Strong planning under partial observability

Piergiorgio Bertoli *, Alessandro Cimatti, Marco Roveri, Paolo Traverso

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

Received 4 April 2004; received in revised form 1 May 2005; accepted 10 January 2006

Abstract

Rarely planning domains are fully observable. For this reason, the ability to deal with partial observability is one of the most important challenges in planning. In this paper, we tackle the problem of strong planning under partial observability in nondeterministic domains: find a conditional plan that will result in a successful state, regardless of multiple initial states, nondeterministic action effects, and partial observability.

We make the following contributions. First, we formally define the problem of strong planning within a general framework for modeling partially observable planning domains. Second, we propose an effective planning algorithm, based on and-or search in the space of beliefs. We prove that our algorithm always terminates, and is correct and complete. In order to achieve additional effectiveness, we leverage on a symbolic, BDD-based representation for the domain, and propose several search strategies. We provide a thorough experimental evaluation of our approach, based on a wide selection of benchmarks. We compare the performance of the proposed search strategies, and identify a uniform winner that combines heuristic distance measures with mechanisms that reduce runtime uncertainty. Then, we compare our planner MBP with other state-of-the-art systems. MBP is able to outperform its competitor systems, often by orders of magnitude.

Keywords: Planning under partial observability; Planning in nondeterministic domains; Heuristic search in belief space; Symbolic model checking; Binary decision diagrams

1. Introduction

Very often, planning domains are partially observable and nondeterministic: at execution time, the state of the world can not be completely observed, and actions may have several possible outcomes. In this case, the status of the domain can not be uniquely determined. Planning for nondeterministic and partially observable domains is a significant, well known and difficult problem. It is a significant problem since several realistic applications require to deal with sources of nondeterminism, within the general case of partial observability. The special cases of full observability, where every state variable can be observed at every step, and null observability, where no observation is ever possible, only cover a limited set of realistic situations. Indeed, the problem has been extensively addressed in the literature. Different approaches include extensions to techniques for classical planning, see e.g., [73,74,88], Partially Observable Markov Decision Processes (POMDPs), see e.g., [14,15,22,32,60,69,70,83], logical frameworks [41,51],

* Corresponding author.

E-mail addresses: bertoli@itc.it (P. Bertoli), cimatti@itc.it (A. Cimatti), roveri@itc.it (M. Roveri), traverso@itc.it (P. Traverso).

0004-3702/S – see front matter © 2006 Elsevier B.V. All rights reserved.

doi:10.1016/j.artint.2006.01.004
planning based on Quantified Boolean Formulas (QBF) \[77,78\], and planning based on symbolic model checking [7,9,79]. The problem has been shown to be hard, both theoretically and experimentally. Compared to planning under full observability, planning under partial observability must deal with uncertainty about the state in which the actions will be executed. This makes the search space no longer the set of states of the domain, but its powerset, i.e. the space of “belief states” \[14,43\]. Compared to the case of null observability, called conformant planning, plans are no longer sequential, but conditional, in order to represent a conditional course depending on the observations performed at execution time.

In this paper, we address the problem of strong planning under partial observability, i.e., the problem of generating conditional plans that are guaranteed to achieve the goal in spite of the nondeterminism and partial observability of the domain. Strong planning can be formalized as a problem of search in the space of a (possibly cyclic) and-or graph induced by the domain \[9\], where each node in the graph corresponds to the set of possible states for the current situation, i.e., a belief state. Or-nodes correspond to alternative actions that can be applied to a belief state, while and-nodes correspond to observations that partition a belief state into subsets for all of which a solution must be found. We present a novel approach to the problem and provide evidence that the approach is well-founded and practical:

- We present a planning algorithm that performs heuristic search over the and-or graph. Differently from AO* \[66\], we cannot assume that the graph is acyclic, and differently from LAO* \[53\], we do not accept cyclic plans, which might not terminate. As such, our algorithm must generate conditional plans that are guaranteed to achieve the goal by ruling out possible cycles. This is achieved by keeping track of the part of the graph that has been explored so far in an effective way. We prove that the algorithm is correct and complete, i.e., it finds a solution if it exists and, if no solution exists, it terminates with failure.
- We show that the search in the and-or graph can be implemented by means of symbolic model checking techniques \[64\]. In particular, Binary Decision Diagrams (BDDs) \[25\] can be used to compactly represent and efficiently manipulate belief states.
- We show that our algorithm can easily embed different search strategies, and that BDD-based techniques can be fruitfully exploited to speed up the computation of heuristic values that drive those strategies.
- We implement the algorithm in the Model Based Planner (MBP) \[5\]. We provide an extensive experimental evaluation of our approach, considering various search strategies, and comparing our system with other planners. The experimental evaluation includes problems available from the distribution set of other planners. We show that MBP outperforms the other planners that can deal with nondeterministic domains under partial observability.

The paper is organized as follows. We provide a formal definition of partially observable planning domains, problems, and conditional plans (Section 2). We describe how planning under partial observability can be seen as searching through an and-or graph of belief states (Section 3). In Section 4, we define the planning algorithm, and prove its properties. In Section 5, we discuss how BDD-based, symbolic techniques can be effectively used in the search. We present the experimental evaluation in Section 7. Finally, we discuss related work in Section 8, while Section 9 draws some conclusions and discusses future research directions.

2. Domains, observations, plans and problems

Partially observable planning domains can be generally presented by means of two components: a nondeterministic state-transition system, describing the effect of actions, and an observation function that describes the available sensing over the domain.

2.1. Actions in nondeterministic domains

A nondeterministic state-transition system is defined in terms of its states, its actions, and of a transition function that describes how (the execution of) an action leads from one state to possibly many different states.

Definition 2.1 (State transition system). A nondeterministic state transition system is a tuple $\Sigma = (S, A, R)$, where $S$ is a finite set of states, $A$ is a finite set of actions, and $R : S \times A \rightarrow 2^S$ is the transition function. The transition function associates to each state $s \in S$ and to each action $a \in A$ the set $R(s, a) \subseteq S$ of next states.
An action $a$ is applicable in a state $s$ (denoted with $\text{applicable}(a, s)$) iff $|\mathcal{R}(s, a)| > 0$; it is deterministic (non-deterministic) in $s$ iff $|\mathcal{R}(s, a)| = 1$ ($|\mathcal{R}(s, a)| > 1$). If $a$ is applicable in $s$, then $\mathcal{R}(s, a)$ is the set of states that can be reached from $s$ by performing $a$.

Example 2.2. Fig. 1 shows a simple robot navigation domain and its corresponding state transition system. The robot can be in four positions, corresponding to the states of the domain: $\mathcal{S} = \{\text{NW, NE, SW, SE}\}$. It can move in the four directions, corresponding to actions $\mathcal{A} = \{\text{GoNorth, GoSouth, GoEast, GoWest}\}$. An action is applicable only if there is no wall in the direction of motion, e.g. $\mathcal{R}(\text{NW, GoSouth}) = \emptyset$. All the actions are deterministic (e.g., $\mathcal{R}(\text{SW, GoEast}) = \{\text{SE}\}$), except for action GoEast when on NW, where the robot may slip and end up either in NE or SE, i.e. $\mathcal{R}(\text{NW, GoEast}) = \{\text{NE, SE}\}$.

2.2. Observations

A domain may not be completely observable—different states may be impossible to distinguish. For instance, it may be possible for the robot to observe the presence or absence of walls in its neighboring locations, but it may not be possible to observe the absolute location. In order to model partial observability, we define a set of observations, which intuitively correspond to the possible readings conveyed by sensors at each time instant; an observation function defines the correspondence between the (hidden) states of the domain and the possible observations (of which, of course, only one will be nondeterministically chosen to hold by the domain, at each runtime step).

Definition 2.3 (Observation function). Let $\Sigma = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ be a nondeterministic state transition system. Let $\mathcal{O}$ be a finite set of observations. An observation function over $\mathcal{S}$ and $\mathcal{O}$ is a function $\mathcal{X} : \mathcal{S} \to 2^\mathcal{O} \setminus \emptyset$, which associates to each state $s$ the nonempty set of possible observations $\mathcal{X}(s) \subseteq \mathcal{O}$.

Our model of observation is inspired by the idea of automatic sensing [86], where information is conveyed by sensors without an explicit action being required, and each state is associated with at least one observation. In the following, $\mathcal{X}^-(o)$ denotes the set of states which are compatible with the observation $o$, i.e. $\mathcal{X}^-(o) = \{s : o \in \mathcal{X}(s)\}$. We also use $\mathcal{X}^-(\bar{o})$, called observation complement, to indicate that observation $o$ does not occur. We write $\mathcal{O}$ to denote the set of all observation complements, i.e. $\mathcal{O} = \{\bar{o} | o \in \mathcal{O}\}$. We denote with $\mathcal{X}^-(\bar{o})$ the set of states which are compatible with any observation other than $o$, i.e. $\mathcal{X}^-(\bar{o}) = \{s : \exists o' \in \mathcal{X}(s), o' \neq o\}$.

Despite its simplicity, our model of observation is rather general. The special cases of full observability is modeled by defining the observation function $\mathcal{X}$ to be a bijection between $\mathcal{O}$ and $\mathcal{S}$. In our robot domain, the observation function would associate to state SW an observation $\text{sw}$, to SE an observation $\text{se}$, and so on. The special case of null observability (which is the assumption for conformant planning) is captured when $\mathcal{O}$ is a singleton: since the same observation is associated to all the states, it conveys no information. With the following example, we show how we model the general situation where sensing does not give complete information on the current state of the domain. This happens when different states may result in the same observation, i.e. when $\mathcal{X}(s_1) \cap \mathcal{X}(s_2) \neq \emptyset$ for $s_1 \neq s_2$. 

![Fig. 1. A simple domain represented as a state transition system.](image-url)
Example 2.4. If we assume that the robot can only perceive the position of the walls neighboring its location, then it cannot distinguish between the two positions NW and SW. This situation can be modeled with a set of observations $\mathcal{O} = \{\text{nsw}, \text{ne}, \text{se}\}$, and an observation function $\mathcal{X}(\text{NW}) = \mathcal{X}(\text{SW}) = \{\text{nsw}\}$, $\mathcal{X}(\text{NE}) = \{\text{ne}\}$, and $\mathcal{X}(\text{SE}) = \{\text{se}\}$.

Our model also allows for noisy observations, i.e. the case where the same state can be associated with several different observations. Formally, we say that an observation $o$ is noisy iff for some $s \in \mathcal{X}^{-1}(o)$, $|\mathcal{X}(s)| > 1$.

Example 2.5. Let us consider the case where the robot sensors may sometimes not work, and return a meaningless value noinfo: $\mathcal{X}(\text{NW}) = \mathcal{X}(\text{SW}) = \{\text{nsw}, \text{noinfo}\}$, $\mathcal{X}(\text{NE}) = \{\text{ne}, \text{noinfo}\}$, and $\mathcal{X}(\text{SE}) = \{\text{se}, \text{noinfo}\}$. In this case, all the observations are noisy; e.g., $\mathcal{X}^{-1}(\text{nsw}) = \{\text{NW}\}$, and $|\mathcal{X}(\text{NW})| = 2$; this indicates that in state NW the observation nsw may or may not occur.

Finally, as shown in the following example, our framework is capable of modeling situations where the ability to observe is conditioned to the prior execution of “sensor-enabling” actions.

Example 2.6. Consider the case where the robot has to perform a WallScan sensing action in order to “set up” the sensors that allow it to observe the configuration of the surrounding walls; prior to executing such action, sensors return no information. This domain can be modeled as depicted in Fig. 2, by adding to the domain the new states NW0, SW0, SE0 and NE0, which represent states NW, SW, SE and NE prior to the execution of the WallScan action (depicted as a dashed arrow). The observation function is the same as the one defined in Example 2.4 for the original states, while it returns a noinfo value for the new states: the intuition is that if WallScan has not been just executed, the same observation noinfo is bound to occur, thus conveying no information.

A planning domain is composed by a nondeterministic state transition system, and an observation function over a finite set of observations.

Definition 2.7 (Planning domain). A planning domain $D$ is a tuple $(\Sigma, \mathcal{O}, \mathcal{X})$, where:

- $\Sigma = \langle S, A, R \rangle$ is a nondeterministic state transition system,
- $\mathcal{O}$ is a finite set of observations, and
- $\mathcal{X}$ is an observation function over $S$ and $\mathcal{O}$.

We remark that, in our framework, domains are described at a semantic level, independently of a concrete language that can be used to describe them: states and observations are atomic entities. In practice, however, we use a concrete language where variables can be used in order to express domains in a compact manner: the sets of states and observations are presented by means of state variables and observation variables, while the transition and observation functions are expressed as formulae over such variables.

![Fig. 2. The simple domain where sensors may be broken.](image-url)
2.3. Conditional plans

Differently from classical planning, we consider tree-structured plans with conditional courses of action, where different behaviors can be determined by different run-time observations.

**Definition 2.8 (Conditional plan).** The set of conditional plans $\Pi$ for a domain $D = (\Sigma, \mathcal{O}, \mathcal{X})$ is the minimal set such that:

- $\epsilon \in \Pi$;
- if $a \in \mathcal{A}$ and $\pi \in \Pi$, then $a \circ \pi \in \Pi$;
- if $o \in \mathcal{O}$, and $\pi_1, \pi_2 \in \Pi$, then $\mathbf{if} \ a \ \mathbf{then} \ \pi_1 \ \mathbf{else} \ \pi_2 \in \Pi$.

$\epsilon$ is the empty plan, representing the end of the execution. The plan $a \circ \pi$ is the sequential composition of the action $a$ with plan $\pi$. The plan $\mathbf{if} \ a \ \mathbf{then} \ \pi_1 \ \mathbf{else} \ \pi_2$ is a conditional plan, that branches on the occurrence of the observation $o$. In the following, we only consider finite conditional plans, and we use $\pi, \pi', \pi'', \pi_1, \pi_2, \ldots$ to denote them.

2.3.1. Syntactic properties of plans

We call an action-observation path (or simply a path) an element of $(\mathcal{A} \cup \mathcal{O} \cup \mathcal{O})^*$, i.e. a finite, possibly empty sequence composed of actions, observation, and observation complements.

**Definition 2.9 (Path of a plan).** An action-observation path $p$ is a path of a plan $\pi$ iff either

- $p = \epsilon$; or
- $\pi = a \circ \pi'$, $p = a \circ p'$, and $p'$ is a path of $\pi'$; or
- $\pi = \mathbf{if} \ a \ \mathbf{then} \ \pi_1 \ \mathbf{else} \ \pi_2$, and either (i) $p = o \circ p_1$ and $p_1$ is a path of $\pi_1$, or (ii) $p = \overline{o} \circ p_2$ and $p_2$ is a path of $\pi_2$.

The symbol $\epsilon$ denotes the empty path; we assume, as standard for strings, that $p \circ \epsilon = \epsilon \circ p = p$. We use $p, p', p'', p_1, p_2, \ldots$ to denote paths, and we write $\text{Paths}(\pi)$ for the set of paths in $\pi$. We notice that, if $p \in \text{Paths}(\pi)$, then $p' \in \text{Paths}(\pi)$ for every $p'$ prefix of $p$. Furthermore, $p \circ o \in \text{Paths}(\pi)$ iff $p \circ \overline{o} \in \text{Paths}(\pi)$. We say that $p \in \text{Paths}(\pi)$ is a maximal path of $\pi$ if $p$ is not a prefix of any other $p' \in \text{Paths}(\pi)$; we write $\text{MaxPaths}(\pi)$ for the set of maximal paths of $\pi$. If $p$ is a path and $P$ is a set of paths, we write $p \circ P$ for $\{p \circ p' \mid p' \in P\}$. Given two paths $p, p'$, we indicate with $p \leq p'$ ($p < p'$) the fact that $p$ is a (strict) prefix of $p'$.

Intuitively, a path $p$ of a plan $\pi$ identifies a (partial) course of action among the possible ones that are compatible with $\pi$, depending on the observations conveyed by the sensors. A path also identifies a subplan, i.e. the rest of the plan to be executed.

**Definition 2.10 (Subplan).** Let $p$ be a path of $\pi$. Then the subplan of $\pi$ identified by $p$, written $\pi(p)$, is defined as follows:

- if $p = \epsilon$, then $\pi(p) = \pi$;
- if $\pi = a \circ \pi'$ and $p = a \circ p'$, then $\pi(p) = \pi'(p')$;
- if $\pi = \mathbf{if} \ a \ \mathbf{then} \ \pi_1 \ \mathbf{else} \ \pi_2$ and $p = o \circ p'$, then $\pi(p) = \pi_1(p')$;
- if $\pi = \mathbf{if} \ a \ \mathbf{then} \ \pi_1 \ \mathbf{else} \ \pi_2$ and $p = \overline{o} \circ p'$, then $\pi(p) = \pi_2(p')$.

We say that $\pi'$ is a (strict) subplan of $\pi$ iff there exists a (nonempty) $p \in \text{Paths}(\pi)$ such that $\pi' = \pi(p)$.

2.3.2. Semantic notions: Plan applicability and execution

In general, we are interested in applicable plans, i.e. plans whose execution guarantees that an action is never attempted unless it is applicable, regardless of nondeterminism.

**Definition 2.11 (Applicability).** A plan $\pi$ is applicable in state $s$ iff either
– \( \pi \) is the empty plan \( \varepsilon \); or
– \( \pi = a \circ \pi_1 \), \( a \) is applicable in \( s \), and \( \pi_1 \) is applicable in every \( s' \in R(s, a) \); or
– \( \pi = \text{if } o \text{ then } \pi_1 \text{ else } \pi_2 \), and:
  – if \( s \in X^-(o) \), \( \pi_1 \) is applicable in \( s \), and
  – if \( s \in X^-(\overline{\delta}) \), \( \pi_2 \) is applicable in \( s \).

Considering the case of noisy sensing, where \( X^-(o) \cap X^-(\overline{\delta}) \neq \emptyset \), the definition deals with branching by requiring that both \( \pi_1 \) and \( \pi_2 \) are applicable to states associated both to \( o \) and to \( \overline{\delta} \).

The execution of a plan is defined in terms of the runs associated to it. Intuitively, a run contains the states, observations and actions encountered while executing the plan. Depending on the nondeterministic behavior of the domain, and on the uncertainty on the initial state, a plan can be associated with different runs.

**Definition 2.12 (Runs of a plan).** A run is a sequence \( \sigma = (s_0, o_0) \circ a_1 \circ (s_1, o_1) \circ a_2 \circ \cdots \circ a_n \circ (s_n, o_n) \), where \( s_i \in S \), \( o_i \in O \), and \( a_i \in A \). A sequence \( \sigma \) is a run of a plan \( \pi \) from state \( s \) iff either

– \( \pi = \varepsilon \), and \( \sigma = (s, o) \) with \( o \in X(s) \); or
– \( \pi = a \circ \pi_1 \) and \( \sigma = (s, o) \circ a \circ \sigma_1 \) with \( o \in X(s) \), and \( \sigma_1 \) is a run for \( \pi_1 \) for some \( s_1 \in R(s, a) \neq \emptyset \); or
– \( \pi = \text{if } o \text{ then } \pi_1 \text{ else } \pi_2 \), \( s \in X^-(o) \), and \( \sigma \) is a run of \( \pi_1 \) starting from \( (s, o) \); or
– \( \pi = \text{if } o \text{ then } \pi_1 \text{ else } \pi_2 \), \( s \in X^-(\overline{\delta}) \), \( o' \in X(s) \), and \( \sigma \) is a run of \( \pi_2 \) starting from \( (s, o') \).

**Example 2.13.** Consider the following plan for the robot domain of Example 2.4:

\[
\text{GoEast} \circ (\text{if } \text{ne} \text{ then } \text{GoSouth} \circ \text{GoWest} \text{ else } \text{GoWest})
\]

Action \text{GoEast} is executed first; then, if observation \text{ne} occurs, then \text{GoSouth} \circ \text{GoWest} is executed; otherwise, the plan executes \text{GoWest}. The plan is applicable in the two states \text{NW} and \text{SW}: \text{GoEast} is applicable in both states, and \( (\text{if } \text{ne} \text{ then } \text{GoSouth} \circ \text{GoWest} \text{ else } \text{GoWest}) \) is applicable in the resulting states \{\text{NE}, \text{SE}\}, since if the observation \text{ne} occurs, \text{GoSouth} \circ \text{GoWest} is applicable in \text{NE}, and otherwise \text{GoWest} is applicable in \text{SE}. Fig. 3 depicts the set of runs of the plan in Example 2.4, starting from the states \text{NW} and \text{SW}; in each run, the robot position is indicated as a soft cornered box. There are three possible runs, which correspond to the fact that we consider two possible initial
states, and for one of them, the first executed action GoEast may produce two different possible outcomes. Thus, the execution of the plan may originate one of runs $\sigma_1, \sigma_2$ and $\sigma_3$, where:

- $\sigma_1 = (\text{NW}, \text{nsw}) \circ \text{GoEast} \circ (\text{NE}, \text{ne}) \circ \text{GoSouth} \circ (\text{SE}, \text{se}) \circ \text{GoWest} \circ (\text{SW}, \text{nsw}),$
- $\sigma_2 = (\text{NW}, \text{nsw}) \circ \text{GoEast} \circ (\text{SE}, \text{se}) \circ \text{GoWest} \circ (\text{SW}, \text{nsw})$, and
- $\sigma_3 = (\text{SW}, \text{nsw}) \circ \text{GoEast} \circ (\text{SE}, \text{se}) \circ \text{GoWest} \circ (\text{SW}, \text{nsw}).$

We see that this plan is guaranteed to reach SW for all its runs, from any of the initial states, either with three actions (if the outcome of the first action is NE), or with two actions (if it is SE).

In the following, we write $\text{Runs}(s, \pi)$ to denote the set of runs of $\pi$ starting from $s$. We write $\text{FinalStates}(s, \pi)$ to indicate the set of the final states of the runs of $\text{Runs}(s, \pi)$. These notions can be generalized to sets of states: if $B$ is a set of states, we write $\text{Runs}(B, \pi)$ for $\bigcup_{s \in B} \text{Runs}(s, \pi)$, and $\text{FinalStates}(B, \pi)$ to indicate the set of final states of the runs of $\text{Runs}(B, \pi)$.

2.4. The problem of strong planning

A planning problem under partial observability for a given domain is described by a set of possible initial states, and a set of goal states. We are interested in strong solutions, i.e. plans that are guaranteed to be applicable in all the initial states, and that are guaranteed to end in a goal state for any possible run starting from an initial state. Other definitions of solution are also possible: for instance, a more optimistic approach would be to look for weak solutions, which only require the goal to be reached for some possible run. (See [34] for a general discussion in the case of full observability.)

Definition 2.14 (Planning problem and strong solution). A planning problem is a 3-tuple $\langle D, I, G \rangle$, where $D = \langle \Sigma, O, X \rangle$ is a planning domain, $\emptyset \neq I \subseteq S$ is the set of initial states, and $\emptyset \neq G \subseteq S$ is the set of goal states. The plan $\pi$ is a strong solution to the problem $\langle D, I, G \rangle$ iff

- $\pi$ is applicable in every state of $I$, and
- every run of $\pi$ from a state in $I$ ends in $G$, i.e. $\text{FinalStates}(I, \pi) \subseteq G$.

Example 2.15. Consider the planning problem given by the domain in Example 2.4, the initial set of states $I = \{\text{NW}, \text{SW}\}$, and the goal states $G = \{\text{SW}\}$. The plan in Example 2.13 is a strong solution to the planning problem.

3. Strong planning in the belief space

A major consequence of partial observability is that uncertainty must be explicitly dealt with at planning time. It may be impossible to detect at run time which is the actual initial state among the possible ones, even with the support of observations; for instance, in Example 2.13, the robot can not uniquely identify its initial state, since the states NW and SW are associated with the same observation nsw. Similarly, even if the current state is known, nondeterministic action effects may result in a set of possible states which is impossible to distinguish.

We model the impossibility of uniquely identifying the state of the domain by means of belief states [16,43]. A belief state is a set of states, intuitively all the states which are possible but indistinguishable. In the following, $B, B_0, B_1, \ldots$ are belief states. The basic notions related to plan execution can be naturally extended to the case of belief states as follows.

Definition 3.1 (Applicability in a belief). An action $a$ is applicable in the belief state $B$, written $\text{applicable}(a, B)$, iff it is applicable in all the states in $B$.

This definition captures the idea that, since the states in $B$ are indistinguishable, we never “risk” by attempting the execution of an action which might not be applicable. Plan applicability over a belief is also defined similarly, based on Definition 2.11.
Definition 3.2 (Execution in a belief). The execution of \( a \) in \( B \), written \( \text{Exec}(B, a) \), is defined as \( \bigcup_{s \in B} \mathcal{R}(s, a) \) when \( a \) is applicable in \( B \), and \( \emptyset \) otherwise.

Since the states in \( B \) are indistinguishable, when the action is executed, every state that may result from executing the action on any of the states in \( B \) must be taken into account. The effect of an observation can be modeled as follows. Let \( B \) be the belief state containing all the states of the domain that are possible but indistinguishable. If the observation \( o \) occurs, then our belief can be refined to \( B \cap \mathcal{X}^-(o) \): that is, we can rule out the states that are not compatible with an occurrence of \( o \). Similarly, if \( \bar{o} \) (i.e., an observation other than \( o \)) occurs, then our belief is given by \( B \cap \mathcal{X}^-(\bar{o}) \), since we can rule out the states that are not compatible with \( o \) not occurring. Given a belief state, it is sometimes possible to know prior to observing whether an observation \( o \) occurs or not. In this case, we say the observation is predetermined, and observing does not convey any additional information over the current belief. Formally, an observation \( o \) is predetermined over a belief \( B \) iff either \( B \cap \mathcal{X}^-(\bar{o}) = \emptyset \) (i.e., we know \( o \) will take place), or \( B \cap \mathcal{X}^-(o) = \emptyset \) (i.e., we know \( o \) will not take place).

Example 3.3. Let us consider the robot domain in Example 2.4, and assume the robot can be either in \( \text{NE} \) or in \( \text{SE} \). The corresponding belief state \( B = \{ \text{NE}, \text{SE} \} \) can be refined when either observation \( \text{ne} \) or \( \text{se} \) occurs. With observation \( \text{ne} \), we have that \( B \cap \mathcal{X}^-(\text{ne}) = \text{NE} \) and \( B \cap \mathcal{X}^-(\text{ne}) = \text{SE} \). When observation \( \text{se} \) occurs, we have that \( B \cap \mathcal{X}^-(\text{se}) = \text{SE} \) and \( B \cap \mathcal{X}^-(\text{se}) = \text{NE} \). We notice that \( \text{ne} \) and \( \text{se} \) convey equivalent information, since they split the belief in the same way. The observation \( \text{nsw} \) is predetermined, since \( B \cap \mathcal{X}^-(\text{nsw}) = \emptyset \); that is, we know that \( \text{nsw} \) cannot occur on \( B \), and indeed, \( B \cap \mathcal{X}^-(\text{nsw}) = B \).

Given this, we can now define the way a belief evolves along the execution of one of the possible paths of a plan:

Definition 3.4 (Belief execution along a path). Let \( \pi \) be a plan applicable in a belief state \( B \), and let \( p \in \text{Paths}(\pi) \). The belief execution along \( p \), written \( \text{Exec}(B, p) \), is defined as follows:

- if \( p = \varepsilon \), then \( \text{Exec}(B, p) = B \);
- if \( p = a \circ p' \), then \( \text{Exec}(B, p) = \text{Exec}(\text{Exec}(B, a), p') \);
- if \( p = o \circ p' \), then \( \text{Exec}(B, p) = \text{Exec}(B \cap \mathcal{X}^-(o), p') \);
- if \( p = \bar{o} \circ p' \), then \( \text{Exec}(B, p) = \text{Exec}(B \cap \mathcal{X}^-(\bar{o}), p') \).

Intuitively, the belief execution along a path is the belief resulting from the execution of the actions and the occurrence of observations in the path. We write \( \text{FinalBels}(B, \pi) = \{ \text{Exec}(B, p) \mid p \in \text{MaxPaths}(\pi) \} \) for the set of beliefs associated to maximal paths in \( \pi \).

The following theorem shows that it is possible to define strong plans in terms of belief states; this allows recasting strong planning under partial observability as a problem of search in the belief space. The proof is reported in Appendix A.

Theorem 3.5. Let \( \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle \) be a planning problem, and let \( \pi \) be applicable in \( \mathcal{I} \). Then \( \pi \) is a strong solution to \( \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle \) iff \( \bigcup \text{FinalBels}(\mathcal{I}, \pi) \subseteq \mathcal{G} \).

We can collect together the belief executions along the paths of a plan; once these are associated with their paths, they form a tree structure which represents the way beliefs can be reached by acting or observing. We call this structure a Belief Execution Tree, and represent it as a labelled directed hypergraph, that is a graph structure whose nodes are connected by labelled directed hyperarcs. Each hyperarc associates a source node to a set of target nodes; in particular, the hyperarcs of a Belief Execution Tree associate a node (a belief) either to a single node, representing the execution of an action, or to a pair of nodes, representing the two possible outcomes of executing an observation. A Belief Execution Tree can be univoquely associated to a plan, where a subtree corresponds to a subplan; these features will be relevant in the design of a search algorithm based on iterative extension of a tree.
Definition 3.6 (Belief Execution Tree). Let \( \pi \) be a plan applicable in the belief \( B \). Then, the Belief Execution Tree of \( \pi \) from \( B \), written \( \text{BET}(\pi, B) \), is the directed hypergraph with nodes the following set

\[
\left\{ \langle p, \text{Exec}(B, p) \rangle \mid p \in \text{Paths}(\pi) \right\}
\]

and arcs the least set containing, for all \( p, p' \in \text{Paths}(\pi) \) with \( p' = p \circ a \), the or arc

\[
\langle \langle p, \text{Exec}(B, p) \rangle, a, \langle p', \text{Exec}(B, p') \rangle \rangle
\]

and for all \( p, p', p'' \in \text{Paths}(\pi) \) with \( p' = p \circ o \) and \( p'' = p \circ \delta \), the and arc

\[
\langle \langle p, \text{Exec}(B, p) \rangle, o, \langle p', \text{Exec}(B, p') \rangle, \langle p'', \text{Exec}(B, p'') \rangle \rangle
\]

Since each path in \( \text{Paths}(\pi) \) is associated with exactly one node in \( \text{BET}(\pi, B) \), in the following we also simply write \( p \) to indicate the node \( \langle p, \text{Exec}(B, p) \rangle \). Given the properties of \( \text{Paths} \) and the structure of plans, it is easy to see that the Belief Execution Tree is indeed a tree: every non-leaf node is the source of one outcoming arc, and every non-root node is the target to one incoming arc.

More specifically, the root of the tree is the node associated with \( \varepsilon \), and the leaves are the nodes associated to maximal paths (and thus, to the beliefs in \( \text{FinalBels}(B, \pi) \)). Intuitively, the belief associated with a leaf node corresponds to a complete execution of a maximal path of the plan. The father and son relations between nodes are defined in the obvious way. We also say that two nodes are brothers iff they are indexed by two paths of the form \( p \circ o \) and \( p \circ \delta \); the node \( p \) is an ancestor of node \( p' \) if \( p \) is a strict prefix of \( p' \). From now on, we will also represent a Belief Execution Tree as a triple \( \langle n_0, N, E \rangle \), with \( N \) and \( E \) the sets of nodes and arcs, and \( n_0 \) the root node. We will say a tree \( T \) is a (strict) prefix of a tree \( T' \) if the nodes and edges of \( T \) are (strict) subsets of those in \( T' \).

Example 3.7. Fig. 4 depicts the belief execution tree associated to the strong solution of Example 2.15. The set \( \{Bs6\} \) is the corresponding final belief set, where \( Bs6 \) is the belief associated to both leaf nodes in the Belief Execution Tree.

Given a belief execution tree \( T \) and a node \( n \) belonging to it, we can extract the subtree of \( T \) rooted at \( n \):

Definition 3.8 (Subtree). Given a belief execution tree \( T \) and a node \( n \) of \( T \), associated with path \( p_n \), its subtree from \( n \), indicated with \( \text{SubTree}(T, n) \), is a belief execution tree such that:

- for every node of \( T \) indexed with a path \( p \) s.t. \( p_n \preceq p \), there exists a node \( n' \in \text{SubTree}(T, n) \). If \( p = p_n \), then \( n' \) is indexed by path \( \varepsilon \); if \( p = p_n \circ p' \), then \( n' \) is indexed by path \( p' \).
- for every arc that associates nodes of \( T \) that have correspondent nodes in \( \text{SubTree}(T, n) \), there exists one correspondent arc in \( \text{SubTree}(T, n) \), connecting the correspondent nodes.

It can be easily argued that a subtree is unique, and is a belief execution tree. Moreover, the association of a belief execution tree with an executable plan is bijective:
Definition 3.9 (Plan associated to belief execution tree). Given a belief execution tree $T = \langle n_0, N, E \rangle$, its associated plan $\text{PlanOf}(T)$ is defined as follows:

- if $E = \emptyset$, then $\text{PlanOf}(T) = \varepsilon$;
- if $(n_0, a, n) \in E$, then $\text{PlanOf}(T) = a \circ \text{PlanOf}(\text{SubTree}(T, n))$;
- if $(n_0, o, n_1, n_2) \in E$, then $\text{PlanOf}(T) = \text{if } o \text{ then } \pi_1 \text{ else } \pi_2$, where $\pi_1 = \text{PlanOf}(\text{SubTree}(T, n_1))$ and $\pi_2 = \text{PlanOf}(\text{SubTree}(T, n_2))$.

Also in this case, uniqueness is immediate (given the uniqueness of subtrees). If $\pi$ is a plan applicable on belief $I$, $\text{PlanOf}(\text{BET}(\pi, I)) = \pi$, as it is possible to prove by recursion on the plan structure.

4. Planning as and-or search

In this section, we present the algorithm for strong planning under partial observability (Section 4.1), and formally prove its properties (Section 4.2).

4.1. The planning algorithm

4.1.1. Top level

Our approach for strong planning under partial observability is based on progressing from the initial belief state $I$ towards the goal $G$. The belief space is explored by incrementally constructing a search tree, which collects together a set of belief execution trees, each of them being associated to one plan that has been expanded during the search. The search takes place by constructing belief states, either by applying actions or by means of observations. The application of an action starting from a belief state results in a belief state, while applying observations produces sets of belief states (that is, the starting belief state is split amongst the possible observations). We essentially perform a kind of and-or search with loop detection. The application of an action corresponds to an “or” expansion: to reach the goal from a belief $B$, it is enough to find a strong solution for any of the actions applicable in $B$. Applying an observation induces an “and” expansion: once we decide which observation $o$ to consider, the plan must take into account both cases where $o$ holds and does not hold. Indeed, the selection of the observation to be considered is per se “or” choice; however, we avoid introducing an explicit “or-arc” and corresponding node to represent this choice; instead, we represent the chosen observation as a label of the “and-arc”, consistently with what we do in Belief Execution Trees.

By analyzing the beliefs associated to the ancestors of a node, we are able to detect and avoid loop-backs, i.e. paths that traverse two nodes associated to the same belief. In this way, the search restricts to plans that can never traverse a belief more than once during the execution. Loop detection and avoidance is crucial since, differently from well-known algorithms as AO* [66], we can not assume that the search space is acyclic, and differently from more recent extensions such as LAO* [53], we do not accept cyclic solutions, whose execution is not guaranteed to terminate. A certain node is closed with success if a plan exists that leads from it to the goal; it is closed with failure if all possible plans lead to loop-backs.

The search terminates when either the expanded search tree has its root tagged as successful, or when no further expansion is possible and no solution has been found.

Example 4.1. Fig. 5 represents the search tree for the domain in Example 2.4. For instance, node $n_2$ is expanded into node $n_3$ by the only applicable action on it, GoWest, and into nodes $n_4$ and $n_5$ by the observation se. The figure does not report predetermined observations, such as sw for node $n_1$. Also, for sake of clarity, we do not report in the figure actions or observations that behave exactly as other (reported) actions or observations, since intuitively they are redundant, and would originate replicated portions of the search tree. (E.g., we omit observation ne, applied to node $n_2$, since it has the same effect of observation $se$.) Notice that even in this simple case, loops are possible; for instance, node $n_1$ loops back to node $n_2$. Also notice that several nodes associated to the same belief appear in the search tree, e.g. $n_4$, $n_8$ and $n_12$ are all associated to $Bs4$. This represents the fact that the same situation can be reached by different paths (in general, belonging to different plan prefixes). Each of such paths may or may not correspond to a loop: in this example, $n_12$ is associated to a looping path, while $n_4$ and $n_8$ are not.
The planning algorithm is described in Fig. 6. It takes as input the initial belief state and the goal belief state, while the domain representation is assumed to be globally available to the subroutines. The state of the search is represented within a search tree structure $ST$, which represents an acyclic portion of the search space of beliefs. Each node $n$ in the search tree is associated with an action-observation path $PATH(n)$, and with a belief state $BEL(n)$, that is reached by executing the action-observation path. In particular, the root $ROOT(ST)$ is associated to the initial belief, and to the empty path. In addition, each node generated in the search is associated with a ‘tag’, with three possible values:
Success, meaning that a plan is found that leads from the node to the goal; Failure, meaning that no such plan exists; or Undetermined, meaning that neither a plan has been found for the node, nor the algorithm has been able (yet) to detect that no such plan exists. The corresponding predicates are isSucc, isFail, and isUndet. The TagNode primitive sets the value of the tag for a node. The set of the undetermined leaves of the search tree is called frontier.

The algorithm first initializes the search tree with the start node (line 2), and checks if the empty plan is a solution to the problem (lines 3–4). Then, the main loop is entered (lines 6–16), where the tree is iteratively expanded until the root is tagged either as success or as failure. When the loop is exited (lines 17–20), a plan is constructed and returned in case of success; otherwise, Failure is returned.

The body of the main loop proceeds by selecting and extracting a node from the frontier, and expanding it. With the first step in the loop, at line 7, a node is selected for expansion, and is extracted from the frontier. The ExtractNodeFromFrontier primitive embodies the selection criterion, and is responsible for the style of the search (e.g. depth-first versus breadth-first). The selection criterion does not affect the properties of the algorithm, namely completeness, correctness and termination.

At line 8, node is expanded, considering every possible action and observation. For each applicable action, a new node is generated, and then connected to node via an or-arc. For every observation that is not predetermined, two new nodes are generated, and then connected to node via an and-arc. This expansion step also decides the status of each newly generated node n, updating the frontier consistently: namely, n is tagged Success if $Bel(n) \subseteq G$; it is tagged Failure iff there exists an ancestor node associated to the same belief, i.e. a loop has been generated; it is tagged Undetermined otherwise, and added to the frontier.

If it is possible to state the success of node based on the status of the newly introduced sons (primitive SonSucceed at line 9), i.e. if an or-son is successful node, or a pair of and-sons are successful, then node is tagged as success (line 10). In this case (line 11), the recursive PropagateSuccessOnTree primitive propagates success bottom-up on the search tree, to the ancestors of node, towards the initial node.

If the node is not detected to be successful, the SONSucceedFailure primitive tries to state if a node is a failure. This happens when every or-son causes a loop, and for every and-arc, at least one of the and-sons causes a loop. In this case, the node is tagged as failure, and the failure is propagated bottom-up, in order to cut the search in branches which are bound to fail because some leaf has failed.

**Example 4.2.** Fig. 7 describes a possible behavior of the algorithm for the example of Fig. 4, by depicting the search tree at the start of each iteration of the main loop. In the tree, nodes tagged as Undetermined are white, nodes tagged with Failure are shaded with a light grey, and nodes tagged with Success are shaded with a deeper grey. In this example, we consider a selection strategy such that, at each iteration, the frontier node which is leftmost in Fig. 7 is picked and expanded; the arcs built by the expansion are represented by continuous lines, while loops discovered at each iteration are shown as dashed lines. In this case, failure propagation only takes place at iteration 5, while success propagation takes place at iterations 6 and 7 (where it reaches the root of the tree, causing the algorithm to stop). Notice that we do not report irrelevant observations, such as, e.g., those that can be considered in the initial belief, since the algorithm, after detecting them, does not insert them into the search tree.

### 4.1.2. Expansion primitives

We now detail each of the expansion primitives in the algorithm, in order to provide a clear basis for discussion of the properties of the algorithm. We remark that, for the sake of clarity, we describe here naive implementations of the primitives, disregarding optimizations such as caching of previous results, or lazy evaluation.

**ExtendTree** The ExtendTree primitive, presented in Fig. 8, expands a frontier node by every non-predetermined observation and applicable action. For every applicable action, a node and an or-arc are respectively added to the nodes Nodes(ST) and to the arcs Arcs(ST) of the current search tree. Similarly, for every non-predetermined observation, a pair of nodes and an and-arc are added to the current search tree. For every new node n generated, the TagNewNode subroutine evaluates whether it is a success (i.e. $Bel(n) \subseteq G$), a failure (i.e. some node in the set of ancestors of n, Ancestors(n, ST), is associated to the same belief), or neither success nor failure; on the basis of this evaluation, n is associated to the corresponding tag.

We briefly observe here that in the actual implementation, we will represent states, actions and observations by using state variables, action variables and observations variables respectively. This representation proves convenient
Fig. 7. The behavior of the algorithm for the example.

under many aspects. First of all, in terms of modeling, it makes it easy to represent parameterized actions, and to modularly describe independent features of the domain state. Second, it makes it possible to implement a lazy approach when extending the tree: rather than assigning every action (observation) variable, which corresponds to selecting a single action (observation), we can assign a subset of them, and generate a node that collects together the outcomes of a set of actions (observations). This idea is particularly simple to realize for observations. More specifically, we repre-
sent observations by boolean observation variables, and we apply observations to a node by splitting (in a binary way) on a single observation variable at a time. Since observation variables naturally model sensors, this may be thought of as splitting on one binary sensor at a time. As a result, we introduce nodes where sets of observations (intended to be in disjunction) are evaluated together. In many cases, evaluating a limited set of observation variables is enough to gather a sufficient amount of knowledge, whereas it would be otherwise necessary to test a wide set of observations; in these cases, we effectively reduce the branching in the search and improve its efficiency.

While relevant to the performance of the algorithm, we will leave out these representation-dependent implementation details here, in order not to clutter the conceptual representation of the algorithm (intuitively, the main modification consists in having observation variables, rather than observations, considered in the loop at lines 10–20; but this also impacts on the way plans are built by BUILDPLAN). We remark that these representation choices have no impact on the properties of the algorithm.

\textbf{SONSYIELDSUCCESS, SONSYIELDFAILURE} The \textbf{SONSYIELDSUCCESS} primitive, see Fig. 9, returns true on node \(n\) iff either (a) there exists an or-arc from \(n\) to \(n_1\) in the tree, and \(n_1\) is a success node in the tree, or (b) there exists an and-arc from \(n\) to \(n_T\) and \(n_F\) in the tree, and both \(n_T\) and \(n_F\) are success nodes in the tree. The \textbf{SONSYIELDFAILURE} primitive returns true for a node \(n\) if all the actions applied on it lead to failure, and for every observation taken, at least one of the results leads to a failed node.

Notice that for any non-leaf node \(n\), \textbf{SONSYIELDFAIL}(\(n\)) and \textbf{SONSYIELDSUCC}(\(n\)) can not hold together. This comes from the fact that, for \textbf{SONSYIELDSUCC}(\(n\)) to hold, \(n\) must have at least a successful or-son, or a pair of successful brother and-sons; but in that case, it can not be true that every or-son is a failure and that for every pair of brother and-sons, at least one is a failure, so \textbf{SONSYIELDFAIL}(\(n\)) can not hold.
\begin{Verbatim}
/ function SONYIELDSUCC(n, ST) 
  res := false
  forall (n, a, n') \in Arcs(ST)
    res := res \lor IS\text{SUCC}(n')
endfor
return res

/ function SONYIELDFAIL(n, ST) 
  res := false
  forall (n, a, n') \in Arcs(ST)
    res := res \land IS\text{FAIL}(n')
endfor
return res

Fig. 9. The procedures to detect failure or success of an expanded node.

/ procedure PROPAGATESUCCONTREE(n, ST) 
  if (n \neq \text{ROOT}(ST)) then
    nfat := FATHER(n)
    if (SONYIELDSUCC(nfat, ST)) then
      TAGNODE(nfat, Success)
      PROPAGATESUCCONTREE(nfat, ST)
    else
      PROPAGATESUCCONTREE(n, ST)
  fi
end

/ procedure PROPAGATEFAILONTREE(n, ST) 
  if (n \neq \text{ROOT}(ST)) then
    nfat := FATHER(n)
    if SONYIELDFAIL(nfat, ST) then
      TAGNODE(nfat, Failure)
      PROPAGATEFAILONTREE(nfat, ST)
    fi
  fi
end

Fig. 10. The procedures to propagate success/failure of a node.

/ function BUILDPLAN(n, ST) 
  if BE\ell(n) \subseteq G then
    return \varepsilon
  else if \exists (n, a, n') \in Arcs(ST): IS\text{SUCC}(n') then
    pick (n, a, n') \in Arcs(ST): IS\text{SUCC}(n')
    \pi' := BUILDPLAN(n', ST)
    return a \circ \pi'
  else
    pick (n, o, n_T, n_F) \in Arcs(ST): IS\text{SUCC}(n_T) \land IS\text{SUCC}(n_F)
    \pi_T := BUILDPLAN(n_T, ST)
    \pi_F := BUILDPLAN(n_F, ST)
    return if o then \pi_T else \pi_F
  fi
end

Fig. 11. The plan building procedure.

PROPAGATESUCCONTREE, PROPAGATEFAILONTREE The success and failure propagation routines, see Fig. 10, are defined recursively, and simply traverse the tree bottom up, from the current node. During this propagation, PROPAGATESUCCONTREE tags a node with Success if its success can be established by SONYIELDSUCC, while PROPAGATEFAILONTREE tags a node with Failure if its success can be established by SONYIELDFAIL.

BUILDPLAN The BUILDPLAN primitive is reported in Fig. 11, and is used to build the plan associated to a successful node n. It assumes that n is either a successful leaf, or it has at least a successful or-son, or a pair of successful and-sons, and that the same is valid for its descendants (i.e. every successful descendant is either a leaf, or has a successful or-son, or a pair of successful and-sons). The function implements a top-down recursion scheme over the search tree structure, starting from n, and, under the above assumption, it stops at successful leaf nodes.

4.2. Properties of the algorithm

We now show that the algorithm is correct, complete and terminating. To do so, we first introduce a structural notion of search tree to represent the status of the search at a given iteration of the algorithm, and we observe that the algorithm restricts the search to a class of plans that feature no loops nor predetermined observations; we show that
this restriction gives no loss of generality. Finally, we state and prove the termination, correctness and completeness properties. For the sake of readability, the full body of lemmas and proofs are reported in Appendix B.

4.2.1. Search tree

We define a search tree as the tree resulting from the union of the belief execution trees associated to each plan in a set. This relies on an notion of node equivalence: two nodes are equivalent if they are associated to the same belief, and to the same path. This notion corresponds to the search tree which is expanded by the algorithm at each iteration. In the following, we denote with $ST_i$ the search tree at the beginning of the $i$th iteration of the loop of the planning algorithm in Fig. 6, i.e. prior to the evaluation of the stop condition at line 6 of the algorithm.

Definition 4.3 (Search tree). Let $\{\pi_1, \ldots, \pi_n\}$ be a set of plans for a domain $D$, where each plan $\pi_i$ is applicable in a belief $B$. The search tree associated to the set of plans $\{\pi_1, \ldots, \pi_n\}$ is the graph such that:

- its set of nodes is the collection of every node in $\text{BET}(\pi_i, B)$;
- its set of arcs is the collection of every arc in $\text{BET}(\pi_i, B)$.

We say that a search tree $ST$ contains a node $n$ if there is a node of $ST$ equivalent to $n$; a search tree $ST$ contains a belief execution tree $T$ iff it contains every node and arc of $T$.

4.2.2. Non-redundant solutions

The design of the extension primitives, the mechanism of loop detection and avoidance, and the stopping criteria are such that the algorithm restricts its search to the space of non-redundant plans, i.e. plans that are applicable, loop-free, terminate as soon as the goal is known to be reached, and never consider predetermined observations. We formally define non-redundant solutions, and prove that restricting the search to such class of solutions preserves completeness. Intuitively, a strong solution is non-redundant iff no belief execution along any of its paths may traverse the same belief more than once, and no belief execution along any of its paths may continue after achieving the goal.

Definition 4.4 (Non-redundant plan and solution). Let $\pi$ be an applicable plan for a problem $\langle D, I, G \rangle$. Then $\pi$ is non-redundant iff the following conditions hold:

1. For any maximal path $p$ of $\pi$, the belief execution associated to $p$ can not traverse two nodes associated to the same belief;
2. For any path $p$ of $\pi$, if the belief execution associated to $p$ traverses a node $n$ associated with a belief $B \subseteq G$, then $n$ is the terminal node of the execution.

A strong solution $\pi$ for $\langle D, I, G \rangle$ is called a non-redundant solution iff $\pi$ is non-redundant.

We observe that any (possibly redundant) plan can be normalized into a unique non-redundant plan by the NORMALIZE procedure reported in Fig. 12. The following theorem states that restricting the search to non-redundant plans does not affect completeness of the algorithm.

Theorem 4.5. Let $\pi$ be a strong solution for a problem $\langle D, B, G \rangle$. Then $\text{NORMALIZE}(\pi, B, G)$ is a non-redundant solution for $\langle D, B, G \rangle$.

4.2.3. Main properties

We now state the main properties of the algorithm.

Theorem 4.6 (Termination). Given a planning problem, the execution of the planning algorithm in Fig. 6 terminates.

Termination is proved by showing that BUILDPLAN terminates, and that the main loop of the algorithm terminates. Intuitively, BUILDPLAN terminates because the generated plans are finite, and each recursion in BUILDPLAN adds either an action or an observation to the produced plan.
To prove that the main loop terminates, we prove that each iteration of the main loop terminates, and that the number of such iterations is finite.

The first statement is simple to prove: the success/failure propagation routines terminate, since they implement bottom-up traversal of a finite tree, and EXTENDTREE terminates, since the number of actions and observations that can be applied to a node is finite.

To prove that the number of iterations is finite, we observe that in the worst case, the algorithms terminates after having generated every possible acyclic paths, and having detected, for each of them, that it is impossible to further expand them without looping. But this involves a number of iterations bounded by the (finite) number of acyclic paths in the belief space, plus the (finite) number of beliefs.

**Theorem 4.7 (Correctness).** When the algorithm returns a plan \( \pi \), then \( \pi \) is a strong solution for the problem.

The proof relies on the following facts:

- Success and Failure tags are “persistent” during the search: once a node is tagged with one of those values, it will remain tagged with the same value in any successive iteration of the algorithm;
- at a given iteration \( i \), a node \( n \) becomes tagged Success only if a strong solution exists, within the search tree \( ST[\pi][i] \), that leads from \( n \) to the goal;
- if a plan exists from a node \( n \) to the goal, BUILDPPLAN returns one such plan.

These statements are valid for every node of \( ST[\pi][i] \), at each iteration \( i \) of the algorithm; once considered together and instantiated on the root node, correctness follows immediately.

**Theorem 4.8 (Completeness).** If a strong solution exists, then the algorithm returns a strong solution.

The proof is performed by reductio ad absurdum: we assume that after a certain number \( k \) of iterations, the algorithm returns Failure, while a (non-redundant) solution \( \pi \) to the problem exists.

But this can not be the case because, would such execution exist, we show that \( ST[\pi][k] \) would contain a prefix of the belief execution tree associated with \( \pi \), and that each node of \( ST[\pi][k] \) in that prefix (including the root of \( ST[\pi][k] \)) could not be marked Failure, which is necessary for the algorithm to terminate with Failure.
5. Symbolic implementation

In this section, we show how Symbolic Model Checking techniques can be extended to provide an efficient implementation platform for the search paradigm presented in previous section. Model Checking is a formal verification technique based on the exploration of finite state automata [30]. Symbolic model checking [64] is a particular form of model checking using Binary Decision Diagrams (BDDs) [25] to compactly represent and efficiently analyze finite state automata. The introduction of symbolic techniques into model checking led to a breakthrough in the size of models which could be analyzed [4], and made it possible for model checking to be routinely applied in industry, especially in the design of semiconductors (see [38] for a survey).

In the rest of this section, we will give an introduction to BDDs (Section 5.1), then we will show how BDDs are used to represent planning domains (Section 5.2); we will discuss the extension which allows to symbolically represent belief states and their transformations, thus allowing for an efficient implementation of the algorithm described in previous section (Section 5.3).

5.1. Binary decision diagrams

A Reduced Ordered Binary Decision Diagram [25,27] (improperly called 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 branches. Fig. 13(a) depicts a BDD for \((a_1 \leftrightarrow b_1) \land (a_2 \leftrightarrow b_2) \land (a_3 \leftrightarrow b_3)\). At each non-terminal node, the left [right, respectively] branch is depicted as a solid [dashed, resp.] line, and represents the assignment of the value True [False, resp.] to the corresponding variable. A BDD represents a boolean function. For a given truth assignment to the variables in the BDD, the value of the function is determined by traversing the graph from the root to the leaves, following each branch indicated by the value assigned to the variables.\(^1\) The reached leaf node is labeled with the resulting truth value. If \(\phi\) is a BDD, its size \(\|\phi\|\) is the number of its nodes. If \(n\) is a node, \(\text{var}(n)\) indicates the variable indexing node \(n\).

\(^1\) 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. 13(a).
BDDs are a canonical representation of Boolean functions. The canonicity follows by imposing a total ordering $\prec$ 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. $\text{var}(n) \prec \text{var}(m)$, 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 can also be represented as BDD transformations. In the following, if $v$ is a variable, and $\Phi$ and $\psi$ are BDDs, we indicate with $\Phi[v/\psi]$ the BDD resulting from the substitution of $v$ with $\psi$ in $\Phi$. If $v$ is a vector of BDD variables, we indicate with $\|v\|$ the number of elements of the vector. If $v_1$ and $v_2$ are vectors of distinct variables such that $\|v_1\| = \|v_2\|$, we indicate with $\Phi[v_1/v_2]$ the parallel substitution in $\Phi$ of the variables in vector $v_1$ with the (corresponding) variables in $v_2$.

BDDs also allow for transformations described as quantifications, in the style of Quantified Boolean Formulae (QBF). QBF is a definitional extension to propositional logic, where propositional variables can be universally and existentially quantified. In QBF, quantifiers can be arbitrarily applied and nested. In general, a QBF formula has an equivalent propositional formula, but the conversion is subject to an exponential blow-up. If $\Phi$ is a formula, and $v_i$ is one of its variables, the existential quantification of $v_i$ in $\Phi$, written $\exists v_i.\Phi(v_1,\ldots,v_n)$, is equivalent to $\Phi(v_1,\ldots,v_n)[v_i/\text{False}] \lor \Phi(v_1,\ldots,v_n)[v_i/\text{True}]$. Analogously, the universal quantification $\forall v_i.\Phi(v_1,\ldots,v_n)$ is equivalent to $\Phi(v_1,\ldots,v_n)[v_i/\text{False}] \land \Phi(v_1,\ldots,v_n)[v_i/\text{True}]$. In terms of BDD computations, a quantification corresponds to a transformation mapping the BDD of $\Phi$ 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 quadratic in the size of the BDD being quantified, and linear in the number of variables being quantified, i.e. $O(\|v\| \cdot \|f\|^2)$ [25,27].

BDD packages are efficient implementations of such data structures and algorithms (see [24,33,82,89]). Basically, 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) [24,82]. Time efficiency is obtained by maintaining a “computed table”, which keeps track of the results of recently computed transformations, thus using memorization to avoid the re-computation.

A critical computational factor with BDDs is the order of the variables. (Fig. 13 shows an example of the impact of a change in the variable ordering on the size of a BDD.) For a certain class of boolean functions, the size of the corresponding BDD is exponential in the number of variables for any possible variable ordering [26]. 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).

5.2. Symbolic representation of planning domains

A planning domain $\mathcal{D} = \langle \mathcal{S}, \mathcal{O}, \mathcal{X} \rangle$, with $\mathcal{S} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$, can be represented symbolically using BDDs. Let us first focus on the representation of the nondeterministic state transition system of the domain. A set of (distinct) BDD variables, called state variables (or fluents), is devoted to the representation of the states $S$ of the domain. For non-boolean variables, a logarithmic bit encoding is performed as common practice in Symbolic Model Checking. For instance, the robot navigation domain of Fig. 14 is described by means of two fluents $x$ and $y$, both having range $\{0,1,2,3\}$. Each range can be encoded using two boolean variables ($x_{1,2}$ and $y_{1,2}$, respectively), with the following relation to the values of the two state variables.

$$x = 0 \mapsto \neg x_1 \land \neg x_2 \quad y = 0 \mapsto \neg y_1 \land \neg y_2$$
In the following, we write \( x \) for the vector of (BDD variables representing the) state variables of the domain. Because the particular order is irrelevant (but for performance issues), in the rest of this section we will not distinguish between a proposition and the corresponding BDD representation.

A state is a complete set of assignments to state variables. For each state \( s \), there is a corresponding assignment to the state variables \( x \), i.e. the assignment where each variable corresponding to a proposition \( p \) holding in \( s \) is assigned to True, and each other variable is assigned to False. We represent \( s \) with the BDD \( \xi(s) \), having such an assignment as its unique satisfying assignment. For instance, the BDD for the state \((0, 2)\), written \( \xi((0, 2)) \), is \((\neg x_1 \land \neg x_2) \land (y_1 \land \neg y_2)\).

The above representation naturally extends to any set of states \( Q \subseteq S \) as follows:

\[
\xi(Q) \doteq \bigvee_{s \in Q} \xi(s)
\]

In other words, we associate a set of states with the generalized disjunction of the BDDS representing each of the states. Notice that the satisfying assignments of the \( \xi(Q) \) are exactly the assignments representing the states in \( Q \). This representation mechanism is very natural. For instance, the BDD \( \xi(I) \) representing the set of initial states \( I \doteq \{(0, 1), (0, 2)\} \) of the robot navigation domain of Fig. 14 is:

\[
\xi(I) \doteq ((\neg x_1 \land \neg x_2) \land ((\neg y_1 \land y_2) \lor (y_1 \land \neg y_2)))
\]

while for the set of goal states \( G \doteq \{(3, 1), (3, 2)\} \) the corresponding BDD is

\[
\xi(G) \doteq ((x_1 \land x_2) \land ((\neg y_1 \land y_2) \lor (y_1 \land \neg y_2)))
\]

A BDD is also used to represent the set \( S \) of all the states of the planning domain. In the robot navigation example we are considering, \( \xi(S) = True \), since every assignment to \( x \) corresponds to a state in \( S \).

In general, a BDD represents the set of (states which correspond to) its models. As a consequence, set theoretic transformations are naturally represented by propositional operations, as follows.

\[
\xi(S \setminus Q) \doteq \xi(S) \land \neg \xi(Q)
\]
\[
\xi(Q_1 \cup Q_2) \doteq \xi(Q_1) \lor \xi(Q_2)
\]
\[
\xi(Q_1 \cap Q_2) \doteq \xi(Q_1) \land \xi(Q_2)
\]

The main efficiency of this symbolic representation lies in the fact that the cardinality of the represented set is not directly related to the size of the BDD. For instance, \( \xi([x = 0]) = \xi(([0, 0], (0, 1), (0, 2), (0, 3)) = \neg x_1 \land \neg x_2 \) uses two (non-terminal) nodes to represent four states (see Fig. 17), while the set of states where state variable \( y \) has a value in the set \( \{1, 2\} \) is represented with three nonterminal nodes. \( \xi(I) \) uses five nonterminal nodes to represent two states. As limit cases, for this example \( \xi(S) \) and \( \xi(\emptyset) \) are (the leaf BDDs) True and False, respectively. Indeed, symbolic representation is extremely efficient in dealing with irrelevant information. Notice, for instance, that only the boolean variable \( x_1 \) occurs in \( \xi([[x \in \{2, 3\}]])) = \xi([[2, \{0, 1, 2, 3\}, (3, \{0, 1, 2, 3\})]) = x_1 \). For this reason, a
symbolic representation can have dramatic advantages over an explicit, enumerative representation. This is what allows symbolic, BDD-based model checkers to handle finite state automata with a very large number of states (see for instance [4]). In the following, we will collapse a set of states and the BDD representing it.

To represent actions, we use another set of BDD variables, called action variables, distinct from state variables, and denoted \( a \) in the following. Under the assumption of sequentiality of the system, i.e. that no concurrent actions are allowed, it is possible to encode actions into action variables using a logarithmic schema, similarly to what we did for state variables. This schema is actually used in our implementation. However, for the sake of simplicity, here we refer to an encoding where each possible action in \( A \) is associated with one action variable in \( a \). Intuitively, a BDD action variable is true if and only if the corresponding action is being executed. We use a BDD, \( \text{SEQ}(a) \), to express that exactly one of the action variables must be true at each time. For the robot navigation problem, where \( A \) contains four actions, we use the four BDD variables \( \text{GoNorth}, \text{GoSouth}, \text{GoWest} \) and \( \text{GoEast} \), while we express the serial encoding constraint with the following BDD:

\[
\text{SEQ}(a) \doteq (\text{GoNorth} \lor \text{GoSouth} \lor \text{GoWest} \lor \text{GoEast}) \land \\
\neg(\text{GoNorth} \land \text{GoSouth}) \land \neg(\text{GoNorth} \land \text{GoWest}) \land \\
\neg(\text{GoNorth} \land \text{GoEast}) \land \neg(\text{GoSouth} \land \text{GoWest}) \land \\
\neg(\text{GoSouth} \land \text{GoEast}) \land \neg(\text{GoWest} \land \text{GoEast}).
\]

Similarly to the case of state variables, we are referring to BDD action variables with symbolic names for the sake of simplicity. The position of the BDD action variables in the ordering of the BDD package is irrelevant from the logical point of view.

To represent the transition function, another vector \( x' \) of BDD variables, called next state variables, is allocated in the BDD package; these variables are used to represent the states resulting from transitions triggered by actions. We write \( \xi'(s) \) for the representation of the state \( s \) in the next state variables. With \( \xi'(Q) \) we denote the construction of the BDD corresponding to the set of states \( Q \), using each variable in the next state vector \( x' \) instead of each current state variables \( x \). We require that \( |x| = |x'| \), and assume that variables in equal position in \( x \) and in \( x' \) correspond. We define the representation of a set of states in the next variables as \( \xi'(s) \doteq \xi(s)[x/x'] \). We call the operation \( \Phi[x/x'] \) “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 \( \Phi[x'/x] \) is called “backward shifting”. In the following, we call \( x \) current state variables to distinguish them from next state variables \( x' \). 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); thus, it is represented as an assignment to \( x, a \) and \( x' \). For the robot navigation domain, the transition corresponding to the application of action \( \text{GoEast} \) in state \( (0, 0) \), resulting in state \( (1, 0) \), is represented by the following BDD

\[
\xi([(0,0), \text{GoEast}, (1,0)]) \doteq \xi((0,0)) \land \text{GoEast} \land \xi'(1,0))
\]

The transition function \( R \) of the automaton corresponding to a planning domain is simply a set of transitions, and is thus represented by a BDD \( R(x, a, x') \) in the BDD variables \( x, a \) and \( x' \), where each satisfying assignment represents a possible transition:

\[
\xi(R) \doteq \text{SEQ}(a) \land \bigvee_{t \in R} \xi(t)
\]

The \( \text{APPL}(x, a) \) BDD representing the applicability relation can be directly obtained with the following computation.

\[
\text{APPL}(x, a) \doteq \exists x'. R(x, a, x')
\]

The BDD representing the states reachable from \( Q \) by executing an action \( a \in A \) is directly obtained by means of the following computation (where we write \( Q(x) \) to stress the fact that the BDD representing \( Q \) is over the state variables in \( x \)):

\[
\text{Exec}(a, Q(x)) \doteq \exists x. \exists a. (R(x, a, x') \land Q(x) \land \xi(a))[x'/x].
\]

The BDD representing the states reachable from \( Q \) in one step is obtained with the following computation:

\[
\exists x. \exists a. (R(x, a, x') \land Q(x))[x'/x].
\]
Let us consider now the symbolic encoding of observations. We introduce a new set \( y \) of BDD variables, called *observation variables*, distinct from state and action variables. Similarly to what we have shown for state variables, we associate each observation \( o \in O \) to a complete assignment to observation variables \( \xi(o) \), that is a BDD in the variables \( y \). A set of observations \( \{o_1, \ldots, o_n\} \) is associated to the BDD \( \xi(o_1) \lor \cdots \lor \xi(o_n) \).

**Example 5.1.** In a \( 4 \times 4 \) robot navigation grid, each observation corresponds to a wall configuration around the robot, for a total—in principle—of sixteen combinations. (In the specific case of the domain in Fig. 14, twelve observations can actually occur: nowall, e, s, ne, nw, sw, se, ew, ns, nsw, nwe, swe, and with the obvious interpretation that nsw corresponds to a wall north, a wall south, and a wall west, and similarly for the others.) The observation set can be represented by means of 4 observation variables \( \text{WALLW}, \text{WALLE}, \text{WALLN}, \text{and WALLS} \); each of these is intended to be true if wall is perceived in one of the directions. The correspondence between observations and assignments to observation variables is straightforward: for instance, \( \xi(\text{nsw}) \) is the (complete) assignment \( \text{WALLN} \land \text{WALLS} \land \text{WALLW} \land \lnot \text{WALLE} \). Sets of observations can be represented similarly: \( \xi(\{\text{nsw}, \text{ns}\}) \) is the partial assignment identified by the BDD \( \text{WALLN} \land \text{WALLS} \land \lnot \text{WALLE} \). We notice again the impact of the symbolic representation, that enables for the manipulation of sets of observations. For instance, the \( \lnot \text{WALLN} \) BDD represents the complement set of observations \( \{\text{s}, \text{w}, \text{e}, \text{sw}, \text{se}, \text{we}, \text{swe}, \text{nowall}\} \), while \( \lnot \text{WALLN} \) represents the complement set of observations \( \{\text{n}, \text{nw}, \text{ne}, \text{nsw}, \text{nse}, \text{nw}, \text{nsw}\} \).

For the symbolic representation of the observation function, we consider that \( X \) is a function over \( O \times S \), and it is therefore possible to proceed as in the case of the transition function. We represent each state-observation pair \( (s, o) \) with \( \xi(s) \land \xi(o) \), that is a BDD in the \( x \) and \( y \) variables. The representation of the observation function is therefore the BDD

\[
\xi(X) \doteq \bigvee_{(s,o) \in X} \xi(s) \land \xi(o)
\]

Rather than a monolithic representation of the observation function, we adopt a partitioned approach, and we construct, for each observation variable \( y \), the two sets of states \( X^-(y) \) (the states compatible with \( y \)), and \( X^-(\overline{y}) \) (the states compatible with \( \lnot y \)). This can be done by means of the following symbolic operations:

\[
\xi(X^-(y)) \doteq \exists y. (\xi(X) \land y) \quad \xi(X^-(\overline{y})) \doteq \exists y. (\xi(X) \land \lnot y)
\]

Notice that, when no noisy sensing is present, there is no need to introduce two distinct \( X^-(y) \) and \( X^-(\overline{y}) \), since \( X^-(\overline{y}) = S \setminus X^-(y) \). However, in the case of noisy sensing, for some observation variable \( y \), \( X^-(y) \cap X^-(\overline{y}) \) may be nonempty.

**Example 5.2.** Fig. 15 depicts the information associated to sensors for the domain in Fig. 14. In the first row, each of the pictures shows the information associated to each of the observation variables \( \text{WALLN}, \text{WALLS}, \text{WALLW}, \text{and WALLE} \). (A triangle is meant to represent the “echo” of a sensor in one of the directions.) For the first three variables, sensing is not noisy, and a partition of the state space is induced: for instance, \( X^-(\text{WALLW}) \) is the set of all cells without the triangle, while \( X^-(\text{WALLE}) \) is the complement set \( S \setminus X^-(\text{WALLW}) \) containing the cells with the triangle. In the case of \( \text{WALLE} \) (fourth picture), we assume noisy sensing in the four central cells; grey triangles represent the overlapping between \( X^-(\text{WALLE}) \) and \( X^-(\text{WALLE}) \); \( X^-(\text{WALLE}) \cap X^-(\text{WALLE}) = \{(1, 1), (2, 2), (2, 1), (1, 2)\} \). The rightmost picture depicts the case where the observation \( \text{WALLE} \) is not noisy, and thus a partition of the state space is induced as for the other three observations.

The second row shows the information associated to pairs of observation variables (assuming \( \text{WALLE} \) to be noisy). Each of the shapes in a cell corresponds to a different assignment to observation variables. For instance, in the leftmost picture, we see that \( \text{WALLN} \) and \( \text{WALLS} \) partition the state space in the four different classes \( \{0, 0\}, \{2, 0\}, \{2, 3\}, \{2, 2\}, \{1, 0\}, \{3, 0\}, \{1, 2\}, \{2, 2\}, \{0, 3\}, \{3, 3\}, \{1, 1\}, \{3, 1\}, \{0, 2\}, \{3, 2\}\). In the rightmost picture, we see that \( \text{WALLE} \) and \( \text{WALLW} \) identify the following sets \( \{0, 0\}, \{2, 0\}, \{0, 1\}, \{2, 1\}, \{1, 2\}, \{1, 3\}, \{1, 0\}, \{3, 0\}, \{1, 1\}, \{2, 2\}, \{3, 2\}, \{3, 3\}, \{2, 1\}, \{3, 1\}, \{0, 2\}, \{1, 2\}, \{0, 3\}, \{2, 3\}, \{1, 1\}, \{2, 2\}\).

We notice that, due to noisy sensing, some classes overlap.

The third row shows the information associated to triples of observation variables. Finally, the information associated to the combination of all the observation variables is depicted in the last row, in the case of \( \text{WALLE} \) being noisy.
Fig. 15. The information associated to observation variables in the robot navigation domain of Fig. 14.

(left), and not noisy (right). We see that some states are indistinguishable (e.g. \((0, 0)\), \((2, 0)\) and \((1, 3)\), or \((3, 1)\) and \((0, 2)\), or \((1, 0)\) and \((3, 0)\)) even in the more informative case of non-noisy sensing (right). We also see that some states (e.g. \((2, 1)\), \((1, 2)\), and \((2, 2)\)) can be uniquely identified, but this is no longer the case when noisy sensing is considered (left): for instance, \((2, 2)\) can no longer be distinguished from \((3, 3)\), and \((1, 2)\) can not be distinguished from \((0, 3)\).

In the rest of this paper, we assume that the BDD representation of a planning problem is given. In particular, we assume as given the vectors of variables \(x, x', a\) and \(y\), the encoding functions \(\xi\) and \(\xi'\), and we simply call \(S, R, \mathcal{X}, I\) and \(G\) the BDD representing the states of the domain, the transition functions, the observation function, the initial states and the goal states, respectively.

5.3. Symbolic search in the belief space

In symbolic model checking, BDDS provide a way for compactly representing and efficiently expanding the search frontier for the exploration of the space of states. This machinery is used, with small variations, in [37], where conditional plans are constructed by symbolic breadth-first search in the space of states under the hypothesis of full observability. However, this machinery can not be applied to tackle planning under partial observability. Our approach is rather based on the fact that a belief state \(B_\mathcal{S} \subseteq S\) is directly represented by the corresponding BDD \(B_\mathcal{S}(x)\). The basic step of the algorithms is the expansion of a belief state, that results in a list of newly expanded nodes.
Given a belief state \( B_s \), \textsc{FwdExpandBS}(B_s)\) computes the set of belief state action pairs \( \langle B_{s1}, a_i \rangle \) such that \( a_i \) is applicable in \( B_s \) and \( \text{Exec}(B_s, a_i) = B_{s1} \). This computation can be performed symbolically as follows:

\[
\text{FwdExpandBS}(B_s(x)) \doteq (\exists x. (B_s(x) \land R(x, a, x') \land (\forall x. (B_s(x) \rightarrow \text{Appl}(x, a)))))[x'/x].
\]

The result is a BDD, in variables \( x \) and \( a \), whose satisfying assignments represent the belief state action pairs \( \langle B_{s1}, a \rangle \).

To apply observations starting from a belief \( B(x) \), we exploit the representation of observations in terms of the observation variables \( y \), and the BDD-based representation of the states \( \mathcal{X}^\sim(y), \mathcal{X}^\sim(\tilde{y}) \).

To restrict \( B(x) \) to the states compatible with a given value of the observation variable \( y \), it is enough to conjoin it with the BDD representing \( \mathcal{X}^\sim(y) \) or \( \mathcal{X}^\sim(\tilde{y}) \). To restrict it to the states compatible with an observation, it is enough to conjoin it with the BDDS associated to the assignments of the observation variables for that observation.

Notice that the symbolic representation helps us to reduce the branching associated to observations. In fact, rather than extending the plan with a decision on the occurrence of an observation, we consider the partition induced by the value of observation variables. We use the following \textsc{ExpandObsVar} primitive, that returns a pair of beliefs, given a starting belief:

\[
\text{ExpandObsVar}(Bs(x), y) \doteq \langle Bs(x) \cap \mathcal{X}^\sim(y), Bs(x) \cap \mathcal{X}^\sim(\tilde{y}) \rangle.
\]

Considering plans which branch on the value of an observation variable amounts to branching on sets of observations: the plan \( \text{if } y \text{ then } \pi_1 \text{ else } \pi_2 \) can be seen as a (much more compact) representation of the plan \( \text{if } o_1 \text{ then } \pi_1 \text{ else if } o_2 \text{ then } \pi_1 \text{ else } ... \text{if } o_n \text{ then } \pi_1 \text{ else } \pi_2 \), where \( o_i \) are the observations compatible with \( y \). In fact, the effect of this representation is even more striking when we consider that it represents all the plans of the above form, for different orderings of the \( o_i \). Notice also that this representation is complete, i.e. every plan branching on an observation \( \text{if } o \text{ then } \pi_1 \text{ else } \pi_2 \) can be represented by a plan branching on observation variables.

**Example 5.3.** Consider the robot navigation in Fig. 16, where the initial states are \( \{(1, 2), (1, 3), (2, 2), (2, 3)\} \) and the goal is \( \{(3, 1), (3, 2)\} \). Let us call \( \pi_2 = \text{GoEast} \circ \text{GoNorth} \) and \( \pi_1 = \text{GoEast} \circ \text{GoEast} \circ \text{GoNorth} \). Then the plan \( \text{if } \text{WALLW then } \pi_1 \text{ else } \pi_2 \) is a strong solution for the problem. Since WALLW corresponds to the set of observations \( \{\text{nw, sw, ew, nsw, nwe, swe}\} \) the corresponding plan in terms of observations is significantly more complex:

\[
\begin{align*}
\text{if } \text{nw then } \pi_1 \text{ else if } \text{sw then } \pi_1 \text{ else } ... \\
\text{if } \text{nwe then } \pi_1 \text{ else if } \text{swe then } \pi_1 \text{ else } \pi_2
\end{align*}
\]

In terms of the implementation, we exploit BDD technology in a fundamental way, to directly link the symbolic machinery with the search framework. Given the canonical form of BDDS, a belief state is simply a pointer to the unique corresponding BDD in the BDD package (see Fig. 17). The (BDD pointers representing) beliefs generated during the search are stored by means of a hash table, external to the BDD package; this is accessed directly using BDD pointers as keys. In this way, it is very easy to associate additional information to belief states (e.g., the corresponding plan, heuristic information, or tag values such as those described in Section 4). There are several algorithm phases that benefit of these features, in our implementation. In particular, we observe that a solution for a given search node \( n \) can be reused for any node associated to the same belief of \( n \). Since equivalence between beliefs represented as BDDS can be performed in constant time, reuse of subplans is a simple and effective optimization that is applied.
Fig. 17. A symbolic representation of belief states for the robot navigation domain.

```
1 function EXTRACT_BSACTION_PAIRS(B, StateVars, action)
2     if (B = false) then
3         return Φ;
4     fi
5     actvar := BDD_VAR(B);
6     if (actvar ∉ StateVars) then
7         bsp1 := EXTRACT_BSACTION_PAIRS(BDD_GETTHEN(B), StateVars, action ∧ actvar);
8         bsp2 := EXTRACT_BSACTION_PAIRS(BDD_GETELSE(B), StateVars, action ∧ ¬actvar);
9     else
10         return bsp1 ∪ bsp2;
11     fi
12 end
```

Fig. 18. The extraction of belief-action pairs.

when expanding a node, and when propagating bottom-up the success of a node. An additional optimization can be applied by considering that a solution for a node \( n \) is also a solution for any node associated to a belief entailed by that of \( n \), and exploiting the effective implementation of the BDD primitive for entailment check. Moreover, in certain cases it is possible to effectively establish the failure of a node based on the failure of belief-equivalent nodes, or of nodes associated to entailed beliefs (by also comparing the paths associated to the nodes). Although these additional optimizations are present in our implementation, their effectiveness may vary depending on the domains, and for the purposes of this paper, only the reuse of solution over belief-equivalent nodes is considered.

The symbolic machinery is also crucial to effectively transform a BDD \( B'(x, a) \) in the state and action variables, representing the result of an expansion of a belief \( B(x) \), into a list of pairs, each pair coupling one action (a BDD in the variables \( a \)) to the belief state (a BDD in \( x \)) resulting from the expansion of \( B \) via that action. This operation is necessary for the planning algorithm to deal separately with the actions and their results, in order to construct the tree representing the portion of search space being visited. This is efficiently performed by the `EXTRACT_BSACTION_PAIRS` routine depicted in Fig. 18, recursively traversing the action variables in \( B'(x, a) \) and identifying the BDDS in the \( x \) variables associated to the leaves of such traversal.

For the sake of clarity, let us assume that action variables precede state variables in the BDD ordering, and let us not consider observation and next state variables. The routine interprets as a belief state every node associated with a state variable, having a parent node associated with an action variable, and associates to it the action traversed so far, i.e. the set of assignments to the action variables accumulated during the traversal (in the top level call, this assignment
must be set as true). If the level of state variables has not been reached yet, the routine recurs on the “if” and “then” branches of the BDD, accumulating the respective assignments to the current action variables.

6. Search strategies

It is well known that different search strategies may have a dramatic impact on the effectiveness of the search. In this section, we discuss several possible search strategies for the algorithm presented in Section 4. We remark that the algorithm retains all the properties (termination, correctness and completeness) for all the strategies that correctly implement the EXTRACT NODE FROM FRONTIER primitive. Also, we will not focus on strategies that guarantee optimality.

6.1. Basic strategies

The first strategy we consider is Depth-First Search, which be easily implemented within our framework by means of a stack-based management of the list of open nodes. In this way, we can simulate the algorithm presented in [9]. The converse is not possible, since the algorithm in [9] implements a depth-first strategy in a hardcoded manner, and can not simulate other search styles. We consider two variants of this strategy: in the first, FIX, the order in which the effect of actions and observations are expanded is fixed throughout the search. In the second, RND, the choice is randomized by shuffling the results of the expansion primitives. The main problem of these strategies is that, because of a bad choice, the search may get trapped into a dead-end, i.e. a portion of the search space from where the goal is unreachable.

A possible alternative is to consider some heuristic measure to decide which node should be expanded next. Such an attempt was made in an earlier version of this work [7], where the node to be expanded is selected, among all the open nodes, according to the following function: higher score is given to nodes with (i) highest number of open nodes with the same belief state, and (ii) least number of failed nodes with the same belief state. This heuristic turned out to be not effective for at least two reasons. First, it does not take into account any distance to the goal. Second, it disregards structural information (e.g. the depth of the nodes, or the number of open nodes for the plan associated to the node). In the rest of this section, we present some new, more advanced search strategies.

6.2. Strategies based on strong distance

The Strong Distance heuristics (SD in the following), proposed in [36] in the setting of conformant planning, is a way to estimate the distance between a belief state and the goal in a nondeterministic domain. Intuitively, the SD heuristics is based on the relaxation of the problem to the case of full observability: for each individual state in the belief state, we denote with $SD(s)$ the shortest worst-case (for all possible nondeterministic action effects) distance to the goal; for a belief $B$, we write $SD(B) = \max_{s \in B} SD(s)$.

In the first strategy based on Strong Distance, called SD PURE, we select, among all open nodes in the frontier, the one with lowest strong distance to the goal. We call this strategy SD PURE, since it resembles “pure” search, where the cost of reaching the node is not taken into account.

With SD PURE, the search does not guarantee that a given plan is expanded until its failure or success. This can be often undesirable: for instance, after a plan branch resulting from an observation is discovered to be successful, this strategy may decide to expand a branch of a different plan prefix, wasting the previous effort. With the strategy SD, we select the best node (according to SD) among the ones that have been produced by the last expansion step.

Although the SD heuristic may be very informative, it may still be too coarse an approximation for a search in the belief states. In fact, it disregards the amount of knowledge associated to the search nodes. Thus, the search greedily proceeds towards the goal, even when the necessary degree of knowledge has not been gathered. This is the case, for instance, when a fluent $f$ has several values in a belief state, but only a specific value in the goal. (See [36] for a more thorough discussion.) Therefore, we propose the SD OBS strategy, that is a further refinement of SD, where we favour information gathering by giving priority to nodes resulting from observations over nodes resulting from observations over nodes resulting from...

---

2 The actual order depends on the order of actions in the input file and on the actual BDD encoding.
actions. Notice also that one action can always be postponed to observing, while relevant information can be no longer available after acting.

6.3. Strategies based on fluent distance

The exact computation of the SD function can be a serious bottleneck in the case of large domains, even when exploiting symbolic techniques (see [37] for the details). For this reason, we propose the FD strategy, that is basically a modification of SD0BS, where Strong Distance is replaced by another function, the Fluent Distance, that is often much simpler to compute. Basically, the Fluent Distance of a belief state from the goal is the sum, over each of the fluents \( f \) in the domain, of its \( f \)-distance to the goal.

We say that \( v \) is a possible value for a fluent \( f \) in a belief state \( B \) if \( f \) has value \( v \) in some state \( s \) belonging in \( B \). Then, the \( f \)-distance for \( B \) to \( G \) is defined as follows: if the number of possible values of \( f \) in \( G \) is less than the number of values of \( f \) in \( B \),\(^3\) then \( f \)-distance for \( B \) to \( G \) is set to a high, threshold value \( M_f \). Otherwise, the distance is based on the following construction. Let \( LB[-] \) be the sequence of sets of values for \( f \), such that \( LB[0] \) is the set of values of \( f \) in \( G \), and \( LB[i+1] \) is the union of \( LB[i] \) with the values of \( f \) in the preimage of \( LB[i] \). Then, the \( f \)-distance of \( B \) from \( G \) is the least \( i \) such that \( LB[i] \) contains all the possible values of \( f \) in \( B \).

Some remarks are in order. We notice that the iteration is guaranteed to terminate, since the range of values for \( f \) is finite. The threshold value \( M_f \) is actually the number of iterations needed to reach a fix point. The sequence of \( LB \) layers can be computed by means of symbolic operations; in particular, the range of \( f \) in \( B \) is basically the results of the quantification in \( B \) of all the fluents but \( f \). The computation is much easier than the strong distance, since the fluents are analyzed independently, and the accumulated BDDS contain a much lower number of variables. The layers can be computed once and for all before the search is started.

6.4. Strategies based on dynamic target selection

The last strategy we propose, called FD\(_{DT}\), is an extension of FD based on the idea of Dynamic Target selection. The idea is to “guess” a subset of \( G \) as a current target, say \( t_i \), and to evaluate each node based on the fluent distance to \( t_i \) rather than to \( G \). Given that the range of each fluent in \( t_i \) is likely to be smaller than in \( G \), we hope that the corresponding fluent distance is somewhat more informative.

The search is an enforced hill climbing on the selected target: once a node with distance \( N \) is expanded, the next node to be expanded will one of its sons that has either distance which is lower or equal to \( N \). When it is impossible to decrease the distance, a new target is guessed, based on the idea that the selected target may not be informative enough.

In order to mitigate the risk of a bad target, we exploit the idea of invariant fluents. Intuitively, a fluent is invariant in a domain if its value can not be modified by any action.\(^4\) Invariant fluents are often necessary to model hidden structural information (e.g. the status of a fault variable in a plant). A fluent \( f \) can be detected to be invariant by means of simple symbolic operations.\(^5\)

Once invariants have been discovered, any selected target is a subset of the goal containing a selected invariant configurations that is possible in the current belief. The use of invariants also provides a sufficient condition for a belief state \( B \) to be unsolvable. This is the case when \( B \) contains a configuration of invariants which is not present in \( G \).

7. Experimental evaluation

In this section, we experimentally evaluate the algorithm and the search strategies proposed in previous sections. We do this by reporting two sets of experiments. In the first, we show the comparison between the different heuristics,

\(^3\) Intuitively, reaching the goal requires reducing the uncertainty over \( f \): if for \( f \) more values are possible in \( B \) than in \( G \), then \( B \) can not be contained in \( G \).

\(^4\) We point out two major differences with respect to the idea of invariants in classical planning. First, here we are dealing with nondeterministic action effects, and our analysis has to take into account all possible outcomes. Second, invariant fluents can not be simplified away to constants, since the same fluent can occur both positively and negatively within the same belief state.

\(^5\) It is enough to check if \( f = f' \) is a logical consequence of the transition relation.
both in terms of nodes and in terms of run-times. In the second, we compare the run times of our planner MBP with a selected search strategy and two other competitor systems.

Both experiments are on a set of over 700 problems, that include most\(^6\) of the established benchmark domains for planning under partial observability. All the experiments were run on a Pentium III Xeon 700 MHz with 6 GB RAM running Linux, fixing a memory limit of 512 MB. The MBP executable and all the models used for the experiments can be found at http://sra.itc.it/tools/mbp/AIJ_PO.tar.gz.

7.1. The MBP planner

The algorithm and the search strategies described in previous sections have been all implemented within the MBP planner [5]. MBP is a system for planning in nondeterministic domains, based on the planning via symbolic model checking paradigm. MBP is implemented on top of the NuSMV symbolic model checker [29], from which it inherits the language and the machinery to describe and explore finite-state nondeterministic systems. The language of NuSMV has been extended with constructs to specify observation variables as boolean relations. In addition, MBP is extended with a symbolic representation of beliefs based on the use of BDDS.

MBP integrates capabilities for plan generation, plan simulation and plan verification for a variety of classes of planning problems. These include strong planning for planning under full, partial or null observability [8,9,37]; strong cyclic planning under full observability [34]; planning for temporal and intentional goals under full observability [39,76]. For most of these problems, more than one algorithm is available, e.g. exploiting regression vs. progression techniques. MBP is also the basis for the reactive planning platforms described in [1,11].

7.2. The test domains

We considered several test domains. Most of them are parameterized, in order to show the asymptotic behaviour of the different search strategies and systems.

7.2.1. Empty room

This is a robot navigation domain consisting of an empty square room of size $N \times N$, surrounded by border walls. A robot may start from any position in the room, and has to reach the center of the room. The robot is equipped with four wall proximity sensors, one for each direction, and may be moved in the four directions.

7.2.2. Maze

The maze domain we consider is taken from [9,86] and adds some complexity to the empty room domain, by having walls also inside the room, while maintaining the same actions and sensing scheme. Considering connected mazes whose topmost left corner is reachable, the problem is again that of positioning the robot in that corner, given that its initial position is unknown.

For this domain, we consider mazes of increasing size; for each size, we consider a set of 16 randomly generated mazes, differing for the way the walls are positioned and density of the walls in the room.

7.2.3. Ring

The ring domain, first described in [35], is a different navigation domain, where a ring of $N$ rooms can be traversed by a robot. Each room features a window, which can be either open, closed or locked; the robot can observe the status of the window in its current room, and close it (if it is open) or lock it (if it is closed). Initially, the windows can be in any state, and the goal is to have all of them locked.

7.2.4. Bomb in the toilet

This domain follows the formulation presented in [88]: the domain consists of a number of toilets, and a number of packages. Packages may contain bombs that can only be disarmed by dunking them into the toilets. Once a package is dunked into a toilet, the toilet gets clogged, and no packages can be further dunked into it prior to flushing (which

\(^6\) The benchmark domains were chosen to be able to compare all systems on every domain.
unclogs the toilet). A sensing action, “sniff package”, is available to tell whether a certain package contains the bomb. The goal is to disarm the bomb that is contained in one of the packages, its position being initially unknown.

7.2.5. Medical

In the medical domain [88], the goal is to heal a patient that suffers from one of many diseases. For each disease, a curing action is available; but if a wrong cure is applied, it causes the death of the patient. A number of boolean sensing actions (“medical exams”) are also available, whose results split the possible diagnoses in two subsets. Notice that solutions to this problem basically consist in first identifying a perfect diagnosis, that is fully resolving the state uncertainty, and only then applying a single curing action.

7.2.6. Power supply restoration

The Power Supply Restoration domain was first presented in [84], and then tackled in [10,28]. A network is composed of electrical lines, connected by switch devices in a generic topology; some open line end is fed by power engines through breakers. When a short-circuit (fault) takes place on some line, the breaker feeding that line opens to protect the power engine; notice that in doing so, also other non-faulty lines may be left without power. Faults can not be directly observed; however, each switch and breaker is equipped with a directional sensor that signals whether there has been a fault “downstream” the power flow, since the last time the device has been traversed by power. Switches and breakers can be operated by opening and closing them. The goal is that of feeding every line that can be fed; this can be achieved under the proviso that devices and sensors are reliable.

We experiment for a fixed, medium size network, consisting of 6 lines, 3 breakers and 7 switches (see [10,28]). We report results for two different kinds of problems: when the number of faults in the network is stated to be equal to a given number (denoted with PSR Eq), and when it is lower than or equal to a given number (denoted with PSR Leq).

7.3. Comparison between search strategies

7.3.1. Empty room

The results for this set of tests are shown in Fig. 19. We observe that the FDDT and Fd strategies are the ones that work best, and perform similarly. FIX is also able to scale up reasonably well, while the others performs rather poorly.

As expected for SDPURE and SD, strong distance alone may be a very misleading measure for search in the belief space: the search repeatedly directs the robot to the center of the room before its position has been actually localized, which leads to a large number of long backtracking instances. FDDT and Fd behave well, since they demote nodes with high uncertainty, and try to gather as much knowledge as possible by means of observations before aiming at the goal position.

It is interesting to compare the behavior of FIX and RND: in this domain, the fixed ‘preference order’ among actions used by F1X seems to result in a systematic visit of the room, while RND results in complex, “curly” trajectories that fail because of loopbacks.

Fig. 19. Results for the Empty Room domain.
7.3.2. Maze

The results are presented in Fig. 20. In order to be able to move the robot satisfying the action preconditions, the search often ends up performing enough observations to localize the robot position. In fact, most of the search time seems to be spent in the generation of the first localization subplans. Once this is done, the success propagation mechanism works in such a way that increasingly large areas of the maze are marked as solved—and once a few paths in the maze are detected as successful, there is a high chance that, from anywhere in the maze, a few moves lead to a “successful” cell.

The problem has a rather constrained nature: undirected search such as FIX and RND scale up well, similarly to the more informed FD and FdDT. We also notice a difference in performance, since FD is heavier to compute than FdDT.

Strategies based on Strong Distance are less effective in terms of expanded nodes, and also heavier to compute. The difference between SDPUPURE and Sd is due to the fact that SDPURE analyses the whole search frontier, and may decide to expand branches of unrelated plans, therefore delaying the effective identification of a solution.

7.3.3. Ring

The results for this comparison are shown in Fig. 21. The most effective search is obtained by the most informed FD and FdDT: they tend to gather knowledge as much as possible, their search seems to be directed well enough to achieve a good performance, expanding a limited number of nodes.

FIX is also surprisingly effective: using a fixed preference order in FIX leads to a systematic traversal of the state space, which turns out to be convenient to avoid long loops. The randomized RND performs very poorly, since extremely long belief sequences may be traversed before encountering a loop, or meeting the goal. SDPURE is much
more effective than SD: considering the whole set of open nodes enables to avoid high penalties imposed by a bad choice.

In terms of CPU time, we notice that the computation of strong distances in S^D PURE and SD is rather expensive—this explains the relative degrade for S^D PURE. Again, we see that F^DDT is less expensive than FD.

7.3.4. Bomb in the toilet

The results for this comparison are shown in Fig. 22 (we report the results for the case with 20 toilets; the other cases are qualitatively similar). We see that the behavior in this domain is also rather constrained, and all the search strategies behave reasonably well. The only exception is S^D PURE, which analyses the whole frontier at each step for selecting a node to expand. Given the little information provided by strong distance, this results in a breadth-first search, with a strong degrade in performance.

7.3.5. Medical

The results for this comparison are shown in Fig. 23. For this domain, F^DDT and FD are substantially superior to all the others, for their ability to gather knowledge, and to approximate distances. In this problem, it is possible to somministrate medicines even after the patient is dead. This makes a large portion of the reachable belief state to be unsolvable. This is a serious problem for RND and FIX, which often end up in these dead-end areas. SD does not scale up well, since the relaxation of the problem to full observability provides little guidance; however, thanks to the backward computation of strong distance, it is possible to detect unsolvable belief states, from which the advantage over RND and FIX. S^D PURE is significantly better than SD in terms of expanded nodes, but the evaluation of all the open nodes results in a strong penalty in terms of CPU time.
7.3.6. Power supply restoration

The results for this comparison are shown in Fig. 24. In this domain, the solution for these problems requires setting the switches in configurations that depend on the (initially unknown) fault position. Therefore, it is important to gather knowledge before attempting to restore the network. For this reason, strong distance alone does not seem to drive the search effectively.

Furthermore, such initial conditions are likely to feature a mix of open and closed switches, and using a fixed action preference leads as in Fix leads to aiming at “all-open” or “all-closed” configurations, which are unlikely to be satisfactory. This causes a large number of computationally expensive backtracking instances. Instead, the randomized search RND scale up reasonably well. Fd and FdDT are also very effective. The dynamic target mechanism of FdDT basically partitions the problem, and is able to exploit distance-directed search in each subproblem.

7.3.7. General remarks

It is clear from the experiments that the effectiveness of the search in MBP strongly depends on the implemented search strategy. In particular, uninformed DFS search may perform reasonably well either where the domain is rather constrained (such as e.g. the Maze domain), or where a large set of solutions exist (such as e.g. the Empty Room, the Ring or the Bomb in the Toilet domains). For some domains, such as e.g. the Empty Room, choosing a fixed preference order amongst actions can lead to better performance than randomizing, since this may prevent forming long self-loops. On the other side, using distance alone is not enough in a setting where the degree of uncertainty plays a key role; as such, the Strong Distance Heuristics fail to scale up in most cases. Heuristics which favour knowledge gathering, and combine this idea with the usage of approximated distance measures, seem to work better, especially when problems can be partitioned by detecting selected targets inside the goal (like in the PSR domain).

Additional considerations can be drawn on the structure of plans. For domains such as the Maze, featuring many constraints on action executability and a rather rich sensing, knowledge gathering is (easily) solved in the first plan steps, and thus plans feature a limited branching factor. Plans found for $51 \times 51$ mazes feature around 500 branches,
and an average depth of 300 actions. In other domains, such as Medical, the core of the problem consists in knowledge gathering; in these cases, solution plans are shallow and feature a bigger branching factor. For instance, plans in the Medical domains are only 3 actions deep, but feature a number of branches equal to the number of possible illnesses in the problem.

Also, we remark that the success reuse mechanism in the search algorithm induces an internal plan representation as a DAG, rather than a tree. The effect of this is evident in domains such as Empty Room, Maze, or Ring, where the number of nodes of the internal plan representation is only a small portion of what would be if a tree representation was adopted instead. For instance, the FDDT-generated DAGs for plans for a $51 \times 51$ feature around 2500 nodes, while their correspondent trees would feature over $5 \times 10^5$ nodes.

7.4. Competitor planners

We compared MBP using FDDT with the two most advanced competitor systems, GPT [16,17] and JUSSIPOP [81]. We do not directly include other relevant systems such as SGP [88], CASSANDRA [73], YKA [79] and QBfPLAN [77]. An indirect comparison with SGP and CASSANDRA, showing they are outperformed by GPT and MBP, can be derived from [9]. JUSSIPOP consistently improves over YKA and QBfPLAN (personal communication by Jussi Rintanen).

For each of the systems, we attempted to maximize the performance by using existing hand-crafted encodings, and acquiring direct information from the authors about optimizing the encodings and the behavior of their planner.

For each of the planners, we used a fixed setup throughout the whole evaluation. The default parameters have been adopted for every planner. For GPT and JUSSIPOP, this follows the authors’ indication. For MBP, this means that the reuse of failures and successes via equivalence classes is enabled, and the remaining optimizations are disabled. This setup represents a reasonable trade-off between the overhead introduced by the computation of the optimizations, and their induced speed-up. Indeed, while reusing failures and successes by equivalence classes can be perceived as generic search controls, crucial to inhibit useless expansions of relevant portions of search space, the remaining optimizations can be more or less effective depending on the features of the domain.

7.4.1. JUSSIPOP

JUSSIPOP is a regression planner for nondeterministic, partially observable domains, based on the ideas of YKA. It relies on BDD-based representation of observational classes, which are constructed as a pre-processing of the domain. The search exploits such observational classes to recombine beliefs (also represented via BDDs), and is driven by a cardinality heuristic that tries to achieve the maximal belief: when progressing backward, the intuition is that the larger the belief, the more likely it is that the initial situation is included, and a solution is found. The input language of JUSSIPOP is an extension of PDDL that allows for nondeterministic action effects and multiple initial states; observability is modeled by declaring a subset of state variables to be observable.

7.4.2. GPT

GPT models planning domains as Markov Decision Processes, and, based on the probability distributions, produces an optimal policy associating actions to belief states. The search in GPT exploits the Labeled-RTDP algorithm described in [17]; for partially observable domains such as those we are interested in, the search is performed in belief space (for POMDPs). LRTDP is an optimal and complete algorithm for these models, computing a partial optimal policy closed with respect to the initial (belief) state.

GPT adopts an extension of PDDL (different to JUSSIPOP’s) that allows describing sensing actions, nondeterministic (also probabilistic) action effects, multiple initial states, and functions, which can either be defined by PDDL operators, or computed by external C++ modules.

Notice that the problem solved by GPT is quite different from that tackled by MBP and by JUSSIPOP. First, GPT produces a (memoryless) policy, rather than a conditional plan. Second, it solves an optimization problem. These differences must be kept in mind when evaluating the experimental results. Also notice that, differently from JUSSIPOP and from our implementation, GPT does not exploit a symbolic representation of beliefs; indeed, it explicitly manipulates each state in a belief.
7.5. Comparison between planners

7.5.1. Empty room

The results are reported in Fig. 25, for increasing room sizes. This domain appears not particularly difficult for MBP and JUSSIPOP, while the behavior of GPT degrades rapidly, due to the increasing number of initial states for which a policy has to be computed, and to the exponential number of optimal policies in the problem. While MBP and JUSSIPOP both rely on symbolic, BDD-based representations to avoid explicitly dealing with each single state, GPT does not, which also explains the differently scaling behaviors. Notice that MBP and JUSSIPOP exhibit a similar curve, with MBP consistently outperforming JUSSIPOP by about two orders of magnitude. The difference between the performances of MBP and JUSSIPOP, in this case, may be explained by the fact that JUSSIPOP’s problem representation is constrained to adopting boolean fluents, which is not convenient to represent N-valued coordinate positions, leading to an unnecessarily high number of fluents in the domain.

7.5.2. Maze

The results reported in Fig. 26, left part, present average CPU times for each maze size for the three systems. For MBP and JUSSIPOP, the evaluation of averages is simple: MBP terminates all tests within the limit, and for a given size of the problem, JUSSIPOP either completes all the tests, or does not complete any of them. The situation is more complex for GPT, which presents a high variance in his behavior on mazes of the same size, and completes every sample only up to size 33. In this case, the averages from size 35 onward only refer to the completed tests. The results reported in the two scatterplots in Fig. 26, right part, report a pairwise comparison between MBP and the two competing systems; they give a more detailed view of the results. Here, as it is standard for scatterplots, overtimes are reported on the x and y axis.

Interestingly, the hardness of this domain seems comparable to that of the empty room for MBP, while it seems much simpler than the empty room for GPT, and much harder for JUSSIPOP.

Our intuition for the improved GPT behavior w.r.t. the empty room is that the number of initial states is lower, and the number of optimal plans to reach the desired situation can only be exponential in very particular maze topologies,
and is in many cases rather limited. For most cases of large mazes, GPT is capable of tackling the problem within the
given time limit (even though less efficiently than MBP, due to the explicit state handling).

We believe that JUSSIPOP’s degraded performance is largely due to regression: the number of possible different
observational classes that result from backward composition via observations can grow extremely vast. Furthermore,
due to the fact that observations must be represented as “observable state variables”, the results of observations after
executing one action must be computed together with the effects of the action. While for a domain for a simple
structure such as the empty room this introduces no great complexity, for the maze domain this causes the need to
introduce complex conditional effects into actions, which may make the domain very hard to analyze for the planner.
As a result, JUSSIPOP seems to exhibit an exponential behavior, while MBP maintains the polynomial behavior shown
for the empty room domain.

7.5.3. Ring

The results for this domain are reported in Fig. 27, for different values of N. This problem seems very simple for
MBP, but hard for GPT (due to the huge number of states that have to be explicitly dealt with) and JUSSIPOP (probably
due to the fact that the cardinality heuristic is not informed enough to direct the search effectively).

7.5.4. Bomb in the toilet

The results (for the problem instances with toilets and packages ranging from 1 to 30) are reported in Fig. 28. Each
graph refers to a fixed number of toilets; on the x axis, the number of packages is reported. In Fig. 29 the corresponding
scatterplots are also reported. The results are qualitatively similar to those obtained for the empty room domain: the
problem seems not particularly difficult for MBP and JUSSIPOP while GPT’s performance degrades rapidly as the size
of the problem increases. Again, our intuition is that this is due to GPT’s explicit state handling, opposed to MBP
and JUSSIPOP’s underlying symbolic engine. Notice that the behavior of MBP and JUSSIPOP are also quantitatively
comparable for the simpler problems, with MBP scaling up more nicely for the harder problems and dominating
JUSSIPOP by more than two orders of magnitude on the bigger instances.

7.5.5. Medical

The results for this domain are presented in Fig. 30, for varying number of illnesses (and related sensing). For
MBP, the problem seems simple, as it scales up quite smoothly. For JUSSIPOP, the problem appears to be quite hard;
one more, this is likely to be due to the high number of observational classes that have to be constructed in the
pre-processing phase. Although GPT seems to scale well, we notice that the time spent in finding a solution largely
depends on the discount factor, a parameter used in building the heuristics. We report the data referring to a discount
factor of 0.9; higher times are obtained with either higher or lower factors. We remark that, while the same 0.9 value
is used in all the test cases, this instability phenomena appears only in this domain.

7.5.6. Power supply restoration

The results are reported in Fig. 31. MBP is able to efficiently solve all the problems, exhibiting an easy-hard-easy
pattern for the problems with exact number of faults, and an easy-hard pattern for the problems with maximal number
of faults. The same behavior is evident for the GPT planner; notice however that the GPT modeling of the problem is
rather that of an optimization problem, whose goal is that of minimizing the number of unfed, feedable lines. JUSSIPOP
Fig. 28. Tests for the BMTC domain.

Fig. 29. Scatterplots for the BMTC domain.

also seem to exhibit a similar behavior, but it times out for the most difficult problems. Notice that, since neither MBP nor JUSSIPOP handle PDDL’s directional axioms, the encoding of problems must “unfold” the ramifications in the consequences of propagating faults; this means that the size of such modelings, which we build via a script, may grow up exponentially (see [85]).
7.5.7. Concluding remarks

The experiments show that MBP is able to deal with all the domains considered in our test, and consistently outperforms both GPT and JUSSIPOP.

The comparison with JUSSIPOP is interesting, given the similarities of the two systems: in fact JUSSIPOP tackles the very same problem of MBP (strong planning under partial observability), and adopts a similar internal representation of beliefs. The most evident difference is in the fact that JUSSIPOP is regressive; this seems to be somewhat of a problem when the backward combination of a large number of observation classes can not be easily represented with symbolic techniques. Moreover, while a forward search ensures that every represented state is legal, a backward search may involve reasoning about illegal states, which can contribute to the overall complexity. Finally, we believe that JUSSIPOP’s simple cardinality heuristic may often provide too little guidance to be effective.

The comparison between MBP and GPT is somewhat harder, since GPT tackles a different problem (identifying optimal, possibly cyclic policies). This may offer a partial explanation to the different scale-up behaviors in some of the domains. More importantly, we believe that GPT suffers from the space explosion problem since it does not rely on a symbolic representation. Notice that for some specific domains, whose recasting as an optimization problem is convenient, or where suboptimal policies are easily identified and pruned, GPT proves extremely competitive.

8. Related work

Different approaches have been devised to address the problem of planning under partial observability in nondeterministic domains. The problem has been deeply investigated in the framework of Markov Decision Processes (MDP) (see [12] for an extensive survey). MDP planning allows for stochastic domains, i.e., nondeterministic domains where probabilities can be assigned to transitions. There are substantial differences with our approach, 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, and algorithms that optimize a utility function are defined on the basis of costs/rewards. While in this paper we propose an algorithm that is guaranteed to return strong solutions, in MDP, whether the solution is strong depends on the probability and cost distribution. For instance, in [53], cycles are not avoided, and solutions are not guaranteed to terminate. On the one hand, the MDP approach pro-
vides 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 crucial to guarantee that the plan will terminate and will lead to the desired states. A further conceptual difference is the fact that in this paper we focus on reachability goals. In MDP, costs and rewards provide the ability to represent goals that are more expressive than reachability goals. The extensions of MDP to express goals in temporal logic in [2,3,87] are somehow paralleled by similar extensions in our framework [6,23].

From the standpoint of complexity, the problem we tackle has been shown to be 2-EXP-complete [80], while policy existence in POMDPs is NP-complete for stationary policies, and PSPACE-complete for time-dependent ones [44,75]. However, from the practical point of view, the expressiveness of the MDP approach is more difficult to manage in the case of large state spaces. Our approach allows us to fully exploit the BDDS-based symbolic model checking techniques to tackle problems of significant size. A lot of work has been done to tackle the state-explosion problem in MDP. Notable examples are based on abstraction, reachability analysis, and decomposition (see [12] for a review). The SPUDD planner [52] tackles MDP problems by making use of Algebraic Decision Diagrams, data structures similar to BDDS. 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. SPUDD is restricted to the case of full observability; some other probabilistic models, instead, are specialized to the case of null observability, i.e., conformant planning, like the work on Conformant Probabilistic Planning [49,59,65].

The more general MDP approach to partially observable domains (POMDP) has also been extensively studied [14, 15,22,32,60,69,70,83] (see also [12] for a more exhaustive list of references). The POMDP approach inherits from MDP the conceptual and practical differences with our work. Moreover, in its general formulation, the POMDP problem is complicated by the fact that belief states are defined as probability distributions over the set of states of the domain, and the set of belief states is not finite. However, practical approaches to this problem have been devised. This is the case of the solution implemented in GPT, the available planner based on MDP which can deal with partial observability. As the experimental results in Section 7 show, MBP outperforms GPT, since an enumerative approach in the case of large state spaces and-or high uncertainty can hardly scale up. However, the experimental comparison should be interpreted very carefully, keeping in mind that POMDP planners solve a different problem. In [57], the classical planner TLplan [20] is extended to the possibilistic/probabilistic conditional planner PTLplan. Similarly to TLplan, PTLplan makes use of knowledge encoded in a temporal logic to guide the search for a conditional plan in a probabilistic domain.

The problem of planning in nondeterministic domains and under partial observability has also been addressed with approaches different from MDP and probabilistic planning in a qualitative setting. A first attempt is described in [47], which presents an off-line planning algorithm performing a breadth-first search on an and-or graph. Other initial attempts have extended classical planning to the problem of planning in nondeterministic domains (see for instance [73,74]). More recent approaches extend planners based on planning graphs [13] and satisfiability [61]. Most of them are limited either to full observability, or to conformant planning, see, e.g., [19,21,31,41]. Among those that can deal with partial observability, the first significant result was SGP [88], an extension of GRAPHPLAN that provides significant improvements in performance compared with previous extensions to classical planners. SGP produces acyclic conditional plans, but it is unable to deal with nondeterministic action effects, i.e., uncertainty is limited to the initial condition. The algorithm is enumerative in nature: a planning graph is built for each initial state that can be distinguished by observation. For this reason, SGP is not competitive with more recent planners considered in this paper. For instance, in the empty room problem, SGP was unable to solve the 3 × 3 version of the problem in 12 hours of CPU time.

Among the planners based on satisfiability, only QBFPLAN [77] can deal with partial observability. The planning problem is reduced to a QBF satisfiability problem, which is input to an efficient solver [78]. QBFPLAN is limited to bounded-length planning, i.e. it looks for a strong solution of specified length l. When this does not exist, it iteratively increases l until a solution is found or a specified limit is reached. QBFPLAN is thus unable to detect when the problem is unsolvable. QBFPLAN exploits its symbolic approach to avoid exponential blow up caused by the explicit enumeration of states, but seems unable to scale up to large problems.

A different approach to the problem of planning under partial observability is the idea of “Planning at the Knowledge Level”, implemented in the PKS planner [71]. This approach is based on a representation of incomplete knowledge and sensing at a higher level of abstraction. While this approximation can effectively improve perfor-
mance, it makes it impossible to model a variety of problems that can be dealt with an exact representation like the one proposed in this paper. The extension presented in [34] provides a limited solution to the problem of deriving complete conclusions from observations.

The work presented in this paper is closely related to the research line of ”Planning via symbolic model checking”, where the special cases of full observability [34] and conformant planning [35,36] have been tackled. The main difference is that here we provide a solution to the general problem of planning under partial observability. This increased generality comes to a price: we can expect that tackling both special cases with the general approach described in this paper would be significantly less effective than the specialized algorithms proposed in [34] and in [36]. On the other hand, we remark that the MBP planner implements all the above algorithms in a uniform framework, and the selection of the most appropriate algorithm depending on the problem at hand can be completely automatized. It is also interesting to notice that the problem of strong planning under full observability is tackled in [34] with a BDD-based, backward and-or search algorithm, which is significantly different from the one proposed in this paper. In principle, it would be possible to tackle the problem of strong planning under partial observability by means of the very same approach of [34]; however, this would require the construction of the powerset automaton of the domain, which is potentially associated with an exponential blow up. In addition, the progression algorithm presented in this paper has the advantage that, with respect to a backward search algorithm, it always restricts the search to reachable belief states.

This paper substantially extends some preliminary results presented in [7,9]. In particular, none of those papers provided a full formal definition of the framework, nor a proof of the properties of the algorithm; the work in [9] is based on a hardcoded depth-first search algorithm, not amenable to the usage of heuristic measures; and the experimental analysis in [9] and [7] is much less detailed of the one we present here.

Other BDD-based approaches to the problem of strong planning under partial observability have been proposed in the YKA [79] and JUSSIPOP planners [81]. Similarly to our approach, these planners adopt an and-or search schema, but contrary to MBP they perform a backward search in the space of beliefs. As such, observations are used to recombine beliefs, according to a fixed cardinality-based heuristics. YKA is consistently outperformed by JUSSIPOP (which is, in turn, outperformed by MBP, as shown in Section 7).

Other planners that are based on symbolic model checking techniques restrict to the case of full observability, see, e.g., UMOP [55,56], or to classical planning, see, e.g., MIPS [42]. Other approaches are based on different model checking techniques, e.g., on explicit-state representations, and most of them are also limited to the case of full observability. This is the case of SIMPLAN [58]. [40] presents a automata based approach to formalize planning in deterministic domains. The work in [45,46,48] presents a method where model checking with timed automata is used to verify that generated plans meet timing constraints.

In all the approaches mentioned above, uncertainty is dealt with at planning time. Methods that interleave planning and execution (see, e.g., [1,11,62]) 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 can not guarantee to find a solution, unless assumptions are made about the domain.

Finally, we remark that our algorithm differs from and-or search algorithms such as AO* and LAO*, that have been around for a long time. AO* [66,68] and its variations such as HS [67] or CF [63] aim at discovering a cost-minimal solution for an acyclic and-or graph whose arcs represent action outcomes associated with a cost. The assumption of acyclicity of the search space, necessary for AO*’s cost revision step, does not generally hold for the belief spaces associated to planning problems for nondeterministic, partially observable planning domains; this inhibits the usage of AO* and its variants in such a setting. On the other hand, LAO* [53] is able to deal with cyclic graphs, but the notion of acceptable solution includes cyclic plans—whose execution would not be guaranteed to terminate. In this paper, we require solutions to be strong, i.e. to guarantee that they reach the goal in a finite number of execution steps, a feature LAO* can not guarantee. The idea of “mimicking” the requirement of acyclicity in LAO* by playing with the weights appears unnatural from a representational point of view, and difficult to achieve in a principled manner.

Recent work [18] focuses on the problem finding optimal worst-case cost solutions in cyclic and-or graphs. This is based on an IDA* search mechanism, where backtracking is triggered by the inconsistency of the cost value associated to a node with those associated to its children. The focus on optimality and the iterative DFS kind of search make the approach and the algorithm rather different from the one we propose here. A similar problem is tackled in [54]; again, the focus on optimization of the cost leads to a complex algorithm which relies on bottom-up cost updating, and which is therefore very different from the one we present here.
Very recent work extends the symbolic conformant planner CFF [19] to deal with contingent planning by combining AO* with a cycle detection mechanism [50]. There are several differences with respect to our approach. First, CFF is based on SAT rather than on BDDs, and uses planning graphs to compute heuristic measures. Second, CFF is only able to deal with a limited form of nondeterministic action effects. Third, for efficiency reasons CFF only implements an approximate cycle detection mechanism, while MBP has an exact cycle detection mechanism, thanks to the BDD machinery. In [50], CFF is compared against MBP implementation of the techniques proposed in [9].

9. Conclusions and future work

In this paper we have presented a new, comprehensive approach to the problem of planning under partial observability. First, we formally defined the problem of strong planning under partial observability, within an expressive framework that captures a wide class of nondeterministic, partially observable domains. Then, we proposed a new algorithm for strong planning under partial observability. The algorithm is proved to always terminate, and to be correct and complete. We discussed how the algorithm is amenable to a symbolic, BDD-based implementation, and proposed several search strategies. We carried out a thorough experimental evaluation, over a wide class of problems in the literature, evaluated the various search strategies, and compared our approach with the other available systems. Our MBP planner is able to outperform its competitors, often by orders of magnitude.

In the future, we plan to extend this work along the following directions. First, we will investigate the use of heuristic search to enhance the quality of the search, e.g. by exploiting the ideas in [36], and the use of classical planning technologies such as planning graphs. Second, we will investigate variations and extensions of the problem, in particular strong cyclic planning, and planning for temporally extended goals. Finally, we will address how to integrate the planning algorithm within the setting of interleaving planning and execution.

Appendix A. Proofs of theorems on strong planning and belief execution

Theorem A.1. Let \( \pi \) be a plan applicable in belief state \( B \). Then the following properties hold:

- if \( \pi = a \circ \pi' \), then \( \text{FinalBels}(B, \pi) = \text{FinalBels}(\text{Exec}(a, B), \pi') \);
- if \( \pi = \text{if o then } \pi_1 \text{ else } \pi_2 \), then \( \text{FinalBels}(B, \pi) = \text{FinalBels}(B \cap X^-(o)), \pi_1) \cup \text{FinalBels}(B \cap X^-(\bar{o}), \pi_2) \).

Proof. (i) Let \( \pi = a \circ \pi' \). Then \( \text{MaxPaths}(\pi) = a \circ \text{MaxPaths}(\pi') \). This induces a one-to-one correspondence between \( \text{MaxPaths}(\pi) \) and \( \text{MaxPaths}(\pi') \): any \( p \in \text{MaxPaths}(\pi) \) has the form \( p = a \circ p' \) for some \( p' \in \text{MaxPaths}(\pi') \), and for each \( p' \in \text{MaxPaths}(\pi') \). Let \( p = a \circ p' \) be a generic element of \( \text{MaxPaths}(\pi) \). From the definition of \( \text{Exec} \), \( \text{Exec}(B, p) = \text{Exec}(\text{Exec}(B, a), p') \). The thesis follows from the definition of \( \text{FinalBels} \).

(ii) Let \( \pi = \text{if o then } \pi_1 \text{ else } \pi_2 \). Then \( \text{MaxPaths}(\pi) = o \circ \text{MaxPaths}(\pi') \cup \bar{o} \circ \text{MaxPaths}(\pi') \). This induces a one-to-one correspondence between \( \text{MaxPaths}(\pi) \) and \( \text{MaxPaths}(\pi') \cup \text{MaxPaths}(\pi') \): any \( p \in \text{MaxPaths}(\pi) \) is either \( p = o \circ p' \) for some \( p' \in \text{MaxPaths}(\pi') \) or \( p = \bar{o} \circ p'' \) for some \( p'' \in \text{MaxPaths}(\pi') \); furthermore, for each \( p' \in \text{MaxPaths}(\pi') \), \( o \circ p' \in \text{MaxPaths}(\pi) \), and for each \( p'' \in \text{MaxPaths}(\pi') \), \( \bar{o} \circ p'' \in \text{MaxPaths}(\pi) \). Let \( p \) be a generic element of \( \text{MaxPaths}(\pi) \). If \( p = o \circ p' \), from the definition of \( \text{Exec} \), \( \text{Exec}(B, p) = \text{Exec}(B \cap X^-(o), p') \). If \( p = \bar{o} \circ p'' \), from the definition of \( \text{Exec} \), \( \text{Exec}(B, p) = \text{Exec}(B \cap X^-(\bar{o}), p') \). The thesis follows from the definition of \( \text{FinalBels} \).

Theorem 3.5. Let \( (D, I, G) \) be a planning problem, and let \( \pi \) be applicable in \( I \). Then \( \pi \) is a strong solution to \( (D, I, G) \) iff \( \bigcup \text{FinalBels}(I, \pi) \subseteq G \).

Proof. We prove the stronger statement that, for any belief \( B \subseteq S \), if \( \pi \) is applicable in \( B \), then \( \text{FinalStates}(B, \pi) = \bigcup \text{FinalBels}(B, \pi) \). We proceed by induction on the structure of \( \pi \):

- if \( \pi = \varepsilon \), \( \text{FinalStates}(B, \pi) = B \), and \( \text{FinalBels}(B, \pi) = \{B\} \);
- if \( \pi = a \circ \pi' \), then \( a \) is applicable in \( B \), and \( \pi' \) is applicable in \( \text{Exec}(a, B) \).

From the inductive hypothesis, \( \text{FinalStates}(\text{Exec}(a, B), \pi') = \bigcup \text{FinalBels}(\text{Exec}(a, B), \pi) \).

From Definition 2.12, we have that \( \text{FinalStates}(\text{Exec}(a, B), \pi') = \text{FinalStates}(B, a \circ \pi') \).
From Theorem A.1, we have that $\text{FinalBels}(\text{Exec}(a, B), \pi') = \text{FinalBels}(B, a \circ \pi')$.

- If $\pi$ is $\text{if o then } \pi' \text{ else } \pi''$, then $\pi'$ is applicable in $B \cap X^-(o)$, and $\pi''$ is applicable in $B \cap X^-(\bar{o})$.

From the inductive hypothesis, $\text{FinalStates}(B \cap X^-(o), \pi') = \bigcup \text{FinalBels}(B \cap X^-(o), \pi')$, and $\text{FinalStates}(B \cap X^-(\bar{o}), \pi' - o) = \bigcup \text{FinalBels}(B \cap X^-(\bar{o}), \pi' - o)$.

From Definition 2.12, we have that $\text{FinalBels}(B, \text{if o then } \pi' \text{ else } \pi'') = \text{FinalBels}(B \cap X^-(o), \pi') \cup \text{FinalBels}(B \cap X^-(\bar{o}), \pi' - o)$.

From Theorem A.1, we have that $\text{FinalStates}(B, \text{if o then } \pi' \text{ else } \pi'') = \text{FinalStates}(B \cap X^-(o), \pi') \cup \text{FinalStates}(B \cap X^-(\bar{o}), \pi' - o)$.

**Appendix B. Properties of the algorithm**

This section contains the formal proofs showing that the algorithm is correct, complete and terminating, together with the necessary lemmas and introductory structural notions. We first introduce some additional structural size and distance notions (Section B.1); then, we show that the algorithm restricts the search to a class of plans that feature no loops nor predetermined observations, and we prove this gives no loss of generality (Section B.2). Then, we prove some invariants about the main loop of the algorithm (Section B.3), which are finally used in the proofs of termination, correctness and completeness (Section B.4).

**B.1. Structural notions**

For the purpose of performing structural induction on search tree and plan structures, we define a notion of distance of a node to the leaves, which intuitively provides the maximum number of arcs that must be traversed from that node to reach a leaf of the tree, and a notion of plan size, indicating the maximal depth of the associated tree structure.

**Definition B.1 (Maximal distance from leaves).** The maximal distance from leaves of a node $n$ of a belief execution tree $T$ is defined as

$$d_{\text{MAX}}(n, T) = \max_{n', \text{PATH}(n) \leq \text{PATH}(n')} |\text{PATH}(n')| - |\text{PATH}(n)|$$

It is easy to see that, if $n_F$ is the father node of $n$, then $d_{\text{MAX}}(n_F, T) \geq d_{\text{MAX}}(n, T) + 1$, and that $d_{\text{MAX}}(n, T)$ is finite for every node of a belief execution tree $T$. The same notion also applies to a search tree $ST$.

**Definition B.2 (Size of plan).** The size of a plan $\pi$, indicated with $|\pi|$, is the length of its longest maximal path, and can be defined as follows:

- $|\epsilon| = 0$;
- $|a \circ \pi'| = 1 + |\pi'|$;
- $|\text{if o then } \pi_1 \text{ else } \pi_2| = 1 + \max(|\pi_1|, |\pi_2|)$.

**B.2. Non-redundant solutions**

Here, we prove that restricting the search to non-redundant plans does not affect completeness of the search: any (possibly redundant) plan can be normalized into a unique non-redundant plan by the NORMALIZE procedure in Fig. 12.

In the procedure, TRAVERSED(T) identifies the nodes of $T$ other than its root, and TRABELS(T) the associated set of beliefs. Intuitively, NORMALIZE simplifies a plan $\pi$ according to the following ideas:

- every possible execution of the plan stops as soon as the goal is reached (lines 4–5);
- if the plan traverses a belief $B$ more than once, the prefix that causes multiple traversals of $B$ is removed (lines 6–8);
- every predetermined observation is removed, keeping only the subplan associated to the one possible observation branch (lines 11–15).
As a result, it is easy to see that the normalization of a (possibly redundant) solution \( \pi \) returns a plan \( \pi' \) such that every belief associated to a node of \( BET(\pi', \mathcal{I}) \) is also associated to some node of \( BET(\pi, \mathcal{I}) \), i.e. the execution of \( \pi' \) may traverse a subset of the beliefs that can be traversed by executing \( \pi \).

**Theorem 4.5.** Let \( \pi \) be a strong solution for a problem \( \langle \mathcal{D}, \mathcal{B}, \mathcal{G} \rangle \). Then \( \text{NORMALIZE}(\pi, \mathcal{B}, \mathcal{G}) \) is a non-redundant solution for \( \langle \mathcal{D}, \mathcal{B}, \mathcal{G} \rangle \).

**Proof.** We prove the theorem by induction on the size of the plan \( \pi \).

In the base case, \(|\pi| = 0\), i.e. \( \pi = \varepsilon \); then, \( \text{NORMALIZE}(\pi, \mathcal{B}, \mathcal{G}) = \pi = \varepsilon \). Since the plan is a solution and \( \varepsilon \) is not redundant, the theorem holds.

In the induction step, we reason by cases, following the definition of the \( \text{NORMALIZE} \) routine:

1. When \( \mathcal{B} \subseteq \mathcal{G} \), the empty plan \( \varepsilon \) is returned, which is a non-redundant solution if \( \mathcal{B} \subseteq \mathcal{G} \);
2. (1) does not hold and \( \mathcal{B} \in \text{TRAVBELS}(\text{BET}(\pi, \mathcal{B})) \).
   Then, \( \text{SubTree}(\text{BET}(\pi, \mathcal{B}), n_B) \) is a belief execution tree whose root is associated to \( \mathcal{B} \). Since its leaves are leaves of \( \text{BET}(\pi, \mathcal{B}) \), \( \text{PlanOf}(\text{SubTree}(\text{BET}(\pi, \mathcal{B}), n_B)) \) is a subplan of \( \pi \) which is a solution for \( \langle \mathcal{D}, \mathcal{B}, \mathcal{G} \rangle \). Also, \(|\text{PlanOf}(\text{SubTree}(\text{BET}(\pi, \mathcal{B}), n_B))| < |\pi| \); so, for the inductive hypothesis, the returned plan \( \text{NORMALIZE}(\text{PlanOf}(\text{SubTree}(\text{BET}(\pi, \mathcal{B}), n_B)), \mathcal{B}, \mathcal{G}) \) is a non-redundant solution for the problem.
3. (1) and (2) do not hold, and \( \pi = a \circ \pi' \).
   Then \( \pi' \) is a solution for \( \langle \mathcal{D}, \text{Exec}(B, a), \mathcal{G} \rangle \), and since \(|\pi'| < |\pi| \), by the inductive hypothesis, \( \text{NORMALIZE}(\pi', \text{Exec}(B, a), \mathcal{G}) \) is a non-redundant solution for \( \langle \mathcal{D}, \text{Exec}(B, a), \mathcal{G} \rangle \). Then since (2) does not hold, the returned plan \( a \circ \text{NORMALIZE}(\pi', \text{Exec}(B, a), \mathcal{G}) \) is a non-redundant solution for \( \langle \mathcal{D}, \mathcal{B}, \mathcal{G} \rangle \).
4. (1), (2), (3) do not hold, and \( \pi = \text{if } o \text{ then } \pi_1 \text{ else } \pi_2 \) and \( \mathcal{B} \cap \mathcal{X}^{-}(\overline{o}) = \emptyset \).
   Then \( \pi_1 \) is a solution for \( \langle \mathcal{D}, \mathcal{B}, \mathcal{G} \rangle \) (since \( \mathcal{B} \cap \mathcal{X}^{-}(o) = \mathcal{B} \)), and since \(|\pi_1| < |\pi| \), by the induction step, the returned plan \( \text{NORMALIZE}(\pi_1, \mathcal{B}, \mathcal{G}) \) is a non-redundant solution for the same problem.
5. (1), (2), (3), (4) do not hold, and \( \pi = \text{if } o \text{ then } \pi_1 \text{ else } \pi_2 \) and \( \mathcal{B} \cap \mathcal{X}^{-}(\overline{o}) = \emptyset \).
   Then \( \pi_1 \) is a solution for \( \langle \mathcal{D}, \mathcal{B}, \mathcal{G} \rangle \) (since \( \mathcal{B} \cap \mathcal{X}^{-}(o) = \mathcal{B} \)), and since \(|\pi_2| < |\pi| \), by the induction step, the returned plan \( \text{NORMALIZE}(\pi_2, \mathcal{B}, \mathcal{G}) \) is a non-redundant solution for the same problem.
6. (1), (2), (3), (4), (5) do not hold, and \( \pi = \text{if } o \text{ then } \pi_1 \text{ else } \pi_2 \).
   Then \( \pi_1 \) is a solution for \( \langle \mathcal{D}, \mathcal{B} \cap \mathcal{X}^{-}(o), \mathcal{G} \rangle \). By the inductive hypothesis, since \(|\pi_1| < |\pi| \), \( \pi_1 N = \text{NORMALIZE}(\pi_1, \mathcal{B} \cap \mathcal{X}^{-}(o), \mathcal{G}) \) is a non-redundant solution for \( \langle \mathcal{D}, \mathcal{B} \cap \mathcal{X}^{-}(o), \mathcal{G} \rangle \). By a similar reasoning, \( \pi_2 N = \text{NORMALIZE}(\pi_2, \mathcal{B} \cap \mathcal{X}^{-}(\overline{o}), \mathcal{G}) \) is a non-redundant solution for \( \langle \mathcal{D}, \mathcal{B} \cap \mathcal{X}^{-}(\overline{o}), \mathcal{G} \rangle \). Then, since (2), (3), (4) do not hold, if \( o \in \mathcal{O} \) then \( \pi_1 N \text{ else } \pi_2 N \) is a non-redundant solution.

**B.3. Loop invariants**

In order to prove the key properties of termination, correctness and completeness of the algorithm, it is convenient to state and prove a number of loop invariants over the search tree constructed by the main loop. In the following, we denote with \( \text{ST}_i \) the search tree at the beginning of the \( i \)th iteration of the loop of the planning algorithm in Fig. 6, i.e. prior to the evaluation of the stop condition at line 6 of the algorithm.

**Lemma B.3.** For every non-leaf node \( n \in \text{ST}_i \), the following conditions hold:

- if \( a \in \mathcal{A} \) is applicable on \( \text{BEL}(n) \), then \( \text{ST}_i \) contains a node \( n' = \langle \text{Exec}((\text{BEL}(n), a), \mathcal{P} \mathcal{A} \mathcal{H} n), o a \rangle \), and an or-arc \langle \langle n, a, n' \rangle \rangle;
- if \( o \in \mathcal{O} \) is not predetermined on \( \text{BEL}(n) \), then \( \text{ST}_i \) contains two nodes \( n_T = \langle \text{BEL}(n) \cap \mathcal{X}^{-}(o), \mathcal{T} \mathcal{P} \mathcal{A} \mathcal{H} n, o \rangle \) and \( n_F = \langle \text{BEL}(n) \cap \mathcal{X}^{-}(\overline{o}), \mathcal{T} \mathcal{P} \mathcal{A} \mathcal{H} n, \overline{o} \rangle \), and an and-arc \langle \langle n, o, \langle n_T, n_F \rangle \rangle \).

**Proof.** We prove the theorem by induction on the number of top-level iterations performed by the algorithm. The proof is trivial for the initial search tree \( \text{ST}_0 \), whose only node is a leaf. It is also easy to see that, it the theorem holds for \( \text{ST}_i \), it holds for \( \text{ST}_{i+1} \), since the internal nodes of \( \text{ST}_{i+1} \) are those of \( \text{ST}_i \), plus the one node \( n \) having been
expanded during the \( i \)th iteration by \textsc{ExtendTree}; but \textsc{ExtendTree} is defined in a way that directly maps into the property holding also for \( n \). \( \square \)

**Lemma B.4.** For every non-leaf node of \( ST_{[i]} \), if \textsc{Sons yield Succ} holds, the node is tagged Success; if \textsc{Sons yield Fail} holds, the node is tagged Failure; otherwise, it is tagged Undetermined.

**Proof.** We prove the theorem by induction on the number of top-level iterations performed by the algorithm. In the base case, the tree \( ST_{[0]} \) is built exclusively by a leaf node, so the theorem trivially holds. Now, suppose the theorem holds for the tree \( ST_{[i]} \). At the \((i + 1)\)th iteration, a leaf \( n \) is chosen and expanded, and its tag is set depending on whether \textsc{Sons yield Fail}(\( n \)) or \textsc{Sons yield Succ}(\( n \)) hold; as such, node \( n \) obeys the requirement of the theorem. If neither \textsc{Sons yield Fail}(\( n \)) or \textsc{Sons yield Succ}(\( n \)) hold, \( n \) is left as Undetermined, and no success or failure propagation takes place. In this case, the theorem holds, since by the induction step, it holds for every node of \( ST_{[i+1]} \) different from \( n \). If \textsc{Sons yield Succ}(\( n \)) holds, due to the induction step, the only nodes for which the theorem has to be proved are the ancestors of \( n \), which may have to be set as successful. But it is easy to see that the \textsc{PropagateSuccTree} routine recursively recomputes the tag of every ancestor for which \textsc{Sons yield Succ} holds, proving the theorem in this case. Similarly for the case of \textsc{Sons yield Fail}(\( n \)). \( \square \)

**Lemma B.5.** If \( n \) is tagged Success (Failure) in \( ST_{[i]} \), it is also contained in \( ST_{[j]} \), where \( j > i \), and is tagged Success (Failure resp.) in \( ST_{[j]} \).

**Proof.** We first observe that, since \( ST_{[i+1]} \) has every node of \( ST_{[i]} \), plus possibly some additional nodes resulting from the expansion of a frontier node, every node of \( ST_{[i]} \) is contained in \( ST_{[i+1]} \). By induction, for every \( j > i \), every node of \( ST_{[i]} \) is contained in \( ST_{[j]} \).

We prove the statement by induction on the number of top-level iterations of the algorithm: we prove it holds considering \( j = i + 1 \), and this implies it holds for any \( j > i \). For a single step of the induction, we prove the theorem by induction on the frontier distance of nodes: we first prove it holds for nodes of \( ST_{[i]} \) at distance 0, then we prove that, if it holds for nodes at distance \( D \), it holds for nodes at distance \( D + 1 \).

Success/failure leaves of \( ST_{[i]} \) are also success/failure leaves of \( ST_{[i+1]} \), so the base step is immediate. Now consider a determined node of \( ST_{[i]} \) at distance \( D + 1 \) from the frontier. Its sons are all at distance at most \( D \), so, if their tag is Success or Failure in \( ST_{[i]} \), according to the inductive hypothesis, they have the same tag in \( ST_{[i+1]} \). Given this, it is easy to see that if \textsc{Sons yield Succ} holds for \( n \) on \( ST_{[i]} \), it also holds on \( ST_{[i+1]} \), regardless of the fact that some son of \( n \) may be undetermined in \( ST_{[i]} \), and determined in \( ST_{[i+1]} \). The same goes for \textsc{Sons yield Fail}. This, together with Lemma B.4, proves the theorem. \( \square \)

**Lemma B.6.** If the frontier of \( ST_{[i]} \) is empty, then the root node is tagged either as a failure or as a success.

**Proof.** We prove the following stronger statement: let \( n \) be a node of \( ST_{[i]} \), and let \( N \) be the set of its descendant leaves (or \( \{ n \} \), if \( n \) is a leaf). Then, if every node in \( N \) is determined, so is \( n \).

We prove this statement by induction on the number of iterations performed by the algorithm. The base case is trivial: \( ST_{[0]} \) only has a (leaf)node, and for a leaf node the theorem is a tautology. Now let us assume the theorem holds on the tree \( ST_{[i]} \) produced after \( i \) iterations. The tree \( ST_{[i+1]} \) is produced by expanding a leaf \( l \) of \( ST_{[i]} \), tagging it according to \textsc{Sons yield Fail} and \textsc{Sons yield Succ}, and, if \( l \) is tagged Failure or Success, propagating bottom-up the failure or success to its ancestors. Consider a generic node \( n \) of \( ST_{[i+1]} \); one of the following conditions hold:

- \( n \) is a leaf of \( ST_{[i+1]} \), i.e. either a leaf of \( ST_{[i]} \) or one of the newly introduced leaves. The statement in this case reduces to a tautology;
- \( n \) is \( l \); also in this case, since its tag is established by \textsc{Sons yield Succ} and \textsc{Sons yield Fail}, the statement is proved immediately;
- \( n \) is a non-leaf node of \( ST_{[i]} \), and either \( l \) is not one of its descendants, or \( l \) has not been tagged Success or Failure after its expansion. Then it is easy to see neither \( n \) nor any of its descendants have their tag affected by the tree expansion; then, given the inductive hypothesis, the theorem holds for \( n \);
If \( n \) is a non-leaf node of \( ST[i] \), having \( l \) as one of its descendants, and \( l \) has been tagged Success or Failure after its expansion. To prove that the theorem holds for \( n \), we reason inductively on the maximal frontier distance of its descendant nodes. The theorem trivially holds for its descendants which are leaves (at distance 0). If the theorem holds for nodes at distance \( D \), it holds for nodes at distance \( D + 1 \), since they are either ancestors of \( l \) (in which case the bottom-up propagation of failure/success guarantees they are properly tagged), or determined nodes of \( ST[i] \) (their descendant leaves are determined, and belonging to \( ST[i] \)). \( \square \)

**Lemma B.7.** If a node \( n \in ST[i] \) is tagged Success, then \( BUILDPLAN(n,ST[i]) \) is a solution for the problem \( \langle D, BEl(n), G \rangle \).

**Proof.** We prove the theorem by induction over the maximal distance from leaves of the nodes: we first prove it for nodes at distance 0, and then we prove that, if it holds for nodes at distance \( D \), it holds for nodes at distance \( D + 1 \). If \( n \) is a leaf, at distance 0, the proof is trivial: a leaf is tagged success iff its associated belief is entailed by the goal, and \( BUILDPLAN \) returns the empty plan. Now assume that a node \( n \) at distance \( d > 0 \) is tagged success; then, its successful sons are at a distance \( d' \leq d - 1 \); thus, by the inductive hypothesis, they admit at least a solution, produced by \( BUILDPLAN \). Then, we observe that:

1. There is a solution for \( \langle D, BEl(n), G \rangle \).
   - If \( n \) has some successful or-son \( n_{suc} \) s.t. \( \pi' \) is a solution for \( \langle D, BEl(n_{suc}), G \rangle \), since \( a \) is executable on \( BEl(n) \) and \( BEl(n_{suc}) = Exec(BEl(n), a) \) (where \( \langle n, a, n_{suc} \rangle \in ST[i] \)), the plan \( a \circ \pi' \) is a solution for \( \langle D, BEl(n), G \rangle \).
   - Otherwise, \( n \) has some successful pair of brother and-sons \( n_T \) and \( n_F \) s.t. \( \langle n, o, n_T, n_F \rangle \in ST[i] \), and there are two plans \( \pi_T, \pi_F \) which are respectively solutions for \( \langle D, BEl(n_T), G \rangle \) and \( \langle D, BEl(n_F), G \rangle \). In this case, since \( BEl(n_T) = BEl(n) \cap X^-(o) \) and \( BEl(n_F) = BEl(n) \cap X^-(\bar{o}) \), if \( o \) then \( \pi_T \) else \( \pi_F \) is a solution for \( \langle D, BEl(n), G \rangle \).
2. The plan returned by \( BUILDPLAN \) is a solution for \( \langle D, BEl(n), G \rangle \). This is immediate to see, since it returns a plan constructed according to the ideas in point 1. \( \square \)

**Lemma B.8.** If \( ST[i] \) contains a belief execution tree \( T \) whose leaves are all non-failure nodes, no node of \( ST[i] \) in \( T \) can be tagged as Failure.

**Proof.** We prove the statement by induction on the structure of the belief execution tree \( T \), based on the maximal distance of nodes from the frontier. The proof goes as follows:

- base step: all nodes at distance 0, i.e. the leaves of \( T \), are non-failure nodes;
- induction step: if all nodes at distance \( D \) are non-failures, all those at distance \( D + 1 \) are non-failures. This is easily shown by the fact that a node \( n' \) at distance \( D + 1 \) features at least one or-son or a pair of brother and-sons at distance at most \( n \) which are not failures (namely, the sons in \( T \)), thus \( SONS\text{YIELDFAIL}(n') \) returns false, and as such \( n' \) is not tagged Failure (see Lemma B.4). \( \square \)

**Lemma B.9.** Let \( \pi \) be a non-redundant solution for \( \langle D, I, G \rangle \). A leaf of \( BET(\pi, I) \) is either a successful leaf of \( ST[i] \), or a node not in \( ST[i] \), in which case it has an ancestor in the frontier of \( ST[i] \).

**Proof.** We prove the theorem by induction on the number of iterations performed by the algorithm.

In the base case, the theorem is immediate. If \( I \subseteq G \), the one non-redundant solution is \( \varepsilon \), and the only leaf of \( BET(\varepsilon, I) \) is the success leaf of \( ST[0] \). If \( I \not\subseteq G \), the only node of \( ST[0] \) is a leaf, and, for any plan \( \pi \), is also the root of \( BET(\pi, I) \), thus an ancestor to every leaf of \( BET(\pi, I) \).

Now assume the theorem holds for \( ST[i] \); \( ST[i+1] \) is obtained by picking a frontier node \( n \) of \( ST[i] \), expanding it, tagging it and propagating success/failure if those are detected. Now consider a non-redundant solution \( \pi \) and a leaf \( l \) of \( BET(\pi, I) \). The following cases are possible:

- \( l \) is a node of \( ST[i] \); thus it is also a node of \( ST[i+1] \), and, given the inductive hypothesis, the theorem holds for \( l \);
- $l$ is not a node of $ST_{[i]}$, and its ancestor on the frontier of $ST_{[i]}$ is a node $n'$ different from $n$. Then $n'$ is also its ancestor in the frontier of $ST_{[i+1]}$.
- $l$ is not a node of $ST_{[i]}$, and its ancestor on the frontier of $ST_{[i]}$ is $n$. Then two cases are possible:
  - $l$ is one of the nodes resulting from the expansion of $n$. But then, since $l$ is associated to a belief entailed by $G$ (it is a leaf of $BET(\pi, I)$), the EXTENDTREE procedure has it marked as Success, so it is a successful leaf of $ST_{[i+1]}$;
  - $l$ is not one of the nodes resulting from the expansion of $n$. Since EXTENDTREE expands $n$ by every applicable action and non-predetermined observation, a node $n'$ resulting from such expansion is an ancestor of $l$. But notice that, since $\pi$ is non-redundant, $n'$ cannot be associated to a belief entailed by $G$, so it is not marked Success. Also, since $\pi$ is non-redundant, no belief on its belief path is traversed more than once; so $n'$ cannot be causing a loop, and is not marked as Failure. As such, $n'$ is a frontier node of $ST_{[i+1]}$. □

B.4. Main properties

Theorem B.10. Every execution of PROPAGATESUCCONTREE and PROPAGATEFAILONTREE on a node of $ST_{[i]}$ terminates.

Proof. Both propagation routines perform a bottom-up traversal of the (finitely long) path of the node they receive as argument, and call terminating subroutines within the recursion. Thus we prove the termination by proving the recursion schema terminates. We proceed by induction on the length of the action-observation path. The base case is immediately to prove given the finiteness of the actions and observations that can be applied to a node.

To prove the second, we observe that at each loop, the algorithm (a) removes a node from the frontier, and (b) may add some undetermined nodes on the frontier, each being associated to an acyclic belief path. We observe that the number of acyclic belief paths is finite (since beliefs are finite), and so is the number of possible nodes in the frontier (since they correspond to acyclic belief paths). Let $N'$ be the number of acyclic belief paths for a domain $D$.

Then, after at most $N'$ iterations, every acyclic path will be generated, and the frontier will contain $N_M \leq N'$ nodes, each corresponding to a maximal acyclic belief path. Then, after further $N_M$ iterations, each node will be removed from the frontier, and its extension generates a cyclic path, no more nodes will be added to the frontier. That is, the frontier will become empty, causing the failure of the root node (see Lemma B.6), and as such the termination of the algorithm. □

Theorem B.11. Every execution of BUILDPLAN on a successful node of $ST_{[i]}$ terminates.

Proof. This originates from Lemma B.7: if the node is successful, a solution (of finite size) exists, and is produced by BUILDPLAN. But the number of recursions of BUILDPLAN is bounded by the number of arcs in the belief execution tree associated to the plan, since each recursion adds one (or- or and-) arc to the resulting tree. This number is finite, proving termination. □

Theorem 4.6 (Termination). Given a planning problem, the execution of the planning algorithm in Fig. 6 terminates.

Proof. We first observe that it is enough to prove that the main loop of the algorithm terminates; to do so, we first prove that each iteration of the main loop terminates, and then we prove that the number of such iterations is finite.

The first statement derives directly from the above theorems, and from the termination of EXTENDTREE, which is immediate to prove given the finiteness of the actions and observations that can be applied to a node.

To prove the second, we observe that at each loop, the algorithm (a) removes a node from the frontier, and (b) may add some undetermined nodes on the frontier, each being associated to an acyclic belief path. We observe that the number of acyclic belief paths is finite (since beliefs are finite), and so is the number of possible nodes in the frontier (since they correspond to acyclic belief paths). Let $N'$ be the number of acyclic belief paths for a domain $D$.

Then, after at most $N'$ iterations, every acyclic path will be generated, and the frontier will contain $N_M \leq N'$ nodes, each corresponding to a maximal acyclic belief path. Then, after further $N_M$ iterations, each node will be removed from the frontier, and its extension generates a cyclic path, no more nodes will be added to the frontier. That is, the frontier will become empty, causing the failure of the root node (see Lemma B.6), and as such the termination of the algorithm. □

Theorem 4.7 (Correctness). When the algorithm returns a plan $\pi$, then $\pi$ is a strong solution for the problem.

Proof. This is a direct derivation of Lemma B.7, instantiated for the root node of the tree. □

Theorem 4.8 (Completeness). If a strong solution exists, then the algorithm returns a strong solution.
Proof. We prove the theorem by reductio ad absurdum. Assume that after \( k \) iterations, the algorithm returns Failure; let \( ST[k] \) be the search tree expanded by the algorithm after the \( k \)-th iteration.

Assume that a non-redundant solution \( \pi \) exists, and consider its associated belief execution tree \( T = BET(I, \pi) \). According to Lemma B.9, any leaf node \( l \) of \( T \) is either a successful node of \( ST[k] \), or a node not in \( ST[k] \) with an ancestor on the frontier of \( ST[k] \). Thus, \( ST[k] \) contains a prefix \( T' \) of \( T \) whose leaves are tagged either Success or Undetermined.

But then, according to Lemma B.8, the internal nodes of \( T' \) cannot be failure nodes, including its root, which is also the root of \( ST[k] \). As such, the algorithm cannot have stopped with failure. \( \square \)

References


