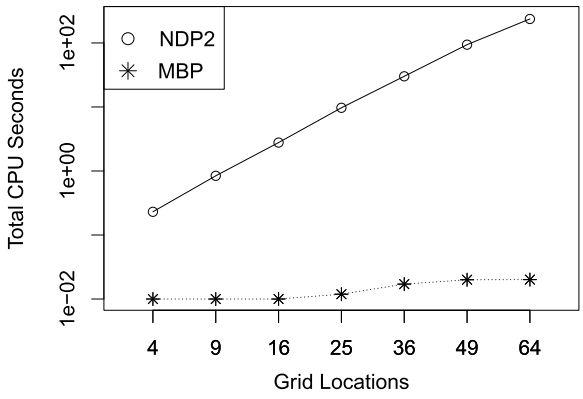

Fig. 7. Average CPU times in seconds in Hunter–Prey problems with one prey, as a function of grid size.

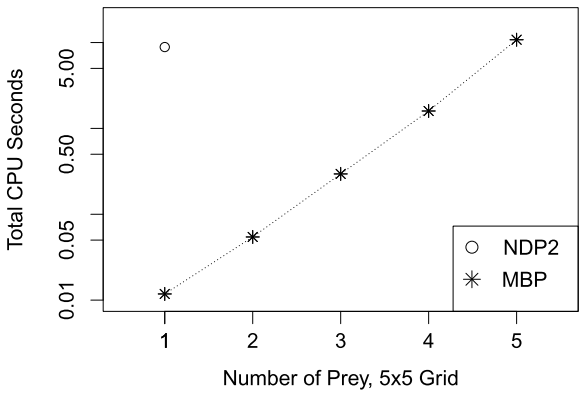

Fig. 8. Average CPU times in seconds in Hunter–Prey problems as a function of the number of prey, on a 5×5 grid.

Fig. 7 shows running times when there is just one prey and the grid size varies from 2×2 to 8×8, and Fig. 8 shows running times when the grid size is fixed at 5×5 and the number of prey varies from 1 to 5. Each data point is the average of 100 randomly generated problems.

MBP's running times were quite good, because MBP's BDDs did quite well at compressing the search space.[2] By the nature of the domain, any strong cyclic policy must cover most of the problem's reachable states, yet MBP could use a single BDD to represent the set of all states in which the hunter needed to move in a particular direction.

In contrast, NDP2 had to reason about each of those states separately. When there was just one prey, the number of states, and thus NDP2's running time, grew polynomially with the number of locations. But the number of states grew exponentially with the number of prey, so NDP2 did not solve any problems with more than one prey.

Nondeterministic Blocks World [26] The nondeterministic Blocks World is like the classical Blocks World, except that an action may have three possible outcomes: (1) the same outcome as in the classical case, (2) the block slips out of the gripper and drops on the table, and (3) the action fails completely and the state does not change. Neither of the abstraction techniques can be used in this domain, for the same reason as in the Hunter–Prey domain.

Fig. 9 shows the planners' average CPU times in this domain, as a function of the number of blocks. Each data point represents the average running time on 100 random problems. NDP2 outperformed MBP for three reasons:

1. There were no large sets of states that could be clustered together; hence MBP's BDD-based representation could not make much difference.
2. MBP did not exploit the heuristics used in the classical planners, hence MBP searched most of the state space in most planning problems.
3. Every action has three outcomes, but they are structured so that at least one of them (and often two) lead to a state already seen by the planner. Thus the number of calls NDP2 must make to FF scales linearly with the number of blocks in the problem.

[2] MBP did not work as well on Hunter–Prey problems in [27,29], because those papers used a version of the Hunter–Prey domain in which prey could not occupy adjacent squares—a restriction that interfered greatly with the effectiveness of MBP's BDDs. We did not use such a restriction in this paper.

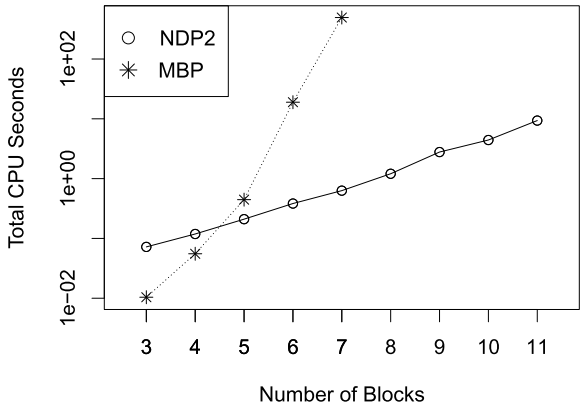

Fig. 9. Average CPU times in the nondeterministic Blocks World, as a function of the number of blocks.

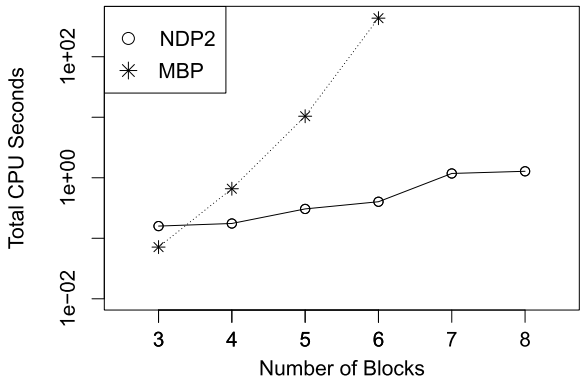

Fig. 10. Average CPU times in Exploding Blocks World, as a function of the number of blocks.

6.2. Planning domains with unsolvable states

Exploding Blocks World [6] The nondeterministic Exploding Blocks World is much like the classical Blocks World except that there may be one or more exploding blocks, which may or may not destroy the table or block underneath them when they are put down. In order for a problem to have a solution, there must be enough accessible spare blocks to defuse the exploding blocks. In any solution, a spare block must be uncovered and placed on the table before an exploding block is moved. Then the exploding blocks must be repeatedly placed on the spare until it explodes, making it safe to move the exploding block elsewhere.

Fig. 10 shows the completion times for each planner when there is a single exploding block and a single spare block, with a total of $n+1$ blocks in the initial state, and n blocks in the goal state. There were 100 planning problems for each number of blocks between 3 and 8. As with the previous blocks world variant, NDP2 outperformed MBP for all but the smallest problems. Most likely, MBP performed poorly in the exploding blocks domain for much the same reasons it performed poorly with nondeterministic blocks world problems, that is the lack of a heuristic function and lack of clusterable states.

In the exploding blocks world, the only nondeterministic actions are actions that move unexploded blocks. Thus the amount of nondeterminism is lower than in the nondeterministic blocks world, so we might expect NDP2 to perform much better than it did on the nondeterministic blocks world problems. However, moving an exploding block before defusing it with the spare block leads to an unsolvable state, and there is no reason for FF to avoid this sequence of events. Even when an exploding block is in hand and a spare block is clear and on the table, there are as many actions available in the initial state which lead to unsolvable states as there are clear blocks, and NDP2 may need to rule out each action in turn. Somewhat surprisingly, the relative lack of nondeterminism balances out with the propensity to find unsolvable states, and NDP2 performs similarly in both the exploding and nondeterministic blocks world variants, despite vastly different structures in their nondeterminism.

Tire World [6] Our Tire World variant consists of a triangular grid of connected places with tires interspersed between them, and the goal is to move the car from the initial location to a goal location. The car may get a flat tire after every move, meaning the car must carry a spare tire, and replace it once it is consumed. In our experiments, we added tires at random to the initial state until the problem was solvable.

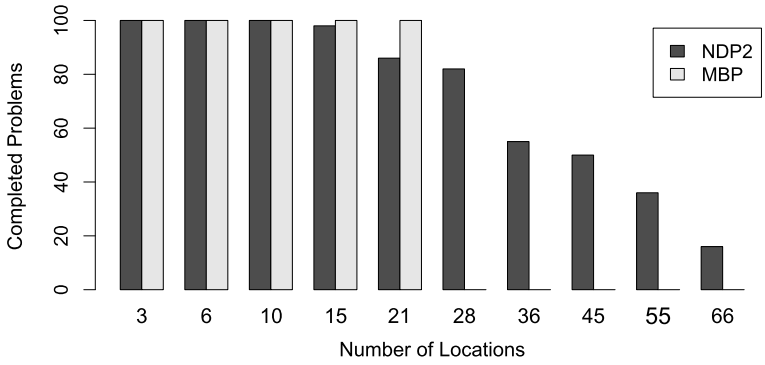

Fig. 11. Completed problems (out of 100) in Triangle Tire World, as a function of the number of locations.

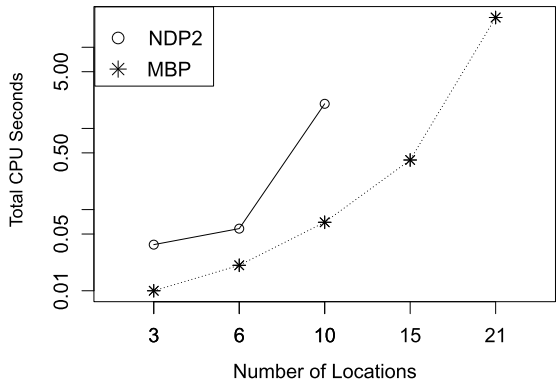

Fig. 12. Average CPU time in Triangle Tire World, as a function of the number of locations, for the cases (see Fig. 12) where the planner completed all 100 problems of that size.

A before, we tested the planners on 100 problems of each size. Fig. 11 shows how many problems of each size the planners solved, and Fig. 12 shows the average CPU times on the problem sizes in cases where the planners solved all problems of that size. MBP solved every problem with 3 to 21 locations, and generally solved these problems faster than NDP2; but did not solve any problem with more than 21 locations. NDP2 solved all problems with 3 to 10 locations, most of the problems with 15 to 45 locations, and some problems with up to 66 locations.

In each Tire World problem, the number of states in the smallest strong cyclic solution is exponential in the length of the shortest solution to the determinized problem. Furthermore, many times the shortest determinized solution leads through an area where there would not be enough spare tires if any flats occur, hence the nondeterministic domain contains an exponential number of unsolvable states (not all of which are immediately apparent).

This means NDP2's running time is potentially doubly-exponential due to the number of calls it must make to *CP*: exponential in the length of the shortest determinized solution, and exponential in the difference in length between the shortest successful path to the goal if no flat tires occur, and the length of the shortest successful path to the goal if a flat tire occurs at every move. Consequently, for the problems of sizes 15 and 21, there were few problems that NDP2 did not solve within the time limit, even though MBP solved all 100 problems of each size. This is why Fig. 12 contains data points for MBP but not NDP2 at those sizes.

On the other hand, many of the problems have solutions that differ only slightly from the shortest path, and NDP2's performance is "only" exponential in the length of that path, and so NDP2's indirect use of FF's heuristic function enabled it to solve some of the planning problems all the way up to size 66, even though MBP could not solve any problems larger than size 21.

Lost in space As we mentioned earlier, our original purpose in developing the Lost in Space (LiS) domain was to test NDP2's subroutines for avoiding unsolvable states. But the domain has another property that made it useful for our experiments: the domain is simple enough that we can use it to gauge the worst case performance of the Find-Acceptable-Plan subroutine.

A planning problem instance in the LiS planning domain is a simple line of locations, with the agent at one end and the goal at the other. The solution to an LiS planning problem is a policy that moves the agent from its initial location to the goal. The agent can move between locations by using one of two actions: walking between connected locations; and teleporting between any two locations, which can succeed or leave the agent lost and unable to move. This means that for a problem with n locations, there are $n+1$ states, n^2+2n-2 actions, and a single correct policy. Since the teleport action

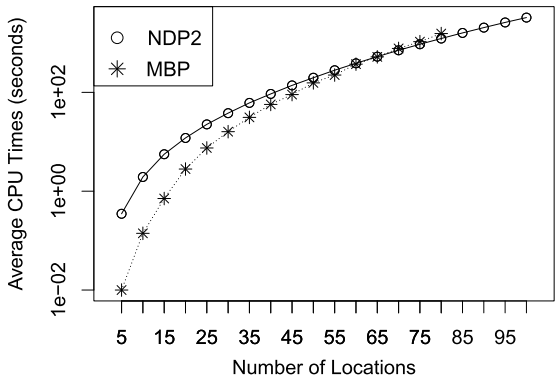

Fig. 13. Average CPU times of MBP and NDP2 on problems (out of 20) in Lost in Space, as a function of the number of locations.

in our determinization of LiS always leads to the goal, FF will almost always return plans that use it. Thus NDP2 should have to make $O(n^3)$ calls to the classical planner to develop a policy for an LiS planning problem.

We ran both NDP2 and MBP 20 times on each of 20 problem instances with 5 to 100 locations. There is only one problem instance for each problem size in the LiS planning domain, but we ran the algorithms 20 times on each instance in order to reduce statistical variations in the running times—especially the running times of NDP2's calls to FF, which makes some random choices that cause its running time to vary.

NDP2 was able to solve all problems. MBP did not solve problems with more than 80 locations. In addition to the results above, we also report the average CPU times of the planners in our experiments. Fig. 13 shows the average CPU time for each planner per size of problem. As expected, FF consistently used the determinized version of teleport for every state until ConstrainProblem removed that option. Both NDP2 and MBP showed sub-exponential growth of CPU time in the number of locations, though NDP2 has a slower growth rate, overcoming its initial disadvantage for problems with 70 or more locations.

6.3. Summary and discussion of the experimental results

Here is a quick summary of the results in each domain, along with our understanding of the reasons for those results:

- In the Robot Navigation domain, the amount of nondeterminism was extremely high. Here, NDP2's performance against MBP depended on how good a way we gave it to deal with the nondeterminism. Without abstraction, it did quite badly. With ordinary abstraction it did a little better, and with compound abstraction it did much better.
- In the Hunter–Prey domain, our abstraction techniques weren't applicable, so we couldn't give NDP2 a way to deal with the nondeterminism in this domain. Consequently, NDP2 did badly.
- In the Nondeterministic Blocks World and the Exploding Blocks World domains, the amount of nondeterminism was relatively small, and FF's search heuristics worked well. Thus NDP2 did much better than MBP.
- In the Triangle Tire World domain, NDP2's performance on each problem depended on whether the plans returned by FF contained "bad" actions (i.e., actions that looked good in the determinized domain but led to unsolvable states in the nondeterminized domain). Consequently, NDP2's performance is in some ways better than MBP's (e.g., how many problems it could solve), and in some ways worse than MBP's (e.g., the amount of CPU time it used).
- What happened in the Lost in Space domain was similar to what happened in the Triangle Tire World domain. But in this case, the number of bad actions in this domain is much smaller, so NDP2 did much better overall.

7. Related work

7.1. Using a classical planner as a "black box"

There have been several other works that proposed to use classical planning algorithms as a "black box" to generate solutions for non-classical planning problems. The most notable one is FF-Replan [42], which uses the FF planner [21] to first generate a plan (i.e., a weak policy) for a *determinization* of a Markov Decision Process (MDP). *Markov Decision Processes (MDPs)* are like nondeterministic planning domains in the sense that each action can have more than one possible outcome, but they differ from the latter in that each possible outcome of an action has a probability attached to it; costs and rewards are attached to the actions and states, respectively.

FF-Replan introduced several determinization strategies for probabilistic PDDL actions; among which, *all-effects* determinization is the basis for our determinization mechanism in NDP2. While FF-Replan is an on-line replanning algorithm that ensures a single execution to be realized and only works for everywhere solvable planning problems, NDP2 generates, offline, a solution for all possible outcomes of the nondeterminism in the execution and it can deal with planning problems

that are not everywhere solvable. Both approaches have different advantages and disadvantages as discussed in the literature several times previously. NDP2 differs from FF-Replan in several ways: NDP2 can use any classical planning algorithm, unmodified; it does offline generation of a complete solution policy rather than online generation of a single execution trace; and it finds strong cyclic solutions in nondeterministic domains.

There are several relatively recent MDP planners that use a classical planner (typically FF) as a black-box. An example for this class of MDP planners include RFF [41]. Like FF-Replan, RFF uses FF to generate weak plans. It then runs several Monte-Carlo simulations to determine the probability of the execution of a policy π ending in a non-goal π-result of an initial state. RFF then uses FF to generate weak plans from those states, integrates them into the policy, and reruns the Monte-Carlo simulations. RFF repeats this process until the probability of an execution failing falls below a fixed parameter. Although both RFF and NDP2 can handle dead-ends in planning domains and incrementally build the policy, NDP2 does so by explicitly and symbolically reasoning about them; RFF does so by reasoning about failure probabilities. Thus, each planner has access to different kinds of knowledge and models.

FF-Hindsight [43] is also an MDP planner inspired by FF-Replan which uses FF as part of its heuristic and generates weak solutions to a planning problem. For each state evaluated, FF-Hindsight creates sets of time-varying classical planning problems and uses FF to find which portions are solvable. This is an optimistic measure of how likely it is that the agent can reach the goal from this state. FF-Hindsight then uses the solvability estimate to pick which action is most likely to lead to the goal. Although the idea of using heuristics to generate execution paths in FF-Hindsight is similar to that of FF-Replan and NDP2, FF-Hindsight differs from NDP2 in that it does not generate strong cyclic solution policies.

There are also other approaches for planning with nondeterministic actions based on the idea of classical planners. The work described in [1] is also for non-probabilistic settings, but it's aimed for contingency planning in partial-observable domains. The work of [34] uses determinizations of probabilistic actions and use classical planners to generate sequences of actions for execution.

FIP [14] is a recent NDP-inspired planner which shows a number optimizations that can be done if the classical planner is treated not as a black box, but as a glass box, directly incorporated into the planner. Optimizations include directly removing state-action pairs from the domain (eliminating the need for Find-Acceptable-Plan), preferring deterministic operators, and stopping the search for a weak plan when a solved state is found. An additional optimization, the goal-alternative search, would be easy to implement in NDP2, but it requires an additional call to Find-Acceptable-Plan, which is already the bottleneck in most of our experiments. Since FIP is based on NDP, it has NDP's plan incorporation bug described in Appendix C, but it should be straightforward to incorporate our fix for this bug into FIP.

Another recent work, described in [33], has made incremental extensions to some of the ideas in NDP. This work introduces a definition of solution quality, and PrP looks for policies that are optimal according to that definition. Another difference is that PrP's implementation is based on the SAS+ formalism, whereas NDP2's implementation uses a non-probabilistic version of PPDDL.

The planner described in [22] generates cyclic solutions to partially observable planning problems by successively producing linear plans (i.e., weak policies) and combining those plans into a conditional and cyclic plan, in a way similar to our work, However, this work cannot use classical planners as NDP2 does; instead, it requires substantially rewriting those planners for bookkeeping for policy generation.

The GAMER planner [11] translates nondeterministic problems into a PDDL-like language for describing two-player games and uses a game solver to find a solution. GAMER performed well in ICAPS-08 planning competition, but a bug in its grounding process prevented us from running it in our experiments.

7.2. Other planning techniques for nondeterministic planning domains

Probably the first work on planning in fully-observable nondeterministic domains is described in [15], which is a breadth-first search algorithm over an AND–OR tree. Other early works on fully-observable nondeterministic domains include the Cassandra planning system [39], CNLP [36], Plinth [18], and UCPOP [35], and QBFPlan [40]. However, all these works describe a special-purpose planning algorithm for nondeterministic planning domains, and thus, do not focus on using classical planners as a black box.

One of the earliest attempts to use model-checking techniques for planning under nondeterminism was first introduced in the SimPlan planner of [25]. SimPlan is based on model checking techniques that work over explicit representations of states in the state space; i.e., the planner represents and reasons explicitly about every state visited during the search. Symbolic model-checking approaches to planning in nondeterministic domains were first introduced in [17,9]. MBP is one of the best planners that uses Binary Decision Diagrams (BDDs) for this purpose.

UMOP [23,24] exploits some of the ideas from the MBP planner, as a starting point for multi-agent planning, and combines BDDs with a heuristic-search algorithm for strong and strong cyclic planning [24]. Heuristic search provides some performance improvements over unguided BDD-based planning, such as in MBP on some simpler examples than MBP was tested on. We have not compared UMOP to NDP2 in this paper because of this reason; the authors of UMOP discussed and suggested some possibilities for scaling their approach to larger problems.

ND-SHOP2 [26] uses HTN planning techniques to control the search space in nondeterministic planning. ND-SHOP2 showed how HTN knowledge could improve nondeterministic planning performance, and performed competitively with MBP. Yoyo [29] extended this line of work by combining HTN planning with a compact BDD state representation to get

several orders of magnitutde in performance gains over ND-SHOP2 and MBP. Both of these planners use domain-specific planning knowledge to organize the search space while generating solution policies. Unlike them, NDP2 relies solely on the classical planner's domain-independent heuristic search capabilities.

Planners such as MBP [4], POND [7] and Contingent-FF [20] can generate solution policies for partially observable planning problems. Most of them cannot generate cyclic solutions, except for an extended version of MBP [3], which can generate strong cyclic solutions to a class of partially observable problems. We believe the ideas in NDP2 could also be generalized to partial observability.

Finally, [13] reports an approach for analyzing deterministic planning domains and identifying structural features and dependencies among those features using model-checking techniques. Although this approach has some similarities to our pairwise effect abstraction technique, their approach focusses on using the results of a domain analysis to prune the search space whereas we use pairwise effect abstractions for state-space compression. It would be interesting to investigate as a future work if the domain analysis method can be used for identifying more general and effective features for state compression.

7.3. A final note on MDPs

MDP problems for control theory and operations research do not usually include a notion of goal states; when they do, they are usually formulated as *stochastic shortest-path (SSP)* problems. See [32] for an excellent survey of MDP planning and planning techniques from an AI perspective.

In SSPs, every action has nonzero probabilities for all of its outcomes, whence the probability that we'll never leave the cycle is zero. Algorithms for solving SSP problems attempt to compute a policy that will achieve the goals with probability 1 [31]. Note also that this property is analogous to the "fairness" assumption in strong-cyclic solutions in nondeterministics planning domains [9] (and as also defined in Section 2.1 in this paper).

SSPs can be solved either by MDPs or by nondeterministic planning models, and the planners using the latter have been shown empirically to be more efficient on such problems [5]. The primary reason is that planners that use nondeterministic models do less search than MDP planners because they are not looking for optimal solutions.

8. Conclusions

NDP2, like the earlier NDP algorithm [30], solves nondeterministic planning problems by calling a classical planner on a sequence of deterministic planning problems, and using the classical planner's plans to construct a strong cyclic solution policy for the nondeterministic problem. However, in order to avoid NDP's difficulties with unsoundness and combinatorial explosion in the presence of unsolvable states, NDP2 has a different (and provably correct) way of dealing with unsolvable states.

We also have provided algorithms to translate a planning problem P into two different "abstract" versions of P in which there are states that represent sets of P's states. These overcome another limitation of [30], which described a similar "conjunctive abstraction" technique without providing an algorithm to compute it. The well-known MBP planner uses BDDs to compute abstractions that are significantly more powerful than ours—but since our abstractions do not use BDDs, they preserve NDP2's ability to be used with any classical planner.

NDP2's primary advantage over MBP is that MBP uses none of the sophisticated search heuristics used in classical planners, hence can sometimes visit many more states than it needs to. Since NDP2 uses a classical planner as a subroutine, the classical planner's search heuristics can sometimes help NDP2 to visit significantly fewer states than MBP. This happened in the Robot Navigation domain and the Nondeterministic Blocks World, where NDP2 did much better than MBP.

NDP2's main disadvantage compared to MBP is that in many of the cases where MBP's BDDs can represent a set of states as a single abstract state, our abstraction algorithms cannot do so. Thus there are cases where NDP2 must plan for different states separately but MBP can plan for the entire set of states at once. This happened in our Hunter–Prey experiments, where MBP performed much better than NDP2.

Our experimental results with Exploding Blocks World, Tire World, and Lost in Space showed that NDP2's technique for avoiding unsolvable states works quite well: NDP2 completed nearly every problem that MBP completed, and many more that MBP could not complete. In the Exploding Blocks World and Lost in Space domain, where both planners completed enough problems to compare speed, NDP2 completed large problems much faster than MBP.

Future work Since MBP's BDD-based abstractions give it an advantage in some cases, and NDP2's access to classical search heuristics gives it an advantage in other cases, it might be possible to obtain better performance than both MBP and NDP2 by writing an NDP2-like planner that incorporates an FF-like algorithm operating over BDDs, or by finding other ways to combine BDDs and relaxed planning graphs. Existing work such as [7] has already investigated ways to combine planning graphs and BDDs, but these approaches typically require complicated and potentially exponential representations due to the mutex conditions in planning graphs, which degrade the abstraction capabilities in BDDs. Since FF's relaxed planning graphs do not include mutex conditions, they might be a better fit for BDDs.

Further improvements may also be achievable by coupling NDP2 and FF more tightly. When NDP2 calls FF, it must wait until FF reaches a goal. If we could intervene to stop FF as soon as it reaches a state that is already part of NDP2's current

partial solution, this would provide a substantial speedup because it would prevent FF from wasting time retracing large parts of the solutions that it found during the previous times NDP2 called it.

We note that some MDP planning algorithms (e.g., LAO* [19]) can generate cyclic solution policies. With proper modifications to these planners and their inputs, it would be interesting to compare them with NDP2 and classical planners. This may provide a path toward developing an NDP2-like algorithm for MDPs.

Acknowledgements

This work was supported in part by ARO grant W911NF1210471, ONR grants N000141210430 and N000141310597, and a UMIACS New Research Frontiers Award. Ron Alford performed part of this work while supported by an ASEE postdoctoral fellowship at the U.S. Naval Research Laboratory. The information in this paper does not necessarily reflect the position or policy of the funders, and no official endorsement should be inferred.

Appendix A. Theoretical properties

This appendix provides the theoretical properties of the NDP2 planning procedure, its subroutines Find-Acceptable-Plan and ConstrainProblem, and the abstraction and compound abstraction techniques. Most of the lemmas in this appendix are not mentioned in the body of the paper, but they are used in the proofs of the theorems.

Lemma 1. *A nondeterministic planning problem $P = (D, S_0, G)$ is everywhere weakly solvable iff it is everywhere strong cyclically solvable.*

Proof. ($\Rightarrow$): Let $s_0, \ldots, s_k$ be the set of states reachable from s_0. Since P is everywhere weakly solvable, let $p_0, p_1, \ldots, p_k$ be a set of weak solutions for each s_i.

Let π_0 be the policy formed by setting $\pi_0(s) = a$ for every $(s, a) \in p_0$. By construction, if $\pi_0(s)$ is defined, there is a goal π_0-descendant of s.

Let π_{i+1} be the policy formed by setting $\pi_{i+1}(s) = \pi_i(s)$ for every state s such that π_i is defined, and $\pi_{i+1}(s) = a$ for every state such that $\pi_i(s)$ is not defined and $(s, a) \in p_{i+1}$. Again, by construction, every state for which π_{i+1} is defined has a goal π_{i+1}-descendant.

Since π_k-descendants of s_0 are a subset of the states reachable from s_0, every π_k descendant of s_0 has a path to the goal, and π_k is a strong cyclic solution to P.

($\Leftarrow$): Suppose P is everywhere strong cyclicly solvable and let s be a state in P reachable from S_0. If P were not everywhere weakly solvable, then there would exist at least one state s that is reachable from s but there is no path from s to a goal. But this is a contradiction by definition of strong cyclic solutions. □

Lemma 2. *For every state s in a nondeterministic planning problem P, s is weakly solvable in P if and only if it is solvable in $\overline{P}$.*

Proof. Let $P = (D, \{s_0\}, G)$ be a nondeterministic planning problem, and $\overline{P}$ be a determinization of P.

($\Rightarrow$): Let s be a state in D and suppose $(D, \{s\}, G)$ has a weak solution $\pi = \{(s_{i-1}, a_i)\}_{i=1}^{n}$ where $s_0, \ldots, s_n$ is the sequence of states produced by π in D. Let $p = \langle a'_1, \ldots, a'_{n+1} \rangle$ be a plan such that each a'_i is a determinization of a_i and $\gamma(s_{i-1}, a'_i) = \{s_i\}$ since π is a weak solution for $(D, \{s\}, G)$. By construction, the plan p is a solution for the classical planning problem $(\overline{D}, s, G)$.

($\Leftarrow$): Suppose $\overline{P}$ has a solution plan $p = \langle a_1, \ldots, a_n \rangle$. It follows that from the way determinizations are constructed, π is a weak solution for P: if a_i is applied in the state s_{i-1} in p, then $(s_{i-1}, a'_i) \in \pi$ such that $a_i \in \overline{a'_i}$. □

Lemma 3. *Let CP be a sound classical planner that is guaranteed to terminate. Then* Find-Acceptable-Plan *returns in at most $|S| \cdot |A| + 1$ calls to CP, where S and A are the set of states and actions in the classical domain, respectively.*

Proof. To prove the bounds in the lemma, we need to show that after every call to *CP*, if Find-Acceptable-Plan did not exit then it adds a new state-action pair to B. From this it follows that since there is only a finite number of states and actions, Find-Acceptable-Plan must eventually return.

Note that the only time a state is removed from S is when *CP* returns failure, after which Find-Acceptable-Plan adds the state to K (Line 21). This means that once NDP2 adds a state to S, it either stays in S or is moved to K. Since Line 12 forbids adding a state to S which is already in either S or K, every state in S is unique, and we never add a state to S more than once.

Now we need to show that after every call to *CP*, Find-Acceptable-Plan either returns success or failure, or adds a new state-action pair to B. Look at what happens when *CP* returns a plan $\langle a_0, \ldots, a_n \rangle$ from the current state s to the goal, going through states $\langle s_1, \ldots, s_{n+1} \rangle$. Since *CP* is sound, s_{n+1} is a goal state. If Find-Acceptable-Plan accepts the whole plan, Find-Acceptable-Plan will return success on the next iteration.

Suppose Find-Acceptable-Plan rejects the first action a_0. We know $(s, a_0) \notin B$, since ConstrainProblem prevents those actions from being applicable as the first action in the plan. So if Line 12 rejects a_0, then (s, a_0) is a new pair added to B. Otherwise, a_0 is added to the current plan, and the current state is set to s_1.

Note that everywhere Find-Acceptable-Plan adds state-action pairs to B, the state part of the pair is the last state in S. So suppose Find-Acceptable-Plan accepts actions $a_0 \ldots a_i$, and rejects action a_{i+1}. Since Find-Acceptable-Plan only accepts actions that lead to states never before in S, the state-action pairs $(s_1, a_1), \ldots, (s_i, a_i), (s_{i+1}, a_{i+1}) \notin B$. And so (s_{i+1}, a_{i+1}) is a new state-action pair added to B.

Now look at what happens when *CP* returns failure. Find-Acceptable-Plan removes the last action from the plan (a'), and, if it does not return failure, adds the pair (s', a') to B, where s' is the previous state in the current plan. From above, we know that when we added a' to the plan that $(s', a') \notin B$. Furthermore, since Find-Acceptable-Plan added a' to p, s' hasn't been the last state in S until now, so (s', a') is still not in B. So (s', a') is a new pair added to B. □

Having shown termination, we can now show that Find-Acceptable-Plan returns failure or returns an acyclic plan whose policy image avoids the states in U. As shorthand, we call these plans U-acceptable. More formally, a plan p is *U-acceptable* in a state s with respect to a nondeterministic domain D, its determinization $\overline{D}$, a classical planning problem $(\overline{D}, s, G)$ and a set of states U if:

- p is applicable in the state s in the classical planning domain $\overline{D}$.
- It is acyclic (no repeated states).
- For every state-action pair $(s_i, \overline{a_i})$ in the image of p on $(\overline{D}, s, G)$, let a_i be the corresponding nondeterministic action in D. Then $\gamma_D(s_i, a_i) \cap U = \emptyset$.

Lemma 4. *If CP is sound and guaranteed to terminate, then* Find-Acceptable-Plan *is sound and it returns either a failure or a U-acceptable plan that ends in a goal state.*

Proof. Since Find-Acceptable-Plan checks if the last state in its partial plan p reaches the goal before returning a plan, it is enough to show that the partial plan p in the procedure is always a U-acceptable plan.

Since any prefix of a U-acceptable plan is still U-acceptable, we can do induction on the size of p in Find-Acceptable-Plan, looking only at additions to p. In the base case, p is empty, meeting the U-acceptable requirements trivially.

By the inductive hypothesis, assume p is U-acceptable and Find-Acceptable-Plan is adding an action a to p. Since the only location for this is in Line 12, Find-Acceptable-Plan has already checked in Line 12 that it does not form a loop in p, and that its corresponding action in D does not lead to any state in U. If *CP* is sound, then a is applicable in s. So a appended to p is a U-acceptable plan. □

Before we can show the completeness of Find-Acceptable-Plan, we need a utility lemma that says we can take a U-acceptable plans from states a to b and b to c to produce a U-acceptable plan from a to c. Note that the concatenation of any two U-acceptable plans may not be U-acceptable, since the resultant plan may visit some states twice.

Lemma 5. *With a nondeterministic domain D, its determinization $\overline{D}$, a classical planning problem $(\overline{D}, s_0, G)$ and a set of states U, let p be a U-acceptable plan from s_0 to some state s_1 and p' be a U-acceptable plan from s_1 to some state s_2. Let S be a directed graph where the nodes are the states associated with p and p' and the edges are the actions from p and p'.*

Then any acyclic path from s_0 to s_2 in S corresponds to a U-acceptable plan from s_0 to s_2.

Proof. Let p'' be any acyclic path through S. Then p'' is U-acceptable since p'' only goes through state transitions appearing in p applied at s_0 or p' applied at s_1, p'' is by definition acyclic, and no action in S leads to a state in U. □

Now we can show that Find-Acceptable-Plan is not only sound and guaranteed to terminate, but also complete:

Lemma 6. *If CP is sound and complete,* Find-Acceptable-Plan *is complete.*

Proof. To show that Find-Acceptable-Plan is complete, it is enough to show that Find-Acceptable-Plan never backtracks from a state along a U-acceptable path to a goal. We show this by contradiction.

Suppose Find-Acceptable-Plan is backtracking for the first time from a partial plan p with associated sates $s_0, \ldots, s$ from which there is a U-acceptable plan to the goal. Since p is U-acceptable, there is an action a applicable in s, the last state of p, which is along U-acceptable path to the goal. Since *CP* is complete, Find-Acceptable-Plan added the transition (s, a) to B sometime before hitting Line 21.

Now we reason about how and when (s, a) appeared in B. There are two locations in Find-Acceptable-Plan where B is modified. Either (s, a) must have been added via Line 13 or Line 21. We now show contradictions in four cases:

1. (s, a) was added to B with a partial plan that was a strict prefix of p.
2. (s, a) was added to B with the partial plan p.
3. (s, a) was added to B with a partial plan p' where p is a strict prefix of p'.
4. (s, a) was added to B with a partial plan p' where neither p nor p' is a prefix of the other.

Case 1. (s, a) *was added to* B *with a partial plan that was a strict prefix of* p. Since, by the soundness of Find-Acceptable-Plan (Lemma 4), p is irredundant, a proper prefix will not go through s, which is a necessary condition to hit both Line 13 and Line 21.

Case 2. (s, a) *was added to* B *with the partial plan* p. For Line 13, since a appended to p is part of a U-acceptable path, it will not create a cycle, and its policy image contains no states in U, so the conditional on Line 12 would prevent Find-Acceptable-Plan from reaching that line. For Line 21, this would mean that the state s immediately precedes itself in p, which violates the soundness lemma for Find-Acceptable-Plan.

Case 3. (s, a) *was added to* B *with a partial plan* p' *where* p *is a strict prefix of* p'. Say (s, a) was added to B while Find-Acceptable-Plan had a partial plan p' with a prefix of p that goes through states $s_0, \ldots, s, \ldots, s'$. To ban (s, a) in Line 13, *CP* must have produced a plan that goes from s' through s and then some $s'' = \gamma_{\overline{D}}(s, a)$ which forms a cycle. But since s is already in S, Line 12 would have stopped integrating the plan when it hit the action that lead to s. To ban (s, a) in Line 21, we would have to be planning from a state directly following s with the previous action a. By the soundness lemma the current plan is irredundant, so p' must be a appended to p. Since this is also along a U-acceptable path to the goal, it violates our assumption that the first backtrack from a U-acceptable path happened with a current plan of p.

Case 4. (s, a) *was added to* B *with a partial plan* p' *where neither* p *nor* p' *is a prefix of the other.* In order to place (s, a) in B, p' must either terminate at s for Line 13 or terminate at $s_a = \gamma_{\overline{D}}(s, a)$ for Line 21. In either case, p' is U-acceptable. Since p is U-acceptable and s is along a U-acceptable path to the goal, then there is a plan p_g from s to the goal such that p adjoined p_g is a U-acceptable plan. Since p' is U-acceptable, by Lemma 5, one can construct a U-acceptable plan p_s from just the states and actions in p' and p_g. Since p adjoined p_g must be irredundant, p_g must not lead to any state in p' that also appears in p. Let s_d be the first state along p' that differs from the states along p or the last state along p' if no states differ. Since p' is irredundant and p_g references no states in p' before s_d, p_s must go through s_d. However, this means that Find-Acceptable-Plan must have backtracked past s_d when there was a U-acceptable plan to the goal from that point. So the backtracking at s is not the first time, which contradicts our assumption.

Therefore, since any backtracking along a U-acceptable path to the goal causes a contradiction, Find-Acceptable-Plan is complete. □

Theorem 1. *Let CP be a sound and complete classical planner, U be a set of states, D be a nondeterministic planning domain, and* $\overline{D} = (\mathcal{L}, \overline{O})$ *be the determinization of D. Let S be the set of all states in* $\mathcal{L}$ *(i.e.,* $S = 2^F$*),* $F = \{$*all ground atoms over* $\mathcal{L}\}$*, and* $\mathcal{A}$ *be the set of all possible actions (i.e., all possible ground instantiations of the planning operators in O).*

Then Find-Acceptable-Plan$(D, \overline{D}, s_0, G, CP, U)$ *makes at most* $|S| \cdot |A| + 1$ *calls to CP, and returns an acyclic plan, if such a plan exists, whose policy image in D avoids the states in U.*

Proof. Immediately follows from Lemmas 3, 4, and 6. □

Lemma 7. *If CP is sound and guaranteed to terminate, then* NDP2 *returns in at most* $|S|^2$ *calls to* Find-Acceptable-Plan*, where S is the set of states in the domain.*

Proof. By Lemma 4, we have that Find-Acceptable-Plan is sound and terminates. Every iteration of NDP2 selects s, a non-goal π-result of S_0, and either produces a weak plan from that state to a goal state, or fails to find a plan, and adds s to U.

If NDP2 found a plan for s, it will not be a non-goal π-result of S_0 again unless NDP2 adds a child of s to U. Since there are finitely many states, there can be at most $|S|$ many iterations of the main planning loop before NDP2 either returns or adds a state to U.

Once a state is in U, since Find-Acceptable-Plan is sound, no action added to the policy will lead to that state. So again, NDP2 can only add at most $|S|$ states to U before there is no path from any leaf state to a goal that does not lead to a state in U.

With at most $|S|$ iterations between adding a state to U and at most $|S|$ additions to U, NDP2 must return in at most $|S|^2$ calls to Find-Acceptable-Plan. □

As written, this means that NDP2 will make $O(|S|^3 \cdot |A|)$ calls to *CP*. Notice, however, that we only add states to U. This means that in Find-Acceptable-Plan, we can cache B and K per starting state (caching p will not be helpful). This means that Find-Acceptable-Plan will only call *CP* $O(|S| \cdot |A|)$ times per starting state, which reduces NDP2's number of calls to *CP* to $O(|S|^2 \cdot |A|)$.

Lemma 8. *If CP is sound, then after each iteration of* NDP2*, there are no inescapable cycles in π. That is, every π-descendant state of the initial state has a path to a π-result of the initial state.*

Proof. The proof is by induction on the changes to π.

When π is empty, the initial state is a π-result of itself, so the lemma is trivially true. For the induction step, there are two ways NDP2 can change π:

1. Merging a plan from Find-Acceptable-Plan to π (Line 11).
2. Find-Acceptable-Plan returns failure (Line 19).

Case 1. Merging a plan from Find-Acceptable-Plan *to π.* NDP2 will merge the plan until it reaches the end of the plan, which by Lemma 4 is a goal state, or reaches a state in π which has a goal state π-descendant. In either case, all modified states in π and any state that had modified states as π-results now have a path to a goal π-result of S_0.

Notice that if that if NDP2 did not change the action for states already in the policy, that when NDP2 integrated a plan that went from a state s through a π-ancestor s' of s, it could create an inescapable cycle.

Case 2. Find-Acceptable-Plan *returns failure.* When Find-Acceptable-Plan returns failure on a state s, actions which lead to s are removed from π. So any state which claimed s as a π-descendant can now claim one of s's parents as a π-result. □

Lemma 9. *If CP is sound,* NDP2 *is sound.*

Proof. If NDP2 returns a policy, by the previous lemma all π-descendants of the initial state have a path to a non-goal π-result or a goal state. Since NDP2 terminated without failure, there are no more non-goal π-results in the policy, so all states have a path to a goal state, and π is a valid strong cyclic plan. □

Lemma 10. *If CP is sound and complete, at every point in the execution of* NDP2 *on a nondeterministic problem $P = (D, S_0, G)$, the set U is a subset of all unsolvable states. Thus any state in U cannot appear in any strong-cyclic solution policy for P.*

Proof. The proof is by induction on the size of U.

Let s be the first state added to U, which means Find-Acceptable-Plan returned failure when planning from s. Since U is empty and by Lemma 6 Find-Acceptable-Plan is complete, there is no path to a goal state from s, and so s would not be a π-descendant of s_0 in any valid strong cyclic policy.

Induct. Assume U contains only states which may not appear in any strong cyclic policy. Let s be the next state added to U, which means Find-Acceptable-Plan returned failure when planning from that state. Since Find-Acceptable-Plan is complete, all possible paths from s to a goal state also lead to a state in U, and thus s must also not appear in any valid policy. □

Lemma 11. *If CP is sound and complete,* NDP2 *is complete.*

Proof. Proof by contradiction. Assume NDP2 is not complete. Then there is a domain D, initial states S_0, goal set G, and strong cyclic policy π such that NDP2(D, S_0, G, CP) returns failure, even though π is a valid strong cyclic policy.

This means Find-Acceptable-Plan returned failure from an initial state, and so there is no path to the goal which also doesn't lead to a state in U. However, π has paths from each of the initial states to the goal, and so some action along each of those paths must lead to a state in U. This is a contradiction with the above lemma, that U will never contain states that appear in any strong-cyclic solution. □

Theorem 2. *Let CP be a sound and complete classical planner and $P = (D, S_0, G)$ be a nondeterministic planning problem with $D = (\mathcal{L}, O)$. Let S be the set of all states in $\mathcal{L}$ (i.e., $S = 2^F$), $F = \{$all ground atoms over $\mathcal{L}\}$, and $\mathcal{A}$ be the set of all possible actions (i.e., all possible ground instantiations of the planning operators in O).*

Then NDP2(D, S_0, G, CP) *is sound and complete, and returns at most in $|S|^2$ calls to* Find-Acceptable-Plan.

Proof. Immediately follows from Lemmas 7, 9, and 11. □

Appendix B. Extracting solutions from abstract and compound-abstract problems

Given a problem P and it's abstraction P^*, let s^* be an abstraction of a state s, let a^* from P^* be an action such that $\gamma_{P^*}(s^*, a^*) = \{{s_1}^*, \ldots, {s_n}^*, s'_1, \ldots, s'_j\}$, where states $s'_1, \ldots, s'_j$ are non-abstract, and let a be the action in P whose abstraction is a^* where $\gamma_P(s, a) = \{s_1, \ldots, s_m, s'_1, \ldots, s'_j\}$ $(n < m)$. Then for each s_i $(i = 1, \ldots, m)$, there is a merge action $\mathsf{merge}(\{\ldots\})_i$ such that the policy $\{(s, a), (s_1, \mathsf{merge}(\{\ldots\})_1), \ldots, (s_m, \mathsf{merge}(\{\ldots\})_m)\}$ has exactly the same π results as the

Algorithm 7: Algorithm to map an abstract policy π^* into a policy that isn't abstract.

Procedure unabstract($P = ((\mathcal{L}, O), S_0, G), P^*, \pi^*$)
while $\exists (s, a^*) \in \pi^*$ *where* s *is not abstract and* $a^* \notin O$ **do**
  Let s^* be the π^*-corresponding state of s
  Let $\{(s, a), (s_1, \text{merge}(\{\ldots\})_1), \ldots, (s_n, \text{merge}(\{\ldots\})_n)\}$ be the unabstracted image of (s^*, a^*) in s
  $\pi^* \leftarrow (\pi^* \setminus \{(s, \pi^*(s))\}) \cup \{(s, a)\}$
  $\pi^* \leftarrow \pi^* \cup \{(s_i, \text{merge}(\{\ldots\})_i) \mid \pi^*(s_i)$ is not defined$\}$
  if s *has no goal* π^**-descedants* **then**
    Pick $s' \in \gamma(s, a^*)$ such that s' has a path to the goal
    Let s'' be the first state on the path from s' to the goal such that $\pi^*(s'')$ is not a merge($\{\ldots\}$) or split-p operator
    Pick $s_j \in \gamma(s, a)$ such that s'' is an abstraction of s_j
    $\pi^* \leftarrow (\pi^* \setminus \{(s_j, \pi^*(s_j))\}) \cup \{(s_j, \text{merge}(\{\ldots\})_j)\}$
  Remove any non-π^*-descendants of S_0
return π^*

Algorithm 8: Algorithm to map a compound-abstract policy into an abstract policy that isn't compound.

Procedure Map-to-Uncompound(π^{**}, O^{**}, Σ)
  while π^{**} *contains any compound actions* **do**
    foreach $(s, a) \in \pi^{**}$ **do**
      if a *has the form* split-$p(\ldots) \cdot o(\ldots)$ *where* split-$p \in \Sigma$ *and* $o \in O^{**}$ **then**
        remove (s, a) from π^{**}
        $\pi^{**} \leftarrow \pi^{**} \cup \{(s, \text{split-}p(\ldots)\}$
        $s' \leftarrow \{s \setminus \{p^*(\ldots)\}\} \cup p(\ldots)$
        if $\pi^{**}(s')$ *has no goal* π^{**}*-descendant* **then**
          remove $(s', \pi^{**}(s'))$ from π^{**}
          $\pi^{**} \leftarrow \pi^{**} \cup \{(s', o(\ldots)\}$
  return π^{**}

policy $\{(s^*, a^*)\}$. We call this policy the *unabstracted image* of (s^*, a^*) in s. We skip the details of how to find merge($\{\ldots\}$)$_i$ from s_i, since this is just a variant of the maximal abstraction algorithm.

For any non-goal non-abstract state s, we define the *π-path corresponding to state s* recursively as a sequence of states, starting with s. If s' is on the π-path corresponding to s, then:

- If $\pi(s')$ is not a split-p or merge($\{\}$) action, then s' is the last state on the π-path corresponding to s.
- If $\pi(s')$ is the action merge($\{\ldots\}$), then $\gamma(s', \text{merge}(\{\ldots\}))$ is the next state in the π-path corresponding to s.
- If $\pi(s')$ is the action split-$p(\ldots)$ and $\gamma(s', \text{split-}p(\ldots)) = \{s_1, s_2\}$, then only one of s_1 or s_2 is consistent with s, and that state is on the π-path corresponding to s.

If the π-path corresponding to s is finite (it does not loop forever), then the last state s' is the *π-corresponding state to s*, and the action $\pi(s')$ is executable in state s.

Given a solution π^* for P^*, Algorithm 7 can extract a solution for P. It loops over the solution, picking a non-abstract state s where π assigns an action with abstract effects. After the first iteration, this may include states where the current policy assigns a merge($\{\ldots\}$) action.

Algorithm 7 then finds the π-corresponding state s^* (which is potentially equal to s) and replaces $\pi(s)$ with $\pi(s^*)$. Algorithm 7 then, for every child s_i of s for which $\pi(s_i)$ is undefined, adds the corresponding action from the unabstracted image of (s^*, a^*).

The policy π should be a strong cyclic solution to a partially unabstracted P^* after every iteration of the main loop. So if after merging the unabstracted image of (s^*, a^*), s no longer has a path to the goal, then Algorithm 7 picks a state in the children of s replaces the policy for it with a merge($\{\ldots\}$) action pointing to one of the former children of s (one in $\gamma(s, a^*)$). Algorithm 7 terminates when no abstracted actions are left in the solution.

Let P^{**} be a compound-abstract version of P^*, and π^{**} be a solution for P^{**}. Algorithm 8 is an algorithm to extract an abstract plan π^* from π^{**}. It works by iterating over the state-action pairs in π^{**}, modifying them to translate each compound action split-$p(\ldots) \cdot o^{**}(\ldots)$ into its components split-$p(\ldots)$ and $o^{**}(\ldots)$. With each pass of the main loop, Algorithm 8 reduces the maximum amount of compounding. It terminates when none of the actions in π^{**} is compound.

Appendix C. Unsoundness of NDP

Algorithm 9 is the NDP algorithm from [30]. Unlike NDP2, NDP incorporates a plan by first converting it to a policy (presumably by first removing any cycles from the from the plan), and then incorporating any state-action pairs for any state where the current policy is undefined. Here we show by example that this is unsound. NDP, if used in the presence of unsolvable states, may return policies which are not strong cyclic solutions.

Algorithm 9: The NDP algorithm from [30], which is unsound in domains which have unsolvable states.

```
1  Procedure NDP(D, S_0, G, CP)
2      π ← ∅
3      D̄ ← a determinization of D; U ← ∅
4      loop
5          S ← {all non-goal π-results of S_0}
6          if S = ∅ then return π
7          arbitrarily select a state s ∈ S
8          call CP(D̄, s, G)
9          if CP returns a solution plan p then
10             Let π' be the policy image of p applied at s
11             π ← π ∪ {(s, a) ∈ π' | π(s) is not defined}
12         else
               // CP returned Failure
13             if s ∈ S_0 then return Failure
14             foreach s' such that s ∈ γ(s', π(s')) do
15                 modify D̄ to make the determinizations of π(s') inapplicable at s'
16                 π ← π \ {(s', π(s'))}
```

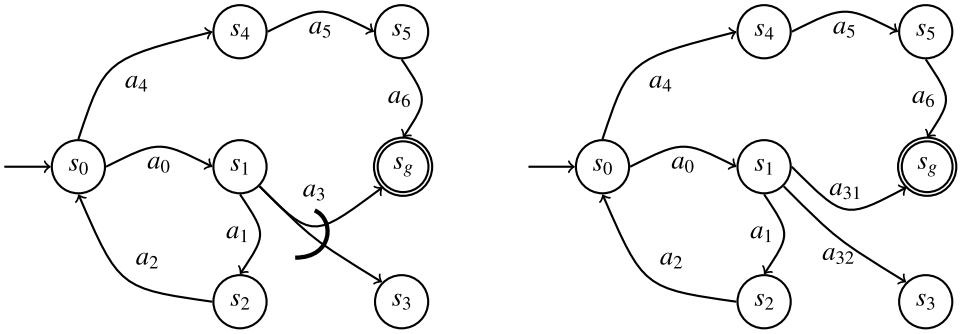

Fig. C.14. A nondeterministic domain D and its determinization, $\overline{D}$.

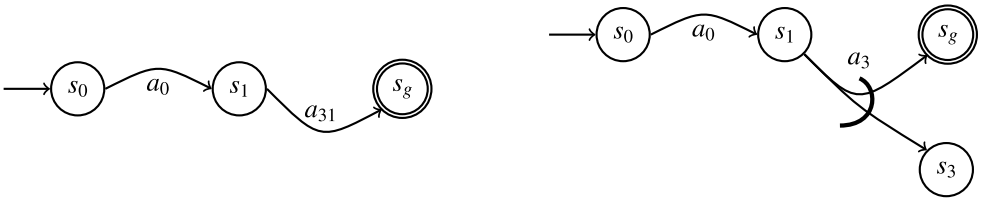

Fig. C.15. A classical solution to $(\overline{D}, s_0, s_g)$ and its incorporation into the empty policy.

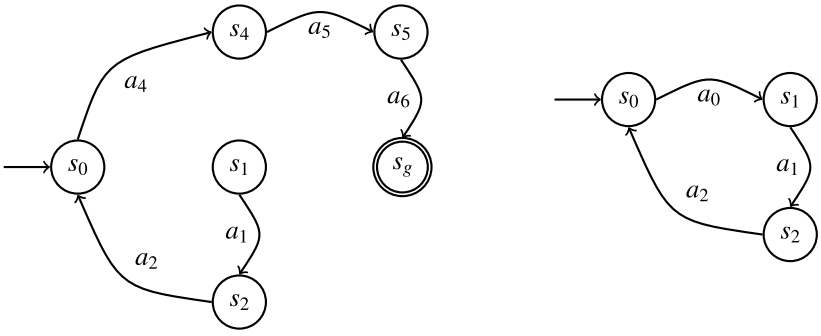

Fig. C.16. A classical solution to $(\overline{D}, s_1, s_g)$ that avoids s_3, and the resulting policy produced by NDP.

Example. Consider the nondeterministic planning problem $P = (D, S_0, G)$ where $S_0 = \{s_0\}$ and $G = \{s_g\}$. Fig. C.14 illustrates the nondeterministic planning problem P. In the figure, $\overline{D}$ is the determinization of D.

In NDP's first iteration, let CP return the shortest classical solution: $\langle a_0, a_{31} \rangle$ (Fig. C.15). After NDP incorporates the plan into the current policy π, there will be one non-goal π-result of S_0, s_3. On the next iteration, NDP will select s_3, and call

CP on the problem $(\overline{D}, s_3, G)$. Since there are no solutions to that problem, NDP will remove (s_1, a_3) from π and will make a_{31} and a_{32} inapplicable at s_1 in $\overline{D}$.

Now s_1 is the only non-goal π-result of S_0. NDP will now call *CP* on the classical planning problem $(\overline{D}, s_1, G)$. *CP* will return the plan $\langle a_1, a_2, a_4, a_5, a_6 \rangle$. NDP will incorporate just the first two actions from the plan, but it will not incorporate a_4, since s_0 already has an action, i.e., a_0, in the current policy. This leaves us with the policy shown on the right in Fig. C.16, where there is an inescapable loop between s_0, s_1, and s_2. There are now no non-goal π-results of S_0, and so NDP will exit on the next iteration of the loop, returning the policy found in Fig. C.16 which is not a solution to the original problem. □

So by never changing the action already associated with a state, NDP can create loops where states have no path to the goal. This violates one of the invariants that makes NDP2 work, which is that after every iteration, every state in the execution structure has a path to a goal or leaf state. This invariant is made explicit and proven in Lemma 8 in Appendix A.

References

[1] A. Albore, Planning with contingencies via a fast and informed action selection mechanism, in: ICAPS 2007 Doctoral Consortium, 2007.
[2] M. Benda, V. Jagannathan, R. Dodhiawala, On optimal cooperation of knowledge sources—an empirical investigation, Tech. Rep. BCS–G2010–28, Boeing Advanced Technology Center, Boeing Computing Services, 1986.
[3] P. Bertoli, A. Cimatti, M. Pistore, Strong cyclic planning under partial observability, in: ECAI, 2006, pp. 580–584.
[4] P. Bertoli, A. Cimatti, M. Roveri, P. Traverso, Strong planning under partial observability, Artif. Intell. 170 (2006) 337–384.
[5] B. Bonet, H. Geffner, GPT: a tool for planning with uncertainty and partial information, in: IJCAI, 2001, pp. 82–87.
[6] D. Bryce, O. Buffet, International planning competition—uncertainty track, http://ippc-2008.loria.fr/wiki/index.php/Main_Page, 2008.
[7] D. Bryce, S. Kambhampati, D.E. Smith, Planning graph heuristics for belief space search, J. Artif. Intell. Res. 26 (2006) 35–99.
[8] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang, Symbolic model checking: 10^{20} states and beyond, Inf. Comput. 98 (2) (June 1992) 142–170.
[9] A. Cimatti, M. Pistore, M. Roveri, P. Traverso, Weak, strong, and strong cyclic planning via symbolic model checking, Artif. Intell. 147 (1–2) (2003) 35–84.
[10] A. Cimatti, M. Roveri, P. Traverso, Automatic OBDD-based generation of universal plans in non-deterministic domains, in: AAAI, 1998, pp. 875–881.
[11] S. Edelkamp, P. Kissmann, Fully-observable non-deterministic planning via PDDL-translation into a game, in: International Conference on Automated Planning and Scheduling, 2008.
[12] K. Erol, D.S. Nau, V.S. Subrahmanian, Complexity, decidability and undecidability results for domain-independent planning, Artif. Intell. 76 (1–2) (1995) 75–88.
[13] M. Fox, D. Long, S. Bradley, J. McKinna, Using model checking for pre-planning analysis, in: Proceedings of the AAAI Symposium on Model-Based Validation of Intelligence, 2001.
[14] J. Fu, V. Ng, F.B. Bastani, I.-L. Yin, Simple and fast strong cyclic planning for fully-observable nondeterministic planning problems, in: IJCAI, 2011, pp. 1949–1954.
[15] M. Genesereth, I. Nourbakhsh, Time-saving tips for problem solving with incomplete information, in: AAAI, 1993.
[16] M. Ghallab, D. Nau, P. Traverso, Automated Planning: Theory and Practice, Morgan Kaufmann, 2004.
[17] F. Giunchiglia, P. Traverso, Planning as model checking, in: ECP, 1999.
[18] R.P. Goldman, M.S. Boddy, Conditional linear planning, in: AIPS, 1994.
[19] E. Hansen, S. Zilberstein, LAO*: a heuristic search algorithm that finds solutions with loops, Artif. Intell. 129 (2001) 35–62.
[20] J. Hoffmann, R. Brafman, Contingent planning via heuristic forward search with implicit belief states, in: ICAPS, 2005.
[21] J. Hoffmann, B. Nebel, The FF planning system: fast plan generation through heuristic search, J. Artif. Intell. Res. 14 (2001) 253–302.
[22] L. Iocchi, D. Nardi, M. Rosati, Strong cyclic planning with incomplete information and sensing, in: Workshop on Planning and Scheduling for Space, 2004.
[23] R. Jensen, M.M. Veloso, M.H. Bowling, OBDD-based optimistic and strong cyclic adversarial planning, in: ECP, 2001.
[24] R. Jensen, M.M. Veloso, R. Bryant, Guided symbolic universal planning, in: ICAPS, 2003.
[25] F. Kabanza, M. Barbeau, R. St-Denis, Planning control rules for reactive agents, Artif. Intell. 95 (1) (1997) 67–113.
[26] U. Kuter, D. Nau, Forward-chaining planning in nondeterministic domains, in: AAAI-2004, 2004.
[27] U. Kuter, D. Nau, M. Pistore, P. Traverso, A hierarchical task-network planner based on symbolic model checking, in: ICAPS, 2005.
[28] U. Kuter, D.S. Nau, Using domain-configurable search control for probabilistic planning, in: M.M. Veloso, S. Kambhampati (Eds.), AAAI, AAAI Press/The MIT Press, 2005, pp. 1169–1174.
[29] U. Kuter, D.S. Nau, M. Pistore, P. Traverso, Task decomposition on abstract states, for planning under nondeterminism, Artif. Intell. 173 (5–6) (2009) 669–695.
[30] U. Kuter, D.S. Nau, E. Reisner, R.P. Goldman, Using classical planners to solve nondeterministic planning problems, in: J. Rintanen, B. Nebel, J.C. Beck, E.A. Hansen (Eds.), ICAPS, AAAI, 2008, pp. 190–197.
[31] M.L. Littman, Probabilistic propositional planning: representations and complexity, in: AAAI, AAAI Press/MIT Press, Providence, Rhode Island, 1997, pp. 748–761.
[32] Mausam, A. Kolobov, Planning with Markov Decision Processes: An AI Perspective, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers, 2012.
[33] C. Muise, S.A. McIlraith, J.C. Beck, Improved non-deterministic planning by exploiting state relevance, in: ICAPS-12, 2012.
[34] P. Nyblom, Handling uncertainty by interleaving cost-aware classical planning with execution, in: Swedish AI Society Workshop, 2005.
[35] J.S. Penberthy, D. Weld, UCPOP: a sound, complete, partial order planner for ADL, in: KR, 1992.
[36] M. Peot, D. Smith, Conditional nonlinear planning, in: AIPS, 1992.
[37] M. Pistore, R. Bettin, P. Traverso, Symbolic techniques for planning with extended goals in non-deterministic domains, in: ECP, 2001.
[38] M. Pistore, P. Traverso, Planning as model checking for extended goals in non-deterministic domains, in: IJCAI, 2001.
[39] L. Pryor, G. Collins, Planning for contingency: a decision based approach, J. Artif. Intell. Res. 4 (1996) 81–120.
[40] J. Rintanen, Improvements to the evaluation of quantified boolean formulae, in: IJCAI, 1999.
[41] F. Teichteil-Konigsbuch, U. Kuter, G. Infantes, Incremental plan aggregation for generating policies in MDPs, in: AAMAS, 2010.
[42] S. Yoon, A. Fern, R. Givan, FF-Replan: a baseline for probabilistic planning, in: ICAPS, 2007.
[43] S.W. Yoon, A. Fern, R. Givan, S. Kambhampati, Probabilistic planning via determinization in hindsight, in: AAAI, 2008, pp. 1010–1016.
[44] H. Younes, M. Littman, D. Weissman, J. Asmuth, The first probabilistic track of the international planning competition, J. Artif. Intell. Res. 24 (1) (2005) 851–887.