Planning as BDD-Based Model Checking

ITC-irst
Università di Genova
Università di Trento

Abstract.
Planning is a longstanding key research area in AI. The last decade has witnessed large improvements in performance of planning techniques and systems, to the point where complex, realistic problems are often solved in a fraction of a second. At the same time, thanks to the generality of the problem formulation, several applicative problems can be recast into planning. This is also for automated software development: e.g., planning can be used to automatically compose distributed services to provide a new service and satisfy a given requirement. In this deliverable, we describe the general planning framework, and the specific planning techniques and tools, which we developed to provide the basis for solving a variety of automated software development problems, especially in the area of distributed service computing.

Document Identifier	Deliverable D4.1
Project	MIUR-FIRB project RBNE0195K5 "Knowledge Level Automated Software Engineering"
Version	v1.0
Date	Oct.31, 2006
State	Final
Distribution	Public

Acknowledgements.

This document is part of a research project funded by the FIRB 2001 Programme of the “Ministero dell’Istruzione, dell’Università e della Ricerca” as project number RBNE0195K5.

The partners in this project are: Istituto Trentino di Cultura (Coordinator), Università degli Studi di Trento, Università degli Studi di Genova, Università degli Studi di Roma “La Sapienza”, DeltaDator S.p.A..

Executive Summary

Planning is, since the early days of AI, one of the most prominent and active research areas. One of the key motivations for such a relevant and longstanding investment stands in the generality of the problem formulation, which makes its solution applicable to a variety of real-life situations. Indeed, ad-hoc planners have been adopted to support diverse activities such as military logistics, remote control of Mars Rovers, smart game playing, and so on.

Concerning automated software development, planning can be adopted to automatically synthesize software, starting from formal requirements. For instance, in a setting where distributed (web) services are available and we desire to obtain new integrated functionalities on top of them, planning can perform the automated composition by appropriately interpreting the component services as the planning domain that needs to be controlled.

However, while solving this problem would have a major impact on the practical possibility to re-use existing services and to integrate different businesses in a flexible way, the specific features of the elements into play make it very challenging for existing planning approaches and call for developing new advanced techniques.

In this deliverable we recap the research performed by the partners of the KLASE project to design advanced planning techniques that are capable of effectively dealing with the automated software development issues, in particular as featured by service composition. The research has proceeded by first designing a reference framework, general enough to encompass all the kinds of problems of interest for web service composition, and then tackling them following an incremental approach, from the less complex to the more complex. This deliverable is organized to reflect such an incremental approach to the problem, and provides the basis over which deliverable D4.3 is funded.

Contents

Chapter 1

Introduction

Since the early days of AI, planning has been receiving considerable amount of attention. One of the key motivations for such a relevant and longstanding investment stands in the generality of the problem formulation, which makes its solution applicable to a variety of real-life situations. Automated planners have been adopted to support diverse activities such as military logistics, remote control of Mars Rovers, smart game playing, and so on.

Focusing on the area of Service-Oriented Computing, planning can be adopted to solve the crucial issue of automatically composing existing services to obtain new integrated functionalities, by appropriately interpreting the component services as the planning domain that needs to be controlled.

However, while solving this problem would have a major impact on the practical possibility to re-use existing services and to integrate different businesses in a flexible way, the specific features of the elements into play make it very challenging for existing planning approaches and call for developing new advanced techniques.

In particular, composing independent web services means to interact with components that (a) exhibit a partially controllable behavior, implementing choices based on internal, and possibly undisclosed, logics, and (b) evolve asynchronously from each other. These conditions break some of the standard assumptions used in most planning approaches: full predictability and observability of the domain behavior, and synchronism.

In this deliverable we recap the research performed by the partners of the KLASE project to design planning techniques that are capable of dealing with sets of unpredictable, partially observable, asynchronous services, and of doing so in an effective way.

The deliverable is organized in two major parts. In the first we describe off-line approaches, i.e. those planning approaches where a complete plan is produced prior to its execution, and the possibility of modifying the plan to respond to unpredicted behaviors is not taken into account. On one side this simplifies the problem statement and avoids certain problems associated to the interleaving of plan generation and execution; on the other, it forces the construction of plans as much robust as possible to the possible responses of the domain. This can be computationally very expensive, and sometimes the required degree of robustness cannot be obeyed. In the second part we describe on-line approaches, that address this problem by allowing for plan execution to be monitored and analyzed so that re-planning is triggered when certain "danger" conditions are met.

Both off-line and on-line planning rely on a formal framework that define the key elements into play; its statement makes up the first chapter of this deliverable. Then, both for the off-line and on-line approaches, we describe the variety of problems that have been tackled. We follow an incremental approach, starting from the simpler problems to the more complex ones, which in the end provide the key techniques for solving automated software development problems, as discussed in deliverable D4.3.

Chapter 2

Framework

In this chapter, we provide a general framework for plan execution and monitoring, giving a formal definition of each of the components into play, and we describe the way this framework can be integrated with a plan generator to obtain off-line or on-line planning systems. The intuition underlying our framework is outlined in Figure 2.1. A domain is a generic system, possibly with its own dynamics, such as a power plant or an aircraft. The plan can control the evolutions of the domain by triggering *actions*. We assume that, at execution time, the state of the domain is only partially visible to the plan; the part of a domain state that is visible to the plan is called the *observation* of the state. In essence, planning is building a suitable plan that can guide the evolutions of the domain in order to achieve the specified goals.

In this setting, a *monitor* is in charge, during the execution, of gathering the actions and observations exchanged between the plan and the domain, and providing, on the basis of an internal model of the behavior of the domain, an evaluation of the current domain state. Given the limited amount of available runtime information, in general, the monitor will not be able to uniquely identify the domain state; as such, it will return a *belief*: a set of states plausible given the history of actions and observations perceived so far.

In the following, we give a formal specification of the components into play. At a high level, they will reflect the following high level features:

- the behavior of a planning domain is only partially predictable, since actions may fail, have different plausible outcomes, or be affected by the uncontrollable intervention of external agents;
- in general, the state of a domain can only be inspected by means of a limited sets of sensors, i.e. it is partially observable;
- while controlling plans are usually designed to have a fully predictable behavior, it is also possible, more general, and in some cases, more convenient, to think of plans that act following some “randomized” logics;

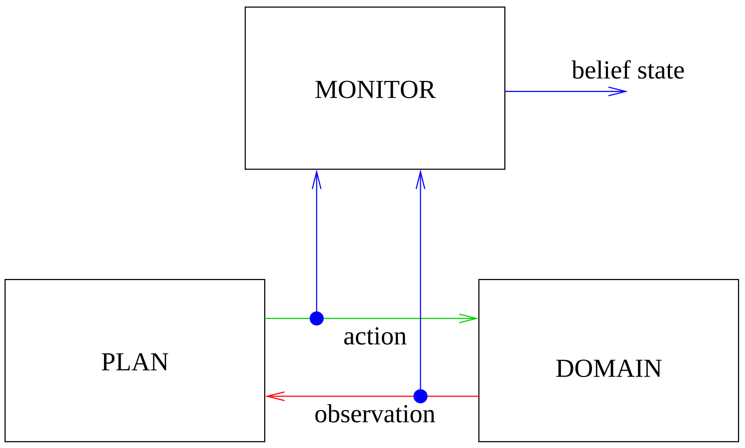


Figure 2.1: The general framework.

- on the contrary, monitors that provide non-deterministic state evaluations would be of no use; therefore, monitors will be thought of as fully deterministic machines.

2.1 Planning Domains

A planning domain is defined in terms of its *states*, of the *actions* it accepts, and of the possible *observations* that the domain can exhibit. Some of the states are marked as valid *initial states* for the domain. A *transition function* describes how (the execution of) an action leads from one state to possibly many different states. Finally, an *observation function* defines what observations are associated to each state of the domain.

Definition 2.1.1 (planning domain) *A* nondeterministic planning domain with partial observability *is a tuple $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{R}, \mathcal{X} \rangle$, where:*

- *$\mathcal{P}$ is the finite set of propositions,*
- *$\mathcal{S} \subseteq 2^{\mathcal{P}}$ is the set of* states.
- *$\mathcal{A}$ is the set of* actions.
- *$\mathcal{O}$ is the set of* observations.
- *$\mathcal{I} \subseteq \mathcal{S}$ is the set of* initial states*; we require $\mathcal{I} \neq \emptyset$.*
- *$\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow 2^{\mathcal{S}}$ is the* transition function*; it associates to each current state $s \in \mathcal{S}$ and to each action $a \in \mathcal{A}$ the set $\mathcal{R}(s, a) \subseteq \mathcal{S}$ of next states.*
- *$\mathcal{X} : \mathcal{S} \rightarrow 2^{\mathcal{O}}$ is the* observation function*; it associates to each state s the set of possible observations $\mathcal{X}(s) \subseteq \mathcal{O}$.*

We say that action a is executable *in state s if $\mathcal{R}(s, a) \neq \emptyset$. We require that in each state $s \in \mathcal{S}$ there is some executable action, that is some $a \in \mathcal{A}$ such that $\mathcal{R}(s, a) \neq \emptyset$. We also require that some observation is associated to each state $s \in \mathcal{S}$, that is, $\mathcal{X}(s) \neq \emptyset$.*

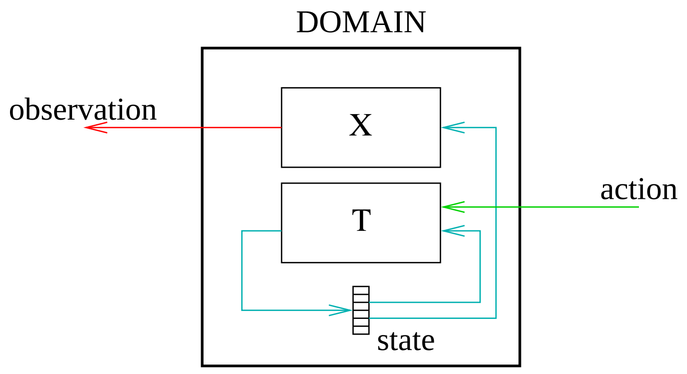


Figure 2.2: The model of the domain.

A picture of the model of the domain corresponding to this definition is given in Figure 2.2. Technically, a domain is described as a nondeterministic Moore machine, whose outputs (i.e., the observations) depend only on the current state of the machine, not on the input action. Uncertainty is allowed in the initial state and in the outcome of action execution. Also, the observation associated to a given state is not unique. This allows modeling noisy sensing and lack of information.

Notice that the definition provides a general notion of domain, abstracting away from the language that is used to describe the domain. For instance, a planning domain is usually defined in terms of a set of *fluents* (or state variables), and each state corresponds to an assignment to the fluents. Similarly, the possible observations of the domain, that are primitive entities in the definition, can be presented by means of a set of *observation variables*, as in [BCRT01]: each observation variable can be seen as an input port in the plan, while an observation is defined as a valuation to all the observation variables. The definition of planning domain does not allow for a direct representation of *action-dependent* observations, that is, observations that depend on the last executed action. However, these observations can be easily modeled by representing explicitly in the state of the domain (the relevant information on) the last executed action.

Indeed, the mechanism of observations allowed by the model presented in Definition 2.1.1 is very general. It can model *no observability* and *full observability* as special cases. *No observability* (conformant planning) is represented by defining $\mathcal{O} = \{\bullet\}$ and $\mathcal{X}(s) = \{\bullet\}$ for each $s \in \mathcal{S}$. That is, observation $\bullet$ is associated to all states, thus conveying no information. *Full observability* is represented by defining $\mathcal{O} = \mathcal{S}$ and $\mathcal{X}(s) = \{s\}$. That is, the observation carries all the information contained in the state of the domain.

2.2 Plans

Now we present a general definition of plans, that encode sequential, conditional and iterative behaviors, and are expressive enough for dealing with partial observability and

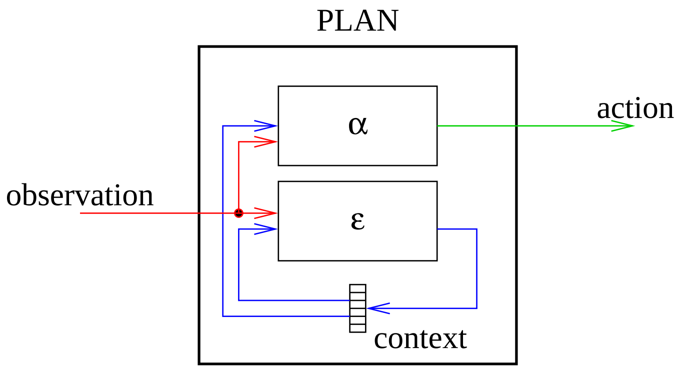


Figure 2.3: The model of the plan.

with extended goals. In particular, we need plans where the selection of the action to be executed depends on the observations, and on an "internal state" of the executor, that can take into account, e.g., the knowledge gathered during the previous execution steps. A plan is defined in terms of an *action function* that, given an observation and a *context* encoding the internal state of the executor, specifies the action to be executed, and in terms of a *context function* that evolves the context.

Definition 2.2.1 (plan) *A* plan *for planning domain* $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{R}, \mathcal{X} \rangle$ *is a tuple* $\Pi = \langle \Sigma, \sigma_0, \alpha, \epsilon \rangle$*, where:*

- Σ *is the set of* plan contexts.
- $\sigma_0 \in \Sigma$ *is the* initial context.
- $\alpha : \Sigma \times \mathcal{O} \rightharpoonup 2^{\mathcal{A}}$ *is the* action function*; it associates to a plan context* c *and an observation* o *a set of actions* $\alpha(c, o)$ *that may be executed.*
- $\epsilon : \Sigma \times \mathcal{O} \rightharpoonup 2^{\Sigma}$ *is the* context evolutions function*; it associates to a plan context* c *and an observation* o *a set of new plan contexts* $\epsilon(c, o)$.

A picture of the model of plans is given in Figure 2.3. Technically, a plan is described as a Mealy machine, whose outputs (the action) depends in general on the inputs (the current observation). Functions α and ϵ can be partial, since a plan may be undefined on the context-observation pairs that are never reached during execution. In general, we consider functions α and ϵ that may be nondeterministic, to allow for "randomized" plans as well as fully predictable ones. In the latter case, functions α and ϵ are deterministic, i.e. they return a unique action and context respectively.

We remark that the general specification provided above can be instantiated in different ways to solve specific kinds of problems.

For instance, with the special cases of fully observable domains, and when only reachability goals are considered, plans can be simplified significantly: implicitly, the plan context is given by the current domain state, and the action and context evolution only depend on the domain state (coincident with the observation). Thus in this case plans can be represented as sets of pairs $\langle s, a\rangle$, with $s \in \mathcal{S}$ and $a \in \mathcal{A}$. These sets are referred to as *state-action tables*.

Another extreme case is the one where the domain is fully unobservable; then the only plans that make sense are sequences of actions, since no observation can be used. The context in this case is implicitly given by the index of the current action.

But even for partially observable domains, when reachability goals are considered, the representation can be simplified. In that case, plans can be perceived as acyclic and-or graphs, where the edges leaving and-nodes are associated to different observations, and or-nodes only originate a single edge associated to an action. Of course, also this case falls into the general one, with the implicit context being given by the unique identifier of the and-or graph node.

In each section discussing a specific planning problem, we will recap the specific notion of plan representation adopted for the purpose, and clarify its relationship with the general notion discussed above.

2.3 Monitors

A monitor is a machine that observes the execution of the plan on the domain and reports a belief state, i.e., a set of possible current states of the domain (see Figure 2.1). Differently from the belief states that appear in a bs-configuration, the belief states reported by the monitor may be a super-set of the states that are compatible with the past history. As we will see, it is this possibility of approximating the possible current states that makes monitors usable in practice for validating plans.

Definition 2.3.1 (monitor) *A* monitor *for a domain $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{R}, \mathcal{X}\rangle$ is a tuple $\mathcal{M} = \langle \mathcal{MS}, m_0, \mathcal{MT}, \mathcal{MO}\rangle$, where:*

- *$\mathcal{MS}$ is the set of states of the monitor.*
- *$m_0 \in \mathcal{MS}$ is the initial state of the monitor.*
- *$\mathcal{MT} : \mathcal{MS} \times \mathcal{O} \times \mathcal{A} \rightharpoonup \mathcal{MS}$ is the transition function of the monitor; it associates to state m of the monitor, observation o, and action a, an updated state of the monitor $m' = \mathcal{MT}(m, o, a)$.*
- *$\mathcal{MO} : \mathcal{MS} \times \mathcal{O} \to 2^{\mathcal{S}}$ is the output function of the monitor; it associates to each state m of the monitor and observation o the corresponding belief state $\mathcal{MO}(m, o)$.*

Definition 2.3.2 (m-configuration) *A* m-configuration *for domain $\mathcal{D}$, plan Π and monitor $\mathcal{M}$ is a tuple (s, o, c, a, m) such that $s \in \mathcal{S}$, $o \in \mathcal{X}(s)$, $c \in \Sigma$, and $m \in \mathcal{MS}$.*

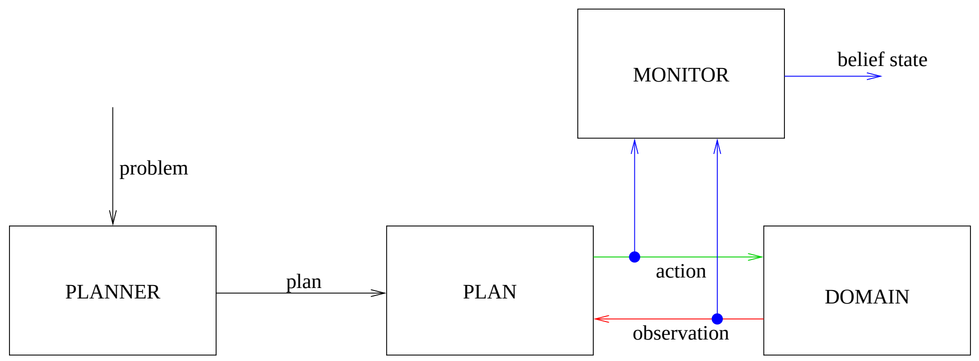


Figure 2.4: The off-line framework.

M-configuration (s, o, c, a, m) *may evolve into m-configuration* (s', o', c', a', m')*, written* $(s, o, c, a, m) \rightarrow (s', o', c', a', m')$*, if:*

- $(s, o, c, a) \rightarrow (s', o', c', a')$*, and*
- $m' = \mathcal{MT}(m, o, a)$*.*

M-configuration (s, o, c, a, m) *is initial if* $s \in \mathcal{I}$*,* $c = \sigma_0$*, and* $m = m_0$*.*

We say that a monitor is *correct* for a given domain and plan if the belief state reported by the monitor after a certain evolution contains *all* the states that are compatible with the observation gathered during the evolution. In the following definition, this property is expressed by requiring that there are no computations along which a state of the domain is reached that is not contained in the belief state reported by the monitor.

Definition 2.3.3 (correct monitor) *Monitor* $\mathcal{M}$ *is correct for domain* $\mathcal{D}$ *and plan* Π *if the following conditions holds for all the reachable m-configurations* (s, o, c, a, m)*:*

- $s \in \mathcal{MO}(m, o)$*;*
- $\mathcal{MT}(m, o, a)$ *is defined.*

From now on we consider only correct monitors.

2.4 Off-line vs. on-line frameworks

The framework discussed above can be embedded in one of two major family of approaches: off-line and on-line planning approaches.

In off-line planning, the idea is that the plan is built once and for all, by a planning system, prior to execution. At runtime, monitoring is used as a sanity check (and the planner plays no role at all), see Fig. 2.4. As such, for off-line approaches, the aim is to construct plans that are as much robust as possible against the unpredictability of the domain's behavior. When domains are highly unpredictable, it may become extremely difficult to take every contingency into account, to the extent that "strong" plans that guarantee success in all cases may not exist. In those cases, it is necessary to either relax the goal, or recur to on-line planning.

In on-line planning, the idea is that one explicitly admits plan whose execution may not satisfy, in some cases or even in all cases, the goal. To compensate for this, in this approach, monitoring is used not just for a sanity check, but also to evaluate the reached situation in the domain, so to trigger again novel plan construction whenever needed. That is, the planner is now part of the execution architecture, as shown in Fig. 2.5.

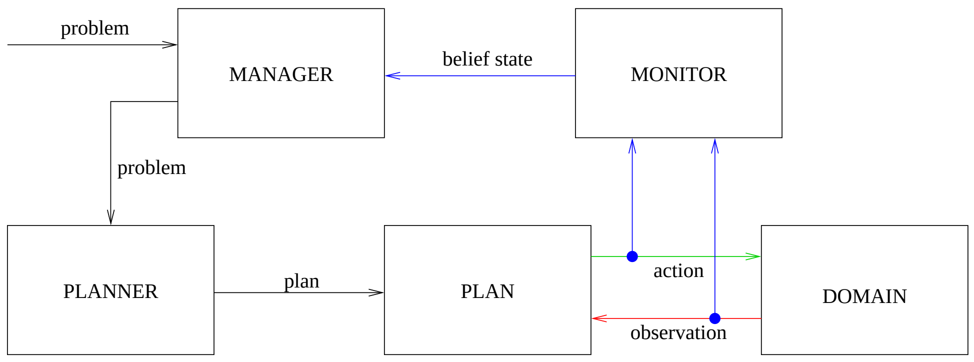


Figure 2.5: The on-line framework.

In both approaches, the planner's performance is crucial to the applicability of the resulting architecture; in the next chapter, we discuss the relationship of planning with model checking, and in particular, the benefits of making use of symbolic model checking techniques.

Chapter 3

Planning and Model Checking

In this chapter we discuss the Planning via Symbolic Model Checking approach. We first present the ideas underlying Symbolic Model Checking (SMC), and point out what our approach inherits from SMC and in which directions it extends SMC (Section 3.1). Then, we show how planning domains and primitives can be represented in terms of logical, symbolic transformations (Sections 3.2 and 3.3). Finally, we discuss Binary Decision Diagrams (BDDs), the machinery for an efficient implementation of the symbolic approach (Section 3.4).

3.1 Symbolic Model Checking: Overview and Discussion

Model checking is a formal verification technique, where a reactive system (e.g., a communication protocol, a hardware design) is modeled as a Finite State Machine (FSM). Requirements over the behaviors of the system are modeled as formulæin a temporal logic, for instance Computation Tree Logic (CTL) [Eme90]. The model checking problem $\mathcal{M} \models \phi$ is to detect if all the behaviors of the FSM $\mathcal{M}$ satisfy the constraints specified by the temporal formula ϕ. Model checking algorithms are based on the exhaustive exploration of the FSM [CGP99]. When the specification is not satisfied, they are able to construct a counterexample, i.e., to produce a description of the system behavior that does not satisfy the specification.

Our approach is related to model checking by the fact that a planning domain is represented as a FSM, and we inherit from model checking the standard techniques for the representation and traversal of FSMs. Thus, there is substantial ground to tackle planning by means of model checking techniques, as first proposed in [GT00, CGGT97, MGR98]. The main difference between the two problems is that planning tackles the more complex problem of *finding* a plan such that a certain behavior is achieved when the plan is executed in the domain. There are cases in which a planning problem can be reduced to a model checking problem. This is the case, for instance, of classical planning, i.e., planning for reachability goals in deterministic domains, where finding a plan corresponds to the model checking problem of finding one path from the initial state to the goal; the plan to the goal is the sequence of actions that produce the path. In the general case of nondeterministic domains, however, a planning problem cannot be reduced to model checking. Consider for instance the case of a strong planning problem: it requires to *find* a suitable restriction of the behaviors of the domain, i.e., the plan, such that, on *all* the compatible paths, a goal state is reached.

There are further relationships between model checking and planning. The execution structure induced from the connection of a plan π to the domain $\mathcal{D}$ can be presented as a simple synchronous composition of FSMs, $\mathcal{M}_{\mathcal{D}} \times \mathcal{M}_{\pi}$. $\mathcal{M}_{\mathcal{D}}$ describes the domain, while $\mathcal{M}_{\pi}$ represents the plan. Moreover, weak, strong and strong cyclic goals can be expressed in temporal logic. For instance, strong solutions can be expressed by the CTL formula AF($\mathcal{G}$), read "for all paths, there is a future instant where $\mathcal{G}$ holds".[1] Thus, plan *validation* can be carried out with a straightforward application of standard model checking techniques, by checking if the formula corresponding to the goal holds on the execution structure. For instance, $\mathcal{M}_{\mathcal{D}} \times \mathcal{M}_{\pi} \models AF(\mathcal{G})$ is the model checking problem corresponding to the validation that plan π is a strong solution for goal $\mathcal{G}$.

Symbolic model checking [McM93] is a particular form of model checking, where propositional formulæare used for the compact representation of FSMs, and transformations over propositional formulæprovide a basis for efficient exploration. The symbolic encoding is efficiently implemented by means of BDDs [Bry86]. This allows for the analysis of extremely large systems [BCM$^+$92]. As a result, symbolic model checking is routinely applied in industrial hardware design, and is taking up in other application domains (see [CW96] for a survey). In our approach, we inherit the symbolic mechanisms and the BDD-based implementation techniques for representing and exploring planning domains. We extend these techniques with a symbolic representation of state-action tables, and with algorithms that are able to synthesize the plan during the exploration of the state space of the FSM.

3.2 Symbolic Representation of Planning Domains

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

A state is the set of propositions of $\mathcal{P}$ that are intended to hold in it. For each state s, there is a corresponding assignment to the state variables in $\boldsymbol{x}$, i.e., the assignment where

[1]See [PT01a] for a more thorough discussion of the relations with temporally extended goals.

each variable in s is assigned to $True$, and all the other variables are assigned to $False$. We represent s with a propositional formula $\xi(s)$, having such an assignment as its unique satisfying assignment. This representation naturally extends to any *set of states* $Q \subseteq \mathcal{S}$ as follows:

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

In this way, we associate a set of states with the generalized disjunction of the formulærepresenting each of the states. The satisfying assignments of $\xi(Q)$ are exactly the assignments representing any state in Q. We can use such a formula to represent the set $\mathcal{S}$ of all the states of the domain. We use a propositional formula as representative for the set of assignments that satisfy it (and hence for the corresponding set of states), so we abstract away from the actual syntax of the formula used: we do not distinguish among equivalent formulæas they represent the same sets of assignments. (Although the actual syntax of the formula may have a computational impact, the use of BDDs as representatives of sets of models is indeed practical.) The main efficiency of the symbolic representation is in that the cardinality of the represented set is not directly related to the size of the formula. For instance, in the limit cases, $\xi(2^{\mathcal{P}})$ and $\xi(\emptyset)$, are the $True$ and $False$ formulæ, independently of the cardinality of $\mathcal{P}$. As a further advantage, the symbolic representation can deal quite effectively with irrelevant information. For this reason, a symbolic representation can have a dramatic improvement over an explicit, enumerative representation. This is what allows symbolic model checkers to handle finite state automata with a very large number of states (see for instance [BCM$^+$92]).

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

$$\begin{array}{lcl}
\xi(Q_1 \cup Q_2) & \doteq & \xi(Q_1) \vee \xi(Q_2) \\
\xi(Q_1 \cap Q_2) & \doteq & \xi(Q_1) \wedge \xi(Q_2) \\
\xi(\mathcal{S} \backslash Q) & \doteq & \xi(\mathcal{S}) \wedge \neg \xi(Q).
\end{array}$$

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

In order to represent actions, we use another set of propositional variables, called *action* variables, written $\boldsymbol{a}$. One approach is to use one action variable for each possible action in $\mathcal{A}$. Intuitively, an action variable is true if and only if the corresponding action is being executed. In principle this allows for the representation of concurrent actions. If a sequential encoding is used, i.e., no concurrent actions are allowed, a mutual exclusion constraint stating that exactly one of the action variables must be true at each time must imposed. In the following, we call $\textsc{Seq}_{\mathcal{A}}(\boldsymbol{a})$ the formula, in the action variables, expressing the mutual exclusion constraint over $\mathcal{A}$. In the specific case of sequential encoding, it is possible to use only $\lceil \log \|\mathcal{A}\| \rceil$ action variables, where each assignment to the action variables denotes a specific action to be executed. Furthermore, being two assignments mutually exclusive, the constraint $\textsc{Seq}(\boldsymbol{a})$ needs not to be represented. When the cardinality of $\mathcal{A}$ is not a power of two, the standard solution is to associate more than one assignment to certain values.

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

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

We call the operation $\Phi[\boldsymbol{x}/\boldsymbol{x}']$ "forward shifting", because it transforms the representation of a set of "current" states in the representation of a set of "next" states. The dual operation $\Phi[\boldsymbol{x}'/\boldsymbol{x}]$ is called "backward shifting". In the following, we call $\boldsymbol{x}$ *current* state variables to distinguish them from next state variables.

A transition is represented as an assignment to $\boldsymbol{x}$, $\boldsymbol{a}$ and $\boldsymbol{x}'$.

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

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

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

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

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

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

QBF formulæallow for an exponentially more compact representation than propositional formulæ.

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

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

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

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

To encode observations, we exploit an additional set $\boldsymbol{o}$ of *boolean observation variables*, distinct from state and action variables. Similarly to what we have shown for state variables, an observation $o \in \mathcal{O}$ corresponds to a complete assignment of observation variables, i.e. by a BDD in the variables $\boldsymbol{o}$, $\xi(o) = \bigwedge_{y_i \in \boldsymbol{o}_T} \xi(y_i) \wedge \bigwedge_{y_i \in \boldsymbol{o}_F} \neg\xi(o_i)$, with $\boldsymbol{o}_T \cup \boldsymbol{o}_F = \boldsymbol{o}$ and $\boldsymbol{o}_T \cap \boldsymbol{o}_F = \emptyset$. A partial assignment of observation variables is represented by a BDD $P = \bigwedge_{o_i \in \boldsymbol{o}_T} o_i \wedge \bigwedge_{o_i \in \boldsymbol{o}_F} \neg o_i$, with $\boldsymbol{o}_T \cup \boldsymbol{o}_F \subset \boldsymbol{o}$ and $\boldsymbol{o}_T \cap \boldsymbol{o}_F = \emptyset$, and corresponds to the set of observations that are compatible with such a constraint, i.e. $\{o \in \mathcal{O} : \xi(o) \models P\}$.

Observation variables allow a direct presentation for the very wide class of domains that allow inquiring about the state of the domain by means of a set of sensors. In this case, each sensor can be mapped onto one or more observation variables, depending on whether the sensor values range over a binary or an n-elements set.

Each observation variable y is associated with a pair of sets of states, which we indicate, overloading the notation, with $\mathcal{X}^-(y)$ and $\mathcal{X}^-(y)$. $\mathcal{X}^-(y)$ represents the states compatible with y, while $\mathcal{X}^-(\overline{y})$ represents the states compatible with $\neg y$. That is,

$$\mathcal{X}^-(y) = \bigcup_{o \in \mathcal{O}: o \models y} \mathcal{X}^-(o)$$

$$\mathcal{X}^-(\overline{y}) = \bigcup_{o \in \mathcal{O}: o \models \neg y} \mathcal{X}^-(\overline{o})$$

Considering a (possibly) partial assignment $P = \bigwedge_{y_i \in \boldsymbol{o}_T} y_i \wedge \bigwedge_{y_i \in \boldsymbol{o}_F} \neg y_i$, with $\boldsymbol{o}_T \cup \boldsymbol{o}_F \subseteq \boldsymbol{o}$ and $\boldsymbol{o}_T \cap \boldsymbol{o}_F = \emptyset$, the states compatible with some observation amongst those represented by P is indicated with $\mathcal{X}^-(P)$:

$$\mathcal{X}^-(P) = \bigcap_{y_i \in \boldsymbol{o}, P \models y_i} \mathcal{X}^-(y_i) \cap \bigcap_{y_i \in \boldsymbol{o}, P \models \neg y_i} \mathcal{X}^-(\overline{y_i})$$

These sets are symbolically represented by means of BDDs in the state variables, which we indicate as $\mathcal{X}^-(y, \boldsymbol{x})$ and $\mathcal{X}^-(\overline{y}, \boldsymbol{x})$, and which can be easily computed by means of BDD set operations over the BDD representations of $\mathcal{X}^-(y_i)$ and $\mathcal{X}^-(\overline{y_i})$:

$$\mathcal{X}^{-}(P, \boldsymbol{x}) = \bigwedge_{y_i \in \boldsymbol{o}', P \models y_i = \top} \mathcal{X}^{-}(y_i, \boldsymbol{x}) \wedge \bigwedge_{y_i \in \boldsymbol{o}', P \models y_i = \bot} \mathcal{X}^{-}(\overline{y_i}, \boldsymbol{x})$$

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

3.3 Symbolic Representation of Plans

The machinery for the symbolic representation of planning domains can be used to represent and manipulate symbolically the structures of the planning algorithms. More specifically, let us consider plans represented as state-action tables. A state-action table *SA* is a relation between states and actions, and can be represented symbolically as a formula in the $\boldsymbol{x}$ and $\boldsymbol{a}$ variables. In the following, we write *SA* $(\boldsymbol{x}, \boldsymbol{a})$ for the formula corresponding to state-action table *SA*: each satisfying assignment to *SA* $(\boldsymbol{x}, \boldsymbol{a})$ represents a state-action pair in *SA*. This view inherits all the properties seen above for sets of states. For instance, the symbolic representation of the union of two state-action tables $SA_1 \cup SA_2$ is represented by the disjunction of their symbolic representations $SA_1(\boldsymbol{x}, \boldsymbol{a}) \vee SA_2(\boldsymbol{x}, \boldsymbol{a})$. The set of states of a state-action table STATESOF$(SA(\boldsymbol{x}, \boldsymbol{a}))$ is represented symbolically as $\exists \boldsymbol{a}. SA(\boldsymbol{x}, \boldsymbol{a})$. The set of actions associated in SA with a given state s, i.e., the set of possible results for the GETACTION primitive in the reactive execution loop presented in Section 4.3.1, is represented symbolically by $\exists \boldsymbol{x}.(SA(\boldsymbol{x}, \boldsymbol{a}) \wedge \xi(s))$. The symbolic representation of the universal state-action table *UnivSA* is $\exists \boldsymbol{x}'. \mathcal{R}(\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{x}')$, that also represents the applicability relation of actions in states.

Of course, state-action tables are just one of the cases where the symbolic representation is extremely useful. The algorithms used to solve the various planning problems which will be presented all make use of symbolic primitives to manipulate sets of states presented as formulæ. For instance, pruning primitives, and different (forward, backward, strong, weak) image primitives, can all be expressed as QBF formulæin the style of the one presented for the standard (forward) image.

3.4 Binary Decision Diagrams

Reduced Ordered Binary Decision Diagrams [Bry86] (in the following simply called BDDs) are the first and most popular implementation device of symbolic model checking (see also [BCCZ99, ABE00] for alternative symbolic representation mechanisms).

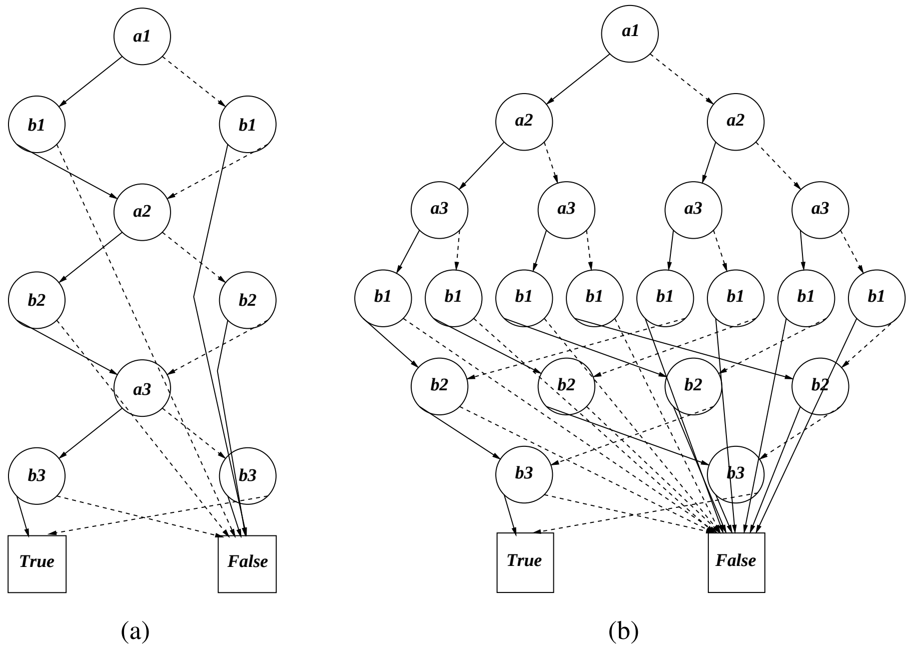


Figure 3.1: Two BDD for the formula $(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2) \wedge (a_3 \leftrightarrow b_3)$.

BDDs provide a general interface that allows for a direct map of the symbolic representation mechanisms presented in previous section (e.g., tautology checking, quantification, shifting).

A BDD is a directed acyclic graph (DAG). The terminal nodes are either $True$ or $False$. Each non-terminal node is associated with a Boolean variable, and two BDDs, called left and right (or high and low) branches. Figure 3.1 (a) depicts a BDD for $(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2) \wedge (a_3 \leftrightarrow b_3)$. At each non-terminal node, the right (left, respectively) branch is depicted as a solid (dashed, resp.) line, and represents the assignment of the value $True$ ($False$, resp.) to the corresponding variable. BDDs provide a canonical representation of Boolean functions. Given a BDD, the value of the function corresponding to a given truth assignment to the variables is determined by traversing the graph from the root to the leaves, following each branch indicated by the value assigned to the variables. (A path from the root to a leaf can visit nodes associated with a subset of all the variables of the BDD. See for instance the path associated with $a_1, \neg b_1$ in Figure 3.1(a).) The reached leaf node is labeled with the resulting truth value. If v is a BDD, its size $\|v\|$ is the number of its nodes. If n is a node, $var(n)$ indicates the variable indexing node n.

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

BDDs can be combined with the usual Boolean transformations (e.g., negation, conjunction, disjunction). Given two BDDs, for instance, the conjunction operator builds and returns the BDD corresponding to the conjunction of its arguments. Substitution and quantification can also be efficiently represented as BDD transformations. In terms of BDD computations, a quantification corresponds to a transformation mapping the BDD of Φ and the variable v_i being quantified into the BDD of the resulting (propositional) formula. The time complexity of the algorithm for computing a truth-functional Boolean transformation $f_1 <op> f_2$ is $O(\|f_1\| \cdot \|f_2\|)$. As far as quantifications are concerned, the time complexity is exponential in the number of variables being quantified.

BDD *packages* are efficient implementations of such data structures and algorithms (see [BRB90]). 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) [BRB90, Som97]. Time efficiency is obtained by maintaining a "computed table", which keeps tracks of the results of recently computed transformations, thus avoiding the re-computation.

A critical computational factor with BDDs is the order of the variables used. (Figure 3.1 shows an example of the impact of a change in the variable ordering on the size of a BDD.) For certain classes of Boolean functions, the size of the corresponding BDD is exponential in the number of variables for any possible variable ordering. In many practical cases, however, finding a good variable ordering is rather easy. Beside affecting the memory used to represent a Boolean function, finding a good variable ordering can have a big impact on computation times, since the complexity of the transformation algorithms depends on the size of the operands. Most BDD packages provide heuristic algorithms for finding good variable orderings, which can be called to try to reduce the overall size of the stored BDDs. The reordering algorithms can also be activated dynamically by the package, during a BDD computation, when the total amount of nodes in the package reaches a predefined threshold (dynamic reordering).

Chapter 4

Off-line planning

In this chapter, we discuss the variety of off-line approaches developed during the course of the project. We start from approaches that act under the assumption that the domain is fully observable. Such simplifying assumption simplifies both the statement of the problem, and its solution by symbolic techniques. In spite of that, a variety of requirements make full sense for such a setting. It is possible to consider just reachability goals, and require that they are obeyed in all cases or just sometimes; it is possible to consider the possibility of looping, and more in general, of complex goals that state conditions that concern the domain state during the whole plan execution, and not just at the end.

We then step to the general case of partially observable domains. Here, things get further complicated, and the general handling of complex goals is an open research issue. We discuss our results for what concerns reachability goals, and the extension to admit loops.

4.1 Weak and Strong Planning under Full Observability

We now want to tackle planning for fully observable domains, and focus on reachability goals, i.e. requiring just conditions on the final state of the domain, at the end of plan execution. From now on, in this section, we will state our domain by letting the observation function implicitly defined to be the identity function.

We remark that, while this is a restrictive setting w.r.t. our general framework, it is far beyond what has been handled by the vast majority of the planning approaches until the last few years: in the "classical planning" approach, no form of unpredictability is accounted for, therefore ruling out a very large number of interesting and realistic scenarios.

4.1.1 Problem Statement

To state the problem, we proceed by specializing the general model of plan to our purposes, and we express the problem in terms of plan executions. In particular, to control fully observable (but nondeterministic) domains to reach some final state, it is enough to think of plans structured as sets of pairs that associate a set of actions to a state that can be executed in such state. We call them *state-action tables*. They resemble universal plans [Sch87] and policies [BBS95, BDH96]. Such plans express conditional and iterative behaviors, and can be easily thought of as a particular representation of general plans, where the context is the current domain state, and the evolution functions are described by the domain relation transition (for the context) and by the state-action association (for the action).

Definition 4.1.1 (State-Action Table) *A* state-action table π *for a planning domain* $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ *is a set of pairs* $\langle s, a \rangle$*, where* $s \in \mathcal{S}$ *and* $a \in \mathcal{A}(s)$*. A state-action table* π *is deterministic if for any state* s *there is at most one action* a *such that* $\langle s, a \rangle \in \pi$*, otherwise it is nondeterministic.*

We call $\langle s, a \rangle$ a *state-action pair*. According to Definition 4.1.1, action a of state-action pair $\langle s, a \rangle$ must be executable in s. Hereafter, we write $\textsc{StatesOf}\pi$ for the set of states of the state-action table π:

$$\textsc{StatesOf}\pi \doteq \{s : \langle s, a \rangle \in \pi\}.$$

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

Definition 4.1.2 (Execution Structure) *Let* π *be a state-action table of a planning domain* $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$*. The* execution structure *induced by* π *from the set of initial states* $\mathcal{I} \subseteq \mathcal{S}$ *is a tuple* $K = \langle Q, T \rangle$*, where* $Q \subseteq \mathcal{S}$ *and* $T \subseteq \mathcal{S} \times \mathcal{S}$ *are the minimal sets satisfying the following rules:*

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

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

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

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

Definition 4.1.3 (Execution Path) *Let $K = \langle Q, T \rangle$ be the execution structure induced by a state-action table π from $\mathcal{I}$. An* execution path *of K from $s_0 \in \mathcal{I}$ is a possibly infinite sequence $s_0, s_1, s_2, \ldots$ of states in Q such that, for all states s_i in the sequence:*

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

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

Definition 4.1.4 (Planning Problem) *Let $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ be a planning domain. A* planning problem *for $\mathcal{D}$ is a triple $\langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{I} \subseteq \mathcal{S}$ and $\mathcal{G} \subseteq \mathcal{S}$.*

The above definition takes into account two forms of nondeterminism. First, we have a set of initial states, and not a single initial state. This allows for expressing partially specified initial conditions. Second, the execution of an action from a state results in a set of states, and not necessarily in a single state (see $\mathcal{R}$ in Definition 2.1.1). This allows for expressing nondeterministic action executions.

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

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

Definition 4.1.5 (Deterministic Solutions) *Let $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ be a planning domain. Let $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a planning problem. Let π be a deterministic state-action table for $\mathcal{D}$. Let $K = \langle Q, T \rangle$ be the execution structure induced by π from $\mathcal{I}$.*

1. *π is a* weak solution *to P if for any state in $\mathcal{I}$ some terminal state of K is reachable that is also in $\mathcal{G}$.*

2. *π is a* strong solution *to P if K is acyclic and all terminal states of K are in $\mathcal{G}$.*

The execution of a state-action table is defined to terminate only when there is no action associated to the terminal state in the state-action table. Therefore, in Definition 4.1.5 we do not consider to be successful those execution paths that contain goal states but that terminate in a state that is not in $\mathcal{G}$.

Weak solutions are plans that may achieve the goal, but are not guaranteed to do so. This amounts to saying that at least one of the many possible execution paths of the state-action table should result in a terminal state that is a goal state. *Strong solutions* are plans that are guaranteed to achieve the goal in spite of nondeterminism, i.e., all the execution paths should result in a terminal state that is a goal state.

In the general case of *nondeterministic* state-action tables, we say that π is a strong (weak) solution to a planning problem if all the determinizations of π are, according to Definition 4.1.5. Formally, a determinization of π is any deterministic state-action table $\pi_d \subseteq \pi$ such that $\text{STATESOF}(\pi_d) = \text{STATESOF}(\pi)$. In this way, we model the fact that the executor can choose any action arbitrarily when building the deterministic state-action table, i.e., there are no compatibility constraints among the actions in different states. This guarantees that the plan is correct also in the general case where the executor selects at run-time one action among the possible actions associated to the current state by the state-action table.

Definition 4.1.6 (Nondeterministic Solutions) *Let $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ be a planning domain. Let $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a planning problem. Let π be a (nondeterministic) state-action table for $\mathcal{D}$. π is a weak (resp. strong, strong cyclic) solution to P if all the determinizations π_d of π are weak (resp. strong, strong cyclic) solutions to P according to Definition 4.1.5.*

4.1.2 Planning Algorithms

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

The algorithms, presented in Figure 4.1, are based on a breadth-first search proceeding backwards from the goal, towards the initial states. For both algorithms, at each iteration step, the set of states for which a solution has been already found is used as a target for the expansion preimage routine at line 5, that returns a new "slice" to be added to the state-action table under construction. The algorithms are actually identical, except

```
function WeakPlan($I, G$);
  $OldSA := Fail$;
  $SA := \emptyset$;
  while ($OldSA \neq SA \land I \not\subseteq (G \cup \textsc{StatesOf}(SA))$) do
    $PreImage := \textsc{WeakPreImage}(G \cup \textsc{StatesOf}(SA))$;
    $NewSA := \textsc{PruneStates}(PreImage, G \cup \textsc{StatesOf}(SA))$;
    $OldSA := SA$;
    $SA := SA \cup NewSA$;
  done;
  if ($I \subseteq (G \cup \textsc{StatesOf}(SA))$) then
    return $SA$;
  else
    return $Fail$;
  fi;
end;
```

```
function StrongPlan($I, G$);
  $OldSA := Fail$;
  $SA := \emptyset$;
  while ($OldSA \neq SA \land I \not\subseteq (G \cup \textsc{StatesOf}(SA))$) do
    $PreImage := \textsc{StrongPreImage}(G \cup \textsc{StatesOf}(SA))$;
    $NewSA := \textsc{PruneStates}(PreImage, G \cup \textsc{StatesOf}(SA))$;
    $OldSA := SA$;
    $SA := SA \cup NewSA$;
  done;
  if ($I \subseteq (G \cup \textsc{StatesOf}(SA))$) then
    return $SA$;
  else
    return $Fail$;
  fi;
end;
```

Figure 4.1: The algorithms for weak and strong planning.

for the fact that the extension primitive is the function WEAKPREIMAGE in the case of weak planning, and STRONGPREIMAGE in the case of strong planning. Functions WEAKPREIMAGE and STRONGPREIMAGE are defined as follows:

$$\begin{array}{rcl} \textsc{WeakPreImage}(S) & \doteq & \{\langle s,\, a\rangle : \textsc{Exec}(s,a) \cap S \neq \emptyset\}, \\ \textsc{StrongPreImage}(S) & \doteq & \{\langle s,\, a\rangle : \emptyset \neq \textsc{Exec}(s,a) \subseteq S\}. \end{array}$$

Intuitively, WEAKPREIMAGE(S) returns the set of state-action pairs $\langle s,\, a\rangle$ such that the execution of a in s may lead inside S. STRONGPREIMAGE(S) returns the set of state-action pairs $\langle s,\, a\rangle$ such that the execution of a in s is guaranteed to lead to states inside S, regardless of nondeterminism.

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

$$\textsc{PruneStates}(\pi, S) \doteq \{\langle s,\, a\rangle \in \pi : s \notin S\}.$$

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

The weak and strong planning algorithms proposed above are correct, namely they always terminate and that they return a solution if (and only if) such a solution exists.

Theorem 4.1.7 (Correctness) *Let $\pi \doteq$ WEAKPLAN$(\mathcal{I}, \mathcal{G}) \neq$ Fail. Then π is a weak solution of the planning problem $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G}\rangle$.*
If WEAKPLAN$(\mathcal{I}, \mathcal{G}) =$ Fail, instead, then there is no weak solution for planning problem P.

Theorem 4.1.8 (Termination) *Function WEAKPLAN always terminates.*

Theorem 4.1.9 (Correctness) *Let $\pi \doteq$ STRONGPLAN$(\mathcal{I}, \mathcal{G}) \neq$ Fail. Then π is a strong solution of the planning problem $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G}\rangle$.*
If STRONGPLAN$(\mathcal{I}, \mathcal{G}) =$ Fail, instead, then there is no strong solution for planning problem P.

Theorem 4.1.10 (Termination) *Function STRONGPLAN always terminates.*

The proofs of the theorems can be found in the Appendix, together with a proof of the fact that, thanks to the (symbolic) breadth-first nature of the search, the state-action table returned by the algorithms is “optimal” w.r.t. a suitable measure of the distance of a state from the goal.

The two algorithms have been thoroughly tested and compared against the state-of-the art systems in planning for nondeterministic domains, using standard benchmarks and providing new ones. Results are presented in [CPRT03].

4.2 Strong Cyclic Planning under Full Observability

In this section we tackle the problem of strong cyclic planning. The main difference with the algorithms presented in previous section is that here the resulting plans allow for infinite behaviors: loops must no longer be eliminated, but rather controlled, i.e., only certain, "good" loops must be kept.

Strong cyclic solutions formalize the intuitive notion of "acceptable" iterative trial-and-error strategies: all their partial execution paths can be extended to a finite execution path whose terminal state is a goal state. Strong cyclic solutions can produce executions that loop forever. However, this can only happen under an infinite sequence of "failures" in the execution of some action. That is, all the infinite execution paths are "unfair", since they eventually enter loops where some actions are executed infinitely often in given states, but some of its outcomes (the ones leading to the goal) never occur.

4.2.1 Problem Statement

We formalize the notion of strong cyclic solutions as follows.

Definition 4.2.1 (Deterministic Solutions) *Let $\mathcal{D} = \langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ be a planning domain. Let $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a planning problem. Let π be a deterministic state-action table for $\mathcal{D}$. Let $K = \langle Q, T \rangle$ be the execution structure induced by π from $\mathcal{I}$.*

π is a strong cyclic solution *to P if from any state in Q some terminal state of K is reachable and all the terminal states of K are in $\mathcal{G}$.*

4.2.2 Planning Algorithm

The strong cyclic planning algorithm is presented in Figure 4.2. The algorithm starts to analyze the universal state-action table with respect to the problem being solved, and eliminates all those state-action pairs which are discovered to be source of potential "bad" loops, or to lead to states which have been discovered not to allow for a solution. With respect to the algorithms presented in previous section, here the set of states associated with the state-action table being constructed is reduced rather than being extended: this approach amounts to computing a greatest fix-point.

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

$$\mathit{UnivSA} \doteq \{\langle s, a \rangle : a \in \mathcal{A}(s)\}.$$

```
function StrongCyclicPlan($(I, G)$);
  $OldSA := \emptyset$;
  $SA := UnivSA$;
  while $(OldSA \neq SA)$ do
    $OldSA := SA$;
    $SA :=$ PruneUnconnected(PruneOutgoing($SA, G$), $G$);
  done;
  if $(I \subseteq (G \cup$ StatesOf($SA$))) then
    return RemoveNonProgress($SA, G$);
  else
    return $Fail$;
  fi;
end;
```

```
function PruneUnconnected($(SA, G)$);
  $NewSA := \emptyset$;
  repeat
    $OldSA := NewSA$;
    $NewSA := SA \cap$ WeakPreImage($G \cup$ StatesOf($NewSA$));
  until $(OldSA = NewSA)$;
  return $NewSA$;
end;
```

```
function PruneOutgoing($(SA, G)$);
  $NewSA := SA \setminus$ ComputeOutgoing($SA, G \cup$ StatesOf($SA$));
  return $NewSA$;
end;
```

```
function RemoveNonProgress($(SA, G)$);
  $NewSA := \emptyset$;
  repeat
    $PreImage := SA \cap$ WeakPreImage($G \cup$ StatesOf($NewSA$));
    $OldSA := NewSA$;
    $NewSA := NewSA \cup$ PruneStates($PreImage, G \cup$ StatesOf($NewSA$));
  until $(OldSA = NewSA)$;
  return $NewSA$;
end;
```

Figure 4.2: The algorithm for strong cyclic planning.

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

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

In particular, it may contain state-action pairs like that, while preserving the reachability of the goal, still do not perform any progress toward it.

In the strong cyclic planning algorithm, function RemoveNonProgress on line 9 takes care of removing all those actions from a state whose outcomes do not lead to any progress toward the goal. This function is very similar to the weak planning algorithm: it iteratively extends the state-action table by considering states at an increasing distance from the goal. In this case, however, the weak preimage computed at any iteration step is restricted to the state-action pairs that appear in the input state-action table, and hence that are “safe” according to the elimination phase.

Functions PruneOutgoing, PruneUnconnected, and RemoveNonProgress are presented in Figure 4.2. They are based on the primitives WeakPreImage and PruneStates already defined in Section 4.1.2, and on the primitive ComputeOutgoing, that takes as input a state-action table SA and a set of states S, and returns those state-action pairs which are not guaranteed to result in states in S:

$$\textsc{ComputeOutgoing}(SA, S) \doteq \{\langle s, a\rangle \in SA : \textsc{Exec}(s, a) \not\subseteq S\}.$$

Similar to the weak and strong planning algorithm, also the strong cyclic algorithm is correct and terminating (see Appendix):

Theorem 4.2.2 (Correctness) *Let $\pi \doteq \textsc{StrongCyclicPlan}(\mathcal{I}, \mathcal{G}) \neq Fail$. Then π is a strong cyclic solution of the planning problem $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G}\rangle$.*
If $\textsc{StrongCyclicPlan}(\mathcal{I}, \mathcal{G}) = Fail$, instead, then there is no strong cyclic solution for planning problem P.

Theorem 4.2.3 (Termination) *Function StrongCyclicPlan always terminates.*

Moreover, the symbolic implementation of the algorithm is very effective, as shown in [CPRT03], where several tests are run and comparison are taken with the state-of-the art GPT system [BG00].

4.3 Planning for Extended Goals under Full Observability

We now step to more expressive, and complex to deal with, kinds of goals, namely "extended goals". By "extended goals" we mean requirements that specify the way the domain state should evolve in time - opposed to reachability goals, which just require that some condition is valid at the end of plan execution. The increase in expressiveness is evident, as for instance it becomes possible to state that some condition must always be valid, while some other should eventually be reached, and some other must hold until a certain situation arise. Requirements such as the above are typical in several planning applications.

Among the many languages that could be used to formulate extended goals, we chose CTL [Eme90], a branching-time temporal logics where it is possible to express both requirements about "when" conditions should hold (eventually, always, until some other condition), and "in which cases" (in all cases, or sometimes). This is very convenient since both aspects are present when dealing with nondeterministic domains. Formally, extended goals are expressed as follows:

Definition 4.3.1 *Let $\mathcal{B}$ be the set of basic propositions of a domain D and let $b \in \mathcal{B}$. The syntax of an* (extended) goal *g for D is the following:*

$$\begin{array}{rl} g ::= & \top \mid \bot \mid b \mid \neg b \mid g \wedge g \mid g \vee g \mid \mathrm{AX}\, g \mid \mathrm{EX}\, g \mid \\ & \mathrm{A}(g \,\mathrm{U}\, g) \mid \mathrm{E}(g \,\mathrm{U}\, g) \mid \mathrm{A}(g \,\mathrm{W}\, g) \mid \mathrm{E}(g \,\mathrm{W}\, g) \end{array}$$

"X", "U", and "W" are the "next time", "(strong) until", and "weak until" temporal operators, respectively. "A" and "E" are the universal and existential path quantifiers, where a path is an infinite sequence of states. They allow us to specify requirements that take into account non-determinism. Intuitively, the formula $\mathrm{AX}\, g$ ($\mathrm{EX}\, g$) means that g holds in every (in some) immediate successor of the current state. $\mathrm{A}(g_1 \,\mathrm{U}\, g_2)$ ($\mathrm{E}(g_1 \,\mathrm{U}\, g_2)$) means that for every path (for some path) there exists an initial prefix of the path such that g_2 holds at the last state of the prefix and g_1 holds at all the other states along the prefix. The formula $\mathrm{A}(g_1 \,\mathrm{W}\, g_2)$ ($\mathrm{E}(g_1 \,\mathrm{W}\, g_2)$) is similar to $\mathrm{A}(g_1 \,\mathrm{U}\, g_2)$ ($\mathrm{E}(g_1 \,\mathrm{U}\, g_2)$) but allows for paths where g_1 holds in all the states and g_2 never holds. Formulas $\mathrm{AF}\, g$ and $\mathrm{EF}\, g$ (where the temporal operator "F" stands for "future" or "eventually") are abbreviations of $\mathrm{A}(\top \,\mathrm{U}\, g)$ and $\mathrm{E}(\top \,\mathrm{U}\, g)$, respectively. $\mathrm{AG}\, g$ and $\mathrm{EG}\, g$ (where "G" stands for "globally" or "always") are abbreviations of $\mathrm{A}(g \,\mathrm{W}\, \bot)$ and $\mathrm{E}(g \,\mathrm{W}\, \bot)$, respectively. A remark is in order: even if $\neg$ is allowed only in front of basic propositions, it is easy to define $\neg g$ for a generic CTL formula g, by "pushing down" the negations: for instance $\neg\mathrm{AX}\, g \equiv \mathrm{EX}\, \neg g$ and $\neg\mathrm{A}(g_1 \,\mathrm{W}\, g_2) \equiv \mathrm{E}(\neg g_2 \,\mathrm{U}(\neg g_1 \wedge \neg g_2))$.

Goals as CTL formulæallow us to specify different interesting requirements on plans.

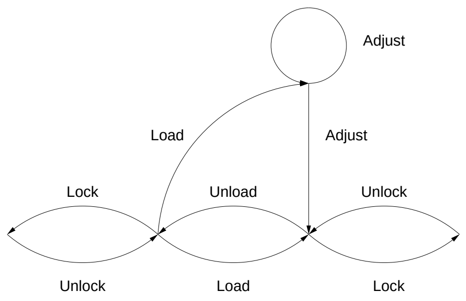


Figure 4.3: A simple non-deterministic domain

Let us consider first some examples of *reachability goals*. $\mathrm{AF}\, g$ ("reach g") states that a condition should be guaranteed to be reached by the plan, in spite of non-determinism. $\mathrm{EF}\, g$ ("try to reach g") states that a condition might possibly be reached, i.e., there exists at least one execution that achieves the goal. As an example, in Figure 4.3, the strong requirement $(\mathsf{locked} \wedge \neg \mathsf{loaded}) \rightarrow \mathrm{AF}\,(\mathsf{locked} \wedge \mathsf{loaded})$ cannot be satisfied, while the weaker requirement $(\mathsf{locked} \wedge \neg \mathsf{loaded}) \rightarrow \mathrm{EF}\,(\mathsf{locked} \wedge \mathsf{loaded})$ can be satisfied by unlocking the container, loading the item and then (if possible) locking the container. A reasonable reachability requirement that is stronger than $\mathrm{EF}\, g$ is $\mathrm{A}(\mathrm{EF}\, g \,\mathrm{W}\, g)$: it allows for those execution loops that have always a possibility of terminating, and when they do, the goal g is guaranteed to be achieved. In Figure 4.3, the goal $(\mathsf{locked} \wedge \neg \mathsf{loaded}) \rightarrow \mathrm{A}(\mathrm{EF}\,(\mathsf{locked} \wedge \mathsf{loaded})\,\mathrm{W}(\mathsf{locked} \wedge \mathsf{loaded}))$ can be satisfied by a plan that unlocks, loads, and, if the outcome is state $\mathsf{3}$, locks again, while if the item is misplaced (state $\mathsf{5}$) repeatedly tries to adjust the position of the item until (hopefully) state $\mathsf{3}$ is reached, and finally locks the container.

We can distinguish among different kinds of *maintainability goals*, e.g., $\mathrm{AG}\, g$ ("maintain g"), $\mathrm{AG}\, \neg g$ ("avoid g"), $\mathrm{EG}\, g$ ("try to maintain g"), and $\mathrm{EG}\, \neg g$ ("try to avoid g"). For instance, a robot should never harm people and should always avoid dangerous areas. Weaker requirements might be needed for less critical properties, like the fact that the robot should try to avoid to run out of battery.

We can *compose reachability and maintainability goals*. $\mathrm{AF}\,\mathrm{AG}\, g$ states that a plan should guarantee that all executions reach eventually a set of states where g can be maintained. For instance, an air-conditioner controller is required to reach eventually a state such that the temperature can then be maintained in a given range. Alternatively, if you consider the case in which a pump might fail to turn on when it is selected, you might require that "there exists a possibility" to reach the condition to maintain the temperature in a desired range ($\mathrm{EF}\,\mathrm{AG}\, g$). As a further example, the goal $\mathrm{AG}\,\mathrm{EF}\, g$ intuitively means "maintain the possibility of reaching g".

Reachability – preserving goals make use of the "until operators" ($\mathrm{A}(g_1 \,\mathrm{U}\, g_2)$ and $\mathrm{E}(g_1 \,\mathrm{U}\, g_2)$) to express reachability goals while some property must be preserved. For instance, an air-conditioner might be required to reach a desired temperature while leaving at least n of its m pumps off.

As a last example in the domain of Figure 4.3, consider the goal "from state 2, where the container is unlocked and empty, lock the container first, and then maintain the possibility of reaching a state where the item is loaded and the container is locked (state 4)". It can be formalized as $(\neg\mathsf{locked} \wedge \neg\mathsf{loaded} \wedge \neg\mathsf{misplaced}) \rightarrow (\mathrm{AF}\,(\mathsf{locked} \wedge \mathrm{AG}\,\mathrm{EF}\,(\mathsf{locked} \wedge \mathsf{loaded})))$. In the rest of the paper, we call this example of goal "*lock-then-load goal*".

Notice that in all examples above, the ability of composing formulæwith universal and existential path quantifiers is essential. Logics that do not provide this ability, like LTL [Eme90], cannot express these kinds of goals[1].

4.3.1 Problem Statement

As usual, to state our problem, we start from defining the expected structure of the solution plan. In this case, we can again slightly simplify the general description of a plan, by only considering a deterministic version of it:

Definition 4.3.2 *A* plan *for a domain D is a tuple $\langle C, c_0, \mathit{act}, \mathit{ctxt}\rangle$, where:*

- *C is a set of (execution) contexts,*
- *$c_0 \in C$ is the initial context,*
- *$\mathit{act} : Q \times C \rightharpoonup A$ is the action function,*
- *$\mathit{ctxt} : Q \times C \times Q \rightharpoonup C$ is the context function.*

If we are in state q and in execution context c, then $\mathit{act}(q, c)$ returns the action to be executed by the plan, while $\mathit{ctxt}(q, c, q')$ associates to each reached state q' the new execution context. Functions *act* and *ctxt* may be partial, since some state-context pairs are never reached in the execution of the plan. An example of a plan that satisfies the lock-then-load goal is shown in Figure 4.4. Notice that the context changes from c_0 to c_1 when the execution reaches state 1. This allows the plan to execute different actions in state 2.

In the rest of the paper we consider only plans that are *executable and complete*. We say that plan π is *executable* if, whenever $\mathit{act}(q, c) = a$ and $\mathit{ctxt}(q, c, q') = c'$, then $q \xrightarrow{a} q'$. We say that π is *complete* if, whenever $\mathit{act}(q, c) = a$ and $q \xrightarrow{a} q'$, then there is some context c' such that $\mathit{ctxt}(q, c, q') = c'$ and $\mathit{act}(q', c')$ is defined. Intuitively, a complete plan always specifies how to proceed for all the possible outcomes of any action in the plan.

The execution of a plan results in a change in the current state and in the current context. It can therefore be described in terms of transitions between pairs state-context. Formally, given a domain D and a plan π, a transition of plan π in D is a tuple $(q, c) \xrightarrow{a} (q', c')$ such that $q \xrightarrow{a} q'$, $a = \mathit{act}(q, c)$, and $c' = \mathit{ctxt}(q, c, q')$. A *run* of plan π from state

[1] In general, CTL and LTL have incomparable expressive power (see [Eme90] for a comparison). We focus on CTL since it provides the ability of expressing goals that take into account non-determinism.

act$(2, c_0)$ = *lock*	*ctxt*$(2, c_0, 1) = c_1$
act$(1, c_1)$ = *unlock*	*ctxt*$(1, c_1, 2) = c_1$
act$(2, c_1)$ = *load*	*ctxt*$(2, c_1, 3) = c_1$
	ctxt$(2, c_1, 5) = c_1$
act$(5, c_1)$ = *adjust*	*ctxt*$(5, c_1, 5) = c_1$
	ctxt$(5, c_1, 3) = c_1$
act$(3, c_1)$ = *lock*	*ctxt*$(3, c_1, 4) = c_1$
act$(4, c_1)$ = *wait*	*ctxt*$(4, c_1, 4) = c_1$

Figure 4.4: An example of plan

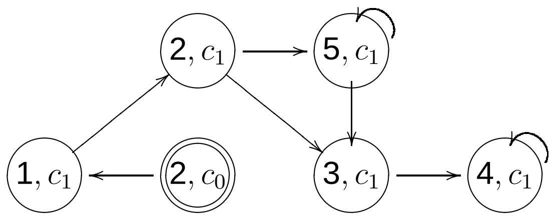


Figure 4.5: An example of execution structure

q_0 is an infinite sequence $(q_0, c_0) \xrightarrow{a_0} (q_1, c_1) \xrightarrow{a_1} (q_2, c_2) \xrightarrow{a_2} (q_3, c_3) \cdots$ where $(q_i, c_i) \xrightarrow{a_i} (q_{i+1}, c_{i+1})$ are transitions. Given a plan, we may have an infinite number of runs due to the non-determinism of the domain. This is the case of the plan in Figure 4.4, since the execution can loop non-deterministically over the pair state-context $(5, c_1)$. We provide a finite presentation of the set of all possible runs of a plan with an *execution structure*, i.e, a Kripke Structure [Eme90] whose set of states is the set of state-context pairs, and whose transition relation corresponds to the transitions of the runs.

Definition 4.3.3 *The* execution structure *of plan π in a domain D from state q_0 is the structure $K = \langle S, R, L \rangle$, where:*

- $S = \{(q, c) :$ *act*(q, c) *is defined*$\}$,
- $((q, c), (q', c')) \in R$ *if* $(q, c) \xrightarrow{a} (q', c')$ *for some* a,
- $L(q, c) = \{b : b \in q\}$

As an example, the execution structure of the plan in Figure 4.4 is depicted in Figure 4.5.

We define when a goal g is true in (q, c), written $K, (q, c) \models g$ by using the standard semantics for CTL formulæover the Kripke Structure K. The complete formal definition can be found in, e.g., [Eme90]. In order to make the paper self contained, we present here some cases. Propositional formulæare treated in the usual way. $K, (q, c) \models \mathrm{AX}\, g$ iff for every path $(q, c)_0(q, c)_1(q, c)_2 \cdots$, with $(q, c) = (q, c)_0$, we have $K, (q, c)_1 \models g$. $K, (q, c) \models \mathrm{A}(g_1 \,\mathrm{U}\, g_2)$ iff for every path $(q, c)_0(q, c)_1(q, c)_2 \cdots$, with $(q, c) = (q, c)_0$, there exists $i \geq 0$ such that $K, (q, c)_i \models g_2$ and, for all $0 \leq j < i$, $K, (q, c)_j \models g_1$.

The definition is similar in the case of existential path quantifiers. We can now define the notion of plan that satisfies a given goal.

Definition 4.3.4 *Let D be a planning domain and g be a goal for D. Let π be a plan for D and K be the corresponding execution structure. Plan π satisfies goal g from initial state q_0, written $\pi, q_0 \models g$, if $K, (q_0, c_0) \models g$. Plan π satisfies goal g from the set of initial states Q_0 if $\pi, q_0 \models g$ for each $q_0 \in Q_0$.*

For instance, the plan in Figure 4.4 satisfies the lock-then-load goal from state 2.

4.3.2 Planning Algorithm

The planning algorithm searches through the domain by trying to satisfy a goal g in a state q. Goal g defines conditions on the current state and on the next states to be reached. Intuitively, if g must hold in q, then some conditions must be projected to the next states. The algorithm extracts the information on the conditions on the next states by "progressing" the goal g. For instance, if g is $\mathrm{EF}\, g'$, then either g' holds in q or $\mathrm{EF}\, g'$ must still hold in some next state, i.e., $\mathrm{EX}\,\mathrm{EF}\, g'$ must hold in q. One of the basic building blocks of the algorithm is the function *progr* that rewrites a goal by progressing it to next states. *progr* is defined by induction on the structure of goals.

- $\mathit{progr}(q, \top) = \top$ and $\mathit{progr}(q, \bot) = \bot$;
- $\mathit{progr}(q, b) =$ if $b \in q$ then $\top$ else $\bot$;
- $\mathit{progr}(q, \neg b) =$ if $b \in q$ then $\bot$ else $\top$;
- $\mathit{progr}(q, g_1 \wedge g_2) = \mathit{progr}(q, g_1) \wedge \mathit{progr}(q, g_2)$;
- $\mathit{progr}(q, g_1 \vee g_2) = \mathit{progr}(q, g_1) \vee \mathit{progr}(q, g_2)$;
- $\mathit{progr}(q, \mathrm{AX}\, g) = \mathrm{AX}\, g$ and $\mathit{progr}(q, \mathrm{EX}\, g) = \mathrm{EX}\, g$;
- $\mathit{progr}(q, \mathrm{A}(g\, \mathrm{U}\, g')) = (\mathit{progr}(q, g) \wedge \mathrm{AX}\,\mathrm{A}(g\, \mathrm{U}\, g')) \vee \mathit{progr}(q, g')$ and $\mathit{progr}(q, \mathrm{E}(g\, \mathrm{U}\, g')) = (\mathit{progr}(q, g) \wedge \mathrm{EX}\,\mathrm{E}(g\, \mathrm{U}\, g')) \vee \mathit{progr}(q, g')$;
- $\mathit{progr}(q, \mathrm{A}(g\, \mathrm{W}\, g')) = (\mathit{progr}(q, g) \wedge \mathrm{AX}\,\mathrm{A}(g\, \mathrm{W}\, g')) \vee \mathit{progr}(q, g')$ and $\mathit{progr}(q, \mathrm{E}(g\, \mathrm{W}\, g')) = (\mathit{progr}(q, g) \wedge \mathrm{EX}\,\mathrm{E}(g\, \mathrm{W}\, g')) \vee \mathit{progr}(q, g')$.

The formula $\mathit{progr}(q, g)$ can be written in a normal form. We write it as a disjunction of two kinds of conjuncts, those of the form $\mathrm{AX}\, f$ and those of the form $\mathrm{EX}\, h$, since we need to distinguish between formulæthat must hold in all the next states and those that must hold in some of the next states:

$$\mathit{progr}(q, g) = \bigvee_{i \in I} \Big(\bigwedge_{f \in A_i} \mathrm{AX}\, f \wedge \bigwedge_{h \in E_i} \mathrm{EX}\, h \Big)$$

where $f \in A_i$ ($h \in E_i$) if $\mathrm{AX}\, f$ ($\mathrm{EX}\, h$) belongs to the i-th disjunct of $\mathit{progr}(q, g)$. We have $|I|$ different disjuncts that correspond to alternative evolutions of the domain, i.e., to alternative plans we can search for. In the following, we represent $\mathit{progr}(q, g)$ as a set of pairs, each pair containing the A_i and the E_i parts of a disjunct:

$$\mathit{progr}(q, g) = \{(A_i, E_i) \mid i \in I\}$$

with $\mathit{progr}(q, \top) = \{(\emptyset, \emptyset)\}$ and $\mathit{progr}(q, \bot) = \emptyset$.

Given a disjunct (A, E) of $\mathit{progr}(q, g)$, we can define a function that assigns goals to be satisfied to the next states. We denote with $\mathit{assign\text{-}progr}((A, E), Q)$ the set of all the possible assignments $i : Q \to 2^{A \cup E}$ such that each universally quantified goal is assigned to all the next states (i.e., if $f \in A$ then $f \in i(q)$ for all $q \in Q$) and each existentially quantified goal is assigned to one of the next states (i.e., if $h \in E$ and $h \notin A$ then $f \in i(q)$ for one particular $q \in Q$). Consider the following example in the domain of Figure 4.3. Let g be $\mathrm{AF}\,\mathsf{locked} \wedge \mathrm{EX}\,\mathsf{misplaced} \wedge \mathrm{EX}\,\mathsf{loaded}$ and let the current state q be $\mathsf{2}$. We have that $\mathrm{AX}\,\mathrm{AF}\,\mathsf{locked} \wedge \mathrm{EX}\,\mathsf{misplaced} \wedge \mathrm{EX}\,\mathsf{loaded} \in \mathit{progr}(\mathsf{2}, g)$. If we consider action *load*, the next states are $\mathsf{3}$ and $\mathsf{5}$. Then $\mathrm{AF}\,\mathsf{locked}$ must hold in $\mathsf{3}$ *and* in $\mathsf{5}$, while $\mathsf{misplaced}$ and loaded must hold in $\mathsf{3}$ *or* in $\mathsf{5}$. We have therefore four possible state-formulæassignments $i_1, \ldots, i_4$ to be explored (in the following we write f_1 for $\mathrm{AF}\,\mathsf{locked}$, h_1 for $\mathsf{misplaced}$, and h_2 for loaded):

$$\begin{array}{ll}
i_1(\mathsf{3}) = f_1 \wedge h_1 \wedge h_2 & i_1(\mathsf{5}) = f_1 \\
i_2(\mathsf{3}) = f_1 \wedge h_1 & i_2(\mathsf{5}) = f_1 \wedge h_2 \\
i_3(\mathsf{3}) = f_1 \wedge h_2 & i_3(\mathsf{5}) = f_1 \wedge h_1 \\
i_4(\mathsf{3}) = f_1 & i_4(\mathsf{5}) = f_1 \wedge h_1 \wedge h_2
\end{array}$$

In this simple example, it is easy to see that the only assignment that may lead to a successful plan is i_3.

Given the two basic building blocks *progr* and *assign-progr*, we can now describe the planning algorithm *build-plan* that, given a goal g_0 and an initial state q_0, returns either a plan or a failure.[2] The algorithm is reported in Figure 4.6. It performs a depth-first forward search: starting from the initial state, it picks up an action, progresses the goal to successor states, and iterates until either the goal is satisfied or the search path leads to a failure. The algorithm uses as the "contexts" of the plan the list of the active goals that are considered at the different stages of the exploration. More precisely, a context is a list $c = [g_1, \ldots, g_n]$, where the g_i are the active goals, as computed by functions *progr* and *assign-progr*, and the order of the list represents the *age* of these goals: the goals that are active since more steps come first in the list.

The main function of the algorithm is function $\mathit{build\text{-}plan\text{-}aux}(q, c, \mathit{pl}, \mathit{open})$, that builds the plan for context c from state q. If a plan is found, then it is returned by the function. Otherwise, $\bot$ is returned. Argument *pl* is the plan built so far by the algorithm. Initially, the argument passed to *build-plan-aux* is $\mathit{pl} = \langle C, c_0, \mathit{act}, \mathit{ctxt}\rangle = \langle \emptyset, g_0, \emptyset, \emptyset\rangle$. Argument *open* is the list of the pairs state-context of the currently open problems: if

[2]It is easy to extend the algorithm to the case of more than one initial state.

```
function build-plan($q_0, g_0$) : Plan
  return build-plan-aux($q_0, [g_0], \langle\emptyset, g_0, \emptyset, \emptyset\rangle, \emptyset$)

function build-plan-aux($q, c, pl, open$) : Plan
  if $(q, c) \in open$ then
    if is-good-loop($(q, c), open$) then return $pl$
    else return $\perp$
  if defined $pl.act[q, c]$ then return $pl$
  foreach $a \in \mathcal{A}(p)$ do
    foreach $(A, E) \in progr(q, c)$ do
      foreach $i \in assign\text{-}progr((A, E), \textsc{Exec}(q, a))$ do
        $pl' := pl$
        $pl'.C := pl'.C \cup \{c\}$
        $pl'.act[q, c] := a$
        $open' :=$ conc$((q, c'), open)$
        foreach $q' \in \textsc{Exec}(q, a)$ do
          $c' := order\text{-}goals(i[q'], c)$
          $pl'.ctxt[q, c, q'] := c'$
          $pl' := build\text{-}plan\text{-}aux(q', c', pl', open')$
          if $pl' = \perp$ then next $i$
        return $pl'$
  return $\perp$

```

Figure 4.6: The planning algorithm.

$(q, c) \in$ *open* then we are currently trying to build a plan for context c in state q. Whenever function *build-plan-aux* is called with a pair state-context already in *open*, then we have a loop of states in which the same sub-goal has to be enforced. In this case, function *is-good-loop*$((q, c), open)$ is called that checks whether the loop is valid or not. If the loop is good, plan *pl* is returned, otherwise function *build-plan-aux* fails.

Function *is-good-loop* computes the set *loop-goals* of the goals that are active during the whole loop: iteratively, it considers all the pairs (q', c') that appear in *open* up to the next occurrence of the current pair (q, c), and it intersects *loop-goals* with the set **setof**(c') of the goals in list c'. Then, function *is-good-loop* checks whether there is some strong until goal among the *loop-goals*. If this is a case, then the loop is bad: the semantics of CTL requires that all the strong until goals are eventually fulfilled, so these goals should not stay active during a whole loop. In fact, this is the difference between strong and weak until goals: executions where some weak until goal is continuously active and never fulfilled are acceptable, while the strong untils should be eventually fulfilled if they become active.

```
function is-good-loop($(q, c)$, open) : boolean
  loop-goals := setof($c$)
  while $(q, c) \neq$ head(open) do
    $(q', c')$ := head(open)
    loop-goals := loop-goals $\cap$ setof($c'$)
    open := tail(open)
  if $\exists g \in$ loop-goals : $g = \mathrm{A}(\_ \mathrm{U} \_)$ or $g = \mathrm{E}(\_ \mathrm{U} \_)$ then
    return false
  else
    return true
```

Figure 4.7: The loop checking routine.

If the pair (q, c) is not in *open* but it is in the plan *pl* (i.e., (q, c) is in the range of function *act* and hence condition "**defined** *pl.act*$[q, c]$" is true), then a plan for the pair has already been found in another branch of the search, and we return immediately with a success. If the pair state-context is neither in *open* nor in the plan, then the algorithm considers in turn all the executable actions a from state q, all the different possible progresses (A, E) returned by function *progr*, and all the possible assignments i of (A, E) to $\textsc{Exec}(q, a)$. Function *build-plan-aux* is called recursively for each destination state in $q' \in \textsc{Exec}(q, a)$. The new context is computed by function *order-goals*$(i[q'], c)$: this function returns a list of the goals in $i[q']$ that are ordered by their "age": namely those goals that are old (they appear in $i[q']$ and also in c) appear first, in the same order as in c, and those that are new (they appear in $i[q']$ but not in c) appear at the end of the list, in any order. Also, in the recursive call, argument *pl* is updated to take into account the fact that action a has been selected from state q in context g. Moreover, the new list of open problems is updated to **conc**$((q, c), open)$, namely the pair (q, c) is added in front of argument *open*.

Any recursive call of *build-plan-aux* updates the current plan *pl'*. If all these recursive calls are successful, then the final value of plan *pl'* is returned. If any of the recursive calls returns $\perp$, then the next combination of assign decomposition, progress component and action is tried. If all these combinations fail, then no plan is found and $\perp$ is returned.

As an example, call *build-plan*$(2, \text{lock-then-load})$ is successful and returns the plan in Figure 4.4 where c_0 and c_1 are goals $\mathrm{AF}\,(\mathsf{locked} \wedge \mathrm{AG}\,\mathrm{EF}\,(\mathsf{locked} \wedge \mathsf{loaded}))$ and $\mathrm{AG}\,\mathrm{EF}\,(\mathsf{locked} \wedge \mathsf{loaded})$, respectively.

The algorithm always terminates, and it is correct and complete:

Theorem 4.3.5 (Termination) *Function build-plan always terminates.*

Theorem 4.3.6 (Correctness) *Given a state q of a domain D and a goal g for D, if build-plan$(q, g) = \pi$ then $\pi, q \models g$, and if build-plan$(q, g) = \perp$ then there is no plan π*

such that $\pi, q \models g$.

In [PT01a], we show the scalability properties of the algorithm, and compare it to the behavior of SIMPLAN [KBSD97], one of the few planners capable of dealing with complex goals (expressed in the MTL metric temporal logics). While the possibility of expressing user-defined control strategies helps SIMPLAN in pruning major portions of the search, the symbolic nature of our implementations more than compensates when significant sources of nondeterminism enter into play.

4.4 Weak and Strong Planning under Partial Observability

In this section, we remove the simplifying assumption of total observability of the domain, and step to consider the general case partially observable domain, as defined in the framework.

Partial observability introduces a significant source of complexity, due to the fact that, at runtime, the status of the domain may not be uniquely determined from the observations. This means that the plan executor needs to act on the basis of its current *"belief"* on the domain state, considering every plausible situation. This reflects on the way plan synthesis needs to be performed: the elements of the search space will not be anymore associated domain states, but to sets of them.

For this reason, in this section, we limit to reachability goals, therefore lifting the kind of problems presented in Section 4.1 to the more general, partially observable setting.

To explain the concepts underlying our approach, we will adopt a simple robot navigation problem as a running example:

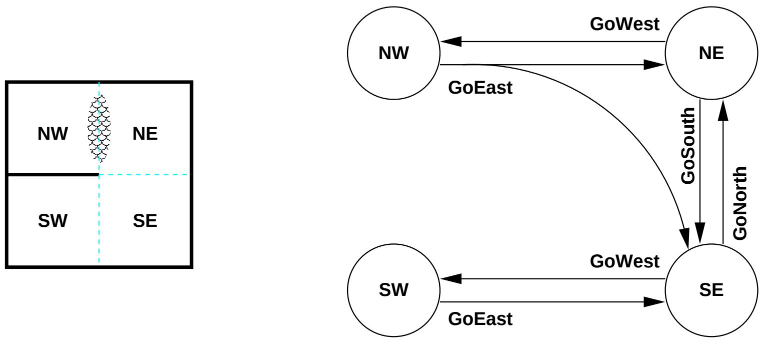


Figure 4.8: A simple robot navigation domain.

Example 4.4.1 *Fig. 4.8 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: $\mathcal{S} = \{NW, NE, SW, SE\}$. It can move in the four directions, corresponding to actions $\mathcal{A} = \{GoNorth, GoSouth, GoEast, GoWest\}$. An action is applicable only if there is no wall in the direction of motion, e.g. $\mathcal{R}(NW, GoSouth) = \emptyset$. All the actions are deterministic (e.g., $\mathcal{R}(SW, GoEast) = \{SE\}$), except for action GoEast when on NW, where the robot may slip and end up either in NE or SE, i.e. $\mathcal{R}(NW, GoEast) = \{NE, SE\}$.*

If we assume that the robot can only perceive the position of the walls neighboring its location, then it can not distinguish between the two positions NW and SW. This situation can be modeled with a set of observations $\mathcal{O} = \{nsw, ne, se\}$, and an observation function $\mathcal{X}(NW) = \mathcal{X}(SW) = \{nsw\}$, $\mathcal{X}(NE) = \{ne\}$, and $\mathcal{X}(SE) = \{se\}$.

We now first model the problem, starting as usual from modeling the structure of solution plans, and then provide a (symbolic) algorithm that solves it.

4.4.1 Problem Statement

To handle reachability goals under the general setting of partial observability, state-action tables are not sufficient anymore, since they implicitly assume that an observation is associated to a unique state. However, we may restrict our attention to loop-free plans, and therefore, take a simpler view than the general one, and consider tree-structured plans with conditional courses of action, where different behaviors can be determined by different run-time observations.

Definition 4.4.2 (Conditional Plan) *The set of conditional plans* Π *for a domain* $\mathcal{D} = \langle \Sigma, \mathcal{O}, \mathcal{X} \rangle$ *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* $\mathbf{if}\ o\ \mathbf{then}\ \pi_1\ \mathbf{else}\ \pi_2 \in \Pi$.

ϵ 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 π. The plan $\mathbf{if}\ o\ \mathbf{then}\ \pi_1\ \mathbf{else}\ \pi_2$ 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.

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

Definition 4.4.3 (Path of a plan) *An action-observation path* p *is a path of a plan* π *iff either*

- $p = \epsilon$*; or*
- $\pi = a \circ \pi'$*,* $p = a \circ p'$*, and* p' *is a path of* π'*; or*
- $\pi = \mathbf{if}\ o\ \mathbf{then}\ \pi_1\ \mathbf{else}\ \pi_2$*, and either (i)* $p = o \circ p_1$ *and* p_1 *is a path of* π_1*, or (ii)* $p = \overline{o} \circ p_2$ *and* p_2 *is a path of* π_2.

The symbol ϵ 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 $Paths(\pi)$ for the set of paths in π. We notice that, if $p \in Paths(\pi)$, then $p' \in Paths(\pi)$ for every p' prefix of p. Furthermore, $p \circ o \in Paths(\pi)$ iff $p \circ \overline{o} \in Paths(\pi)$. We say that $p \in Paths(\pi)$ is a *maximal path* of π if p is not a prefix of any other $p' \in Paths(\pi)$; we write $MaxPaths(\pi)$ for the set of maximal paths of π. If p is a path and P is a set of paths, we write $p \circ P$ for $\{p \circ p' : p' \in P\}$. 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 π identifies a (partial) course of action among the possible ones that are compatible with π, depending on the observations conveyed by the sensors.

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 4.4.4 (Applicability) *A plan π is applicable in state s iff either*

- *π is the empty plan ϵ; or*
- *π is $a \circ \pi_1$, a is applicable in s, and π_1 is applicable in every $s' \in \mathcal{R}(s, a)$; or*
- *π is* **if** *o* **then** *π_1* **else** *π_2, and:*
  - *if $s \in \mathcal{X}^-(o)$, π_1 is applicable in s, and*
  - *if $s \in \mathcal{X}^-(\overline{o})$, π_2 is applicable in s.*

Considering the case of noisy sensing, where $\mathcal{X}^-(o) \cap \mathcal{X}^-(\overline{o}) \neq \emptyset$, the definition deals with branching by requiring that both π_1 and π_2 are applicable to states associated both to o and to $\overline{o}$.

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 4.4.5 (Runs of a plan) *A run is a sequence $\sigma = (s_0, o_0) \circ a_1 \circ (s_1, o_1) \circ a_2 \circ \ldots \circ a_n \circ (s_n, o_n)$, where $s_i \in \mathcal{S}$, $o_i \in \mathcal{O}$, and $a_i \in \mathcal{A}$. A sequence σ is a run of a plan π from state s iff either*

- *$\pi = \epsilon$, and $\sigma = (s, o)$ with $o \in \mathcal{X}(s)$; or*
- *$\pi = a \circ \pi_1$ and $\sigma = (s, o) \circ a \circ \sigma_1$ with $o \in \mathcal{X}(s)$, and σ_1 is a run for π_1 for some $s_1 \in \mathcal{R}(s, a) \neq \emptyset$; or*
- *$\pi =$* **if** *o* **then** *π_1* **else** *π_2, $s \in \mathcal{X}^-(o)$, and σ is a run of π_1 starting from (s, o); or*
- *$\pi =$* **if** *o* **then** *π_1* **else** *π_2, $s \in \mathcal{X}^-(\overline{o})$, $o' \in \mathcal{X}(s)$, and σ is a run of π_2 starting from (s, o').*

Example 4.4.6 *Consider the following plan for the robot domain of Example 4.4.1:*

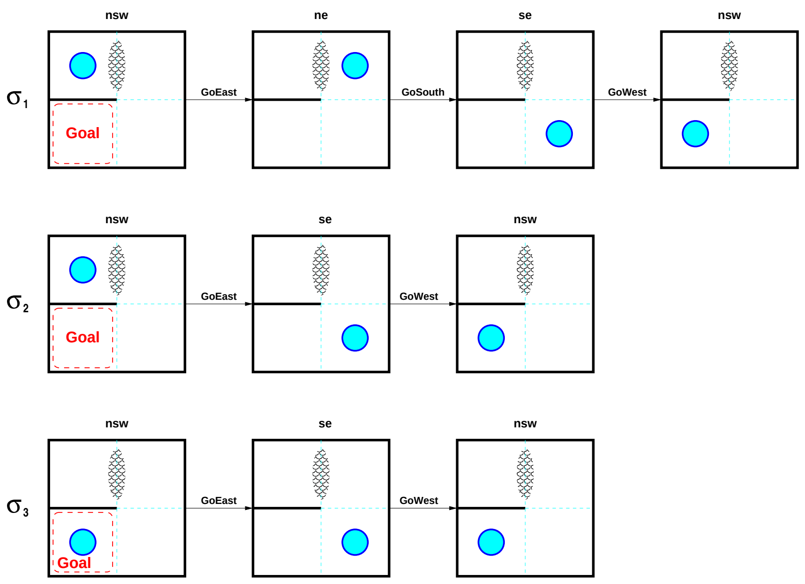


Figure 4.9: Runs of a conditional plan.

$$\mathsf{GoEast} \circ (\textbf{if } \mathsf{ne} \textbf{ then } \mathsf{GoSouth} \circ \mathsf{GoWest} \textbf{ else } \mathsf{GoWest})$$

*Action GoEast is executed first; then, if observation ne occurs, then GoSouth ∘ GoWest is executed; otherwise, the plan executes GoWest. The plan is applicable in the two states NW and SW: GoEast is applicable in both states, and (***if*** *ne* ***then*** *GoSouth ∘ GoWest* ***else*** *GoWest) is applicable in the resulting states {NE, SE}, since if the observation ne occurs, GoSouth ∘ GoWest is applicable in NE, and otherwise GoWest is applicable in SE. Fig. 4.9 depicts the set of runs of the plan in Example 4.4.1, starting from the states NW and 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* σ_1, σ_2 *and* σ_3, *where:*

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

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

Similar to the case of fully observable domains, we can identify *strong* and *weak* solutions. Strong solutions are 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. Weak solutions are are still guaranteed to be applicable in all the initial states, but only guarantee to end in a goal state for *some* possible run starting from an initial state.

Definition 4.4.7 (Strong Solution) *The plan π is a strong solution to the problem $\langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ iff*

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

Definition 4.4.8 (Weak Solution) *The plan π is a weak solution to the problem $\langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ iff*

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

4.4.2 Planning algorithm

To design our approach to solve the problem, we first observe that, due to partial observability, uncertainty must be explicitly dealt with at planning time. This leads us to recast the problems in terms of beliefs, i.e. sets of plausible states, and to think of a search structure whose nodes represent beliefs, so to design an approach based on forward generation of such a search structure. In the discussion, we will mostly focus on strong planning, which is in practice the most interesting (and difficult) of the two problems presented here. As we shall see, weak planning can be solved by a simple adaptation of the search for strong planning.

Our initial remark is that, in our setting, 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 4.4.6, 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* [BG00, FHMV95]. 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 4.4.9 (Applicability in a belief) *An action a is applicable in the belief state B, written 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 Def. 4.4.4.

Definition 4.4.10 (Execution in a belief) *The execution of a in B, written 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 $\overline{o}$ (i.e., an observation other than o) occurs, then our belief is given by $B \cap \mathcal{X}^-(\overline{o})$, 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 $B \cap \mathcal{X}^-(\overline{o}) = \emptyset$ (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 4.4.11 *Let us consider the robot domain in Example 4.4.1, and assume the robot can be either in NE or in SE. The corresponding belief state $B = \{\mathsf{NE}, \mathsf{SE}\}$ can be refined when either observation ne or se occurs. With observation ne, we have that $B \cap \mathcal{X}^-(\mathsf{ne}) = \mathsf{NE}$ and $B \cap \mathcal{X}^-(\overline{\mathsf{ne}}) = \mathsf{SE}$. When observation se occurs, we have that $B \cap \mathcal{X}^-(\mathsf{se}) = \mathsf{SE}$ and $B \cap \mathcal{X}^-(\overline{\mathsf{se}}) = \mathsf{NE}$. We notice that ne and se convey equivalent information, since they split the belief in the same way. The observation nsw is predetermined, since $B \cap \mathcal{X}^-(\mathsf{nsw}) = \emptyset$: that is, we know that nsw can not occur on B, and indeed, $B \cap \mathcal{X}^-(\overline{\mathsf{nsw}}) = B$.*

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

Definition 4.4.12 (Belief Execution along a Path) *Let π be a plan applicable in a belief state B, and let $p \in Paths(\pi)$. The belief execution along p, written $Exec(B, p)$, is defined as follows:*

- *if $p = \epsilon$, then $Exec(B, p) = B$;*
- *if $p = a \circ p'$, then $Exec(B, p) = Exec(Exec(B, a), p')$;*
- *if $p = o \circ p'$, then $Exec(B, p) = Exec(B \cap \mathcal{X}^-(o), p')$;*
- *if $p = \overline{o} \circ p'$, then $Exec(B, p) = Exec(B \cap \mathcal{X}^-(\overline{o}), p')$;*

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.

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 presented in [BCRT06].

Theorem 4.4.13 *Let $\langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a planning problem, and let π be applicable in $\mathcal{I}$. Then π is a strong solution to $\langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ iff $\bigcup$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 labeled directed hyper-graph, that is a graph structure whose nodes are connected by labeled directed hyper-arcs. Each hyper-arc associates a source node to a set of target nodes; in particular, the hyper-arcs 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 uniquely 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 4.4.14 (Belief Execution Tree) *Let π be a plan applicable in the belief B. Then, the Belief Execution Tree of π from B, written $BET(\pi, B)$, is the directed hyper-graph with nodes the set*

$$\{\langle p,\ Exec(B, p)\rangle : p \in Paths(\pi)\}$$

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

$$\langle \langle p,\ Exec(B, p)\rangle,\ a,\ \langle p',\ Exec(B, p')\rangle\rangle$$

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

$$\langle \langle p,\ Exec(B, p)\rangle,\ o,\ \langle p',\ Exec(B, p')\rangle\ \langle p'',\ Exec(B, p'')\rangle\rangle$$

Since each path in $Paths(\pi)$ is associated with exactly one node in $BET(\pi, B)$, in the following we also simply write p to indicate the node $\langle p,\ Exec(B, p)\rangle$. Given the properties of $Paths$ 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 ϵ, and the leaves are the nodes associated to maximal paths. 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 $p \circ \overline{o}$; 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'.

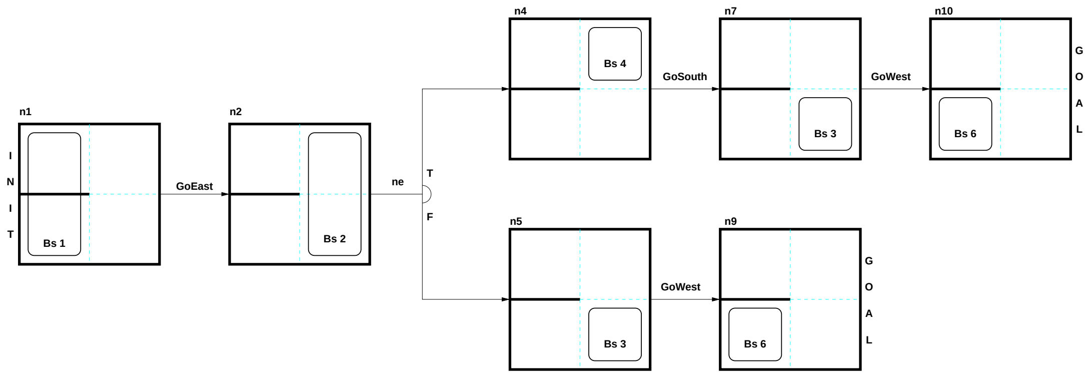


Figure 4.10: The Belief Execution Tree associated to plan π_2.

Example 4.4.15 *Fig. 4.10 depicts the belief execution tree associated to the strong solution of our running example. 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 4.4.16 (Subtree) *Given a belief execution tree T and a node n of T, associated with path p_n, its subtree from n, indicated with $SubTree(T, n)$, is a belief execution tree such that:*

- *for every node of T indexed with a path p s.t. $p_n \leq p$, there exists a node $n' \in SubTree(T, n)$. If $p = p_n$, then n' is indexed by path ϵ; 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 $SubTree(T,n)$, there exists one correspondent arc in $SubTree(T,n)$, 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 4.4.17 (Plan associated to belief execution tree) *Given a belief execution tree $T = \langle n_0, N, E\rangle$, its associated plan $PlanOf(T)$ is defined as follows:*

- *if $E = \emptyset$, then $PlanOf(T) = \epsilon$*
- *if $\langle n_0, a, n\rangle \in E$, then $PlanOf(T) = a \circ PlanOf(SubTree(T,n))$*
- *if $\langle n_0, o, n_1, n_2\rangle \in E$, then $PlanOf(T) = \textbf{if } o \textbf{ then } \pi_1 \textbf{ else } \pi_2$, where $\pi_1 = PlanOf(SubTree(T,n_1))$ and $\pi_2 = PlanOf(SubTree(T,n_2))$.*

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

At this point, we can tackle strong planning under partial observability, based on the idea of progressing from the initial belief state $\mathcal{I}$ towards the goal $\mathcal{G}$. 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* ([MM73]), we can not assume that the search space is acyclic, and differently from more recent extensions such as LAO* ([HZ01]), we do not

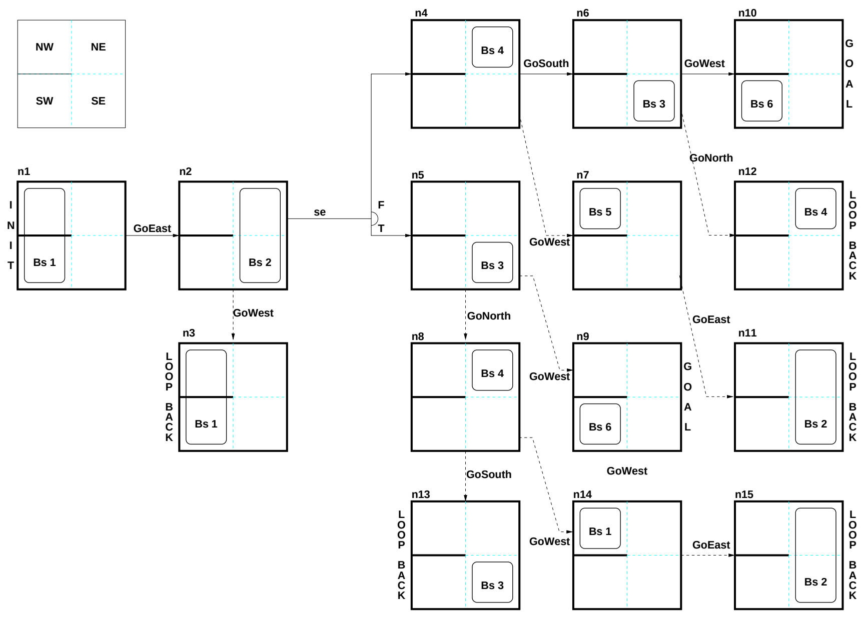


Figure 4.11: The search tree for the example.

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 4.4.18 *Fig. 4.11 represents the search tree for the domain in Example 4.4.1. For instance, node $n2$ is expanded into node $n3$ by the only applicable action on it,* GoWest*, and into nodes $n4$ and $n5$ by the observation* se*. The figure does not report predetermined observations, such as* nsw *for node $n1$. 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 $n2$, since it has the same effect of observation* se*). Notice that even in this simple case, loops are possible; for instance, node $n11$ loops back to node $n2$. Also notice that several nodes associated to the same belief appear in the search tree, e.g. $n4$, $n8$ and $n12$ 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, $n12$ is associated to a looping path, while $n4$ and $n8$ are not.*

The planning algorithm is described in Fig. 4.12. 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*

```
 1 procedure StrongPO(I,G)
 2   $ST$ := MkInitialTree(I);
 3   if $(I \subseteq G)$ then
 4     TagNode(Root($ST$), $Success$);
 5   fi
 6   while (¬isSucc(Root($ST$)) ∧ ¬isFail(Root($ST$))) do
 7     node := ExtractNodeFromFrontier($ST$);
 8     ExtendTree(node, $ST$);
 9     if (SonsYieldSuccess(node, $ST$)) then
10       TagNode(node, $Success$);
11       PropagateSuccessOnTree(node, $ST$);
12     else if (SonsYieldFailure(node, $ST$)) then
13       TagNode(node, $Failure$);
14       PropagateFailureOnTree(node, $ST$);
15     fi
16   done
17   if isSucc(Root($ST$)) then
18     return BuildPlan(Root($ST$), $ST$);
19   else
20     return $Failure$;
21   fi
22 end
```

Figure 4.12: The basic planning algorithm.

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 $\text{BEL}(n) \subseteq \mathcal{G}$; 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 SONSYIELDSUCCESS 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 SONSYIELDFAILURE 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 4.4.19 *Fig. 4.13 describes a possible behavior of the algorithm for the example of Fig. 4.10, 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. 4.13 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.*

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.

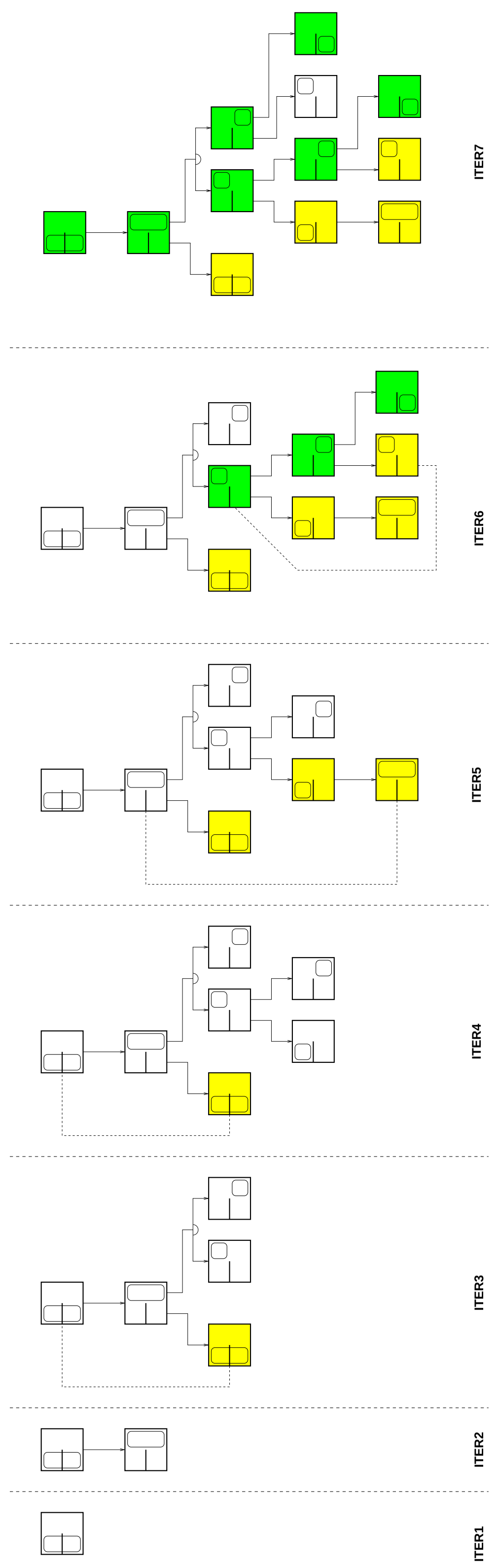


Figure 4.13: The behavior of the algorithm for the example.

EXTENDTREE The EXTENDTREE primitive, presented in Fig. 4.15, 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. $\text{BEL}(n) \subseteq \mathcal{G}$), a failure (i.e. some node in the set of ancestors of n, $\text{ANCESTORS}(n, ST)$, 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 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 represent 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.

SONSYIELDSUCCESS, SONSYIELDFAILURE The SONSYIELDSUCCESS primitive, see Fig. 4.16, 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 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, $\text{SONSYIELDFAILURE}(n)$ and $\text{SONSYIELDSUCCESS}(n)$ can not hold together. This comes from the fact that, for

```
procedure TagNewNode$(n, ST)$
  if Bel$(n') \subseteq \mathcal{G}$ then
    TagNode$(n', Success)$
  else if $\exists n_a \in$ ancestors$(n', ST)$ : Bel$(n')$ = Bel$(n_a)$ then
    TagNode$(n', Failure)$
  else
    TagNode$(n', Undetermined)$
  fi
end
```

Figure 4.14: The node marking subroutine.

```
procedure ExtendTree$(n, ST)$
  forall $a \in \mathcal{A}$
    if $applicable($Bel$(n), a)$ then
      $n' := \langle$Exec(Bel$(n), a)$, Path$(n) \circ a\rangle$
      $Nodes(ST) := Nodes(ST) \cup \{n'\}$
      $Arcs(ST) := Arcs(ST) \cup \langle n, a, n'\rangle$
      TagNewNode$(n', ST)$
    fi
  endfor
  forall $o \in \mathcal{O}$
    $B_T :=$ Bel$(n) \cap \mathcal{X}^-(o)$
    $B_F :=$ Bel$(n) \cap \mathcal{X}^-(\overline{o})$
    if $(B_T \neq \emptyset)$and $(B_F \neq \emptyset)$
      $n'_T := \langle B_T,$ Path$(n) \circ o\rangle$
      $n'_F := \langle B_F,$ Path$(n) \circ \overline{o}\rangle$
      $Nodes(ST) := Nodes(ST) \cup \{n'_T, n'_F\}$
      $Arcs(ST) := Arcs(ST) \cup \langle n, o, n'_T, n'_F\rangle$
      TagNewNode$(n'_T, ST)$
      TagNewNode$(n'_F, ST)$
    endfor
  end
```

Figure 4.15: The node expansion primitive.

```
function SonsYieldSuccess$(n, ST)$
    $res := false$
    forall $\langle n, a, n' \rangle \in Arcs(ST)$
        $res := res \lor$ isSucc$(n')$
    endfor
    forall $\langle n, o, n_T, n_F \rangle \in Arcs(ST)$
        $res := res \lor$ (isSucc$(n_T) \land$ isSucc$(n_F)$)
    endfor
    return $res$
```

```
function SonsYieldFailure$(n, ST)$
    $res := true$
    forall $\langle n, a, n' \rangle \in Arcs(ST)$
        $res := res \land$ isFail$(n')$
    endfor
    forall $\langle n, o, n_T, n_F \rangle \in Arcs(ST)$
        $res := res \land$ (isFail$(n_T) \lor$ isFail$(n_F)$)
    endfor
    return $res$
```

Figure 4.16: The procedures to detect failure or success of an expanded node.

SonsYieldSuccess(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 SonsYieldFailure(n) can not hold.

PropagateSuccessOnTree, PropagateFailureOnTree The success and failure propagation routines, see Fig. 4.17, are defined recursively, and simply traverse the tree bottom up, from the current node. During this propagation, PropagateSuccessOnTree tags a node with $Success$ if its success can be established by SonsYieldSuccess, while PropagateFailureOnTree tags a node with $Failure$ if its failure can be established by SonsYieldFailure.

BuildPlan The BuildPlan primitive is reported in Fig. 4.18, 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.

As usual, we can prove that the search algorithm is correct and complete, and always terminates (proofs can be found in the Appendix):

Theorem 4.4.20 (Termination) *Given a planning problem, the execution of the planning algorithm in Fig. 4.12 terminates.*

Theorem 4.4.21 (Correctness) *When the algorithm returns a plan π, then π is a strong solution for the problem.*

```
procedure PropagateSuccessOnTree($(n, ST)$)
  if $(n \neq \textsc{Root}(ST))$ then
    $n_{fat} := \textsc{Father}(n)$
    if $(\textsc{SonsYieldSuccess}(n_{fat}, ST))$ then
      $\textsc{TagNode}(n_{fat}, Success)$
      $\textsc{PropagateSuccessOnTree}(n_{fat}, ST)$
    fi
  fi
end
```

```
procedure PropagateFailureOnTree($(n, ST)$)
  if $(n \neq \textsc{Root}(ST))$ then
    $n_{fat} := \textsc{Father}(n)$
    if $\textsc{SonsYieldFailure}(n_{fat}, ST)$ then
      $\textsc{TagNode}(n_{fat}, Failure)$
      $\textsc{PropagateFailureOnTree}(n_{fat}, ST)$
    fi
  fi
end
```

Figure 4.17: The procedures to propagate success/failure of a node.

```
function BuildPlan($(n, ST)$)
  if $\textsc{Bel}(n) \subseteq \mathcal{G}$ then
    return $\epsilon$
  else if $\exists\langle n, a, n'\rangle \in Arcs(ST) : \textsc{isSucc}(n')$ then
    pick $\langle n, a, n'\rangle \in Arcs(ST) : \textsc{isSucc}(n')$
    $\pi' := \textsc{BuildPlan}(n', ST)$
    return $a \circ \pi'$
  else
    pick $\langle n, o, n_T, n_F\rangle \in Arcs(ST) : \textsc{isSucc}(n_T) \wedge \textsc{isSucc}(n_F)$
    $\pi_T := \textsc{BuildPlan}(n_T, ST)$
    $\pi_F := \textsc{BuildPlan}(n_F, ST)$
    return if $o$ then $\pi_T$ else $\pi_F$)
```

Figure 4.18: The plan building procedure.

Theorem 4.4.22 (Completeness) *If a strong solution exists, then the algorithm returns a strong solution.*

In [BCRT06], it is possible to find a thorough experimental evaluation of this approach, and a comparison against alternate approaches based on either symbolic backward search (JUSSIPOP ([Rin04]), or POMDP representation (GPT, [BG00]). The tests show that this approach is state of the art, proviso that informed search strategies are plugged in the EXTRACTNODEFROMFRONTIER primitive. In particular, we designed a strategy based on dynamically identifying a subset of the goal as the actual "target", and to evaluate distances to the target based on the projection of the beliefs over the domain fluents. This proved far more effective than other strategies based on merely evaluating distances between the current belief and the whole set of goal states.

Finally, we remark that while the discussion focused on strong planning, the approach can be readily adapted to solve weak planning. In particular, the recasting of the problem in terms of beliefs would only require that some final belief is not disjoint with the goal, and correspondingly, the following adaptations to the algorithm are in order:

1. A node must be tagged success if its associated belief has a non-empty intersection with the goal, relaxing the entailment requirement. This impacts on the condition at line 1 of TAGNEWNODE.
2. A node is successful if either some or-successor of it is successful, or if *some* and-successor of it is successful. That is, the requirement that all observations need to lead to the goal is also relaxed. This implies a simple change in the way success is computed at line 7 of SONSYIELDSUCCESS.

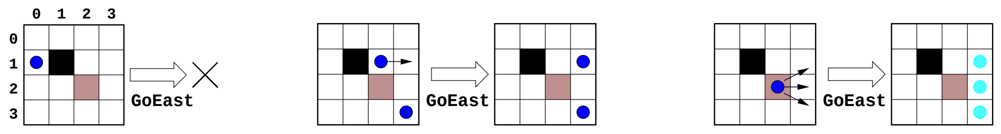


Figure 4.19: A simple nondeterministic robot navigation domain.

4.5 Conformant planning

We now consider an "extreme" case of domain with limited sensing, namely the case where no sensing at all is possible. Planning under these premises is known as *conformant planning*, and is interesting because of several features. Conformant solutions are simply plans made up of sequences of actions, just as in classical planning; however, to search for them, we need to consider the space of beliefs, like we did for partially observable domains. Thus conformant planning is the easier setting possible, when limited sensing is available, to study search in belief space, and in particular, to consider belief-level heuristics. Also, conformant planning is directly linked to some applicative problems, namely reset sequencing - the problem of "resetting" a machine with a fixed sequence of actions, even in the total absence of responses. As usual, we will state the problem based on the structure of its solutions, and then discuss solutions to it.

4.5.1 Problem statement

The reference domain for conformant planning is a particular case of the general model presented in the framework, where the lack of sensing is modeled by having just one observation, which is associated to every state and therefore brings no information at all. The following example will be used as an explanatory reference.

Example 4.5.1 *Consider a simple navigation domain depicted in Figure 4.19. A robot can move in the four directions in a 4x4 square. The positions in the square can be represented by means of two state variables* x *and* y*, both ranging from 0 to 3. Thus,* $\mathcal{X} = \{x, y\}$ *and* $\mathcal{V}(x) = \mathcal{V}(y) = \{0, 1, 2, 3\}$*. The state corresponding to the robot being in the upper-left corner is the tuple* $(\langle \mathtt{x},\ \mathtt{0}\rangle, \langle \mathtt{y},\ \mathtt{0}\rangle)$*. In the following we simply refer to* $(\langle \mathtt{x},\ \mathtt{v_x}\rangle, \langle \mathtt{y},\ \mathtt{v_y}\rangle)$ *with* $(\mathtt{v_x}, \mathtt{v_y})$*. Thus, for this domain the set of states is* $\{(\mathtt{v_x}, \mathtt{v_y}) : \mathtt{v_x}, \mathtt{v_y} \in \{0, 1, 2, 3\}\}$*. The set of actions* $\mathcal{A}$ *that model the movement directions of the robot is* $\{\mathtt{GoWest}, \mathtt{GoEast}, \mathtt{GoNorth}, \mathtt{GoSouth}\}$*. The black location in* $(1, 1)$ *is a hole that, if entered, will cause the robot to crash. This means, for instance, that the* `GoEast` *action is not applicable in* $(0, 1)$*. When moving towards a wall, the robot does not move. For instance, if the robot performs a* `GoEast` *action in location* $(3, 3)$*, it will remain in* $(0, 0)$*. The location in* $(2, 2)$ *is a slippery spot, that will make the robot move unpredictably sideways when performing an action in it. This introduces nondeterministic action effects. For instance, the effect of performing a* `GoEast` *action starting from* *state* $(2,2)$ *may results in any of the states in* $\{(3,1),(3,2),(3,3)\}$.

Since no observation at all is possible, and therefore a plan executor would have no criteria for branching or looping, it only makes sense to consider plans consisting of sequences of actions. Indeed, such structure, and the associated notion of run and applicability, can be perceived as a simplification of the correspondent notions for partially observable domains.

Definition 4.5.2 (Plan, Plan length) *A plan* π *for a planning domain* $\mathcal{D}$ *is an element of* $\mathcal{A}^*$*, i.e. a finite, possibly empty sequence* $a_1, \ldots, a_n$ *such that for all* $a_i \in \mathcal{A}$ *with* $1 \leq i \leq n$. n *is the length of the plan.*

We use ϵ to denote the 0-length plan, π and ρ to denote plans, $\pi \circ \rho$ to denote plan concatenation, and $\|\pi\|$ to denote the length of π.

Definition 4.5.3 (Runs of a plan) *Let* $\pi = a_1, \ldots, a_n$ *be a plan of length* n. *The sequence* $\sigma = s_0, s_1, \ldots, s_n$ *is a run for* π *from* s_0 *iff, for* $0 \leq i < n$, $(s_i, a_{i+1}, s_{i+1}) \in \mathcal{R}$. s_n *is the final state of the run. The set of runs of* π *from* $B \subset \mathcal{S}$ *is the set of all runs of* π *from any* $s \in B$.

We notice that, whenever S contains more than one state, the set of runs of π from S also contains multiple runs. Furthermore, a plan can be associated with many runs, even if starting from the same state, due to the uncertain effects of actions. Our definition of plan applicability guarantees that no action may be attempted in a state where it is not applicable.

Definition 4.5.4 (Plan applicability) *The empty plan* ϵ *is applicable in every state* $s \in \mathcal{S}$. *The plan* $\alpha \circ \pi$ *is applicable in* $s \in \mathcal{S}$ *iff* α *is applicable in* s *and* π *is applicable in all* s' *such that* $(s, \alpha, s') \in \mathcal{R}$. *Let* S *be a nonempty set of states. A plan* π *is applicable in* S *iff it is applicable in all* $s \in S$.

Example 4.5.5 *The plan* π_1 = `GoNorth; GoEast; GoEast; GoEast; GoSouth` *is applicable in state* $(0,1)$. *The only associated run is:*

$$\sigma_0 = (0,1),(0,0),(1,0),(2,0),(3,0),(3,1)$$

Plan π_1 *is not applicable in* $(0,2)$, *since the second action would be attempted in* $(0,1)$, *which violates the applicability condition. Plan* π_2 = `GoEast; GoEast; GoEast; GoSouth` *is applicable in state* $(0,2)$ *and its three runs are:*

$$\begin{aligned}\sigma_1 &= (0,2),(1,2),(2,2),(3,1),(3,2)\\ \sigma_2 &= (0,2),(1,2),(2,2),(3,2),(3,3)\\ \sigma_3 &= (0,2),(1,2),(2,2),(3,3),(3,3)\end{aligned}$$

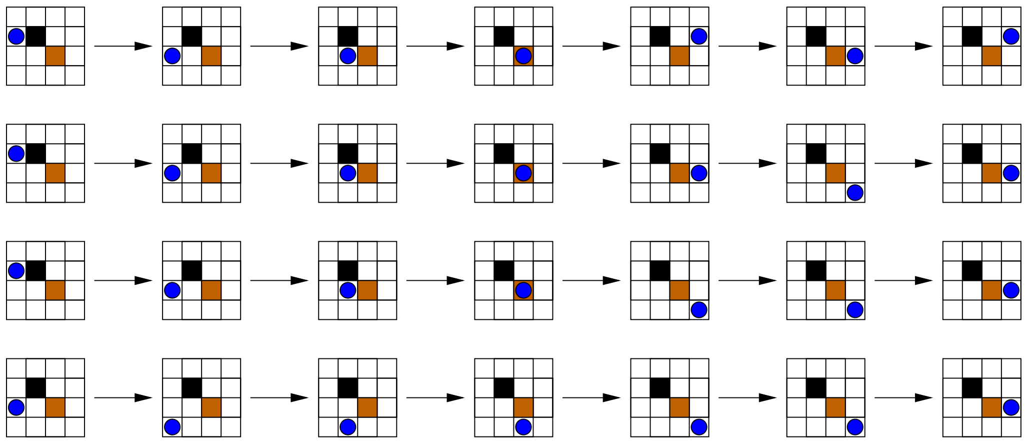


Figure 4.20: The runs associated with the plan `GoSouth`; `GoEast`; `GoEast`; `GoEast`; `GoSouth`; `GoNorth`.

Plan $\pi_3 =$ *`GoSouth`; `GoEast`; `GoEast` `GoEast`; `GoSouth`; `GoNorth` is applicable in* $\{(0,1),(0,2)\}$*, and the corresponding runs, depicted in Figure 4.20, are:*

$$\begin{aligned}
\sigma_4 &= (0,1),(0,2),(1,2),(2,2),(3,1),(3,2),(3,1)\\
\sigma_5 &= (0,1),(0,2),(1,2),(2,2),(3,2),(3,3),(3,2)\\
\sigma_6 &= (0,1),(0,2),(1,2),(2,2),(3,3),(3,3),(3,2)\\
\sigma_7 &= (0,2),(0,3),(1,3),(2,3),(3,3),(3,3),(3,2)
\end{aligned}$$

Intuitively, a conformant planning problem for a given domain is characterized by a set of possible initial states, and a set of goal states. A plan is a solution to a conformant planning problem if it is applicable in all the initial states, and all the associated runs end in a goal state.

Definition 4.5.6 (Conformant Planning Problem, Solution) *A conformant planning problem is a triple* $\langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ *where* $\mathcal{D}$ *is a planning domain,* $\mathcal{I}$ *and* $\mathcal{G}$ *are non empty sets of states, called the set of initial states and the set of goal states, respectively. A plan* π *is a solution to a conformant planning problem* $\langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ *iff*

- *it is applicable in* $\mathcal{I}$*, and*
- *every run of* π *from* $\mathcal{I}$ *has its final state in* $\mathcal{G}$*.*

Example 4.5.7 *In the robot navigation domain of Figure 4.19, let us consider the conformant planning problem where the initial set of states is* $\{(0,1),(0,2)\}$ *and the set of goal states is* $\{(3,1),(3,2)\}$*. A possible solution to this problem is the plan*

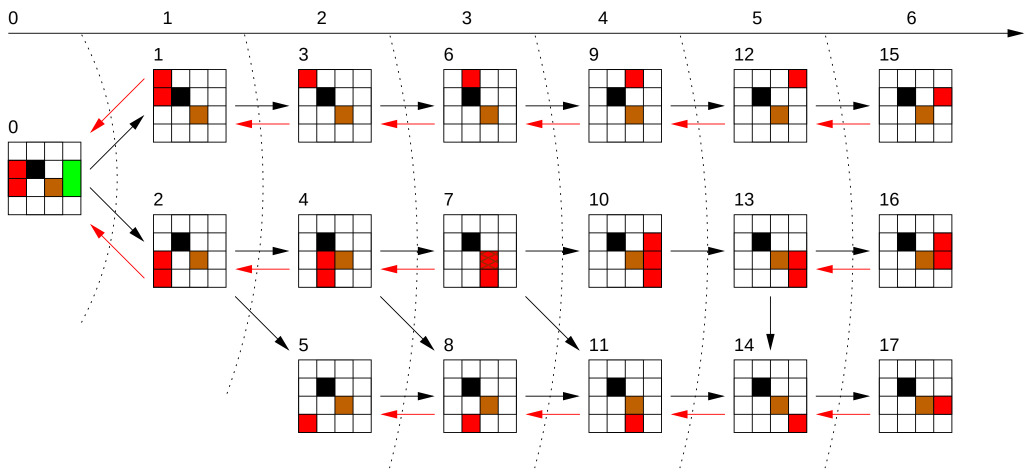


Figure 4.21: A fragment of the forward search space for the robot navigation domain.

$\pi_3 \doteq$ `GoSouth`; `GoEast`; `GoEastGoEast`; `GoSouth`; `GoNorth` described in Example 4.5.5. The plan is a conformant solution: for all the possible associated runs (depicted in Figure 4.20) the plan never violates the applicability conditions of the attempted actions, and the final states are all in the goal. Notice that the plan is not particularly "smart", if the runs are considered individually. For instance, the first two runs reach the goal after four actions. However, other actions are needed to make sure that the problem is solved for all runs.

4.5.2 Planning Algorithms

We now address the problem of searching the belief space in order to solve a conformant planning problem. In essence, in the conformant setting, the belief space reduces to a directed, cyclic *or*-graph whose arcs represent actions, and whose nodes are beliefs. Such a search space can be traversed in several ways to identify a solution. In the following, we consider a forward-directed search, and a backward-directed one, providing algorithm and discussing their properties.

Forward Search in the Belief Space

Example 4.5.8 *Figure 4.21 outlines (a subset of) the search space for the problem described above, constructed forward, starting from the initial set of states toward the goal. An expansion step consists in considering the belief states resulting from the execution of all the applicable actions. Notice that different plans can result in the same belief state, and that cycles are possible. Furthermore, even for this simple example, three different solutions are possible (since the three leftmost belief states are all goal states). Their*

```
1  **function** HEURCONFORMANTFWD($\mathcal{I}, \mathcal{G}$)
2    *Open* := $\{\langle \mathcal{I},\ \epsilon \rangle\}$;
3    *Closed* := $\emptyset$;
4    *Solved* := $False$;
5    **while** (*Open* $\neq \emptyset \wedge \neg$*Solved*) **do**
6      $\langle Bs,\ \pi \rangle$ := EXTRACTBEST(*Open*);
7      INSERT($\langle Bs,\ \pi \rangle$, *Closed*);
8      **if** $Bs \subseteq \mathcal{G}$ **then**
9        *Solved* := $True$; *Solution* := $\pi$;
10     **else**
11       *BsExp* := FWDEXPANDBS($Bs$);
12       *BsPList* := PRUNEBSEXPANSION(*BsExp*, *Closed*);
13       **for** $\langle Bs_i,\ \alpha_i \rangle$ **in** *BsPList* **do**
14         INSERT($\langle Bs_i,\ \pi \circ \alpha_i \rangle$, *Open*)
15       **endfor**
16     **fi**
17   **done**
18   **if** *Solved* **then**
19     **return** *Solution*;
20   **else**
21     **return** *Fail*;
22   **fi**
23 **end**
```

Figure 4.22: The forward conformant planning algorithm.

characteristics are quite different: for instance, after the plan associated with the upper row (`GoNorth` twice; `GoEast` three times; `GoSouth`) the uncertainty in the robot location is eliminated.

Finally, notice the potential complexity of the problem. Since a belief state is a subset of $\mathcal{S}$, conformant planning amounts to searching all possible paths in the belief space. In the simple example depicted in Figure 4.19, 15 states (assuming the hole is not an admissible condition) induce 2^{15} belief states (although not all of them are reachable).

The forward progression algorithm depicted in Figure 4.22 searches the belief space, proceeding forwards from the set of initial states $\mathcal{I}$ towards the goal $\mathcal{G}$. The algorithm can be seen as a standard best-first algorithm, where search nodes are (uniquely indexed by) belief states. *Open* contains a list of open nodes to be expanded, and *Closed* contains a list of closed nodes that have already been expanded. After the initialization phase, *Open* contains (the node indexed by) $\mathcal{I}$, while *Closed* is empty. The algorithm then enters a loop, where it extracts a node from the open list, stores it into the closed list, and checks if it is a success node (line 8) (i.e. it a subset of $\mathcal{G}$); if so, a solution has been found and the iteration is exited. Otherwise, the successor nodes are generated, and the ones that have already been expanded are pruned. The remaining nodes are stored in *Open*, and the iteration restarts. Each belief state Bs is associated with a plan π, that is applicable in $\mathcal{I}$, and that results exactly in Bs, i.e. *Exec*$(\pi, \mathcal{I}) = Bs$.

The algorithm loops (lines 5–17) until either a solution has been found (*Solved* = $True$) or all the search space has been exhausted (*Open* $= \emptyset$). A belief state Bs is extracted from the open pool (line 6), and it is inserted in closed pool (line 7). The belief states Bs is expanded (line 11) by means of the FWDEXPANDBS primitive. PRUNEB-SEXPANSION (line 12) removes from the result of the expansion of Bs all the belief state that are in the *Closed*, and returns the pruned list of belief states. If *Open* becomes empty and no solution has been found, the algorithm returns with *Fail*to indicate that the planning problem admits no conformant solution. The expansion primitive FWDEXPANDBS takes as input a belief state Bs, and builds a set of pairs $\langle Bs_i, \alpha_i\rangle$ such that α_i is executable in Bs and the execution of α_i in Bs is contained in Bs_i. Notice that α_i is a conformant solution for the planning problem of reaching Bs_i from any non-empty subset of Bs.

$$\textsc{FwdExpandBs}(Bs) \doteq \{\langle Bs_i, \alpha_i\rangle : Bs_i = \textit{Exec}(\alpha_i, Bs) \neq \bot\}$$

Function PRUNEBSEXPANSION takes as input a result of an expansion of a belief state and *Closed*, and returns the subset of the expansion containing the pairs where each belief state has not been expanded. The PRUNEBSEXPANSION function can be defined as:

$$\textsc{PruneBsExpansion}(BsP, \textit{Closed}) \doteq \{\langle Bs_i, \alpha_i\rangle : \langle Bs_i, \alpha_i\rangle \in BsP \text{ and } \langle Bs_i, \pi\rangle \in \textit{Closed} \text{ for no plan } \pi\}$$

When an annotated belief state $\langle Bs, \pi\rangle$ is inserted in *Open*, INSERT checks if another annotated belief state $\langle Bs, \pi'\rangle$ exists; the length of π and π' are compared, and only the pair with the shortest plan is retained.

The algorithm described above can implement several search strategies, e.g. depth-first or breadth-first, depending on the implementation of the functions EXTRACTBEST (line 6) and INSERT (line 14). In the following sections, we will interpret the search algorithm as a standard A* algorithm, implementing a best-first heuristic search.

Example 4.5.9 *A portion of the search space traversed by the algorithm while attempting to solve the problem of reaching* $\{(0,1),(0,2)\}$ *from the* $\{(3,1),(3,2)\}$ *is depicted in Figure 4.21. Numbers on the upper left corner name belief states. Open is initialized with belief state 0 annotated with the empty plan. When expanded, it results in belief state 1 and 2 (`GoNorth` and `GoSouth` actions, respectively). 0 is also re-generated (action `GoEast`), but it is pruned away since already closed. Action `GoWest` is not applicable. Since neither 1 nor 2 are subsets of* $\mathcal{G}$*, they are added to Open. Say 1 is extracted; from its expansion we re-obtain belief states 0 (action `GoSouth`), pruned since closed, and 3 (action `GoEast`), new. Belief state 3 is not a subset of the goal, and it is thus added* *to Open for further expansion. The search proceeds by considering belief state 2 from Open. The expansion of belief state 2 leads to belief states 0, 4, 5. Belief state 0 has been visited before and is discarded, while belief states 4 and 5 that are not subset of the goal are added to Open.*

The algorithm enjoys the following properties. First, it always terminates. Second, it returns a solution if the problem is solvable, otherwise it returns failure.

Theorem 4.5.10 *Let $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a planning problem.* HEURCONFORMANTFWD$(\mathcal{I}, \mathcal{G})$ *always terminates.*

Termination is proved by noticing that at each step, at least one belief state is extracted from *Open* and inserted into *Closed*, while a possibly empty set of new ⟨belief state, plan⟩ pairs, such that the belief state has not yet been visited, is inserted in *Open*. Given that the number of belief states is finite and we do not add to *Open* already visited belief states, the algorithm is guaranteed to terminate. The formal proof is presented in the Appendix.

Theorem 4.5.11 *Let $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a planning problem and let π be the value returned by the function* HEURCONFORMANTFWD$(\mathcal{I}, \mathcal{G})$.

- *If $\pi \neq$ Fail, then π is a conformant solution for the planning problem P.*
- *If $\pi =$ Fail, instead, then there is no conformant solution for the planning problem P.*

To prove this theorem we notice that the invariant preserved by the algorithm is that each belief state plan pair $\langle Bs, \pi \rangle$ in the *Open* is such that π is applicable in $\mathcal{I}$ and *Exec*$(\pi, \mathcal{I}) = Bs$. In other words, the plan is applicable in the initial belief state, and it results in the belief state Bs. Again, the formal proof is presented in the Appendix.

Backward Search in the Belief Space

The dual for the algorithm described in Figure 4.22 where the belief space is traversed backward, from the goal towards the initial belief state, is depicted in Figure 4.23. The algorithms share the very same control structure, but different computations are performed. In particular, *Open* is initialized with the goal belief state annotated with the empty plan (line 2); the success test (at line 8) checks if the extracted belief state Bs contains $\mathcal{I}$ (rather than checking for containment in $\mathcal{G}$); the expansion of Bs (at line 11) is carried out by the BWDEXPANDBS primitive; finally, the annotated belief state inserted at line 14 is associated with a plan where the action α_i is prepended to π (rather than being appended as in the forward case).

```
function HeurConformantBwd($\mathcal{I}, \mathcal{G}$)
   Open := $\{\langle \mathcal{G}, \epsilon \rangle\}$;
   Closed := $\emptyset$;
   Solved := $False$;
   while (Open $\neq \emptyset \wedge \neg$Solved) do
      $\langle Bs, \pi \rangle$ := ExtractBest(Open);
      Insert($\langle Bs, \pi \rangle$, Closed);
      if $\mathcal{I} \subseteq Bs$ then
         Solved := $True$; Solution := $\pi$;
      else
         BsExp := BwdExpandBs($Bs$);
         BsPList := PruneBsExpansion(BsExp, Closed);
         for $\langle Bs_i, \alpha_i \rangle$ in BsPList do
            Insert($\langle Bs_i, \alpha_i \circ \pi \rangle$, Open);
         endfor
      fi
   if Solved then
      return Solution;
   else
      return Fail;
   fi
end
```

Figure 4.23: The backward conformant planning algorithm.

BwdExpandBs, similarly to FwdExpandBs, takes as input a belief state Bs, and builds a set whose elements $\langle Bs_i, \alpha_i \rangle$ are such that α_i is executable in Bs_i and the execution of α_i in Bs_i is guaranteed to end up in states contained in Bs. Thus, α_i is a conformant solution for the planning problem of reaching Bs from any non-empty subset of Bs_i.

$$\text{BwdExpandBs}(Bs) \doteq \{\langle Bs_i, \alpha_i \rangle : Bs_i = \text{StrongPreImage}[\alpha_i](Bs) \neq \emptyset\}$$

where

$$\text{StrongPreImage}[\alpha](Bs)\{s : \text{Appl}(\alpha, s) \text{ and, for all } s' \in Bs, \mathcal{R}(s, \alpha, s')\}$$

A fragment of the search space traversed by the algorithm of Figure 4.23 is depicted in Figure 4.24. Compare the expansion primitives for the backward and forward search algorithms: while proceeding backwards, the expansion step guarantees the applicability of the extended plans, while in the forward case we first need to compute the applicable actions, and then to project their effects. In the forward case, a plan is associated with the "minimal" belief state resulting from its execution in the initial condition; in the backward

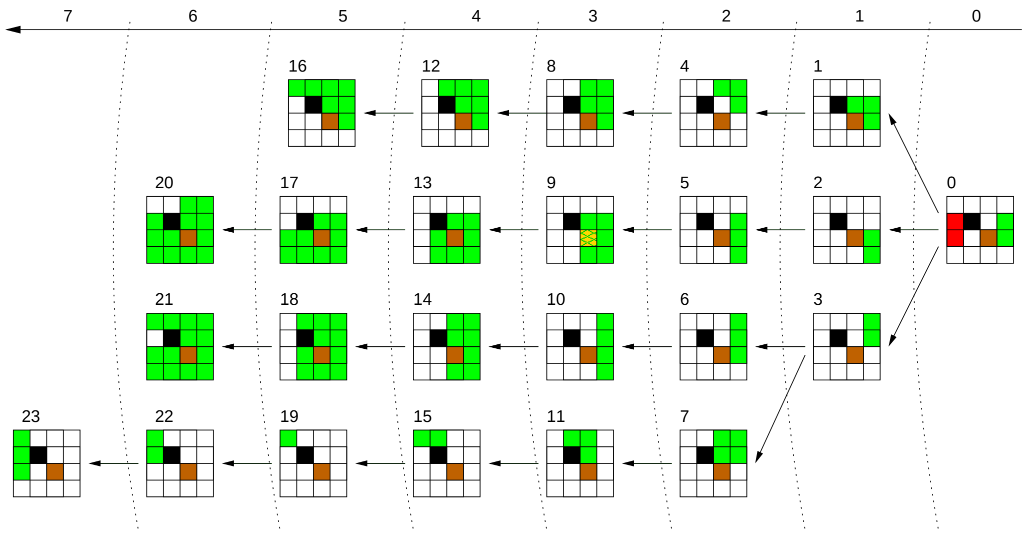


Figure 4.24: A fragment of the backward search space for the robot navigation domain.

case, a plan is associated with the "maximal" set for which it guarantees reaching the goal. These differences may result in very different search spaces (compare also Figure 4.21 and Figure 4.24).

This algorithm enjoys the same properties of the one of Figure 4.22. Termination can be proved similarly to the forward case. Correctness is proved by considering that the algorithm preserves the following invariants: each belief state plan pair $\langle Bs, \pi\rangle$ in *Open* is such that π is applicable in Bs, and *Exec*$(\pi, Bs) \subseteq \mathcal{G}$. In other words, the plan is applicable in the belief state it is associated with, and it ends up in the goal.

Belief-level Heuristics

For both algorithms for conformant planning, a critical factor is the ability to drive the search in the "right" direction, i.e. to limit the number of expanded belief states. In [CRB03], two "symbolic reachability" heuristics functions are proposed, MAXSDIST and MAXWDIST, based on the idea of computing the distance between two beliefs based on a relaxation of the problem where full observability is assumed.

In particular, $MaxSDist$ is computed by a simplified version of STRONGPLAN, the algorithm for strong planning in nondeterministic domains under full observability, and relies on the STRONGPREIMAGE primitive, see Def. 4.1. In both search directions, MAXSDIST provides admissible heuristics, and therefore allows construction of optimal plans. The $MaxWDist$ is computed similarly, but adapts the WEAKPREIMAGE preimage primitive and therefore performs a "best case" analysis when multiple outcomes are possible, opposed to $MaxSDist$'s worst case; this causes MAXWDIST not to be admissible.

In [CRB03], the two heuristic approaches are thoroughly compared, and tested against other state-of-the art approaches, on a variety of benchmarks. A different heuristic-based approach is also evaluated, where the notion of “knowledge over domain fluents” is used to dynamically identify, during the search, an agenda of subgoals that need be traversed before the goal is achieved.

4.6 Strong Cyclic Planning under Partial Observability

In this section, we intend to deal with the generation of strong cyclic plans, lifting to the general case of partial observability. However, such lifting to the case of partial observability is not immediate: while, in the fully observable case, after executing an action, we can always decide whether the goal has been reached, in the partially observable case, such "immediate" goal detection may be impossible.

There is a fundamental difference between a plan that only achieves a certain goal condition, and a plan that, in addition, guarantees that the executor will know when the goal is achieved (and will stop accordingly). The first interpretation is the standard one used in strong planning under partial observability: a solution plan is associated with final belief states that are entirely contained in the goal. The second interpretation is new: a solution plan guarantees that, for each of the states in any final belief states, a goal state was previously visited. Accordingly, we distinguish between goal achievement *with immediate detection*, and goal achievement *with delayed detection*. Obviously, the requirement of immediate achievement detection is stronger than that of delayed achievement detection.

In the following, we first state the various problems, and then we propose a family of algorithms to solve them.

4.6.1 Problem Statement

To formally characterize (strong and) strong cyclic solutions under partial observability, we will refer to the standard notion of plan presented in the reference framework. Based on that, we recap the notion of plan configuration and plan paths, and we define a notion of *observational equivalence* among paths which are indistinguishable under the point of view of an external observer. Then, based on such a notion, we can define the different Strong cyclic solutions by providing different requirements on the of traversed paths.

Definition 4.6.1 (configuration) *A* configuration *for domain* $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{R}, \mathcal{X} \rangle$ *and plan* $\Pi = \langle \Sigma, \sigma_0, \alpha \rangle$ *is a tuple* (s, o, c) *such that* $s \in \mathcal{S}$*,* $o \in \mathcal{X}(s)$*, and* $c \in \Sigma$*.*

Configuration (s, o, c) *may* evolve into *configuration* (s, o, c)*, written* $(s, o, c) \rightarrow (s', o', c')$*, if there is some action* a *such that:*

- $(a, c') \in \alpha(c, o)$*,*
- $s' \in \mathcal{R}(s, a)$*,*
- $o' \in \mathcal{X}(s')$*.*

Configuration (s, o, c) *is* initial *if* $s \in \mathcal{I}$ *and* $c = \sigma_0$*. The* reachable configurations *for domain* $\mathcal{D}$*, plan* Π*, and goal* $\mathcal{G}$*, are defined as those in the transitive closure of* $\rightarrow$ *from the initial configurations. Configuration* (s, o, c) *is* terminal *iff it is reachable and* $\alpha(c, o) = \emptyset$*.*

Definition 4.6.2 (path) *A finite [infinite]* path PATH *for a domain $\mathcal{D}$ and a plan Π is a finite [infinite] sequence of configurations $\gamma_0, \gamma_1, \ldots$ such that γ_0 is initial and $\gamma_i \rightarrow \gamma_{i+1}$ for all i. Path prefixes are defined in the usual way.*

We denote with PATHS$(\mathcal{D}, \Pi)$ *the set of the paths for $\mathcal{D}$ and Π that either are infinite or end with a terminal configuration.*

We are interested in plans that guarantee the executability of actions they produce on each reachable configuration.

Definition 4.6.3 (executable plan) *Plan Π is* executable *on domain $\mathcal{D}$ iff for each reachable configuration (s, o, c), if $(a, c') \in \alpha(c, o)$, then $\mathcal{R}(s, a) \neq \emptyset$.*

Definition 4.6.4 (equivalent paths) *Paths* PATH $= (s_0, o_0, c_0), (s_1, o_1, c_1), \ldots$ *and* PATH$' = (s'_0, o'_0, c'_0), (s'_1, o'_1, c'_1), \ldots$ *are said to be equivalent, denoted with* PATH $\equiv$ PATH$'$, *iff for every i, $o_i = o'_i$.*

Definition 4.6.5 (strong cyclic solution) *Let plan Π be executable in domain $\mathcal{D}$.*

Π *is a* strong cyclic solution *to $\mathcal{G}$ for*

- achievement with immediate detection (SCID) *iff $s \in \mathcal{G}$ holds for each terminal configuration (s, o, c), and each prefix of an infinite path in* PATHS$(\mathcal{D}, \Pi)$ *is observationally equivalent to some prefix of a finite path in* PATHS$(\mathcal{D}, \Pi)$.
- achievement with delayed detection (SCDD) *iff each finite path in* PATHS$(\mathcal{D}, \Pi)$ *traverses some state $s \in \mathcal{G}$, and each prefix of an infinite path in* PATHS$(\mathcal{D}, \Pi)$ *is observationally equivalent to some prefix of a finite path in* PATHS$(\mathcal{D}, \Pi)$.
- achievement without detection (SCND) *iff each (finite and infinite) path in* PATHS$(\mathcal{D}, \Pi)$ *traverses some state $s \in \mathcal{G}$.*

We remark that, even within a strong cyclic solution for SCND, there may exist paths for which (immediate/delayed) detection takes place. That is, there may be cases where an SCND solution terminates, just as well as for SCID and SCDD solutions; but as soon as detection cannot be always guaranteed, only SCND solutions may exist for the problem at hand.

4.6.2 Planning Algorithm

The algorithms to solve the various strong cyclic planning problems share the same top-level structure, presented in Fig. 4.25. The TopLevelPlanner routine takes in input a problem class (SCID, SCDD, SCND); $\mathcal{D}$ and $\mathcal{G}$ are assumed to be globally available to

```
function $TopLevelPlanner(PBclass)$
  $n.bs = \mathcal{I}$
  $n.os = \mathcal{I} \setminus \mathcal{G}$
  $Gph.storeNode(n)$
  $PBclass.propagateLabels(Gph)$
  while $(\neg Gph.RootSolved() \wedge \neg Gph.EmptyFrontier())$
    $n := Gph.pickNode()$
    forall $o \in \mathcal{O}.a \in \mathcal{A}$
      if $((n.bs \cap \mathcal{X}^{-1}(o)) \neq \emptyset \wedge$
        $\forall s \in (n.bs \cap \mathcal{X}^{-1}(o)) : \mathcal{R}(s,a) \neq \emptyset)$ then
        let $n'$
        $n'.bs := \mathcal{R}((n.bs \cap \mathcal{X}^{-1}(o)), a)$
        $n'.os := \mathcal{R}((n.os \cap \mathcal{X}^{-1}(o)), a) \setminus \mathcal{G}$
      $Gph.storeNode(n')$
      $Gph.storeEdge((n, o, a, n'))$
    fiendfor
  $PBclass.propagateLabels(Gph)$
return $(Gph.RootSolved())$
```

Figure 4.25: The Generic Planning Routine

the subroutines. The search iterates by extracting and expanding one node from the search graph and computing whether a solution exists; it returns when either all the nodes have been expanded (and the search frontier is empty), or when a solution has been found. Gph contains the search space explored so far. Intuitively, the search space is an and-or graph, where each edge represents a possible observation o (produced by the domain), and a subsequent action a (which can be executed given that observation). All through the search, each search node n is associated with two pieces of information: the belief state $n.bs \subseteq \mathcal{S}$ contains all the states that are compatible with the history of previous observations/actions, and the set of open states $n.os \subseteq n.bs$ stores the states of $n.bs$ that have not reached the goal in their past history.

The algorithms differ only in the labeling routine.

The main difficulty is in handling loops: it is not possible to consider a single fixed-point computation for labeling nodes as success. Admitting loops means that "weakly successful" nodes, from which termination is not guaranteed, must also be considered as taking part to possible solutions. Such nodes may be computed by a least fix point computation, starting from locally successful ones, and establishing that a node is "weakly successful" if it has at least one weakly successful successor. Once weakly successful nodes are computed, we have to discard loops where there is no chance to reach the goal. This can be performed by a subsequent greatest fix point computation, that sets as false the labels of nodes for which some observation does not lead to success (or to a successful successor). These two computations influence each other, and are interleaved until the situation stabilizes, i.e. a (higher-level) fix point is reached.

```
1 **define** $ID.gotG(n) := (n.bs \subseteq \mathcal{G})$
2 **define** $ID.gotO(n) := (n.bs \cap \mathcal{X}^{-1}(o) \subseteq \mathcal{G})$
3 **define** $DD.gotG(n) := (n.os = \emptyset)$
4 **define** $DD.gotO(n) := (n.os \cap \mathcal{X}^{-1}(o) = \emptyset)$
```

```
1  **function** SCID$.propagateLabels()$
2     **forall** $n \in Gph.nodes$
3        $label(n) := true$**endfor**
4     **repeat**
5        **forall** $n \in Gph.nodes$
6           $newlabel(n) := ID.gotG(n)$**endfor**
7        **repeat on** $n \in Gph.nodes$
8           $newlabel(n) := label(n) \wedge$
9              $(\exists o \in \mathcal{X}(n.bs) : (ID.gotO(n)) \vee$
10              $\exists n' : Gph.edge(n, o, a, n') \wedge newlabel(n'))$
11       **until fixpoint for** $newlabel(n)$
12       **repeat on** $n \in Gph.nodes$
13          $newlabel(n) := newlabel(n) \wedge$
14             $\forall o \in \mathcal{X}(n.bs) :$
15                $(ID.gotO(n)) \vee$
16                $\exists n' : \ Gph.edge(n, o, a, n') \wedge newlabel(n'))$
17       **until fixpoint for** $newlabel(n)$
18       **forall** $n \in Gph.nodes$
19          $label(n) := newlabel(n)$**endfor**
20    **until fixpoint for** $label(n)$
21 **end**
```

Figure 4.26: Labeling Routine for SCID

In the case of SCID (Fig. 4.26), the labeling function contains a high-level fix point computation at lines 4-20; within this, we first label nodes as weak solutions with the least fix point at lines 7-11; then, at lines 12-17, we relabel as not successful all those nodes for which success is not supported for every possible observation (either by a an action leading to a successful node, or by direct containment in the goal). The case of SCDD is analogous, and can be obtained by replacing the $ID.gotG$ and $ID.gotO$ success checks with $DD.gotG$ and $DD.gotO$ respectively.

The labeling routine for the case of SCND shares the same nested fix-point structure as the previous cases. However, it is more complicated, since it is no longer possible to detect whether a node can be labeled as success simply based on its belief state (as in SCID) or on its set of open states (as in SCDD). Thus, a node is no longer labeled with

```
function SCND.$propagateLabels()$
  forall $n \in Gph.nodes$
    $label(n) := n.os$ endfor
  repeat
    forall $n \in Gph.nodes$
      $newlabel(n) := \emptyset$ endfor
    repeat on $n \in Gph.nodes$
      $newlabel(n) :=$
        $\{s \in label(n) \mid$
          $\exists o \in \mathcal{X}(s).\exists n' : Gph.edge(n, o, a, n') \wedge$
            $(\mathcal{R}(s, a) \cap (newlabel(n') \cup \mathcal{G}) \neq \emptyset)\}$
    until fixpoint for $newlabel(n)$
    repeat on $n \in Gph.nodes$
      $newlabel(n) :=$
        $\{s \in newlabel(n) \mid$
          $\forall o \in \mathcal{X}(s).\exists n' : Gph.edge(n, o, a, n') \wedge$
            $\mathcal{R}(s, a) \subseteq (newlabel(n') \cup \mathcal{G})\}$
    until fixpoint for $newlabel(n)$
    forall $n \in Gph.nodes$
      $label(n) := newlabel(n)$ endfor
  until fixpoint for $label(n)$
end
```

Figure 4.27: Labeling Routine for SCND

a truth value, but with a set of states (in particular, a subset of the open states). A state is in the label when it has been checked to satisfy the goal, i.e. all its possible future executions either reach $\mathcal{G}$ or loop through states from which $\mathcal{G}$ can be reached. A node is then successful iff its label is equal to its set of open states. The external fix point at lines 4-21 iterates by inserting into the label every weakly successful state (lines 7-12), and then (lines 13-18) by removing those states whose presence is not justified by the existence, for each observation, of an action that guarantees either reaching $\mathcal{G}$ or a state (temporarily) labeled as success.

All the reported algorithms are correct and complete, and always terminate.

Notice that in this case, the algorithms, in general, actually produce nondeterministic plans. These can either be directly executed via a scheduler that must choose "fairly" amongst different possible actions, or transformed offline into deterministic plans. Producing nondeterministic plans is crucial to the effectiveness of the algorithms, since the deterministic counterpart Π_D of a nondeterministic plan Π_{ND} may be exponentially larger than Π_{ND}, since the states of Π_D must encode the history of the traversed states of Π_{ND}.

Chapter 5

On-line planning

In this chapter, we discuss on-line based approaches to solve planning problems. We will always refer to the general setting of partially observable domains, to focus on issues that are independent from the degree of run-time observability featured by the domains. In particular, we consider two alternate, orthogonal approaches to relieve from the computational cost of computing (off-line) "complete" plans that, alone, solve the specified problem.

The first approach is inspired by "reactive" planning approaches such as those in [KS97, Koe01]. Here, instead of waiting the planner to generate a plan that gets to the goal, in general, we interrupt the search based on some stopping criteria (e.g. time consumption), and take as a plan the most promising course of actions found so far. This means that, after executing the plan, unless the goal has been achieved, we will need to plan again, and this may take place several times before the goal is reached. This way of interleaving planning with execution saves a lot of effort, since only a limited amount of computational resources is spent, at each episode, to consider contingencies, and we never consider the enormous number of contingencies that might happen far away in the future. Indeed, execution and monitoring makes us tackle, in the future, just the one behavior that actually took place. This is explained by Fig. 5.1: the search space is visited bit by bit by planning, and most of it is never considered. Of course this comes at a cost: plans built with a finite horizon might commit to dangerous actions that can never be retracted, or different planning episodes may form an infinite loop which cannot be discovered by the planner alone. For the former problem, several solutions have been proposed, mainly based on the idea of approximate reachability analysis. We will focus on the latter problem and propose a solution.

The second approach is based on the idea of using "assumptions" that identify expected/nominal behaviors of the environment, and then use them to constrain plan generation. In this way, the generated plan would control the domain to achieve the goal, proviso the assumption holds. Again, using assumptions may rule out a very large portion of the contingencies that, otherwise, would be considered during the search, therefore im-

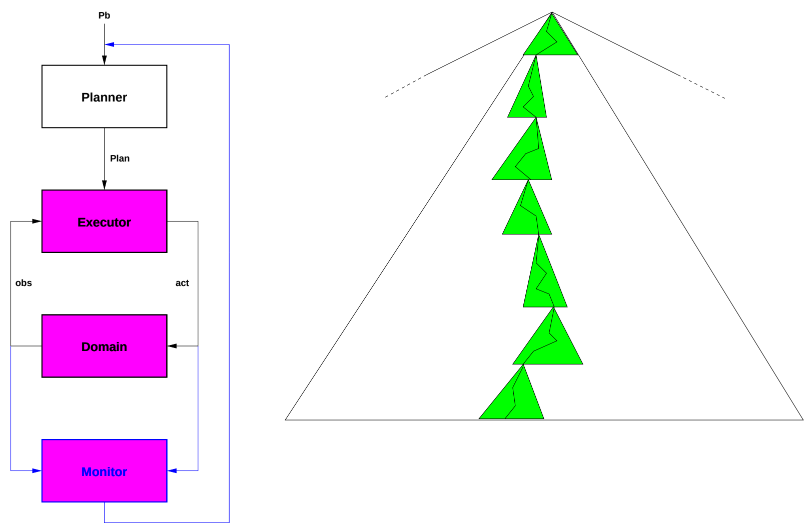


Figure 5.1: Framework and reactive execution

proving performance in a very significant way. Notice that the way in which the search space is reduced is, in a sense, orthogonal to the way it happens in reactive planning: here, we do not limit the "depth" of the search, but rather its "width", see Fig. 5.2.

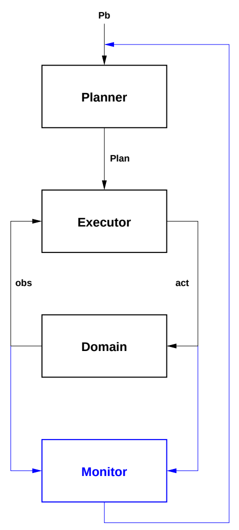
Also in this case, in general, it becomes necessary to be able to perform re-planning. This is because the assumptions taken prior to execution might actually not hold, in which case the plan should be stopped and a new (possibly assumption-based) plan must be produced, taking as a starting point the new domain situation. One issue related with this approach is that, due to partial observability, a monitor may have limited abilities in discovering the failure of assumptions. This makes it hard to decide whether re-planning must take place, and may originate run-time problems, to the point of preventing the achievement of goals. Our assumption-based approach tackles this issue by requiring that generated plans obey some monitorability constraint, called "safety". In this way, monitoring can gather enough information to be able to uniquely decide whether re-planning must actually be invoked.

In the following two sections, we discuss our on-line planning approaches in turn.

5.1 Loop-free reactive planning

Here, we intend to develop a planning algorithm that, once embedded in a reactive platform, guarantees the key property of execution termination, but referring to the overall architecture. That is, it is not enough to guarantee that the single planning instance terminates, since, unless special care is taken, re-planning could take place an infinite number of times.

To do so, we start from our standard planning algorithm for strong goals under partial



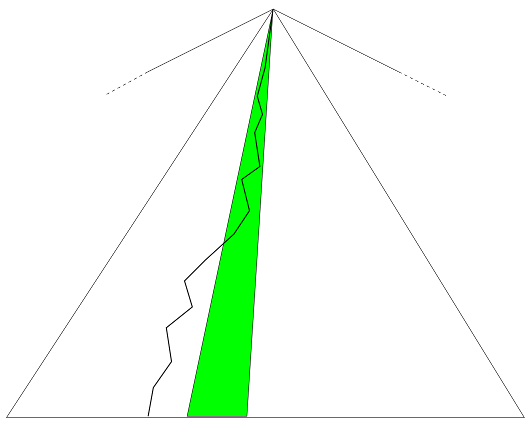


Figure 5.2: Framework and assumption-based execution

observability, and enrich it by adding some termination condition that force the newly produced plan to make some progress towards the goal. Such a condition, which we will call "progressiveness", is based on storing the history of traversed beliefs of past plans, called "belief execution", see Def.4.4.12.

Thus, our starting point is the planning algorithm depicted in Figure 5.3, disregarding the lines with boldfaced labels. This is a slight modification for strong planning under partial observability described in Section 4.4. The algorithm takes as input the initial belief state and the goal belief state, and proceeds by incrementally constructing a finite acyclic prefix of the search space, implemented as a $graph$. In the graph, each node n is associated with a belief state $b(n)$; a directed connection between a node n_1 and a node n_2 results either from an action α such that $Exec(\alpha, b(n_1)) = b(n_2)$, or from an observation o such that $b(n_1) \cap \mathcal{X}_o[v] = b(n_2)$, with $v = \top$ or $v = \bot$. We call n_1 the father of n_2 and n_2 the son of n_1; we call "brothers" all the nodes that result from the same observation expansion of the same node. The graph is annotated with a frontier of the nodes that have not yet been expanded, and with a success pool, containing the nodes for which a strong plan has been found.

The algorithm has its core in a search loop (lines 2-21), iteratively selecting and expanding a node in the graph. Namely, at each iteration, a node is extracted from the frontier, and evaluated for success against the success pool (lines 6-7). If the node successful, a strong plan is extracted and associated to it, the success pool is expanded, and success is propagated backward on the graph (lines 8-10). Otherwise, the node is expanded by applying every executable action, and non-trivial observation to it, resulting into a graph expansion (lines 12-13). The expansion routine avoids generating ancestors of the expanded node, inhibiting the presence of loops in the graph. The search loop terminates either when (a) the root of the graph is signaled as a success node, (b) the graph frontier is empty, or (c) a termination criterion is met. Condition (a) signals that a strong

```
ProgressivePlan($I$, $G$)
1   $graph$ := MkInitialGraph($I$, $G$);
2   while ( ¬isSucc(getRoot($graph$)) ∧
3             ¬isEmptyFrontier($graph$) ∧
4             ¬(isProgress(getRoot($graph$)) ∧
5                TerminationCriterion($graph$)))
6        $node$ := ExtractNodeFromFrontier($graph$);
7        if (SuccessPoolYieldsSuccess($node$, $graph$))
8             MarkSuccess($node$);
9             NodeSetPlan($node$,RetrievePlan($node$, $graph$));
10            PropagateSuccess($node$,$graph$);
11       else
12            $exp$ := ExpandNode($node$);
13            ExtendGraph($exp$, $node$, $graph$);
14       if (¬isExecuted($node$))
15            MarkProgress($node$, $graph$);
16            PropagateProgress($node$, $graph$);
17  end while
18  if (isSucc(getRoot($graph$)))
19       return ExtractSuccessPlan($graph$);
20  if (reachedTermination)
21       return ExtractPartialPlan($graph$);
20  if (isProgress(getRoot($graph$)))
21       return ExtractProgressingPlan($graph$);
22  return $Failure$;
```

Figure 5.3: The planning algorithm

plan has been found; condition (b) indicates that no strong plan exists; condition (c) is responsible for the search being stopped while only partial plans have been expanded.

Notice that the criterion defining condition (c) is the only distinction with the original off-line approach, for the purpose of integrating the planner within the interleaving framework, added to generate possibly partial plans rather than searching the whole search space. (Different termination criteria could be envisaged, e.g. partial success, number of nodes, run times. The specific details are not relevant here.) Unfortunately, this simple minded approach does not guarantee termination of the overall interleaving loop, even if a solution exists. The problem is that the planner should guarantee that, for every possible belief execution, at least a new belief state is reached during execution. If this is not so, plan-execution loops are possible that keep visiting the same beliefs over and over, never terminating. If instead the guarantee is achieved, termination of plan-execution loops follows from the fact that belief states are finitely many. (Notice that weakening the condition, and accepting plans which only *might* result in beliefs never been visited before, does not guarantee termination.) The notion that guarantees the termination of the top level is what we call *progressiveness* of the planner: each plan must guarantee that at least one *belief state* (not just a state) is traversed that has not been previously encountered during execution.

Definition 5.1.1 (Progressive Plan) *Let r be a belief execution $b_1, \ldots, b_n$. Let π be a plan for $\mathcal{D}$. The plan π is progressive for the belief execution r iff, for any belief execution r_π of π from b_n, there is at least one belief state in r_π that is not a belief state of r.*

Let us consider again the statements in Figure 5.3 (with lines 20 and 21 bold, replacing the non-bold counterparts). In order to obtain a progressive planning algorithm, we consider all the plans originating from a node, to make sure that at least for one of them, execution will visit a belief state, which has never been visited during previous executions. The graph is therefore extended in order to maintain up-to-date information on progress of nodes. In order to guarantee progressiveness, at line 14, we check if $b(node)$ has already been visited at execution time. If not, we mark $node$ as a "progress" node (line 15) (we remark that a successful node has surely not been visited by a previous belief execution, and as such it is marked as progress). In that case, the progress information is recursively propagated bottom-up on the tree (PROPAGATEPROGRESSONTREE, line 16): if the node is the result of the application of an action, then its father is marked as progress. If the node is the result of an observation, in order to propagate its progress backward it is necessary to check that all of its brothers are also marked as progress nodes.

Finally, when the loop is exited, either a strong plan has been found, and is returned by EXTRACTSUCCESSPLAN; or, a progressing plan exists in the graph, and is extracted by EXTRACTPROGRESSINGPLAN; or, failure is returned. While extracting the success plan is simple (it is associated with I by the bottom-up propagation), the progressing plan might not be unique: several such plans may exist. The selection operated by EXTRACT-PROGRESSINGPLAN may affect the overall performance. Our implementation privileges, amongst progressing plans, the ones performing more observations.

The algorithm presented above is guaranteed to terminate, based on the fact that the explored search space is finite. It either returns a strong solution plan, a progressing plan w.r.t. the current belief execution for the top level, or failure. The planner is guaranteed to be progressive by the fact that the main loop cannot exit before a progressing plan is found, unless the whole reachable belief space has been searched, or success takes place. Notice that the termination criterion is inhibited if the progressiveness condition holds false. Also, the architecture is guaranteed not to terminate before the reachable belief space is exhausted, thus it will not return failure for the planning problem corresponding to the current belief execution, unless no chance of producing a strong plan for the goal exists.

5.2 Assumption-based (re-)planning

In this section, we aim to make optimal use of the fact that, in many cases, it is possible to express reasonable assumptions over the expected dynamics of the domain, e.g. by identifying "nominal" behaviors. Using these assumptions to constrain the search may greatly ease the planning task, allowing the efficient construction of assumption-based solutions. Of course, assumptions taken when generating a plan may turn out to be incorrect when executing it. For this reason, assumption-based plans must be executed within reactive architectures such as [MNPW98, MW98], where a monitoring component traces the status of the domain, in order to abort plan execution and replan whenever an unexpected behavior has compromised the success of the plan. However, due to the incomplete run-time knowledge on the domain state, it may not be possible for a monitor to establish unambiguously whether the domain status is evolving as expected or not; in this case, replanning must occur whenever a dangerous state *may* have been reached. But if the actual domain state is one of those planned for, replanning is unnecessary and undesired. These situations can only be avoided if states not planned for can unambiguously be identified at plan execution time; in turn, whether this is possible crucially depends on the actions performed by the plan.

Here, we consider an expressive language that provides us with the key ability to specify assumptions over the domain dynamics, called linear temporal logics (LTL, [Eme90]). First, we provide an effective, symbolic mechanism to constrain forward and-or search to generate (conditional) LTL assumption-based solutions for nondeterministic, partially observable domains. Second, we further constrain the search to obtain *safe* LTL assumption-based solutions, i.e. plans that not only guarantee that the goal is reached when the given assumption holds, but also guarantee that, during their execution, the monitor will be able to unambiguously distinguish whether the current domain status has been planned for or not. In this way, we guarantee that, during plan execution, the monitor will not trigger any plan abortion unless really needed.

We start by redefining execution traces of plans in order to distinguish whether the plan attempted a non-executable action, originating a run-time failure:

Definition 5.2.1 (Traces of a plan) *A trace of a plan is a sequence $[s^0, o^0, \alpha^0, \ldots, s^n, o^n, End]$, where s^i, o^i are the domain state and the related observation at step i of the plan execution, and α^i is the action produced by the plan on the basis of o^i (and of the plan's internal state).* End *can either be* Stop*, indicating that the plan has terminated, or $Fail(\alpha^n)$, indicating execution failure of action α^n on s^n. We also indicate a trace with $\langle \bar{s}, \bar{o}, \bar{\alpha} \rangle$, splitting it into sequences of states, observations, and actions, respectively, and omitting the final symbol.*

A trace t is a goal trace *for problem $\langle \mathcal{D}, \mathcal{G} \rangle$ iff it is not a failure trace, and its final state, denoted $final(t)$, belongs to $\mathcal{G}$. We indicate with $Trs(\pi, \mathcal{D})$, the set of traces associated to plan π in domain $\mathcal{D}$. $TrsFail(\pi, \mathcal{D})$ and $TrsG(\pi, \mathcal{D}, \mathcal{G})$ are, respectively, the subsets of the failure and goal traces in $Trs(\pi, \mathcal{D})$.*

With this definition, a plan is said to be a strong solution for a planning problem $\langle \mathcal{D}, \mathcal{G} \rangle$ iff every execution does not fail, and ends in $\mathcal{G}$:

Definition 5.2.2 (Strong solution) *A plan π is a strong solution for a problem $\langle \mathcal{D}, \mathcal{G} \rangle$ iff $Trs(\pi, \mathcal{D}) = TrsG(\pi, \mathcal{D}, \mathcal{G})$*

We now observe that the paths of plans induce a partition in the traces:

Definition 5.2.3 (Traces bound to a path) *Given a path p, a planning domain $\mathcal{D}$, and a belief state B, we call $Trs_{\mathcal{D}}(p, B)$ the set of traces associated to the path p:*

- $Trs_{\mathcal{D}}(\epsilon, B) = \{[s, o, Stop] : s \in B, o \in \mathcal{X}(s)\}$
- $Trs_{\mathcal{D}}(\alpha \circ p, B) =$
  $\{[s, o, Fail(\alpha)] : s \in B, o \in \mathcal{X}(s), \alpha(s) = \emptyset\} \cup$
  $\{[s, o, \alpha] \circ t : s \in B, o \in \mathcal{X}(s), \alpha(s) \neq \emptyset, t \in Trs_{\mathcal{D}}(p, \alpha(s))\}$
- $Trs_{\mathcal{D}}(o \circ p, B) = Trs_{\mathcal{D}}(p, B \cap [[o]])$
- $Trs_{\mathcal{D}}(\tilde{o} \circ p, B) = Trs_{\mathcal{D}}(p, B \cap [[\tilde{o}]])$, *where* $[[\tilde{o}]] = \bigcup_{o' \in \mathcal{O} \setminus \{o\}} [[o']]$

To express assumptions over the behavior of $\mathcal{D}$, we adopt *Linear Temporal Logic* (LTL) [Eme90], whose underlying ordered structure of time naturally models the dynamic evolution of domain states [CDGV02]. LTL expresses properties over sequences of states, by introducing the operators **X** (next) and **U** (until):

Definition 5.2.4 (LTL syntax) *The language $\mathcal{L}(\mathcal{P})$ of the LTL formulæ φ on $\mathcal{P}$ is defined by the following grammar, where $q \in \mathcal{P}$:*

$$\varphi := q || \neg\varphi || \varphi \wedge \varphi || X\varphi || \varphi U \varphi$$

The derived operators $\mathbf{F}$ (future) and $\mathbf{G}$ (globally) are defined on the basis of $\mathbf{U}$: $\mathbf{G}\varphi = \varphi\,\mathbf{U}\,\bot$ and $\mathbf{F}\varphi = \neg\mathbf{G}\neg\varphi$. The semantics of LTL formulæ are given inductively on infinite state sequences, see [MP92].

Definition 5.2.5 (LTL semantics) *Given an infinite state sequence $\sigma = [s^0, \ldots, s^n, \ldots]$, we denote by $(\sigma, i) \models \varphi$ whether a formula $\varphi \in \mathcal{L}(\mathcal{P})$ holds at a position $i \geq 0$ in σ. The semantics for each of the operators and subformulæ provided by the syntax is defined as follows:*

$$\begin{array}{ll}
(\sigma,i) \models \varphi \quad \textit{iff} & s^i \models \varphi \textit{ and } \varphi \in \mathcal{P}rop(\mathcal{P}) \\
(\sigma,i) \models \neg\varphi \quad \textit{iff} & (\sigma,i) \not\models \varphi \\
(\sigma,i) \models \varphi \wedge \psi \quad \textit{iff} & (\sigma,i) \models \varphi \textit{ and } (\sigma,i) \models \psi \\
(\sigma,i) \models \mathbf{X}\varphi \quad \textit{iff} & (\sigma,i+1) \models \varphi \\
(\sigma,i) \models \varphi\,\mathbf{U}\psi \quad \textit{iff} & \exists k \geq j : (\sigma,k) \models \psi \textit{ and} \\
\qquad \forall i : j \leq i < k, & (\sigma,i) \models \varphi \textit{ or } (\sigma,j) \models \mathbf{G}\varphi
\end{array}$$

We say that an LTL formula φ is satisfiable over a plan trace $\langle \bar{s}, \bar{o}, \bar{\alpha} \rangle \in Trs(\pi, \mathcal{D})$ iff it holds at position 0 for some infinite prolongation of $\bar{s}$.

Thus, given an LTL assumption $\mathcal{H}$, the finite traces $Trs(\pi, \mathcal{D})$ of a plan π can be partitioned into those traces for which $\mathcal{H}$ is satisfiable, and those for which it is not, denoted $Trs_{\mathcal{H}}(\pi, \mathcal{D})$ and $Trs_{\bar{\mathcal{H}}}(\pi, \mathcal{D})$ respectively. The failure traces $TrsFail(\pi, \mathcal{D})$ are partitioned analogously into $TrsFail_{\mathcal{H}}(\pi, \mathcal{D})$ and $TrsFail_{\bar{\mathcal{H}}}(\pi, \mathcal{D})$, and so for $TrsG(\pi, \mathcal{D}, \mathcal{G})$, partitioned into $TrsG_{\mathcal{H}}(\pi, \mathcal{D}, \mathcal{G})$ and $TrsG_{\bar{\mathcal{H}}}(\pi, \mathcal{D}, \mathcal{G})$.

If every possible execution for which $\mathcal{H}$ is satisfiable succeeds, then the plan is a solution under assumption $\mathcal{H}$:

Definition 5.2.6 (Solution under Assumption) *A plan π is a solution for the problem $\langle \mathcal{D}, \mathcal{G} \rangle$ under the assumption $\mathcal{H} \in \mathcal{L}(\mathcal{P})$ iff* $\quad Trs_{\mathcal{H}}(\pi, \mathcal{D}) = TrsG_{\mathcal{H}}(\pi, \mathcal{D}, \mathcal{G})$

Example 5.2.7 *We introduce a simple navigation domain for explanatory purposes, see Fig. 5.4. A mobile robot, initially placed in room I, must reach room K_3. The shaded cells K_1, K_2, K_3 are kitchens, while the other ones are normal rooms. The robot is only equipped with a smell sensor **K**, that allows to detect whether it is in a kitchen room. The robot moves at each step in one of the four compass directions (n, e, s, o); moving onto a wall is not possible.*

We do not know the speed of the robot: thus a movement might terminate in any of the rooms in the movement's direction (for instance, moving north from room I may end up in K_1, R_2 or in R_3).

A strong solution for this problem does not exist, as it is easy to see. However, solutions exists if we assume that the robot steps of one room at a time, at least until it reaches the goal. The considered assumption formula is $\mathcal{H}_0 = (\mathbf{X}(\delta = 1)\,\mathbf{U}\,K_3)$,

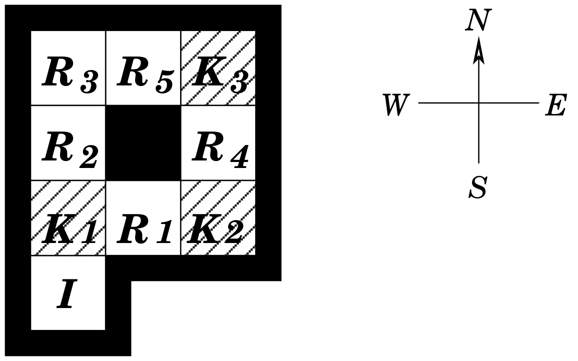


Figure 5.4: An example domain.

where δ *models the Manhattan distance between two successive cells. The simple plan* $\pi_1 = n \circ n \circ n \circ e \circ e \circ \epsilon$ *is a solution under* $\mathcal{H}_0$*. This plan has the following possible traces* $Trs(\pi_1, \mathcal{D})$*:*

$$t_1 : [I, \overline{K}, n, K_1, K, n, R_2, \overline{K}, n, R_3, \overline{K}, e, R_5, \overline{K}, e, K_3, K, Stop.]$$
$$t_2 : [I, \overline{K}, n, K_1, K, n, R_3, \overline{K}, Fail(n)]$$
$$t_3 : [I, \overline{K}, n, R_2, \overline{K}, n, R_3, \overline{K}, Fail(n)]$$
$$t_4 : [I, \overline{K}, n, R_3, \overline{K}, Fail(n)]$$
$$t_5 : [I, \overline{K}, n, K_1, K, n, R_2, \overline{K}, n, R_3, \overline{K}, e, K_3, K, Fail(e)]$$

Indeed $Trs_{\mathcal{H}_0}(\pi_1, \mathcal{D}) = TrsG_{\mathcal{H}_0}(\pi_1, \mathcal{D}, \mathcal{G}) = \{t_1\}$.

At run-time, the domain may exhibit a behavior that does not satisfy the assumptions considered at planning-time. For this reason, assumption-based plans are usually executed within a reactive framework (see e.g. [BCR01a, MNPW98]), where an external beholder of the execution (called *monitor*) observes the responses of the domain to the actions coming from the plan, and triggers replanning when such observations are not compatible with some expected behavior.

However, the monitor may not always be able to decide whether the domain is reacting as expected or not to the stimuli of a plan π. This happens when the observations gathered are not informative enough to decide whether the domain is behaving as expected or not, i.e. when given a sequence of actions $\bar{\alpha}$, the domain produces a sequence of observations $\bar{o}$ compatible both with some assumed domain behavior, and with some behavior falsifying the assumption:

$$\exists \bar{s}, t, \bar{s}', t' : \langle \bar{s}, \bar{o}, \bar{\alpha} \rangle \circ t \in Trs_{\mathcal{H}}(\pi, \mathcal{D}) \wedge \langle \bar{s}', \bar{o}, \bar{\alpha} \rangle \circ t' \in Trs_{\overline{\mathcal{H}}}(\pi, \mathcal{D}).$$

In situations where such ambiguity on the current state makes the applicability of the next action also ambiguous, the monitor will trigger replanning, in order to rule out any chance of a run-time failure. But the next action might actually be applicable, even though this is not discernible: then, plan execution should not be interrupted, and replanning is unnecessary and undesirable.

Example 5.2.8 *Consider the plan π_1 from example 5.2.7. Trace t_2 produces the same observations (namely $[\bar{K}, K, \bar{K}]$) of the successful trace t_1, in response to the same actions ($n \circ n$), up to its failure. Thus, after the first n move, the monitor knows the robot is in K_1; after the second, the monitor cannot distinguish if the robot is in R_2 or R_3. In case it is in R_3, the next north action is inapplicable, so the plan execution has to be stopped. This makes π_1 practically useless, since its execution is always halted after two actions, even if the assumption under which it guarantees success holds.*

Similarly, an assumption-based solution plan may terminate without the monitor being able to distinguish whether the goal has been reached or not, therefore triggering replanning at the end of plan execution.

Whether such situations occur depend on the nature of the problem and, crucially, on the plan chosen as an assumption-based solution. We are interested in characterizing and generating assumption-based solutions such that these situations do not occur, i.e. every execution that causes action inapplicability or ends up outside the goal, must be distinguishable from every successful execution.

Thus, we define safe LTL assumption-based plans as those plans that (a) achieve the goal whenever the assumption holds, and (b) guarantee that each execution where an assumption failure compromises the success of the plan is observationally distinguishable (from the monitor's point of view) from any successful execution.

Definition 5.2.9 (Distinguishable traces) *Let t and t' be two traces; let $L = min(t, t')$, $\lfloor t \rfloor_L = \langle \bar{s}, \bar{o}, \bar{\alpha} \rangle$ and $\lfloor t' \rfloor_L = \langle \bar{s}', \bar{o}', \bar{\alpha}' \rangle$. Then t, t' are distinguishable, denoted $Dist(t, t')$, iff $(\overline{o} \neq \overline{o}') \vee (\overline{\alpha} \neq \overline{\alpha}')$.*

Definition 5.2.10 (Safe LTL assumption-based solution) *A plan π is a safe assumption-based solution for assumption $\mathcal{H} \in \mathcal{L}(\mathcal{P})$ iff the conditions below are met:*

$$\begin{aligned} &a) \ Trs_{\mathcal{H}}(\pi, \mathcal{D}) = TrsG_{\mathcal{H}}(\pi, \mathcal{D}, \mathcal{G}) \\ &b) \ \forall t \in Trs_{\overline{\mathcal{H}}}(\pi, \mathcal{D}) \backslash TrsG_{\overline{\mathcal{H}}}(\pi, \mathcal{D}, \mathcal{G}), \\ &\quad \forall t' \in Trs_{\mathcal{H}}(\pi, \mathcal{D}) \cup TrsG_{\overline{\mathcal{H}}}(\pi, \mathcal{D}, \mathcal{G}) : Dist(t, t') \end{aligned}$$

Example 5.2.11 *Consider the plan $\pi_2 = n \circ e \circ e \circ n \circ n \circ \epsilon$. This is a safe assumption-based solution for the problem of example 5.2.7, as it is easy to see. The traces $Trs(\pi_2, \mathcal{D})$ are:*

$$\begin{aligned} &t'_1 : [I, \overline{K}, n, K_1, K, e, R_1, \overline{K}, e, K_2, K, n, R_4, \overline{K}, n, K_3, K, Stop] \\ &t'_2 : [I, \overline{K}, n, K_1, K, e, R_1, \overline{K}, e, K_2, K, n, K_3, K, Fail(n)] \\ &t'_3 : [I, \overline{K}, n, K_1, K, e, K_2, K, Fail(e)] \\ &t'_4 : [I, \overline{K}, n, R_2, \overline{K}, Fail(e)] \\ &t'_5 : [I, \overline{K}, n, R_3, \overline{K}, e, R_5, \overline{K}, e, K_3, K, Fail(n)] \\ &t'_6 : [I, \overline{K}, n, R_3, \overline{K}, e, K_3, Fail(e)] \end{aligned}$$

We have $TrsG_{\mathcal{H}_0}(\pi_2, \mathcal{D}, \mathcal{G}) = Trs_{\mathcal{H}_0}(\pi_2, \mathcal{D}) = \{t'_1\}$*, and every trace* $t'_2, t'_3, t'_4, t'_5, t'_6$ *is distinguishable from* t'_1*.*

Again, to efficiently generate safe LTL assumption-based plans for partially observable, nondeterministic domains. we take as a starting point the plan generation approach presented in Section 4.4. There, an and-or graph representing an acyclic prefix of the search space of beliefs is iteratively expanded: at each step, observations and actions are applied to a fringe node of the prefix, removing loops. Each node in the graph is associated with a belief in the search space, and to the path of actions and observations that is traversed to reach it. When a node is marked success, by goal entailment or by propagation on the graph, its associated path is eligible as a branch of a solution plan. The implementation of this approach by symbolic techniques has proved very effective in dealing with complex problems, where uncertainty results in manipulating large beliefs.

To generate plans under an LTL assumption $\mathcal{H}$, using this approach, we have to adapt this schema so that the beliefs generated during the search only contain states that can be reached if $\mathcal{H}$ is satisfiable. Moreover, in order to constrain the algorithm to produce safe plans, success marking of a leaf node n must require the safety conditions of def.5.2.10 (recast to those traces in $Trs_{\mathcal{H}}$ and $Trs_{\overline{\mathcal{H}}}$ that can be traversed to reach n). We now describe in detail these adaptations, and the way they are efficiently realized by means of symbolic representation techniques.

5.2.1 Assumption-induced pruning

In order to prune states that may only be reached if the assumption is falsified, we annotate each state s in a (belief associated to a) search node with an LTL formula φ, representing the current assumption on how s will evolve over the timeline, and we progress the pair $\langle s, \varphi \rangle$ in a way conceptually similar to [KBSD97]. Thus, a graph node will now be associated to a set $\mathcal{B} = \{\langle s, \varphi \rangle : s \in \mathcal{S}, \varphi \in \mathcal{L}(\mathcal{P})\}$ called *annotated belief*, whose states will be denoted with $St(\mathcal{B}) = \{s : \langle s, \varphi \rangle \in \mathcal{B}\}$. Formulæ will be expressed by unrolling top-level $\mathbf{U}$s in terms of their recursive semantic definition, to allow evaluating them on the current state:

$$\begin{array}{l}
Unroll(\psi) = \psi \quad \text{if } \psi \in \{\top, \bot\} \cup \mathcal{P}rop(\mathcal{P}) \\
Unroll(\neg\varphi) = \neg Unroll(\varphi) \\
Unroll(\varphi \vee \psi) = Unroll(\varphi) \vee Unroll(\psi) \\
Unroll(\varphi \wedge \psi) = Unroll(\varphi) \wedge Unroll(\psi) \\
Unroll(\mathbf{X}\varphi) = \mathbf{X}\varphi \\
Unroll(\varphi \, \mathbf{U} \, \psi) = Unroll(\psi) \vee (Unroll(\varphi) \wedge \mathbf{X}(\varphi \, \mathbf{U} \, \psi))
\end{array}$$

Thus, the initial graph node will be

$$\mathcal{B}_{\mathcal{I}} = \{\langle s, Unroll(\mathcal{H})|_s \rangle : \; s \in \mathcal{I}, \; Unroll(\mathcal{H})|_s \neq \bot\} \tag{5.1}$$

where $\varphi|_s$ denotes the formula resulting from φ by replacing its subformulæ outside the scope of temporal operators with their evaluation over state s. Notice that the formulæ associated to states have the form $\bigwedge \varphi_i \wedge \bigwedge \mathbf{X}\psi_j$, with $\varphi_i, \psi_j \in \mathcal{L}(\mathcal{P})$. When a fringe node in the graph is expanded, its associated annotated belief $\mathcal{B}$ is progressed as follows:

- if an observation o is applied, $\mathcal{B}$ is restricted to

$$\mathcal{E}(\mathcal{B}, o) = \{\langle s, \varphi\rangle \in \mathcal{B} : \ s \in [[o]]\} \tag{5.2}$$

- if an (applicable) action α is applied, each pair $\langle s, \varphi\rangle \in \mathcal{B}$ is progressed by rewriting φ to refer to the new time instant, unrolling untils, and evaluating it on every state in $\alpha(s)$:

$$\mathcal{E}(\mathcal{B}, \alpha) \ = \ \{\langle s', X^{-1}(\varphi)|_{s'}\rangle \ : \ \langle s, \varphi\rangle \ \in \ \mathcal{B}, \ \ s' \ \in \ \alpha(s), X^{-1}(\varphi)|_{s'} \ \neq \ \bot\} \tag{5.3}$$

where $X^{-1}(\varphi)$ is defined as follows:

$$\begin{aligned}
&X^{-1}(\psi) = \psi \quad \text{if } \ \psi \in \{\top, \bot\}\\
&X^{-1}(\neg\psi) = \neg X^{-1}(\psi)\\
&X^{-1}(\varphi \vee \psi) = X^{-1}(\varphi) \vee X^{-1}(\psi)\\
&X^{-1}(\varphi \wedge \psi) = X^{-1}(\varphi) \wedge X^{-1}(\psi)\\
&X^{-1}(\mathbf{X}\varphi) = Unroll(\varphi)
\end{aligned}$$

5.2.2 Enforcing safety

Def. 5.2.10.a is easily expressed as an entailment between the states of the current annotated belief and the goal. To efficiently compute the distinguishability (requirement 5.2.10.b), we associate to a search node the sets of indistinguishable final states of $Trs_{\mathcal{H}}(p, \mathcal{D})$ and $Trs_{\bar{\mathcal{H}}}(p, \mathcal{D})$, storing them within a couple of annotated beliefs $\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}}\rangle$. When $\mathcal{B}_{\mathcal{H}}$ and $\mathcal{B}_{\bar{\mathcal{H}}}$ only contain goal states, this indicates that the only indistinguishable behaviors lead to success, so the path satisfies requirement 5.2.10.b (in particular, if $\mathcal{B}_{\mathcal{H}}$ and $\mathcal{B}_{\bar{\mathcal{H}}}$ are empty, this indicates that the monitor will be able to distinguish any assumption failure along p).

During the search, $\mathcal{B}_{\mathcal{H}}$ and $\mathcal{B}_{\bar{\mathcal{H}}}$ are progressed similarly as the annotated belief representing the search node; but on top of this, they are pruned from the states where the success or failure of the assumption can be distinguished by observations. In fact, while progressing $\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}}\rangle$, we detect situations where indistinguishable assumption failures may compromise action executability. These situations inhibit the safety of the plan, and as such we cut the search on these branches of the graph.

Initially, we compute $\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}}\rangle$ by considering those states of $\mathcal{I}$ for which $\mathcal{H}$ (resp. $\bar{\mathcal{H}}$) is satisfiable, and by eliminating from the resulting couple the states distinguishable thanks to some initial observation, i.e.

$$\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}}\rangle = \big\langle prune\big(\mathcal{B}^0_{\mathcal{H}}, \mathcal{B}^0_{\bar{\mathcal{H}}}\big), prune\big(\mathcal{B}^0_{\bar{\mathcal{H}}}, \mathcal{B}^0_{\mathcal{H}}\big)\big\rangle \tag{5.4}$$

where

$$\begin{aligned}
&\mathcal{B}^0_{\mathcal{H}} = \big\{\langle s, Unroll(\mathcal{H})|_s\rangle : s \in \mathcal{I},\ Unroll(\mathcal{H})|_s \neq \bot\big\} \\
&\mathcal{B}^0_{\bar{\mathcal{H}}} = \big\{\langle s, Unroll(\bar{\mathcal{H}})|_s\rangle : s \in \mathcal{I},\ Unroll(\bar{\mathcal{H}})|_s \neq \bot\big\} \\
&prune(\mathcal{B}, \mathcal{B}') = \big\{\langle s, \varphi\rangle \in \mathcal{B} :\ \exists \langle s', \varphi'\rangle \in \mathcal{B}' : \mathcal{X}(s) \cap \mathcal{X}(s') \neq \emptyset\big\}
\end{aligned}$$

When a node is expanded, its associated pair is expanded as follows:

- if an observation o is applied, $\mathcal{B}_{\mathcal{H}}$ and $\mathcal{B}_{\bar{\mathcal{H}}}$ are simply pruned from the states not compatible with o:

$$\mathcal{E}(\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}}\rangle, o) = \langle \mathcal{E}(\mathcal{B}_{\mathcal{H}}, o), \mathcal{E}(\mathcal{B}_{\bar{\mathcal{H}}}, o)\rangle$$

- If the node is expanded by an action α, and α is not applicable on some state of $\mathcal{B}_{\bar{\mathcal{H}}}$, then a “dangerous” action is attempted on a state that can be reached by an indistinguishable assumption failure. This makes the plan unsafe: as such, we mark the search node resulting from this expansion as failure.
- If the node is expanded by an action α, and α is applicable on every state of $\mathcal{B}_{\mathcal{H}}$ and $\mathcal{B}_{\bar{\mathcal{H}}}$, then
  $\mathcal{E}(\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}}\rangle, \alpha) = \langle prune(\mathcal{B}'_{\mathcal{H}}, \mathcal{B}'_{\bar{\mathcal{H}}}), prune(\mathcal{B}'_{\bar{\mathcal{H}}}, \mathcal{B}'_{\mathcal{H}})\rangle$
  where $\mathcal{B}'_{\mathcal{H}} = \mathcal{E}(\mathcal{B}_{\mathcal{H}}, \alpha)$ and $\mathcal{B}'_{\bar{\mathcal{H}}} = \mathcal{E}(\mathcal{B}_{\bar{\mathcal{H}}}, \alpha)$.

The safety of a plan π can be evaluated by considering an associated execution tree $\tau(\pi, \mathcal{I}, \mathcal{H})$, built by progressing and pruning annotated beliefs as described so far. The nodes of $\tau(\pi, \mathcal{I}, \mathcal{H})$ are either annotated beliefs or $Fail$ nodes, and we denote its leaves with $leaf\big(\tau(\pi, \mathcal{I}, \mathcal{H})\big)$. For π to be safe, its associated execution tree must be failure-free, and the states in the leaves, unless unambiguously related to the assumption being false, must be associated to goal states:

Theorem 5.2.12 *Let π be a conditional plan for a domain $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{R}, \mathcal{X}\rangle$, let $\mathcal{G} \subseteq \mathcal{S}$ be a goal, and let $\mathcal{H} \in \mathcal{L}(\mathcal{P})$ be an assumption. If $\tau(\pi, \mathcal{I}, \mathcal{H})$ does not contain the Fail node, and $\forall \langle \mathcal{B}, \langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}}\rangle\rangle \in leaf\big(\tau(\pi, \mathcal{I}, \mathcal{H})\big) :\ St(\mathcal{B}) \cup St(\mathcal{B}_{\bar{\mathcal{H}}}) \subseteq \mathcal{G}$, then π is a safe solution for $\mathcal{G}$ under $\mathcal{H}$.*

A detailed proof of the theorem can be found in the Appendix.

5.2.3 Symbolic annotated beliefs

Progressing annotated belief states by explicitly enumerating each state becomes soon unfeasible when beliefs contain large sets of states. To scale up on significant problems, we exploit symbolic techniques based on BDDs [Bry86] that allow efficiently manipulating sets of states; such techniques are indeed the key behind the effectiveness of approaches such as [BCR01a]. To leverage on BDD primitives, we group together states associated with the same formula, and represent annotated beliefs as sets of pairs $\langle B, \varphi \rangle$, where B is a set of states (a belief). Inside a *symbolic annotated belief*, we do not impose that two beliefs are disjoint. A state belonging to two pairs $\langle B_1, \varphi_1 \rangle$ and $\langle B_2, \varphi_2 \rangle$ is in fact associated to $\varphi_1 \vee \varphi_2$.

Symbolic annotated beliefs are progressed and handled by operating on their belief-formula pairs, based on the consideration that LTL formulæ can be represented in a disjunctive normal form:

$$\phi = \bigvee (\bigwedge \psi_i \wedge \bigwedge \neg \rho_i)$$

where ψ_i, ρ_i are either basic propositions, or formulæ whose top-level is either **U** or **X**. In particular, a disjunct t can be split into a purely propositional part $Prop(t)$ (a conjunction of literals), and a temporal part $Time(t)$ (a conjunction of possibly negated **U** and **X** formulæ):

$$\phi = \bigvee_{t \in Dnf(\phi)} (Prop(t) \wedge Time(t))$$

If no top-level **U**s are present in a disjunct t, $Prop(t)$ unambiguously identifies the states for which t is satisfiable, while $Time(t)$ determines which formula remains to be verified in the future. Thus, we can use this representation to identify for which portions of a belief B is an LTL formula ϕ satisfiable, and which LTL formula must they satisfy in the future, by unrolling the **U**s, and by partitioning the belief according to the propositional parts of the disjuncts of ϕ:

$$\{\langle B \cap Prop(t), Time(t) \rangle : t \in Dnf(Unroll(\phi))\}$$

Applying this idea, the initialization and progression of search nodes given in section 5.2.1 are represented as follows:

$$\begin{aligned} \mathcal{B}_{\mathcal{I}} &= \{\langle \mathcal{I} \cap Prop(t), Time(t) \rangle : \\ &\qquad t \in Dnf(Unroll(\mathcal{H})), \mathcal{I} \cap Prop(t) \neq \emptyset\} \\ \mathcal{E}(\mathcal{B}, o) &= \{\langle B \cap [[o]], \varphi \rangle : \langle B, \varphi \rangle \in \mathcal{B}, B \cap [[o]] \neq \emptyset\} \\ \mathcal{E}(\mathcal{B}, \alpha) &= \{\langle \alpha(B) \cap Prop(t), Time(t) \rangle : \\ &\quad \langle B, \varphi \rangle \in \mathcal{B}, t \in Dnf(X^{-1}(\varphi)), \alpha(B) \cap Prop(t) \neq \emptyset\} \end{aligned}$$

The indistinguishable sets can also be represented symbolically, and their initialization and progression follow the same ideas above; notice that also the pruning operation is represented symbolically:

$$\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle = \left\langle prune\left(\mathcal{B}^0_{\mathcal{H}}, \mathcal{B}^0_{\bar{\mathcal{H}}}\right), prune\left(\mathcal{B}^0_{\bar{\mathcal{H}}}, \mathcal{B}^0_{\mathcal{H}}\right) \right\rangle$$

where

$$\mathcal{B}^0_{\mathcal{H}} = \left\{\langle \mathcal{I} \cap Prop(t), Time(t) \rangle : t \in Dnf(Unroll(\mathcal{H})),\ \mathcal{I} \cap Prop(t) \neq \emptyset\right\}$$
$$\mathcal{B}^0_{\bar{\mathcal{H}}} = \left\{\langle \mathcal{I} \cap Prop(t), Time(t) \rangle : t \in Dnf(Unroll(\bar{\mathcal{H}})),\ \mathcal{I} \cap Prop(t) \neq \emptyset\right\}$$

$$prune(\mathcal{B}, \mathcal{B}') = \left\{ \langle prune_D(B, \mathcal{B}'), \varphi \rangle : \langle B, \varphi' \rangle \in \mathcal{B} \right\}$$
$$prune_D(B, \mathcal{B}') = B \cap \bigcup_{o \in \mathcal{X}(B) \cap \mathcal{X}(St(\mathcal{B}'))} [[o]]$$

The above approach is provably correct, and has been tested for a variety of scenarios, demonstrating the improvements in runtime behavior that come from the safety requirement, and showing that no significant overhead occurs at plan generation time (see [AB06]).

Chapter 6

History of the Deliverable

In this chapter, we summarize the way in which the activities described in this deliverable have evolved along the 4 years of the project.

6.1 1st Year

Our first year of the project was devoted to surveying the state of the art in planning, and assessing whether the techniques developed so far, and in particular those introduced by the project partners, could be exploited to tackle our reference problem in automated software development, i.e. automated composition of web services.

Our analysis concluded that it was necessary to consider planning techniques capable to deal with nondeterminism. While the first attempts at solving nondeterministic planning problems can be found in works such as [SW98, CKL94], the technologies adopted at the time were not mature enough to scale up to reasonable extent. Only the techniques based on model-checking, first introduced by ITC-irst in 1998 [CRT98], seemed mature and powerful enough to be credible candidates for the task.

Prior to the beginning of our project, such techniques were used to solve a variety of different problems (e.g. "weak and strong planning' [CRT98] and "strong-cyclic planning" [DTV99] for fully observable planning domains, contingent planning [BCRT01], conformant planning [BCR01b], planning for temporal and extended goals [PT01b, DLPT02]). However, such works did not consider at all the relationship between plan construction and execution, and were not designed around a unique framework. As our starting step, we proposed a unifying framework [BCPT03], and relied on that to extend our approaches to satisfy the goals of the project.

Also, in year one, we used the existing tools to perform the first web service composition tests by means of "ground-level" encoding. This gave us hints on how to optimize the search mechanisms in our algorithms to improve performance.

6.2 2nd Year

In the second year of the project, we achieved the full integration of our planning techniques (in particular, those focusing on fully observable domains) within a toolset that provided end-to-end automated service composition, as shown in [PBB$^+$04]. This integration also involves a pre-analysis of the services in order to compile away the indeterminacy due to the limited sensing available.

In parallel, we devised an approach to solve planning problems with temporal goals in the presence of limited sensing - a setting general enough to encompass most service composition problems without a need for computationally expensive analysis and recasting into simpler frameworks. Some results, presented in [BP04], make evident scalability issues and hint at the need for more specific approaches.

Concerning the integration of planning, monitoring and execution, we started activities in two main directions: reactive planning, where partial plans are produced and extended step by step on the basis of monitoring results, and assumption-based planning, where complete plans are produced under the hypothesis that the environment will follow certain "assumed" behaviors. In the first case, we tackled the problem of preventing long-term execution loops, by insuring that every partial plan obeys an appropriately defined "progressiveness" property; these results appeared in [BCT04]. In the second case, we tackled the problem of "safety" of the plan, i.e. of guaranteeing that no run-time state ambiguity may lead to execution failures or to miss the chosen goal. We first considered this problem in the case of simple, propositional assumptions over the domain behavior [AB04].

6.3 3rd Year

In year three of the project, we optimized our search techniques used in the "ground-level" approach; this made it possible to deliver the results in [PTB05], noticeably improved w.r.t. those of [PBB$^+$04].

Once achieved substantial maturity of the foundational search algorithms, then, our focus shifted more and more towards applications and experimenting, as witnessed in Deliverables D4.3 and D7.3.

6.4 4th Year

In year four of the project, we achieved two important results that extend those achieved in the previous years, and therefore enlarge the field of applicability of the associated techniques.

First, we have been able to express sets of goal preferences, and to deal with them effectively in a partially observable setting, therefore lifting the approach of [DLPT02] and responding to some crucial issues which arose in [BP04]. This work is presented in [SPT06].

Second, we adopted a powerful temporal logics to describe assumptions in terms of execution paths, and we showed how it is possible to generate appropriately monitorable plans under such premises [AB06].

Bibliography

[AB04] A. Albore and P. Bertoli. Generating Safe Assumption-Based Plans for Partially Observable, Nondeterministic Domains. In *Proceedings of AAAI-04*, 2004.

[AB06] A. Albore and P. Bertoli. Safe LTL Assumption-Based Planning. In *Proceedings of ICAPS'06*, 2006.

[ABE00] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In S. Graf and M. Schwartzbach, editors, *Proceedings of the Sixth Conference Tools and Algorithms for the Construction and Analysis of Systems, Berlin, Germany*, volume 1785 of *LNCS*, pages 411–425. Springer-Verlag, 2000.

[BBS95] A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1–2):81–138, 1995.

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In R. Cleaveland, editor, *Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *LNCS*, pages 193–207. Springer-Verlag, 1999.

[BCM$^+$92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BCPT03] P. Bertoli, A. Cimatti, M. Pistore, and P. Traverso. A Framework for Planning with Extended Goals and Partial Observability. In *Proc. of ICAPS03*, June 2003.

[BCR01a] P. Bertoli, A. Cimatti, and M. Roveri. Conditional Planning under Partial Observability as Heuristic-Symbolic Search in Belief Space. In *Proc. of ECP'01*, 2001.

[BCR01b] P. Bertoli, A. Cimatti, and M. Roveri. Heuristic search + symbolic model checking = efficient conformant planning. In B. Nebel, editor, *Proceedings* *of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001*, pages 467–472. Morgan Kaufmann Publishers, August 2001.

[BCRT01] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in nondeterministic domains under partial observability via symbolic model checking. In B. Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001*, pages 473–478. Morgan Kaufmann Publishers, August 2001.

[BCRT06] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Strong Planning under Partial Observability. *Artificial Intelligence*, 170:337–384, 2006.

[BCT04] P. Bertoli, A. Cimatti, and P. Traverso. Interleaving Execution and Planning for Nondeterministic, Partially Observable Domains. In *Proceedings of ECAI-04*, 2004.

[BDH96] C. Boutilier, T. Dean, and S. Hanks. Planning under uncertainty: Structural assumptions and computational leverage. In M. Ghallab and A. Milani, editors, *New Directions in AI Planning*, pages 157–172. IOS Press (Amsterdam), 1996.

[BG00] B. Bonet and H. Geffner. Planning with incomplete information as heuristic search in belief space. In S. Chien, S. Kambhampati, and C.A. Knoblock, editors, *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, pages 52–61. AAAI Press, April 2000.

[BP04] P. Bertoli and M. Pistore. Planning with Extended Goals and Partial Observability. In *Proc. of ICAPS04*, June 2004.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, Orlando, Florida, June 1990. ACM/IEEE, IEEE Computer Society Press.

[Bry86] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CDGV02] D. Calvanese, G. De Giacomo, and M. Y. Vardi. Reasoning about actions and planning in LTL action theories. In *Proc. of the 8th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2002)*, pages 593–602, 2002.

[CGGT97] A. Cimatti, F. Giunchiglia, E. Giunchiglia, and P. Traverso. Planning via Model Checking: A Decision Procedure for AR. In *Proc. of 3rd European Conference on Planning, ECP'99,*, pages 130–142, 1997.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[CKL94] A. Cassandra, L. Kaelbling, and M. Littman. Acting optimally in partially observable stochastic domains. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 1023–1028. AAAI Press, August 1994.

[CPRT03] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. *Artificial Intelligence*, 147(1-2):35–84, July 2003.

[CRB03] A. Cimatti, M. Roveri, and P. Bertoli. Conformant Planning via Symbolic Model Checking and Heuristic Search. *Artificial Intelligence*, 2003.

[CRT98] A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based Generation of Universal Plans in Non-Deterministic Domains. Technical Report 9801-10, ITC-IRST, Trento, Italy, January 1998. Submitted to the Fifteenth National Conference on Artificial Intelligence July 26-30, 1998, Madison, Wisconsin.

[CW96] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[DLPT02] U. Dal Lago, M. Pistore, and P. Traverso. Planning with a Language for Extended Goals. In *Proc. AAAI'02*, 2002.

[DTV99] M. Daniele, P. Traverso, and M. Y. Vardi. Strong Cyclic Planning Revisited. In *ECP*, pages 35–48, 1999.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter 16, pages 995–1072. Elsevier, 1990.

[FHMV95] R. Fagin, J.Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about knowledge*. MIT Press, 1995.

[GT00] F. Giunchiglia and P. Traverso. Planning as Model Checking. In *Proc. of 5th European Conference on Planning, ECP'99*, volume 1809 of *Lecture Notes in Computer Science*, pages 1–20, 2000.

[HI94] R. Hähnle and O. Ibens. Improving temporal logic tableaux using integer constraints. In *ICTL '94: Proceedings of the First International Conference on Temporal Logic*, pages 535–539, London, UK, 1994. Springer-Verlag.

[HZ01] E. A. Hansen and S. Zilberstein. LAO * : A Heuristic Search Algorithm that finds Solutions with Loops. *Artificial Intelligence*, 129(1-2):35–62, 2001.

[KBSD97] F. Kabanza, M. Barbeau, and R. St-Denis. Planning Control Rules for Reactive Agents. *Artificial Intelligence*, 95(1):67–113, 1997.

[Koe01] S. Koenig. Minimax Real-Time Heuristic Search. *Artificial Intelligence*, 129(1):165–197, 2001.

[KS97] S. Koenig and Y. Smirnov. Sensor-Based Planning with the Freespace Assumption. In *Proc. of ICRA'97*, 1997.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publ., 1993.

[MGR98] M. Di Manzo, E. Giunchiglia, and S. Ruffino. Planning via Model Checking in Deterministic Domains: Preliminary Report. In *Proceedings of AIMSA'98*, pages 221–229, 1998.

[MM73] A. Martelli and U. Montanari. Additive AND/OR Graphs. In *Proc. of IJCAI-73*, pages 1–11, 1973.

[MNPW98] N. Muscettola, P. P. Nayak, B. Pell, and B. C. Williams. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5–47, 1998.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[MW98] K. L. Myers and D. E. Wilkins. Reasoning about locations in theory and practice. *Computational Intelligence*, 14(2):151–187, 1998.

[PBB$^+$04] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso. Planning and monitoring web service composition. In *Proc. 11th Int. Conf. on Artificial Intelligence: Methodology, Systems, Architectures*, 2004.

[PT01a] M. Pistore and P. Traverso. Planning as model checking for extended goals in non-deterministic domains. In B. Nebel, editor, *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 479–486. Morgan Kaufmann Publisher, August 2001.

[PT01b] M. Pistore and P. Traverso. Planning as Model Checking for Extended Goals in Non-deterministic Domains. In *Proc. IJCAI'01*. AAAI Press, 2001.

[PTB05] M. Pistore, P. Traverso, and P. Bertoli. Automated Composition of Web Services by Planning in Asynchronous Domains. In *Proc. ICAPS'05*, 2005.

[Rin04] J. Rintanen. Research on Conditional Planning with Partial Observability: the Jussi-POP / BBSP planning system. Webpage: http://www.informatik.uni-freiburg.de/ rintanen/planning.html, 2004.

[Sch87] M. J. Schoppers. Universal plans for Reactive Robots in Unpredictable Environments. In *Proc. of the 10th International Joint Conference on Artificial Intelligence*, pages 1039–1046, 1987.

[Som97] F. Somenzi. CUDD: CU Decision Diagram package — release 2.1.2. Department of Electrical and Computer Engineering — University of Colorado at Boulder, April 1997.

[SPT06] D. Shaparau, M. Pistore, and P. Traverso. Contingent Planning with Goal Preferences. In *Proceedings of AAAI'06*, 2006.

[SW98] D. E. Smith and D. S. Weld. Conformant graphplan. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 889–896, Menlo Park, July 26–30 1998. AAAI Press.

Appendix A

Appendix: Proofs

A.1 Weak and Strong Planning under Full Observability

A.1.1 Formal Properties of the Weak Planning Algorithm

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

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

Definition A.1.1 (Trace) *A* trace *from state s to state s' is a sequence of states $s_0, s_1, \ldots, s_i$ with $s_0 = s$, $s_i = s'$ such that for all $j = 0, \ldots, i-1$ there is some action a_j such that $\mathcal{R}(s_j, a_j, s_{j+1})$. Let S' be a set of state; then a trace from s to S' is a trace from s to any state $s' \in S'$. We say that trace $s = s_0, s_1, \ldots, s_i = s'$ has length i. We say that trace $s = s_0, s_1, \ldots, s_i = s'$ is compatible with state-action table π if for all $j = 0, \ldots, i-1$ there is some $\langle s_j,\ a_j \rangle \in \pi$ such that $\mathcal{R}(s_j, a_j, s_{j+1})$.*

Definition A.1.2 (Weak Distance) *The* weak distance *$\mathcal{WD}ist(s, \mathcal{G})$ of a state s from goal $\mathcal{G}$ is the smallest integer i such that there is a trace of length i from s to $\mathcal{G}$. If no trace from s to $\mathcal{G}$ exists, then we define $\mathcal{WD}ist(s, \mathcal{G}) = \infty$.*

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

We now formalize the idea that states at a growing distance from goal $\mathcal{G}$ are added to state-action table *SA* at each iteration of the algorithm (Proposition A.1.5 below). We start with some notations and auxiliary lemmas.

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

Lemma A.1.3 $\textsc{StatesOf}(\pi_i) = \bigcup_{j=1..i} S_j$ *and* $\mathcal{G} \cup \textsc{StatesOf}(\pi_i) = \bigcup_{j=0..i} S_j$.

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

Lemma A.1.4 *The sets of states* S_i *are disjoint, i.e.,* $S_i \cap S_j = \emptyset$ *if* $i \neq j$.

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

Proposition A.1.5 *Let* s *be a state in the planing domain. Then* $s \in S_i$ *iff* $\mathcal{WD}ist(s, \mathcal{G}) = i$.

Proof: By induction on i.

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

Case $i > 0$**.** By the inductive hypothesis, for all $j < i$ we have $s' \in S_j$ iff $\mathcal{WD}ist(s', \mathcal{G}) = j$. So, by Lemma A.1.3, $s' \in \mathcal{G} \cup \textsc{StatesOf}(\pi_{i-1})$ iff $s \in \bigcup_{j=0..i-1} S_j$ iff $\mathcal{WD}ist(s, \mathcal{G}) < i$.

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

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

and

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

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

Assume now $\mathcal{WD}ist(s, \mathcal{G}) = i$. Then, by definition of weak distance, there is some a and some $s' \in \textsc{Exec}(s, a)$ such that $\mathcal{WD}ist(s', \mathcal{G}) = i - 1$. Then $\langle s, a \rangle \in \textsc{WeakPreImage}(S_{i-1})$ and, since $S_{i-1} \subseteq \mathcal{G} \cup \textsc{StatesOf}(\pi_{i-1})$ by Lemma A.1.3, we obtain

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

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

$$\begin{aligned}\langle s,\, a\rangle \in \pi_i = \text{PRUNESTATES}(&\text{WEAKPREIMAGE}(\mathcal{G} \cup \text{STATESOF}(\pi_{i-1})),\\ &\mathcal{G} \cup \text{STATESOF}(\pi_{i-1})).\end{aligned}$$

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

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

Proposition A.1.6 *Let $i > 0$. If $s \in S_i$ and $\langle s,\, a\rangle \in \pi_i$, then:*

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

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

$$\langle s,\, a\rangle \in \text{WEAKPREIMAGE}(\mathcal{G} \cup \text{STATESOF}(\pi_{i-1}))$$

and that

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

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

$$\begin{aligned}\langle s,\, a\rangle \in \text{PRUNESTATES}(&\text{WEAKPREIMAGE}(\mathcal{G} \cup \text{STATESOF}(\pi_j)),\\ &\mathcal{G} \cup \text{STATESOF}(\pi_j)),\end{aligned}$$

that is $\langle s,\, a\rangle \in \pi_{j+1}$. This implies $s \in S_{j+1}$ and, since $s \in S_i$ by hypothesis, Lemma A.1.4 forces $i = j + 1$. This concludes the proof of the first item.

For the second item, assume by contradiction $\text{EXEC}(s, a) \cap S_{i-1} = \emptyset$. We have just proved that $\text{EXEC}(s, a) \cap S_j = \emptyset$ for $j < i - 1$, so, by Lemma A.1.3 we conclude $\text{EXEC}(s, a) \cap (\mathcal{G} \cup \text{STATESOF}(\pi_{i-1})) = \emptyset$. Therefore, $\langle s,\, a\rangle \notin \text{WEAKPREIMAGE}(\mathcal{G} \cup$

$\text{STATESOF}(\pi_{i-1}))$ and hence $\langle s, a\rangle \notin \pi_i$, which is an absurd. By contradiction, we conclude that $\text{EXEC}(s,a) \cap S_{i-1} \neq \emptyset$. ■

Propositions A.1.5 and A.1.6 are the basic ingredients for proving the correctness of function WEAKPLAN. The correctness depends on the fact that all the states from which the goal may be reached have a finite weak distance from the goal, and hence are eventually added to the state-action table built by the algorithm (Proposition A.1.5); and on the fact that any action that appears in the state-action table has some outcome that lead to a shorter distance from the goal (Proposition A.1.6), so that a path to the goal can be obtained by following the outcomes of the actions.

Theorem 4.1.7 *[Correctness] Let* $\pi \doteq \text{WEAKPLAN}(\mathcal{I},\mathcal{G}) \neq$ *Fail. Then* π *is a weak solution of the planning problem* $P = \langle \mathcal{D},\mathcal{I},\mathcal{G}\rangle$.
If $\text{WEAKPLAN}(\mathcal{I},\mathcal{G}) =$ *Fail, instead, then there is no weak solution for planning problem* P.

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

Since $\pi \neq$ *Fail*, there must be some $i \geq 0$ such that $\pi = \pi_i$ and $\mathcal{I} \subseteq \mathcal{G} \cup \text{STATESOF}(\pi_i)$. Let $s \in \mathcal{I}$: we have to show that there exists some path in the execution structure from s to $\mathcal{G}$. We know, by Lemma A.1.3, that $s \in \mathcal{G} \cup \text{STATESOF}(\pi_i)$ implies $s \in S_j$ for some $j \leq i$. If $s \in S_0 = \mathcal{G}$ then there is the trivial path. Otherwise, $s \in \text{STATESOF}(\pi_i)$ and, by definition of determinization, there is some $\langle s, a\rangle \in \pi_d$ such that $\langle s, a\rangle \in \pi_i$. Since $s \in S_j$, we also have $\langle s, a\rangle \in \pi_j$. By Proposition A.1.6 we know that there is some $s' \in \text{EXEC}(s,a)$ such that $s' \in S_{j-1}$; hence, (s, s') is a transition of the execution structure. By induction on j, it is easy to conclude that there is an execution path of length j from s to $\mathcal{G}$.

Assume now $\pi =$ *Fail*. Then there is some i such that $\pi_i = \pi_j$ for any $j > i$, and there exists some $s \in \mathcal{I}$ such that $s \notin \text{STATESOF}(\pi_i) \cup \mathcal{G}$. Now, assume by contradiction that there is a deterministic state-action table π_d and a finite execution path from s to $\mathcal{G}$ in the execution structure for π_d. Then, clearly, there is some trace in the domain that corresponds to that execution path, and hence $\mathcal{WD}ist(s,\mathcal{G})$ is finite. Let $k = \mathcal{WD}ist(s,\mathcal{G})$: then $s \in S_k$. If $k \leq i$ then we would have $s \in \mathcal{G} \cup \text{STATESOF}(\pi_i)$ by Lemma A.1.3, and this contradicts the hypotheses. If $k > i$, then $s \in S_k$; this is absurd since $\pi_k = \pi_i = \pi_{k-1}$ and hence $S_k = \emptyset$. So, we conclude that no weak solution π_d can exist if $\pi =$ *Fail*. ■

Algorithm WEAKPLAN always terminates: indeed, at any iteration of the while loop, either the cardinality of set *SA* strictly grows, or the loop ends due to condition "*OldSA* $\neq$ *SA*" in the guard of the while loop. Also, the cardinality of *SA* cannot grow without a bound, as there are only finitely many valid state-action pairs.

Theorem 4.1.8 *[Termination] Function* WEAKPLAN *always terminates.*

Proof: The only possible cause of non-termination is the while loop at lines 4-9. We can see (line 8) that during an iteration of the loop the cardinality of set *SA* cannot reduce.

Moreover, if the value of variable SA does not change during an iteration, then the loop terminates (condition $\mathit{OldSA} \neq \mathit{SA}$ in the guard of the loop). Since set $\mathit{SA} \subseteq \mathcal{S} \times \mathcal{A}$ and $\mathcal{S} \times \mathcal{A}$ is finite, it is not possible for the cardinality of set SA to strictly grow indefinitely, so the loop will eventually terminate. ■

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

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

Theorem A.1.7 (Optimality) *Let $\pi \doteq \text{WEAKPLAN}(\mathcal{I}, \mathcal{G}) \neq \mathit{Fail}$. Then plan π is optimal with respect to the weak distance: namely, for each $s \in \mathcal{I}$, in the execution structure for π there exists an execution path from s to $\mathcal{G}$ of length $\mathcal{WD}ist(s, \mathcal{G})$.*

Proof: In the proof of the correctness theorem we have already proved that if $s \in \mathcal{I}$ and $s \in S_j$ then there is a path of length j in the execution structure for π. By Proposition A.1.5, $j = \mathcal{WD}ist(s, \mathcal{G})$. ■

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

A.1.2 Formal Properties of the Strong Planning Algorithm

The formal results for the strong planning algorithm can be easily obtained by adapting those for the weak planning algorithm presented in Section A.1.1. Here we only discuss the most relevant differences.

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

Definition A.1.8 (Complete Set of Traces) *A* complete set of traces *from s to $\mathcal{G}$ is a set T of traces from s to states in $\mathcal{G}$ such that, whenever $s = s_0, s_1, \ldots, s_i$ is in the set T and $j = 0, \ldots, i-1$, then there is some action a such that $\mathcal{R}(s_j, a, s_{j+1})$ and, whenever $\mathcal{R}(s_j, a, s'_{j+1})$, then there is some trace in T that extends $s = s_0, s_1, \ldots, s_j, s'_{j+1}$.*

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

Definition A.1.9 (Strong Distance) *The* strong distance $\mathcal{SD}ist(s, \mathcal{G})$ *of a state s from a goal $\mathcal{G}$ is the smallest integer i such that there is some complete set of traces T from s to $\mathcal{G}$ and i is the length of the longest trace in T. If no complete set of traces from s to $\mathcal{G}$ exists, then we define $\mathcal{SD}ist(s, \mathcal{G}) = \infty$.*

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

Proposition A.1.10 *Let s be a state in the planning domain. Then $s \in S_i$ iff $\mathcal{SD}ist(s, \mathcal{G}) = i$.*

Proposition A.1.6 has also to be adapted to take into account the fact that STRONGPREIMAGE replaces WEAKPREIMAGE in the algorithm.

Proposition A.1.11 *Let $i > 0$. If $s \in S_i$ and $\langle s, a \rangle \in \pi_i$, then:*

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

The main results on the strong planning algorithm follow.

Theorem A.1.12 (Correctness) *Let $\pi \doteq \textsc{StrongPlan}(\mathcal{I}, \mathcal{G}) \neq$ Fail. Then π is a strong solution of the planning problem $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$.*
If $\textsc{StrongPlan}(\mathcal{I}, \mathcal{G}) =$ Fail, instead, then there is no strong solution for planning problem P.

Theorem A.1.13 (Termination) *Function* STRONGPLAN *always terminates.*

Theorem A.1.14 (Optimality) *Let $\pi \doteq \textsc{StrongPlan}(\mathcal{I}, \mathcal{G}) \neq$ Fail. Then plan π is optimal with respect to the strong distance: namely, for each $s \in \mathcal{I}$, in the execution structure for π all the execution paths from s to $\mathcal{G}$ have a length smaller of equal to $\mathcal{SD}ist(s, \mathcal{G})$.*

A.2 Strong Cyclic Planning under Full Observability

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

We start by defining when a state-action table π is *SC-valid* (or *strong-cyclical* valid) for a set of states G. Intuitively, if a state-action table *SA* is SC-valid for G, then it is a strong cyclic solution of the planning problem of reaching G from any of the states in *SA*. Informally, a SC-valid state-action table should guarantee that the execution stops once the goal is reached: there is no reason to continue the execution in a goal state. Also, the execution of a SC-valid plan should never lead to states where the plan is undefined (with the exception of the goal states). Finally, we require that any action in the plan may lead to some progress in reaching the goal.

Definition A.2.1 (SC-valid state-action table) *State-action table π is* SC-valid *for G if:*

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

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

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

Proposition A.2.2 *Let π be a SC-valid plan for $\mathcal{G}$ and let $\mathcal{I} \subseteq \mathcal{G} \cup \textsc{StatesOf}(\pi)$. Then π is a strong cyclic solution for the planning problem $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G}\rangle$.*

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

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

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

The main result that guarantees the correctness of algorithm STRONGCYCLICPLAN is the fact that the state-action table computed by the while loop at lines 4-7 is a SC-valid state-action tables. This result is formalized in Proposition A.2.6, and its proof relies on the following auxiliary lemmas.

Lemma A.2.3 *Let π be the value of variable SA after the while loop at lines 4-7 in function* STRONGCYCLICPLAN(I, G).

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

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

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

For the second item, it is sufficient to observe that

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

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

Lemma A.2.4 *Let π be a generic state-action table and let*

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

Then:

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

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

Now we prove the other two properties.

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

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

(a) $s \in S_i$ iff the shortest trace from s to G that is compatible with π has length i: this is analogous to Proposition A.1.5.

(b) If $i \neq j$, then $S_i \cap S_j = \emptyset$: this is analogous to Lemma A.1.4.

(c) If $s \in S_i$ and $\langle s, a\rangle \in \pi_i$ then there is some $s' \in \text{EXEC}(s, a)$ such that $s' \in S_{i-1}$ and there is no $s' \in \text{EXEC}(s, a)$ such that $s' \in S_j$ for $j < i - 1$: this is analogous to Proposition A.1.6.

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

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

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

Lemma A.2.5 *Let π be the value of variable SA after the while loop at lines 4-7 in function* STRONGCYCLICPLAN(I, G)*. Moreover, let*

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

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

Proof: By Lemma A.2.3(2), if $s \in \text{STATESOF}(\pi)$ then there is a trace from s to G that is compatible with π. Let i be the length of the shortest trace from s to G that is compatible with π. By using the notations introduced in the proof of Lemma A.2.4, and by exploiting property (a) in that prove, we deduce that $s \in S_i$. Now, if $i = 0$ then $s \in S_0 = G$; otherwise $s \in \bigcup_{i>0} S_i = \text{STATESOF}(\pi')$. Therefore $\text{STATESOF}(\pi) \subseteq \text{STATESOF}(\pi') \cup G$, and hence $\text{STATESOF}(\pi) \setminus G \subseteq \text{STATESOF}(\pi')$.

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

Proposition A.2.6 *Let π be the value of variable SA after the while loop at lines 4-7 in function* $\textsc{StrongCyclicPlan}(I, G)$. *Moreover, let*

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

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

Proof: By Lemma A.2.4(2), $\textsc{StatesOf}(\pi') \cap G = \emptyset$.

Let $\langle s, a\rangle \in \pi'$ and $s' \in \textsc{Exec}(s, a)$. We prove that $s' \in G \cup \textsc{StatesOf}(\pi')$. By Lemma A.2.4(1), $\langle s, a\rangle \in \pi$. Then, by Lemma A.2.3(1) $s' \in G \cup \textsc{StatesOf}(\pi)$. Hence $s' \in G \cup \textsc{StatesOf}(\pi')$ by Lemma A.2.5.

Finally, by Lemma A.2.4(3), for any determinization π'_d of π' and for any state $s \in \textsc{StatesOf}(\pi')$ there is a trace from s to G compatible with π'_d.

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

By combining Propositions A.2.2 and A.2.6 it is easy to prove that, if function $\textsc{StrongCyclicPlan}$ returns a state-action table $\pi \neq$ *Fail*, then π is a strong cyclic solution. The converse proof, namely that a state-action table is returned whenever a strong cyclic solution exists, relies on the fact that function $\textsc{StrongCyclicPlan}$ starts the elimination loop on the universal state-action table: starting from *UnivSA*, it is impossible to prune away states for which the goal is reachable in a strong cyclic way. This fact is proved in the following lemma.

Lemma A.2.7 *Let π be a state-action table that is SC-valid for G and let π_p be the value of variable SA in function* $\textsc{StrongCyclicPlan}(I, G)$ *at the end of the while loop at lines 4-7. Then $\pi \subseteq \pi_p$.*

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

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

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

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

Similarly, by the monotonicity of function $\textsc{PruneUnconnected}$ we deduce

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

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

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

this concludes the proof of the inductive step. ■

Theorem 4.2.2 *[Correctness] Let* $\pi \doteq \text{StrongCyclicPlan}(\mathcal{I}, \mathcal{G}) \neq$ *Fail. Then* π *is a strong cyclic solution of the planning problem* $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$.
If $\text{StrongCyclicPlan}(\mathcal{I}, \mathcal{G}) =$ *Fail, instead, then there is no strong cyclic solution for planning problem* P.

Proof: Assume $\pi \neq$ *Fail*; we show that π is a strong cyclic solution for P. First of all, $\mathcal{I} \subseteq \mathcal{G} \cup \text{StatesOf}(\pi)$, otherwise function StrongCyclicPlan returns *Fail*. Also, by Proposition A.2.6, state-action table π is SC-valid for $\mathcal{G}$. Then, by Proposition A.2.2, π is a strong cyclic solution. This concludes the proof that any deterministic plan in $\pi \neq$ *Fail* is a strong cyclic solution for planning problem P.

Now we show that, if $\pi =$ *Fail* then there is no plan that is a strong cyclic solution for the planning problem. By contradiction, assume π is a solution. We can assume, without loss of generality, that all the states in $\text{StatesOf}(\pi)$ are reachable from $\mathcal{I}$, and that no states in $\mathcal{G}$ appear in π. Indeed, if we restrict any solution π to these states we still have a valid solution. It is easy to check that π is a SC-valid state-action table for $\mathcal{G}$. So, by Lemma A.2.7, $\pi \subseteq \pi_p$ and hence $\text{StatesOf}(\pi) \subseteq \text{StatesOf}(\pi_p)$, where π_p is the value of variable *SA* in function $\text{StrongCyclicPlan}(I, G)$ at the end of the while loop at lines 4-7. Also $\mathcal{I} \subseteq \mathcal{G} \cup \text{StatesOf}(\pi)$, otherwise π would not be a solution for P. Summing up, $\mathcal{I} \subseteq \mathcal{G} \cup \text{StatesOf}(\pi) \subseteq \mathcal{G} \cup \text{StatesOf}(\pi_p)$. Therefore, condition on line 8 is satisfied and function $\text{StrongCyclicPlan}(\mathcal{I}, \mathcal{G})$ does not return *Fail*, which contradicts our hypothesis. ■

The strong cyclic planning algorithm terminates.

Theorem 4.2.3 *[Termination] Function* StrongCyclicPlan *always terminates.*

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

- function PruneOutgoing always terminates: it does not contain loops;
- function PruneUnconnected always terminates: the proof is similar to the one of Theorem 4.1.8;
- at any iteration, either the value of variable *SA* strictly decreases, or the loop terminates: indeed, functions PruneOutgoing and PruneUnconnected return a subset of the state-action table that receive as parameter.

In order to prove that function StrongCyclicPlanAux terminates, it remains to show that function RemoveNonProgress always terminates. The proof of this property is similar to the proof of Theorem 4.1.8. ■

A.3 Weak and Strong Planning under Partial Observability

We now prove that the algorithm for strong planning under partial observability is correct, complete and terminating, together with the necessary lemmas and introductory structural notions. We start by introducing some lemmas about belief executions; we then step to some additional structural size and distance notions (Sec. A.3). 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 (Sec. A.3). Then, we prove some invariants about the main loop of the algorithm (Sec. A.3), which are finally used in the proofs of termination, correctness and completeness (Sec. A.3).

Strong Planning and Belief Execution

Theorem A.3.1 *Let π be a plan applicable in belief state B. Then the following properties hold:*

- *if $\pi = a \circ \pi'$, then $FinalBels(B, \pi) \doteq FinalBels(Exec(a, B), \pi'))$;*
- *if $\pi = \textbf{if } o \textbf{ then } \pi_1 \textbf{ else } \pi_2$, then $FinalBels(B, \pi) \doteq FinalBels(B \cap \mathcal{X}^-(o)), \pi_1) \cup FinalBels(B \cap \mathcal{X}^-(\overline{o}), \pi_2)$*

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

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

Theorem 4.4.13 *Let $\langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a planning problem, and let π be applicable in $\mathcal{I}$. Then π is a strong solution to $\langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ iff $\bigcup$FinalBels$(\mathcal{I}, \pi) \subseteq \mathcal{G}$.*

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

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

  From the inductive hypothesis, $FinalStates($*Exec*$(a, B), \pi') = \bigcup$*FinalBels*$($*Exec*$(a, B), \pi)$.

  From Definition 4.4.5, we have that $FinalStates($*Exec*$(a, B), \pi') = FinalStates(B, a \circ \pi')$.

  From Lemma A.3.1, we have that *FinalBels*$($*Exec*$(a, B), \pi') =$ *FinalBels*$(B, a \circ \pi')$.
- if π is **if** o **then** π' **else** π'', then π' is applicable in $B \cap \mathcal{X}^-(o)$, and π'' is applicable in $B \cap \mathcal{X}^-(\overline{o})$.

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

  From Definition 4.4.5, we have that *FinalBels*$(B,$ **if** o **then** π' **else** $\pi'') =$ *FinalBels*$(B \cap \mathcal{X}^-(o), \pi') \cup$ *FinalBels*$(B \cap \mathcal{X}^-(\overline{o}), \pi'')$.

  From Lemma A.3.1, we have that

  $$FinalStates(B, \textbf{if } o \textbf{ then } \pi' \textbf{ else } \pi'') = FinalStates(B \cap \mathcal{X}^-(o), \pi') \cup FinalStates(B \cap \mathcal{X}^-(\overline{o}), \pi''$$

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 A.3.2 (Maximal distance from leaves) *The maximal distance from leaves of a node n of a belief execution tree T is defined as*

$$d_{MAX}(n, T) = max_{n' \in T, \textsc{Path}(n) \leq \textsc{Path}(n')} \textsc{Path}\ (n') - \textsc{Path}\ (n)$$

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

```
1  function NORMALIZE($\pi, B, \mathcal{G}$)
2    if ($\pi = \epsilon$) then
3      return $\pi$
4    else if ($B \subseteq \mathcal{G}$) then
5      return $\epsilon$
6    else if $B \in$ TRAVBELS($BET(\pi, B)$) then
7      pick $n_B \in$ TRAVERSED($BET(\pi, B)$) : BEL($n_B$) $= B$
8      return NORMALIZE($PlanOf(SubTree(BET(\pi, B), n_B))$)
9    else if $\pi = a \circ \pi'$ then
10     return $a \circ$ NORMALIZE($\pi', Exec(B, a), \mathcal{G}$)
11   else if ($\pi = \textbf{if } o \textbf{ then } \pi_1 \textbf{ else } \pi_2$) then
12     if $B \cap \mathcal{X}^-(\overline{o}) = \emptyset$ then
13       return NORMALIZE($\pi_1, B, \mathcal{G}$)
14     else if $B \cap \mathcal{X}^-(o) = \emptyset$ then
15       return NORMALIZE($\pi_2, B, \mathcal{G}$)
16     else
17       $\pi_1{}'$ := NORMALIZE($\pi_1, B \cap \mathcal{X}^-(o), \mathcal{G}$)
18       $\pi_2{}'$ := NORMALIZE($\pi_2, B \cap \mathcal{X}^-(\overline{o}), \mathcal{G}$)
19       return $\textbf{if } o \textbf{ then } \pi_1{}' \textbf{ else } \pi_2{}'$
20     fi
21   end
```

Figure A.1: The plan normalization routine.

Definition A.3.3 (Size of plan) *The size of a plan π, indicated with $\|\pi\|$, is the length of its longest maximal path, and can be defined as follows:*

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

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. A.1.

In the procedure, TRAVERSED(T) identifies the nodes of T other than its root, and TRAVBELS(T) the associated set of beliefs. Intuitively, NORMALIZE simplifies a plan π 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 π returns a plan π' 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 π' may traverse a subset of the beliefs that can be traversed by executing π.

Theorem A.3.4 *Let π be a strong solution for a problem $\langle \mathcal{D}, B, \mathcal{G} \rangle$. Then* $\textsc{Normalize}(\pi, B, \mathcal{G})$ *is a non-redundant solution for $\langle \mathcal{D}, B, \mathcal{G} \rangle$.*

Proof: We prove the theorem by induction on the size of the plan π.

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

In the induction step, we reason by cases, following the definition of the $\textsc{Normalize}$ routine:

1. When $B \subseteq \mathcal{G}$, the empty plan ϵ is returned, which is a non-redundant solution if $B \subseteq \mathcal{G}$;

2. (1) does not hold and $B \in \textsc{TravBels}(BET(\pi, B))$.
   Then, $SubTree(BET(\pi, B), n_B)$ is a belief execution tree whose root is associated to B. Since its leaves are leaves of $BET(\pi, B)$, $PlanOf(SubTree(BET(\pi, B), n_B))$ is a subplan of π which is a solution for $\langle \mathcal{D}, B, \mathcal{G} \rangle$. Also, $\|PlanOf(SubTree(BET(\pi, B), n_B))\| < \|\pi\|$; so, for the inductive hypothesis, the returned plan $\textsc{Normalize}(PlanOf(SubTree(BET(\pi, B), n_B)), B, \mathcal{G})$ is a non-redundant solution for the problem.

3. (1) and (2) do not hold, and $\pi = a \circ \pi'$.
   Then π' is a solution for $\langle \mathcal{D}, \mathit{Exec}(B, a), \mathcal{G} \rangle$, and since $\|\pi'\| < \|\pi\|$, by the inductive hypothesis, $\textsc{Normalize}(\pi', \mathit{Exec}(B, a), \mathcal{G})$ is a non-redundant solution for $\langle \mathcal{D}, \mathit{Exec}(B, a), \mathcal{G} \rangle$. Then since (2) does not hold, the returned plan $a \circ \textsc{Normalize}(\pi', \mathit{Exec}(B, a), \mathcal{G})$ is a non-redundant solution for $\langle \mathcal{D}, B, \mathcal{G} \rangle$.

4. (1),(2),(3) do not hold, $\pi = \textbf{if } o \textbf{ then } \pi_1 \textbf{ else } \pi_2$ and $B \cap \mathcal{X}^-(\overline{o}) = \emptyset$.
   Then π_1 is a solution for $\langle \mathcal{D}, B, \mathcal{G} \rangle$ (since $B \cap \mathcal{X}^-(o) = B$), and since $\|\pi_1\| < \|\pi\|$, by the induction step, the returned plan $\textsc{Normalize}(\pi_1, B, \mathcal{G})$ is a non-redundant solution for the same problem.

5. (1),(2),(3),(4) do not hold, $\pi = \textbf{if } o \textbf{ then } \pi_1 \textbf{ else } \pi_2$ and $B \cap \mathcal{X}^-(o) = \emptyset$.
   Then π_2 is a solution for $\langle \mathcal{D}, B, \mathcal{G} \rangle$ (since $B \cap \mathcal{X}^-(\overline{o}) = B$), and since $\|\pi_2\| < \|\pi\|$, by the induction step, the returned plan NORMALIZE$(\pi_2, B, \mathcal{G})$ is a non-redundant solution for the same problem.

6. (1),(2),(3),(4),(5) do not hold, and $\pi = \textbf{if } o \textbf{ then } \pi_1 \textbf{ else } \pi_2$.
   Then π_1 is a solution for $\langle \mathcal{D}, B \cap \mathcal{X}^-(o), \mathcal{G} \rangle$. By the inductive hypothesis, since $\|\pi_1\| < \|\pi\|$, π_{1N} = NORMALIZE$(\pi_1, B \cap \mathcal{X}^-(o), \mathcal{G})$ is a non-redundant solution for $\langle \mathcal{D}, B \cap \mathcal{X}^-(o), \mathcal{G} \rangle$. By a similar reasoning, π_{2N} = NORMALIZE$(\pi_2, B \cap \mathcal{X}^-(\overline{o}), \mathcal{G})$ is a non-redundant solution for $\langle \mathcal{D}, B \cap \mathcal{X}^-(\overline{o}), \mathcal{G} \rangle$. Then, since (2),(3),(4) do not hold, $\textbf{if } o \textbf{ then } \pi_{1N} \textbf{ else } \pi_{2N}$ is a non-redundant solution.

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 $ST_{[i]}$ the search tree at the beginning of the i-th iteration of the loop of the planning algorithm in Fig. 4.12, i.e. prior to the evaluation of the stop condition at line 6 of the algorithm.

Lemma A.3.5 *For every non-leaf node $n \in ST_{[i]}$, the following conditions hold:*

- *if $a \in \mathcal{A}$ is applicable on* BEL(n)*, then $ST_{[i]}$ contains a node $n' = \langle Exec(\text{BEL}(n), a), \text{PATH}(n) \circ a \rangle$, and an or-arc $\langle n, a, n' \rangle$.*
- *if $o \in \mathcal{O}$ is not predetermined on* BEL(n)*, then $ST_{[i]}$ contains two nodes $n_T = \langle \text{BEL}(n) \cap \mathcal{X}^-(o), \text{PATH}(n) \circ o \rangle$ and $n_F = \langle \text{BEL}(n) \cap \mathcal{X}^-(\overline{o}), \text{PATH}(n) \circ \overline{o} \rangle$, and an and-arc $\langle n, o, n_T, n_F \rangle$.*

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 $ST_{[0]}$, whose only node is a leaf. It is also easy to see that, it the theorem holds for $ST_{[i]}$, it holds for $ST_{[i+1]}$, since the internal nodes of $ST_{[i+1]}$ are those of $ST_{[i]}$, plus the one node n having been expanded during the i-th iteration by EXTENDTREE; but EXTENDTREE is defined in a way that directly maps into the property holding also for n.

Lemma A.3.6 *For every non-leaf node of $ST_{[i]}$, if* SONSYIELDSUCCESS *holds, the node is tagged $Success$; if* SONSYIELDFAILURE *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 SONSYIELDFAILURE(n) or SONSYIELDSUCCESS(n) hold; as such, node n obeys the requirement of the theorem. If neither SONSYIELDFAILURE(n) or SONSYIELDSUCCESS(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 SONSYIELDSUCCESS(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 PROPAGATESUCCESSONTREE routine recursively recomputes the tag of every ancestor for which SONSYIELDSUCCESS holds, proving the theorem in this case. Similarly for the case of SONSYIELDFAILURE(n).

Lemma A.3.7 *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 SONSYIELDSUCCESS 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 SONSYIELDFAILURE. This, together with Lemma A.3.6, proves the theorem.

Lemma A.3.8 *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 it to SONSYIELDFAILURE and SONSYIELDSUCCESS, 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 SONSYIELDSUCCESS and SONSYIELDFAILURE, 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;
- 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 $ST_{[i]}$).

Lemma A.3.9 *If a node $n \in ST_{[i]}$ is tagged $Success$, then* BUILDPLAN$(n, ST_{[i]})$ *is a solution for the problem $\langle \mathcal{D}, \text{BEL}(n), \mathcal{G} \rangle$.*

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 $\langle \mathcal{D}, \text{BEL}(n), \mathcal{G} \rangle$.
   If n has some successful or-son n_{succ} s.t. π' is a solution for $\langle \mathcal{D}, \text{BEL}(n_{succ}), \mathcal{G} \rangle$, since a is executable on $\text{BEL}(n)$ and $\text{BEL}(n_{succ}) = \textit{Exec}(\text{BEL}(n), a)$ (where $\langle n, a, n_{succ} \rangle \in ST_{[i]}$), the plan $a \circ \pi'$ is a solution for $\langle \mathcal{D}, \text{BEL}(n), \mathcal{G} \rangle$.

   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 π_T, π_F which are respectively solutions for $\langle \mathcal{D}, \text{BEL}(n_T), \mathcal{G} \rangle$ and $\langle \mathcal{D}, \text{BEL}(n_F), \mathcal{G} \rangle$. In this case, since $\text{BEL}(n_T) =$

$\text{BEL}(n) \cap \mathcal{X}^-(o)$ and $\text{BEL}(n_F) = \text{BEL}(n) \cap \mathcal{X}^-(\overline{o})$, **if** o **then** π_T **else** π_F is a solution for $\langle \mathcal{D}, \text{BEL}(n), \mathcal{G} \rangle$.

2. the plan returned by BUILDPLAN is a solution for $\langle \mathcal{D}, \text{BEL}(n), \mathcal{G} \rangle$. This is immediate to see, since it returns a plan constructed according to the ideas in point 1.

Lemma A.3.10 *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 SONSYIELDFAILURE (n') returns false, and as such n' is not tagged $Failure$ (see Lemma A.3.6).

Lemma A.3.11 *Let π be a non-redundant solution for $\langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$. A leaf of $BET(\pi, \mathcal{I})$ 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 $\mathcal{I} \subseteq \mathcal{G}$, the one non-redundant solution is ϵ, and the only leaf of $BET(\epsilon, \mathcal{I})$ is the success leaf of $ST_{[0]}$. If $\mathcal{I} \not\subseteq \mathcal{G}$, the only node of $ST_{[0]}$ is a leaf, and, for any plan π, is also the root of $BET(\pi, \mathcal{I})$, thus an ancestor to every leaf of $BET(\pi, \mathcal{I})$.

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 π and a leaf l of $BET(\pi, \mathcal{I})$. 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 $\mathcal{G}$ (it is a leaf of $BET(\pi, \mathcal{I})$), 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 π is non-redundant, n' cannot be associated to a belief entailed by $\mathcal{G}$, so it is not marked $Success$. Also, since π 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 $ST_{[i+1]}$.

Main properties

Theorem A.3.12 *Every execution of* PROPAGATESUCCESSONTREE *and* PROPAGATEFAILUREONTREE *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 trivial, since the only root node has no father. In the inductive step, the length is greater than 0; in this case, propagation recurs on FATHER(N), and since PATH *(*FATHER *(N))* $=$ PATH *(N)* $- 1$, by the inductive hypothesis, the recursive call on FATHER(N) terminates, proving the theorem.

Theorem A.3.13 *Every execution of* BUILDPLAN *on a successful node of* $ST_{[i]}$ *terminates.*

Proof: This originates from Lemma A.3.9: 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 4.4.20 *[Termination] Given a planning problem, the execution of the planning algorithm in Fig. 4.12 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 $\mathcal{N}$ be the number of acyclic belief paths for a domain $\mathcal{D}$.

Then, after at most $\mathcal{N}$ iterations, every acyclic path will be generated, and the frontier will contain $\mathcal{N}_M \leq \mathcal{N}$ nodes, each corresponding to a maximal acyclic belief path. Then, after further $\mathcal{N}_M$ iterations, each node will be removed from the frontier, and since 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 A.3.8), and as such the termination of the algorithm.

Theorem 4.4.21 *[Correctness] When the algorithm returns a plan π, then π is a strong solution for the problem.*

Proof: This is a direct derivation of Lemma A.3.9, instantiated for the root node of the tree.

Theorem 4.4.22 *[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 π exists, and consider its associated belief execution tree $T = BET(\mathcal{I}, \pi)$. According to Lemma A.3.11, 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 A.3.10, 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.

A.4 Conformant Planning

Here, we prove the formal properties of the forward-chaining conformant planning algorithm. Our proofs rely on first proving that two invariants hold, stating that the open and closed belief pools are always disjoint and make up the set of visited beliefs.

Definition A.4.1 (Set of Visited Belief States) *Let $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a planning problem. The set of visited belief states at the i-th iteration of the* while *loop of the* $\textsc{HeurConformantFwd}(\mathcal{I}, \mathcal{G})$*, written* $\textsc{BsVisited}(i)$ *is the set*

$$\begin{aligned} \textsc{BsVisited}(i) \quad \doteq \quad & \textsc{PoolBStates}(\mathit{Open}, i) \ \cup \\ & \textsc{PoolBStates}(\mathit{Closed}, i) \end{aligned}$$

where $\textsc{PoolBStates}(\mathit{Open}, i) = \{Bs : \langle Bs, \pi \rangle \in \mathit{Open}\}$*, and* $\textsc{PoolBStates}(\mathit{Closed}, i) = \{Bs : \langle Bs, \pi \rangle \in \mathit{Closed}\}$ *denote the set of belief states in Open and in Closed at the i-th iteration of the* while *loop.*

At each iteration i of the *while* loop of the algorithm the set of belief states contained in *Open* and *Closed* are disjoint.

Lemma A.4.2 *Let $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a planning problem. At the i-th iteration of the* while *loop of* $\textsc{HeurConformantFwd}(\mathcal{I}, \mathcal{G})$ *it holds*

$$\textsc{PoolBStates}(\mathit{Open}, i) \cap \textsc{PoolBStates}(\mathit{Closed}, i) = \emptyset$$

Proof: *The proof is by induction on i.*

Case $i = 0$.

Initially Open $= \{\langle \mathcal{I}, \epsilon \rangle\}$ and Closed $= \emptyset$, thus $\{\mathcal{I}\} \cap \emptyset = \emptyset$.

Case $i = i' + 1$.

By the induction hypothesis we have that

$$\textsc{PoolBStates}(\mathit{Open}, i') \cap \textsc{PoolBStates}(\mathit{Closed}, i') = \emptyset$$

At iteration i one element is removed by Open (line 6) and is added to Closed (line 7). At line 14, BsP pairs which are neither in Open nor in Closed are added to Open (at line 12 the call to $\textsc{PruneBsExpansion}$ *guarantee the removal of already visited belief states). Hence, the relation among the belief states in both pools is maintained.*

□

The set of visited belief states is monotonically increasing and there is a bound on the size of the set of visited belief states.

Lemma A.4.3 *Let $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a planning problem. At the i-th iteration of the* while *loop of* $\textsc{HeurConformantFwd}(\mathcal{I}, \mathcal{G})$*,*

1. $\textsc{BsVisited}(i) \subseteq \textsc{BsVisited}(i + 1)$

2. *There exists an index i such that* $\textsc{BsVisited}(i) = \textsc{BsVisited}(i+j)$ *for all* $j \geq 0$.

Proof:

1. *At each iteration i of the* while *loop, if Open is not empty, a BsP pair $\langle Bs, \pi \rangle$ is removed from Open and added to Closed (lines 9-10). No other removal may take place on Open or Closed. Thus if $Bs \in \textsc{BsVisited}(i)$, $Bs \in \textsc{BsVisited}(i+1)$. Moreover, for each BsP pair $\langle Bs, \alpha \rangle$ in BsPList a $\langle Bs, \pi \circ \alpha \rangle$ is added to Open at $i+1$. BsP pairs contained in BsPList are such that they have not been visited at previous iterations. Hence, $Bs \notin \textsc{BsVisited}(i)$. If BsPList $= \emptyset$ then the only change is that an element is removed from Open and added to Closed, thus $\textsc{BsVisited}(i) = \textsc{BsVisited}(i+1)$. Otherwise, a BsP pair $\langle Bs, \pi \circ \alpha \rangle$, such that $Bs \notin \textsc{BsVisited}(i)$, is added to Open, and thus there exists a belief state $Bs \notin \textsc{BsVisited}(i)$ and $Bs \in \textsc{BsVisited}(i+1)$.*

2. *The proof relies on the consideration that for all i, $\textsc{BsVisited}(i) \subseteq Pow(\mathcal{S})$. By (1) and by the finiteness of $\mathcal{S}$ (and thus of $Pow(\mathcal{S})$) it follows that i can grow at most to $\|Pow(\mathcal{S})\| + 1$.*

□

Lemma A.4.4 *Let $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a planning problem. Let i be the minimum integer such that* $\textsc{BsVisited}(i) = \textsc{BsVisited}(i+j)$ *for all* $j \geq 0$.

1. *For all $k \geq i$,*
$$\textsc{PoolBStates}(\textit{Open}, k) \supset \textsc{PoolBStates}(\textit{Open}, k+1)$$

2. *There exists an index $k \geq i$ such that*
$$\textsc{PoolBStates}(\textit{Open}, k) = \emptyset$$

Proof:

1. *Since* $\textsc{BsVisited}(i) = \textsc{BsVisited}(i+j)$ *for all $j \geq 0$ it is easy to show that no new belief states are added to Open. At each iteration $k \geq i$ an element $\langle Bs, \pi \rangle$ is removed from Open and added to Closed. Hence there is a belief state Bs such that $Bs \in \textsc{PoolBStates}(\textit{Open}, k)$ and $Bs \notin \textsc{PoolBStates}(\textit{Open}, k+1)$.*

2. *Since* $\textsc{BsVisited}(i) = \textsc{BsVisited}(i+j)$ *for all $j \geq 0$ then no new belief states are added to Open (See (1)). Let $l = \|\textsc{PoolBStates}(\textit{Open}, i)\|$ be the cardinality of the open pool at the i-th iteration. At each iteration $k \geq i$ an element is removed from Open and added to Closed. Thus, $\|\textsc{PoolBStates}(\textit{Open}, k+1)\| = \|\textsc{PoolBStates}(\textit{Open}, k)\| - 1$. Hence, at most $l+1$ iterations after i will be performed, and at $k = i + l + 1$ Open is empty.*

$\square$

Theorem A.4.5 *Let $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a planning problem. Procedure* HEURCONFORMANTFWD$(\mathcal{I}, \mathcal{G})$ *always terminates.*

Proof: *The only possible cause of non-termination of the* HEURCONFORMANTFWD *algorithm is the* while *loop at lines 5-17. After a bounded number of iterations the set of visited belief states cannot grow anymore (Lemma A.4.3(2)). Moreover, by Lemma A.4.3(2) after a further bounded number k of iterations,* POOLBSTATES$(Open, k)$ *will be empty, causing the algorithm to terminate.*

$\square$

Lemma A.4.6 *Let $\mathcal{D}$ be a planning domain; let $S_1 \subseteq S_2 \subseteq \mathcal{S}$ be two sets of states; let π be a plan applicable in S_2. Then Exec$(\pi, S_1) \subseteq$ Exec(π, S_2).*

Proof: *Trivial from the definition of Exec(π, S).*

$\square$

Lemma A.4.7 *Let $\mathcal{D}$ be a planning domain; let $\mathcal{I} \subseteq \mathcal{S}$ be a set of states; let $\langle Bs, \pi \rangle$ be a BsP pair such that $\perp \neq$ Exec$(\pi, \mathcal{I}) \subseteq Bs$; let BsExp $=$* FWDEXPANDBS(Bs). *Then for all $\langle Bs_i, \alpha \rangle \in$ BsExp we have $\pi' = \pi \circ \alpha$ is a solution for the conformant planning problem $\langle \mathcal{D}, \mathcal{I}, Bs_i \rangle$.*

Proof: *By definition each $\langle Bs_i, \alpha \rangle \in$ BsExp $=$* FWDEXPANDBS(Bs) *is such that*

$$\textit{Exec}(\alpha, Bs) \subseteq Bs_i$$

By hypothesis we have Exec$(\pi, \mathcal{I}) \subseteq Bs$, and applying α to both sides we have:

$$\textit{Exec}(\alpha, \textit{Exec}(\pi, \mathcal{I})) \subseteq \textit{Exec}(\alpha, Bs)$$

Since Exec$(\alpha,$ Exec$(\pi, \mathcal{I})) =$ Exec$(\pi \circ \alpha, \mathcal{I})$ it follows that Exec$(\pi \circ \alpha, \mathcal{I}) \subseteq Bs_i$. Hence, $\pi' = \pi \circ \alpha$ is a solution for the conformant planning problem $P = \langle \mathcal{D}, \mathcal{I}, Bs_i \rangle$.

$\square$

Lemma A.4.8 *Let $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a planning problem; Each BsP pair $\langle Bs, \pi \rangle \in$ Open at the i-th iteration of* HEURCONFORMANTFWD$((\mathcal{I}, \mathcal{G})$ *is such that Exec$(\pi, \mathcal{I}) \subseteq Bs$.*

Proof: *Trivial by Lemma A.4.7.*

□

Theorem A.4.9 *Let $P = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$ be a planning problem and let π be the value returned by procedure* $\textsc{HeurConformantFwd}(\mathcal{I}, \mathcal{G})$.

- *If $\pi \neq$ Fail, then π is a conformant solution for the planning problem P.*
- *If $\pi =$ Fail, instead, then there is no conformant solution for the planning problem P.*

Proof:

- *Let us assume $\pi \neq$ Fail. This means that there is an action α_i, a belief state Bs_i and a BsP pair $\langle Bs', \pi' \rangle$ such that $\pi = \pi' \circ \alpha_i$, $Bs_i \subseteq \mathcal{G}$, $\langle Bs', \pi' \rangle \in$ Open and*

  $$\langle Bs_i, \alpha_i \rangle \in \textsc{FwdExpandBs}(Bs')$$

  By Lemma A.4.8 we have that

  $$\textit{Exec}(\pi', \mathcal{I}) \subseteq Bs'$$

  By Lemma A.4.7 we have that

  $$\textit{Exec}(\pi' \circ \alpha_i, \mathcal{I}) \subseteq Bs_i$$

  Since $Bs_i \subseteq \mathcal{G}$ it follows that

  $$\textit{Exec}(\pi' \circ \alpha_i, \mathcal{I}) \subseteq \mathcal{G}$$

- *Let us assume $\pi = Fail$. Then there is an index i such that at the i-th iteration of the* while *loop $\textsc{PoolBStates}(\textit{Open}, i) = \emptyset$ and there is no $Bs \in \textsc{BsVisited}(i)$ such that $Bs \subseteq \mathcal{G}$. Let us assume for contradiction that there is a conformant solution π'. There is an index $j \leq i$ such that at the i-th iteration of the* while *loop there is an action α_j, a belief state Bs_j and a BsP pair $\langle Bs', \pi' \rangle$ such that $\pi = \pi' \circ \alpha_j$, $Bs_j \subseteq \mathcal{G}$, $\langle Bs', \pi' \rangle \in$ Open and*

  $$\langle Bs_j, \alpha_j \rangle \in \textsc{FwdExpandBs}(Bs')$$

  Thus, $Bs_j \in \textsc{BsVisited}(j) \subseteq \textsc{BsVisited}(i)$ and $Bs_j \subseteq \mathcal{G}$ which is absurd.

□

A.5 Assumption-based (re-)planning

Here, we show how the safety of a plan π can be established while progressing and pruning annotated beliefs, by checking goal entailment on the leaves of an associated execution (and-or) tree. We do so in three steps.

First, we define the tree structure associated to a plan π and an assumption $\mathcal{H}$ as a collection of nodes of the form $\langle \mathcal{B}, \langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle, p \rangle$. The accessibility relation between nodes is implicitly defined by the prefix relation among paths. This structure will be built following the progression and pruning rules described so far. Second, we show that for each node $\langle \mathcal{B}, \langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle, p \rangle$, $\mathcal{B}$ contains the final states of the traces along p for which $\mathcal{H}$ is satisfiable. Third, we prove that for each node $\langle \mathcal{B}, \langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle, p \rangle$, if t and t' are two indistinguishable traces along p for which $\mathcal{H}$ is respectively satisfiable and unsatisfiable, then $final(t') \in \mathcal{B}_{\bar{\mathcal{H}}}$.

Definition A.5.1 (Tree structure induced by a plan) *Let π be a conditional plan for a domain $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{R}, \mathcal{X} \rangle$, and let $\mathcal{H} \in \mathcal{L}(\mathcal{P})$. The tree structure induced by π from $\mathcal{I}$, denoted $\tau(\pi, \mathcal{I}, \mathcal{H})$, is defined as follows:*

- $\tau(\pi, \mathcal{I}, \mathcal{H}) = \tau_0(\pi, \mathcal{B}_{\mathcal{I}}, \langle \mathcal{B}^0_{\mathcal{H}}, \mathcal{B}^0_{\bar{\mathcal{H}}} \rangle, \epsilon)$*, where $\mathcal{B}_{\mathcal{I}}$ is defined as in* (5.1)*, and* $\langle \mathcal{B}^0_{\bar{\mathcal{H}}}, \mathcal{B}^0_{\mathcal{H}} \rangle$*as in* (5.4).
- $\tau_0(\epsilon, \mathcal{B}, \langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle, p) =$
  $\{\langle \mathcal{B}, \langle prune(\mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}}), prune(\mathcal{B}_{\bar{\mathcal{H}}}, \mathcal{B}^0_{\mathcal{H}}) \rangle, p \rangle\}$
- $\tau_0(\alpha \circ \pi, \mathcal{B}, \langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle, p) = \tau_0(\epsilon, \mathcal{B}, \langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle, p) \cup$
  $\tau_0(\pi, \mathcal{E}(\mathcal{B}, \alpha), \mathcal{E}(\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle, \alpha), p \circ \alpha)\}$
  if $\alpha(St(\mathcal{B})) \neq \emptyset \ \vee \ \alpha(St(\mathcal{B}_{\bar{\mathcal{H}}}) \neq \emptyset$
- $\tau_0(\alpha \circ \pi, \mathcal{B}, \langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle, p) =$
  $\{\langle Fail, \langle Fail, Fail \rangle, p \circ Fail(\alpha) \rangle\}$
  if $\alpha(St(\mathcal{B})) = \emptyset \ \vee \ \alpha(St(\mathcal{B}_{\bar{\mathcal{H}}}) = \emptyset$*. We call nodes whose annotated beliefs are* $Fail$ *"failure nodes".*
- $\tau_0(\text{if } o \text{ then } \pi_1 \text{ else } \pi_2, \mathcal{B}, \langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle, p) =$
  $\tau_0(\pi_1, \mathcal{E}(\mathcal{B}, o), \mathcal{E}(\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle, o), p \circ o) \cup$
  $\tau_0(\pi_2, \mathcal{E}(\mathcal{B}, \tilde{o}), \mathcal{E}(\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle, \tilde{o}), p \circ \tilde{o})$

We denote the leaves of $\tau(\pi, \mathcal{I}, \mathcal{H})$ *with* $leaf(\tau(\pi, \mathcal{I}, \mathcal{H}))$.

An annotated belief associated to a graph node contains the final states of the traces $Trs_{\mathcal{H}}(p, \mathcal{D})$ which traverse the path p associated to the node.

Lemma A.5.2 *Let $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{R}, \mathcal{X} \rangle$ be a planning domain, $\mathcal{H} \in \mathcal{L}(\mathcal{P})$, and π a conditional plan for $\mathcal{D}$. Then $\forall \langle \mathcal{B}, \langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle, p \rangle \in \tau(\pi, \mathcal{I}, \mathcal{H}) : St(\mathcal{B}) = \{final(t) : \forall t \in Trs_{\mathcal{H}}(p, \mathcal{D})\}$.*

Proof: The lemma is proved by induction on the length of p. We exploit the fact that tableaux rewriting rules like the $Unroll$ function preserve satisfiability of the rewritten formula [HI94]; thus, since $Unroll(\mathcal{H})|_s \equiv \mathcal{H}|_s$, we have that $Trs_{\mathcal{H}}(p, \mathcal{D}) = Trs_{Unroll(\mathcal{H})}(p, \mathcal{D})$.

The *base case* is trivial: the path p is empty, and the root of $\tau(\pi, \mathcal{I}, \mathcal{H})$ is built according to (5.1). Consequently its annotated belief contains every final state of $Trs_{\mathcal{H}}(\epsilon, \mathcal{D})$.

In the *induction step* we assume that for a node $\langle \mathcal{B}', \langle \mathcal{B}'_{\mathcal{H}}, \mathcal{B}'_{\bar{\mathcal{H}}} \rangle, p' \rangle$ the lemma holds, and we prove it for a direct descendant, i.e for a node $\langle \mathcal{B}, \langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle, p \rangle$ such that $\boldsymbol{p} = \boldsymbol{p}' + 1$ and p' is a prefix of p. Three cases arise:

- if $p = p' \circ \alpha$, the belief $St(\mathcal{B})$ is progressed following (5.3), so $St(\mathcal{B}) = \{\alpha(s) : s \in St(\mathcal{B}'), X^{-1}(\mathcal{H})|_{\alpha(s)} \neq \bot\}$. Given that $St(\mathcal{B}') = \{final(t')\}$, from the latter operation we have that $St(\mathcal{B}) = \{s \in final(t) : t = t' \circ [s, o, \alpha] : \forall t' \in Trs_{\mathcal{H}}(p', \mathcal{D}), o \in \mathcal{X}(s)\}$; this belief corresponds to the set of final states of the traces bound to path $p = p' \circ \alpha$ (cf. def. 5.2.3).
- If $p = p' \circ o$, from expansion rule (5.2), the belief $St(\mathcal{B}) = St(\mathcal{E}(\mathcal{B}', o)) = \{final(t')\} \cap [[o]]$, indicating all those states in $\{final(t) : \forall t = Trs_{\mathcal{D}}\big(o \circ p', \mathcal{I}\big) = Trs_{\mathcal{D}}\big(p, \mathcal{I}\big)\}$. From def. 5.2.3 we have that $t \in Trs_{\mathcal{H}}(p, \mathcal{D})$.
- If $p = p' \circ \tilde{o}$, the case is very similar to the latter. Again, from def. 5.2.3 we observe that the expansion of $\mathcal{B}'$ by $\tilde{o}$ corresponds to the final states of the traces $t \in Trs_{\mathcal{H}}(p, \mathcal{D}) \subseteq Trs_{\mathcal{D}}\big(p' \circ \tilde{o}, \mathcal{I}\big) = Trs_{\mathcal{D}}\big(p, \mathcal{I}\big)$.

Lemma A.5.3 *Let $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{R}, \mathcal{X} \rangle$ be a planning domain, a formula $\mathcal{H} \in \mathcal{L}(\mathcal{P})$, and let π be a conditional plan for $\mathcal{D}$. Let $\langle \mathcal{B}, \langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle, p \rangle$ be a node of $\tau(\pi, \mathcal{I}, \mathcal{H})$. Then,*

$$\forall t \in Trs_{\mathcal{H}}(p, \mathcal{D}),\ \forall t' \in Trs_{\bar{\mathcal{H}}}(p, \mathcal{D}) :\ \neg Dist(t, t') \qquad \Longrightarrow final(t') \in St(\mathcal{B}_{\bar{\mathcal{H}}})$$

Proof: We prove the lemma by induction of the length of p. We show that in the case of not distinguishable traces, the pruning operation is ineffective, and therefore the path traces end in the beliefs progressed by satisfying the assumption.

From def.5.2.9 and the hypothesis, we have that t, t' are bound to a path p, as defined in def. 5.2.3, consequently $\forall t = [\overline{s}, \overline{o}, \overline{\alpha}] \in Trs_{\mathcal{H}}(p, \mathcal{D}), \forall t' = [\overline{s'}, \overline{o'}, \overline{\alpha'}] \in Trs_{\bar{\mathcal{H}}}(p, \mathcal{D})$

$$\begin{gathered} \neg Dist(t, t') \Rightarrow \bar{o} = \bar{o'} \wedge \bar{\alpha} = \bar{\alpha'}, \quad \text{and} \\ \bar{o} = \bar{o'} \Rightarrow \forall n \leq \boldsymbol{t}, \forall s' \in t', \forall s \in t : \mathcal{X}(s^n) \cap \mathcal{X}(s'^n) \neq \emptyset \end{gathered} \tag{A.1}$$

The traces t, t' are bound to the same path p, which implies that $\bar{\alpha} = \bar{\alpha'}$.

We consider, for the *base case* of the proof by induction, a pair of annotated beliefs $\langle \mathcal{B}^0_{\mathcal{H}}, \mathcal{B}^0_{\bar{\mathcal{H}}} \rangle$, defined as in (5.4). It is trivial to see that all the final states of t' (resp. t) are in $St(\mathcal{B}^0_{\bar{\mathcal{H}}})$ (resp. $St(\mathcal{B}^0_{\mathcal{H}})$). Following (5.4), the pair $\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle$ is built up by applying $prune$ to $\mathcal{B}^0_{\mathcal{H}}$ and $\mathcal{B}^0_{\bar{\mathcal{H}}}$. From (A.1) and the definition of $prune$ function, it comes that $prune(\mathcal{B}^0_{\mathcal{H}}, \mathcal{B}^0_{\bar{\mathcal{H}}}) = \mathcal{B}^0_{\mathcal{H}}$, and that $prune(\mathcal{B}^0_{\bar{\mathcal{H}}}, \mathcal{B}^0_{\mathcal{H}}) = \mathcal{B}^0_{\bar{\mathcal{H}}}$, which proves the lemma at base step.

By *induction step* we assume that, given a pair of annotated belief $\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle$ progressed on p, via the function $\mathcal{E}$: $\forall t' \in Trs_{\bar{\mathcal{H}}}(p, \mathcal{D}) : final(t') \in St(\mathcal{B}_{\bar{\mathcal{H}}})$. We now prove the lemma on the progression of $\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle$.

- If an observation o is applied, then $\mathcal{E}(\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle, o) = \langle \mathcal{B}'_{\mathcal{H}}, \mathcal{B}'_{\bar{\mathcal{H}}} \rangle$, i.e. $St(\mathcal{B}'_{\bar{\mathcal{H}}}) = \{s \in \mathcal{B}_{\bar{\mathcal{H}}} : s \in [[o]]\}$, as defined in (5.2). Thus $St(\mathcal{B}'_{\bar{\mathcal{H}}}) = \{final(t'') : \forall t'' \in Trs_{\bar{\mathcal{H}}}(p \circ o, \mathcal{D})\}$.

- If an observation $\tilde{o}$ is applied, then the case is very similar to the latter, and we observe that if $\mathcal{E}(\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle, \tilde{o}) = \langle \mathcal{B}'_{\mathcal{H}}, \mathcal{B}'_{\bar{\mathcal{H}}} \rangle$, then $St(\mathcal{B}'_{\bar{\mathcal{H}}}) = \{final(t'') : \forall t'' \in Trs_{\bar{\mathcal{H}}}(p \circ \tilde{o}, \mathcal{D})\}$.

- If an action α is applied, then $\mathcal{E}(\langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle, \alpha) = \langle \mathcal{B}'_{\mathcal{H}}, \mathcal{B}'_{\bar{\mathcal{H}}} \rangle$, where $\mathcal{B}'_{\bar{\mathcal{H}}} = prune\big(\mathcal{E}(\mathcal{B}_{\bar{\mathcal{H}}}, \alpha), \mathcal{E}(\mathcal{B}_{\mathcal{H}}, \alpha)\big)$.

  Given that $\forall t' \in Trs_{\bar{\mathcal{H}}}(p, \mathcal{D}) : final(t') \in St(\mathcal{B}_{\bar{\mathcal{H}}})$, from the latter operation we have that $St(\mathcal{E}(\mathcal{B}_{\bar{\mathcal{H}}}, \alpha)) = \{s \in final(t'') : t'' = t' \circ [s, o, \alpha] : \forall t' \in Trs_{\bar{\mathcal{H}}}(p, \mathcal{D}), o \in \mathcal{X}(s)\}$; this belief corresponds to the set of final states of the traces bound to path $p' = p \circ \alpha$.

  From the relation (A.1), applying the pruning operator to build $\mathcal{B}'_{\bar{\mathcal{H}}}$ brings that $St(\mathcal{B}'_{\bar{\mathcal{H}}}) = \mathcal{E}(\mathcal{B}_{\bar{\mathcal{H}}}, \alpha)$, i.e. no states are pruned because of indistinguishableness of the traces. Consequently, this brings to the set of final states of the trace: $\{final(t) : \forall t' \in Trs_{\bar{\mathcal{H}}}(p \circ \alpha, \mathcal{D})\} \subseteq St(\mathcal{B}'_{\bar{\mathcal{H}}})$.

Theorem A.5.4 *Let π be a conditional plan for a domain $\mathcal{D} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{I}, \mathcal{R}, \mathcal{X} \rangle$, let $\mathcal{G} \subseteq \mathcal{S}$ be a goal, and let $\mathcal{H} \in \mathcal{L}(\mathcal{P})$ be an assumption. If $\tau(\pi, \mathcal{I}, \mathcal{H})$ does not contain the Fail node, and $\forall \langle \mathcal{B}, \langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle \rangle \in leaf\big(\tau(\pi, \mathcal{I}, \mathcal{H})\big) : St(\mathcal{B}) \cup St(\mathcal{B}_{\bar{\mathcal{H}}}) \subseteq \mathcal{G}$, then π is a safe solution for $\mathcal{G}$ under $\mathcal{H}$.*

Proof: Let us consider a generic leaf $\langle \mathcal{B}, \langle \mathcal{B}_{\mathcal{H}}, \mathcal{B}_{\bar{\mathcal{H}}} \rangle \rangle$ of $\tau(\pi, \mathcal{I}, \mathcal{H})$.

From lemma A.5.2, we derive that $\forall t \in Trs_{\mathcal{H}}(\pi, \mathcal{D}) : final(t) \in \mathcal{G}$, i.e. that

$$Trs_{\mathcal{H}}(\pi, \mathcal{D}) = TrsG(\pi, \mathcal{D}, \mathcal{G}) \tag{A.2}$$

From lemma A.5.3, we derive that $\forall t' \in Trs_{\bar{\mathcal{H}}}(\pi, \mathcal{D}) : final(t') \in \mathcal{G}$, i.e. that

$$Trs_{\bar{\mathcal{H}}}(\pi, \mathcal{D}) = TrsG_{\bar{\mathcal{H}}}(\pi, \mathcal{D}, \mathcal{G}) \tag{A.3}$$

By conjoining (A.2) and (A.3) over every leaf of $\tau(\pi, \mathcal{I}, \mathcal{H})$, we obtain def.5.2.10.