Artificial Intelligence 173 (2009) 669–695

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Task decomposition on abstract states, for planning under nondeterminism

Ugur Kuter [a,*], Dana Nau [a], Marco Pistore [b], Paolo Traverso [b]

[a] *Department of Computer Science and Institute of Systems Research and Institute of Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA*
[b] *Institute for Scientific and Technological Research (IRST), Fondazione Bruno Kessler, Via Sommarive 18, Povo, 38050 Trento, Italy*

A R T I C L E   I N F O

Article history:
Received 1 October 2007
Received in revised form 26 November 2008
Accepted 26 November 2008
Available online 6 December 2008

Keywords:
Planning in nondeterministic domains
Hierarchical task-network (HTN) planning
Binary decision diagrams

A B S T R A C T

Although several approaches have been developed for planning in nondeterministic domains, solving large planning problems is still quite difficult. In this work, we present a new planning algorithm, called Yoyo, for solving planning problems in fully observable nondeterministic domains. Yoyo combines an HTN-based mechanism for constraining its search and a Binary Decision Diagram (BDD) representation for reasoning about sets of states and state transitions.

We provide correctness theorems for Yoyo, and an experimental comparison of it with MBP and ND-SHOP2, the two previously-best algorithms for planning in nondeterministic domains. In our experiments, Yoyo could easily deal with problem sizes that neither MBP nor ND-SHOP2 could scale up to, and could solve problems about 100 to 1000 times faster than MBP and ND-SHOP2.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Although many highly efficient algorithms have been built for classical planning, the applicability of these algorithms has been quite limited, due to the restrictive assumptions of classical planning. Hence, there is rapidly growing interest in planning domains that violate some of these assumptions—for example, *nondeterministic planning domains*, in which the actions may have nondeterministic outcomes.[1]

Although several approaches have been developed for planning in nondeterministic domains, the problem is still very hard to solve in practice, even under the simplifying assumption of *full observability*, i.e., the assumption that the state of the world can be completely observed at run-time. Indeed, in the case of nondeterministic domains, the planning algorithm must reason about all possible different execution paths to find a plan that works despite the nondeterminism, and the dimension of the generated conditional plan may grow exponentially.

Before our development of the Yoyo algorithm described in this paper, the two best-performing algorithms for planning in nondeterministic domains were MBP [1,2] and ND-SHOP2 [3]:

- MBP uses a suite of planning algorithms based on *Symbolic Model Checking*, which represent states symbolically using *Ordered Binary Decision Diagrams* (BDDs) [4]. In experimental studies, MBP's planning algorithms have easily scaled up to rather large-sized problems [2]. This is due largely to the fact that BDDs can represent large sets of states as compact

* Corresponding author.
E-mail addresses: ukuter@cs.umd.edu (U. Kuter), nau@cs.umd.edu (D. Nau), pistore@itc.it (M. Pistore), traverso@itc.it (P. Traverso).

[1] Unfortunately, the phrase "nondeterministic outcomes" seems to have two meanings in the current literature: some researchers attach probabilities to the outcomes (as in a Markov Decision Process), and others omit the probabilities (as in a nondeterministic automaton). Our usage is the latter one.

0004-3702/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.artint.2008.11.012

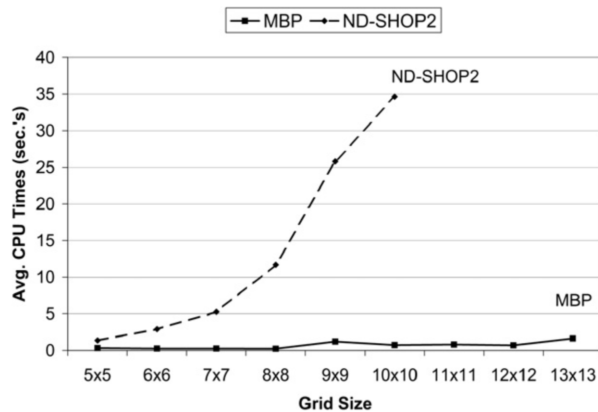

Fig. 1. Average running times in seconds for MBP and ND-SHOP2 in the Hunter–Prey Domain as a function of the grid size, with one prey. ND-SHOP2 was not able to solve planning problems in grids larger than 10×10 due to memory-overflow problems.

formulae that refer to the salient properties of those states. As an example, suppose that in the blocks world, we have a formula f that represents some set of states S, and we want to reason about the set of all states in S in which the block a is on the table. This set of states could be represented by a formula like $f \wedge ontable(a)$.

- ND-SHOP2 is a planner for nondeterministic domains that uses a *Hierarchical Task Network* (HTN) decomposition technique that is like that of the well-known SHOP2 planner [5] for deterministic domains. ND-SHOP2's HTNs provide domain-specific information to constrain the planner's search. Among the actions that are applicable to a state, ND-SHOP2 will only consider those actions that it can obtain via HTN decomposition, and this can prevent exploration of large portions of the search space if there are reasons to believe that those portions are unpromising. For example, in the blocks world, if the objective is to move a stack of blocks from one location to another, then we would only want to consider actions that move those particular blocks, rather than any of the other blocks in the domain. Under the right conditions, ND-SHOP2 can perform quite well; for example, [3] describes some cases where it outperforms MBP.

ND-SHOP2 and MBP use very different techniques for reducing the size of the search space: MBP reasons about large sets of states as aggregate entities, and ND-SHOP2 focuses only on those parts of the search space that are produced via HTN decomposition. As a consequence, there are situations in which each algorithm can substantially outperform the other.

As a simple example, consider the well-known Hunter–Prey domain [6]. In the Hunter–Prey domain, the world is an $n \times n$ grid in which the planner is a hunter that is trying 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). The prey has also five actions: the four movement actions plus a stay-still action—but instead of representing the prey as a separate agent, its possible actions are encoded as the nondeterministic outcomes for the hunter's actions. In this domain, a solution is any *policy* (i.e., a set of state-action pairs telling the hunter what to do under various conditions) for which there is a guarantee that all of the prey will eventually be captured.

There are some sets of Hunter–Prey problems in which ND-SHOP2 is exponentially faster than MBP, and other sets of Hunter–Prey problems where the reverse is true:

- Fig. 1 shows the average running times required by MBP and ND-SHOP2 on hunter–and–prey problems with one prey, as a function of increasing grid sizes.[2] ND-SHOP2 ran out of memory in large problems, because the solution policies in this domain contain a huge number of state-action pairs and ND-SHOP2 represents most of these state-action pairs explicitly. MBP, on the other hand, uses compact propositional formulas to represent sets of states that share salient common properties, and it searches a search space whose nodes are these formulas (rather than having a separate node for each state). This dramatically reduces the size of the search space, hence MBP's small running time.
- Fig. 2 shows the average running times required by MBP and ND-SHOP2 when we fix the grid size to 4×4, but increase the number of prey to catch (we made the movements of prey dependent on each other by assuming that a prey cannot move to a location next to another prey). In this case, ND-SHOP2 outperforms MBP, because it is able to use a simple yet extremely effective pruning heuristic to constrain its search: "choose one prey and chase it while ignoring others; when you catch that prey, choose another and chase it, and continue in this way until all of the prey are caught".

[2] All of the experiments were run on an AMD Duron 900 MHz laptop with 256 MB memory running Linux Fedora Core 2, using a time limit of 40 minutes for each problem. Each data point is an average of 20 randomly-generated problems. In our experiments, if a planning algorithm could not solve a problem within this time limit (or it required more memory than available on our experimental computer), it was run again on another problem of the same size. Each data point for which there were more than five such failures was omitted from the results shown in the figures. Thus the data make the worse-performing algorithm (ND-SHOP2 in Fig. 1 and MBP in Fig. 2) look better than it really was—but this makes little difference since the disparity in the algorithms' performance is so great.

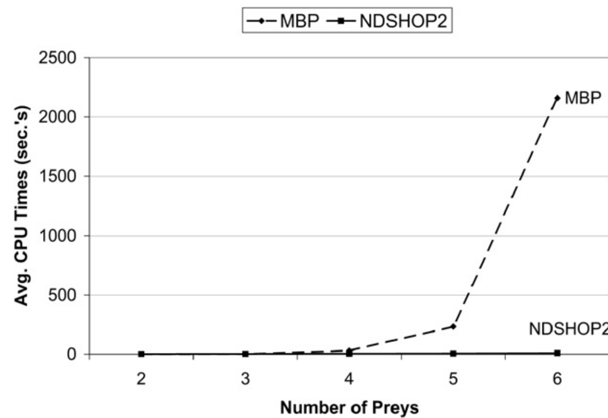

Fig. 2. Average running times in seconds for MBP and ND-SHOP2 in the Hunter–Prey Domain as a function of the number of prey, with a fixed 4×4 grid.

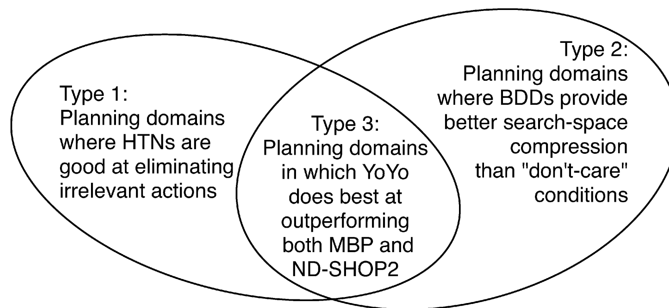

Fig. 3. A visual characterization of the planning domains in which Yoyo does best.

MBP, on the other hand, must consider all applicable actions at each node of its search space, which produces a larger branching factor and hence a much larger search space.

This paper presents a formalism and a novel algorithm, called Yoyo, that combines the power of the HTN-based search-control strategies with a BDD-based state representation. Yoyo implements an HTN-based forward-chaining search as in ND-SHOP2, built on top of MBP's techniques for representing and manipulating BDDs.

This combination has required a complete rethinking of the ND-SHOP2 algorithm, in order to take advantage of situations where the BDD representation will allow it to avoid enumerating states explicitly. In a backward-search planner such as MBP, each goal or subgoal is a set of states that can be represented quite naturally as a BDD. But forward-search algorithms like ND-SHOP2 normally apply actions to individual states, hence BDDs cannot be used effectively unless we can determine *which* sets of states can usefully be combined into a single BDD, and *how* to apply actions to these sets of states. One of the contributions of this paper is a way to do these things.

Besides the definition of the Yoyo planning algorithm, we provide theorems about its soundness and completeness, and experimental results demonstrating its performance. Our experiments show that planning domains can be divided into three types (see the corresponding parts of Fig. 3):

(1) Planning domains in which Yoyo and ND-SHOP2 both do much better than MBP. In these domains, the HTNs used in Yoyo and ND-SHOP2 provided good pruning of the search space, but MBP's BDDs provided no special advantage. The reason for the latter was that in these domains, ND-SHOP2 could use "don't care" conditions (i.e., a particular type of state abstraction enabled by HTNs in some of the effects and preconditions of the actions of ND-SHOP2, as described below) to get just as much search-space compression as Yoyo's BDDs.
(2) Planning domains in which Yoyo and MBP both do much better than ND-SHOP2. These were domains in which ND-SHOP2's HTNs did not provide very good pruning of the search space, and in which the BDDs used in Yoyo and MBP could provide better compression of the search space than ND-SHOP2's "don't care" conditions.
(3) Planning domains in which Yoyo does much better than both MBP and ND-SHOP2. In these domains, HTNs and BDDs both were useful for reducing the search space; and since Yoyo used both, it had a significant advantage over both ND-SHOP2 and MBP. ND-SHOP2's and MBP's running times both increased exponentially faster than Yoyo's. Yoyo could solve planning problems about two or three orders of magnitude more quickly then MBP and ND-SHOP2, and could easily deal with problem sizes that neither MBP nor ND-SHOP2 could scale up to.

This paper is organized as follows. Section 2 presents our formalism, definitions, and notation used in this paper. Then, we describe the Yoyo planning algorithm and the mechanisms it uses to generate solutions to nondeterministic planning problems in Sections 3 and 4, respectively. Next, we describe how Yoyo uses BDDs for compact representations of states in Section 5. Section 6 gives our theoretical analysis of the planning algorithm. We describe our experimental evaluation of Yoyo and the results of this evaluation in Section 7. We conclude our paper with a discussion on the related work and our final remarks.

2. Basic definitions and notation

In this section, we give our definitions and notation for the formalism and algorithms we discuss later in the paper.

2.1. Nondeterministic planning domains

Most of our definitions are the usual ones for nondeterministic planning domains and planning problems (e.g., see [2]). A *nondeterministic planning domain* is modeled as a nondeterministic state-transition system $\Sigma = (\mathcal{S}, \mathcal{A}, \gamma)$, where $\mathcal{S}$ and $\mathcal{A}$ are the finite sets of all possible states and actions in the domain, and the state-transition function is

$$\gamma : \mathcal{S} \times \mathcal{A} \to 2^{\mathcal{S}}.$$

An action a is *applicable* in a state s if $\gamma(s,a) \neq \emptyset$. The set of all states in which a is applicable is

$$S_a = \{s \in \mathcal{S} \mid \gamma(s,a) \neq \emptyset\}.$$

Conversely, the set of all actions applicable to a state s is

$$A_s = \{a \in \mathcal{A} \mid \gamma(s,a) \neq \emptyset\}.$$

A state s is *live* if at least one action is applicable to s; otherwise s is *dead*.

We consider a *policy* to be a partial function[3] π from $\mathcal{S}$ into $\mathcal{A}$, and we let $S_\pi \subseteq \mathcal{S}$ be π's domain (hence π is a total function from S_π into $\mathcal{A}$).

The *execution structure* Σ_π for a policy π is a labeled digraph that includes all states and actions that are reachable through π. Formally,

$$\Sigma_\pi = (V_\pi, E_\pi),$$

where $V_\pi \subseteq \mathcal{S}$ and $E_\pi \subseteq \mathcal{S} \times \mathcal{S}$ such that

- $S_\pi \subseteq V_\pi$; and
- for every state $s \in V_\pi$ if $\pi(s)$ is defined (i.e., there is an action a for s in π), then for all $s' \in \gamma(s,a)$, $s' \in V_\pi$ and $E_\pi(s,s')$.

Each edge in the execution structure Σ_π is labeled with the action $\pi(s)$. For the clarity of the discussions, we will denote an edge in Σ_π by a triple $(s, \pi(s), s')$ in the rest of the paper.

A *terminal state* of π is a state $s \in S_\pi$ that has no successors in Σ_π. We let S^t_π denote the set of all terminal states in Σ_π. An *execution path* of π is any path in Σ_π that begins at an initial state and that either is infinite or ends at a terminal state.

For any two states $s, s' \in \Sigma$, if there is a path in Σ from s to s', then s is a *Σ-ancestor* of s', s' is a *Σ-descendant* of s, and s' is *Σ-reachable* from s. We can also define reachability with respect to a policy π. For any two states $s, s' \in \Sigma_\pi$, if there is a path in Σ_π from s to s', then s is a *π-ancestor* of s', s' is a *π-descendant* of s, and s' is *π-reachable* from s.

A *planning problem* in a nondeterministic planning domain Σ is a triple (Σ, S_0, G), where $S_0 \subseteq \mathcal{S}$ is the set of *initial states* and $G \subseteq \mathcal{S}$ is the set of *goal states*. Solutions to planning problems in nondeterministic domains are usually classified as *weak* (at least one execution path will reach a goal), *strong* (all execution paths will reach goals), and *strong-cyclic* (all "fair" execution paths will reach goals) [7–9]. More precisely,

- A *strong solution* is a policy that is guaranteed to reach a goal state, despite the nondeterminism in the domain. That is, a policy π is a strong solution if there are no cycles in the execution structure Σ_π, and every execution path in Σ_π ends at a goal state.
- 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 $s \in S_0$, there is at least one execution path in Σ_π that ends at a goal state.
- A *strong-cyclic solution* is a policy π that has the following properties: every finite execution path of π ends at a goal state, and every cycle in Σ_π contains at least one Σ_π-ancestor of a goal state. Such a policy is guaranteed to reach a

[3] The reason for making π a partial function is so that its domain needn't contain states that it will never reach.

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.[4]

A nondeterministic planning problem is *weakly*, *strongly*, or *strong-cyclicly* solvable if it has a weak, strong, or strong-cyclic solution, respectively.

A policy π is a *candidate* weak, strong, or strong-cyclic solution if it satisfies the conditions stated above, with the following change: we require an execution of a path in Σ_π to end either at a goal state or at a live terminal state. Intuitively, if π is candidate solution then it may be possible to extend it to a solution, but if π is not a candidate solution then there is no way in which π can be extended to a solution.

2.2. Hierarchical task networks (HTNs) in nondeterministic domains

Our definitions for primitive tasks, nonprimitive tasks, task networks, and methods are abstracted versions of the ones for Simple Task Network (STN) planning in [10, Chapter 11].

We assume the existence of a set $\mathcal{T}$ of *tasks* to be performed. $\mathcal{T}$ includes all of the actions in $\mathcal{A}$, as well as some additional tasks called *nonprimitive tasks*.[5] An *HTN* is a pair (T, C), where T is a set of tasks and C is a set of partial ordering constraints on the tasks. The *empty* HTN is the pair (T, C) such that $T = \emptyset$ and $C = \emptyset$.

A *method* describes a possible way of decomposing a nonprimitive task into a set of *subtasks*. Rather than getting into the syntactic details of how a method is represented, we will define a method abstractly as a partial function $m : \mathcal{S} \times \mathcal{T} \to \mathcal{H}$, where $\mathcal{S}$, $\mathcal{T}$, and $\mathcal{H}$ are the sets of all possible states, tasks, and HTNs. If (s, t) is in the domain of m, i.e., if $m(s, t)$ is defined, then we say that m is *applicable* to the task t in the state s, and that $m(s, t)$ is the HTN produced by applying m to t in s. If $m(s, t) = (T, C)$, where $T = \{t_1, \dots, t_k\}$ is a set of tasks and C is a set of constraints, then we say that m *decomposes* t into (T, C) in the state s.

As an example, here is an informal description of a method we might use for the task chase_prey in the Hunter–Prey domain:

method north_chase for the task chase_prey:
applicability conditions: the prey is not caught, and the prey is to the north
subtasks: (1) move_north, (2) chase_prey
constraints: do subtask (1) before subtask (2)

This method is applicable only in states where the prey is not yet caught and is currently to the north of the hunter, and it specifies that the hunter should first move to the north and then continue chasing the prey. One can define similar methods for the cases where the prey is to the east, south, or west of the hunter.

In constructing a plan or a policy for a task, an HTN planner will only consider actions that are produced by applicable methods. For example, if north_chase is applicable and no other methods are applicable, then the next action in the plan or policy will be move_north, regardless of whether any other actions might be applicable in s.

Let s be the current state, $\chi = (T, C)$ be an HTN, and $t_0 \in T$ be a task that has no predecessors in T, i.e., C contains no constraints of the form $t \prec t_0$. Then t_0 will either be primitive, in which case we will want to use an action to accomplish it; or else it will be nonprimitive, in which case we will want to decompose it using a method. We consider both cases below.

Case 1: t_0 is a primitive task (i.e., t_0 is the name of an action). If the action is applicable in s, it will produce a set of successor states $\gamma(s, t_0)$ and a task network (T', C'), where $T' = T - \{t_0\}$, and $C' = C - \{\text{all constraints in } C \text{ that mention } t_0\}$. We will call this task network $\tau(\chi, s, t_0)$.

Case 2: t_0 is a nonprimitive task. If a method m is applicable to t_0 in s, it will produce a task network $m(s, t_0) = (T', C')$. We will now let $\delta(\chi, s, m, t_0)$ be the task network produced by inserting (T', C') into (T, C) in place of t_0. Formally, $\delta(\chi, s, m, t_0) = (T'', C'')$, where

$$T'' = \big(T - \{t_0\}\big) \cup T';$$

$$C'' = C' \cup \{c\theta_t \mid c \in C \text{ and } t \in T'\}, \quad \text{where } \theta_t \text{ is the substitution that replaces } t_0 \text{ with } t.$$

Given an HTN (T, C), a set of methods M, a policy π, an execution path p of π and a current state s in p, we say that path p *accomplishes* (T, C) from s if one of the following conditions is true:[6]

[4] This is equivalent to saying that every action has nonzero probabilities for all of its outcomes, whence the probability that we'll never leave the cycle is zero.

[5] The authors of [10, Chapter 11] give a specific syntactic representation for tasks; but the details of this representation are irrelevant for the purposes of this paper.

[6] These conditions are a straightforward adaptation of the conditions for π to accomplish T in a deterministic planning domain; see Chapter 11 of [10] for details.

(1) T is empty. Then, π accomplishes (T, C) if state s is the terminal state for the path p—i.e., π accomplishes (T, C) if $\pi(s)$ is undefined.
(2) There is a primitive task (i.e., an action) t_0 in T that has no predecessors in T, the action t_0 is applicable to s, the successor state s' of s along path p is such that $s' \in \gamma(s, t_0)$, and p accomplishes $\tau(\chi, s, t_0)$ from s'.
(3) There is a nonprimitive task t_0 in T that has no predecessors in T, there is a method $m \in M$ that is applicable to t_0, and the execution path p accomplishes $\delta(\chi, s_0, m, t_0)$ from s, where δ is as defined above.

Given an HTN (T, C), a set of methods M, and a policy π, we say that the execution structure Σ_π of π *strongly accomplishes* (T, C) if *all* execution paths of π accomplish (T, C) from their initial states.

The definition of *weakly accomplishes* is similar, but only requires that for each initial state $s_0 \in S_0$ there is *some* path starting from s_0 that accomplishes (T, C). Also the definition of *strong cyclicly accomplishes* is similar, but only requires that all the *all fair* execution paths of π accomplish (T, C).

If the execution structure Σ_π of a policy π weakly, strongly, or strong-cyclicly accomplishes an HTN $\chi = (T, C)$ from a state s, then we say that the policy π weakly, strongly, or strong-cyclicly accomplishes (T, C) from s.

We define Σ_π^χ, the execution structure that only contains the execution paths in Σ_π considered in satisfying the above definition of HTN accomplishment. Note that, for weakly accomplishing an HTN χ by a policy π, $\Sigma_\pi^\chi \subseteq \Sigma_\pi$, and for the strong and strong-cyclic case, $\Sigma_\pi^\chi = \Sigma_\pi$.

We extend the definition of planning problems in nondeterministic planning domains and the solutions for those planning problems to refer to HTNs as follows. We define a nondeterministic HTN planning problem as a tuple $P = (\Sigma, S_0, G, \chi, M)$ where Σ is a nondeterministic planning domain, $S_0 \subseteq \mathcal{S}$ is the set of *initial states*, and $G \subseteq \mathcal{S}$ is the set of *goal states*, χ be an HTN, and M be a set of methods.

A policy π is a candidate solution for P if and only if π is a candidate solution for the nondeterministic planning problem (Σ, S_0, G) without any HTNs.

A weak, strong, or strong-cyclic *solution* for a nondeterministic HTN planning problem is a policy π such that

- π weakly, strongly, or strong-cyclicly accomplishes χ from S_0 using the methods in M, and
- π also weakly, strongly, or strong-cyclicly solves the nondeterministic planning problem (Σ, S_0, G) and every execution path in Σ_π^χ ends in a goal state—recall that Σ_π^χ is the execution structure that contains only those execution paths in the execution structure Σ_π of π that are necessary to satisfy the HTN accomplishment requirements given previously in Section 2.2.

2.3. Notation involving sets of states

This section extends the definitions of Sections 2.1–2.2 to refer to sets of states. Later, Section 5 will discuss how these sets of states can be represented using Symbolic Model-Checking primitives.

A *set-based state-transition function* $\bar{\gamma}$ is defined as

$$\bar{\gamma}(S, a) = \bigcup \{\gamma(s, a) \mid s \in S\},$$

where S is a set of states, a is an action, and γ is the state-transition function as defined in Section 2.1. Intuitively, $\bar{\gamma}(S, a)$ is the set of all successor states that are generated by applying the action a in every state s in S in which a is applicable, i.e., every state in which $\gamma(s, a) \neq \emptyset$.

A *set-based policy* is a mapping $\bar{\pi}$ from disjoint sets of states into actions. Formally, $\bar{\pi}$ is a partial function from a partition $\{S_1, \ldots, S_k\}$ of $\mathcal{S}$ into $\mathcal{A}$. Note that if $\bar{\pi} : \{S_1, \ldots, S_k\} \to \mathcal{A}$ is any set-based policy, then we can map $\bar{\pi}$ into an ordinary policy *policy*$(\bar{\pi})$ as follows:

$$\mathit{policy}(\bar{\pi}) = \{(s, \bar{\pi}(S_i)) \mid 1 \leqslant i \leqslant k,\ s \in S_i,\ \text{and } \bar{\pi}(S_i) \text{ is defined}\}.$$

Hence, we will define $\bar{\pi}$'s *execution structure* to be

$$\Sigma_{\bar{\pi}} = \Sigma_{\mathit{policy}(\bar{\pi})}.$$

Let t be a task and m be a method, and suppose there is a set of states S such that $m(s, t) = m(s', t)$ for every $s, s' \in S$. Then we will say that the states in S are (t, m)-equivalent; and we will let $m(S, t) = m(s, t)$ where s is any member of S (it does not matter which one).

Let S be a set of states and χ be an HTN. Suppose t is a nonprimitive task. If $\delta(\chi, s, m, t) = \delta(\chi, s', m, t)$ for any two states $s, s' \in S$, then we will let the notation $\delta(\chi, S, m, t)$ represent the task network produced by decomposing t by m in any member of S. Similarly, if t is a primitive task (i.e., an action) and $\tau(\chi, s, t_0) = \tau(\chi, s', t_0)$for any two states $s, s' \in S$, then $\tau(\chi, S, t_0)$ represents the task network produced by applying t_0 in any member of S.

3. Yoyo = HTNs × BDDs

In this section, we describe Yoyo, our new planning algorithm for nondeterministic domains. Yoyo is an HTN search algorithm that combines ND-SHOP2's HTN-based search control with MBP's symbolic model-checking techniques for reasoning about sets of states and sets of state transitions.

```
Procedure Yoyo(F, G, M, π̄)
1.  if there is a pair (S, χ) ∈ F such that S ⊆ G and χ is not the empty HTN,
2.     then return(FAILURE)
3.  F ← {(S − (G ∪ S_π̄), χ) | (S, χ) ∈ F and S − (G ∪ S_π̄) ≠ ∅}
4.  if there is a pair (S, χ) ∈ F such that χ is the empty HTN,
5.     then return(FAILURE)
6.  if π̄ is not a candidate solution, then return(FAILURE)
7.  if F = ∅, then return(π̄)
8.  arbitrarily select a pair (S, χ) from F and remove it
9.  nondeterministically choose a task t that has no predecessors in χ
10. if t is a primitive task (i.e., an action), then
11.    let S′ be the largest subset of S s.t. t is applicable in every state of S′
12.    if S′ ≠ ∅ then
13.       insert (S′, t) into π̄
14.       insert (γ̄(S′, t), τ(χ, s, t)) into F, for some state s ∈ S′
15.    else, return(π̄)
16. else
17.    nondeterministically choose a method m ∈ M for t
18.    let S′ be the largest subset of S such that
19.       m is applicable to t in every state in S′
20.    if there is no such method m then return(FAILURE)
21.    insert (S′, δ(χ, S′, m, t)) and (S − S′, χ) into F
22. for all pairs (S_i, χ_i) and (S_j, χ_j) in F such that χ_i = χ_j,
23.    remove (S_i, χ_i) and (S_j, χ_j) from F and insert (S_i ∪ S_j, χ_i) into F
24. π̄ ← Yoyo(F, G, M, π̄)
25. if π̄ = FAILURE then return(FAILURE)
26. F ← F ∪ {(S, χ)}
27. F ← {(S′ − S_π̄, χ′) | (S′, χ′) ∈ F and S′ − S_π̄ ≠ ∅}
28. if F = ∅ then return(π̄′)
29. π̄ ← Yoyo(F, G, M, π̄)
30. return(π̄)
end-procedure
```

Fig. 4. Pseudocode for Yoyo. Above, G is the set of goal states and M is the set of HTN methods. In the initial call of the algorithm, the set-based policy $\bar{\pi}$ is empty and the fringe set is $F = \{(S_0, \chi_0)\}$ such that S_0 is the set of initial states and χ_0 is the initial task network.

Fig. 4 shows the Yoyo planning procedure. Yoyo takes a nondeterministic HTN planning problem $P = (\Sigma, S_0, G, \chi_0, M)$ and the empty policy $\bar{\pi}$ in the form of the tuple $(F, G, M, \bar{\pi})$, where F is the initial fringe set that contains the single pair of the set S_0 of the initial states and the initial task network χ_0. With this input, the planning algorithm searches for a solution policy for P.

To explain the role of F, first note that Yoyo does a nondeterministic search through a space of set-based policies. The terminal states of *policy*$(\bar{\pi})$ are analogous to the fringe nodes of a partial solution tree in an AND/OR search algorithm such as AO* [11], in the sense that Yoyo will need to "solve" each of these states in order to find a solution to the planning problem. But in Yoyo, "solving" a state means doing HTN decomposition on the HTN for that state, i.e., the tasks that need to be accomplished in the state.

Hence, Yoyo needs to reason about all pairs (s, χ) such that s is a terminal state and χ is the HTN for s. But for purposes of efficiency, Yoyo does not reason about each pair (s, χ) separately. Instead, it reasons about equivalence classes of states such that all of the states in each equivalence class have the same HTN. Hence, F is a collection of pairs $\{(S_1, \chi_1), \ldots, (S_k, \chi_k)\}$, where each S_i is a set of terminal states and χ_i is the HTN for all of those states. S_i is represented in Yoyo by a Binary Decision Diagram (see Section 5) that is much more compact than an explicit representation of each of the states.

Initially, F contains just one member, namely (S_0, χ_0). In Lines 1–2, Yoyo checks whether there is any pair (S, χ) in the F set such that every state s in S is a goal state but the HTN χ is not the empty HTN (i.e., there are tasks left to accomplish in χ when we reach to a goal state). In this case, Yoyo returns FAILURE because the current partial policy $\bar{\pi}$ does not weakly, strongly, or strong-cyclicly accomplish the input HTN.

Then, in Line 3 of Fig. 4, Yoyo first checks the elements of F for cycles and goal states: for every pair $(S, \chi) \in F$, Yoyo removes from S any state s that already appears in π (in which case an action has already been planned for s) or in G (in which case no action needs to be planned for s). If this makes S empty, then Yoyo simply discards (S, χ) since there is nothing else that needs to be done with it.

In Lines 4, Yoyo checks whether there is any pair (S, χ) in the F set such that χ is the empty HTN; if this is the case, then we have another failure point since there is no HTN decomposition of χ that would generate an action for s. Thus, Yoyo returns FAILURE in such a case.

Next in Line 6, Yoyo performs a candidacy test to see if π satisfies the requirements for a candidate solution. Intuitively, this test examines some or all of the paths in the execution structure Σ_π by performing a backward search from the terminal states of π. The details of Yoyo's candidacy test depends on whether we are searching for weak, strong, or strong-cyclic solutions; and we discuss these details in Section 4.

If π is a candidate solution, then Yoyo continues to develop it further. If F is empty, then π is a solution; hence Yoyo returns it. Otherwise, in Line 8, Yoyo selects any pair (S, χ) in F; note that this choice is arbitrary since they all must be selected eventually. Recall that χ is the HTN associated with the states in S; and, in order to select an action to perform at S, Yoyo will only consider applicable actions that can be produced by HTN decomposition.

In Line 9, Yoyo nondeterministically chooses a task t that has no predecessors in the HTN χ. As in SHOP2 and ND-SHOP2, this ensures that Yoyo always decomposes and accomplishes the tasks in the order they are executed in the world. If the task t is an action, then Yoyo selects the largest subset S' of S such that t is applicable in every state in S'. Then, Yoyo generates all possible successors of the states in S' by computing the result of applying t in those states. This computation is done via the set-based state-transition function $\bar{\gamma}(S', t)$ (see Lines 10–14). Then, Yoyo generates the resulting task network $\tau(\chi, t, s)$ for some state in S' as described in the previous section, and inserts the pair $(\bar{\gamma}(S', t), \tau(\chi, t, s))$ into the F set. Finally, Yoyo updates the current policy by adding the pair (S', t) in $\bar{\pi}$.

If there is at least one state s in S in which the action t is not applicable, then Yoyo immediately returns the current policy $\bar{\pi}$. The reason for returning $\bar{\pi}$ is to enforce Yoyo to select other decompositions in the states $S - S'$ while preserving the candidate policy generated so far.

If the task t is instead a nonprimitive task, Yoyo nondeterministically chooses a pair (S', m) where $m \in M$ is a method and S' is a nonempty subset of S such that the states in S' are (t, m)-equivalent. In principle one could apply m to t in s, producing a new task network $\chi' = \delta(\chi, s, m, t)$, and then one could insert both (s, χ') and $(S - \{s\}, \chi)$ into F. But for efficiency purposes Yoyo does not apply m to just one state at a time; instead, it applies m to all of the states in S' at once, to produce the task network $\chi' = \delta(\chi, s', m, t)$ for some state $s' \in S'$ (it does not matter which one, by the definition of $m(S', t)$). Then Yoyo inserts (S', χ') and $(S - S', \chi)$ into F.

After Lines 8–21 are done, it may sometimes happen that several of the pairs in F have the same task network. For example, there may be pairs $(S_i, \chi_i), (S_j, \chi_j) \in F$ such that $\chi_i = \chi_j$. When this happens, it is more efficient to combine these pairs into a single pair $(S_i \cup S_j, \chi_i)$, so Yoyo does this in Lines 22 and 23.

Yoyo successively performs all of the above operations through recursive invocations until there are no pairs left to explore in F. At that point, the algorithm returns $\bar{\pi}$ as a solution for the planning problem $(\Sigma, S_0, G, \chi_0, M)$.

When a recursive invocation returns in Line 24, Yoyo checks whether the returned value is a FAILURE due to one of the failure cases described above. If so, Yoyo immediately returns FAILURE. If $\bar{\pi}$ specifies an action for every state in F then Yoyo returns $\bar{\pi}$. Otherwise, Yoyo first inserts the pair (S, χ) that it had selected and removed from the F set previously in this invocation, and updates those pairs (S', χ') in F to replace S' with $S' - S_{\bar{\pi}}$; i.e., it updates each pair (S', χ) in which S' has at least one state in which the policy $\bar{\pi}$ is not defined in order to remove all the states from S' in which $\bar{\pi}$ is defined. Then, the planner calls itself recursively with this updated F and current partial policy $\bar{\pi}$ in order to guarantee that $\bar{\pi}$ will eventually specify an action for every state encountered during search.

4. Weak, strong, and strong-cyclic planning in Yoyo

Fig. 5 gives the pseudo-code for the candidacy test used in Line 6 of Yoyo. Intuitively, this test, called IsCandidate, examines some or all of the paths in the execution structure Σ_π by searching backward from π's terminal states toward the initial states. The definition of the Preimage and Good-Policy subroutines depend on whether we want to find weak, strong, or strong-cyclic solutions. We discuss all three cases below.

In weak planning, a candidate solution must have a path from each initial state to a live terminal state. The backward search starts from the terminal states of $\bar{\pi}$ (i.e., the states in $G \cup S_F$, where S_F is the set of all states in Yoyo's F set). The Preimage computation is the WeakPreimage operation defined in [2]:

$$\text{WeakPreimage}(S) = \{(s, a) \mid \gamma(s, a) \cap S \neq \emptyset\}.$$

At the ith iteration of the loop as a result of successive WeakPreimage computations, the set S contains the states of the policy $\bar{\pi}$ from which the goal states and the states in F are reachable in i steps or less. The search stops when no new states are added to S; that is, each state in $\bar{\pi}$ from which a terminal state of $\bar{\pi}$ is reachable has been visited by the backward search. At this point (the function Good-Policy), there are two cases to consider. Suppose S_0 contains a state $s \notin S$. This means that if we follow the actions in π starting from the initial states in S_0, we'll end up in a state from which it is impossible to reach any goal state. Therefore, π is not a candidate solution, so it cannot be extended to a solution during planning. Thus, Good-Policy returns FALSE. Otherwise, π is a candidate solution, so Good-Policy returns TRUE.

In strong planning, for $\bar{\pi}$ to be a candidate solution, every execution path must be acyclic and must end at a live terminal state. In this case, IsCandidate's Preimage computation corresponds to the StrongPreimage operation defined in [2]:

$$\text{StrongPreimage}(S) = \{(s, a) \mid \gamma(s, a) \subseteq S \text{ and } \gamma(s, a) \neq \emptyset\}.$$

At each iteration of the loop, it removes those state-action pairs from $\bar{\pi}$ that have been verified to be in the StrongPreimage of the goal and the terminal states. The Good-Policy subroutine implements two checks when the backward search stops. The first check is the same one as described for weak planning above. The second one involves checking if there is a state-action pair left in the policy. If this latter check succeeds then it means that the policy induces a cycle in the execution structure, hence it cannot be extended to a strong solution for the input planning problem. In this case, IsCandidate returns FALSE. Otherwise, it returns TRUE.

Procedure IsCandidate($\bar{\pi}$, S_F, G, S_0)
1. $S' \leftarrow \emptyset$; $S \leftarrow G \cup S_F$
2. while $S' \neq S$
3. $S' \leftarrow S$
4. $\bar{\pi}' \leftarrow \bar{\pi} \cap$ Preimage(S)
5. $S \leftarrow S \cup$ {all states in $\bar{\pi}'$}
6. $\bar{\pi} \leftarrow \{(s,a) \in \bar{\pi} \mid s \notin S\}$
7. return(Good-Policy($\bar{\pi}$, S_0, S))
end-procedure

Function Good-Policy($\bar{\pi}$, S_0, S) — *for weak planning*
1. If $S_0 \not\subseteq S$ then return FALSE
2. return TRUE
end-function

Function Good-Policy($\bar{\pi}$, S_0, S) — *for strong and strong-cyclic planning*
1. If $S_0 \not\subseteq S$ or $\bar{\pi} \neq \emptyset$ then return FALSE
2. return TRUE
end-function

Fig. 5. Pseudocode for the generic IsCandidate test. Above, S_F is the set of all states in the F set in the current invocation of Yoyo.

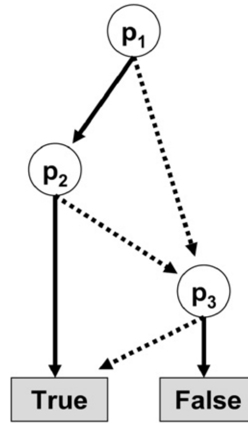

Fig. 6. An illustration of the BDD representation of the propositional formula $(p_1 \wedge p_2) \vee \neg p_3$. The solid arrows represent the case where a proposition p_i is TRUE and the dotted arrows represents the case where p_i is FALSE.

Finally in the strong-cyclic case, a requirement for $\bar{\pi}$ to be a candidate solution is that from every state in π there is an execution path in the execution structure $\Sigma_{\bar{\pi}}$ that ends at a live terminal state. In order to verify this requirement, IsCandidate simply uses the WeakPreimage computation as in the weak-planning case (in order to remove from π the states for which there is an execution path that reaches a live terminal state) and the Good-Policy subroutine as in the strong-planning case (in order to check that π is empty and hence there is no state that has no execution path to a live terminal state). This combination is sufficient to differentiate between the “fair” and “unfair” executions of $\bar{\pi}$.

5. Yoyo’s BBD-based implementation

Binary Decision Diagrams (*BDDs*) [4] are probably the best-known data structures that encode propositional formulae to compactly represent finite state automata (or in our case, nondeterministic planning domains). A BDD is a directed acyclic graph structure that represents a propositional logical formula in a canonical form. Each proposition in the formula corresponds to a node in a BDD that has exactly two outgoing edges: one of the edges represents the case where the proposition is true and the other represents the case where it is false. There are only two terminal nodes in a BDD; these nodes represent the two logical truth values TRUE and FALSE.

As an example, Fig. 6 shows an illustration of the BDD representation of the following propositional formula:

$$(p_1 \wedge p_2) \vee \neg p_3,$$

where each p_i is a proposition, and the solid and the dotted edges outgoing from each p_i represent the cases where p_i is true and false, respectively.

The truth value of a logical formula is evaluated by traversing the BDD representation of it. The traversal starts from the root node (i.e., the node that does not have any incoming edges into it), first evaluates the proposition represented by that node, and then, follows the edge that corresponds to the result of that evaluation to determine the next proposition. This traversal continues until it reaches to a terminal node that specifies the truth value of the logical formula.

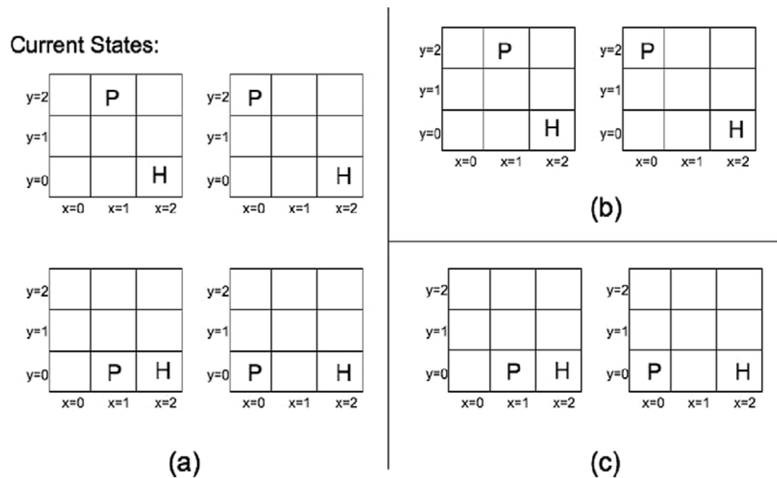

Fig. 7. An illustration of how Yoyo applies a method to the task chase-prey in a set of states. Chart (a) shows the set of current states and an abstract definition of one of the methods for chase-prey. Charts (b) and (c) shows those current states in which the method is applicable and it is not applicable, respectively.

As an example, suppose that in a state of the world, p_1 is FALSE, and p_2 and p_3 are TRUE, so the above formula is FALSE. The evaluation of this formula to this fact is done over the BDD of Fig. 6 as follows. We start from the p_1 node. Since p_1 is FALSE, we follow the dotted arrow out of this node, coming to the p_3 node. Since p_3 is TRUE, we follow the solid arrow out of p_3 and end up in the FALSE node. Note that this is correct since $\neg p_3$ is FALSE, and therefore, the entire formula is FALSE.

The above example also demonstrates that BDDs can be combined to compute the negation, conjunction, or disjunction of propositional formulas. Two BDDs are combined by taking the union of the directed acyclic graph structure of both and by performing bookkeeping operations over the edges between the nodes of the combined BDD in order to achieve the desired negation, conjunction, or disjunction. The combination of two BDDs, say b_1 and b_2, can be performed in quadratic time $O(|b_1|\ |b_2|)$, where $|b_i|$ is the size of b_i [4].

Like MBP, Yoyo uses BDDs to compactly represent sets of states and sets of transitions among those states. This requires two kinds of symbols to be defined: propositional symbols that describe the state of the world and one propositional symbol for each action in a given planning domain.

A set S of states is represented as a BDD that encodes the logical formula of the state propositions that are true in all of the states in S. A set $S_1 \cup S_2$ of states is represented by a BDD that combines the two BDDs for S_1 and S_2, respectively, in order to represent the disjunction of the logical formulae that correspond to those BDDs. Similarly, $S_1 \cap S_2$ is represented by a BDD that encodes the conjunction of the two logical formulae.

A state transition (s, s') such that $s' \in \gamma(s, a)$ for some action a is encoded as a logical formula, and therefore as a BDD, that is a conjunction of three subformulas: a formula of state propositions that hold in the state s, the proposition that represents the action a, and a formula of state propositions that hold in the state s'. Similarly, the state transitions that describe the set-based state-transition function $\bar{\gamma}(S, a)$ can be encoded as a BDD that is a combination of BDDs for S, a, and D, where D is the BDD that encodes every possible state transition in a given planning domain.

Suppose, at a particular iteration, Yoyo selects (S, χ) from its F set, where S is a set of states that is encoded as a BDD and χ is the HTN to be accomplished in the states of S. Let t be a task in χ that has no predecessors. If t is a primitive task (i.e., an action), then Yoyo first generates a BDD that represents the conjunction of the preconditions of t and the formula that represents the set S of states. Then, Yoyo combines the BDD for the applicability conditions of t with the one that represents S such that the combination of the two BDDs represent the conjunction of the logical formulae that represent them. The combined BDD represents the set S' of states in which t is applied in order to generate a BDD that represents the set of all possible successor states.

If t is not a primitive task, then Yoyo first nondeterministically chooses a method m applicable to t in some state $s \in S$. To make that choice, Yoyo first generates a BDD that represents the conditions that must be held in the world so that m could be applied to t. Then, the computation of the subset S' of S (see Line 16 of Fig. 4) is reduced to combining the BDD for the applicability conditions of m with the one that represents S such that the combination of the two BDDs represent the conjunction of the logical formulae that represent them.

Fig. 7 shows an illustration of how Yoyo uses BDDs to apply a method m to a task t in a given set S of states. In particular, we will use the same method we used as an example in Section 2.2:

method north_chase for the task chase_prey:
applicability conditions: the hunter is at the (x,y) location, and the prey is not caught, and the prey is to the north
subtasks: (1) move_north, (2) chase_prey
constraints: do subtask (1) before subtask (2)

Suppose the set S of states contains the four states, as shown in Fig. 7 (a), where the hunter is the bottom rightmost corner of a 3×3 grid. The current task is chase-prey. Given that the hunter is at the $(2, 0)$ location, Line 16 of Yoyo are performed as follows. Yoyo nondeterministically selects a method and generates a BDD that represents the following logical formula for the method's applicability conditions:

$$hunter_loc_x = 2 \wedge hunter_loc_y = 0 \wedge preycaught = no \\ \wedge \bigvee_{0 \leqslant i \leqslant 2,\, 1 \leqslant j \leqslant 2} (prey_loc_x = i \wedge prey_loc_y = j).$$

Then, Yoyo combines the BDD for the above formula with the one that represents the set S of states so that the combined BDD represents the conjunction of two BDDs in order to compute the subset S' of S where the method m is applicable to t in every state $s \in S'$. In our example, these states are shown in Fig. 7(b). If there are no paths in the combined BDD that end in the TRUE node, then this means that the method is not applicable in S. Otherwise, the paths in the combined BDD that end in the TRUE node represent the subset of S in which the method is applicable.

Finally, Yoyo computes the BDD that represents the set difference $S - S'$, which denotes the states of S in which another method should be applied to t (see the states shown in Fig. 7(c)). The set $S - S'$ of states correspond to the logical formula $f \wedge \neg f'$, where f and f' are the two logical formulas that represent the sets S and S', respectively.

The other set-based operations in Yoyo are performed in a similar manner.

6. Formal properties

This section presents theorems showing the soundness and completeness of Yoyo. The full proofs of the theorems are given in Appendix A.

Theorem 1. *Yoyo always terminates.*

The soundness and completeness of Yoyo depends on the soundness and completeness of the IsCandidate function, as Yoyo decides if a policy is a candidate solution based on this function and eliminates a policy for which this function returns FALSE.

Theorem 2. *Let $P = (\Sigma, S_0, G, \chi_0, M)$ be a nondeterministic HTN planning problem. Let $\bar{\pi}$ be a set-based policy and let $T_{\bar{\pi}}$ be the set of terminal states of $\bar{\pi}$.*

If $\bar{\pi}$ is a candidate solution for P, then IsCandidate$(\bar{\pi}, T_{\bar{\pi}}, G, S_0)$ returns TRUE. *Otherwise, IsCandidate$(\bar{\pi}, T_{\bar{\pi}}, G, S_0)$ returns* FALSE.

Theorem 3. *Suppose $P = (\Sigma, S, G, \chi, M)$ is a nondeterministic HTN planning problem.*

(1) *If Yoyo$(F_0, G, M, \bar{\pi}_0)$, where $F_0 = \{(S, \chi)\}$ and $\bar{\pi}_0 = \emptyset$, returns a policy π for P, then π weakly, strongly, or strong-cyclicly accomplishes χ from S using the methods in M.*
(2) *If there is no solution policy for P that accomplishes χ from S given M, then Yoyo returns* FAILURE.

The following theorem establishes the soundness of the Yoyo planning procedure:

Theorem 4. *Suppose Yoyo returns a policy π for a nondeterministic HTN planning problem $P = (\Sigma, S_0, G, \chi, M)$. Then π is a solution for P.*

Theorem 5. *Suppose $P = (\Sigma, S, G, \chi, M)$ is a solvable nondeterministic HTN planning problem. Then, Yoyo returns a policy π for P that weakly, strongly, or strong-cyclicly solves P.*

The above theorem establishes the completeness of Yoyo since the nondeterministic choice point of the Yoyo algorithm in Line 16 of Fig. 4 will allow the planner to consider every decomposition of χ that will generate a solution (if there is one) given the methods in M.

7. Experimental evaluation

The current implementation of Yoyo is built on both the ND-SHOP2 and the MBP planning systems. The core of the planner is implemented in Common Lisp like ND-SHOP2 is. However, Yoyo also includes C/C++ components for some BDD-based functionality to represent and manipulate sets of states, sets of state-transitions, and policies. Yoyo does not use any of the planning algorithms implemented in MBP, but it implements a bridge from Common Lisp to MBP's implementation for accessing and using MBP's BDD manipulation machinery during planning.

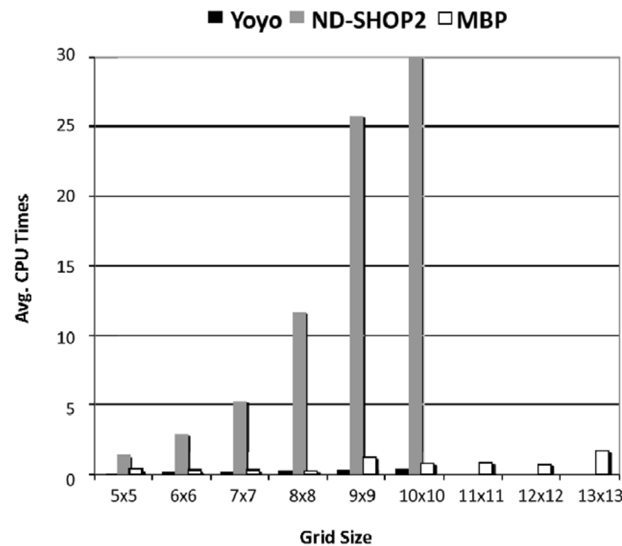

Fig. 8. Average running times in seconds of Yoyo, ND-SHOP2, and MBP in the hunter–prey domain as a function of the grid size, with one prey.

We evaluated Yoyo's performance in three planning domains: the Hunter–Prey domain mentioned in Section 1, the Robot Navigation domain that was used in [2] as one of the testbeds for MBP, and the Nondeterministic Blocks World domain that was used in [3] as a testbed for ND-SHOP2. The subsequent sections present and discuss the results of our experiments in these domains.

7.1. Comparisons of the planners' time performance

7.1.1. Hunter–Prey

In the Hunter–Prey domain, the world is an $n \times n$ grid in which the planner is a hunter that is trying to catch one or more prey. The world is fully-observable in the sense that the hunter can always observe the location of the 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). The prey has also five actions: the four movement actions plus a stay-still action—but instead of representing the prey as a separate agent, its possible actions are encoded as the nondeterministic outcomes for the hunter's actions. The hunter moves first and the prey moves afterwards in this domain.

All experiments in this domain were run on an AMD Duron 900 MHz laptop with 256 MB memory, running Linux Fedora Core 2. For ND-SHOP2 and Yoyo, we used Allegro Common Lisp v6.2 Free Trial Edition in these experiments. The time limit was for 40 minutes.[7]

Experimental Set 1. In these experiments, we compared Yoyo to ND-SHOP2 and MBP in hunter–prey problems with increasing grid sizes and with only one prey so that the nondeterminism in the world is kept at a minimum for the hunter.

Fig. 8 shows the results of the experiments for grid sizes $n = 5, 6, \ldots, 13$. For each value for n, MBP, ND-SHOP2, and Yoyo were run on 20 randomly-generated problems. This figure reports the average running times required by the planners on those problems.

Each time ND-SHOP2 and MBP had a memory overflow or they could not solve a problem within out time limit, we ran them again on another problem of the same size. Each data point on which there were more than five such failures is omitted in this figure, but the data points where it happened 1 to 5 times are included. Thus the data shown in Fig. 8 make the performance of ND-SHOP2 and MBP look better than it really was—but this makes little difference since they performed much worse than Yoyo.

For grids larger than $n = 10$, ND-SHOP2 was not able to solve the planning problems due to memory overflows. This is because ND-SHOP2 cannot reason over sets of states; rather, it has to reason about each state explicitly. However, in this domain, reasoning over clusters of states is possible and effective; for example, in every grid position that the hunter is in and that the prey is to the north of the hunter, the hunter will need to move to the north—i.e., the hunter will take the same north action in all of those states. Representing all of such states as a cluster and planning over such clusters is the main difference between the performances of Yoyo and ND-SHOP2 in this domain.

[7] As in [12], the CPU times for MBP's includes both its preprocessing and search times. Omitting the preprocessing times would not have significantly affected the results: they were never more than a few seconds, and usually below one second.

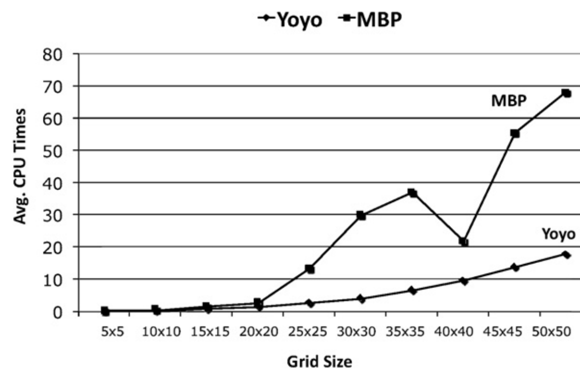

Fig. 9. Average running times in seconds for Yoyo and MBP on some larger grid sizes in the hunter–prey domain, with one prey.

Note also that this domain admits only high-level search strategies such as "look at the prey and move towards it". Although this strategy helps the planner prune a portion of the search space, such pruning alone does not compensate for the explosion in the size of the explicit representations of the search space for the problems.

On the other hand, both Yoyo and MBP were able to solve all of the problems in these experiments. The difference between the performances of Yoyo and ND-SHOP2 demonstrates the impact of the use of BDD-based representations of clusters of states and state transitions as mentioned above: Yoyo, using the same HTN-based pruning heuristic as ND-SHOP2, was able to scale up as good as MBP since it is able to exploit BDD-based representations of the problems and their solutions.

In order to see how Yoyo performs in larger problems compared to MBP, we have also experimented with Yoyo and MBP in much larger grids. Fig. 9 shows the results of these experiments in which, using the same setup as above, we varied the size of the grids in the planning problems as $n = 5, 10, 15, \ldots, 45, 50$.

These results (see Fig. 9) show that as the grid size grows, Yoyo's running time increases much more slowly than MBP's. This happens for the following reasons.

(1) Even though Yoyo does a forward search, its HTN-based search-control mechanism provides it with an ability analogous to that of MBP's backward search: the HTNs do a good job of eliminating actions that aren't relevant for achieving the goal.
(2) Yoyo's forward search enables it to consider only those states that are reachable from the initial states of the planning problems. In contrast, MBP's backward-chaining algorithms explore states that are not reachable from the initial states of the problems at all.
(3) Yoyo's use of BDDs enables it to compress the state space like MBP does. The main difference here is that in MBP, the BDDs arise naturally from the backward search's subgoal chaining; but in Yoyo's forward search, it is necessary to explicitly look for sets of states that are "equivalent" (in the sense that the same actions are applicable to all of them) and formulate BDDs to represent these sets.

Experimental Set 2. In order to further investigate the effect of using BDD-based representations in Yoyo, we used a variation of the Hunter–Prey domain, where there are more than one prey, and the prey i cannot move to any location within the neighbourhood of prey $i + 1$ in the world. In such a setting, the amount of nondeterminism for the hunter after each of its move increases combinatorially with the number of prey in the domain. Furthermore, the BDD-based representations of the underlying planning domain explode in size under these assumptions, because the movements of the prey are dependent to each other.

In this adapted domain, we provided to ND-SHOP2 and Yoyo a search-control strategy that tells the planners to chase the first prey until it is caught, then the second prey, and so on, until all of the prey are caught. Note that this search-control strategy allows for abstracting away from the huge state space: when the hunter is chasing a prey, it does not need to know the locations of the other prey, and therefore, it does not need to reason and store information about those locations.

We varied the number of prey from $p = 2$ to $p = 6$ in a 4×4 grid world and compared the running times of MBP, ND-SHOP2, and Yoyo. Fig. 10 shows the results. Each data point is an average of the running times of all three planners on 20 randomly-generated problems for each experiment with different number of prey. As before, each time ND-SHOP2 and MBP had a memory overflow or they could not solve a problem within out time limit, we ran them again on another problem of the same size. Each data point on which there were more than five such failures is omitted in this figure, but the data points where it happened 1 to 5 times are included.

The results in Fig. 10 demonstrate the power of combining HTN-based search-control strategies with BDD-based representations of states and policies in our planning problems: Yoyo was able to outperform both ND-SHOP2 and MBP. The running times required by MBP grow exponentially faster than those required by Yoyo with the increasing size of the prey,

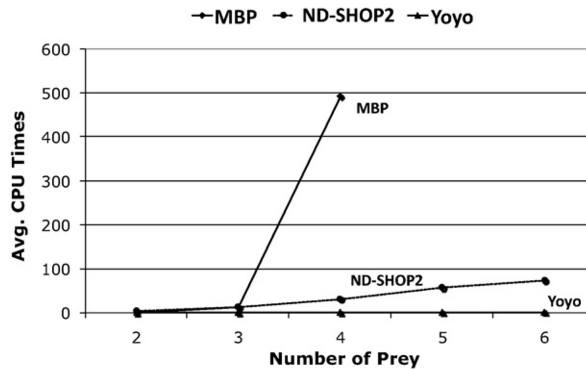

Fig. 10. Average running times in seconds for ND-SHOP2, Yoyo and MBP on problems in the Hunter–Prey domain as a function of the number of prey, with a 4×4 grid. MBP was not able to solve planning problems with 5 and 6 prey within 40 minutes.

Table 1
Average running times in seconds for MBP, ND-SHOP2, and Yoyo on Hunter–Prey problems with increasing number of prey and increasing grid size.

	2 prey		
Grid	MBP	ND-SHOP2	Yoyo
3×3	0.343	0.78	0.142
4×4	0.388	3.847	0.278
5×5	1.387	18.682	0.441
6×6	3.172	76.306	0.551
	3 prey		
Grid	MBP	ND-SHOP2	Yoyo
3×3	1.1	1.72	0.329
4×4	11.534	12.302	0.521
5×5	133.185	58.75	0.92
6×6	368.166	250.315	1.404
	4 prey		
Grid	MBP	ND-SHOP2	Yoyo
3×3	29.554	3.256	0.448
4×4	492.334	31.591	0.759
5×5	>40 mins	176.49	1.818
6×6	>40 mins	547.911	3.295
	5 prey		
Grid	MBP	ND-SHOP2	Yoyo
3×3	233.028	5.483	0.655
4×4	>40 mins	56.714	1.275
5×5	>40 mins	304.03	3.028
6×6	>40 mins	memory-overflow	7.059
	6 prey		
Grid	MBP	ND-SHOP2	Yoyo
3×3	2158.339	8.346	0.781
4×4	>40 mins	73.435	1.786
5×5	>40 mins	486.112	5.221
6×6	>40 mins	memory-overflow	11.826

since MBP cannot exploit HTN-based pruning heuristics. Note that ND-SHOP2 performs much better than MBP in these experiments.

Experimental Set 3. Our final set of experiments with the Hunter–Prey domain was designed to further investigate Yoyo's performance compared to that of ND-SHOP2 and MBP on problems with multiple prey and with increasing grid sizes. In these experiments, the number of prey were varied from $p = 2$ to $p = 6$, and the grid sizes were varied from $n = 3$ to $n = 6$.

Table 1 reports the average running times required by Yoyo, MBP, and ND-SHOP2 in these experiments. Each data point is an average of the running times of all three planners on 20 randomly-generated problems for each experiment with different p and n combinations. These results provide further proof for our conclusions in the previous experiments. Search-control strategies helped both Yoyo and ND-SHOP2, because they both outperformed MBP with an increasing number of prey. However, with increasing grid sizes, ND-SHOP2 ran into memory problems due to its explicit representations of states

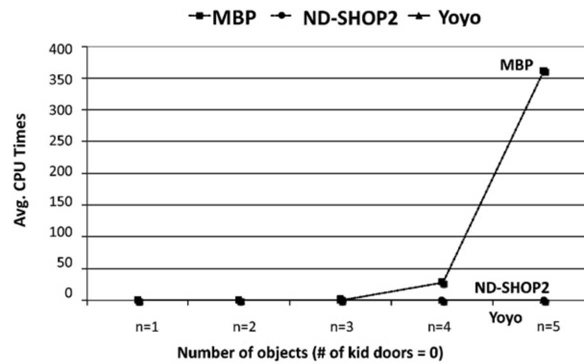

Fig. 11. Average running times in seconds for ND-SHOP2, Yoyo and MBP on problems in the Robot Navigation domain as a function of the number of packages, when none of the doors in the domain are kid doors (i.e., $k = 0$).

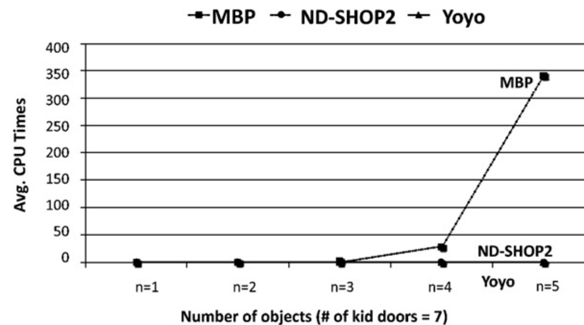

Fig. 12. Average running times in seconds for ND-SHOP2, Yoyo and MBP on problems in the Robot Navigation domain as a function of the number of packages, when all of the doors in the domain are kid doors (i.e., $k = 7$).

and solutions of the problems. Yoyo, on the other hand, was able to cope with very well both with increasing the grid sizes and the number of prey in these problems.

7.1.2. Robot Navigation

The Robot Navigation domain is a standard benchmark that has been used for experimental evaluation of MBP [2,12,13]. This domain is a variant of a similar domain described in [14]. It consists of a building with 8 rooms connected by 7 doors. In the building, there is a robot and there are a number of packages in various rooms. The robot is responsible for delivering packages from their initial locations to their final locations by opening and closing doors, moving between rooms, and picking up and putting down the packages. The robot can hold at most one package at any time. The domain's source of nondeterminism is a "kid" that can close any open doors that has been designated as a "kid-door".

We compared Yoyo, ND-SHOP2 and MBP with the same set of experimental parameters as in [12]: the number of packages n ranged from 1 to 5, and the number of kid-doors k ranged from 0 to 7. For each combination of n and k, we generated 20 random problems, ran the planners on the problems, and averaged the CPU time for each planner.[8]

All experiments in this domain were run on a MacBook laptop computer with 2.16 GHz Intel Core Duo processor, running Fedora Core 6 Linux on a virtual machine with 256 MB memory. For ND-SHOP2 and Yoyo, we used Allegro Common Lisp v8.0 Enterprise Edition in these experiments. The time limit was 1 hour.

Figs. 11 and 12 show the experimental results with varying number of objects $n = 1, \ldots, 5$ when there are no kid-doors (i.e., $k = 0$) and all of the doors are kid-doors (i.e., $k = 7$), respectively. In these experiments, both ND-SHOP2 and Yoyo was able to outperform MBP. Here are the reasons for our experimental results:

- **ND-SHOP2 vs. Yoyo.** In ND-SHOP2 and Yoyo, we used an HTN-based search-control strategy that tells these planners to "select a package, load it, and deliver it to its destination, while ignoring all the other packages. Once that package is delivered, select another one and deliver it, and continue with this process until all of the packages are delivered". This strategy induces a sub-strategy in HTNs for search control that we also encoded in our HTNs: that is, the robot only needs to consider the status of the door in front of it (i.e., whether the door that is in front of the robot and that the robot needs to go through in order to pick up or deliver the currently-selected package, while ignoring the status of

[8] As in [12], the CPU times for MBP's includes both its preprocessing and search times. Omitting the preprocessing times would not have significantly affected the results: they were never more than a few seconds, and usually below one second.

all of the other doors; and in this case, ND-SHOP2 generates state-action pairs in which the "state" actually represents a set of states, because some of the atoms are designated as "don't care": we don't care about any door other than the one in front of the robot. This compresses the state representations and the search space in a manner similar to a BDD; hence in these experiments ND-SHOP2's performance was similar to Yoyo's. In fact, ND-SHOP2's performance was slightly better Yoyo in these experiments, because ND-SHOP2 did not have the overhead incurred by Yoyo's BBD-based manipulation functionality.

- **MBP vs. ND-SHOP2 and Yoyo.** Unlike ND-SHOP2 and Yoyo, MBP was not able to exploit the HTN-based search-control strategies mentioned above, hence it needed to search a much larger search space. Furthermore, its backward breadth-first search considered many states that were not reachable from the initial states of the experimental problems. This hurt MBP performance significantly, despite its use of BDDs.

7.1.3. Nondeterministic Blocks World

The nondeterministic Blocks World domain contains the same state space and the same set of actions as the original Blocks World domain does, except that an action in this version may have its *intended* outcome that is the same outcome it has in the classical case but it may also drop the block on the table (e.g., in the case the gripper is slippery).

We compared Yoyo, ND-SHOP2, and MBP in the following experimental setup. We varied the number of blocks in the world $b = 3, \ldots, 10$. For each world with b many blocks, we randomly generated 20 planning problems and ran the planners on these problems, and measured the average CPU times of the planners.

We used the same experimental setup as in the previous section. In particular, we ran all our experiments on a MacBook laptop computer with 2.16 GHz Intel Core Duo processor, running Fedora Core 6 Linux on a virtual machine with 256 MB memory. For ND-SHOP2 and Yoyo, we used Allegro Common Lisp v8.0 Enterprise Edition in these experiments. The time limit was 1 hour.

Fig. 13 shows the results of these experiments. As before, both ND-SHOP2 and Yoyo were able to outperform MBP. The reason for ND-SHOP2 and Yoyo's outperforming results is that in nondeterministic Blocks World, there is an HTN strategy for these planners that says "if the gripper drops a block on the table, it should pick it up immediately before doing anything else". This strategy enabled the ND-SHOP2 and Yoyo to always generate polynomial-sized solutions, whereas MBP generated exponential-sized solutions since it could not use such HTN-based strategies.

As before, the BDD-based representations used in MBP's backward search algorithms did not help it much since in this domain, the solution policies generally require different actions for different states, rather than using the same action in a large set of states.

For example, whether we should unstack a block from the top of a tower may depend on whether we can move it to its goal position or whether there is another block underneath that can be moved to its goal position. MBP's backward search algorithms cannot use such knowledge to guide their search; ND-SHOP2 and Yoyo's HTNs, on the other hand, can easily encode such knowledge as a part of their search control.

In Fig. 13 it is difficult to compare Yoyo's and ND-SHOP2's CPU times because they both appear to be coincident with the x axis. To make the comparison easier, Fig. 14 gives a semi-logarithmic plot of the same data. This figure shows that both planners are running in polynomial time in this domain (curve-fitting shows that the running times of both planners are $\Theta(n^5)$, where n is the number of blocks). The reason that ND-SHOP2 runs faster is twofold. First, Yoyo's BDD manipulation operations require substantial overhead. Second, in the blocks world the BDD operations do not provide significant compression of the search space. On the other hand, the HTNs used in Yoyo and ND-SHOP2 worked quite well, hence both of these planners performed much better than MBP.

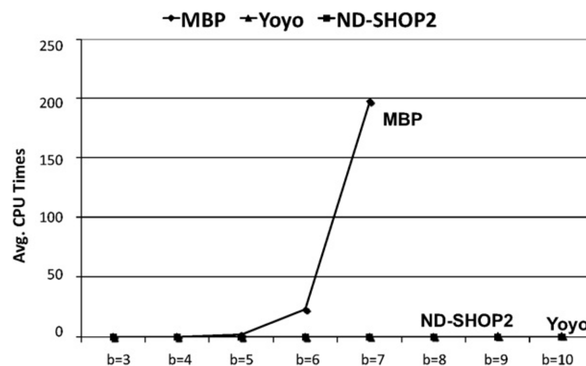

Fig. 13. Average running times in seconds for ND-SHOP2, Yoyo and MBP on problems in the nondeterministic Blocks World as a function of the number of blocks.

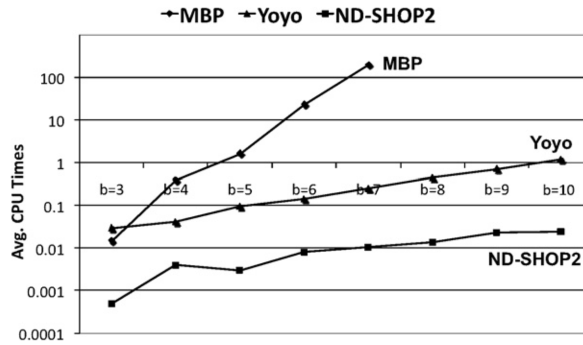

Fig. 14. A semi-log plot of the same data as in Fig. 13.

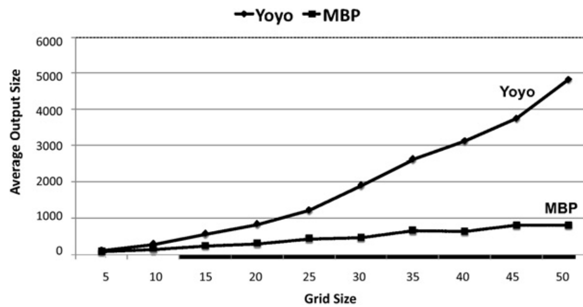

Fig. 15. Average solution sizes for Yoyo and MBP on Hunter–Prey problems with one prey and varying grid size.

7.2. Comparisons of the sizes of the planners' solutions

To the best of our knowledge, no good measure of solution quality has yet been devised for nondeterministic planning problems. One might like to measure something like a solution's average-case or worst-case execution length, but neither of these is possible. A solution's average-case execution length is undefined because there are no probabilities on the actions' outcomes. If a solution contains a cycle, then there will be infinitely many fair executions, with no finite bound on their length.

Lacking any better measure of plan quality, we measured the sizes of the solution policies produced by the planners. As our measure of size, we used the number of terms in the policy representations produced by the planners. In ND-SHOP2, this is the sum, over all state-action pairs in the policy, of the number of atoms in the state plus the additional term that specifies the action. In Yoyo and MBP, it is the number of propositions in the BDD representation of the policy (we don't need to count actions separately, because they're represented by propositions in the BDD).

Fig. 15 shows the sizes of the solutions generated by Yoyo and MBP in the Hunter–Prey problems with one prey and varying grid size. ND-SHOP2 does not appear in this figure for the same reason it didn't appear in Fig. 9: it was unable to handle grid sizes this big. These results show that MBP's solutions are smaller in size than those produced by Yoyo. The reason for this is that MBP has an option for doing boolean simplification on the BDDs it generates, and we used that option in our experiments.[9] Yoyo, although it is based on MBP's BDD implementations, does not use any boolean simplification mechanisms.[10]

Fig. 16 shows the sizes of the solutions generated by all three planners in Hunter–Prey problems with varying number of prey in a fixed 4×4 grid. Even with boolean simplification, MBP's solutions were quite large. The reason was that MBP needed to specify the locations of the prey explicitly because the prey's movements depended on each other (recall that a prey cannot move to a location that is next to another prey). Yoyo's solutions were much smaller because its HTNs enabled it to generate policies that focus on one prey at a time, ignoring the others.

Similarly, in the Robot Navigation and nondeterministic Blocks World domains, Yoyo generated smaller solution policies than MBP and ND-SHOP2, as shown in Figs. 17 and 18.

[9] We also tried running MBP with its boolean-simplification option turned off. In this case, MBP's solutions were 1 to 2 orders of magnitude larger than they had been with boolean simplification, and were 5 to 6 times larger than the solutions generated by Yoyo.

[10] To use MBP's built-in boolean simplification mechanisms for this purpose would have required a very tight integration of Yoyo and MBP, which would have made it hard for Yoyo to use HTNs in the way it does in the current design.

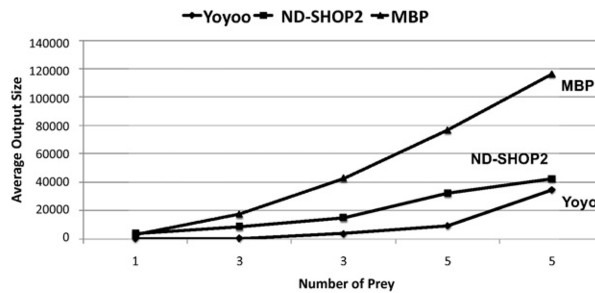

Fig. 16. Average solution sizes for ND-SHOP2, Yoyo and MBP on Hunter–Prey problems with varying numbers of prey on a 4×4 grid.

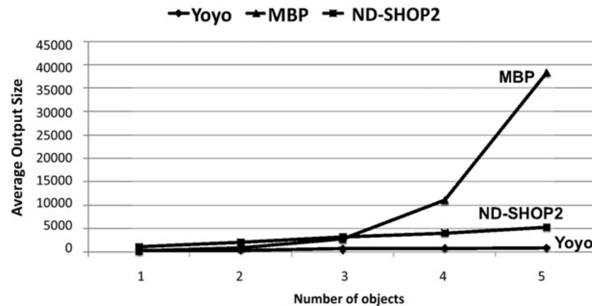

Fig. 17. Average solution sizes for ND-SHOP2, Yoyo and MBP on problems in the Robot Navigation domain as a function of the number of the objects, when all of the doors in the domain are kid doors (i.e., $k = 7$).

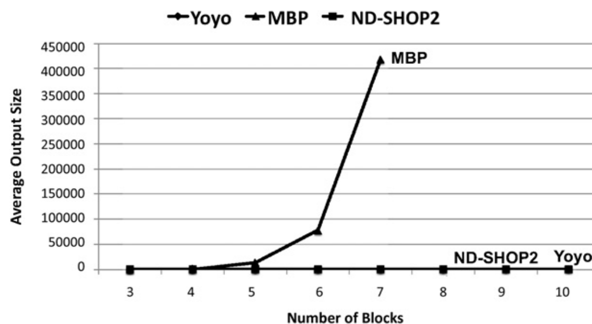

Fig. 18. Average solution sizes for ND-SHOP2, Yoyo and MBP on problems in the nondeterministic Blocks World domain as a function of the number of the blocks.

8. Related work

Probably the first work in a similar vein to ours is described in [15], which is a breadth-first search algorithm over an AND-OR tree that can be used to generate conditional plans and to interleave planning and execution in nondeterministic domains. Other early attempts to extend classical planning to nondeterministic domains include the CASSANDRA planning system [16], CNLP [17], PLINTH [18], UCPOP [19], and MAHINUR [20]. These algorithms do not perform as well as more modern planners such as MBP and ND-SHOP2. Furthermore, these conditional planning techniques usually generate solutions in the form of directed acyclic graphs; thus, they do not address the problem of infinite paths and of generating trial-and-error strategies.

QBFPLAN, a generalization of SATPlan [21], was introduced in [22]. QBFPLAN translates a nondeterministic planning problem into a satisfiability problem over *Quantified Boolean Formulas* (*QBFs*). The QBF problem is then fed to an efficient QBF solver such as the one described in [23]. QBFPLAN generates conditional plans that are bounded in length by a parameter specified as input. If a solution cannot be found within the current length, then the algorithm extends this bound and starts all over again. As its satisfiability based predecessors, QBFPLAN does not seem to scale up to large planning problems.

One of the earliest attempts to use model-checking techniques for planning under nondeterminism was first introduced in [14]. SIMPLAN, the planning system described in [14], is developed for generating plans in reactive environment, where such plans specify the possible reactions of the world with respect to the actions of the plan. Note that such reactions can be modeled as nondeterministic outcomes of the actions. SIMPLAN models the interactions between the environment and the execution of a plan by using a state-transition system, which specifies the possible evolutions of the environment due

to such interactions. Goals over the possible evolutions of the environment are specified by using *Linear Temporal Logics* (*LTL*) as in the classical TLPlan algorithm [24], which, in fact, takes its roots from SimPlan.

The SimPlan planner 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 techniques, such as Binary Decision Diagrams (BDDs), to do planning in nondeterministic domains under the assumptions of fully-observability and classical reachability goals was first introduced in [25,26].

BDDs enable a planner to represent a class of states that share some common properties and the planning is done by transformations over BDD-based representations of those states. In some cases, this approach can provide exponential reduction in the size of the representations of planning problems, and therefore, exponential reduction in the times required from those problems, as both demonstrated in this paper and in previous works [2,13].

The planning algorithms developed within this approach aim to generate solutions in nondeterministic planning domains that are classified as weak (at least one execution trace will reach a goal), strong (all execution traces will reach goals), and strong-cyclic (all "fair" execution traces will reach goals) [7–9]. [2] gives a full formal account and an extensive experimental evaluation of planning for these three kinds of solutions.

Planning as model checking has been extended to deal with partial observability [27,28]. In these works, belief states are defined as sets of states that represent common observations, and compactly implemented by using BDDs. Planning is done by performing a heuristic search over an AND-OR graph that represents the belief-state space. It has been demonstrated in [28] that this approach outperformed two other planning algorithms developed for nondeterministic domains with partial observability; namely GPT [29] and BBSP [30].

Planning with temporally extended goals in nondeterministic planning domains has been also investigated in several works, including [12,13,28,31]. The MBP planner [1] that is used as a benchmark in the experimental evaluations described in this paper is capable of handling both complex goals and partial observability.

Other planning algorithms that are based on model checking techniques include the UMOP planner, described in [32–34] is a symbolic model-checking based planning framework and a novel algorithm for strong and strong-cyclic planning which performs heuristic search based on BDDs in nondeterministic domains [34]. Heuristic search provides a performance improvement over the unguided BDD-based planning techniques on some toy examples (as demonstrated in [34]), but the authors also discuss how the approach would scale up to real-world planning problems.

Planning based on *Markov Decision Processes* (*MDPs*) [35] aims to solve planning problems with actions with more than one possible outcome, but this approach models those outcomes using probabilities and utility functions, and formulates the planning problems as optimization problems. In MDPs, a policy is usually a total function from states to actions, whereas model-checking approaches allow a policy to be partial. In problems that can be solved either by MDPs or by model-checking-based planners, the latter has been empirically shown to be more efficient [29].

Finally, several other approaches have been developed for planning under nondeterminism, mostly focusing on conditional and conformant planning. These approaches extended classical planning techniques based on planning graphs [36] and satisfiability [21]. Satisfiability based approaches, such as the ones described in [37–39], are limited to only *conformant planning*, where the planner has nondeterministic actions and no observability. The planning-graph based techniques [40–45] can address both conformant planning and a limited form of partial observability.

In [3], the authors present a generalization technique to transport the efficiency improvements that has been achieved for forward-chaining planning in deterministic domains over to nondeterministic case. Under certain conditions, they showed that a "nondeterminized" algorithm's time complexity is polynomially bounded by the time complexity of the classical (i.e., deterministic) version. ND-SHOP2 is an HTN planner developed using this technique for SHOP2 [5]. Yoyo, our HTN planner we described in this work, is built on both ND-SHOP2 and MBP.

9. Conclusions

We have described the Yoyo planning algorithm, which combines ND-SHOP2's HTN-based search-control with MBP's BDD-based symbolic model-checking techniques. We have presented theoretical results on Yoyo's soundness, completeness, and termination properties, and have compared it experimentally to both ND-SHOP2 and MBP. In our experiments, Yoyo always ran much faster than at least one of ND-SHOP2 and MBP, and usually ran faster than both of them.

It would also be possible to develop variants of Yoyo that do use temporal-logic control formulas such as those in TLPlan [24] and TALplanner [46], rather than using HTN decomposition. Appendix B discusses this in further detail, but here is a brief summary. Consider how the search-control works in TLPlan. In a state s, TLPlan computes all actions applicable to s such that $\gamma(s,a)$ satisfies the logical formula *Progress*(s, f), where f is a search-constraint formula [24]. Given a set S of states, a variant of Yoyo using TLPlan's search-control formulas would split the set S by generating an action a such that (1) a is applicable in some of the states in S and (2) all of the successor states must satisfy the formula *Progress*(s, f). This will require the formula f be a disjunction of smaller control-rule formulas, in which each disjunct addresses a possible outcome of an action or possible outcomes of a group of actions.

In the near future, we are interested in extending Yoyo to work with sets of HTN methods that are "incomplete" in the sense that they can only solve part of a planning problem rather than all of it. An analogous approach has recently been developed for classical planning, namely the Duet planner [47], which combines the SHOP2 HTN planner [5] with the LPG domain-independent planner [48]. Experimental studies have shown that even a small amount of HTN-planning

knowledge—much less than the amount that SHOP2 would need to work effectively by itself—can enable Duet to solve planning problems easily that LPG would have great difficulty solving. We believe a similar kind of approach (perhaps a combination of Yoyo and MBP) may work well in nondeterministic planning domains.

Acknowledgements

This work was supported in part by DARPA's Transfer Learning and Integrated Learning programs, NSF grant IIS0412812, AFOSR grants FA95500510298 and FA95500610405, and the FIRB-MIUR project RBNE0195k5, Knowledge Level Automated Software Engineering. The opinions in this paper are those of the authors and do not necessarily reflect the opinions of the funders.

Appendix A. Proofs of the theorems

This appendix presents the theorems stated in Section 6 and the proofs of those theorems.

Theorem 1. *Yoyo always terminates.*

Proof. The only possible situation in which Yoyo would not terminate is that the F set never becomes empty. However, this cannot happen because the state spaces of the planning problems are finite, and at the beginning of each invocation, the Yoyo removes the states that it already visited from the states of the F set. Therefore, Yoyo never visits a state more than once, and it does not caught in infinite search traces during planning. Thus, the theorem follows. □

The soundness and completeness of Yoyo depends on the soundness and completeness of the IsCandidate function, as Yoyo decides if a policy is a candidate solution based on this function and eliminates a policy for which this function returns FALSE.

Theorem 2. *Let $P = (\Sigma, S_0, G, \chi_0, M)$ be a nondeterministic HTN planning problem. Let $\bar{\pi}$ be a set-based policy and let $T_{\bar{\pi}}$ be the set of terminal states of $\bar{\pi}$.*

If $\bar{\pi}$ is a candidate solution for P, then IsCandidate$(\bar{\pi}, T_{\bar{\pi}}, G, S_0)$ returns TRUE*. Otherwise, IsCandidate$(\bar{\pi}, T_{\bar{\pi}}, G, S_0)$ returns* FALSE*.*

Proof. Note that the IsCandidate test shown in Fig. 5 does not involve checking HTN accomplishment; this test is designed only for verifying whether the candidate policy π satisfies the requirements of being a weak, strong, or strong-cyclic solution policy according to the definitions in Section 2.

Note also that the IsCandidate test always terminates since it traverses backwards only over the states of a policy $\bar{\pi}$, and $\bar{\pi}$ is always finite by definition. The proof of the theorem is in three parts and the proof of each part is by contradiction.

Weak planning. Suppose $\bar{\pi}$ is the input set-based policy to the IsCandidate test for weak planning. The algorithm performs a backward search via successive WeakPreimage computations, and in doing so, it generates all of the possible paths in the execution structure $\Sigma_{\bar{\pi}}$ that end in a terminal state. At the end of this traversal, IsCandidate computes the set S of the states such that from each state s in S, there is a path that reaches to a terminal state in the execution structure $\Sigma_{\bar{\pi}}$. If there is an initial state $s_0 \in S_0$ that is not in S, then this means that there is no execution starting from s_0 and ending in a terminal state in $\Sigma_{\bar{\pi}}$. In that case, IsCandidate returns FALSE. Otherwise, it returns TRUE. Note that if there is no execution that ends in a terminal state in $\Sigma_{\bar{\pi}}$ for an initial state s_0, this means that the policy $\bar{\pi}$ is not a candidate weak solution for the planning problem P.

To show that IsCandidate does not return any false positives and false negatives, suppose $\bar{\pi}$ is a candidate weak solution for P. By the definition of candidate weak solutions, $\bar{\pi}$ specifies one or more execution paths for each initial state $s_0 \in S_0$ that ends in a terminal state in $T_{\bar{\pi}}$. Assume that the invocation IsCandidate$(\bar{\pi}, S^t_{\bar{\pi}}, G, S_0)$ returns FALSE. The only case in which IsCandidate returns FALSE is when it traverses all paths in the execution structure $\Sigma_{\bar{\pi}}$ and detects that there exists at least one initial state s_0 in S_0 such that s_0 is not visited by the backward search. However, since we assumed that $\bar{\pi}$ is a candidate weak solution, this is a contradiction by the definition of candidate weak solutions.

Now suppose $\bar{\pi}$ is not a candidate weak solution and IsCandidate$(\bar{\pi}, S^t_{\bar{\pi}}, G, S_0)$ function returns TRUE. IsCandidate returns TRUE only if for each initial state $s_0 \in S_0$, there exists at least one path in the execution structure $\Sigma_{\bar{\pi}}$ that starts in s_0 and ends in a goal or non-goal terminal state. However, since $\bar{\pi}$ is not a candidate solution, there must be at least one initial state for which this does not hold, a contradiction.

Thus, the theorem follows for weak planning.

Strong planning. The IsCandidate computation for strong planning is similar to that for weak planning in that it generates all of the execution paths in the execution structure Σ_π induced by π by doing a backward search from the terminal states

of $\Sigma_{\bar{\pi}}$ towards the initial states in S_0. The difference in the IsCandidate computations for strong planning and weak planning is that in the former case, IsCandidate does its backward search by performing successive StrongPreimage operations.

During the backward search, IsCandidate removes the state-action pairs that it visits from the policy π. At the end of this search, if there is a state-action pair, say (s,a), left in $\bar{\pi}$ then this means that the state s has a successor state s' such that s' is a π-ancestor of s. Thus, there is a cyclic path in the execution structure $\Sigma_{\bar{\pi}}$ and IsCandidate returns FALSE and this is correct by the definition of strong solutions to planning problems.

At the end of its backward traversal, IsCandidate generates the set S of states in $\bar{\pi}$ such that from each state s in S, there is a path that reaches to a terminal state in the execution structure $\Sigma_{\bar{\pi}}$. As in the case of weak planning, if there exists at least one initial state s_0 in S_0 such that s_0 is not in S, the IsCandidate returns FALSE.

If neither of the above checks hold, then IsCandidate returns TRUE. This is correct since if both of the two checks above do not fail for the input policy $\bar{\pi}$ then, by definition, $\bar{\pi}$ is a candidate strong solution.

To show that IsCandidate does not return any false positives and false negatives, suppose $\bar{\pi}$ is a candidate strong solution for P. Assume that the invocation IsCandidate$(\bar{\pi}, S^t_{\bar{\pi}}, G, S_0)$ returns FALSE. This means that either or both of the two cases mentioned above must be false given the input policy $\bar{\pi}$ and the initial states S_0. However, by the definition of candidate strong solution, this cannot happen; hence, we have a contradiction.

Now suppose $\bar{\pi}$ is not a candidate strong solution and IsCandidate(π, S^t_{π}, G, S_0) function returns TRUE. If IsCandidate returns TRUE, then both of the checks above must by satisfied by $\bar{\pi}$. However, since $\bar{\pi}$ is not a candidate solution, we again have a contradiction.

Therefore, the theorem follows for strong planning.

Strong-cyclic planning. The proof for this case is the same as the one for the strong-planning case, except that the IsCandidate function only returns FALSE for cyclic paths that do not have any possibility to reach to the goals. □

Theorem 3. *Suppose $P = (\Sigma, S, G, \chi, M)$ is a nondeterministic HTN planning problem.*

(1) *If Yoyo$(F_0, G, M, \bar{\pi}_0)$, where $F_0 = \{(S, \chi)\}$ and $\bar{\pi}_0 = \emptyset$, returns a policy π for P, then π weakly, strongly, or strong-cyclicly accomplishes χ from S using the methods in M.*
(2) *If there is no solution policy for P that accomplishes χ from S given M, then Yoyo returns* FAILURE.

Proof. Let $\bar{\pi}$ be the set-based version of the policy π. Before going into the proof, we remark that although every time Yoyo generates a pair (S,t), it first updates the partial policy that is input to the current invocation and then passes the updated policy to the next invocation, this is equivalent to take the pair (S,t), update the partial policy returned by a recursive invocation, and return the updated policy.[11] Thus, in the following, we establish the proof with the latter view of Yoyo's updates in mind.

Case 1. The proof of this case is in two parts. First suppose that Yoyo$(F_0, G, M, \bar{\pi}_0)$ returns the empty policy. Then, by the definition of the Yoyo planning algorithm, $S \subseteq G$ and χ must be the empty HTN. The proof is immediate: the correctness checks in Lines 1–6 in the pseudo-code of Fig. 4 make sure that if every initial state in S is also a goal state and the task network χ is the empty HTN, then Yoyo returns the empty policy in Line 7.

Now suppose Yoyo$(F_0, G, M, \bar{\pi}_0)$ returns a nonempty policy. The proof is by induction on n, the number of invocations required by Yoyo in order to return $\bar{\pi}$.

Base Case ($n = 1$). In this case, the invocation Yoyo$(F_0, G, M, \bar{\pi}_0)$ returns the policy $\bar{\pi}$. This means that each initial state of P is actually a goal state. Then, since Yoyo did not return FAILURE at this point at Line 2, then we also have that χ is the empty HTN. Thus, Yoyo would remove the initial pair (S, χ) from F_0 in Line 3, leaving $F = \emptyset$ and $\bar{\pi} = \bar{\pi}_0 = \emptyset$. So, it will return the empty $\bar{\pi}$ at Line 7, which accomplishes the empty HTN χ according to our HTN accomplishment definition in Section 2.2.

Induction step. Let $n > 1$ and suppose that the theorem is true for every $k < n$. Let $\bar{\pi}'$ be the current policy and let (S, χ) is the pair Yoyo selected from F in this invocation. We have two possibilities.

- One possibility is that Yoyo updates the F set in Line 13 and goes to the next iteration. In this case, the current task t in χ is primitive and Yoyo updates the F set with $(\bar{\gamma}(S',t), \tau(\chi,s,t))$ for the largest subset S' of S in which t is applicable, and recursively invokes itself. Suppose this recursive invocation returns a policy $\bar{\pi}''$. There are two cases:
  - for every state s that appear in any pair the fringe F that was the input to the returned invocation, there is a state-action pair (s,a) in $\bar{\pi}''$ for some action a. Then, by the induction hypothesis, we know that $\bar{\pi}'' - (\bar{\pi}' \cup \{(S,t)\})$ accomplishes $\tau(\chi,s,t)$ from $\bar{\gamma}(S,t)$ using the methods in M. Thus, the partial policy $\bar{\pi}'' \cup \{(S,t)\}$ accomplishes the current HTN χ.
  - Otherwise, Yoyo calls itself recursively again with the updated fringe F (Line 27), where the states for which the policy $\bar{\pi}$ are removed from the pairs of F. Suppose this recursive invocation returns a policy, say $\bar{\pi}'''$. Then, same

[11] The reason that Yoyo first updates the partial policy and then makes a recursive invocation with it is to be able to check candidacy over that policy.

as above, we know that $\bar{\pi}''' - (\bar{\pi}' \cup \{(S,t)\})$ accomplishes $\tau(\chi, s, t)$ from $\bar{\gamma}(S', t)$ using the methods in M and $\bar{\pi}''' - (\bar{\pi}'' \cup \{(S,t)\})$ accomplishes χ in $S - S'$.

- The other possibility is that Yoyo updates the F set in Line 18 and goes to the next iteration. In this case, the current task t is not primitive and Yoyo updated the F set with two pairs $(S', \delta(\chi, s, m, t))$ and $(S - S', \chi)$ where $m \in M$ is a method that is applicable to t and S' is the subset of S such that the applicability-conditions of m are satisfied in all states in S'. By the induction hypothesis, if Yoyo$(F, G, M, \bar{\pi}')$ returns a policy $\bar{\pi}''$, then $\bar{\pi}''$ accomplishes both $\delta(\chi, s, m, t)$ from S' and χ from $S - S'$, using the methods in M.

Therefore, Case 1 of the theorem follows.

Case 2. We define the *decomposition trace* of χ as the sequence of primitive and nonprimitive task decompositions necessary by an HTN planner to generate $\bar{\pi}$. Here, the decomposition of a primitive task is the application of that task in a state during the planning process.

The proof is by contradiction. Suppose there is a solution policy $\bar{\pi}$ for the planning problem P that accomplishes the input HTN χ in the initial states S by using the methods in M. First, note that since $\bar{\pi}$ is a solution policy, any partial policy in the successive invocations of Yoyo would be a candidate policy, and therefore, Yoyo would never return FAILURE in Line 6 by Theorem 2.

We show that when Yoyo returns a failure, then there is no solution policy for P that accomplishes χ from S given M. Suppose Yoyo returns FAILURE. This means that every nondeterministic trace of this invocation of Yoyo returns FAILURE. There are four cases in which Yoyo could return FAILURE:

- There is a pair (S, χ) in F such that every state in S is a goal state, but χ is not the empty HTN. This means that χ cannot be accomplished from S because the policy $\bar{\pi}$ would never specify an action for any state in S.
- After Yoyo removes all the goal states from the states of F, there is a pair (S, χ) left in F such that χ is the empty HTN. However, the policy $\bar{\pi}$ would specify an action for a state in s if $\bar{\pi}$ is a solution for P. Thus, $\bar{\pi}$ does not accomplish the HTN χ, by the definition of accomplishing an HTN given in Section 2.2.
- Yoyo selected a task t to accomplish in a set of states S and t is a nonprimitive task. In this case, Yoyo could return FAILURE if there is no method $m \in M$ that is applicable to t in S. This means that (1) there is no method in M for t, or (2) the applicability-conditions of all of the methods in M for t are not satisfied in any of the states in S, or (3) there is no set of methods for t such that the applicability conditions of each such method describes a subset of S and the union of all such subsets is equal to S itself. Thus, χ cannot be accomplished from S, by the definition of accomplishing an HTN.
- A recursive invocation of Yoyo returned FAILURE in Line 26. This could happen only when Yoyo returns FAILURE in any of the above cases.

If the policy $\bar{\pi}$ accomplishes the input HTN χ, this means that there should be at least one nondeterministic trace of Yoyo in which none of the cases above would happen since Yoyo considers all possible ways of decomposing a task during planning given the HTN χ and the set M of methods. Note that such a nondeterministic trace would include both recursive invocations of Yoyo in Lines 25 and 29. Therefore, if Yoyo returns FAILURE, then there is no such nondeterministic trace given the HTN methods in M, and thus, there is no solution policy $\bar{\pi}$ that accomplishes the initial HTN χ from the initial states S given the set of HTN methods M. □

The following theorem establishes the soundness of the Yoyo planning procedure:

Theorem 4. *Suppose Yoyo returns a policy π for a nondeterministic HTN planning problem $P = (\Sigma, S_0, G, \chi, M)$. Then π is a solution for P.*

Proof. Let $\bar{\pi}$ be the set-based version of π and suppose one of the nondeterministic traces of Yoyo returns $\bar{\pi}$ for P. The proof is by induction on n, the number of invocations of Yoyo, given P, the initial HTN χ_0 and a set of methods M.
Base Case ($n = 1$). In this case, Yoyo does not perform another recursive invocation, so it must return in Lines 2, 5, or 6 of the pseudo-code shown in Fig. 4 since it did not return FAILURE. Thus, we must have $F = \emptyset$ and $\bar{\pi} = \emptyset$. This means that each initial state of P is a goal state, and therefore, the initial pair (S_0, χ_0) is removed from F in Line 3. So, by the definition of a solution of a nondeterministic planning problem, $\bar{\pi}$ is a solution for P.
Induction step. Let $n > 1$ and suppose that the theorem is true for every $k < n$. By Theorem 2, we know that the partial policy in every previous invocation was a candidate solution since the IsCandidate tests did not fail and Yoyo did not return FAILURE.

Then we have two possibilities. Let $\bar{\pi}'$ be the current policy in this invocation.

- One possibility is that Yoyo updates the F set in Line 13 and goes to the next invocation. In this case, the current task t is primitive. Suppose S' is the largest subset of S in which t is applicable. This means that Yoyo updated $\bar{\pi}'$ with (S', t) and the F set with $(\bar{\gamma}(S', t), \tau(\chi, s, t))$ for some state s in S'.

- The other possibility is that Yoyo updates the F set in Line 18 and goes to the next iteration. In this case, the current task t is not primitive and Yoyo updated the F set with two pairs $(S', \delta(\chi, s, m, t))$ and $(S - S', \chi)$ where $m \in M$ is a method that is applicable to t and S' is the subset of S such that all the states in S' are (t, m)-equivalent.

Suppose Yoyo generated the policy $\bar{\pi}$ in the recursive invocation at Line 25. For every HTN χ that appears anywhere in the F set, let $P_\chi = (\Sigma, S_\chi, G_\chi)$ be a nondeterministic planning problem where Σ is the current planning domain, and

$$S_\chi = \bigcup_{(S,\chi)\in F} S \cap S_{\bar{\pi}},$$

and

$$G_\chi = G \cup S_{\bar{\pi}}.$$

In the above, recall that $S_{\bar{\pi}}$ is the set of all states in the set-based policy $\bar{\pi}$.

From the induction hypothesis, we know that the policy $\bar{\pi}_\chi$ generated by Yoyo for each nondeterministic planning problem P_χ is a solution for P_χ. That is, for each P_χ, there must be a nondeterministic trace of Yoyo that returns a policy $\bar{\pi}_\chi$ such that $\bar{\pi}_\chi \subseteq (\bar{\pi} - \bar{\pi}')$; since, otherwise, Yoyo could not return $\bar{\pi}$ for P.

By the above construction, if $S - S_{\bar{\pi}}$ is not the empty set a pair (S, χ) in F then the policy $\bar{\pi}$ is not defined in the states of $S - S_{\bar{\pi}}$ and Yoyo must plan for those states in order to generate a solution. In Lines 26–27, the planner updates the fringe set F as follows: if $(S, \chi) \in F$ and $S - S_{\bar{\pi}} \neq \emptyset$ then Yoyo replaces the pair (S, χ) with $(S - S_{\bar{\pi}}, \chi)$.

For every HTN χ that appears anywhere in the updated F set, let $P''_\chi = (\Sigma, S_\chi, G_\chi)$ be a nondeterministic planning problem where Σ is the current planning domain, and

$$S''_\chi = \bigcup_{(S'',\chi)\in F} S'' - S_{\bar{\pi}},$$

and

$$G''_\chi = G \cup S_{\bar{\pi}}.$$

And suppose the Yoyo recursively called itself on these planning problems in Line 29 and returned the policy $\bar{\pi}''$. By the reasoning above, we know that the policy, say $\bar{\pi}''_\chi$, generated by Yoyo for each nondeterministic planning problem P''_χ is a solution for P''_χ. Since the current partial policy $\bar{\pi}$ is a candidate solution and $\bar{\pi}'' - \bar{\pi}$ is a solution (since it is the union of all $\bar{\pi}''_\chi$) for the planning problems from the set of the terminal states of $\bar{\pi}$, $\bar{\pi}''$ must be a solution for the original input planning problem P. □

Finally, the following theorem establishes the completeness of Yoyo.

Theorem 5. *Suppose $P = (\Sigma, S, G, \chi_0, M)$ is a solvable nondeterministic HTN planning problem. Then, Yoyo returns a policy π for P that weakly, strongly, or strong-cyclicly solves P.*

Proof. Let π be a solution for P. We will use $\bar{\pi}$ to denote the set-based version of π in the rest of the proof. Then, by definition, π accomplishes the HTN $\chi_0 = (T, C)$ given the set M of methods. We define the *decomposition graph* for π. The decomposition graph for π is a directed graph $R_\pi = (N, E)$. Each node in N is a tuple of the from (S, χ) such that S is a set of states and χ is an HTN. Each edge in E describes a decomposition from a node (S, χ) to (S', χ'). Based on the definition of HTN accomplishment given earlier, the decomposition graph R_π has the following properties:

(1) If χ_0 is the empty HTN (i.e., $T = \emptyset$) then R_π has a single node (S, χ_0) such that $S \subseteq G$, and E is the empty set. We call the node (S, χ_0) a *leaf* node of R_π; i.e., (S, χ_0) does not have any outgoing edges.
(2) There is a primitive task (i.e., an action) t that has no predecessors in T. Let S' be the largest subset of S in which t is applicable. Then, there is an edge from the node (S, χ_0) to the node $(\gamma(S', t), \tau(\chi_0, S, t))$. The edge from (S, χ_0) to $(\gamma(S', t), \tau(\chi_0, S, t))$ is labeled by the action t and we say that $(\gamma(S', t), \tau(\chi_0, S, t))$ is a child of (S, χ_0) in R_π.
(3) There is a nonprimitive task t that has no predecessors in T and there is a method $m \in M$ that is applicable to t in at least one of the states in S. Let S' be the largest subset of S in all of which m is applicable to t. Then, there is an edge from the node (S, χ) to the node $(S', \delta(\chi, S', m, t))$ and we say that $(S', \delta(\chi, S', m, t))$ is a child of (S, χ) in R_π. The edge from $(S', \delta(\chi, S', m, t))$ to (S, χ) is labeled with the pair (t, m).

Let R_π be decomposition graph of π given to Yoyo. Since only the primitive tasks (i.e., actions) produce new states, it follows that the candidate solution $\bar{\pi}$ consists of the set of (S', t) pairs such that (1) (S, χ) is a node in R_π, (2) $S' \subseteq S$ in which t is applicable, and (3) the action t is the label of the outgoing edge from (S, χ); otherwise, there is at least one path in π that does not accomplish χ, which is a contradiction.

The proof of the theorem is as follows. Suppose the decomposition graph R_π contains a single leaf node (S, χ) where χ is the empty HTN. This means that $S \subseteq G$, and $\bar{\pi} = \emptyset$. Indeed, in its initial iteration, the failure conditions at Lines 1, 4, and

6 do not hold, and Yoyo removes all of the states from F (see Line 3) and leaves the set F as the empty set. Then, it must return the empty policy as a solution in Line 7.

Otherwise, suppose R_π contains more than one leaf node. Take any subset N' of the nodes in R_π such that removing the nodes in N' disconnects the graph R_π – i.e., the outgoing edges from the nodes N' forms a minimal cutset of R_π. There must be a nondeterministic trace of Yoyo such that the following holds. Let F be the fringe set in one of the invocations of the Yoyo. Then, for each (S', χ) in F there must be one and only one node (S, χ) in N' such that $S' \subseteq S$. If there is no such nondeterministic trace, then by Theorem 3, this means that R_π cannot be a solution decomposition trace (since the nondeterministic choice of a pair (S', m) for a nonprimitive task t in Line 16 of the Yoyo algorithm ensures, in principle, that the planner will search for every possible choice of decomposition of t eventually).

Let (S, χ) be one of the nodes in the fringe set F at the nondeterministic trace above. If (S, χ) is a leaf node then the failure conditions in Lines 1 and 4 of the Yoyo pseudo-code do not hold. Since R_π is a solution decomposition graph, i.e., since π is a solution policy, Yoyo does not return FAILURE in Line 6 as well, by Theorem 2. Instead, Yoyo must remove this node from F in Line 3 since $S \subseteq G$.

Suppose (S, χ) is a non-leaf node. The failure conditions at Lines 1, 4, and 6 must not hold for the same reasons above. There are two cases. If (S, χ) has only one outgoing edge labeled by an action t, then this means that the primitive task t does not have any predecessors in χ and t is applicable in $S' \subseteq S$. Thus, Yoyo will choose that task in Line 9 and update F with the only child of t in Line 13.

If (S, χ) has one or more outgoing edges that are labeled with a pair of a nonprimitive task and a method m for that task, then in Line 16, Yoyo will choose the pair (S', m) where S' is the set of states that appear in the child of (S, χ) through the edge (t, m). Thus, Yoyo does not return FAILURE in Line 17.

In either of the above cases, suppose $S - S'$ is not the empty set. This means that the current decomposition followed by Yoyo will not generate a policy for the states in $S - S'$ through the decomposition path of R_π. However, after Yoyo finishes the current decomposition path R_π, it will backtrack until it reaches a node (S'', χ) in R_π such that $S - S' \subseteq S''$ and consider other decomposition paths from that node (see the second recursive invocation in Line 29 of the pseudo-code). Since Yoyo will backtrack over all possible such nodes for all possible states in $S - S'$ and since R_π defines a decomposition path for all states in S (otherwise, π would not be a solution), the theorem follows. □

Appendix B. On the use of control rules in Yoyo

In Yoyo, we only focused on combining HTN-based search control with BDD-based representations. However, it is also possible to develop variants of Yoyo, designed to work with search-control techniques other than HTNs. In particular, we have written pseudo-code for algorithms that are similar to Yoyo but whose search control is done not with HTN decomposition, but instead with temporal-logic control formulas such as those in TLPlan [24] and TALplanner [46].

As an example, consider how the search-control works in TLPlan. In a state s, TLPlan computes every action a applicable to s such that the logical formula *Progress*(s, f), where f is a search-constraint formula, is not evaluated to be false in the next state $\gamma(s, a)$ [24]. Given a set S of states, a variant of Yoyo using TLPlan's search-constraint formulas would split the set S by generating an action a such that (1) a is applicable in some of the states in S and (2) the formula *Progress*(s, f) is not false in at least one of the successor states. In the states of S in which the above conditions would not hold for the current action a and the formula f, Yoyo would choose another action for those states and proceed from their successors.

Both HTN-based and control-rule based search control requires a domain expert to author the control knowledge as an input to Yoyo (or to any planner that is able to use such knowledge). The effectiveness of the control knowledge, i.e., how much improvement it provides to the planner in terms of time performance and of pruning the search space, depends on the expertise of the domain expert as well as the properties of the solutions to planning problems in the underlying domain.

Although there are planning domains in which it takes substantial effort to write effective search control, in many domains this process is usually intuitive and rather simple. For example, in the Hunter–Prey domain, our search control that describes a strategy "focus on a prey and ignore the others; always move toward the prey under focus" helped Yoyo generate solution policies very efficiently.

Sometimes, the search control knowledge needs to specify implied properties of solution policies in a planning domain. For example, in the nondeterministic Blocks World, the control that tells the planner "if you drop a block on the table, pick it up immediately" is an important piece of information and it is implied by the fact that if the planner does not a pick a block that it dropped, then the search space is going to be exponential in the number of states explored.

It is possible to write the same search-control strategy using either HTNs or control rules. As an example, Fig. B.1 shows one of our HTN methods that we used in our experiments with the nondeterministic Blocks World domain, as described in Section 7. This method is written for moving a block x from the top of another block y on to a block z. The method is applicable when the gripper is empty, the block x is clear (i.e., there is no other block on top of x), x is on y in the current configuration, the block z is clear so that we can move x on top of z, the goal position of x is on top of z, and z is the top of a good tower.

If the applicability conditions of the method holds in a state, then there are two subtasks to be done in order to move x from y to z. The first subtask is the action that unstacks x from the top of y. The unstack operation is nondeterministic: a possible effect of unstack is that the gripper is holding the block x; another possible outcome is that the gripper dropped

```
method for moving x from y to z
  for the task solve_ndbw:
  applicability conditions: if the gripper is empty and there is a block x
                that can be moved from its current position
                on the block y to its goal position on the block z
                and z is the top of a good tower, then
  subtasks: (1) unstack x from y,
            (2) check the outcome of unstack and proceed accordingly
  constraints: do subtask (1) before subtask (2)
```

Fig. B.1. An HTN method for moving a block from the top of one block to another in the nondeterministic Blocks World domain.

```
Method #1 for checking the outcome of unstack
  for the task check_unstack
  applicability conditions: If the gripper is holding the block, then
  subtasks: (1) stack x on z,
            (2) check the outcome of stack and proceed accordingly
  constraints: do subtask (1) before subtask (2)

Method #2 for checking the outcome of unstack
  for the task check_unstack
  applicability conditions: If the gripper is not holding the block, and
            instead, it is on the table then
  subtasks: (1) pick x up from the table,
            (2) stack x on z,
            (3) check the outcome of stack and proceed accordingly
  constraints: do subtask (1) before subtask (2) and
            do subtask (2) before subtask (3)
```

Fig. B.2. Two HTN methods for checking the outcome of the nondeterministic unstack action.

the block on the table. The second subtask of the method checks which outcome of the unstack has occurred when the plan is later executed and determines the next action.

Fig. B.2 shows the method for checking the outcome of the unstack action and for planning for the next action accordingly. If the gripper is holding the block x, then the planner simply stacks it to its goal position on top of the block x. Otherwise, if the gripper dropped the block x, this method tells the planner to pick it up immediately before doing any other action and to attempt to stack it later.

A possible control rule in the temporal-logic formalism of TLPlan, which corresponds to the set of HTNs shown in Figs. B.1 and B.2, can be written as follows. First, consider the search-control formula described for TLPlan in [24]:

$$\begin{aligned}
\chi :\ & \Box(\forall[?x : clear(?x)]goodtower(?x) \\
& \Rightarrow \odot(clear(?x) \vee \exists[?y : on(?y, ?x)]goodtower(?y)) \wedge \\
& \quad badtower(?x) \Rightarrow \odot(\neg\exists[?y : on(?y, ?x)]) \wedge \\
& \quad (on(?x, table) \wedge \exists[?y : GOAL(on(?x, ?y))]\neg goodtower(?y)) \\
& \qquad \Rightarrow \odot(\neg holding(?x))).
\end{aligned}$$

One can write similar control rules for checking the possible nondeterministic outcomes of an action. The following show the example control rules for unstack and stack:

$$\begin{aligned}
\chi^{unstack}_{dropped} :\ & \Box(on(?x, ?y) \wedge clear(?x) \wedge handempty \wedge \exists[?z : GOAL(on(?x, ?z))] \\
& \wedge goodtower(?z) \wedge \odot(ontable(?x))) \Rightarrow \odot\odot\, holding(?x). \\
\chi^{stack}_{dropped} :\ & \Box(holding(?x) \wedge \exists[?z : GOAL(on(?x, ?z))] \wedge goodtower(?z) \\
& \wedge \odot(ontable(?x))) \Rightarrow \odot\odot\, holding(?x).
\end{aligned}$$

The above formula above specifies the condition when the gripper drops the block on the table as a result of the unstack action and what needs to be true in the world if that happens. More specifically, the conclusions of the implication above states that if a nondeterministic unstack action drops the block on the table, then the gripper must be holding the block in the state immediately following that outcome state, which can only be satisfied by picking up that block from the table.

Theoretically, there is no doubt that a general translation exists between HTNs and control rules, because both formalisms are turing-complete. In principle, such a translation could be developed by performing reductions such as the ones in turing-completeness proofs such as the one in [49]. But it's doubtful that such a translation would look particularly natural to the human eye; and as far as we know, there has not been any work on how to develop a more natural translation.

There certainly may be nondeterministic planning domains in which it is hard to write search-control knowledge, specified as either HTNs or control rules. In such domains, it make sense that the planning algorithm should be capable of using the search-control knowledge whenever that knowledge is available, but still be able to continue planning where it is not. Researchers have argued that using control-rules in a planning algorithm has the advantage that planners that use control-rules can fall back to systematic search when those rules are not available, and therefore, more flexible than HTNs in complex environments. Although Yoyo (or ND-SHOP2) cannot continue planning when HTNs are not completely specified, it is very easy to get any HTN planner to fall back to a forward-chaining search by using the following method template in its input: for each action a in the planning domain,

> Method m for a task for which we do not have search control knowledge
> *applicability conditions*: None
> *subtasks*: (1) do the action a
> (2) invoke the method m recursively
> *constraints*: do subtask (1) before subtask (2)

For example, using the class of methods defined by the above template for each task for which search-control knowledge is not specified in the input, Yoyo would simply perform a forward search over the BDD representations of sets of states until it generates the goals states. Note that the above template is *domain independent*: for any planning domain, we can instantiate it by the actions (i.e., planning operators) specified for that domain.

In summary, it is an interesting research topic to investigate an in-depth analysis of HTNs and control-rules with respect to each other in both classical and nondeterministic planning domains, however this investigation is beyond the scope of this paper and we leave it to a future study.

References

[1] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, P. Traverso, MBP: A model based planner, in: IJCAI-2001 Workshop on Planning under Uncertainty and Incomplete Information, Seattle, USA, 2001.
[2] A. Cimatti, M. Pistore, M. Roveri, P. Traverso, Weak, strong, and strong cyclic planning via symbolic model checking, Artificial Intelligence 147 (1–2) (2003) 35–84.
[3] U. Kuter, D. Nau, Forward-chaining planning in nondeterministic domains, in: AAAI-2004, 2004.
[4] R.E. Bryant, Symbolic boolean manipulation with ordered binary-decision diagrams, ACM Computing Surveys 24 (3) (1992) 293–318.
[5] D. Nau, T.-C. Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu, F. Yaman, SHOP2: An HTN planning system, JAIR 20 (2003) 379–404.
[6] S. Koenig, R.G. Simmons, Real-time search in non-deterministic domains, in: IJCAI-1995, 1995.
[7] A. Cimatti, M. Roveri, P. Traverso, Automatic OBDD-based generation of universal plans in non-deterministic domains, in: AAAI/IAAI Proceedings, 1998, pp. 875–881.
[8] A. Cimatti, M. Roveri, P. Traverso, Strong planning in non-deterministic domains via model checking, in: Proceedings of the International Conference on AI Planning Systems (AIPS), AAAI Press, 1998, pp. 36–43.
[9] M. Daniele, P. Traverso, M. Vardi, Strong cyclic planning revisited, in: Proceedings of the European Conference on Planning (ECP), 1999, pp. 35–48.
[10] M. Ghallab, D. Nau, P. Traverso, Automated Planning: Theory and Practice, Morgan Kaufmann, 2004.
[11] N. Nilsson, Principles of Artificial Intelligence, Morgan Kaufmann, 1980.
[12] M. Pistore, P. Traverso, Planning as model checking for extended goals in non-deterministic domains, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), Seattle, USA, Morgan Kaufmann, 2001, pp. 479–484.
[13] M. Pistore, R. Bettin, P. Traverso, Symbolic techniques for planning with extended goals in non-deterministic domains, in: Proceedings of the European Conference on Planning (ECP), 2001.
[14] F. Kabanza, M. Barbeau, R. St-Denis, Planning control rules for reactive agents, Artificial Intelligence 95 (1) (1997) 67–113.
[15] M. Genesereth, I. Nourbakhsh, Time-saving tips for problem solving with incomplete information, in: Proceedings of the National Conference on Artificial Intelligence (AAAI), 1993.
[16] L. Pryor, G. Collins, Planning for contingency: A decision based approach, Journal of Artificial Intelligence Research 4 (1996) 81–120.
[17] M. Peot, D. Smith, Conditional nonlinear planning, in: Proceedings of the International Conference on AI Planning Systems (AIPS), 1992, pp. 189–197.
[18] R.P. Goldman, M.S. Boddy, Conditional linear planning, in: Proceedings of the International Conference on AI Planning Systems (AIPS), 1994.
[19] J.S. Penberthy, D. Weld, UCPOP: A sound, complete, partial order planner for adl, in: Proceedings of the International Conference on Knowledge Representation and Reasoning (KR), 1992.
[20] N. Onder, M.E. Pollack, Conditional, probabilistic planning: A unifying algorithm and effective search control mechanisms, in: Proceedings of the National Conference on Artificial Intelligence (AAAI), 1999, pp. 577–584.
[21] H. Kautz, B. Selman, Planning as satisfiability, in: Proceedings of the European Conference on Artificial Intelligence (ECAI), 1992, pp. 359–363.
[22] J. Rintanen, Constructing conditional plans by a theorem-prover, Journal of Artificial Intelligence Research 10 (1999) 323–352.
[23] J. Rintanen, Improvements to the evaluation of quantified boolean formulae, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), Stockholm, Sweden, Morgan Kaufmann, 1999, pp. 1192–1197.
[24] F. Bacchus, F. Kabanza, Using temporal logics to express search control knowledge for planning, Artificial Intelligence 116 (1–2) (2000) 123–191.
[25] A. Cimatti, E. Giunchiglia, F. Giunchiglia, P. Traverso, Planning via model checking: A decision procedure for AR, in: Proceedings of the European Conference on Planning (ECP), in: Lecture Notes in Artificial Intelligence (LNAI), vol. 1348, Toulouse, France, Springer-Verlag, 1997, pp. 130–142.
[26] F. Giunchiglia, P. Traverso, Planning as model checking, in: Proceedings of the European Conference on Planning (ECP), 1999, pp. 1–20.
[27] P. Bertoli, A. Cimatti, M. Roveri, P. Traverso, Planning in nondeterministic domains under partial observability via symbolic model checking, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), Seattle, USA, Morgan Kaufmann, 2001, pp. 473–478.
[28] P. Bertoli, A. Cimatti, M. Roveri, P. Traverso, Strong planning under partial observability, Artificial Intelligence 170 (2006) 337–384.
[29] B. Bonet, H. Geffner, GPT: a tool for planning with uncertainty and partial information, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 2001, pp. 82–87.
[30] J. Rintanen, Conditional planning in the discrete belief space, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 2005.

[31] U. Dal Lago, M. Pistore, P. Traverso, Planning with a language for extended goals, in: AAAI/IAAI Proceedings, Edmonton, Canada, AAAI Press/The MIT Press, 2002, pp. 447–454.
[32] R. Jensen, M.M. Veloso, OBDD-based universal planning for synchronized agents in non-deterministic domains, JAIR 13 (2000) 189–226.
[33] R. Jensen, M.M. Veloso, M.H. Bowling, OBDD-based optimistic and strong cyclic adversarial planning, in: Proceedings of the European Conference on Planning (ECP), 2001.
[34] R. Jensen, M.M. Veloso, R. Bryant, Guided symbolic universal planning, in: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), Trento, AAAI Press, 2003.
[35] C. Boutilier, T.L. Dean, S. Hanks, Decision-theoretic planning: Structural assumptions and computational leverage, JAIR 11 (1999) 1–94.
[36] A.L. Blum, M.L. Furst, Fast planning through planning graph analysis, Artificial Intelligence 90 (1–2) (1997) 281–300.
[37] C. Castellini, E. Giunchiglia, A. Tacchella, Sat-based planning in complex domains: Concurrency, constraints and nondeterminism, Artificial Intelligence 147 (1–2) (2003) 85–117.
[38] P. Ferraris, E. Giunchiglia, Planning as satisfiability in nondeterministic domains, in: AAAI/IAAI Proceedings, AAAI Press, 2000, pp. 748–753.
[39] E. Giunchiglia, Planning as satisfiability with expressive action languages: Concurrency, constraints and nondeterminism, in: Proceedings of the International Conference on Knowledge Representation and Reasoning (KR), 2000.
[40] D.E. Smith, D.S. Weld, Conformant Graphplan, in: AAAI/IAAI Proceedings, 1998, pp. 889–896.
[41] D.S. Weld, C.R. Anderson, D.E. Smith, Extending Graphplan to handle uncertainty and sensing actions, in: AAAI/IAAI Proceedings, Menlo Park, AAAI Press, 1998, pp. 897–904.
[42] D. Bryce, S. Kambhampati, Heuristic Guidance Measures for Conformant Planning, in: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), 2004.
[43] R. Brafman, J. Hoffmann, Conformant planning via heuristic forward search: A new approach, in: Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04), Whistler, Canada, Morgan Kaufmann, 2004.
[44] J. Hoffmann, R. Brafman, Contingent planning via heuristic forward search with implicit belief states, in: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), 2005.
[45] D. Bryce, S. Kambhampati, D.E. Smith, Planning graph heuristics for belief space search, Journal of Artificial Intelligence Research 26 (2006) 35–99.
[46] J. Kvarnström, P. Doherty, TALplanner: A temporal logic based forward chaining planner, Annals of Mathematics and Artificial Intelligence 30 (2001) 119–169.
[47] A. Gerevini, U. Kuter, D. Nau, A. Saetti, N. Waisbrot, Combining domain-independent planning and HTN planning: The duet planner, in: Proceedings of European Conference on Artificial Intelligence (ECAI), 2008.
[48] A. Gerevini, A. Saetti, I. Serina, Planning through stochastic local search and temporal action graphs, Journal of Artificial Intelligence Research 20 (2003) 239–290.
[49] K. Erol, J. Hendler, D.S. Nau, Complexity results for hierarchical task-network planning, Annals of Mathematics and Artificial Intelligence 18 (1996) 69–93.