ELSEVIER

Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Artificial Intelligence 147 (2003) 35–84

Artificial Intelligence

www.elsevier.com/locate/artint

Weak, strong, and strong cyclic planning via symbolic model checking

A. Cimatti *, M. Pistore, M. Roveri, P. Traverso

ITC-IRST, Via Sommarive 18, 38055 Povo, Trento, Italy

Received 22 June 2001; received in revised form 3 May 2002

Abstract

Planning in nondeterministic domains yields both conceptual and practical difficulties. From the conceptual point of view, different notions of planning problems can be devised: for instance, a plan might either guarantee goal achievement, or just have some chances of success. From the practical point of view, the problem is to devise algorithms that can effectively deal with large state spaces. In this paper, we tackle planning in nondeterministic domains by addressing conceptual and practical problems. We formally characterize different planning problems, where solutions have a chance of success ("weak planning"), are guaranteed to achieve the goal ("strong planning"), or achieve the goal with iterative trial-and-error strategies ("strong cyclic planning"). In strong cyclic planning, all the executions associated with the solution plan always have a possibility of terminating and, when they do, they are guaranteed to achieve the goal. We present planning algorithms for these problem classes, and prove that they are correct and complete. We implement the algorithms in the MBP planner by using symbolic model checking techniques. We show that our approach is practical with an extensive experimental evaluation: MBP compares positively with state-of-the-art planners, both in terms of expressiveness and in terms of performance.

Keywords: Planning in nondeterministic domains; Conditional planning; Symbolic model-checking; Binary decision diagrams

* Corresponding author.
E-mail addresses: cimatti@irst.itc.it (A. Cimatti), pistore@irst.itc.it (M. Pistore), roveri@irst.itc.it (M. Roveri), traverso@irst.itc.it (P. Traverso).

0004-3702/03/$ – see front matter
doi:10.1016/S0004-3702(02)00374-0

1. Introduction

In this paper we address the problem of planning for reachability goals in nondeterministic domains. *Nondeterministic domains* model a particular form of uncertainty: actions are modeled with different outcomes that cannot be predicted at planning time, i.e., it is impossible for the planner to know *a priori* which of the different possible outcomes will actually take place. *Reachability goals* intuitively express conditions on the final state of the execution of a plan: we want a plan that, when executed, reaches a state that satisfies some condition, i.e., the final state is a goal state. The notion of solution for a "reachability goal" in classical planning, where domains are deterministic, is clear, since the execution of a plan corresponds to a unique sequence of states: the final state must be a goal state. In the case of nondeterminism, the execution of a given plan may result, in general, in more than one sequence of states. Therefore, the solution to a "reachability goal" should be characterized with respect to the many possible executions of a plan. More precisely, we distinguish three possibilities:

(1) *Weak solutions* are plans that may achieve the goal, but are not guaranteed to do so: at least one of the many possible sequences of states corresponding to plan execution reaches the goal (i.e., the final state is a goal state).
(2) *Strong solutions* are plans that are guaranteed to achieve the goal in spite of nondeterminism: all the sequences of states corresponding to its execution reach the goal.
(3) *Strong cyclic solutions* are plans that are guaranteed to achieve the goal under a "fairness" assumption. Their execution can result in an infinite sequence of states, i.e., execution can loop forever. However, this happens only if some action is executed infinitely often in a given state and some of its outcomes (the ones leading to the goal) never occur. We say that these executions are "unfair".

Weak and strong solutions correspond to the two extreme requirements for satisfying reachability goals. Intuitively, weak solutions correspond to "optimistic plans", i.e., to plans that actually reach the goal just if the action outcomes result in one of the sequences that lead to the goal. Strong solutions correspond to "safe plans", i.e., to plans that reach the goal independently of all the uncertainty in possible action outcomes, e.g., failures. However, there might be cases in which weak solutions are not acceptable, and strong solutions do not exist. Strong cyclic solutions are a viable alternative. They correspond to the intuitive notion of "acceptable" iterative trial-and-error strategies. Consider, for instance, the following simple example in the "Block World" domain with one block on the table. The action "pick-up the block" is modeled with two possible outcomes: it may succeed, or it may fail by leaving the block on the table (i.e., in the same state). The reachability goal is to have the "block at hand". There is no strong solution to this planning problem. An example of weak solution is the plan composed of a single action, "pick up the block": if we are lucky, the action succeeds and the goal is reached. An example of strong cyclic solution is the plan "pick up the block until succeed", which repeats the execution of the action. The execution loops forever just in the case the action continuously fails.

However, this is a case of an unfair execution. This solution is much stronger than a weak solution: if at least one of the executions of the action succeeds, then we reach the goal.

Planning for weak, strong, and strong cyclic solutions yields both conceptual and practical difficulties. From the conceptual point of view, we need a formal characterization of the planning problems and well-founded algorithms that generate conditional and iterative plans. From the practical point of view, the problem is to devise algorithms that can effectively deal with large state spaces. Planning algorithms need efficient ways to analyze all possible action outcomes. Furthermore, in the case of strong cyclic planning, there is the additional difficulty that *infinite* execution paths must be analyzed. The results of the work presented in this paper provide both a well-founded and a practical framework that tackles problems that could not be dealt with before:

- We formally characterize the notions of weak, strong, and strong cyclic solution.
- We present planning algorithms for finding weak, strong, and strong cyclic solutions. The algorithms generate iterative and conditional plans that repeatedly sense the world, select an appropriate action, execute it, and iterate until the goal is reached. We prove that the algorithms are correct and complete, i.e., they find a solution if it exists and, if no solution exists, they terminate with failure.
- We show that the algorithms can be formulated by using symbolic model checking techniques [49]. In particular, sets of states are represented as propositional formulae, and search through the state space is performed as a set of logical transformations over propositional formulae. We implement the algorithms in the Model Based Planner (MBP), a planner built on top of the state-of-the-art symbolic model checker NuSMV [18]. MBP is based on Binary Decision Diagrams (BDDs) [15], that allow for a compact representation and effective manipulation of propositional formulae.
- We provide an extensive experimental evaluation of our approach. We show that MBP compares positively with the existing planners that can deal with nondeterministic domains, both in terms of expressiveness and in terms of performance. Most of the other planners cannot deal with all the planning problems the MBP algorithms have been designed for, e.g., strong cyclic planning.

The paper is structured as follows. In Section 2, we define nondeterministic planning domains, conditional and iterative plans, and the different notions of planning problems and solutions. In Section 3 we provide algorithms for strong and weak planning, which share a common structure and are similar in spirit: the resulting plans, if any, achieve the goal in a finite number of steps. In Section 4 we present the algorithm for strong cyclic planning, which generates iterative trial-and-error strategies. In Section 5 we show how the algorithms are implemented by means of symbolic model checking techniques. In Section 6 we describe MBP, our BDD-based planner. In Section 7 we evaluate our approach. We compare MBP with state-of-the-art planners that deal with nondeterministic domains. In Section 8 we present related work, and in Section 9 we draw some conclusions and present the lines of future research.

2. Domains, plans, and planning problems

The aim of this section is to provide a precise notion of planning problems in nondeterministic domains. We first define nondeterministic planning domains (Section 2.1), then the plan structure that we choose to represent conditional and iterative behaviours, and the notion of execution of a plan in a planning domain (Section 2.2). Finally, we define the notion of planning problem with different forms of solutions: weak, strong, and strong cyclic solutions (Section 2.3).

2.1. Nondeterministic domains

A (*nondeterministic*) *planning domain* can be expressed in terms of *propositions*, which may assume different values in different *states*, of *actions*, and of a *transition relation* describing how (the execution of) an action leads from one state to possibly many different states.

Definition 2.1 (*Planning domain*). A *planning domain* $\mathcal{D}$ is a 4-tuple $\langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ where

- $\mathcal{P}$ is the finite set of propositions,
- $\mathcal{S} \subseteq 2^{\mathcal{P}}$ is the set of states,
- $\mathcal{A}$ is the finite set of actions, and
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the transition relation.

Intuitively, a state is a collection of the propositions holding in it. The transition relation describes the effects of action execution. An action a is *executable* in a state s if there exists at least one state s' such that $\mathcal{R}(s, a, s')$. An action a is deterministic (nondeterministic) in a state s if there exists exactly one (more than one) state s' such that $\mathcal{R}(s, a, s')$. We denote with $\textsc{Act}(s)$ the set of actions that are executable in state s:

$$\textsc{Act}(s) \doteq \{a\colon\ \exists s'.\, \mathcal{R}(s, a, s')\}.$$

We denote with $\textsc{Exec}(s, a)$ the execution of a in s which corresponds to the set of the states that can be reached from s performing action $a \in \textsc{Act}(s)$:

$$\textsc{Exec}(s, a) \doteq \{s'\colon\ \mathcal{R}(s, a, s')\}.$$

Example 2.2. As an example, we use a simplified version of the nondeterministic "Omelette" domain [47], depicted in Fig. 1. The intended goal is to have two eggs into a bowl, so that an omelette can be prepared. Eggs can be unpredictably good or bad. A rotten egg in the bowl has the effect of spoiling the bowl. It is possible to determine whether the egg is good or bad only after the egg is broken into the bowl. However, breaking an egg may fail to open it and to drop its content into the bowl. In this case it is still impossible to determine whether an egg is good or bad. If we force the egg to open, we assure that its content is in the bowl. After this we can determine whether the egg is good or bad. A bowl can always be cleaned by discarding its content. We assume to have an infinite number of eggs that can be grabbed and broken into the bowl. The bowl can contain up to 2 eggs.

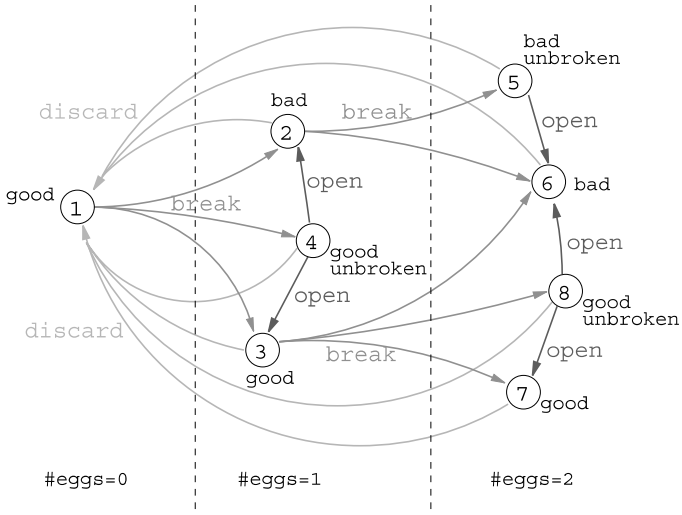

Fig. 1. The "Omelette" example.

In this domain, the propositions in $\mathcal{P}$ are {#eggs=0, #eggs=1, #eggs=2, bad, good, unbroken}. Proposition #eggs=i indicates the number i of eggs that have been broken into a bowl, including the egg that we might have failed to break. Proposition bad holds in a state where at least one of the eggs in the bowl is bad, while good is true in states where all eggs in the bowl are good. Proposition unbroken holds when we have failed to break an egg.

Possible states are sets of propositions. Each state may contain exactly one of the #eggs=i, and exactly one between good and bad. In the following, we associate each state with a numerical label. For instance, state 1 contains the propositions good and #eggs=0; state 5 contains the propositions #eggs=2, bad and unbroken; state 8 contains the propositions #eggs=2, good and unbroken.

The set of actions $\mathcal{A}$ is {break, open, discard}. The transition relation $\mathcal{R}$ represents the precondition of each action, and the effects of an action in a given state. For instance, we have that $\mathcal{R}(1, \text{break}, 2)$, $\mathcal{R}(1, \text{break}, 3)$, and $\mathcal{R}(1, \text{break}, 4)$. This means that the action break can be executed in the state where #eggs=0 holds, leads to a state where #eggs=1 holds, and where nondeterministically either the egg is unbroken and good holds, or unbroken does not hold and the bowl can be unpredictably good or bad.

Planning domains, as given in Definition 2.1, are independent of the language we use to describe them. In fact, all the planning algorithms described in this paper are independent of the domain language, since they work at the level of the (semantic) model of domains described in Definition 2.1. Existing languages for describing nondeterministic domains, and the problem of mapping the description of a domain given in one of these languages into the corresponding semantic model, are discussed in Section 6.

2.2. Plans as state-action tables

We need to express conditional and iterative plans that, when executed, sense the world at run-time and, depending on the state of the world, can execute different actions. These plans can be described by associating a set of actions to a state that can be executed in such state. We call them *state-action tables*. They resemble universal plans [60] and policies [4,12].

Definition 2.3 (*State-action table*). A *state-action table* π for a planning domain $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ is a set of pairs $\langle s, a \rangle$, where $s \in \mathcal{S}$ and $a \in \textsc{Act}(s)$. A state-action table π is deterministic if for any state s there is at most one action a such that $\langle s, a \rangle \in \pi$, otherwise it is nondeterministic.

We call $\langle s, a \rangle$ a *state-action pair*. According to Definition 2.3, action a of state-action pair $\langle s, a \rangle$ must be executable in s. Hereafter, we write $\textsc{StatesOf}(\pi)$ for the set of states of the state-action table π:

$$\textsc{StatesOf}(\pi) \doteq \{s\colon \langle s, a \rangle \in \pi\}.$$

The execution of a state-action table can result in conditional and iterative behaviours. Intuitively, a state-action table execution can be explained in terms of a reactive execution loop that senses the state of the world and chooses one of the corresponding actions, if any, for the execution.

$s :=$ SenseCurrentState();
while $s \in \textsc{StatesOf}(\pi)$ **do**
    $a :=$ GetAction(s, π);
    Execute(a);
    $s :=$ SenseCurrentState();
done

SenseCurrentState returns a description of the status of the world. For a reactive system, this amounts to reading the sensors. GetAction selects one action among the possible actions associated with the current state. Then, the selected action is executed by Execute. The loop is repeated until a terminal state is reached, i.e., a state that is not associated with any action in the state-action table.

Example 2.4. Possible state-action tables for the omelette planning domain are the following. (We will use these state-action tables in the following examples.)

$$\pi_a \doteq \{\langle 1, \texttt{break} \rangle, \langle 3, \texttt{break} \rangle\},$$
$$\pi_b \doteq \{\langle 1, \texttt{break} \rangle, \langle 2, \texttt{break} \rangle, \langle 3, \texttt{break} \rangle, \langle 4, \texttt{open} \rangle, \langle 5, \texttt{open} \rangle, \langle 8, \texttt{open} \rangle\},$$
$$\pi_c \doteq \{\langle 1, \texttt{break} \rangle, \langle 2, \texttt{discard} \rangle, \langle 3, \texttt{break} \rangle, \langle 4, \texttt{open} \rangle, \langle 6, \texttt{discard} \rangle, \langle 8, \texttt{open} \rangle\}.$$

The execution of a state-action table in a planning domain can be described in terms of the transitions that the state-action table induces.

Definition 2.5 (*Execution structure*). Let π be a state-action table of a planning domain $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$. The *execution structure* induced by π from the set of initial states $\mathcal{I} \subseteq \mathcal{S}$ is a tuple $K = \langle Q, T \rangle$, where $Q \subseteq \mathcal{S}$ and $T \subseteq \mathcal{S} \times \mathcal{S}$ are the minimal sets satisfying the following rules:

(1) if $s \in \mathcal{I}$, then $s \in Q$, and
(2) if $s \in Q$ and there exists a state-action pair $\langle s, a \rangle \in \pi$ such that $\mathcal{R}(s, a, s')$, then $s' \in Q$ and $T(s, s')$.

A state $s \in Q$ is a *terminal state* of K if there is no $s' \in Q$ such that $T(s, s')$.

An execution structure is a directed graph, where the nodes are all the states which can be reached by executing actions in the state-action table, and the arcs represent possible action executions. Intuitively, an induced execution structure contains all the states (transitions) that can be reached (fired) when executing the state-action table π from the initial set of states $\mathcal{I}$. An induced execution structure is not required to be total, i.e., it may contain states with no outcoming arcs. Intuitively, terminal states represent states where the execution stops.

Example 2.6. The execution structures induced by the state-action tables π_a, π_b and π_c in Example 2.4 on the omelette domain from state `1` can be depicted as the graphs in Figs. 2, 3, and 4 respectively.

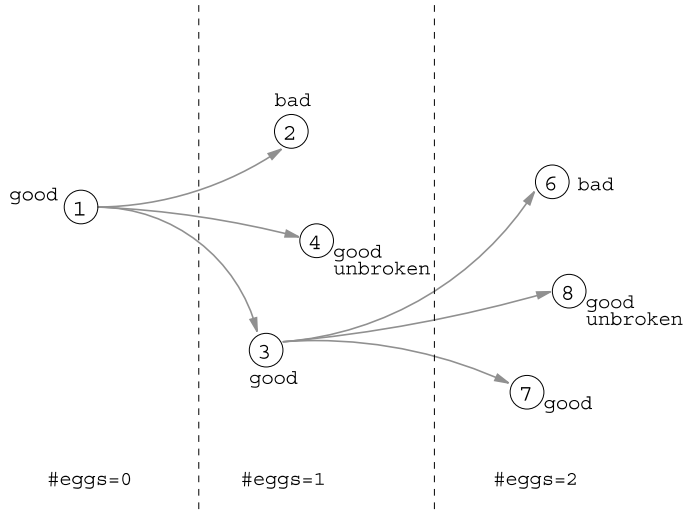

Fig. 2. The execution structure induced by state-action table π_a.

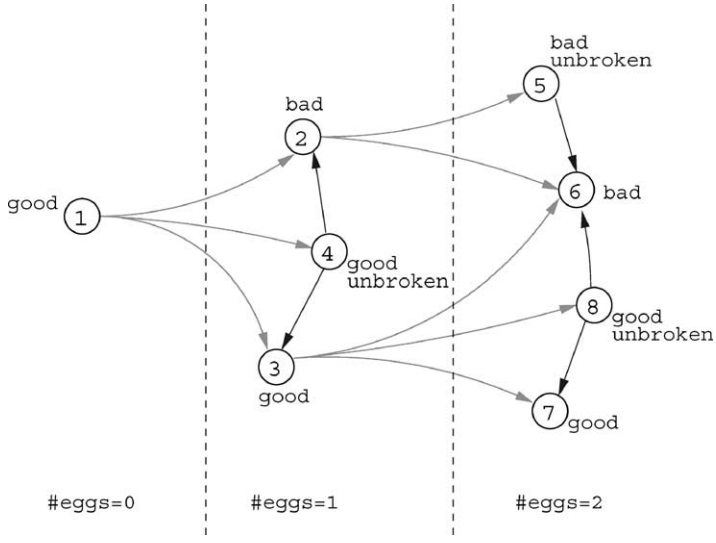

Fig. 3. The execution structure induced by state-action table π_b.

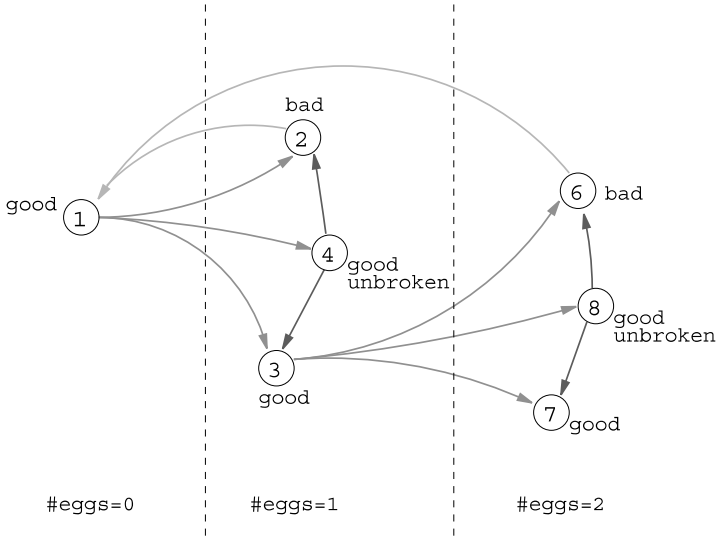

Fig. 4. The execution structure induced by state-action table π_c.

An execution structure is a finite presentation of all the possible executions of a given plan in a given planning domain. An *execution path* of an execution structure is a sequence of states in the execution structure, and can be either a finite path ending in a terminal state, or an infinite path.

Definition 2.7 (*Execution path*). Let $K = \langle Q, T \rangle$ be the execution structure induced by a state-action table π from $\mathcal{I}$. An *execution path* of K from $s_0 \in \mathcal{I}$ is a possibly infinite sequence $s_0, s_1, s_2, \ldots$ of states in Q such that, for all states s_i in the sequence:

- either s_i is the last state of the sequence, in which case s_i is a terminal state of K;
- or $T(s_i, s_{i+1})$.

We say that a state s' is *reachable from* a state s if there is a path from s to s'. K is an *acyclic execution structure* if all its execution paths are finite.

Example 2.8. Some of the infinitely many execution paths for the execution structure induced by the state-action table π_c in Example 2.4 from state `1` are the following.

`1, 3, 7`

`1, 4, 3, 8, 7`

`1, 2, 1, 3, 7`

`1, 2, 1, 2, 1, 3, 7`

`1, 2, 1, 2, ...`

`1, 3, 6, 1, 3, 7`

`1, 3, 6, 1, 3, 6 ...`

All these sequences of states are paths of the execution structure represented in Fig. 4. All the finite paths end in state `7`, that is the only terminal state of the execution structure.

2.3. Planning problems and solutions

A *planning problem* is defined by a planning domain $\mathcal{D}$, a set of initial states $\mathcal{I}$, and a set of goal states $\mathcal{G}$.

Definition 2.9 (*Planning problem*). Let $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ be a planning domain. A *planning problem* for $\mathcal{D}$ is a triple $\langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{I} \subseteq \mathcal{S}$ and $\mathcal{G} \subseteq \mathcal{S}$.

The above definition takes into account two forms of nondeterminism. First, we have a set of initial states, and not a single initial state. This allows for expressing partially specified initial conditions. Second, the execution of an action from a state results in a set of states, and not necessarily in a single state (see $\mathcal{R}$ in Definition 2.1). This allows for expressing nondeterministic action executions.

Intuitively, solutions to a planning problem satisfy a reachability requirement: a solution is a state-action table whose aim is, starting at any state in the set of initial states $\mathcal{I}$, to reach states in a set of final desired states $\mathcal{G}$. In order to make this requirement precise, we need to specify "how" the set of final desired states should be reached, i.e., the "strength" of this requirement.

We formalize the notion of weak, strong and strong cyclic solutions as follows.

Definition 2.10 (*Deterministic solutions*). Let $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ be a planning domain. Let $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a planning problem. Let π be a deterministic state-action table for $\mathcal{D}$. Let $K = \langle Q, T \rangle$ be the execution structure induced by π from $\mathcal{I}$.

(1) π is a *weak solution* to P if for any state in $\mathcal{I}$ some terminal state of K is reachable that is also in $\mathcal{G}$.
(2) π is a *strong solution* to P if K is acyclic and all terminal states of K are in $\mathcal{G}$.
(3) π is a *strong cyclic solution* to P if from any state in Q some terminal state of K is reachable and all the terminal states of K are in $\mathcal{G}$.

According to the definitions given in Section 2.1, the execution of a state-action table terminates only when there is no action associated to the terminal state in the state-action table. Therefore, in Definition 2.10 we do not consider to be successful those execution paths that contain goal states but that terminate in a state that is not in $\mathcal{G}$.

Weak solutions are plans that may achieve the goal, but are not guaranteed to do so. This amounts to saying that at least one of the many possible execution paths of the state-action table should result in a terminal state that is a goal state. *Strong solutions* are plans that are guaranteed to achieve the goal in spite of nondeterminism, i.e., all the execution paths should result in a terminal state that is a goal state. *Strong cyclic solutions* formalize the intuitive notion of "acceptable" iterative trial-and-error strategies: all their partial execution paths can be extended to a finite execution path whose terminal state is a goal state. Strong cyclic solutions can produce executions that loop forever. However, this can only happen under an infinite sequence of "failures" in the execution of some action. That is, all the infinite execution paths are "unfair", since they eventually enter loops where some actions are executed infinitely often in given states, but some of its outcomes (the ones leading to the goal) never occur.

Weak, strong, and strong-cyclic solutions can also be characterized in terms of probabilities. Assume that each outcome $s' \in \text{EXEC}(s, a)$ of action a from state s has a non-zero probability. Then *weak* plans reach the goal with probability strictly greater than 0. *Strong* plans reach the goal with probability 1 in at most n steps.[1] Also *strong cyclic* plans reach the goal with probability 1, but there is no bound on the number of steps; in this case, infinite execution paths are allowed, but they have probability 0 to occur.

In the general case of *nondeterministic* state-action tables, we say that π is a strong (weak, strong cyclic) solution to a planning problem if all the determinizations of π are, according to Definition 2.10. Formally, a determinization of π is any deterministic state-action table $\pi_d \subseteq \pi$ such that $\text{STATESOF}(\pi_d) = \text{STATESOF}(\pi)$. In this way, we model the fact that the executor can choose any action arbitrarily when building the deterministic state-action table, i.e., there are no compatibility constraints among the actions in different states. This guarantees that the plan is correct also in the general case where the executor selects at run-time one action among the possible actions associated to the current state by the state-action table.

[1] The number n of steps required to reach the goal depends on the actual plan. However, n cannot exceed the number of states in the domain, since the execution paths of strong plans cannot pass twice through the same state.

Definition 2.11 (*Nondeterministic solutions*). Let $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ be a planning domain. Let $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a planning problem. Let π be a (nondeterministic) state-action table for $\mathcal{D}$. π is a weak (resp. strong, strong cyclic) solution to P if all the determinizations π_d of π are weak (resp. strong, strong cyclic) solutions to P according to Definition 2.10.

As a final remark, notice that the strong solutions to a planning problem are a subset of the strong cyclic solutions, which are in turn a subset of the weak solutions. Indeed, any state-action table π that is a strong solution to a planning problem P is also a strong cyclic solution to P: if the execution structure K induced by π is acyclic, then all the execution paths are finite. Hence, from any state in K, a terminal state is reached following any path. Moreover, any state-action table π that is a strong cyclic solution to a planning problem P is also a weak solution to P. Indeed, according to the definition of strong cyclic solution, a terminal goal state is reachable from *any* state of the execution structure, so this is true in particular for all the initial states.

Example 2.12. In Example 2.4, state-action table π_a is a weak solution to the planning problem where the initial state is state `1` and the goal state is `7`. It is a plan that may achieve the goal of having two good eggs in the bowl.

State-action table π_b is a strong solution to the planning problem where the initial state is state `1` and the goal states are `6` and `7`. It is a safe plan that guarantees that two eggs (either bad or good) are broken in the bowl.

State-action table π_b is also a weak solution to the planning problem where the initial state is state `1` and the goal state is `7`, even if it is not a strong solution for this planning problem.

State-action table π_c is a strong cyclic solution to the planning problem where the initial state is `1` and the goal state is `7`. It is an iterative trial-and-error strategy that, every time a bad egg is broken into the bowl, cleans the bowl by discarding the egg(s). It iteratively tries to get to the point where two good eggs are in the bowl, so that an omelette can be prepared.

The latter planning problem admits no strong solutions: in principle, it is always possible to encounter an infinite sequence of rotten eggs which will prevent us from making our omelette. This is, however, a case of extreme bad luck. A strong cyclic solution guarantees that, for any case of "finite" bad luck, the goal is reached with a finite execution.

3. Algorithms for weak and strong planning

In this section we describe the algorithms for weak and strong planning. The two algorithms operate on the planning problem: the sets of the initial states $\mathcal{I}$ and of the goal states $\mathcal{G}$ are explicitly given as input parameters, while the domain $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ is assumed to be globally available to the invoked subroutines. Both algorithms either return a solution state-action table, or a distinguished value for state-action tables, called *Fail*, used to represent search failure. In particular, we assume that *Fail* is different from the empty state-action table, that we will denote with $\emptyset$.

```
1. function WeakPlan(I, G);
2.    OldSA := Fail;
3.    SA := ∅;
4.    while (OldSA ≠ SA ∧ I ⊈ (G ∪ StatesOf(SA))) do
5.       PreImage := WeakPreImage(G ∪ StatesOf(SA));
6.       NewSA := PruneStates(PreImage, G ∪ StatesOf(SA));
7.       OldSA := SA;
8.       SA := SA ∪ NewSA;
9.    done ;
10.   if (I ⊆ (G ∪ StatesOf(SA))) then
11.      return SA;
12.   else
13.      return Fail;
14.   fi ;
15. end ;

1. function StrongPlan(I, G);
2.    OldSA := Fail;
3.    SA := ∅;
4.    while (OldSA ≠ SA ∧ I ⊈ (G ∪ StatesOf(SA))) do
5.       PreImage := StrongPreImage(G ∪ StatesOf(SA));
6.       NewSA := PruneStates(PreImage, G ∪ StatesOf(SA));
7.       OldSA := SA;
8.       SA := SA ∪ NewSA;
9.    done ;
10.   if (I ⊆ (G ∪ StatesOf(SA))) then
11.      return SA;
12.   else
13.      return Fail;
14.   fi ;
15. end ;
```

Fig. 5. The algorithms for weak and strong planning.

The algorithms, presented in Fig. 5, are based on a breadth-first search proceeding backwards from the goal, towards the initial states. For both algorithms, at each iteration step, the set of states for which a solution has been already found is used as a target for the expansion preimage routine at line 5, that returns a new "slice" to be added to the state-action table under construction. The algorithms are actually identical, except for the fact that the extension primitive is the function WeakPreImage in the case of weak planning, and StrongPreImage in the case of strong planning. Functions WeakPreImage and StrongPreImage are defined as follows:

$$\textsc{WeakPreImage}(S) \doteq \big\{\langle s, a\rangle\colon \textsc{Exec}(s, a) \cap S \neq \emptyset\big\},$$
$$\textsc{StrongPreImage}(S) \doteq \big\{\langle s, a\rangle\colon \emptyset \neq \textsc{Exec}(s, a) \subseteq S\big\}.$$

Intuitively, WeakPreImage(S) returns the set of state-action pairs $\langle s, a\rangle$ such that the execution of a in s may lead inside S. StrongPreImage(S) returns the set of state-

action pairs $\langle s, a\rangle$ such that the execution of a in s is guaranteed to lead to states inside S, regardless of nondeterminism.

In the weak (strong) planning algorithm, function WEAKPREIMAGE (function STRONGPREIMAGE, resp.) is called using as target the goal states G and the states that are already in the state-action table SA: these are the states for which a solution is already known. In both cases, the returned preimage *PreImage* is then passed to function PRUNESTATES, defined as follows:

$$\text{PRUNESTATES}(\pi, S) \doteq \{\langle s, a\rangle \in \pi\colon\ s \notin S\}.$$

This function removes from the preimage table all the pairs $\langle s, a\rangle$ such that a solution is already known for s. This pruning is important to guarantee that only the shortest solution from any state appears in the state-action table. The termination test requires that the initial states are included in the set of accumulated states (i.e., $G \cup \text{STATESOF}(SA)$), or that a fix-point has been reached and no more states can be added to state-action table SA. In the first case, the returned state-action table is a solution to the planning problem. In the second case, no solution exists.

Example 3.1. Let us consider the omelette domain and the planning problem of reaching state 7 from state 1.

The weak planning algorithm succeeds to build a weak solution for the planning problem. Indeed, the value of variable SA in the algorithm after i iterations of the while loop coincides with state-action table π_i, where:

$$\pi_0 = \emptyset,$$
$$\pi_1 = \{\langle 3, \texttt{break}\rangle, \langle 8, \texttt{open}\rangle\},$$
$$\pi_2 = \{\langle 3, \texttt{break}\rangle, \langle 8, \texttt{open}\rangle, \langle 1, \texttt{break}\rangle, \langle 4, \texttt{open}\rangle\}.$$

After the second iteration of the while loop, the algorithm terminates, as the state-action table contains the initial state 1.

The strong planning algorithm, instead, fails to build a strong solution for the planning problem. The algorithm starts with variable SA set to $\emptyset$ and, since $\text{STRONGPREIMAGE}(\{7\}) = \emptyset$, no new state-action pair is added to variable SA in the first iteration of the while loop: the fix-point is immediately reached. The computed state-action table $\emptyset$ does not include the initial state 1, therefore the strong planning algorithm returns with a failure. This result is correct, as there is no strong solution for this planning problem. In general, it is possible to show that, if the fix-point is reached, the computed state-action table includes all states that admit a strong solution: in this case, the only state that admits a (trivial) strong solution is the goal state 7.

Now we show that the weak and strong planning algorithms proposed above are correct, namely that they always terminate and that they return a solution if (and only if) such a solution exists. Also, we show that the state-action table returned by the algorithms are "optimal" w.r.t. a suitable measure of the distance of a state from the goal.

3.1. Formal properties of the weak planning algorithm

In this section we prove the correctness of the weak planning algorithm.

We have seen that the algorithm incrementally builds the state-action table backwards, proceeding from the goal towards the initial states. In this way, at any iteration of the while loop in the algorithm, states at a growing distance from the goal are added to the state-action table. To formalize this intuition, we define the *weak distance* of a state s from goal $\mathcal{G}$ as that smallest number of transitions that are necessary in the domain to reach from s a state in $\mathcal{G}$.

We start by giving the definition of *trace*: a trace in the planning domain is a sequence of states that are connected by the transition relation.

Definition 3.2 (*Trace*). A *trace* from state s to state s' is a sequence of states $s_0, s_1, \ldots, s_i$ with $s_0 = s$, $s_i = s'$ such that for all $j = 0, \ldots, i-1$ there is some action a_j such that $\mathcal{R}(s_j, a_j, s_{j+1})$. Let S' be a set of states; then a trace from s to S' is a trace from s to any state $s' \in S'$. We say that trace $s = s_0, s_1, \ldots, s_i = s'$ has length i. We say that trace $s = s_0, s_1, \ldots, s_i = s'$ is compatible with state-action table π if for all $j = 0, \ldots, i-1$ there is some $\langle s_j, a_j \rangle \in \pi$ such that $\mathcal{R}(s_j, a_j, s_{j+1})$.

Definition 3.3 (*Weak distance*). The *weak distance* $\mathcal{WD}ist(s, \mathcal{G})$ of a state s from goal $\mathcal{G}$ is the smallest integer i such that there is a trace of length i from s to $\mathcal{G}$. If no trace from s to $\mathcal{G}$ exists, then we define $\mathcal{WD}ist(s, \mathcal{G}) = \infty$.

We remark that, according to this definition, if $s \in \mathcal{G}$, then $\mathcal{WD}ist(s, \mathcal{G}) = 0$, as there is a 0-length trace from s to $\mathcal{G}$.

We now formalize the idea that states at a growing distance from goal $\mathcal{G}$ are added to state-action table *SA* at each iteration of the algorithm (Proposition 3.6 below). We start with some notations and auxiliary lemmas.

Let π_i be the value of variable *SA* after i iterations of the while loop at lines 4–9 of the WEAKPLAN algorithm; also, let $S_0 = \mathcal{G}$ and $S_i = \text{STATESOF}(\pi_i \setminus \pi_{i-1})$ be the set of states added to *SA* during the ith iteration.

Lemma 3.4. $\text{STATESOF}(\pi_i) = \bigcup_{j=1..i} S_j$ *and* $\mathcal{G} \cup \text{STATESOF}(\pi_i) = \bigcup_{j=0..i} S_j$.

Proof. This is a direct consequence of the definitions of π_i and S_i. □

Lemma 3.5. *The sets of states S_i are disjoint, i.e., $S_i \cap S_j = \emptyset$ if $i \neq j$.*

Proof. The call of function PRUNESTATES at line 6 of the algorithm guarantees this property. □

Proposition 3.6. *Let s be a state in the planing domain. Then $s \in S_i$ iff $\mathcal{WD}ist(s, \mathcal{G}) = i$.*

Proof. By induction on i.

Case $i = 0$. By definition of S_0, $s \in S_0$ iff $s \in \mathcal{G}$. Also, by definition of weak distance, $\mathcal{WD}ist(s, \mathcal{G}) = 0$ iff $s \in \mathcal{G}$. So $s \in S_0$ iff $\mathcal{WD}ist(s, \mathcal{G}) = 0$.

Case $i > 0$. By the inductive hypothesis, for all $j < i$ we have $s' \in S_j$ iff $\mathcal{WD}ist(s', \mathcal{G}) = j$. So, by Lemma 3.4, $s' \in \mathcal{G} \cup \textsc{StatesOf}(\pi_{i-1})$ iff $s \in \bigcup_{j=0..i-1} S_j$ iff $\mathcal{WD}ist(s, \mathcal{G}) < i$.

Assume $s \in S_i$. Then, by definition of S_i and of π_i, it holds that

$$s \in \textsc{StatesOf}\big(\textsc{WeakPreImage}\big(\mathcal{G} \cup \textsc{StatesOf}(\pi_{i-1})\big)\big)$$

and

$$s \notin \mathcal{G} \cup \textsc{StatesOf}(\pi_{i-1}).$$

By definition of WEAKPREIMAGE, there is some action a and some $s' \in \textsc{Exec}(s, a)$ such that $s' \in \mathcal{G} \cup \textsc{StatesOf}(\pi_{i-1})$. Therefore, $\mathcal{WD}ist(s, \mathcal{G}) \leqslant \mathcal{WD}ist(s', \mathcal{G}) + 1 \leqslant i$, as $\mathcal{WD}ist(s', \mathcal{G}) \leqslant i - 1$ for all $s' \in \mathcal{G} \cup \textsc{StatesOf}(\pi_{i-1})$. So, we have proved that $\mathcal{WD}ist(s, \mathcal{G}) \leqslant i$. To conclude $\mathcal{WD}ist(s, \mathcal{G}) = i$ we observe that $\mathcal{WD}ist(s, \mathcal{G}) \leqslant i - 1$ is impossible, as $s \notin \mathcal{G} \cup \textsc{StatesOf}(\pi_{i-1})$.

Assume now $\mathcal{WD}ist(s, \mathcal{G}) = i$. Then, by definition of weak distance, there is some a and some $s' \in \textsc{Exec}(s, a)$ such that $\mathcal{WD}ist(s', \mathcal{G}) = i - 1$. Then $\langle s, a\rangle \in \textsc{WeakPreImage}(S_{i-1})$ and, since $S_{i-1} \subseteq \mathcal{G} \cup \textsc{StatesOf}(\pi_{i-1})$ by Lemma 3.4, we obtain

$$\langle s, a\rangle \in \textsc{WeakPreImage}\big(\mathcal{G} \cup \textsc{StatesOf}(\pi_{i-1})\big).$$

Moreover, $s \notin \mathcal{G} \cup \textsc{StatesOf}(\pi_{i-1})$: otherwise $s \in S_j$ for some $j < i$ and hence $\mathcal{WD}ist(s, \mathcal{G}) < i$ by the inductive hypothesis, and this contradicts the hypothesis the $\mathcal{WD}ist(s, \mathcal{G}) = i$. So,

$$\langle s, a\rangle \in \pi_i = \textsc{PruneStates}\big(\textsc{WeakPreImage}\big(\mathcal{G} \cup \textsc{StatesOf}(\pi_{i-1})\big), \mathcal{G} \cup \textsc{StatesOf}(\pi_{i-1})\big).$$

We conclude that $s \in S_i$ by definition of S_i. □

The following proposition captures the main property of the state-action table built by the weak planning algorithm. Namely, assume that state s is at distance i from the goal and that action a is a valid action to be performed in state s according to the state-action table built by the algorithm. Then, some of the outcomes of executing a in s should succeed and lead to a state at distance $i-1$ from the goal, while there may be other executions that fail and lead to a state at a bigger distance from the goal, or to a state from which the goal is unreachable. According to the definition of weak distance, no outcome can lead to a state at a distance less than $i - 1$ from the goal.

Proposition 3.7. *Let* $i > 0$. *If* $s \in S_i$ *and* $\langle s, a\rangle \in \pi_i$, *then*:

- $\textsc{Exec}(s, a) \cap S_j = \emptyset$ *if* $j < i - 1$, *and*
- $\textsc{Exec}(s, a) \cap S_{i-1} \neq \emptyset$.

Proof. By definition of S_i and of π_i, we know that

$$\langle s, a\rangle \in \textsc{WeakPreImage}\big(\mathcal{G} \cup \textsc{StatesOf}(\pi_{i-1})\big)$$

and that

$$s \notin \mathcal{G} \cup \textsc{StatesOf}(\pi_{i-1}).$$

For the first item, assume $\textsc{Exec}(s,a) \cap S_j \neq \emptyset$ for some $j < i$. Then $\langle s, a\rangle \in \textsc{WeakPreImage}(S_j)$ by definition of weak preimage, and since $S_j \subseteq \mathcal{G} \cup \textsc{StatesOf}(\pi_j)$ then $\langle s, a\rangle \in \textsc{WeakPreImage}(\mathcal{G} \cup \textsc{StatesOf}(\pi_j))$. Moreover, $s \notin \mathcal{G} \cup \textsc{StatesOf}(\pi_{i-1})$ and $j < i$ imply $s \notin \mathcal{G} \cup \textsc{StatesOf}(\pi_j)$. Summing up, we have proved that

$$\langle s, a\rangle \in \textsc{PruneStates}\big(\textsc{WeakPreImage}\big(\mathcal{G} \cup \textsc{StatesOf}(\pi_j)\big), \mathcal{G} \cup \textsc{StatesOf}(\pi_j)\big),$$

that is $\langle s, a\rangle \in \pi_{j+1}$. This implies $s \in S_{j+1}$ and, since $s \in S_i$ by hypothesis, Lemma 3.5 forces $i = j+1$. This concludes the proof of the first item.

For the second item, assume by contradiction $\textsc{Exec}(s,a) \cap S_{i-1} = \emptyset$. We have just proved that $\textsc{Exec}(s,a) \cap S_j = \emptyset$ for $j < i-1$, so, by Lemma 3.4 we conclude $\textsc{Exec}(s,a) \cap (\mathcal{G} \cup \textsc{StatesOf}(\pi_{i-1})) = \emptyset$. Therefore, $\langle s, a\rangle \notin \textsc{WeakPreImage}(\mathcal{G} \cup \textsc{StatesOf}(\pi_{i-1}))$ and hence $\langle s, a\rangle \notin \pi_i$, which is absurd. By contradiction, we conclude that $\textsc{Exec}(s,a) \cap S_{i-1} \neq \emptyset$. □

Propositions 3.6 and 3.7 are the basic ingredients for proving the correctness of function WeakPlan. The correctness depends on the fact that all the states from which the goal may be reached have a finite weak distance from the goal, and hence are eventually added to the state-action table built by the algorithm (Proposition 3.6); and on the fact that any action that appears in the state-action table has some outcome that lead to a shorter distance from the goal (Proposition 3.7), so that a path to the goal can be obtained by following the outcomes of the actions.

Theorem 3.8 (Correctness). *Let* $\pi \doteq \textsc{WeakPlan}(\mathcal{I}, \mathcal{G}) \neq \mathit{Fail}$. *Then* π *is a weak solution of the planning problem* $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G}\rangle$.

If $\textsc{WeakPlan}(\mathcal{I}, \mathcal{G}) = \mathit{Fail}$, *instead, then there is no weak solution for planning problem* P.

Proof. Assume $\pi \neq \mathit{Fail}$ and let π_d be any deterministic plan in π. We show that π_d is a weak solution for planning problem P.

Since $\pi \neq \mathit{Fail}$, there must be some $i \geqslant 0$ such that $\pi = \pi_i$ and $\mathcal{I} \subseteq \mathcal{G} \cup \textsc{StatesOf}(\pi_i)$. Let $s \in \mathcal{I}$: we have to show that there exists some path in the execution structure from s to $\mathcal{G}$. We know, by Lemma 3.4, that $s \in \mathcal{G} \cup \textsc{StatesOf}(\pi_i)$ implies $s \in S_j$ for some $j \leqslant i$. If $s \in S_0 = \mathcal{G}$ then there is the trivial path. Otherwise, $s \in \textsc{StatesOf}(\pi_i)$ and, by definition of determinization, there is some $\langle s, a\rangle \in \pi_d$ such that $\langle s, a\rangle \in \pi_i$. Since $s \in S_j$, we also have $\langle s, a\rangle \in \pi_j$. By Proposition 3.7 we know that there is some $s' \in \textsc{Exec}(s,a)$ such that $s' \in S_{j-1}$; hence, (s, s') is a transition of the execution structure. By induction on j, it is easy to conclude that there is an execution path of length j from s to $\mathcal{G}$.

Assume now $\pi = \mathit{Fail}$. Then there is some i such that $\pi_i = \pi_j$ for any $j > i$, and there exists some $s \in \mathcal{I}$ such that $s \notin \textsc{StatesOf}(\pi_i) \cup \mathcal{G}$. Now, assume by contradiction that there is a deterministic state-action table π_d and a finite execution path from s to $\mathcal{G}$ in the

execution structure for π_d. Then, clearly, there is some trace in the domain that corresponds to that execution path, and hence $\mathcal{WD}ist(s, \mathcal{G})$ is finite. Let $k = \mathcal{WD}ist(s, \mathcal{G})$: then $s \in S_k$. If $k \leqslant i$ then we would have $s \in \mathcal{G} \cup \text{STATESOF}(\pi_i)$ by Lemma 3.4, and this contradicts the hypotheses. If $k > i$, then $s \in S_k$; this is absurd since $\pi_k = \pi_i = \pi_{k-1}$ and hence $S_k = \emptyset$. So, we conclude that no weak solution π_d can exist if $\pi = \mathit{Fail}$. □

Algorithm WEAKPLAN always terminates: indeed, at any iteration of the while loop, either the cardinality of set *SA* strictly grows, or the loop ends due to condition "*OldSA* $\neq$ *SA*" in the guard of the while loop. Also, the cardinality of *SA* cannot grow unboundedly, as there are only finitely many valid state-action pairs.

Theorem 3.9 (Termination). *Function* WEAKPLAN *always terminates.*

Proof. The only possible cause of non-termination is the while loop at lines 4–9. We can see (line 8) that during an iteration of the loop the cardinality of set *SA* cannot reduce. Moreover, if the value of variable *SA* does not change during an iteration, then the loop terminates (condition *OldSA* $\neq$ *SA* in the guard of the loop). Since set *SA* $\subseteq \mathcal{S} \times \mathcal{A}$ and $\mathcal{S} \times \mathcal{A}$ is finite, it is not possible for the cardinality of set *SA* to strictly grow indefinitely, so the loop will eventually terminate. □

It is easy to observe that also STATESOF(*SA*) grows strictly at all iterations of the loop except the last one: due to the usage of PRUNESTATES, if $\text{STATESOF}(\pi_{i+1}) = \text{STATESOF}(\pi_i)$, then $\pi_{i+1} = \pi_i$. So, the loop is executed at most $|\mathcal{S}| + 1$ times.

The backward construction performed by the algorithm guarantees the optimality of the computed solution with respect to the weak distance of the initial states from the goal.

Theorem 3.10 (Optimality). *Let* $\pi \doteq \text{WEAKPLAN}(\mathcal{I}, \mathcal{G}) \neq \mathit{Fail}$. *Then plan* π *is optimal with respect to the weak distance: namely, for each* $s \in \mathcal{I}$, *in the execution structure for* π *there exists an execution path from* s *to* $\mathcal{G}$ *of length* $\mathcal{WD}ist(s, \mathcal{G})$.

Proof. In the proof of the correctness theorem we have already proved that if $s \in \mathcal{I}$ and $s \in S_j$ then there is a path of length j in the execution structure for π. By Proposition 3.6, $j = \mathcal{WD}ist(s, \mathcal{G})$. □

We remark that, by definition of weak distance, there can be no execution path shorter than $\mathcal{WD}ist(s, \mathcal{G})$ in the execution structure corresponding to a weak solution. This guarantees the optimality of plan π.

3.2. Formal properties of the strong planning algorithm

The formal results for the strong planning algorithm can be easily obtained by adapting those for the weak planning algorithm presented in Section 3.1. Here we only discuss the most relevant differences.

The strong planning algorithm differs from the weak one only for the preimages computed in the while loop: the former computes strong preimages, while the latter

computes weak preimages. The consequence is that, during the iterations of the strong planning algorithm, states are added to state-action tables *SA* according to a different distance from the goal states. The *strong distance* of a state from a goal takes into account that a strong solution must guarantee to reach the goal in spite of the possible nondeterministic outcomes of the executed actions. To define the strong distance, we first introduce the notion of a complete set T of traces from state s to $\mathcal{G}$. It is a set of traces that covers all the possible nondeterministic outcomes of actions: if a particular outcome is considered in any trace of T, then all the other nondeterministic outcomes must be considered in some other traces of T.

Definition 3.11 (*Complete set of traces*). A *complete set of traces* from s to $\mathcal{G}$ is a set T of traces from s to states in $\mathcal{G}$ such that, whenever $s = s_0, s_1, \ldots, s_i$ is in the set T and $j = 0, \ldots, i-1$, then there is some action a such that $\mathcal{R}(s_j, a, s_{j+1})$ and, whenever $\mathcal{R}(s_j, a, s'_{j+1})$, then there is some trace in T that extends $s = s_0, s_1, \ldots, s_j, s'_{j+1}$.

The strong distance of a state s from $\mathcal{G}$ corresponds to the length of the shortest complete set of traces from s to $\mathcal{G}$, where the length of a complete set of traces is the length of its longest trace.

Definition 3.12 (*Strong distance*). The *strong distance* $\mathcal{SD}ist(s, \mathcal{G})$ of a state s from a goal $\mathcal{G}$ is the smallest integer i such that there is some complete set of traces T from s to $\mathcal{G}$ and i is the length of the longest trace in T. If no complete set of traces from s to $\mathcal{G}$ exists, then we define $\mathcal{SD}ist(s, \mathcal{G}) = \infty$.

Let π_i be the value of variable *SA* after i iterations of the while loop and let S_i be the set of states for which a solution is found at the ith iteration. Similarly to what happens in the weak case, S_i are exactly the states at strong distance i from $\mathcal{G}$.

Proposition 3.13. *Let s be a state in the planning domain. Then $s \in S_i$ iff $\mathcal{SD}ist(s, \mathcal{G}) = i$.*

Proposition 3.7 has also to be adapted to take into account the fact that StrongPreImage replaces WeakPreImage in the algorithm.

Proposition 3.14. *Let $i > 0$. If $s \in S_i$ and $\langle s, a\rangle \in \pi_i$, then*:

- $\text{Exec}(s, a) \subseteq \bigcup_{j<i} S_j$, *and*
- $\text{Exec}(s, a) \cap S_{i-1} \neq \emptyset$.

The main results on the strong planning algorithm follow.

Theorem 3.15 (Correctness). *Let $\pi \doteq \text{StrongPlan}(\mathcal{I}, \mathcal{G}) \neq Fail$. Then π is a strong solution of the planning problem $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G}\rangle$.*

If $\text{StrongPlan}(\mathcal{I}, \mathcal{G}) = Fail$, instead, then there is no strong solution for planning problem P.

Theorem 3.16 (Termination). *Function* StrongPlan *always terminates.*

Theorem 3.17 (Optimality). *Let* $\pi \doteq \text{StrongPlan}(\mathcal{I}, \mathcal{G}) \neq \textit{Fail}$. *Then plan* π *is optimal with respect to the strong distance: namely, for each* $s \in \mathcal{I}$, *in the execution structure for* π *all the execution paths from* s *to* $\mathcal{G}$ *have a length smaller of equal to* $\mathcal{SD}ist(s, \mathcal{G})$.

4. Algorithm for strong cyclic planning

In this section we tackle the problem of strong cyclic planning. The main difference with the algorithms presented in previous section is that here the resulting plans allow for infinite behaviours: loops must no longer be eliminated, but rather controlled, i.e., only certain, "good" loops must be kept. As discussed in Section 2.3, infinite executions are accepted only if they correspond to "unlucky" patterns of nondeterministic outcomes, and if a goal state can be reached from each state of the execution under different patterns of nondeterministic outcomes.

The strong cyclic planning algorithm is presented in Fig. 6. The algorithm starts to analyze the universal state-action table with respect to the problem being solved, and eliminates all those state-action pairs which are discovered to be source of potential "bad" loops, or to lead to states which have been discovered not to allow for a solution. With respect to the algorithms presented in previous section, here the set of states associated with the state-action table being constructed is reduced rather than being extended: this approach amounts to computing a greatest fix-point.

The starting state-action table in function StrongCyclicPlan is the universal state-action table *UnivSA*. It contains all state-action pairs that satisfy the applicability conditions:

$$\textit{UnivSA} \doteq \{\langle s, a\rangle\colon a \in \text{Act}(s)\}.$$

The "elimination" phase, where unsafe state-action pairs are discarded, corresponds to the while loop of function StrongCyclicPlan. It is based on the repeated application of the functions PruneOutgoing and PruneUnconnected. The role of PruneOutgoing is to remove all those state-action pairs which may lead out of $G \cup \text{StatesOf}(SA)$, which is the current set of potential solutions. Because of the elimination of these actions, from certain states it may become impossible to reach the set of goal states. The role of PruneUnconnected is to identify and remove such states. Due to this removal, the need may arise to eliminate further outgoing transitions, and so on. The elimination loop is quit when convergence is reached. The resulting state-action table is guaranteed to generate executions which either terminate in the goal or loop forever on states from which it is possible to reach the goal. Function StrongCyclicPlan then checks whether the computed state-action table *SA* defines a plan for all the initial states, i.e., $\mathcal{I} \subseteq \mathcal{G} \cup \text{StatesOf}(SA)$. If this is not the case a failure is returned.

As the following example shows, the state-action table obtained after the elimination loop is not necessarily a valid solution for the planning problem.

```
1. function StrongCyclicPlan(I, G);
2.    OldSA := ∅;
3.    SA := UnivSA;
4.    while (OldSA ≠ SA) do
5.       OldSA := SA;
6.       SA := PruneUnconnected(PruneOutgoing(SA, G), G);
7.    done ;
8.    if (I ⊆ (G ∪ StatesOf(SA))) then
9.       return RemoveNonProgress(SA, G);
10.   else
11.      return Fail;
12.   fi ;
13. end ;

1. function PruneUnconnected(SA, G);
2.    NewSA := ∅;
3.    repeat
4.       OldSA := NewSA;
5.       NewSA := SA ∩ WeakPreImage(G ∪ StatesOf(NewSA));
6.    until (OldSA = NewSA);
7.    return NewSA;
8. end ;

1. function PruneOutgoing(SA, G);
2.    NewSA := SA \ ComputeOutgoing(SA, G ∪ StatesOf(SA));
3.    return NewSA;
4. end ;

1. function RemoveNonProgress(SA, G);
2.    NewSA := ∅;
3.    repeat
4.       PreImage := SA ∩ WeakPreImage(G ∪ StatesOf(NewSA));
5.       OldSA := NewSA;
6.       NewSA := NewSA ∪ PruneStates(PreImage, G ∪ StatesOf(NewSA));
7.    until (OldSA = NewSA);
8.    return NewSA;
9. end ;
```

Fig. 6. The algorithm for strong cyclic planning.

Example 4.1. Consider the omelette domain, and the planning problem of reaching state `7` from state `1`. Action `discard` in state `3` is "safe": if executed, it leads in state `1`, where the goal is still reachable. However, this action does not contribute to reach the goal. On the contrary, it leads back to the initial state and, to reach the goal from `1`, it is necessary to move again to state `3`. Moreover, if action `discard` is performed whenever the execution is in state `3`, then the goal is never reached.

The state-action table obtained after the elimination loop may contain state-action pairs like ⟨3, `discard`⟩ that, while preserving the reachability of the goal, still do not perform any progress toward it. In the strong cyclic planning algorithm, function REMOVENONPROGRESS on line 9 takes care of removing all those actions from a state whose outcomes do not lead to any progress toward the goal. This function is very similar to the weak planning algorithm: it iteratively extends the state-action table by considering states at an increasing distance from the goal. In this case, however, the weak preimage computed at any iteration step is restricted to the state-action pairs that appear in the input state-action table, and hence that are "safe" according to the elimination phase.

Functions PRUNEOUTGOING, PRUNEUNCONNECTED, and REMOVENONPROGRESS are presented in Fig. 6. They are based on the primitives WEAKPREIMAGE and PRUNESTATES already defined in Section 3, and on the primitive COMPUTEOUTGOING, that takes as input a state-action table *SA* and a set of states *S*, and returns those state-action pairs which are not guaranteed to result in states in *S*:

$$\text{COMPUTEOUTGOING}(SA, S) \doteq \{\langle s, a\rangle \in SA\colon \text{EXEC}(s, a) \not\subseteq S\}.$$

Example 4.2. Let us consider the omelette domain and the planning problem of reaching goal state `7` from the initial state `1`. The "elimination" phase of the algorithm does not remove any state-action pair from *UnivSA*. Indeed, the goal state is reachable from any state of the domain, and, as a consequence, there are no outgoing actions. Function REMOVENONPROGRESS, hence, takes as input the universal state-action table, and refines it taking only those actions that may lead to a progress toward the goal. The sequence π_i of state-action tables built by function REMOVENONPROGRESS is the following:

$$\begin{aligned}
\pi_0 &= \emptyset,\\
\pi_1 &= \{\langle 3, \texttt{break}\rangle, \langle 8, \texttt{open}\rangle\},\\
\pi_2 &= \{\langle 3, \texttt{break}\rangle, \langle 8, \texttt{open}\rangle, \langle 1, \texttt{break}\rangle, \langle 4, \texttt{open}\rangle\},\\
\pi_3 &= \{\langle 3, \texttt{break}\rangle, \langle 8, \texttt{open}\rangle, \langle 1, \texttt{break}\rangle, \langle 4, \texttt{open}\rangle,\\
&\qquad \langle 2, \texttt{discard}\rangle, \langle 5, \texttt{discard}\rangle, \langle 6, \texttt{discard}\rangle\},\\
\pi_4 &= \pi_3.
\end{aligned}$$

Since in this particular case the state-action table passed to REMOVENONPROGRESS is *UnivSA*, the initial iterations of the function are identical to the ones of algorithm WEAKPLAN described in Example 3.1. Here, however, the computation stops only after four iterations, when a fix-point is reached. The final state-action table is a strict subset of *UnivSA*. For instance, action `discard` is not allowed in states `3`, `4`, `7`, and `8`. Indeed, in these states we have made some progress toward the goal, and there is no reason to go back to the initial state.

4.1. Formal properties of the algorithm

Now we prove the correctness of the strong cyclic planning algorithm.

We start by defining when a state-action table π is *SC-valid* (or *strong-cyclical* valid) for a set of states G. Intuitively, if a state-action table *SA* is SC-valid for G, then it is a

strong cyclic solution of the planning problem of reaching G from any of the states in SA. Informally, a SC-valid state-action table should guarantee that the execution stops once the goal is reached: there is no reason to continue the execution in a goal state. Also, the execution of a SC-valid plan should never lead to states where the plan is undefined (with the exception of the goal states). Finally, we require that any action in the plan may lead to some progress in reaching the goal.

Definition 4.3 (*SC-valid state-action table*). State-action table π is *SC-valid* for G if:

- $\text{STATESOF}(\pi) \cap G = \emptyset$;
- if $\langle s, a\rangle \in \pi$ and $s' \in \text{EXEC}(s, a)$ then $s' \in \text{STATESOF}(\pi) \cup G$;
- given any determinization π_d of π and any state $s \in \text{STATESOF}(\pi)$ there is some trace from s to G that is compatible with π_d.

We remark that, in the last item, it is important to require that a trace from s to G exists for any determinization of π. Indeed, in this way we require that there are no state-action pairs that do not contribute to reach the goal.

A SC-valid state-action table is almost what is needed to have a strong cyclic solution for a planning problem. The only other property that we need to enforce on the state-action table is that the plan is defined for all the initial states.

Proposition 4.4. *Let π be a SC-valid plan for $\mathcal{G}$ and let $\mathcal{I} \subseteq \mathcal{G} \cup \text{STATESOF}(\pi)$. Then π is a strong cyclic solution for the planning problem $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G}\rangle$.*

Proof. Let π_d be any deterministic plan in π and let $K = \langle Q, T\rangle$ be the execution structure corresponding to π_d. We have to show that (1) all the terminal states of K are in $\mathcal{G}$ and that (2) all the states in K have a path to a state in $\mathcal{G}$.

To prove (1) we show, by induction on the definition of K, that $s \in Q$ implies $s \in \mathcal{G} \cup \text{STATESOF}(\pi_d)$: then, if $s \in Q$ and $s \notin \mathcal{G}$, we must have $s \in \text{STATESOF}(\pi_d)$, so s in not terminal in K. For the base step, if $s \in \mathcal{I}$, then $s \in \text{STATESOF}(\pi_d) \cup \mathcal{G}$, as $\mathcal{I} \subseteq \mathcal{G} \cup \text{STATESOF}(\pi)$ by hypothesis. For the inductive step, if $s' \in \text{EXEC}(s, a)$ with $s \in Q$ and $\langle s, a\rangle \in \pi_d \subseteq \pi$, then $s' \in \mathcal{G} \cup \text{STATESOF}(\pi)$ by definition of SC-valid state-action table. Since $\text{STATESOF}(\pi_d) = \text{STATESOF}(\pi)$, this implies $s' \in \mathcal{G} \cup \text{STATESOF}(\pi)$.

To prove (2) we exploit the fact that from any state $s \in \text{STATESOF}(\pi_d)$ there is a trace from s to $\mathcal{G}$ that is compatible with π_d. Indeed, it is easy to check that this trace corresponds to an execution path in K. Since $Q = \mathcal{G} \cup \text{STATESOF}(\pi_d)$, and since a trivial execution exists in K from $s \in \mathcal{G}$ to $\mathcal{G}$, we conclude that an execution in K exists for any $s \in Q$ to $\mathcal{G}$. □

The main result that guarantees the correctness of algorithm STRONGCYCLICPLAN is the fact that the state-action table computed by the while loop at lines 4–7 is a SC-valid state-action tables. This result is formalized in Proposition 4.8, and its proof relies on the following auxiliary lemmas.

Lemma 4.5. *Let π be the value of variable SA after the while loop at lines* 4–7 *in function* StrongCyclicPlan(I, G).

(1) *If* $\langle s, a\rangle \in \pi$ *and* $s' \in \text{Exec}(s, a)$, *then* $s' \in G \cup \text{StatesOf}(\pi)$.
(2) *If* $s \in \text{StatesOf}(\pi)$ *then there is a trace from* s *to* G *that is compatible with* π.

Proof. It is easy to see that $\pi = \text{PruneOutgoing}(\pi, G)$ and that $\pi = \text{PruneUnconnected}(\pi, G)$: indeed, both functions return a subset of the state-action table that they receive as a parameter, and the while loop at lines 4–7 ends only when a fix-point is reached.

In order to prove the first property, assume by contradiction that $\langle s, a\rangle \in \pi$ and $s' \in \text{Exec}(s, a)$, but $s' \notin G \cup \text{StatesOf}(\pi)$. By definition of ComputeOutgoing, we have $\langle s, a\rangle \in \text{ComputeOutgoing}(\pi)$ and hence $\langle s, a\rangle \notin \text{PruneOutgoing}(\pi) = \pi$, which is absurd.

For the second item, it is sufficient to observe that

$$G \cup \text{StatesOf}\big(\text{PruneUnconnected}(\pi, G)\big)$$

is exactly the set of states from which there is a trace compatible with π that leads to a state in G: this property is guaranteed by the call to function WeakPreImage in any iteration of the repeat-until loop at lines 3–6 of function PruneUnconnected. □

Lemma 4.6. *Let π be a generic state-action table and let*

$$\pi' = \text{RemoveNonProgress}(\pi, G).$$

Then:

(1) $\pi' \subseteq \pi$;
(2) $\text{StatesOf}(\pi') \cap G = \emptyset$;
(3) *given any determinization* π'_d *of* π' *and any state* $s \in \text{StatesOf}(\pi')$ *there is some trace from* s *to* G *that is compatible with* π'_d.

Proof. By inspection of the code of function RemoveNonProgress, it is possible to check that only pairs $\langle s, a\rangle \in \pi$ are added to the state-action table *NewSA*. Hence, $\pi' \subseteq \pi$ must hold.

Now we prove the other two properties.

Let π_i be the value of variable *NewSA* after i iterations of the repeat-until loop at lines 3–7 of function RemoveNonProgress. Also, let S_i be the set of states that are added to *NewSA* during the ith iteration (and $S_0 = G$).

By following arguments similar to the ones used in the proofs for the weak planning algorithm, it is possible to prove, by induction on i, that:

(a) $s \in S_i$ iff the shortest trace from s to G that is compatible with π has length i: this is analogous to Proposition 3.6.
(b) If $i \neq j$, then $S_i \cap S_j = \emptyset$: this is analogous to Lemma 3.5.

(c) If $s \in S_i$ and $\langle s, a\rangle \in \pi_i$ then there is some $s' \in \text{EXEC}(s, a)$ such that $s' \in S_{i-1}$ and there is no $s' \in \text{EXEC}(s, a)$ such that $s' \in S_j$ for $j < i - 1$: this is analogous to Proposition 3.7.

The fact that $\text{STATESOF}(\pi') \cap G = \emptyset$ is a consequence of property (b), since $G = S_0$ and $\text{STATESOF}(\pi') = \bigcup_{i>0} S_i$.

Let π'_d be any determinization of π' and let $s \in \text{STATESOF}(\pi')$. By induction on i it is possible to show that if $s \in S_i$ then there is some trace from s to G that is compatible with π'_d. This is a consequence of properties (c): indeed, the trace from s to G visits, in turn, states in S_j for $j = i, i-1, \ldots, 1, 0$.

Since $\text{STATESOF}(\pi'_d) = \text{STATESOF}(\pi') = \bigcup_{i>0} S_i$, we conclude that for any $s \in \text{STATESOF}(\pi')$ there is some trace from s to G that is compatible with π'_d. □

Lemma 4.7. *Let π be the value of variable SA after the while loop at lines* 4–7 *in function* STRONGCYCLICPLAN(I, G). *Moreover, let*

$$\pi' = \text{REMOVENONPROGRESS}(\pi, G).$$

Then $\text{STATESOF}(\pi') = \text{STATESOF}(\pi) \setminus G$.

Proof. By Lemma 4.5(2), if $s \in \text{STATESOF}(\pi)$ then there is a trace from s to G that is compatible with π. Let i be the length of the shortest trace from s to G that is compatible with π. By using the notations introduced in the proof of Lemma 4.6, and by exploiting property (a) in that proof, we deduce that $s \in S_i$. Now, if $i = 0$ then $s \in S_0 = G$; otherwise $s \in \bigcup_{i>0} S_i = \text{STATESOF}(\pi')$. Therefore $\text{STATESOF}(\pi) \subseteq \text{STATESOF}(\pi') \cup G$, and hence $\text{STATESOF}(\pi) \setminus G \subseteq \text{STATESOF}(\pi')$.

The converse inclusion, namely $\text{STATESOF}(\pi') \subseteq \text{STATESOF}(\pi) \setminus G$, is a consequence of $\pi' \subseteq \pi$ and of $\text{STATESOF}(\pi') \cap G = \emptyset$. □

Proposition 4.8. *Let π be the value of variable SA after the while loop at lines* 4–7 *in function* STRONGCYCLICPLAN(I, G). *Moreover, let*

$$\pi' = \text{REMOVENONPROGRESS}(\pi, G).$$

Then π' is a SC-valid state-action table for G.

Proof. By Lemma 4.6(2), $\text{STATESOF}(\pi') \cap G = \emptyset$.

Let $\langle s, a\rangle \in \pi'$ and $s' \in \text{EXEC}(s, a)$. We prove that $s' \in G \cup \text{STATESOF}(\pi')$. By Lemma 4.6(1), $\langle s, a\rangle \in \pi$. Then, by Lemma 4.5(1) $s' \in G \cup \text{STATESOF}(\pi)$. Hence $s' \in G \cup \text{STATESOF}(\pi')$ by Lemma 4.7.

Finally, by Lemma 4.6(3), for any determinization π'_d of π' and for any state $s \in \text{STATESOF}(\pi')$ there is a trace from s to G compatible with π'_d.

This concludes the proof that π' is a SC-valid state-action table for G. □

By combining Propositions 4.4 and 4.8 it is easy to prove that, if function STRONGCYCLICPLAN returns a state-action table $\pi \neq \mathit{Fail}$, then π is a strong cyclic solution. The converse proof, namely that a state-action table is returned whenever a strong

cyclic solution exists, relies on the fact that function StrongCyclicPlan starts the elimination loop on the universal state-action table: starting from *UnivSA*, it is impossible to prune away states for which the goal is reachable in a strong cyclic way. This fact is proved in the following lemma.

Lemma 4.9. *Let π be a state-action table that is SC-valid for G and let π_p be the value of variable SA in function* StrongCyclicPlan(I, G) *at the end of the while loop at lines* 4–7. *Then $\pi \subseteq \pi_p$.*

Proof. We show, by induction on i, that $\pi \subseteq \pi_i$, where π_i is the value of variable *SA* in function StrongCyclicPlan(I, G) after i iterations of the while loop.

Case $i = 0$. By hypothesis, $\pi \subseteq \pi_0$.

Case $i = i' + 1$. We observe that, since π is a SC-valid state-action table for G, then it satisfies properties $\pi =$ PruneOutgoing(π, G) and $\pi =$ PruneUnconnected(π, G): indeed, by definition SC-valid state-action tables cannot contain outgoing actions, nor states from which the goal is unreachable.

Now we prove that $\pi \subseteq$ PruneOutgoing$(\pi_{i'}, G)$. By the inductive hypothesis, $\pi \subseteq \pi_{i'}$. By inspection it is easy to check that function PruneOutgoing is monotonic in its first parameter. Therefore, $\pi =$ PruneOutgoing$(\pi, G) \subseteq$ PruneOutgoing$(\pi_{i'}, G)$.

Similarly, by the monotonicity of function PruneUnconnected we deduce

$$\pi \subseteq \text{PruneUnconnected}\big(\text{PruneOutgoing}(\pi_{i'}, G), G\big)$$

from $\pi \subseteq$ PruneOutgoing$(\pi_{i'}, G)$. Since, by definition,

$$\pi_i = \pi_{i'+1} = \text{PruneUnconnected}\big(\text{PruneOutgoing}(\pi_{i'}, G), G\big),$$

this concludes the proof of the inductive step. □

Theorem 4.10 (Correctness). *Let $\pi \doteq$ StrongCyclicPlan$(\mathcal{I}, \mathcal{G}) \neq$ Fail. Then π is a strong cyclic solution of the planning problem $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$.*

If StrongCyclicPlan$(\mathcal{I}, \mathcal{G}) =$ *Fail, instead, then there is no strong cyclic solution for planning problem P.*

Proof. Assume $\pi \neq$ *Fail*; we show that π is a strong cyclic solution for P. First of all, $\mathcal{I} \subseteq \mathcal{G} \cup$ StatesOf(π), otherwise function StrongCyclicPlan returns *Fail*. Also, by Proposition 4.8, state-action table π is SC-valid for $\mathcal{G}$. Then, by Proposition 4.4, π is a strong cyclic solution. This concludes the proof that any deterministic plan in $\pi \neq$ *Fail* is a strong cyclic solution for planning problem P.

Now we show that, if $\pi =$ *Fail* then there is no plan that is a strong cyclic solution for the planning problem. By contradiction, assume π is a solution. We can assume, without loss of generality, that all the states in StatesOf(π) are reachable from $\mathcal{I}$, and that no states in $\mathcal{G}$ appear in π. Indeed, if we restrict any solution π to these states we still have a valid solution. It is easy to check that π is a SC-valid state-action table for $\mathcal{G}$. So, by Lemma 4.9, $\pi \subseteq \pi_p$ and hence StatesOf$(\pi) \subseteq$ StatesOf(π_p), where π_p is the value of variable *SA* in function StrongCyclicPlan(I, G) at the end of the while loop at lines 4–7. Also $\mathcal{I} \subseteq \mathcal{G} \cup$ StatesOf(π), otherwise π would not be a solution for

P. Summing up, $\mathcal{I} \subseteq \mathcal{G} \cup \text{STATESOF}(\pi) \subseteq \mathcal{G} \cup \text{STATESOF}(\pi_p)$. Therefore, condition on line 8 is satisfied and function STRONGCYCLICPLAN$(\mathcal{I}, \mathcal{G})$ does not return *Fail*, which contradicts our hypothesis. □

The strong cyclic planning algorithm terminates.

Theorem 4.11 (Termination). *Function* STRONGCYCLICPLAN *always terminates.*

Proof. The fact that the while loop at lines 4–7 of function STRONGCYCLICPLAN always terminates derives from the following observations:

- function PRUNEOUTGOING always terminates: it does not contain loops;
- function PRUNEUNCONNECTED always terminates: the proof is similar to the one of Theorem 3.9;
- at any iteration, either the value of variable *SA* strictly decreases, or the loop terminates: indeed, functions PRUNEOUTGOING and PRUNEUNCONNECTED return a subset of the state-action table that receive as parameter.

In order to prove that function STRONGCYCLICPLANAUX terminates, it remains to show that function REMOVENONPROGRESS always terminates. The proof of this property is similar to the proof of Theorem 3.9. □

5. Planning via symbolic model checking

In this section we discuss the Planning via Symbolic Model Checking approach. We first present the ideas underlying Symbolic Model Checking (SMC), and point out what our approach inherits from SMC and in which directions it extends SMC (Section 5.1). Then, we show how planning domains and primitives can be represented in terms of logical, symbolic transformations (Sections 5.2 and 5.3). Finally, we discuss Binary Decision Diagrams (BDDs), the machinery for an efficient implementation of the symbolic approach (Section 5.4).

5.1. Symboling model checking: Overview and discussion

Model checking is a formal verification technique, where a reactive system (e.g., a communication protocol, a hardware design) is modeled as a Finite State Machine (FSM). Requirements over the behaviours of the system are modeled as formulae in a temporal logic, for instance Computation Tree Logic (CTL) [33]. The model checking problem $\mathcal{M} \models \phi$ is to detect if all the behaviours of the FSM $\mathcal{M}$ satisfy the constraints specified by the temporal formula ϕ. Model checking algorithms are based on the exhaustive exploration of the FSM [26]. When the specification is not satisfied, they are able to construct a counterexample, i.e., to produce a description of the system behaviour that does not satisfy the specification.

Our approach is related to model checking by the fact that a planning domain is represented as a FSM, and we inherit from model checking the standard techniques for the representation and traversal of FSMs. The main difference is that planning tackles the more complex problem of *finding* a plan such that a certain behaviour is achieved when the plan is executed in the domain. There are cases in which a planning problem can be reduced to a model checking problem. This is the case, for instance, of classical planning, i.e., planning for reachability goals in deterministic domains, where finding a plan corresponds to the model checking problem of finding one path from the initial state to the goal; the plan to the goal is the sequence of actions that produce the path. In the general case of nondeterministic domains, however, a planning problem cannot be reduced to model checking. Consider for instance the case of a strong planning problem: it requires to *find* a suitable restriction of the behaviours of the domain, i.e., the plan, such that, on *all* the compatible paths, a goal state is reached.

There are further relationships between model checking and planning. The execution structure induced from the connection of a plan π to the domain $\mathcal{D}$ can be presented as a simple synchronous composition of FSMs, $\mathcal{M}_{\mathcal{D}} \times \mathcal{M}_{\pi}$. $\mathcal{M}_{\mathcal{D}}$ describes the domain, while $\mathcal{M}_{\pi}$ represents the plan. Moreover, weak, strong and strong cyclic goals can be expressed in temporal logic. For instance, strong solutions can be expressed by the CTL formula AF($\mathcal{G}$), read "for all paths, there is a future instant where $\mathcal{G}$ holds".[2] Thus, plan *validation* can be carried out with a straightforward application of standard model checking techniques, by checking if the formula corresponding to the goal holds on the execution structure. For instance, $\mathcal{M}_{\mathcal{D}} \times \mathcal{M}_{\pi} \models AF(\mathcal{G})$ is the model checking problem corresponding to the validation that plan π is a strong solution for goal $\mathcal{G}$.

Symbolic model checking [49] is a particular form of model checking, where propositional formulae are used for the compact representation of FSMs, and transformations over propositional formulae provide a basis for efficient exploration. The symbolic encoding is efficiently implemented by means of BDDs [15]. This allows for the analysis of extremely large systems [16]. As a result, symbolic model checking is routinely applied in industrial hardware design, and is taking up in other application domains (see [27] for a survey). In our approach, we inherit the symbolic mechanisms and the BDD-based implementation techniques for representing and exploring planning domains. We extend these techniques with a symbolic representation of state-action tables, and with algorithms that are able to synthesize the plan during the exploration of the state space of the FSM.

5.2. Symbolic representation of planning domains

A planning domain $\langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ can be symbolically represented by the standard machinery developed for symbolic model checking. A vector of (distinct) propositional variables $\boldsymbol{x}$, called *state* variables, is devoted to the representation of the states of the domain. Each of these variables has a direct association with a proposition of the domain in $\mathcal{P}$ used in the description of the domain. Therefore, in the rest of this section we will not distinguish a proposition and the corresponding propositional variable. For instance, for

[2] See [53] for a more thorough discussion of the relations with temporally extended goals.

the omelette domain, $\boldsymbol{x}$ is {#eggs=0, #eggs=1, #eggs=2, bad, good, unbroken}. A state is the set of propositions of $\mathcal{P}$ that are intended to hold in it. For each state s, there is a corresponding assignment to the state variables in $\boldsymbol{x}$, i.e., the assignment where each variable in s is assigned to *True*, and all the other variables are assigned to *False*. We represent s with a propositional formula $\xi(s)$, having such an assignment as its unique satisfying assignment. For instance, the formulae representing state 2 and 8 in the omelette example are:

$$\begin{aligned}\xi(2) \doteq\ & \neg\texttt{\#eggs=0} \wedge \texttt{\#eggs=1} \wedge \neg\texttt{\#eggs=2} \wedge \neg\texttt{good} \wedge \texttt{bad} \\ & \wedge \neg\texttt{unbroken},\\ \xi(8) \doteq\ & \neg\texttt{\#eggs=0} \wedge \neg\texttt{\#eggs=1} \wedge \texttt{\#eggs=2} \wedge \texttt{good} \wedge \neg\texttt{bad} \\ & \wedge \texttt{unbroken}.\end{aligned}$$

This representation naturally extends to any *set of states* $Q \subseteq \mathcal{S}$ as follows:

$$\xi(Q) \doteq \bigvee_{s \in Q} \xi(s).$$

In this way, we associate a set of states with the generalized disjunction of the formulae representing each of the states. The satisfying assignments of $\xi(Q)$ are exactly the assignments representing any state in Q. We can use such a formula to represent the set $\mathcal{S}$ of all the states of the domain. In the case of the omelette example, the formula $\xi(\mathcal{S})$ would be (equivalent to) the following formula:

$$\begin{pmatrix} (\ \texttt{\#eggs=0} \wedge \neg\texttt{\#eggs=1} \wedge \neg\texttt{\#eggs=2}) \vee \\ (\neg\texttt{\#eggs=0} \wedge\ \texttt{\#eggs=1} \wedge \neg\texttt{\#eggs=2}) \vee \\ (\neg\texttt{\#eggs=0} \wedge \neg\texttt{\#eggs=1} \wedge\ \texttt{\#eggs=2}) \end{pmatrix} \wedge \\ \begin{pmatrix} (\ \texttt{good} \wedge \neg\texttt{bad}) \vee \\ (\neg\texttt{good} \wedge\ \texttt{bad}) \end{pmatrix} \wedge \begin{pmatrix} (\texttt{\#eggs=0} \rightarrow \neg\texttt{bad} \wedge \neg\texttt{unbroken}) \vee \\ (\texttt{\#eggs=1} \rightarrow \neg\texttt{bad} \vee \neg\texttt{unbroken}) \end{pmatrix}.$$

We use a propositional formula as representative for the set of assignments that satisfy it (and hence for the corresponding set of states), so we abstract away from the actual syntax of the formula used: we do not distinguish among equivalent formulae as they represent the same sets of assignments. (Although the actual syntax of the formula may have a computational impact, the use of BDDs as representatives of sets of models is indeed practical.) The main efficiency of the symbolic representation is in that the cardinality of the represented set is not directly related to the size of the formula. For instance, in the limit cases, $\xi(2^{\mathcal{P}})$ and $\xi(\emptyset)$, are the *True* and *False* formulae, independently of the cardinality of $\mathcal{P}$. As a further advantage, the symbolic representation can deal quite effectively with irrelevant information. For instance, notice that the variables good, bad and unbroken need not to appear in the formula $\xi(\{5,6,7,8\}) = \neg\texttt{\#eggs=2}$. For this reason, a symbolic representation can have a dramatic improvement over an explicit, enumerative representation. This is what allows symbolic model checkers to handle finite state automata with a very large number of states (see for instance [16]).

Another advantage of the symbolic representation is the natural encoding of set theoretic transformations (e.g., union, intersection, complementation) into propositional operations, as follows:

$$\xi(Q_1 \cup Q_2) \doteq \xi(Q_1) \vee \xi(Q_2),$$
$$\xi(Q_1 \cap Q_2) \doteq \xi(Q_1) \wedge \xi(Q_2),$$
$$\xi(\mathcal{S} \backslash Q) \doteq \xi(\mathcal{S}) \wedge \neg\xi(Q).$$

Also the predicates over sets of states have a symbolic counterpart: for instance, testing $Q_1 = Q_2$ amounts to checking the validity of the formula $\xi(Q_1) \leftrightarrow \xi(Q_2)$, while testing $Q_1 \subseteq Q_2$ corresponds to checking the validity of $\xi(Q_1) \rightarrow \xi(Q_2)$.

In order to represent actions, we use another set of propositional variables, called *action* variables, written $\boldsymbol{\alpha}$. One approach is to use one action variable for each possible action in $\mathcal{A}$. Intuitively, an action variable is true if and only if the corresponding action is being executed. In principle this allows for the representation of concurrent actions. If a sequential encoding is used, i.e., no concurrent actions are allowed, a mutual exclusion constraint stating that exactly one of the action variables must be true at each time must imposed. In the following, we call $\textsc{Seq}_{\mathcal{A}}(\boldsymbol{\alpha})$ the formula, in the action variables, expressing the mutual exclusion constraint over $\mathcal{A}$. In the case of the omelette example,

$$\textsc{Seq}_{\mathcal{A}}(\boldsymbol{\alpha}) \doteq \begin{pmatrix} (\ \texttt{break} \wedge \neg\texttt{open} \wedge \neg\texttt{discard}) \vee \\ (\neg\texttt{break} \wedge\ \ \texttt{open} \wedge \neg\texttt{discard}) \vee \\ (\neg\texttt{break} \wedge \neg\texttt{open} \wedge\ \ \texttt{discard}) \end{pmatrix}.$$

In the specific case of sequential encoding, it is possible to use only $\lceil \log \|\mathcal{A}\| \rceil$ action variables, where each assignment to the action variables denotes a specific action to be executed. Furthermore, being two assignments mutually exclusive, the constraint $\textsc{Seq}_{\mathcal{A}}(\boldsymbol{\alpha})$ needs not to be represented. When the cardinality of $\mathcal{A}$ is not a power of two, the standard solution is to associate more than one assignment to certain values. For instance, in the omelette domain two variables α_0 and α_1 are sufficient to represent actions `break`, `open` and `discard`. A possible encoding is the following:

$$\xi(\texttt{break}) \doteq \neg\alpha_0, \qquad \xi(\texttt{open}) \doteq \alpha_0 \wedge \neg\alpha_1, \qquad \xi(\texttt{discard}) \doteq \alpha_0 \wedge \alpha_1.$$

A transition is a 3-tuple composed of a state (the initial state of the transition), an action (the action being executed), and a state (the resulting state of the transition). To represent transitions, another vector $\boldsymbol{x}'$ of propositional variables, called *next state* variables, is used. We require that $\boldsymbol{x}$ and $\boldsymbol{x}'$ have the same number of variables, and that the variables in similar positions in $\boldsymbol{x}$ and in $\boldsymbol{x}'$ correspond. We write $\xi'(s)$ for the representation of the state s in the next state variables. With $\xi'(Q)$ we denote the formula corresponding to the set of states Q, using each variable in the next state vector $\boldsymbol{x}'$ instead of each current state variables $\boldsymbol{x}$. In the following, we indicate with $\Phi[v/\Psi]$ the formula resulting from the substitution of v with Ψ in Φ, where v is a variable, and Φ and Ψ are formulae. If $\mathbf{v_1}$ and $\mathbf{v_2}$ are vectors of (the same number of) distinct variables, we indicate with $\Phi[\mathbf{v_1}/\mathbf{v_2}]$ the parallel substitution in Φ of the variables in vector $\mathbf{v_1}$ with the (corresponding) variables in $\mathbf{v_2}$. We define the representation of a set of states in the next variables as follows:

$$\xi'(s) \doteq \xi(s)[\boldsymbol{x}/\boldsymbol{x}'].$$

We call the operation $\Phi[\boldsymbol{x}/\boldsymbol{x}']$ "forward shifting", because it transforms the representation of a set of "current" states in the representation of a set of "next" states. The dual operation

$\Phi[\boldsymbol{x}'/\boldsymbol{x}]$ is called "backward shifting". In the following, we call $\boldsymbol{x}$ *current* state variables to distinguish them from next state variables.

A transition is represented as an assignment to $\boldsymbol{x}$, $\boldsymbol{\alpha}$ and $\boldsymbol{x}'$. For the omelette example, the single transition corresponding to the application of action `break` in state 1 and resulting exactly in state 2 is represented by the formula

$$\xi\big(\langle 1, \texttt{break}, 2\rangle\big) \doteq \xi(1) \wedge \texttt{break} \wedge \xi'(2).$$

The transition relation $\mathcal{R}$ of the automaton corresponding to a planning domain is simply a set of transitions, and is thus represented by a formula in the variables $\boldsymbol{x}$, $\boldsymbol{\alpha}$ and $\boldsymbol{x}'$, where each satisfying assignment represents a possible transition:

$$\xi(\mathcal{R}) \doteq \textsc{Seq}_{\mathcal{A}}(\boldsymbol{\alpha}) \wedge \bigvee_{t \in \mathcal{R}} \xi(t).$$

In the rest of this paper, we assume that the symbolic representation of a planning domain and of a planning problem are given. In particular, we assume as given the vectors of variables $\boldsymbol{x}$, $\boldsymbol{\alpha}$ and $\boldsymbol{x}'$, the encoding functions ξ and ξ', and we simply call $\mathcal{S}(\boldsymbol{x})$, $\mathcal{R}(\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{x}')$, $\mathcal{I}(\boldsymbol{x})$ and $\mathcal{G}(\boldsymbol{x})$ the formulae representing the states of the domain, the transition relation, the initial states and the goal states, respectively. Also, we will represent the formula $\xi(Q)$, corresponding to the set of states $Q \subseteq \mathcal{S}$, with $Q(\boldsymbol{x})$, and $\xi'(Q)$ as $Q(\boldsymbol{x}')$.

In order to operate over relations, we use quantification in the style of QBF (the logic of Quantified Boolean Formulae), a definitional extension to propositional logic, where propositional variables can be universally and existentially quantified. If Φ is a formula, and v_i is one of its variables, the existential quantification of v_i in Φ, written $\exists v_i.\Phi(v_1, \ldots, v_n)$, is equivalent to

$$\Phi(v_1, \ldots, v_n)[v_i/\mathit{False}] \vee \Phi(v_1, \ldots, v_n)[v_i/\mathit{True}].$$

Analogously, the universal quantification $\forall v_i.\Phi(v_1, \ldots, v_n)$ is equivalent to

$$\Phi(v_1, \ldots, v_n)[v_i/\mathit{False}] \wedge \Phi(v_1, \ldots, v_n)[v_i/\mathit{True}].$$

QBF formulae allow for an exponentially more compact representation than propositional formulae.

As an example of the application of QBF formulae, the symbolical representation of the *image* of a set of states Q, i.e., the set of states reachable from any state in Q by applying any action, is the following:

$$\big(\exists \boldsymbol{x}\boldsymbol{\alpha}.\big(\mathcal{R}(\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{x}') \wedge Q(\boldsymbol{x})\big)\big)[\boldsymbol{x}'/\boldsymbol{x}].$$

Notice that, with this single operation, we symbolically simulate the effect of the application of any applicable action in $\mathcal{A}$ to any of the states in Q. The dual backward image is described as follows:

$$\big(\exists \boldsymbol{x}'\boldsymbol{\alpha}.\big(\mathcal{R}(\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{x}') \wedge Q(\boldsymbol{x}')\big)\big).$$

5.3. Symbolic representation of plans and planning algorithms

The machinery for the symbolic representation of planning domains can be used to represent and manipulate symbolically the other structures of the planning algorithms,

most notably state-action tables. A state-action table SA is a relation between states and actions, and can be represented symbolically as a formula in the $\boldsymbol{x}$ and $\boldsymbol{\alpha}$ variables. In the following, we write $SA(\boldsymbol{x}, \boldsymbol{\alpha})$ for the formula corresponding to state-action table SA: each satisfying assignment to $SA(\boldsymbol{x}, \boldsymbol{\alpha})$ represents a state-action pair in SA. This view inherits all the properties seen above for sets of states. For instance, the symbolic representation of the union of two state-action tables $SA_1 \cup SA_2$ is represented by the disjunction of their symbolic representations $SA_1(\boldsymbol{x}, \boldsymbol{\alpha}) \vee SA_2(\boldsymbol{x}, \boldsymbol{\alpha})$. The set of states of a state-action table STATESOF($SA(\boldsymbol{x}, \boldsymbol{\alpha})$) is represented symbolically as $\exists \boldsymbol{\alpha}.SA(\boldsymbol{x}, \boldsymbol{\alpha})$. The set of actions associated in SA with a given state s, i.e., the set of possible results for the GETACTION primitive in the reactive execution loop presented in Section 2.2, is represented symbolically by $\exists \boldsymbol{x}.(SA(\boldsymbol{x}, \boldsymbol{\alpha}) \wedge \xi(s))$. The symbolic representation of the universal state-action table *UnivSA* is $\exists \boldsymbol{x}'.\mathcal{R}(\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{x}')$, that also represents the applicability relation of actions in states.

We now describe how the planning algorithms can be seen in terms of transformations over propositional formulae. The basic steps of the algorithms are the generalizations of the preimage operations, that rather than returning sets of states, construct state-action tables. WEAKPREIMAGE(Q) corresponds to

$$\exists \boldsymbol{x}'.\big(\mathcal{R}(\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{x}') \wedge Q(\boldsymbol{x}')\big)$$

while STRONGPREIMAGE(Q) corresponds to

$$\forall \boldsymbol{x}'.\big(\mathcal{R}(\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{x}') \rightarrow Q(\boldsymbol{x}')\big) \ \wedge \ \text{APPLICABLE}(\boldsymbol{x}, \boldsymbol{\alpha})$$

where the applicability relation APPLICABLE($\boldsymbol{x}, \boldsymbol{\alpha}$) is

$$\exists \boldsymbol{x}'.\mathcal{R}(\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{x}').$$

In both cases, the resulting formula is obtained as a one-step computation, and may compactly describe an extremely large set. The other basic ingredients in the planning algorithms are functions PRUNESTATES(SA, Q), that can be computed as

$$SA(\boldsymbol{x}, \boldsymbol{\alpha}) \wedge \neg Q(\boldsymbol{x})$$

and function COMPUTEOUTGOING(SA, Q), that corresponds to

$$SA(\boldsymbol{x}, \boldsymbol{\alpha}) \wedge \exists \boldsymbol{x}'\big(\mathcal{R}(\boldsymbol{x}, \boldsymbol{\alpha}, \boldsymbol{x}') \wedge \neg Q(\boldsymbol{x}')\big).$$

Given the basic building blocks just defined, the algorithms presented in previous sections can be symbolically implemented by replacing, within the same control structure, each function call with the symbolic counterpart, and by casting the operations on sets into the corresponding operations on propositional formulae.

5.4. Binary decision diagrams

Reduced Ordered Binary Decision Diagrams [15] (in the following simply called BDDs) are the first and most popular implementation device of symbolic model checking (see also [1,10] for alternative symbolic representation mechanisms). BDDs provide a general interface that allows for a direct map of the symbolic representation mechanisms presented in previous section (e.g., tautology checking, quantification, shifting).

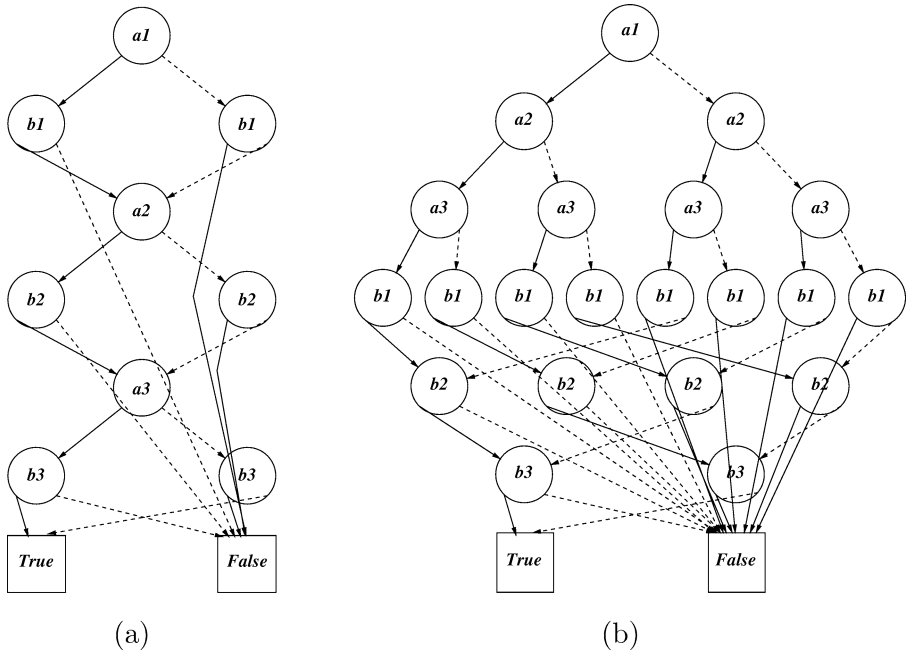

Fig. 7. Two BDD for the formula $(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2) \wedge (a_3 \leftrightarrow b_3)$.

A BDD is a directed acyclic graph (DAG). The terminal nodes are either *True* or *False*. Each non-terminal node is associated with a Boolean variable, and two BDDs, called left and right (or high and low) branches. Fig. 7(a) depicts a BDD for $(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2) \wedge (a_3 \leftrightarrow b_3)$. At each non-terminal node, the right (left, respectively) branch is depicted as a solid (dashed, respectively) line, and represents the assignment of the value *True* (*False*, respectively) to the corresponding variable. BDDs provide a canonical representation of Boolean functions. Given a BDD, the value of the function corresponding to a given truth assignment to the variables is determined by traversing the graph from the root to the leaves, following each branch indicated by the value assigned to the variables. (A path from the root to a leaf can visit nodes associated with a subset of all the variables of the BDD. See for instance the path associated with $a_1, \neg b_1$ in Fig. 7(a).) The reached leaf node is labeled with the resulting truth value. If v is a BDD, its size $\|v\|$ is the number of its nodes. If n is a node, $var(n)$ indicates the variable indexing node n.

The canonicity of BDDs follows by imposing a total order $<$ over the set of variables used to label nodes, such that for any node n and respective non-terminal child m, their variables must be ordered, i.e., $var(n) < var(m)$, and requiring that the BDD contains no isomorphic subgraphs.

BDDs can be combined with the usual Boolean transformations (e.g., negation, conjunction, disjunction). Given two BDDs, for instance, the conjunction operator builds and returns the BDD corresponding to the conjunction of its arguments. Substitution and quantification can also be efficiently represented as BDD transformations. In terms of BDD computations, a quantification corresponds to a transformation mapping the BDD of Φ and

the variable v_i being quantified into the BDD of the resulting (propositional) formula. The time complexity of the algorithm for computing a truth-functional Boolean transformation $f_1 \langle op \rangle f_2$ is $O(\|f_1\| \cdot \|f_2\|)$. As far as quantifications are concerned, the time complexity is exponential in the number of variables being quantified.

BDD *packages* are efficient implementations of such data structures and algorithms (see [14]). A BDD package deals with a single multi-rooted DAG, where each node represents a Boolean function. Memory efficiency is obtained by using a "unique table", and by sharing common subgraphs between BDDs. The unique table is used to guarantee that at each time there are no isomorphic subgraphs and no redundant nodes in the multi-rooted DAG. Before creating a new node, the unique table is checked to see if the node is already present, and only if this is not the case a new node is created and stored in the unique table. The unique table allows to perform the equivalence check between two BDDs in constant time (since two equivalent functions always share the same subgraph) [14,62]. Time efficiency is obtained by maintaining a "computed table", which keeps tracks of the results of recently computed transformations, thus avoiding the recomputation.

A critical computational factor with BDDs is the order of the variables used. (Fig. 7 shows an example of the impact of a change in the variable ordering on the size of a BDD.) For certain classes of Boolean functions, the size of the corresponding BDD is exponential in the number of variables for any possible variable ordering. In many practical cases, however, finding a good variable ordering is rather easy. Beside affecting the memory used to represent a Boolean function, finding a good variable ordering can have a big impact on computation times, since the complexity of the transformation algorithms depends on the size of the operands. Most BDD packages provide heuristic algorithms for finding good variable orderings, which can be called to try to reduce the overall size of the stored BDDs. The reordering algorithms can also be activated dynamically by the package, during a BDD computation, when the total amount of nodes in the package reaches a predefined threshold (dynamic reordering).

6. The MBP planner

In this section, we give an overview of the functionalities and of the architecture of the Model Based Planner MBP, and describe its implementation. The version of MBP that is described here, and that has been used for the experiments, is available from http://sra.itc.it/tools/mbp. MBP is a general system for planning in nondeterministic domains, based on the symbolic model checking techniques described in previous section. It is built on top of the symbolic model checker NuSMV [17,18,20], that provides an efficient implementation of the symbolic techniques for the representation and exploration of domains. MBP can be seen as a two-stage system. The first stage processes the domain description provided in input and constructs an internal, BDD-based representation of the domain. In the second stage, different planning algorithms can be activated to solve planning problems of different kinds. These planning algorithms operate on the internal domain representation, and are therefore independent of the specific domain and of the language used for specifying it.

Domains descriptions can be provided to MBP in different languages. For each of them, a compiler encodes the description into a BDD-based representation of the automaton. Depending on the characteristics of the language, a description can be more or less compact, and different effort can be required to the compiler to generate the automaton. For instance, MBP accepts descriptions in the high-level action language $\mathcal{AR}$ [34], that allows for the specification of conditional effects and uncertain effects of actions by means of high level assertions. The semantics of $\mathcal{AR}$ solves the frame problem and the ramification problem. The corresponding automaton is computed in MBP by means of the minimization procedure described in [19], that can however require a significant amount of computational resources. Furthermore, the semantics of $\mathcal{AR}$ yields a serial encoding, i.e., exactly one action is assumed to be executed at each time. Planning domains can also be described in MBP by means of the language of the NuSMV model checker upon which MBP is built. The level of the NuSMV language is slightly lower than $\mathcal{AR}$: basically, for each fluent in the domain, the effect of the actions is specified. The use of logical assertions allows to maintain the encodings reasonably compact. The use of the NuSMV language allows for a more direct encoding of the automaton, provides for parallel encodings of actions, and makes it possible to exploit all the advanced model checking techniques of NuSMV. The input languages of MBP allow for non-Boolean fluents, in particular for numerical ranges. For certain domains, this results in very compact encodings. Non-Boolean fluents are logarithmically encoded into Boolean variables by MBP during the compilation process. Both $\mathcal{AR}$ and the NuSMV language only allow for the description of *ground* domains, i.e., it is not possible to specify parametric operators, but only action instances. An extension of MBP able to process a nondeterministic extension of PDDL is currently under development. In the rest of this paper, the problems given in input to MBP are written in the language of NuSMV.

After the BDD-based representation of a domain description has been built, different planning algorithms can be applied to the specified planning problems. The planning algorithms operate solely on the automaton representation, and are completely independent of the particular language used to specify the domain. In this paper, we concentrate on the automatic construction of state-action tables under full observability. Other functionalities of MBP, out of the scope of this paper, are conformant planning [5,8,21,22], conditional planning under partial observability [7,9], and planning for temporally extended goals [28, 52,53].

From the point of view of the implementation, MBP is built on top and strictly integrated with NuSMV. NuSMV is a symbolic model checker originated from the reengineering, reimplementation and extension of SMV [49]. NuSMV combines the classical BDD-based techniques with Bounded Model Checking techniques based on propositional satisfiability (SAT) [20], and its development is carried out as an OpenSource project. NuSMV provides an open, general library for the symbolic representation and exploration of FSMs. The library includes advanced conjunctive partitioning techniques [16], that allow for a more effective construction and exploration of the domain FSM. Furthermore, NuSMV provides simulation and symbolic model checking functionalities, upon which we are developing interactive domain exploration and plan validation functionalities for MBP. In its internals, MBP inherits from NuSMV the use of CUDD [62], one of the best

BDD packages available. As we will see in next section, the efficiency of the lower level machinery is an important factor in the performance of MBP.

7. Experimental evaluation

In this section we experimentally evaluate the strength of our approach. We describe the state-of-the-art relevant planners (GPT [11], QBFPLAN [57], SIMPLAN [42], UMOP [40] and SGP [68]), we present the working hypotheses of the experimental evaluation, describe the selected problems and analyze the results.

7.1. State-of-the-art planners for nondeterministic domains

7.1.1. GPT

GPT [11] is an integrated planning environment, able to tackle conformant planning problems, and problems formulated in terms of Markov Decision Processes (MDPs) and Partially-Observable MDP. In this paper we focus on the MDP algorithms of GPT. (See [8, 21,22] for a comparison of GPT with the symbolic model checking approach on conformant planning and [9] for a comparison on partially observable planning.) GPT can deal with probabilistic distributions: in particular, it can take into account probability distributions of action transitions, since it makes use of MDP techniques to search through a stochastic automaton. GPT requires training examples to tend to optimal solutions. This process can be quite consuming, and checking the reached convergence is not obvious. Compared to MBP, GPT can not be forced to return an acyclic policy (or even checking for its existence), and is unable to distinguish between cyclic and acyclic solutions.

7.1.2. QbfPlan

QBFPLAN [57] generalizes the SATPLAN approach [43] to the case of planning in nondeterministic domains. In QBFPLAN, the planning problem is reformulated in QBF, rather than in propositional logic. The problem is then solved by an efficient QBF solver [58]. QBFPLAN is able to tackle a wide variety of conditional planning problems. The user must provide the structure of the plan to be generated (e.g., the length in the case of conformant planning, the number of control points and observations in the plan in the case of conditional planning). This can provide a significant limitation of the search space. In QBFPLAN, the points of nondeterminism are explicitly codified in terms of "environment variables", with a construction known as determinization of the transition relation. Compared to MBP, QBFPLAN is unable to decide whether the given problem admits a solution.

7.1.3. Simplan

SIMPLAN [42] implements different approaches to planning in nondeterministic domains. In this paper we focus on the logic-based planning component, where temporally extended goals can be expressed in (an extension of) Linear Temporal Logic (LTL) [54], and the associated plans are constructed under the hypothesis of full observability. SIMPLAN is very similar in spirit to MBP, since it is based on model checking techniques.

A major difference is that SIMPLAN relies on the *explicit-state* model checking paradigm, where individual states are manipulated (rather than sets of states). Therefore, the algorithms of SIMPLAN are of enumerative nature—this can be a major drawback in large domains. In SIMPLAN, LTL formulae are also used to describe user-defined control strategies, that can provide aggressive pruning of the search space. The classes of problems solved by MBP and SIMPLAN are incomparable. LTL allows for the specification of temporally extended goals. This includes strong planning, but not strong cyclic planning. In order to express strong cyclic planning, properties of branching computation structures are needed. These are expressible, for instance, in CTL [25]. In [53] our planning paradigm is extended to deal with goals expressed in full CTL. See also [66] for a comparison of linear time and branching time temporal logics.

7.1.4. UMOP

UMOP [40] is very similar in spirit to MBP. It implements strong and strong cyclic algorithms starting from the ideas presented in [23,24], and uses a BDD-based implementation. As input, UMOP takes a planning problem described in the expressive Nondeterministic Agent Domain Language (NADL). By exploiting the structure of the language, UMOP is able to activate some advanced model checking techniques, e.g., conjunctive partitioning of the transition relation. UMOP is also able to plan under the hypothesis of an adversarial environment, that is actively working to prevent goal achievement [41].

7.1.5. SGP

SGP [68] implements an observation-based approach to planning. It is able to plan in partially observable domains, where run-time information can be gathered by observation actions. SGP is based on the GRAPHPLAN approach, from which it inherits a remarkable strength in the analysis of parallelizable problems. SGP was the first conditional planner able to solve non-trivial problems. In [68], SGP is shown to outperform the existing planners obtained as extensions to classical planners (e.g., [50,56,67]). The approach underlying SGP is of enumerative nature: for each source of uncertainty, either in the possible initial states, or multiple action effects, a planning graph is built, and then the mutual relations among them are analyzed. The actual SGP system is limited to the case of deterministic domains, where uncertainty is only in the initial condition. Domains are expressed in a generalization of PDDL. Observations are expressed as effects of operators by means of the special Observes predicate. With respect to the problems in this paper, SGP is able to tackle strong (but not strong cyclic) planning problems, producing a conditional, non-iterative plan. SGP is not able to detect when the problem does not admit a solution.

7.2. Experimental evaluation setup

We performed an extensive experimental evaluation of our approach, by comparing MBP with the planners described in previous section. We analyze the behaviour of the different planners on parameterized problem classes, reporting the performance for different parameter values. In the comparison, we do not collect the total run time of

the systems. Rather, we focus on the search time, i.e., the time needed to solve the planning problem once the respective internal representation has been built, and exclude the preprocessing time. This excludes the time needed by QBFPLAN to generate the encodings, the time needed by GPT to generate the source code of its internal representation and to compile it, the time needed by SIMPLAN to compile the domain descriptions, and the time needed by MBP and UMOP to construct the symbolic automaton representation. There are several reasons for this choice. First, we try to highlight the qualitative behaviour of the planning algorithms, that can be spoiled if preprocessing times are included. Second, the preprocessing times is sometimes very hard to measure accurately, and may not be very meaningful. For instance, in the case of GPT, the preprocessing time includes the compilation and the linking to the executable solver of an automatically generated C++ file. Third, the preprocessing times may have different interpretation, depending on the system being run. For instance, for MBP, SIMPLAN and UMOP, the preprocessing time is related to the domain (the internal representation resulting from the preprocessing can be reused for several planning problems), while for GPT and QBFPLAN the overhead is problem specific, and should be taken into account for each problem.

For MBP, unless otherwise specified, all the tests were run with a fixed variable ordering, where actions variables precede state variables, and current and next state variables are interleaved. The ordering within action variables and within state variables depends on the order of variable declarations in the input file. Moreover, dynamic variable reordering was disabled. For the other planners, we tried to optimize the execution parameters in order to obtain the best performance we could.

The testing platform used is an Intel 300 MHz Pentium-II, 512 MB RAM, running Linux. CPU time was limited to 7200 seconds (two hours) for each test, while memory was limited to 500 MB. In the following, times are reported in seconds, and we write "T.O." or "M.O." for a test that did not complete within the above time and memory limits, respectively. In the experimental evaluation, in addition to the search time, we explicitly plot the preprocessing time for MBP, as it appears to be a significant factor. In the tables, we use Str and StrCyc for the search times of the strong and strong cyclic algorithms, and Prep as preprocessing time for MBP. Moreover, we write $P(i)$ for the ith instance of the problem class P.

7.3. Problems and results

In the comparison, we used problems selected from the distributions of the different competitor planners, for their characteristics of scalability and difficulty. This gave us a wide spectrum of tests, and allowed to broaden the analysis of the performance of MBP. Ideally, all the tests should have been encoded and run on all the systems. In practice, domains from one system may be not expressible or hard to apply to other systems, in particular given that there is no accepted standard for describing nondeterministic domains and problems. We compared MBP with the planning system contributing the domain. We also tried to maximize the performance of the competitor planners by using the original encodings provided in their distributions. Often the way a problem is encoded can be a significant factor in the performance of a planner.

7.3.1. Omelette

We first consider the classical OMELETTE(i) problem [47]. The goal is to have i good eggs and no bad ones in one of two bowls of capacity i. There is an unlimited number of eggs, each of which can be unpredictably good or bad. The eggs can be grabbed and broken into a bowl. The content of a bowl can be discarded, or poured to the other bowl. Breaking a rotten egg in a bowl has the effect of spoiling the bowl. A bowl can always be cleaned by discarding its content. In the problem as formulated in the distribution of GPT, an explicit sensing action is required to discover whether the content of the bowl is spoiled. The problem is modeled in MBP as a fully observable problem, without the need of an explicit sensing action, by considering that the result of the action (e.g., the status of the bowl) becomes observable as soon as the action has been executed. The results are reported in Table 1. The problem does not admit a strong solution, but only a strong cyclic solution. Column Str reports the time required by the strong algorithm to discover that the problem admits no strong solution. MBP is able to perform this task quite efficiently (e.g., less than 1 second for the OMELETTE(250)). The strong cyclic algorithm finds a solution, but the computation time is much higher. On the same set of problems, we ran the MDP module of GPT. The search is based on the repeated generation of learning trials (LTs), starting from randomly selected initial state. The policy tends to improve as the number of learning trials grows. However, GPT can not guarantee that the returned policies do not implement cyclic courses of action. In Table 1 we report time needed to perform 1000, 10000 and 100000 learning trials, and the success rate (SR) of the corresponding policy (computed on a basis of 1000, 10000 and 100000 control trials). As shown in [11], the qualitative pattern is that the controller tends to improve with a growing number of learning trials. However, GPT appears to suffer from the enumerative nature of the algorithm, and for increasing problem size the learning time tends to grow while the quality of the controller degrades. In the case of 50 omelettes, the memory is exhausted during the 10000 learning trials.

In order to stress the MBP algorithms for strong planning, we consider the OMELETTE-B(i,r) variation, where the number of rotten eggs is bounded to r. For such problem, a

Table 1
Results for the OMELETTE problems

OMELETTE(i)	MBP			GPT					
	Prep	Str	StrCyc	1k LT	SR	10k LT	SR	100k LT	SR
OMELETTE(2)	0.030	0.000	0.010	0.525	0.994	4.752	0.9994	50.664	0.99994
OMELETTE(4)	0.020	0.000	0.020	1.642	0.978	10.242	0.9978	99.903	0.99977
OMELETTE(6)	0.030	0.000	0.040	5.219	0.877	19.745	0.9880	168.729	0.99878
OMELETTE(8)	0.030	0.010	0.040	12.356	0.537	38.946	0.9413	249.133	0.99414
OMELETTE(10)	0.030	0.010	0.060	18.737	0.227	75.750	0.8093	370.196	0.98012
OMELETTE(12)	0.020	0.010	0.090	23.761	0.038	138.107	0.5255	534.824	0.94887
OMELETTE(14)	0.040	0.010	0.110	27.758	0.003	196.810	0.2323	756.499	0.88869
OMELETTE(16)	0.040	0.010	0.140	31.539	0.003	237.688	0.1073	1055.966	0.78541
OMELETTE(18)	0.040	0.010	0.180	37.045	0.000	268.552	0.0376	1480.801	0.62727
OMELETTE(20)	0.060	0.020	0.220	39.324	0.000	291.644	0.0052	2038.822	0.40691
OMELETTE(50)	0.300	0.050	1.480	73.471	0.000	M.O.			
OMELETTE(100)	1.930	0.130	16.350						
OMELETTE(250)	26.440	0.590	178.690						

Table 2
Results for the bounded OMELETTE problems

OMELETTE-B(20, r)	MBP		
	Prep	Str	StrCyc
OMELETTE-B(20, 1)	0.050	0.310	0.210
OMELETTE-B(20, 2)	0.060	0.470	0.210
OMELETTE-B(20, 3)	0.060	0.630	0.260
OMELETTE-B(20, 4)	0.060	0.840	0.220
OMELETTE-B(20, 5)	0.060	1.000	0.210
OMELETTE-B(20, 6)	0.060	1.140	0.200
OMELETTE-B(20, 7)	0.060	1.270	0.210
OMELETTE-B(20, 8)	0.050	1.410	0.220
OMELETTE-B(20, 9)	0.060	1.510	0.220
OMELETTE-B(20, 10)	0.070	1.710	0.220
OMELETTE-B(20, 20)	0.060	3.130	0.210
OMELETTE-B(20, 50)	0.070	7.720	0.210
OMELETTE-B(20, 100)	0.100	15.560	0.190

strong solution exists. Table 2 depicts the results for increasing number of rotten eggs when the size of the bowls is fixed to 20. The performance of the strong algorithm is sensitive to the number of rotten eggs. The strong cyclic algorithm, instead, appears to be quite stable and is almost insensitive on the number of rotten eggs. This can be explained by considering that the algorithm works by removing bad actions, and the introduction of possibly strong actions does not affect the symbolic operations.

7.3.2. Chain of rooms

We consider now the CHAIN(n) domain from the QBFPLAN distribution [57]. There is a sequence of rooms connected by two doors. For each pair of rooms, only one of the doors is open, while the other is closed. The goal is to go from room 0 to room n. Table 3 lists the results for the CHAIN-I (CHAIN Inertial) problems, where the status of the doors is fixed (i.e., they are inertial). In this case, there are 2^n initial states. In order to solve the ith instance of the problem, QBFPLAN generates and solves a sequence of QBF satisfiability problems, intuitively corresponding to searching for plans of increasing size, until a solution is found. We report the search time for the (largest) unsatisfiable instance (column $i-1$), and for the smallest satisfiable instance (column i). QBFPLAN performs remarkably well on this task: as reported in [57], QBFPLAN has a behaviour roughly linear in the number of packages, and manages to solve the problem up to 50 rooms. We remark, however, that a specific structure of plan is "hardwired" within the QBF encoding of the problem, that appears to be particularly suited for this domain. Without any constrains on the structure of the plan being built, the planning algorithms of MBP tackle the 2000-rooms instance in a few minutes. On the 3000-rooms instance, MBP exhausts the available memory while constructing the domain automaton. Our testing of QBFPLAN was limited to the problems of size 50, for which the encodings were already available. The generation of the encodings appears to be a severe bottleneck: for instance, in the case of CHAIN-I(50), the encoding generation for $i-1$ and i required, respectively, about 70 and 110 minutes. We also ran MBP on the CHAIN-NI problems, NI standing

Table 3
Results for the CHAIN-I problems

CHAIN-I(n)	MBP			QBFPLAN	
	Prep	Str	StrCyc	$i-1$	i
CHAIN-I(6)	0.020	0.010	0.000	0.02	0.04
CHAIN-I(10)	0.020	0.010	0.010	0.05	0.10
CHAIN-I(24)	0.050	0.010	0.020	0.89	2.35
CHAIN-I(30)	0.080	0.010	0.010	1.00	2.91
CHAIN-I(50)	0.120	0.030	0.040	0.84	13.97
CHAIN-I(100)	0.470	0.090	0.110		
CHAIN-I(200)	1.730	0.310	0.410		
CHAIN-I(300)	4.110	0.660	1.030		
CHAIN-I(400)	7.760	1.470	2.160		
CHAIN-I(500)	12.620	2.040	3.580		
CHAIN-I(600)	18.650	3.360	6.230		
CHAIN-I(700)	26.020	5.380	10.910		
CHAIN-I(800)	34.620	7.400	16.420		
CHAIN-I(900)	44.080	8.530	22.530		
CHAIN-I(1000)	57.290	10.400	29.890		
CHAIN-I(1500)	145.930	44.840	124.390		
CHAIN-I(2000)	362.330	75.670	244.250		
CHAIN-I(3000)	M.O.				

Table 4
Results for the non-inertial CHAIN problems

CHAIN-NI(n)	MBP		
	Prep	Str	StrCyc
CHAIN-NI(6)	0.020	0.000	0.000
CHAIN-NI(10)	0.020	0.000	0.000
CHAIN-NI(50)	0.050	0.030	0.030
CHAIN-NI(100)	0.120	0.080	0.100
CHAIN-NI(200)	0.380	0.270	0.400
CHAIN-NI(300)	0.780	0.600	1.000
CHAIN-NI(400)	1.430	1.250	2.130
CHAIN-NI(500)	2.290	1.590	3.180
CHAIN-NI(600)	3.380	2.960	6.760
CHAIN-NI(700)	4.910	4.550	11.140
CHAIN-NI(800)	7.110	6.100	17.110
CHAIN-NI(900)	9.810	7.270	25.600
CHAIN-NI(1000)	12.380	9.190	30.050
CHAIN-NI(1500)	43.590	41.960	147.670
CHAIN-NI(2000)	101.370	82.010	368.590
CHAIN-NI(3000)	445.600	373.780	1340.610
CHAIN-NI(4000)	1368.380	664.490	2373.710

for Non-Inertial doors, i.e., the variation of the CHAIN where the status of doors can change nondeterministically at each time instant. This example gives a clear instance of the non-enumerative nature of the symbolic approach. The branching factor of the search

space is extremely high: at each action execution correspond 2^n possible successor states. However, the symbolic representation allows to contain the resources needed to represent the corresponding automaton. Therefore, the automaton construction does not blow up, and it is possible to tackle configurations up to 4000 rooms with a reasonable search time. The problem admits a strong solution, and the strong algorithm is the most effective. The strong cyclic algorithm is unable to exploit the existence of a strong solution, and therefore has slightly worse performance. QBFPLAN was not run on this domain, because the encoding schema was not available from the QBFPLAN distribution. QBFPLAN is based on a low level input language (problem descriptions must be specified as ML code, which generates the QBF encodings), and writing new encodings turns out to be a very difficult and error-prone task.

7.3.3. Robot Delivery

The Robot Delivery domain, introduced in [42], describes an 8-room building, where a mobile robot can move from room to room, picking up and putting down packages. Nondeterminism in the domain is due to "producer" and "consumer" rooms: an object of a certain type can disappear if positioned in a consumer room, and can then reappear in one of the producer rooms. Furthermore, it is possible to have external processes that can change the position of the doors. We considered three different problems for this domain from the SIMPLAN distribution, with different initial conditions (e.g., robot and package positions), and with different goals. In the first, simple problem, ROBOT(p1), the robot has to reach a room contiguous to the room where it is initially positioned. A plan of length one exists. The second problem, ROBOT(p2), requires the robot to go to a given object, move it in a different location, and then reach a third location. The last problem, ROBOT(p3) is an extension of the ROBOT(p1), with one of the doors that can open and close nondeterministically. We codified two sets of problems. In the first set, only one package is assumed to be present in the domain. In the second set, the domain contains five objects (although only one is relevant for the problems being considered). The results of the comparison are depicted in Table 5. SIMPLAN appears to be very sensitive to nondeterminism. For instance, the difference between p1 and p3 is only in the nondeterministic behaviour of a door that the robot needs not to traverse. SIMPLAN relies on the use of domain specific, user defined control rules to prune the search space. The reported run times, obtained with and without control rules, show that the efficiency of SIMPLAN is very dependent on the quality of the control rule. When a solution does not exist, as in the case of p3, the search space must be exhausted. SIMPLAN, being based on explicit-state model checking techniques, appears to suffer when it has to handle large state spaces. MBP appears to be more stable with respect to nondeterminism. However, the introduction of additional irrelevant packages has an impact both on preprocessing and search.

7.3.4. King Hunter

The "King Hunter" problem, from the UMOP distribution, is a generalization of the Hunter-Prey problem, first presented in [44]. In the King Hunter problem instance KH(s,h), h hunters have to reach a prey moving in a square of side s. At each time, the hunters and the prey can take a "king move" or stay at the same location. Initially, the hunters and the

Table 5
Results for the ROBOT Delivery problems

	MBP			SIMPLAN	
	Prep	Str	StrCyc	CtrlRule	w/o CtrlRule
ROBOT(p1)	0.450	0.010	0.100	0.01	0.00
ROBOT(p2)	0.410	0.080	0.250	0.23	0.41
ROBOT(p3)	0.450	0.010	0.070	T.O.	T.O.
ROBOT(p1+)	150.870	0.050	12.210	0.01	0.02
ROBOT(p2+)	150.760	11.730	15.580	0.35	T.O.
ROBOT(p3+)	235.330	0.060	21.290	T.O.	T.O.

Table 6
Results for the King Hunter and Prey problems

KH(p,h)	MBP			MBP (CP, DR)			UMOP		
	Prep	Str	StrCyc	Prep	Str	StrCyc	Prep	Str	StrCyc
KH(4,1)	0.030	0.010	0.000	0.010	0.000	0.000	0.05	0.00	0.00
KH(8,1)	0.030	0.010	0.070	0.010	0.010	0.180	0.06	0.00	0.04
KH(16,1)	0.040	0.030	0.790	0.030	0.140	0.420	0.06	0.05	1.18
KH(32,1)	0.050	0.160	11.180	0.280	0.250	1.420	0.08	0.61	24.83
KH(4,2)	0.030	0.040	0.070	0.030	0.040	0.170	0.08	0.04	0.19
KH(8,2)	0.040	0.230	2.570	0.240	0.000	0.500	0.12	7.04	41.25
KH(16,2)	0.070	1.480	53.160	0.380	0.010	7.230	3.86	27.05	222.09
KH(32,2)	0.160	13.680	1545.820	1.000	0.030	1968.260	406.63	1853.66	5695.77
KH(4,3)	0.080	0.520	1.040	0.170	0.010	0.190	0.15	13.60	66.56
KH(8,3)	0.140	5.810	46.480	0.250	0.010	5.910	5.01	70.90	316.12
KH(16,3)	0.290	39.870	2679.110	1.040	0.020	1883.540	5066.82	1202.00	T.O.
KH(32,3)	0.470	188.150	M.O.	3.280	2.440	T.O.	T.O.		
KH(4,4)	0.590	335.950	12.280	0.210	248.130	0.290	0.76	1013.44	1112.6
KH(8,4)	1.210	M.O.	1064.370	0.650	T.O.	6.230	514.57	2890.95	T.O.
KH(16,4)	2.150	M.O.	M.O.	1.880	T.O.	T.O.	T.O.		
KH(32,4)	3.980	M.O.	M.O.	7.780	T.O.	T.O.			

prey can be at any location. The goal is for one hunter to end up in the same location of the prey. The results of the comparison of MBP with UMOP are reported in Table 6. UMOP was run with its default options, namely conjunctive partitioning and the dynamic BDD variable reordering active. The results for MBP reported in the first column in Table 6 corresponds to the execution of MBP with the settings used throughout the evaluation (i.e., monolithic transition relation and the dynamic variable reordering disabled). The mid column MBP (CP, DR) corresponds to running MBP with conjunctive partitioning enabled (with threshold set to 10000 BDD nodes) and with the dynamic BDD variable reordering active.

MBP appears to be more efficient than UMOP in model construction, even when using the monolithic representation of the transition relation and no dynamic BDD variable reordering. This is quite surprising, given that the automaton construction is one of the

strengths of UMOP [40]. Possible explanations are the following. First, MBP inherits from NuSMV a very efficient BDD package. Second, with the NuSMV language it is possible to express a higher degree of parallelism than in UMOP, where each agent can execute only one action at a time.

The problem admits strong cyclic solutions but no strong solutions. The search algorithms of MBP appear to be more efficient than the ones in UMOP, even in the monolithic case. When advanced model checking techniques are used within MBP, the preprocessing time slightly increase because of the BDD dynamic variable reordering, but the search time reduces. It is interesting to notice that the instances with four hunters, though conceptually simple, are very hard for both systems. This is an instance of the fact that BDDs can blow up when representing certain classes of Boolean functions. On the KH(8,4) instance, UMOP manages to complete the search for a strong solution, probably thanks to the dynamic reordering, while MBP runs out of time. On larger problems instances, UMOP is unable to complete the automaton construction. In MBP, the size of the BDDs during the search grows out of memory in the case of monolithic representation, while search time grows beyond the limit with dynamic variable reordering and conjunctive partitioning active.

In SGP, uncertainty is limited to the initial condition and nondeterministic actions are not expressible. Therefore we ran SGP on a simplified version of the King-Hunter domain, called HUNTER-S(p,i), where there is only one dimension for movement. Possible positions are range from 0 to $p-1$, and there is no prey. The hunter has the simple goal of reaching position 0 from any of i initial positions. For each position there is an observation action, `check-hunter-`p, that allows to detect if the hunter is in position p. We considered different problems, with different number of initial states (parameter i). The results are the reported in Table 7. When SGP times out, we report the explored level within the computation time. SGP is designed for partial observability, and therefore the

Table 7
Results of SGP on the simplified Hunter problems

HUNTER-S(p,i)	SGP	
	Levels	Time
HUNTER-S(4,1)	2	0.010
HUNTER-S(4,2)	2	0.020
HUNTER-S(4,3)	3	0.250
HUNTER-S(4,4)	3	9.880
HUNTER-S(5,1)	1	0.000
HUNTER-S(5,2)	3	0.060
HUNTER-S(5,3)	3	0.550
HUNTER-S(5,4)	3	206.390
HUNTER-S(5,5)	3	T.O.
HUNTER-S(6,1)	2	0.020
HUNTER-S(6,2)	4	0.100
HUNTER-S(6,3)	4	9.120
HUNTER-S(6,4)	4	6118.310
HUNTER-S(6,5)	3	T.O.

results on problems under the hypothesis of full observability must be carefully interpreted. The degrade in performance, however, appears to be related to the enumerative nature of the search algorithms, that expand a planning graph for each initial situation. In particular, limiting factors appear to be the number of initial states and the number of levels that need to be expanded.

7.4. Analysis of the results

Care must be taken when drawing conclusions from the results of the experimental comparison. First, the systems solve different problems, formulated in different languages. Furthermore, the actual problem encoding can have an impact on the performance, although we tried to be as fair as possible. Given these premises, MBP appears to behave effectively on a wide class of problems taken from the distributions of the other systems. Some of the strengths of MBP are the ability to describe general nondeterministic domains, to distinguish between strong and strong cyclic solutions, and to determine when a problem admits no solution. The use of BDDs provides effective ways of dealing with irrelevant information, and of avoiding the explicit enumeration of the state space. The main weakness of MBP is also a consequence of the use of BDDs, that can suffer from an explosion in space when representing certain classes of Boolean functions. A blow up in space can occur even for problems with a limited number of state variables (see for instance the King Hunter problems with four hunters).

MBP appears to perform better than UMOP. One reason for this behaviour appears to be that the input language of NuSMV allows to express a higher level of parallelism than NADL, and this makes it possible to the BDD-package to carry out its computations more efficiently. Furthermore, MBP relies on the advanced techniques provided by NuSMV for the manipulation of FSMs, that have been highly optimized. The other systems, besides the limitations in expressiveness, tend to have different forms of bottlenecks. GPT appears to require high amounts of memory and computation time to reach convergence. The performance of SIMPLAN is subject to the availability of a suitable, domain dependent control function. QBFPLAN shares with MBP a symbolic approach and performs remarkably well on the analyzed example. While MBP is based on a domain independent algorithm, the efficiency of QBFPLAN appears to depend on the specification of the plan structure in the encoding (in [57] a similar sensitivity is reported). SGP tends to suffer from the enumeration of the initial states. Notice however that the comparison of MBP with the observation-based approach of SGP is rather unfair. While in MBP the assumption of full observability is built-in and explicitly exploited, for SGP the number of observations is actually a factor in the branching rate of the search space.

8. Related work

Our work has several similarities with the work in planning based on Markov Decision Processes (MDP) (see [13] for an extensive survey). MDP planning algorithms deal with nondeterministic domains, and generate policies that are very much like state-action tables. There are however substantial differences, both conceptual and practical. From the

conceptual point of view, the MDP framework is richer than ours. It allows for expressing and dealing with information about costs and probabilities of action transitions and with rewards associated to states. In MDP, problem instances can thus be expressed in more detail. Planning algorithms (like value and policy iteration) can provide solutions that optimize a utility function defined on the basis of costs/rewards. They iteratively compute expected values and always return an optimal policy. MDP planning algorithms can return optimal policies that correspond to either weak, or strong, or strong cyclic solutions. Whether the solution is weak, strong, or strong cyclic depends on the probability and cost distribution. In our approach, we have three different algorithms for weak, strong, and strong cyclic solutions which are guaranteed to find a solution in the desired class. We can force the planner to find a solution in one of the different three classes of solutions. On the one hand, the MDP approach provides the ability to find solutions that have detailed requirements on costs and rewards. On the other hand, in some applications, like safety critical domains, it is rather important to distinguish explicitly between different solutions, e.g., among plans that are guaranteed to terminate (strong solutions) or not (strong cyclic solutions). A further conceptual difference is the fact that costs/rewards provide the ability to represent goals that are more expressive than the reachability goals discussed in this paper. Our framework is extended to deal with temporally extended goals [52,53], where goals corresponding to weak, strong, and strong cyclic solutions are formalized as CTL formulae [33], and a general planning algorithm is devised. From the practical point of view, the expressiveness of the MDP approach is more difficult to be managed in the case of large state spaces. Our approach allows us to fully exploit the BDD-based symbolic model checking techniques to tackle problems of significant size. As the experimental results show, MBP outperforms GPT, the available planner based on MDP, since an enumerative approach in the case of large state spaces and/or high uncertainty can hardly scale up. A lot of work has been done to tackle the state-explosion problem in MDP, like for instance techniques based on abstraction, techniques relying on reachability analysis and decomposition (see [13] for a review). The SPUDD planner [39] makes use of Algebraic Decision Diagrams, data structures similar to BDDs, to do MDP-based planning. In SPUDD, decision diagrams are used to represent much more detailed information than in MBP (e.g., the probabilities associated to transitions). This partly reduces the main practical advantages of decision diagrams as they are used in MBP.

Several works in planning in nondeterministic domains (e.g., [50,56,67,68]) do not address the problem of dealing with infinite paths and of generating iterative trial-and-error strategies. SIMPLAN [42] can deal with temporally extended goals but cannot generate strong cyclic solutions.

Our work is based on the hypothesis known as "full observability" (see, e.g., [11,42]), where the generated plans are executed by a reactive controller, that repeatedly, at each step, senses the world, determines the current state, selects an appropriate action and executes it. Different approaches consider models that distinguish on how information is assumed to be available to the controller at run-time, e.g., just some variables can be observed in different situations. Under this hypothesis, known as "partial observability", planning in nondeterministic domain is addressed by different techniques, see, e.g., [11,13,45,51,56, 57,59,63,68]. A limit case of planning under partial observability is conformant planning, where the assumption is that no information is available at run time, see, e.g., [11,61].

The idea of *Planning via* (*Symbolic*) *Model Checking* has been first introduced in [19] (see also [35] for an introduction). Some of the ideas presented in this paper have been presented in preliminary form in [23,24,29]. In this paper, we make the following contributions. We provide a formal characterization of the different notions, give a new, compositional presentation of the planning algorithms, prove their formal properties, and carry out an extensive experimental evaluation of the approach. Symbolic model checking techniques has also been applied to deal with conformant planning [5,8,21,22], with partial observability [7,9], and with temporally extended goals [28,52,53].

Other planners are based on symbolic model checking techniques. The work in [40] exploits the ideas presented in [23,24] as a starting point for their work on the UMOP planner (see the experimental comparison in Section 7). Jensen et al. [41] extend the framework to deal with adversarial planning. Traverso et al. [64] report about BDD-based planners for classical planning in deterministic domains. Among them, MIPS [32] showed remarkable results in the AIPS'00 planning competition for deterministic planning domains. BDD-based planners like MIPS are specialized to deal with deterministic domains, and are therefore more efficient than MBP on classical planning domains. One of the reasons for this is the use of advanced mechanisms to encode PDDL planning problems into BDDs, see, e.g., [31]. An interesting open research issue is whether these encoding techniques can be lifted to the case of nondeterministic domains. Other approaches are based on different model checking techniques. SIMPLAN [42] implements an approach to planning in nondeterministic domains based on explicit-state model checking. Bacchus and Kabanza [3] use explicit-state model checking to embed control strategies expressed in LTL in a deterministic planner based on forward search. [30] presents an automata based approach to formalize planning in deterministic domains. The work in [36–38] presents a method where model checking with timed automata is used to verify that generated plans meet timing constraints.

In our approach, we deal with uncertainty at planning time. Methods that interleave planning and execution [45] can be considered alternative (and orthogonal) approaches to the problem of planning off-line with large state spaces. On one side, they open up the possibility to deal with larger state spaces. On the other side, these methods cannot guarantee to find a solution, unless assumptions are made about the domain. An interesting open research issue is whether interleaving of planning and execution can be applied to our framework, and whether this can gain some advantages with respect to current approaches.

Our work shares some ideas with work on the automata-based synthesis of controllers (see, e.g., [2,46,48,55,65]). However, most of the work in this area focuses on the theoretical foundations, without providing practical implementations. Moreover, it is based on rather different technical assumptions on actions and on the interaction with the environment.

9. Concluding remarks and future work

The work presented in this paper provides both conceptual and practical contributions to the open problem of planning in nondeterministic domains. First, we provide a formal account for the problem of classifying solutions of different strength in nondeterministic

domains. We identify *weak* solutions, that may achieve the goal but are not guaranteed to, *strong* solutions, that are guaranteed to achieve the goal in spite of nondeterminism, and *strong cyclic* solutions, whose executions always have a possibility of terminating and, when they do, they are guaranteed to achieve the goal. Second, we present a family of algorithms for solving the above mentioned problems. The algorithms are proven to terminate, and to be sound and complete. Third, we implement these algorithms in the MBP planner, by means of symbolic model checking techniques based on BDDs, and we show that the approach is practical. The experimental results show that MBP is able to tackle significant problems, and compares positively with the existing planners that deal with nondeterminism.

Future research objectives are the following. As far as MBP is concerned, we plan to push its expressiveness by means of an extended input language, and to improve its efficiency with a more aggressive use of domain preprocessing techniques and advanced symbolic model checking techniques. Our approach will be extended to the general case of planning under partial observability for temporally extended goals, along the lines described in [6]. Finally, we will investigate the integration, within our framework, of techniques based on heuristic search and POMDP planning. The idea is to use the "qualitative" approach of MBP to drive the MDP approach: the former can guarantee that some returned plans satisfy some properties (e.g., they are strong solutions), and the latter could then be used to select, among all the returned plans, those that are optimal w.r.t. rewards and probability distributions.

References

[1] P.A. Abdulla, P. Bjesse, N. Eén, Symbolic reachability analysis based on SAT-solvers, in: S. Graf, M. Schwartzbach (Eds.), Proceedings of the Sixth Conference Tools and Algorithms for the Construction and Analysis of Systems, Berlin, Germany, in: Lecture Notes in Computer Science, Vol. 1785, Springer, Berlin, 2000, pp. 411–425.

[2] E. Asarin, O. Maler, A. Pnueli, Symbolic controller synthesis for discrete and timed systems, in: P. Antsaklis, W. Kohn, A. Nerode, S. Sastry (Eds.), Hybrid Systems II, in: Lecture Notes in Computer Science, Vol. 999, Springer, Berlin, 1995, pp. 1–20.

[3] F. Bacchus, F. Kabanza, Using temporal logic to express search control knowledge for planning, Artificial Intelligence 16 (1–2) (2000) 123–191.

[4] A.G. Barto, S.J. Bradtke, S.P. Singh, Learning to act using real-time dynamic programming, Artificial Intelligence 72 (1–2) (1995) 81–138.

[5] P. Bertoli, A. Cimatti, Improving heuristics for planning as search in belief space, in: Proceedings of Sixth International Conference on AI Planning and Scheduling (AIPS'02), Toulouse, France, 2002, pp. 83–92.

[6] P. Bertoli, A. Cimatti, M. Pistore, P. Traverso, Plan validation for extended goals under partial observability (preliminary report), in: Proceedings of the AIPS 2002 Workshop on Planning via Model Checking, Toulouse, France, 2002, pp. 14–22.

[7] P. Bertoli, A. Cimatti, M. Roveri, Conditional planning under partial observability as heuristic-symbolic search in belief space, in: Proceedings of the Sixth European Conference on Planning (ECP'01), 2001, pp. 379–384.

[8] P. Bertoli, A. Cimatti, M. Roveri, Heuristic search + symbolic model checking = efficient conformant planning, in: B. Nebel (Ed.), Proc. IJCAI-2001, Seattle, WA, Morgan Kaufmann, San Mateo, CA, 2001, pp. 467–472.

[9] P. Bertoli, A. Cimatti, M. Roveri, P. Traverso, Planning in nondeterministic domains under partial observability via symbolic model checking, in: B. Nebel (Ed.), Proc. IJCAI-2001, Seattle, WA, Morgan Kaufmann, San Mateo, CA, 2001, pp. 473–478.

[10] A. Biere, A. Cimatti, E.M. Clarke, Y. Zhu, Symbolic model checking without BDDs, in: R. Cleaveland (Ed.), Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99), in: Lecture Notes in Computer Science, Vol. 1579, Springer, Berlin, 1999, pp. 193–207.

[11] B. Bonet, H. Geffner, Planning with incomplete information as heuristic search in belief space, in: S. Chien, S. Kambhampati, C. Knoblock (Eds.), Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling, Seattle, WA, AAAI Press, 2000, pp. 52–61.

[12] C. Boutilier, T. Dean, S. Hanks, Planning under uncertainty: Structural assumptions and computational leverage, in: M. Ghallab, A. Milani (Eds.), New Directions in AI Planning, IOS Press, Amsterdam, 1996, pp. 157–172.

[13] C. Boutilier, T. Dean, S. Hanks, Decision-theoretic planning: Structural assumptions and computational leverage, J. Artificial Intelligence Res. 11 (1999) 1–94.

[14] K.S. Brace, R.L. Rudell, R.E. Bryant, Efficient implementation of a BDD package, in: Proceedings of the 27th ACM/IEEE Design Automation Conference, ACM/IEEE, IEEE Computer Society Press, Orlando, FL, 1990, pp. 40–45.

[15] R.E. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Trans. Comput. C 35 (8) (1986) 677–691.

[16] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang, Symbolic model checking: 10^{20} states and beyond, Inform. and Comput. 98 (2) (1992) 142–170.

[17] A. Cimatti, E.M. Clarke, F. Giunchiglia, M. Roveri, NUSMV: A new symbolic model verifier, in: N. Halbwachs, D. Peled (Eds.), Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99), Trento, Italy, in: Lecture Notes in Computer Science, Vol. 1633, Springer, Berlin, 1999, pp. 495–499.

[18] A. Cimatti, E.M. Clarke, F. Giunchiglia, M. Roveri, NUSMV: A new symbolic model checker, Internat. J. Software Tools for Technology Transfer 2 (4) (2000).

[19] A. Cimatti, E. Giunchiglia, F. Giunchiglia, P. Traverso, Planning via model checking: A decision procedure for $\mathcal{AR}$, in: S. Steel, R. Alami (Eds.), Proceeding of the Fourth European Conference on Planning, Toulouse, France, in: Lecture Notes in Artificial Intelligence, Vol. 1348, Springer, Berlin, 1997, pp. 130–142.

[20] A. Cimatti, E. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, Integrating BDD-based and SAT-based symbolic model checking, in: A. Armando (Ed.), Proceedings of the Fourth International Workshop on Frontiers of Combining Systems, in: Lecture Notes in Artificial Intelligence, Vol. 2309, Springer, Berlin, 2002, pp. 49–56.

[21] A. Cimatti, M. Roveri, Conformant planning via model checking, in: S. Biundo (Ed.), Proceeding of the Fifth European Conference on Planning, Durham, UK, in: Lecture Notes in Artificial Intelligence, Vol. 1809, Springer, Berlin, 1999, pp. 21–34.

[22] A. Cimatti, M. Roveri, Conformant planning via symbolic model checking, J. Artificial Intelligence Res. 13 (2000) 305–338.

[23] A. Cimatti, M. Roveri, P. Traverso, Automatic OBDD-based generation of universal plans in non-deterministic domains, in: Proc. AAAI-98, Madison, WI, AAAI Press, 1998, pp. 875–881.

[24] A. Cimatti, M. Roveri, P. Traverso, Strong planning in non-deterministic domains via model checking, in: Proceeding of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98), Carnegie Mellon University, Pittsburgh, AAAI Press, 1998, pp. 36–43.

[25] E.M. Clarke, E.A. Emerson, Synthesis of synchronization skeletons for branching time temporal logic, in: Logic of Programs: Workshop, in: Lecture Notes in Computer Science, Vol. 131, Springer, Berlin, 1981, pp. 52–71.

[26] E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, MIT Press, Cambridge, MA, 1999.

[27] E.M. Clarke, J.M. Wing, Formal methods: State of the art and future directions, ACM Comput. Surveys 28 (4) (1996) 626–643.

[28] U. Dal Lago, M. Pistore, P. Traverso, Planning with a language for extended goals, in: Proc. AAAI-02, Edmonton, AB, AAAI Press/MIT Press, 2002, pp. 447–454.

[29] M. Daniele, P. Traverso, M.Y. Vardi, Strong cyclic planning revisited, in: S. Biundo (Ed.), Proceeding of the Fifth European Conference on Planning, Durham, UK, in: Lecture Notes in Artificial Intelligence, Vol. 1809, Springer, Berlin, 1999, pp. 35–48.

[30] G. De Giacomo, M.Y. Vardi, Automata-theoretic approach to planning for temporally extended goals, in: S. Biundo (Ed.), Proceeding of the Fifth European Conference on Planning, Durham, UK, in: Lecture Notes in Artificial Intelligence, Vol. 1809, Springer, Berlin, 1999, pp. 226–238.

[31] S. Edelkamp, M. Helmert, Exhibiting knowledge in planning problems to minimize state encoding length, in: S. Biundo, M. Fox (Eds.), Proceedings of the Fifth European Conference on Planning (ECP'99), in: Lecture Notes in Artificial Intelligence, Vol. 1809, Springer, Berlin, 1999, pp. 135–147.

[32] S. Edelkamp, M. Helmert, On the implementation of mips, in: AIPS-Workshop on Model-Theoretic Approaches to Planning, 2000, pp. 18–25.

[33] E.A. Emerson, Temporal and modal logic, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics, Elsevier, Amsterdam, 1990, Chapter 16, pp. 995–1072.

[34] E. Giunchiglia, G.N. Kartha, V. Lifschitz, Representing action: Indeterminacy and ramifications, Artificial Intelligence 95 (2) (1997) 409–438.

[35] F. Giunchiglia, P. Traverso, Planning as model checking, in: S. Biundo (Ed.), Proceeding of the Fifth European Conference on Planning, Durham, UK, in: Lecture Notes in Artificial Intelligence, Vol. 1809, Springer, Berlin, 1999, pp. 1–20.

[36] R. Goldman, M. Pelican, D. Musliner, Hard Real-time Mode Logic Synthesis for Hybrid Control: A CIRCA-based approach, Working notes of the 1999 AAAI Spring Symposium on Hybrid Control, 1999.

[37] R.P. Goldman, D.J. Musliner, K.D. Krebsbach, M.S. Boddy, Dynamic abstraction planning, in: Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference (AAAI-97), (IAAI-97), AAAI Press, 1997, pp. 680–686.

[38] R.P. Goldman, D.J. Musliner, M.J. Pelican, Using model checking to plan hard real-time controllers, in: Proceeding of the AIPS2k Workshop on Model-Theoretic Approaches to Planning, Breckeridge, CO, 2000, pp. 2–9.

[39] J. Hoey, R. St-Aubin, A. Hu, C. Boutilier, SPUDD: Stochastic planning using decision diagrams, in: K.B. Laskey, H. Prade (Eds.), Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence (UAI-99), Morgan Kaufmann, San Francisco, CA, 1999, pp. 279–288.

[40] R. Jensen, M. Veloso, OBDD-based universal planning for synchronized agents in non-deterministic domains, J. Artificial Intelligence Res. 13 (2000) 189–226.

[41] R.M. Jensen, M.M. Veloso, M.H. Bowling, OBDD-based optimistic and strong cyclic adversarial planning, in: Proceedings of the Sixth European Conference on Planning (ECP'01), 2001, pp. 265–276.

[42] F. Kabanza, M. Barbeau, R. St-Denis, Planning control rules for reactive agents, Artificial Intelligence 95 (1) (1997) 67–113.

[43] H. Kautz, D. McAllester, B. Selman, Encoding plans in propositional logic, in: L.C. Aiello, J. Doyle, S. Shapiro (Eds.), KR'96: Principles of Knowledge Representation and Reasoning, Morgan Kaufmann, San Francisco, CA, 1996, pp. 374–384.

[44] S. Koenig, R. Simmons, Real-time search in non-deterministic domains, in: Proc. IJCAI-95, Montreal, Quebec, Morgan Kaufmann, San Mateo, CA, 1995, pp. 1660–1667.

[45] S. Koenig, R. Simmons, Solving robot navigation problems with initial pose uncertainty using real-time heuristic search, in: R.G. Simmons, M. Veloso, S. Smith (Eds.), Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-98), AAAI Press, 1998, pp. 145–153.

[46] O. Kupferman, M.Y. Vardi, Synthesis with incomplete information, in: Proc. of 2nd International Conference on Temporal Logic, 1997, pp. 91–106.

[47] H.J. Levesque, What is planning in the presence of sensing?, in: Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference, AAAI Press/MIT Press, Menlo Park, CA, 1996, pp. 1139–1146.

[48] O. Maler, A. Pnueli, J. Sifakis, On the synthesis of discrete controllers, in: E.W. Mayr, C. Puech (Eds.), Proceedings of the Twelfth Annual Symposium on Theoretical Aspects of Computer Science (STACS-95), in: Lecture Notes in Computer Science, Vol. 900, Springer, Berlin, 1995, pp. 229–242.

[49] K. McMillan, Symbolic Model Checking, Kluwer Academic, Dordrecht, 1993.

[50] M. Peot, D. Smith, Conditional nonlinear planning, in: J. Hendler (Ed.), Proceedings of the First International Conference on AI Planning Systems, College Park, MD, Morgan Kaufmann, San Mateo, CA, 1992, pp. 189–197.
[51] R. Petrick, F. Bacchus, A knowledge-based approach to planning with incomplete information and sensing, in: Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS'02), 2002, pp. 37–46.
[52] M. Pistore, R. Bettin, P. Traverso, Symbolic techniques for planning with extended goals in non-deterministic domains, in: Proceedings of the Sixth European Conference on Planning (ECP'01), 2001, pp. 253–264.
[53] M. Pistore, P. Traverso, Planning as model checking for extended goals in non-deterministic domains, in: B. Nebel (Ed.), Proc. IJCAI-01, Seattle, WA, Morgan Kaufmann, San Mateo, CA, 2001, pp. 479–486.
[54] A. Pnueli, The temporal logic of programs, in: Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77), Providence, RI, IEEE Computer Society Press, 1977, pp. 46–57.
[55] A. Pnueli, R. Rosner, On the synthesis of an asynchronous reactive module, in: Proceedings of the Sixth International Colloquium on Automata, Languages and Programming, in: Lecture Notes in Computer Science, Vol. 372, Springer, Berlin, 1989, pp. 652–671.
[56] L. Pryor, G. Collins, Planning for contingency: A decision based approach, J. Artificial Intelligence Res. 4 (1996) 81–120.
[57] J. Rintanen, Constructing conditional plans by a theorem-prover, J. Artificial Intelligence Res. 10 (1999) 323–352.
[58] J. Rintanen, Improvements to the evaluation of quantified Boolean formulae, in: T. Dean (Ed.), Proc. IJCAI-99, Stockholm, Sweden, Morgan Kaufmann, San Mateo, CA, 1999, pp. 1192–1197.
[59] J. Rintanen, Backward plan construction for planning as search in belief space, in: Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS'02), 2002, pp. 93–102.
[60] M.J. Schoppers, Universal plans for reactive robots in unpredictable environments, in: Proc. IJCAI-87, Milan, Italy, 1987, pp. 1039–1046.
[61] D.E. Smith, D.S. Weld, Conformant graphplan, in: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98), AAAI Press, Menlo Park, CA, 1998, pp. 889–896.
[62] F. Somenzi, CUDD: CU Decision Diagram package—Release 2.1.2. Department of Electrical and Computer Engineering—University of Colorado at Boulder, 1997.
[63] C. Tovey, S. Koenig, Gridworlds as testbeds for planning with incomplete information, in: Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence, AAAI Press, 2000, pp. 819–824.
[64] P. Traverso, M. Veloso, F. Giunchiglia (Eds.), AIPS2000 Workshop on Model-Theoretic Approaches to Planning, Breckeridge, CO, 2000.
[65] M.Y. Vardi, An automata-theoretic approach to fair realizability and synthesis, in: P. Wolper (Ed.), Proceedings of the 7th International Conference On Computer Aided Verification, Liege, Belgium, in: Lecture Notes in Computer Science, Vol. 939, Springer, Berlin, 1995, pp. 267–278.
[66] M.Y. Vardi, Branching vs. linear time: Final showdown, in: T. Margaria, W. Yi (Eds.), Proceedings of the Seventh International Conference on Tools and Algorithms for the Construction of Systems (TACAS-2001), in: Lecture Notes in Computer Science, Vol. 2031, Springer, Berlin, 2001, pp. 1–22.
[67] D.H.D. Warren, Generating conditional plans and programs, in: Proceedings of the Summer Conference on Artificial Intelligence and Simulation of Behaviour (AISB-76), Edinburgh, Scotland, 1976, pp. 344–354.
[68] D.S. Weld, C.R. Anderson, D.E. Smith, Extending graphplan to handle uncertainty and sensing actions, in: Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98), AAAI Press, Menlo Park, CA, 1998, pp. 897–904.