Likelihood-based Planning with Loops

Laszlo Treszkai

Master of Science

Artificial Intelligence

School of Informatics

University of Edinburgh

2018

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Laszlo Treszkai*)

Table of Contents

List of Figures

Chapter 1

Introduction

The problem of *automated planning* can be described informally as such: given a description of an environment, an initial state and a set of goal states, devise a sequence of actions that brings the environment to a goal state (Russell and Norvig, 2010).

This problem is one of the earliest ones that AI researchers have been working on. An early attempt for planning in a real-world environment was SHAKEY (Fikes and Nilsson, 1971), which was a mobile robot equipped with sensors and wheels and which could plan how to execute various tasks, including navigation according to a given floor plan and manipulating physical objects.

The earliest formalisations of planning assumed that the environment is deterministic and fully observable, i.e. the agent is always aware of the exact state at any moment, and executing a given action in a given state always brings the environment to the same next state. These assumptions do not hold in the real world, where the environment is usually *partially observable* and *noisy*: the observations of the agent depend on the current state, but an observation does not necessarily determine the state; and an action can bring the environment to any of a number of next states, with different probabilities. Hence, in *probabilistic planning* with these more relaxed conditions, the purpose changes slightly: bring the environment to a goal state either with the highest probability possible, or in the least number of steps. Alternatively, a planner ensures that the probability of ending up in a goal state with a given plan exceeds a given probability.

In *iterative planning*, plans can represent *loops*. This is useful in environments where the same sequence of actions are repeated along a path to the goal: for example, when chopping a tree, a `chop` action is executed (the tree is hit with

the axe) until the tree is up. This property can be exploited if the plan includes loops of some sort: ranging in capability from a sequence of actions with `while` loops that depend on the current observation (Levesque, 2005), through a more general finite state controller (Hu and Levesque, 2009), to a Turing-complete abacus program (Srivastava et al., 2015). These plans can have some internal state such as a program counter, and the proposed action depends on this state; furthermore, the next internal state depends on the current internal state and the current observation; and finally, the next state can be any of the previous states, making these plans capable of representing loops.

Besides making a plan more compact, the presence of loops is also useful for *generalised planning*, where the same plan must be correct for a set of planning problems, not only a single one. For example, a loopy plan for executing the `chop` action until the tree is sensed to be up, will succeed in bringing the tree down, regardless of the number of chops needed initially.

Our work combines the fields of iterative planning and probabilistic planning, and sets out to answer the following question: *Can we synthesise finite state controllers for a noisy environment?*

1.1 Contributions

Our contributions in this work are threefold. First, an analysis of the planning algorithm described in (Hu and De Giacomo, 2013). Second, we propose the PANDOR algorithm, which is a probabilistic planner capable of synthesising loopy plans in noisy environments. Third, we propose new planning domains which contain noisy actions, absorbing non-goal states, and which lend themselves to iterative planning.

1.2 Outline of this thesis

The rest of this thesis is structured as follows. Chapter 2 gives a formal account of planning in noisy environments and iterative planning. This includes conventions for describing the correctness of a probabilistic plan, and describing an existing algorithm for synthesising loopy plans for deterministic environments. Chapter 3 describes the primary contribution of this thesis, the PANDOR algorithm, answering the motivating question positively. Besides a description of the

algorithm and its implementation, a theorem is proved about its correctness. We also state a theorem about sufficient conditions for when an iterative plan generalises in noisy domains. Chapter 4 describes the new planning environments we designed, along with the performance of the planner in one of these environments, and the properties of the synthesised plans. In chapter 5 we summarise this project, and propose possible further research directions.

Chapter 2

Background

The problem of planning starts with some description of an environment: a collection of states and actions, and descriptions of how actions change the current state of the environment. In probabilistic planning, actions have probabilistic action effects: if the environment is in some state, executing an action can bring the environment to any of multiple states, with some pre-defined probability distribution over the possible next states. The goal of a probabilistic planner is to devise a plan that brings the environment to a goal state, while optimising the success probability. In order to avoid the seemingly tautological definitions, we have to resort to formal notions of what exactly is an environment or a plan. After a short historic account of planning, we lay down the definitions that will be used throughout this thesis.

The informal notion of planning given above has been formalised in countless ways, which differ in the used mathematical framework, their definition of the environment, planning problem, and plans. Fikes and Nilsson (1971) defined the STRIPS planning language, to describe states of the environment in a *factored representation*, i.e. as the Cartesian product of variables. The preconditions and effects of actions were likewise defined as sets of propositions, which are true before/after an action. STRIPS gave rise to the more powerful PDDL (Planning Domain Description Language), which is now the de facto standard for describing deterministic planning problems, used for example at the International Planning Competition (Ghallab et al., 1998). Levesque et al. (1998) described how the framework called *situation calculus*, which uses a second-order multi-sorted logic, could be used for modelling planning domains. Such a planning domain can have multiple models, which model would be what others call

an environment. Younes and Littman (2004) introduced an extension to PDDL called PPDDL, for describing planning domains with probabilistic action effects. Other extensions can make a planning problem more realistic through planning with multiple agents or parallel action execution, just to name a few (Russell and Norvig, 2010). We will use the framework by Belle and Levesque (2016), who give "a specification of a (robot) controller operating in a dynamic environment, which completely abstracts from the syntactic or structural characterization of a plan representation". In addition, they specify what it means for a controller to be "correct" or "terminating" for a planning problem, whether in a deterministic or stochastic environment.

2.1 The planning framework

In this framework, an environment and a controller interact with one another in turns. (Sometimes we use the term "agent" as a synonym for controller, to ease imagination.) At each time step, the controller makes an observation about the current state of the environment, and proposes an action based on all of its past observations. As a response to this action, the environment changes its state (possibly stochastically), and this cycle repeats until the controller outputs a `stop` action. The formal definition is as follows.

Definition 1. An *environment* $\mathcal{E}$ is defined as a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \Delta, \Omega \rangle$, whose elements are the following:

- $\mathcal{S}$ is a (possibly infinite) set of states (also called "state space");
- $\mathcal{A}$ is a (possibly infinite) set of actions (also called "action set" or "action space");
- $\mathcal{O}$ is a (possibly infinite) set of observations;
- $\Delta : \mathcal{S} \times \mathcal{A} \to \Pi(\mathcal{S})$ is a stochastic state transition function, where $\Pi(\mathcal{S})$ denotes the set of probability distributions over $\mathcal{S}$;
- $\Omega : \mathcal{S} \to \mathcal{O}$ is an observation function.

Suppose the state of the environment at time step 0 is $s^{(0)}$. The agent makes an observation $o^{(0)} = \Omega(s^{(0)})$, based on which it proposes an action $a^{(0)}$. Then the environment changes its state to $s^{(1)}$, which is sampled from the probability

distribution $\Delta(s^{(1)} \mid s^{(0)}, a^{(0)})$. The new observation of the controller is $o^{(1)} = \Omega(s^{(1)})$, and based on $o^{(0)}$ and $o^{(1)}$, the next action $a^{(1)}$ is chosen, and so on.

Notation. We denote the value of any time-dependent variable x at time step t during the execution by $x^{(t)}$. A sequence with index variable t ranging from i to j is written as $\langle x^{(t)} \rangle_{t=i}^{j} := \langle x^{(i)}, x^{(i+1)}, \ldots, x^{(j)} \rangle$. The subsequence of a sequence $h = \langle x^{(t)} \rangle_t$ between indices i and j is denoted by $h^{(i:j)} := \langle x^{(i)}, x^{(i+1)}, \ldots, x^{(j)} \rangle$, and the beginning subsequence is denoted by $h^{(:j)} := h^{(0:j)}$. The concatenation of two sequences is denoted by $\cdot$, e.g. $h^{(i:j)} \cdot h^{(j+1:k)} = h^{(i:k)}$.

The Ω function is extended to sequences of states in the natural way, defining the *sensed* function: $\mathit{sensed}(\langle s^{(t)} \rangle_{t=i}^{j}) := \langle \Omega(s^{(t)}) \rangle_{t=i}^{j}$.

Remark. Note that observations are made deterministically, but this does not limit the class of environments that can be modelled. Noisy observations can be modelled by a partially observable environment, where randomness is represented by a "hidden variable" in $\mathcal{S}$.

There is also a special action, `stop`, which is not an element of the action set. When the `stop` action is executed, the state→observation→action→next-state cycle stops.

A basic planning problem is defined as an environment, an initial state and a set of goal states:

Definition 2. A *basic planning problem* $\mathcal{P}$ is a triple $\langle \mathcal{E}, s_0, \mathcal{G} \rangle$, where $\mathcal{E}$ is an environment with state space $\mathcal{S}$, $s_0 \in \mathcal{S}$ is called the *initial state*, and $\mathcal{G} \subset \mathcal{S}$ is the *set of goal states*.

2.1.1 Controllers, systems, and notions of adequacy

Intuitively, a controller is usually a solution to a basic planning problem, in the sense that when started from the initial state of the environment, it will output actions at each time step that brings the environment "closer" to the goal, and ideally it ends up in a goal state with certainty in a number of steps bounded from above. It need not be so goal-oriented: a controller is defined by what action it would propose given all its past observations. Formal definitions follow, again from (Belle and Levesque, 2016).

A controller is a mapping from finite sequences of observations to actions:

Definition 3. A *controller* $\mathcal{C}$ for environment $\mathcal{E}$ is any function from $\mathcal{O}^{<\omega}$ to $\mathcal{A} \cup \{\texttt{stop}\}$, where $\mathcal{O}^{<\omega}$ is the set of finite sequences of $\mathcal{O}$.

An environment and a controller together form a *system*:

Definition 4. A *system* is a pair $\langle \mathcal{E}, \mathcal{C} \rangle$, where $\mathcal{E}$ is an environment and $\mathcal{C}$ is a controller with compatible action space. ("Compatible" in the sense that the range of $\mathcal{C}$ must be a subset of the action space of $\mathcal{E}$.)

A *run* of a system from a given state is one possible sequence of states that it follows, not necessarily until termination.

Definition 5. A *run* $\sigma = \langle s^{(t)} \rangle_{t=0}^{T} \in \mathcal{S}^{<\omega}$ of a system $\langle \mathcal{E}, \mathcal{C} \rangle$ (at environment state $s^{(0)}$) is a non-empty sequence of environment states such that $\Delta(s^{(t+1)} \mid s^{(t)}, a^{(t)}) > 0$, where $a^{(t)} = \mathcal{C}\big(\mathit{sensed}(s^{(0:t)})\big)$ for each $0 \leq t < T+1$. A run is *terminating* if at its end, the controller executes `stop`: $\mathcal{C}\big(\mathit{sensed}(\sigma)\big) = \texttt{stop}$. A run for a planning problem is a *goal run* if it is terminating and the last state is a goal state of the problem. Unless otherwise noted, the first state of a run is the initial state of the planning problem under consideration.

Starting from chapter 3, we will rely on the definition of the likelihood of a run, which is the probability that the system will follow the given sequence of states.

Definition 6. The likelihood of a finite run $\sigma = \langle s^{(t)} \rangle_{t=0}^{T}$ of a system $\langle \mathcal{E}, \mathcal{C} \rangle$ is denoted by $\ell(\sigma)$, and is defined inductively based on the length of a run:

- $\ell(\sigma^{(:0)}) = 1$, and
- $\ell(\sigma^{(:t+1)}) = \ell(\sigma^{(:t)}) \cdot \Delta(s^{(t+1)} \mid s^{(t)}, a^{(t)})$, where $a^{(t)} = \mathcal{C}\big(\mathit{sensed}(\sigma^{(:t)})\big)$.

With these tools we can now define the correctness of a plan and other notions of adequacy, all of which are characterised by the runs of a system.

Definition 7. A controller $\mathcal{C}$ is θ-adequate for a basic planning problem $\mathcal{P} = \langle \mathcal{E}, s_0, \mathcal{G} \rangle$, if the system $\mathcal{E}, \mathcal{C}$ fulfils the criteria required for θ. These criteria for different θ are the following:

- **ONE**: The system has at least one goal run.
- **TER**: If σ is a run of the system, then there is a run σ' from $\mathit{end}(\sigma)$ such that $\sigma \cdot \sigma'$ is terminating.

- **PC**: All terminating runs of the system are goal runs.
- **ACYC**: If $s^{(0:t)}$ is a run of the system, then $s^{(i)} \neq s^{(t)}$ for every $i < t$.

PC stands for *partial correctness*, and **ACYC** for *acyclicity*. For noisy environments, further quantitative notions can be defined, using the likelihoods of terminating and goal runs. The termination likelihood is a real number between 0 and 1, denoting the probability that the system will terminate – it can be thought of as the probabilistic counterpart to **TER**:

$$\mathbf{LTER} := \sum_{\{\sigma \mid \sigma \text{ is a terminating run}\}} \ell(\sigma).$$

The (normalised) likelihood of partial correctness is the conditional probability that the system terminates in a goal state, given that it terminates:

$$\mathbf{LPC} := \frac{1}{\mathbf{LTER}} \sum_{\{\sigma \mid \sigma \text{ is a goal run}\}} \ell(\sigma).$$

The following notion is not defined in (Belle and Levesque, 2016), but we will frequently use the *unnormalised* likelihood of partial correctness, which is the total likelihood of goal runs:

$$\mathbf{LTERPC} := \sum_{\{\sigma \mid \sigma \text{ is a goal run}\}} \ell(\sigma) = \mathbf{LTER} \cdot \mathbf{LPC}.$$

2.2 Generalised planning

Often we want a plan that is correct not only for a single basic planning problem, but which is correct on a set of problems. For example, when we task a housekeeping robot with doing the dishes, we do not want to specify the exact initial state of the environment, including the number of plates it should clean: we want it to have a plan that is general enough to handle *any* number of plates. In other words, we want to plan for a set of initial states, which set includes different initial robot locations and numbers of plates to handle.

Definition 8. A *generalised planning problem* $\overline{\mathcal{P}}$ is a triple $\langle \mathcal{E}, S_0, \mathcal{G} \rangle$, where S_0 is a (possibly infinite) set of environment states. Alternatively, $\overline{\mathcal{P}}$ is a collection of basic planning problems, $\langle \mathcal{E}, s_0, \mathcal{G} \rangle$, for every $s_0 \in S_0$.

The notions of adequacy were defined in the previous section only for basic problems; this extends to general problems intuitively.

Definition 9. A controller $\mathcal{C}$ is θ-adequate for a generalised planning problem $\overline{\mathcal{P}} = \langle \mathcal{E}, S_0, \mathcal{G} \rangle$, if $\mathcal{C}$ is θ-adequate for every basic planning problem in $\overline{\mathcal{P}}$.

Brief historic account

KPLANNER of Levesque (2005) was an early attempt at generalised planning. The structure of the generalised problems is given to the planner as input. It operates on planning domains defined in the situation calculus, and generates so-called *robot programs*. These programs consist of a sequence of actions, conditional action sequences, and looping action sequences. It synthesises such robot programs by generating a sequential plan for a basic problem chosen by hand, attempting to "wind up" this sequential plan into a loopy form, and testing if the result is correct for another instance of the problem. They conjectured that under certain conditions, the generated plans are correct for any basic problem of the domain.

FSAPLANNER of Hu and Levesque (2009) is another planner based on the situation calculus. It searched the space of finite state controllers (in the form of Moore machines), which are more general than robot programs, but still have a finite memory. In (Hu and Levesque, 2011), the same authors proved that in so-called *one-dimensional planning problem*s, it is sufficient to verify a plan for a finite number of basic problems in order for it to be correct on an infinite number of basic problems. Given the description of a planning problem, it is simple to decide whether it is one-dimensional or not.

ARANDA of Srivastava et al. (2008) generates sample plans and generalises them, using automatically identified state abstractions to "represent situations with unknown quantities of objects and compute the possible effects of actions on such situations."[1] They used abacus machines to represent plans, and showed that these plans generalise for a class of domains called Extended-LL; membership to this class can be shown inductively. Their method remains under constant development, e.g. (Srivastava et al., 2015) described circumstances under which such abacus programs would terminate or are correct.

[1] Quote from Srivastava (2010).

The planner of Bonet et al. (2009) turned partially observable planning problems into a contingent planning problem described by a set of propositions, and solved this problem with a SAT-solver.

The AND-OR planner of (Hu and De Giacomo, 2013) follows a similar algorithm as (Hu and Levesque, 2011), in that they both search the space of finite state controllers, while enumerating their runs at the same time. Section 2.5 gives a detailed description of this method.

2.3 Loopy plans

Imagine the following basic planning problem: the agent starts out with an axe in hand, and a tree of width N in front of it, and the goal is to fell the tree, by executing a `chop` action, which reduces the width of the tree by 1. In any such problem, the value of N is fixed and known to the planning algorithm, so a correct sequential plan would consist of executing `chop` N times, terminating the plan with `stop`. A more concise representation of this controller would be an *iterative plan*: if we sense the tree to be down, execute the `stop` action; if we sense the tree to be up, execute a `chop` action and repeat from the beginning. (An iterative plan is also called a *loopy plan*.) This iterative plan would also be correct for the generalised planning problem that contains such basic problems with arbitrary initial values of tree width, whereas no finite sequential or conditional plan would suffice.

One way to model an iterative plan is a finite state controller, such as a Mealy machine (Mealy, 1955).

Definition 10. A *finite state controller* (FSC) C is defined by a tuple $\langle Q, q_0, \mathcal{O}, \mathcal{A}, \gamma, \delta \rangle$, where

- $Q = \{q_0, q_1, \ldots, q_N\}$ is a finite set of states (called *controller states* or *machine states*),
- $q_0 \in Q$ is the *initial state* of the controller,
- $\mathcal{O}$ is a set of possible observations,
- $\mathcal{A}$ is a set of possible actions,

- $\gamma : Q \times \mathcal{O} \to (\mathcal{A} \cup \{\texttt{stop}\})$ is a partial function[2] called the *labelling function*, and

- $\delta : Q \times \mathcal{O} \to Q$ is a partial function called the *transition function*.[3]

As a notational convenience, an FSC is sometimes identified with the product of its transition and labelling functions: we write $C(q, o) = \langle q', a \rangle$ or $q \xrightarrow{o/a} q' \in C$ as a shorthand for $(\delta(q, o) = q')$ and $(\gamma(q, o) = a)$. The controller without any defined transitions is denoted by C_ε.

In the context of planning, an FSC forms part of a system $\langle \mathcal{E}, \mathcal{C} \rangle$, for a planning problem $\langle \mathcal{E}, s_0, \mathcal{G} \rangle$. The picture is the following: initially the environment is in state $s^{(0)} = s_0$, and the FSC C is in controller state $q^{(0)} = q_0$. The controller makes an observation $o^{(0)} = \Omega(s_0)$, executes action $a^{(0)} = \gamma(q^{(0)}, o^{(0)})$, and transitions to machine state $q^{(1)} = \delta(q^{(0)}, o^{(0)})$. The environment transitions to state $s^{(1)} \sim \Delta(s^{(1)} | s^{(0)}, a^{(0)})$, and this process is repeated until a `stop` action is executed. (If a controller transition or action is undefined for the current controller state and observation, then in this work we treat it as if it terminated, as it is commonly the case with Turing machines (Boolos et al., 2007).) Hereafter all controllers are finite-state controllers, so for convenience, no distinction will be made between the controller function $\mathcal{C}$ and the FSC C.

For convenience, we introduce the following term:

Definition 11. A *combined state* for an FSC C with states in Q and an environment with state space $\mathcal{S}$ is a pair $\langle q, s \rangle$ such that $q \in Q$ and $s \in \mathcal{S}$.

We say that two FSCs are *isomorphic* if they differ only by state renaming. As the actions of isomorphic controllers is the same for every sequence of observation, we'll assume without loss of generality that a controller with N states has states $q_i = i$ for $0 \leq i \leq N - 1$.

A Mealy machine can be visualised with a circle for each state and labelled arrows between them to denote state transitions, with a "condition / action" pair as each label. (The initial state is marked with an extra incoming arrow.) Figure 2.1 shows two example controllers for the TreeChop domain from the

[2] A function $f : X \to Y$ is called *partial* if it doesn't necessarily define a value for all elements of the domain X. Alternatively, $f : X' \to Y$ is a (total) function with $X' \subseteq X$.

[3] In some formalisms, the transition function takes the role of the labelling function too, i.e. $\delta : Q \times \mathcal{O} \to Q \times (\mathcal{A} \cup \{\texttt{stop}\})$.

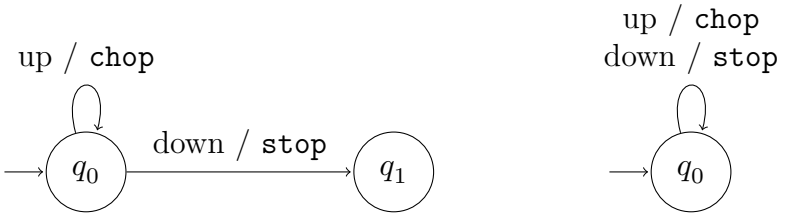


Figure 2.1: Two example controllers for the TreeChop domain.

beginning of this section, which output the same actions, although they are not isomorphic.

A frequently used alternative to this model is the *Moore machine* (Moore, 1956), where the proposed action doesn't depend on the current observation, so with every machine state there is a fixed action associated: $\gamma : Q \to \mathcal{A} \cup \{\texttt{stop}\}$). Given that there is an action that doesn't change the environment state which can be executed at the very beginning, the Moore and Mealy models are computationally equivalent (Babcsányi, 2000), although a Mealy machine typically has a fewer number of states for the same problem. Moore machines are used for example in (Kaelbling et al., 1998) and (Hu and Levesque, 2011).

The next section describes how such loopy plans can be synthesised.

2.4 Planning with loops

A wide range of representations and approaches have been developed for planning with loops, each with different strengths. Many of these were developed for generalised planning, so they have already been presented in section 2.2; in this section here we mention one more method, which was not developed for generalised planning.

The bounded policy iteration of Poupart and Boutilier (2003) approximates the optimal value function of a partially observable Markov decision process with an FSC. Their method is most applicable for generating close to optimal plans when the agent is uncertain about the current state, and when its beliefs have a possibly large support. In an environment with states $\mathcal{S}$, the belief space is the $|\mathcal{S}|$-simplex, i.e. an $|\mathcal{S}|-1$-dimensional manifold. The value function is approximated over this high-dimensional space with the value function of a finite state controller. This method is most applicable when we want the agent to update its beliefs based on partial or noisy observations, and act differently in different belief states.

The next section describes the AND-OR controller search of Hu and De Giacomo (2013) in great detail, as we will use this planner as our foundation for a probabilistic variant in chapter 3.

2.5 The AND-OR search for controllers

A general algorithm for *conformant planning* is the AND-OR planner, which takes as input a discrete finite planning problem (either basic or generalised), and returns a *conditional plan* that is **TER** and **PC** if one exists, or **failure** otherwise (Russell and Norvig, 2010, p. 136). The name "AND-OR" comes from the graph representation of the runs of the environment: at a given state, only one action is chosen (OR), and the plan needs to be correct for all outcomes of an action (AND). The generated plan is a graph itself: a directed tree with the empty history at the root, state-action pairs as nodes, which are connected to all the possible next states. This planner was modified first by Hu and Levesque (2009), then by Hu and De Giacomo (2013), to generate finite state controllers instead of conditional plans.

Combining search in controller space and state space. The following insights were needed for their algorithm. First, that every FSC C for environment $\mathcal{E}$ defines a set of runs for the system formed by $\mathcal{E}$ and C. Second, the extensions of a controller always *extend* the set of possible runs of the system: every run of the smaller controller is a run of the larger controller too. Through exploiting these two facts, one can simulate the runs of the system *together* with searching through the space of possible controllers.

Planning framework. This AND-OR planner operates in what the authors call a "dynamic environment". This is similar to a finite environment of (Belle and Levesque, 2016), but instead of the environment state transition function $\Delta : \mathcal{S} \times \mathcal{A} \to \Pi(\mathcal{S})$, we have a state transition *relation* $\Delta \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$. Executing action a in state s brings the environment to multiple next states s', for which $\langle s, a, s' \rangle \in \Delta$. This implies that an action with multiple outcomes doesn't reduce the likelihood of a run, so when the planner synthesises a controller for an environment, it is either correct or not correct; there is no middle ground, such as a likelihood of partial correctness.

2.5.1 Algorithmic details

The description of the search algorithm begins with some definitions from graph theory.

Definition 12. An *AND-OR graph* is defined as a tuple $\langle V, E, f\rangle$, where $\langle V, E\rangle$ is a directed graph with nodes in V and edges in E, and $f : V \to \{\mathsf{and}, \mathsf{or}\}$ is a function that defines for each node v whether it's an AND node ($f(v) = \mathsf{and}$) or an OR node ($f(v) = \mathsf{or}$). A *rooted AND-OR tree* $G = \langle V, E, f, v_0\rangle$ is a directed AND-OR graph $\langle V, E, f\rangle$ with a dedicated root node $v_0 \in V$, in which for any non-root node $v \in V \backslash \{v_0\}$, there is exactly one directed path from v_0 to v. The *level* of a node v is the distance between v_0 and v. A node w is *child* of v if if there is an edge $\langle v, w\rangle \in E$, i.e. if the level of w is exactly one greater than that of v. A node w is a *descendant* of v if there is a directed path from v to w.

Remark. In this work, such a tree "grows downwards": for example, nodes on level 3 are said to be one level *deeper* or *below* than those on level 2.

The algorithm searches through the subgraphs of a (possibly infinite) rooted AND-OR tree G. Let $\mathcal{E} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \Delta, \Omega\rangle$ be a dynamic environment, and $Q = \{q_0, \ldots, q_{N-1}\}$ a finite set of controller states. (The value of N is an input to the algorithm, hence an FSC synthesised by the algorithm is said to be N-bounded.) Every AND node of G is a $\langle q, a, h\rangle$ tuple, and every OR node is a $\langle q, s, h\rangle$ tuple, where $q \in Q$ is a controller state, $s \in \mathcal{S}$ is an environment state, $a \in \mathcal{A}$ is an action, and h is a finite sequence of $\langle q^{(i)}, s^{(i)}\rangle$ pairs: the controller state and the environment state at time step i. In this chapter we call such a sequence h a *history*. (Note that this does not have the same meaning as in definition 4 of (Belle and Levesque, 2016).) A *run* of a system from a given state is one possible sequence of states that it follows, not necessarily until termination.

Definition 13. Let $C = \langle Q, q_0, \mathcal{O}, \mathcal{A}, \gamma, \delta\rangle$ be a finite state controller, with controller states Q. A *history* $h = \langle q^{(t)}, s^{(t)}\rangle_{t=0}^{T} \in (Q \times \mathcal{S})^{<\omega}$ of a system $\langle \mathcal{E}, C\rangle$ (at environment state $s^{(0)}$) is a non-empty sequence of combined states such that $\Delta(s^{(t+1)} \,|\, s^{(t)}, a^{(t)}) > 0$, where $a^{(t)} = \gamma(q^{(t)}, \Omega(s^{(t)}))$, and $q^{(t+1)} = \delta(q^{(t)}, \Omega(s^{(t)}))$ for each $0 \leq t < T$. A history for a planning problem $\langle \mathcal{E}, s_0, \mathcal{G}\rangle$ is a *goal history* if C terminates in a goal state: $\gamma(q^{(t)}) = \texttt{stop}$ and $s^{(t)} \mathcal{G}$. Unless otherwise noted, the first state of a history is $\langle q_0, s_0\rangle$.

The difference between a run and a history is that the latter includes the controller state at each time step (thus the definition is valid only for finite state controllers).

From an AND node in G all edges lead to OR nodes, and from an OR node edges lead only to AND nodes. The children of an AND node $\langle q^{(t)}, a^{(t-1)}, h^{(t-1)} \rangle$ are all possible tuples $\langle q^{(t)}, s^{(t)}, h^{(t-1)} \rangle$ such that $\langle s^{(t-1)}, a^{(t-1)}, s^{(t)} \rangle \in \Delta$ (with $s^{(t-1)}$ being the second coordinate of $end(h^{(t-1)})$). The children of an OR node $\langle q^{(t)}, s^{(t)}, h^{(t-1)} \rangle$ are all possible tuples $\langle q^{(t+1)}, a^{(t)}, h^{(t)} \rangle$ such that $a^{(t)}$ is a legal action in $s^{(t-1)}$, $h^{(t)} = h^{(t-1)} \cdot \langle q^{(t)}, s^{(t)} \rangle$, and there is no restriction on $q^{(t+1)}$. Thus the children of an AND node are defined by the possible responses of the environment, and as we'll see later, the children of an OR node are the possible responses of some controller, just like in the basic AND-OR search.

The root of the tree is an AND node, but a special one: $\langle q_0, \emptyset, \langle\rangle \rangle$, where q_0 is the initial state of the controller $\mathcal{C}$. The children of this node are OR nodes $\langle q_0, s^{(0)}, \langle\rangle \rangle$, with $s^{(0)} \in S_0$ being any of the possible initial states of the generalised planning problem $\overline{\mathcal{P}} = \langle \mathcal{E}, S_0, \mathcal{G} \rangle$.

This AND-OR graph G defines all possible runs for the environment, and a subtree of G defines the set of all runs of a system $\langle \mathcal{E}, \mathcal{C} \rangle$. For any finite state controller C, denote this subtree by $g(C) \subset G$. Define $\mathfrak{C}$ as the set of all N-bounded finite state controllers for $\mathcal{E}$. $\mathfrak{C}$ can be partially ordered with the ordering relation $\preceq$ as follows: $C_i \preceq C_j$ iff $\delta_i \subseteq \delta_j$ for the state transition functions and $\gamma_i \preceq \gamma_j$ for the labelling functions of C_i and C_j. The algorithm is basically a depth-first search in $\mathfrak{C}$ according to $\preceq$, and it returns the first controller which is found to be correct for $\mathcal{E}$, which is the first controller where all the leaf nodes are in $\mathcal{G}$. Naively implemented, the worst-case time complexity of this search would be linear in the size of $\mathfrak{C}$, the average length of runs of the controllers in $\mathfrak{C}$, and the average number of runs of a controller in $\mathfrak{C}$. This section describes how searching through $\mathfrak{C}$ and the set of runs can be performed at the same time, using the algorithm in Algorithm 1.

The algorithm starts its search with the set of possible initial states S_0, an empty controller C_ε, and an empty history $h = \langle\rangle$. (The empty controller consists of a single state q_0, and all of its state transitions and actions are undefined: $\delta(q, o) = \mathsf{undef}$, $\gamma(q, o) = \mathsf{undef}$ for all q, o.) This means we start at the root of G, and the relevant subgraph defined by the controller, $g(C_\varepsilon)$ is a single layer deep.

Algorithm 1 The AND-OR search algorithm for bounded finite state controllers, as described in (Hu and De Giacomo, 2013, Fig. 4), with slight modifications.

Require: $\overline{\mathcal{P}} = \langle \mathcal{E}, S_0, \mathcal{G} \rangle$, a generalised planning problem;
Require: N, a bound on the number of controller states.

function AndOr-det-synth($\overline{\mathcal{P}}, N$)
  return AND-step$_{\overline{\mathcal{P}},N}(C_\varepsilon, 0, S_0, \langle\rangle)$
end function

function AND-step$_{\overline{\mathcal{P}},N}(C, q, a, h)$
  $S' \leftarrow \{s' \mid \langle s, a, s' \rangle \in \Delta\}$
  for all $s' \in S'$ **do**
    $C \leftarrow$ OR-step$_{\overline{\mathcal{P}},N}(C, q, s', h)$
  end for
  return C
end function

function OR-step$_{\overline{\mathcal{P}},N}(C, q, s, h)$
  if $s \in \mathcal{G}$ **then**
    return C
  else if $\langle q, s \rangle \in h$ **then**
    fail
  else if $q \xrightarrow{\Omega(s)/a} q' \in C$ for some q', a **then**
    if a is not a legal action in s **then**
      fail
    end if
    return AND-step$_{\overline{\mathcal{P}},N}(C', q', a, h \cdot \langle q, s \rangle)$
  else
    non-deterministically choose $a \in \mathcal{A}$ and $q' \in Q_N^+(C)$
    $C' \leftarrow C \cup \{q \xrightarrow{\Omega(s)/a} q'\}$
    return AND-step$_{\overline{\mathcal{P}},N}(C', q', a, h \cdot \langle q, s \rangle)$
  end if
end function

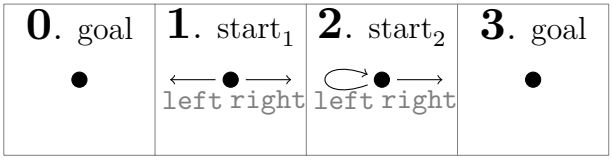


Figure 2.2: The WalkThroughFlap planning problem.

After the initial call to AndOr-det-synth, the program calls the AND-step and OR-step functions recursively, one after the other.

AND-step$_{\mathcal{E},Q}(C,q,a,h)$ synthesises a controller C' that is an extension of C, and which is correct for each possible next states when executing a from the end state of h. If C has no such extension, it returns with failure. OR-step$_{\mathcal{E},Q}(C,q,s,h)$ synthesises a controller C' that's an extension of C, and which takes any possible action a in state s, as restricted by the controller C and the environment $\mathcal{E}$.

The practical view of what happens when a non-deterministic choice is made with n possible choices is that the complete program state is saved, and the choices are enumerated in some order (which is not necessarily defined by the implementation), creating n branches of possible program execution. Upon failure of a non-deterministic branch, the complete program state is backtracked to the last non-deterministic choice, where the next branch is executed. If all possible branches have returned with failure, then the function returns with failure.

The $Q_N^+ \colon \mathfrak{C} \to 2^{\mathbb{N}}$ function used at line 24 is defined as such: if an FSC C has n states $Q = \{0,1,2,\ldots,n,-1\}$, then $Q_N^+(C) = \{0,1,2,\ldots,min(n,N-1)\}$. By picking a next state from $Q_N^+(C)$, the algorithm avoids looping through isomorphic controllers. For example, when selecting a new transition for a controller that has $|Q| = 2$ states while $N = 5$, then all three controller extensions with a new state in $\{2,3,4\}$ are isomorphic, so it is sufficient to test only one of them.

The working of the algorithm is best demonstrated on a toy planning problem we designed for this purpose.

The WalkThroughFlap planning problem. The environment of the WalkThroughFlap problem is a 4×1 grid world (Fig. 2.2). The states space is $\mathcal{S} = \{\mathbf{0},\mathbf{1},\mathbf{2},\mathbf{3}\}$, and the actions are `left` and `right`. The actions have the intuitive effect, except for one move: executing `left` in state **2** leaves the agent in place. The observation doesn't depend on the current state, i.e. the environment is *sensorless*: $\Omega(s) = \emptyset$ for every s. WalkThroughFlap is a generalised planning

problem, with two initial states, $S_0 = \{\mathbf{1}, \mathbf{2}\}$, and two goal states, $\{\mathbf{0}, \mathbf{3}\}$.

The algorithm starts with a call to AND-STEP, which first calls OR-STEP with $\langle q, s\rangle = \langle q_0, \mathbf{1}\rangle$. The OR-STEP notices that it is not yet in a goal state, so it attempts to extend the controller. The first possible action is `left`, so a new transition is added to C: $\langle q_0, \emptyset\rangle \to \langle q_0, \texttt{left}\rangle$. (Denote this new controller by C_{left}.) Then AND-STEP is called, which calls OR-STEP$(C_{left}, q_0, \mathbf{0}, \langle q_0, \mathbf{1}, \texttt{right}\rangle)$ in turn. State **1** is a goal state, so algorithm returns C_{left}, up to the first AND-STEP.

The program continues with OR-STEP$(C_{left}, q_0, \mathbf{2}, \langle\rangle)$. State **2** is not a goal state, but the controller already has an applicable transition defined. Thus the `left` action is taken, which – after an AND-STEP – ends in OR-STEP$(C_{left}, q_0, \mathbf{2}, \langle q_0, \mathbf{2}, \texttt{right}\rangle)$. State **2** is already present in the history, meaning continued execution of the same controller (or one of its extensions) would again bring the environment to the same state, making an endless cycle. For this reason, this controller fails (line 16), and program execution reverts to the point of the last non-deterministic choice: the call of OR-STEP$(C_\varepsilon, q_0, \mathbf{1}, \langle\rangle)$. This time, C_ε is extended with $\langle q_0, \emptyset\rangle \to \langle q_0, \texttt{left}\rangle$. Similar calls reveal that the graph of the resulting controller C_{right} has a goal state at every leaf, so this controller is returned as a correct controller.

Implementation in Python

While a partial implementation of the algorithm was given in (Hu and De Giacomo, 2013, Fig. 5), we could not obtain the full source code. Furthermore, this original implementation was written in Prolog, a logic programming language, with which we had no previous experience. To make further extensions simpler, we implemented the ANDOR-DET-SYNTH algorithm (Alg. 1) in Python 3.6, a high-level imperative programming language, with easy-to-use debugging capabilities. This section explains a few aspects of our implementation.

The planner is implemented in the `AndOrPlanner` class, which is instantiated for solving a generalised planning problem, and thus receives the environment as an argument for its initialisation function. The `Environment` class includes both the set of initial states and the set of goal states, so it is a description of a generalised planning problem. This class has the current controller and the backtracking stack as class properties.

The functions of the `AndOrPlanner` class have the following headers:

```
1 class AndOrPlanner:
2     def __init__(self, env):
```

```
        # initialise class properties

    def synth_plan(self, bound):
        # reset backtracking stack and the current controller,
        #  and call and_step

    def and_step(self, q, action, history):
        # ...

    def or_step(self, q, env_state, history):
        # ...
```

Below we describe two closely related aspects of how the AND-OR graph is traversed: non-deterministic branching and backtracking. These features are both language-level constructs in Prolog, but in Python they required more elaborate program flow.

For comparison, the Python implementation of a recursive depth-first planner is trivial:

```
def depth_first_planner(env, state, goals, history=[]):
    if state in goals:
        return history
    for action in env.legal_actions(state):
        history = history + [action]
        for next_state in env.next_states(state, action):
            plan = depth_first_planner(env, next_state, goals, history)
            if plan:
                return plan
    return None
```

This `depth_first_planner` function starts at the root of the graph G described above, iterates through each next environment state at an AND node and each possible action at an OR node. In an environment with finite state space and at least one goal run, this algorithm will return the sequence of actions that corresponds to a goal run, making the system **ONE**.

The implementation of a depth-first planner is simple because backtracking happens only to ancestor nodes, thus one can use the call stack to aid backtracking. However, in the AND-OR controller search, the call stack represents the currently simulated run, and the algorithm can backtrack to nodes in a different branch (as we saw in the WalkThroughFlap example). This means that the call stack doesn't contain the necessary information for backtracking, so an ordered list of non-deterministic choice points needs to be maintained separately.

Whenever a non-deterministic choice is made in an OR-step, we save all variables that are required to be able to completely restore the program execution to that point. This includes all the local variables at every level of the call stack, and all non-constant properties of the planner object, all of which can be completely different. This would make storing/restoring program execution at/to the

state at the last choice node difficult to implement, if it weren't for the following trick. By keeping the transitions of the controller in an ordered dictionary (which is implemented by the `collections.OrderedDict` class of the standard Python library), the controller will always have as many transitions as the number of non-deterministic choices made. (For simplicity, the non-deterministic choices with only one controller extension possible, are also treated as a choice point.) All other local variables depend only on the history, which means that it is sufficient to store a shallow copy of the current history.

At a non-deterministic choice in an OR node, the current history is pushed onto the backtracking stack (`self.backtrack_stack` of type `list`), and the ordered dictionary[4] of controller transitions is extended. This ensures that at every time, the controller has as many transitions as the depth of the backtracking stack, and the history when any transition was added is known.

Backtracking is achieved by setting a boolean flag `self.backtracking` in `and_step` when a call to an `or_step` failed, and then looking for the last "checkpoint": traversing up the tree to the relevant AND node (i.e. until the current history matches the beginning of the last item on the backtracking stack), and then traversing down the tree until the relevant OR node. There, `self.backtracking` is reset, the last controller transition is removed, and the next possible transition is added. In order for this to work, the possible controller extensions at a given OR node need to be enumerated in the same order, i.e. we cannot use a truly random order. (The Prolog implementation would ensure this by storing the list of possibilities on the backtracking stack.)

When all branches of a non-deterministic choice have failed, an item is popped from the backtracking stack, and the function of the non-det. choice returns with failure as well. This happens when a controller doesn't have an extension that could make a certain run end in a goal state.

The following requirements define the correct behaviour of such a planner:

1. If a child of an OR node v fails and there are yet unexplored children of v in $g(C)$, then one of these should be explored next.

2. If all children of an OR node v fail, then v itself fails.

3. If a child of an AND node v fails, and the last choice node w is a descendant

[4] An ordered dictionary is a dictionary that keeps track of the order in which items were added to it.

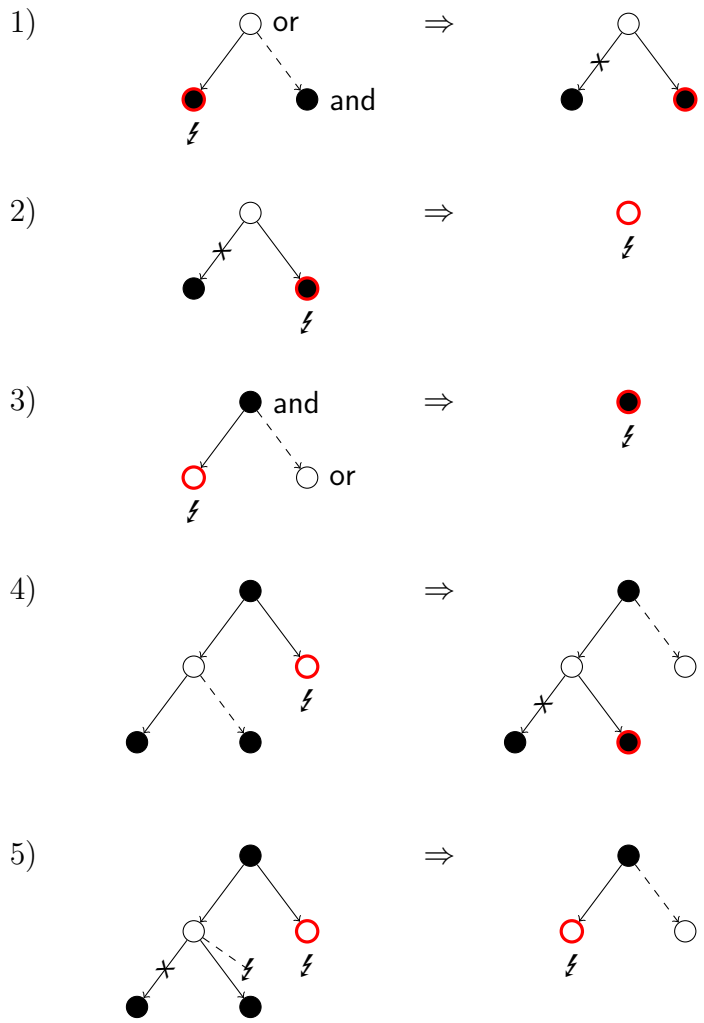


Figure 2.3: Graphs illustrating the requirements for backtracking. Legend: ● AND node. ○ OR node. ◉ or ◯ indicates the node currently active in the execution (i.e. the one at the end of the call stack). ◉ ↯ or ◯ ↯ indicates a node which is currently returning with failure. ●→○ An explored edge. ●→✕→○ An explored and failed edge. ●- - -→○ A yet-unexplored edge.

of v, then the execution should continue at w with the next possible choice, if there is one.

4. If a child of an AND node v fails, and the last choice node w is a descendant of v, and all possible choices at w have been explored, then w fails.

These requirements are illustrated in fig. 2.3. These requirements allow for a relatively simple implementation, although slightly different ones would be more efficient – this idea is explained in section 2.5.3.

2.5.2 Limitations and applicability

Due to the framework in which the planner operates (i.e. the dynamic environment with a state transition relation Δ), AndOr-det can handle non-deterministic environments, but it cannot assign probabilities to different outcomes.

Termination is guaranteed only in finite environments: while the AND-OR graph can be infinite even in a finite state space, it is bound to contain repeating combined states, as there is only a finite number of them. In an infinite state space, there are runs without such repetitions, so the planner would get into an infinite recursion while exploring such a run (in theory), or fail with stack overflow (in practice). This limitation could be removed with the iterative deepening of the run length, as described in the next section.

2.5.3 Potential improvements

This planner proved to be efficient for iterative planning: it found the correct solution faster than two other iterative planners on all the benchmark problems (Hu and De Giacomo, 2013). However, there are some simple changes that could further improve the efficiency of the search process.

Heuristics. Much of the progress in planners of this century came from using heuristics to guide the search process; see for example (Hoffmann and Nebel, 2001). Specifically, the non-deterministic choice in OR-step and the *for all* loop in AND-step (lines 24 and 7) do not specify the order of enumeration. At the action selection in OR-step, one can use a heuristic based on which action reduces the "distance" from the current state to a goal state the most. (Where the used distance metric could be defined based on the factorised state representation as in the PDDL language, or even further tuned for the planning problem by hand.) When picking a next state in AND-step, we want exactly the opposite, and *fail fast*. Here, a randomised enumeration of the alternatives could make the planner notice the failing outcomes sooner. For FSAPLANNER – the predecessor of the planner presented here – the authors reported a 100-fold speed improvement by using these two heuristics on certain planning problems (Hu and Levesque, 2009).

Backtracking. Another source of inefficiency is backtracking always to the *last* choice point. Suppose we have a generalised planning problem in a deterministic domain, with initial states $s_1^{(0)}$ and $s_2^{(0)}$. First, the algorithm synthesises

a controller that is correct for $s_1^{(0)}$. Then, moving on to $s_2^{(0)}$, if the observation is the same as in $s_1^{(0)}$, the controller already has an action defined. If this action leads to a failing state on the first step, the algorithm backtracks to the last added controller transition, and tries another one – needlessly, as the relevant transition was defined in the very first OR-STEP in $s_1^{(0)}$. Generally, one should backtrack until the last transition $\langle q, o\rangle \to \langle q', a\rangle$, where o matches the observation of at least one state in the current history.

Iterative deepening. Finally, we could extend the algorithm with *iterative deepening depth-first search* (Russell and Norvig, 2010, sec. 3.4.5). Iterative deepening refers to always exploring runs of bounded length, and extending this bound iteratively only when all controllers have been found to be inadequate for the current bound. This way one could synthesise controllers whose goal runs are as short as possible, although at the cost of discarding all controllers for every smaller bound. Hu and De Giacomo (2013) already employed iterative deepening, not for the length of simulated runs but the number of controller states. For example, synthesising a 4-state controller involved first proving that no smaller controller is adequate.

Chapter 3

Theoretical results

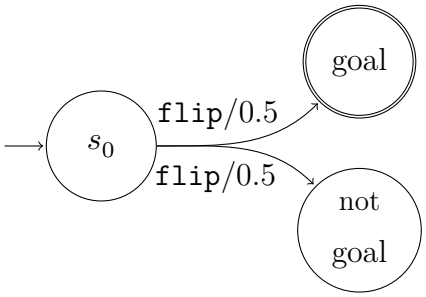
This chapter describes our primary results: the proposed planning algorithm, and a correctness result on probabilistic generalised planning problems. The planner was based on the planner described in section 2.5 and thus shares many of its properties, hence its name, PANDOR, which stands for Probabilistic AND-OR. In terms of functionality, it differs from its predecessor in two main aspects. First, it can synthesise a loopy plan that is not guaranteed to end up in a goal state on every run, but ends up in a goal state with a predefined likelihood. Second, due to the noise in the environment's response to the controller's actions, the planner shouldn't necessarily discard plans that loop in both machine state and environment state, so we propose a way to deal with such loops. In addition, a few minor features are added to the planner (such as the synthesis of terminating plans), and new heuristics for the exploration are explored, which were made possible by the presence of probabilities. Finally, a theorem is proved about the correctness of the planner, and a conjecture is stated about applicability in generalised planning problems.

3.1 Adding noise to action outcomes

In a deterministic planning problem, the environment responds with a single next state to every action, which means that every AND node has exactly one successor. (With the exception of the root node in a generalised planning problem.) This has the implication that a single controller is going to have exactly one run, which either ends up in a goal state or fails for some reason. This is not true in a noisy environment, where a single action can have multiple outcomes, making

the number of possible runs exponential in the number of steps. Unfortunately, many or all of these runs can be goal runs, or possibly none of them is.

Besides increasing the computational complexity, noise brings about a closely related issue. In some planning problems, even the optimal finite state controller might not terminate on every run, or end up in a goal state on every terminating run. (Consider a problem with an unavoidable dead end state, for example one where one outcome of a `coin_flip` action ends up in the goal, another outcome results in death – illustrated below.) In other environments, even a suboptimal controller might end up in a goal state occasionally. How to deal with this problem?



On one end is the case when nothing less than a terminating partially correct plan is satisfactory (i.e. $\mathbf{LTER} = 1$ and $\mathbf{LPC} = 1$). Then we can ignore the outcome probabilities apart from them being 0 or not, and Alg. 1 of the previous chapter will suffice.

The other edge case is when we satisfy with a plan being **ONE** – but then Alg. 1 runs into the complications described earlier, namely, at what point during the simulation do we deem a plan failed? The failure of a single AND node doesn't imply that the plan is not "good enough". Assuming that we don't need a proper controller (or that all controllers are proper, i.e. all actions are legal in every state and undefined transitions equal to a `stop` action), we can do with a simpler algorithm, because now at every step we need only care about a single outcome, and the other branches in $g(C)$ are irrelevant. As **ONE** adequacy is rarely enough, we propose an alternative.

Goal run likelihood. The partial correctness likelihood **LPC** required that a run terminates in a goal state. The controllers synthesised by the basic version of our algorithm do not execute a `stop` action. For this reason, we temporarily introduce **LG** as the likelihood that the controller reaches a goal state.

Definition 14. The goal run likelihood of a system $\langle \mathcal{E}, \mathcal{C} \rangle$ is denoted by **LG**, and is defined as

$$\mathbf{LG} := \sum_{\substack{\{\sigma \mid \sigma \text{is a run of the system} \\ end(\sigma) \in \mathcal{G}\}}} \ell(\sigma).$$

3.2 The Pandor algorithm

The PANDOR algorithm strikes a balance between these two extremes, by taking as a parameter a desired goal run likelihood $LG^\star$, so that the **LG** of the synthesised controller exceeds $LG^\star$. The trick is to keep track of the likelihood of simulated runs, and maintain upper and lower bounds on **LG** of the currently planned controller (denoted by $\overline{LG}$ and $\underline{LG}$, respectively).

The initially empty controller has a lower bound of 0 and an upper bound of 1. When a simulated run σ of likelihood $\ell(\sigma)$ ends in a goal state, the lower bound on **LG** is increased. When σ fails (either for lack of a legal action, or for a repeated combined state), the upper bound on **LG** is decreased by $\ell(\sigma)$. (The potential reasons for failure are detailed later in this section.) When $\underline{LG} \geq LG^\star$, we have found a satisfactory FSC, and it is returned. When $\overline{LG} < LG^\star$, no amount of the remaining runs could make the controller good enough, and we backtrack to the last choice point. Pseudocode for this algorithm is stated in Alg. 2.

We can see that ANDOR-DET-SYNTH is a special case of PANDOR-BASIC, namely with $LG^\star = 1$. In fact, on a deterministic (generalised) planning problem they have the same runtime complexity: their execution time and space requirements only differ by a linear factor, for keeping track of the run likelihoods.

The search graph. Like in section 2.5.1, the searched graph G is again completely defined by the planning problem $\mathcal{P} = \langle \mathcal{E}, s_0, \mathcal{G} \rangle$. While earlier an environment transition $s \xrightarrow{a} s'$ was either possible or not, now they are possible with a certain probability. This means that the nodes of G now include the probability of reaching that node from the root of the tree, according to the transition function of the environment. What we call a *history* in this section is a finite sequence of $\langle q^{(i)}, s^{(i)}, \ell^{(i)} \rangle$ triples: $q^{(i)}$ and $s^{(i)}$ the controller state and environment state at timestep i, and $\ell^{(i)}$ is the likelihood of the run $\langle s^{(t)} \rangle_{t=0}^{i}$ for the current controller C, as defined in the next paragraph. Still, the main difference between a "run" and a "history" is that the latter includes the state of

Algorithm 2 The PANDOR-BASIC algorithm that searches for bounded finite state controllers for a stochastic environment without termination and with bounded runs only. (Note: upon backtracking, the global variables are restored as well.) Red lines indicate the changes against Alg. 1.

Require: $\mathcal{P} = \langle \mathcal{E}, s_0, \mathcal{G} \rangle$, a basic planning problem;
Require: N, a bound on the number of controller states.
Require: $LG^\star$: the desired minimum **LG**

function PANDOR-BASIC-SYNTH($\mathcal{P}, N$)
  (global) $\overline{LG} \leftarrow 1$ ▷ Current upper bound on **LG**
  (global) $\underline{LG} \leftarrow 0$ ▷ Current lower bound on **LG**
  return AND-STEP$_{\mathcal{P},N}(C_\varepsilon, 0, \{\langle s_0, 1.0 \rangle\}, \langle\rangle)$
end function

function AND-STEP$_{\mathcal{P},N}(C, q, a, h)$
  $SP' \leftarrow \{\langle s', p' \rangle \mid \Delta(s' \mid s, a) = p' > 0\}$
  for all $\langle s', p' \rangle \in SP'$ **do**
    $C \leftarrow$ OR-STEP$_{\mathcal{P},N}(C, q, s', p', h)$
    if $\underline{LG} \geq LG^\star$ **then**
      return C ▷ Returns across the whole call stack
    else if $\overline{LG} < LG^\star$ **then**
      fail ▷ Backtrack to last non-deterministic choice
    end if
  end for
  return ▷ Return from AND-node without failure or success
end function

function OR-STEP$_{\mathcal{P},N}(C, q, s, p, h)$
  $\ell \leftarrow \text{end}(h).\ell \cdot p$ (or p if $h = \langle\rangle$)
  if $s \in \mathcal{G}$ **then**
    $\underline{LG} \leftarrow \underline{LG} + \ell$
  else if $\langle q, s, \bullet \rangle \in h$ **then**
    $\overline{LG} \leftarrow \overline{LG} - \ell$
  else if $q \xrightarrow{\Omega(s)/a} q' \in C$ for some q' **then**
    if a is not a legal action in s **then**
      $\overline{LG} \leftarrow \overline{LG} - \ell$
    end if
    AND-STEP$_{\mathcal{P},N}(C', q', a, h \cdot \langle q, s, \ell \rangle)$
  else
    non-deterministically choose $a \in \mathcal{A}$ and $q' \in Q$
      $C' \leftarrow C \cup \{q \xrightarrow{\Omega(s)/a} q'\}$
      AND-STEP$_{\mathcal{P},N}(C', q', a, h \cdot \langle q, s, \ell \rangle)$
    if if all non-deterministic branches failed **then**
      $\overline{LG} \leftarrow \overline{LG} - \ell$
    end if
  end if
end function

the FSC.

Every AND node of G is a $\langle q, a, h\rangle$ tuple, and every OR node is a $\langle q, s, p, h\rangle$ tuple, where $q \in Q$, $s \in \mathcal{S}$, $a \in \mathcal{A}$, as previously. $p \in (0, 1]$, and h is a history as defined above. Again, the even levels of G are occupied by AND nodes, and the the odd levels contain only AND nodes. The root of the tree is a special AND node $\langle q_0, \emptyset, \langle\rangle\rangle$, whose only child is an OR node $\langle q_0, s_0, \langle\rangle\rangle$, with s_0 the initial state of $\mathcal{P}$. The successors of an AND node $\langle q^{(t)}, a^{(t-1)}, h^{(t-1)}\rangle$ are all possible tuples $\langle q^{(t)}, s^{(t)}, p^{(t)}, h^{(t-1)}\rangle$ such that $\Delta(s^{(t)} \mid s^{(t-1)}, a^{(t-1)}) = p^{(t)} > 0$. The successors of an OR node are tuples $\langle q^{(t+1)}, a^{(t)}, h^{(t)}\rangle$ for every legal action a in $s^{(t)}$, such that $h^{(t)} = h^{(t-1)} \cdot \langle q^{(t)}, s^{(t)}, (h^{(t-1)})_\ell \cdot p^{(t)}\rangle$, and there is no restriction on $q^{(t+1)}$. ($(h^{(t-1)})_\ell$ denotes the last coordinate of $h^{(t-1)}$.) The main difference to the graph described in section 2.5.1 is the presence of transition probabilities and run likelihoods. For the explanation of the algorithm, we first introduce the probabilistic planning problem called Climber. This example doesn't demonstrate every functionality of PANDOR, but the more complex behaviours (such as backtracking across different branches) behave as in ANDOR-DET, and were explained in the previous chapter.

The Climber problem. The associated story is copied verbatim from (Little and Thiébaux, 2007). *"You are stuck on a roof because the ladder you climbed up on fell down. There are plenty of people around; if you call out for help someone will certainly lift the ladder up again. Or you can try the climb down without it. You aren't a very good climber though, so there is a 40% chance that you will fall and break your neck if you do it alone. What do you do?"*

This environment has 4 states: *roof*, *roof&waiting*, *down&dead*, *down&alive*. The initial state of the problem is *roof*, and the goal state is *down&alive*. There are three actions: `climb-without-ladder`, `climb-with-ladder`, and `call-for-help`. There are two goal runs: one is the result of executing `climb-without-ladder` and has likelihood 0.6; the other executes first `call-for-help` then `climb-with-ladder`, and has likelihood 1. The environment is fully observable, i.e. $\Omega(s) = s$ for all $s \in \mathcal{S}$. The whole AND-OR tree of this problem is illustrated in Fig. 3.1.

The algorithm. We simulate the execution of PANDOR-BASIC with $LG^\star = 0.7$, $\mathcal{P} =$ Climber, $N = 1$. The algorithm starts at the AND node at root of G, with the empty controller C_ε and empty history $\langle\rangle$. Initially, none of the runs are explored, so the lower and upper bounds on the correctness of likelihood for all possible extensions of C_ε is $\underline{LG} = 0.0$ and $\overline{LG} = 1.0$. The tree corresponding to

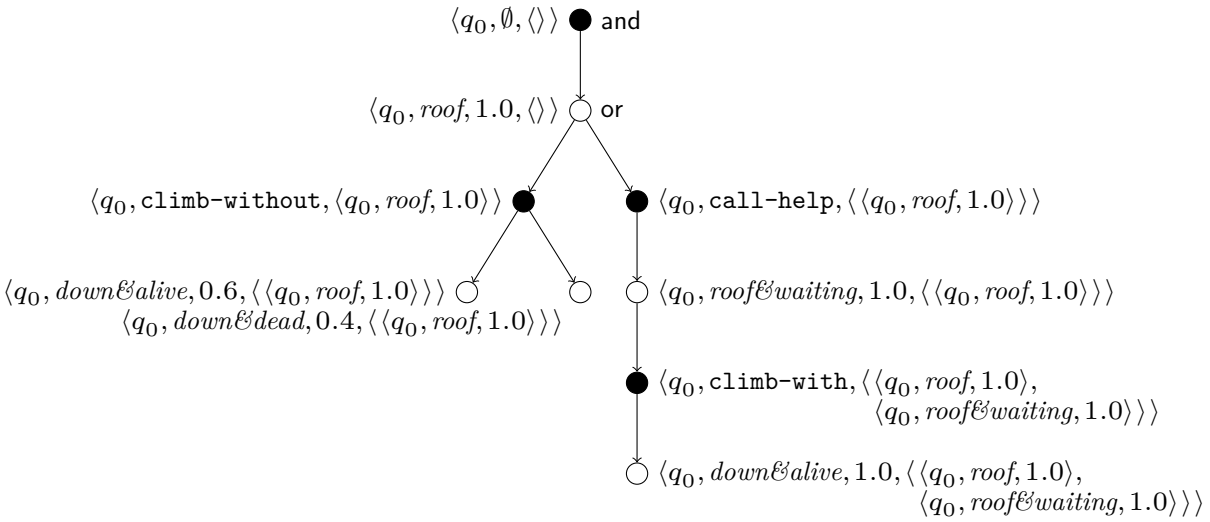


Figure 3.1: The whole AND-OR tree for the Climber problem.

C_ε is shown in Fig. 3.2 (left).

The first AND-STEP calls OR-STEP$(C_\varepsilon, q_0, \mathit{roof}, 1.0, \langle\rangle)$. There is no action defined yet for this observation, so the program state is saved (which now includes the history and $\underline{LG}$ and $\overline{LG}$), and the possible extensions are enumerated. There are two legal actions in *roof*, and as $N = 1$, there is only one possible next controller state (q_0). Suppose the algorithm first chooses the action `climb-without-ladder` for extending the controller, creating $C_{without-ladder}$. The corresponding search tree has now grown to Fig. 3.2 (middle), which the next AND step starts exploring from left to right.

In the OR step of $\langle q_0, \mathit{down\&alive}, 0.6, \langle ... \rangle\rangle$, we are in a goal state, and the likelihood of reaching this point in the graph is 0.6. The lower bound on **LG** is thus increased to 0.6: we know that every controller that is an extension of $C_{without-ladder}$ succeeds with at least 0.6. Execution returns to the AND-step, which sees that the current controller is not yet adequate, so it explores the next OR node. When that OR step recognises that there are no legal actions, it reduces the upper bound on **LG** by the likelihood of the current run: $\overline{LG} \leftarrow 1 - 0.4 = 0.6$. Execution returns to the previous AND step, which sees that $\overline{LG} < LG^\star$, so there is no point in further exploration. (In fact, there are no other runs to explore, and this is also reflected by $\underline{LG} = \overline{LG}$.)

The AND-STEP now backtracks to the last non-deterministic choice point, i.e. to the OR step above. This reverts the last controller transition, restores the values of $\underline{LG}$ and $\overline{LG}$ to the last set of saved values (0 and 1), and explores the

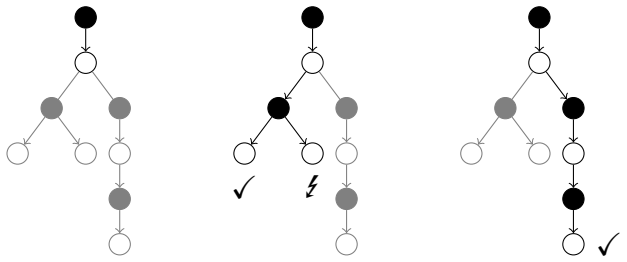


Figure 3.2: AND-OR graphs of different controllers for Climber. Left: C_{ε}. Middle: $C_{climb-without}$. Right: $C_{climb-with}$.

next controller transition. After a few more steps, the environment is in a goal state again, $\underline{LG} \leftarrow \ell = 1.0 > LG^{\star}$, and the adequate controller is returned as a solution.

3.3 Synthesising terminating plans

One issue with this algorithm is that the plans it generates do not necessarily terminate, as they do not execute the `stop` action. Such explicit termination is often required, for example in hierarchical planning, the calling process needs to know when a subprocess ends, so that it can call the next subprocess (Parr and Russell, 1997).

This issue can be fixed by redefining the planning problem $\mathcal{P} = \langle \mathcal{E}, s_0, \mathcal{G} \rangle$ to a new problem $\mathcal{P}' = \langle \mathcal{E}', s_0', \mathcal{G}' \rangle$, so that only after executing the `stop` action in some state $s \in \mathcal{G}$ counts as actually winning.

Specifically, the following changes are required:

- the state space gets two *new* absorbing states: $\mathcal{S}' = \mathcal{S} \cup \{s_{\text{win}}, s_{\text{fail}}\}$,
- the action space is extended with the `stop` action: $\mathcal{A}' = \mathcal{A} \cup \{\texttt{stop}\}$,
- the transition function is changed as such:

$$\Delta'(s' \,|\, s, a) = \begin{cases} \Delta(s' \,|\, s, a) & \text{if } a \neq \texttt{stop} \\ 1 & \text{if } a = \texttt{stop} \text{ and } s \in \mathcal{G} \text{ and } s' = s_{\text{win}} \\ 1 & \text{if } a = \texttt{stop} \text{ and } s \notin \mathcal{G} \text{ and } s' = s_{\text{fail}} \\ 1 & \text{if } s = s' = s_{\text{win}} \text{ or } s = s' = s_{\text{fail}} \\ 0 & \text{otherwise} \end{cases}$$

- $\mathcal{G}' = \{s_{\text{win}}\}$

These changes can be implemented inside the OR-STEP function, as shown in Alg. 3.

Algorithm 3 Changes required in Alg. 2 to generate plans that *terminate explicitly* in goal states.

Require: $LTERPC^\star$: the desired minimum **LTER**$\cdot$**LPC**

function OR-STEP$_{\mathcal{P},N}(C, q, s, p, h)$
  $\ell \leftarrow \text{end}(h).\ell \cdot p$ (or p if $h = \varepsilon$)
  if $s = s_{\text{win}}$ **then**
    $\underline{LTERPC} \leftarrow \underline{LTERPC} + \ell$
  else if $s = s_{\text{fail}}$ **then**
    $\overline{LTERPC} \leftarrow \overline{LTERPC} - \ell$
  else if $\langle q, s, \bullet \rangle \in h$ **then**
    $\overline{LTERPC} \leftarrow \overline{LTERPC} - \ell$
  else if $q \xrightarrow{\Omega(s)/a} q' \in C$ for some q' **then**
    (same as before)
  else
    (same as before)
  end if
end function

Furthermore, the assignment to SP' at line 8 becomes a call to NEXTSTATES$_{\mathcal{P}}(s, a)$, with the new helper function in Alg. 5.

The new algorithm is stated in full in Alg. 4, together with the changes introduced in the next section.

3.4 Correctly counting looping histories

Using the changes in the previous section, one can generate plans with separately specified **LTER** and **LTERPC**, by keeping separate upper and lower bounds for these two values. Whenever the current controller terminates, the lower bound on **LTER** should be increased; whenever a looping history is encountered, the upper bound on **LTER** should be decreased. However, while we were right to discard looping histories in the non-deterministic case with ANDOR-DET, in the presence of probabilities this algorithm would result in an overly cautious controller, namely one whose terminating runs are bounded in length.

Suppose that at an OR-step for the current history $h = \langle \langle q^{(i)}, s^{(i)}, \ell^{(i)} \rangle \rangle_i$ and q, s,ℓ values $\bullet^{(n)}$, there is some k such that $q^{(k)} = q^{(n)}$ and $s^{(k)} = s^{(n)}$. If $\ell^{(n)} = \ell^{(k)}$, then the trajectory between the k-th and n-th steps didn't contain any noisy choices, so continuing the execution of the controller would bring the environment to $\langle q^{(k)}, s^{(k)}, \ell^{(k)} \rangle$ again at every $(n-k)$ steps, never terminating. If $\ell^{(n)} < \ell^{(k)}$, then there are other trajectories from $\langle q^{(k)}, s^{(k)} \rangle$, and these do not necessarily end up at $\langle q^{(k)}, s^{(k)} \rangle$.

- some runs with total likelihood $\alpha_{loop}\ell^{(k)}$ loop back to $\langle q^{(k)}, s^{(k)} \rangle$;
- some runs with total likelihood $\alpha_{goal}\ell^{(k)}$ terminate in a goal state without looping back to $\langle q^{(k)}, s^{(k)} \rangle$;
- some runs with total likelihood $\alpha_{fail}\ell^{(k)}$ terminate not in a state without looping back to $\langle q^{(k)}, s^{(k)} \rangle$;
- some runs with total likelihood $\alpha_{noter}\ell^{(k)}$ never terminate, but do not loop back to $\langle q^{(k)}, s^{(k)} \rangle$.

As these are disjoint sets of runs that all extend $h^{(1:k)}$, and none of these runs extend another, their total likelihood is $\ell^{(k)}$, therefore $\alpha_{loop} + \alpha_{goal} + \alpha_{fail} + \alpha_{noter} = 1$. Notice that of the runs that loop back to $\langle q^{(k)}, s^{(k)} \rangle$, some of them will end up in the goal state (respectively fail or never terminate) without looping back *again* to $\langle q^{(k)}, s^{(k)} \rangle$, with a total likelihood of $\alpha_{goal} \cdot \alpha_{loop}\ell^{(k)}$ (respectively $\alpha_{fail} \cdot \alpha_{loop}\ell^{(k)}$ or $\alpha_{noter} \cdot \alpha_{loop}\ell^{(k)}$). In total, the runs that end up in the goal state after $h^{(1:k)}$, have a likelihood of

- $\alpha_{goal}\ell^{(k)}$ without looping back to $\langle q^{(k)}, s^{(k)} \rangle$;
- $\alpha_{goal}\alpha_{loop}\ell^{(k)}$ with looping back *once* to $\langle q^{(k)}, s^{(k)} \rangle$;
- $\alpha_{goal}^2\alpha_{loop}\ell^{(k)}$ with looping back *twice* to $\langle q^{(k)}, s^{(k)} \rangle$; …
- $\alpha_{goal}^m\alpha_{loop}\ell^{(k)}$ with looping back m times to $\langle q^{(k)}, s^{(k)} \rangle$; and so on.

The sum of these likelihoods is a geometric series, which converges to

$$\alpha_{goal}\ell^{(k)}(1 + \alpha_{loop} + \alpha_{loop}^2 + ...) = \alpha_{goal}\ell^{(k)} \sum_{i=0}^{\infty} \alpha_{loop}^i = \alpha_{goal}/(1-\alpha_{loop})\ell^{(k)}.$$

Similarly for failing runs and non-terminating runs. This means that if we had an oracle at step k that predicted these $\alpha_\bullet$ values, it would update the upper and lower bounds on **LTER** and **LTERPC** as follows:

- increase $\underline{LTER}$ and increase $\underline{LTERPC}$ by $\alpha_{goal}/(1-\alpha_{loop})\ell^{(k)}$;
- increase $\underline{LTER}$ and decrease $\overline{LTERPC}$ by $\alpha_{fail}/(1-\alpha_{loop})\ell^{(k)}$;
- decrease $\overline{LTER}$ and decrease $\overline{LTERPC}$ by $\alpha_{noter}/(1-\alpha_{loop})\ell^{(k)}$.

Lacking an oracle, we can cumulate these values as more and more runs are simulated. The α values for each history length need to be stored separately: let a boldface $\boldsymbol{\alpha}$ denote the $n+1$-long sequence of 4-tuples $\langle \alpha^{(i)}_{loop}, \alpha^{(i)}_{win}, \alpha^{(i)}_{fail}, \alpha^{(i)}_{noter} \rangle$ for $0 \leq i \leq n$ (where n is the length of the current history). In order to revert $\alpha^{(i)}_\bullet$ when backtracking, we need to store $\boldsymbol{\alpha}$ in the backtracking stack when making a non-deterministic choice.

The procedure for updating these values at a history of length n is as follows. Initially, $\alpha^{(\bullet)}_\bullet$ is undefined; $\alpha^{(n)}_\bullet$ is reset to 0 when the first child of an AND node is explored. If the run terminates or loops deterministically, let the relevant $\alpha^{(n)}_x$ value be $p^{(n)}$ (for x either *win*, *fail* or *noter*). If the history loops to step k, increase $\alpha^{(k)}_{loop}$ by $\prod_{i=k+1}^{n} p^{(i)}$. If there are non-deterministic choices to make, then save $\boldsymbol{\alpha}$, and proceed with the AND steps below. When all successors of an AND step at level n are explored, then increase $\alpha^{(n-1)}_x$ by $p^{(n-1)}\alpha^{(n)}_x/(1-\alpha^{(n-1)}_{loop})$ for each $x \in \{win, fail, noter\}$, and reset $\alpha^{(n-1)}_{loop}$ to 0. (Notice that $\alpha^{(n)}_{loop}$ is always 0.) An example trace of the α values for the graph in Fig. 3.3 is given in Table 3.1. (The row marked with $\star$ marks the time when the AND step below node 6 returns.)

Notice that at the end of the execution, the $\alpha^{(1)}_\bullet$ values sum up to 1. Generally, this is true after returning from an AND step only if no runs looped back to above that node.

Now we have all the information to correctly calculate the upper and lower bounds for **LTER** and **LTERPC** at any point during execution. Suppose we are at an AND step at level n, with history h. For each k s.t. $0 \leq k \leq n$, define $\lambda^{(k)}_x$ for $x \in \{win, fail, noter\}$ inductively as follows:

- $\lambda^{(k)}_x := \alpha^{(n)}_x$ if $k = n$;
- $\lambda^{(k)}_x := \alpha^{(k)}_x + p^{(k)}\lambda^{(k+1)}_x/(1-\alpha^{(k)}_{loop})$ for each $0 \leq k < n$.

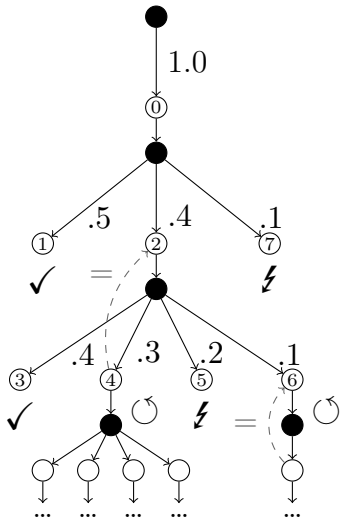


Figure 3.3: The AND-OR tree used for the example calculation of α values. Numbers ① to ⑦ identify the nodes; numbers on the edges are the transition probabilities. ✓ means terminating in a goal state, ↯ means terminating in a non-goal state. ↻ indicates a looping history, with an infinite tree below, and a dashed arrow indicates that the relevant states are equal.

node	$\alpha^{(1)}_{goal}$	$\alpha^{(1)}_{fail}$	$\alpha^{(1)}_{noter}$	$\alpha^{(1)}_{loop}$	$\alpha^{(2)}_{goal}$	$\alpha^{(2)}_{fail}$	$\alpha^{(2)}_{noter}$	$\alpha^{(2)}_{loop}$
0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
1	.5	.0	.0	.0	N/A	N/A	N/A	N/A
2	.5	.0	.0	.0	N/A	N/A	N/A	N/A
3	.5	.0	.0	.0	.4	.0	.0	.0
4	.5	.0	.0	.3	.4	.0	.0	.0
5	.5	.0	.0	.3	.4	.2	.0	.0
6	.5	.0	.0	.3	.4	.2	.1	.0
$\star$	$.5+\frac{.4\cdot.4}{1-.3}$ ≈ 0.73	$0+\frac{.4\cdot.2}{1-.3}$ ≈ 0.11	$0+\frac{.4\cdot.1}{1-.3}$ ≈ 0.06	.0	N/A	N/A	N/A	.0
7	.73	.21	.06	.0	N/A	N/A	N/A	.0

Table 3.1: Trace of the α values for the graph in Fig. 3.3.

Finally, set the lower bound on termination likelihood to $\lambda^{(0)}_{goal} + \lambda^{(0)}_{fail}$; the upper bound to $1 - \lambda^{(0)}_{noter}$. The lower bound on the unnormalised partial correctness likelihood is $\lambda^{(0)}_{goal}$, and the upper bound is $1 - \lambda^{(0)}_{fail} - \lambda^{(0)}_{noter}$.

All of these changes are incorporated in the PANDOR-LOOPY algorithm in Alg. 4, with the helper functions defined in Alg. 5.

3.5 Generalised planning with Pandor

As we saw in section 2.5, the deterministic AND-OR controller search was capable of synthesising a controller that is correct for a generalised planning problem $\overline{\mathcal{P}}$. This is possible with PANDOR too, but with a minor modification.

We cannot simply make the first call to AND-STEP with a set of initial states or an initial probability distribution over states (as it would be customary when solving certain a partially observable *control* problem, see e.g. (Kaelbling et al., 1998)). The reason for this is that the **LPC** values for different initial states would add up: if we were searching for an $\mathbf{LPC} = 0.4$ controller for a problem with two initial states $s^{(0)}_a$ and $s^{(0)}_b$ (each having initial probability 0.5), and the controller were correct for the problem $\langle \mathcal{E}, s^{(0)}_a, \mathcal{G} \rangle$, then after simulating the runs from $s^{(0)}_a$, the algorithm would quit prematurely with $\underline{LTERPC} = 0.5$. This is in conflict with the definition in (Belle and Levesque, 2016): a controller is θ-adequate (e.g. $\mathbf{LPC} \geq \lambda$) for a generalised planning problem $\overline{\mathcal{P}}$ if it is θ-adequate for every basic problem in $\overline{\mathcal{P}}$.

The solution is to search for a plan that is partially correct for *all* initial states with likelihood at least $LTERPC^{\star}$. If this is desired, we can change the first call to AND-STEP, as stated in Alg. 6.

In order for this method to be useful, the enumeration over S_0 must be chosen wisely. Consider the TreeChop problem from section 2.3: a plan that is correct for tree widths $n = 1$ and 2 is not necessarily correct for $n = 2$, e.g. if the plans do not contain loops. On the other hand, most 2-state controllers that are correct for a tree width of 3 are correct for smaller initial tree widths too. In short, the main limitation of this approach is the result of PANDOR being a *blind* search algorithm (Russell and Norvig, 2010): it does not take into account any problem-specific information. This deficiency could be alleviated by guiding the search process by a probabilistic planner that uses a traditional symbolic approach, such as (Rintanen, 2014), or automatically grouping states based on their similarity, as

Algorithm 4 The PANDOR-LOOPY algorithm, which is capable of handling trajectories with repeated environment and controller states.

Require: $\mathcal{P} = \langle \mathcal{E}, s_0, \mathcal{G} \rangle$, a basic planning problem;
Require: N, a bound on the number of controller states.
Require: $LTERPC^\star$: the desired minimum **LTERPC**

function PANDOR-LOOPY-SYNTH($\mathcal{P}, N$)
  (global) $\boldsymbol{\alpha} \leftarrow \langle\rangle$
  return AND-STEP$_{\mathcal{P},N}(C_\varepsilon, 0, \{\langle s_0, 1.0\rangle\}, \langle\rangle)$
end function

function AND-STEP$_{\mathcal{P},N}(C, q, a, h)$
  $n \leftarrow len(h), \quad \alpha^{(n)} \leftarrow \langle 0,0,0,0\rangle$
  $SP' \leftarrow$ NEXTSTATES$_{\mathcal{P}}(s, a)$
  for all $\langle s', p'\rangle \in SP'$ **do**
    $C \leftarrow$ OR-STEP$_{\mathcal{P},N}(C, q, s', p', h)$
    $\boldsymbol{\lambda} \leftarrow$ CALCLAMBDA$(h, \boldsymbol{\alpha})$
    if $\lambda_{goal} \geq LTERPC^\star$ **then**
      return C across the whole call stack
    else if $1 - \lambda_{fail} - \lambda_{noter} < LTERPC^\star$ **then**
      fail this non-deterministic branch
    end if
  end for
  $\alpha_x^{(n-1)} \leftarrow \alpha_x^{(n-1)} + p^{(n-1)}\alpha_x^{(n)}/(1-\alpha_{loop}^{(n-1)}) \qquad$ for $x \in \{win, fail, noter\}$
  $\alpha_{loop}^{(n-1)} \leftarrow 0$
  return
end function

function OR-STEP$_{\mathcal{P},N}(C, q, s, p, h)$
  $\ell \leftarrow \text{end}(h).\ell \cdot p \quad$ (or p if $h = \varepsilon$)
  if $s = s_{\text{win}}$ **then**
    $\alpha_{goal}^{(len(h))} \leftarrow \alpha_{goal}^{(len(h))} + p$
  else if $s = s_{\text{fail}}$ **then**
    $\alpha_{fail}^{(len(h))} \leftarrow \alpha_{fail}^{(len(h))} + p$
  else if $\langle q, s, \ell\rangle \in h$ **then**
    $\alpha_{noter}^{(len(h))} \leftarrow \alpha_{noter}^{(len(h))} + p$
  else if $\langle q^{(k)}, s^{(k)}, \bullet\rangle = \langle q, s, \ell\rangle \in h$ for some k **then**
    $\alpha_{loop}^{(k)} \leftarrow \alpha_{loop}^{(k)} + (\ell/\ell^{(k)})$
  else if $q \xrightarrow{\Omega(s)/a} q' \in C$ for some q' **then**
    AND-STEP$_{\mathcal{P},N}(C', q', a, h \cdot \langle q, s, \ell\rangle)$
  else
    non-deterministically choose $a \in \mathcal{A}$ and $q' \in Q^+(C)$
      $C' \leftarrow C \cup \{q \xrightarrow{\Omega(s)/a} q'\}$
      AND-STEP$_{\mathcal{P},N}(C', q', a, h \cdot \langle q, s, \ell\rangle)$
    if if all non-deterministic branches failed **then**
      $\alpha_{fail}^{(len(h))} \leftarrow \alpha_{fail}^{(len(h))} + p$
    end if
  end if
end function

Algorithm 5 Helper functions used by the PANDOR-LOOPY algorithm.

function NEXTSTATES$_{\mathcal{P}}(s, a)$
  if $a = \texttt{stop}$ and $s \in \mathcal{G}$ **then**
    $SP' \leftarrow \{\langle s_{\text{win}}, 1.0\rangle\}$
  else if $a = \texttt{stop}$ and $s \notin \mathcal{G}$ **then**
    $SP' \leftarrow \{\langle s_{\text{fail}}, 1.0\rangle\}$
  else
    $SP' \leftarrow \{\langle s', p'\rangle \,|\, \Delta(s' \,|\, s, a) = p' > 0\}$ *(as originally)*
  end if
  return SP'
end function

function CALCLAMBDA$(h, \boldsymbol{\alpha})$
  $\lambda_{win} \leftarrow \alpha_{win}^{(len(h))}, \quad \lambda_{fail} \leftarrow \alpha_{fail}^{(len(h))}, \quad \lambda_{noter} \leftarrow \alpha_{noter}^{(len(h))},$
  for $k \leftarrow len(h) - 1 \ldots 0$ **do**
    for all $x \in \{win, fail, noter\}$ **do**
      $\lambda_x \leftarrow \alpha_x^{(k)} + p^{(k)}\,\lambda_x / (1 - \alpha_{loop}^{(k)})$
    end for
  end for
  return $\langle \lambda_{goal}, \lambda_{fail}, \lambda_{noter} \rangle$
end function

Algorithm 6 The modification required for generalised planning with PANDOR.

Require: $\overline{\mathcal{P}} = \langle \mathcal{E}, S_0, \mathcal{G} \rangle$, a generalised planning problem;
Require: N, a bound on the number of controller states.

function PANDOR-GENERALISED-SYNTH$(\overline{\mathcal{P}}, N, LTERPC^{\star})$ $\triangleright\ \overline{\mathcal{P}} = \langle \mathcal{E}, S_0, \mathcal{G} \rangle$
  $C \leftarrow C_{\varepsilon}$
  for $s^{(0)} \in S_0$ **do**
    $\mathcal{P} \leftarrow \langle \mathcal{E}, s^{(0)}, \mathcal{G} \rangle$
    $C \leftarrow$ OR-STEP$_{\mathcal{P}, N}(C, 0, s^{(0)}, 1.0, \langle\rangle)$
    if $\overline{LTERPC}(C, \mathcal{P}) < LTERPC^{\star}$ **then**
      fail
    end if
  end for
end function

done by ARANDA (Srivastava et al., 2008).

The idea for a less wasteful backtracking method in section 2.5.3 would be useful for generalised planning for PANDOR too, for the same reasons as described there. As the search process explores more steps in this algorithm, and the number of explored time steps between two neighbouring backtracks is also greater than earlier, the speed improvement by this technique in many problems would be even larger.

Later we define a class of generalised problems for which the correct plans would generalise from a finite set of basic problems to an infinite set of problems. As this result is independent of the planning algorithm, it is stated in section 3.7.

3.6 Theorem on correctness

In this section we state and prove a theorem about the completeness of the PANDOR-LOOPY search algorithm, i.e. that if there is a an adequate controller, the algorithm will find it.

The machine state and environment state are a sufficient statistic to determine the behaviour of a given FSC in a given environment. In other words, the FSC behaves identically from some combined state $\langle q, s\rangle$, regardless of the preceding history. This we call the *Markov property* of FSCs.

Lemma 1. *Let C be an FSC, $\mathcal{E}$ an environment. For any three histories $h_1, h_2, h' \in (Q \times \mathcal{S})^{<\omega}$, if $end(h_1) = end(h_2)$, and $h1 \cdot h'$ is a valid history, then $C(sensed(h_1 \cdot h')) = C(sensed(h_2 \cdot h'))$.*

Proof sketch. By induction on the length of h', using the behaviour of an FSC as defined in section 2.3. □

Corollary 1. *If $h = h_1 \cdot h_2$ is a valid run of a finite state controller C, and $0 < k < len(h)$, then $\ell(h_1 \cdot h_2) = \ell(h_1) \cdot \ell(\langle end(h_1)\rangle \cdot h_2)$. (Note that the $\cdot$ symbol is used for two distinct operators, namely concatenation and multiplication.)*

The following lemma shows that the lower and upper bound estimates on **LTERPC** are correct.

Lemma 2. *At any point during the execution of* PANDOR-LOOPY *(Alg. 4), $\lambda^{(0)}_{win} \leq$ **LTERPC** $\leq 1 - \lambda^{(0)}_{fail} - \lambda^{(0)}_{noter}$ for the current controller and planning problem, for the λ values returned by* CALCLAMBDA. *The inequalities are strict when not every run of C has been simulated.*

Proof. As the proof is long and does not contain new results other than formalising the ideas in section 3.4, it can be found in the appendix. □

Finally, we have the following theorem, showing that the algorithm is *sound* and *complete*.

Theorem 1. *Let $\mathcal{P}$ be a finite planning problem, N a natural number, and $LTERPC^\star \in (0,1)$. If there exists a finite state controller with at most N states that is* **LTERPC** $\geq LTERPC^\star$*, then the* PANDOR-LOOPY *algorithm will find such a controller.*

Proof sketch. The algorithm simulates the runs of an FSC and those of its extensions. Let C denote the current controller, $hin(Q \times \mathcal{S})^{<\omega}$ the current history, and $n = len(h)$.

Soundness. If h is a valid history of C, then it is a valid history of its extension C', by the definition of a history. This also means that a goal run of C is a goal run of C', and a failing run of C is a failing run of C'. By Lemma 2, if C is discarded because $1-\lambda^{(0)}_{fail}-\lambda^{(0)}_{noter} < LTERPC^\star$, then $1-\lambda^{(0)}_{fail}-\lambda^{(0)}_{noter} <$ **LTER** $< LTERPC^\star$ for C, and also for C'. Thus the algorithm will not return an inadequate controller.

Completeness. Suppose there is a controller C_{good} which is **LTERPC** $\geq LTERPC^\star$. We prove that none of the controllers $C' \preceq C_{good}$ is rejected. C_{good} has finitely many state transitions; proof by induction. The empty controller C_ε has $\lambda^{(0)}_{fail} = \lambda^{(0)}_{noter}$, i.e. $\overline{LTERPC} = 1$, so it is not rejected. Suppose $1-\lambda^{(0)}_{fail}-\lambda^{(0)}_{noter} \geq LTERPC^\star$ for every $C' \prec C_{good}$ with k transitions. If the next controller state results in a C'' s.t. $C' \prec C'' \prec C_{good}$ and $1-\lambda^{(0)}_{fail}-\lambda^{(0)}_{noter} < LTERPC^\star$, that means the sum of likelihoods of the failing or non-terminating runs of C'' is higher than $1-LTERPC^\star$. These runs are also the runs of C_{good}, which would mean that $1-\lambda^{(0)}_{fail}-\lambda^{(0)}_{noter} < LTERPC^\star$ for C_{good} too. By Lemma 2, we would have **LTERPC** $\leq 1-\lambda^{(0)}_{fail}-\lambda^{(0)}_{noter} < LTERPC^\star$ for C_{good}, which is a contradiction.

If C_{good} is not rejected, then by the end of the execution the algorithm has simulated all of its runs, when by Lemma 2, $\lambda^{(0)}_{win} \leq$ **LTERPC**, and C_{good} is returned. □

3.7 One-dimensional probabilistic planning problems

We can identify a class of probabilistic generalised planning problems, where we know certain environment states to be similar to one another. These definitions are similar to the ones of Hu and Levesque (2011), but their statements held for deterministic planning problems defined in the situation calculus.

Intuitively, the state space of such a problem is the Cartesian product of the natural numbers and a finite set, where we call the first coordinate the *planning parameter* of a state. The planning parameter monotonically decreases during a run, and if two states differ only in the planning parameter, then they behave similarly. The following definition formalises this notion:

Definition 15. A generalised planning problem $\overline{\mathcal{P}} = \langle \mathcal{E}, S_0, \mathcal{G} \rangle$ is called a *one-dimensional probabilistic planning problem*, if it fulfils the following conditions:

- The state space $\mathcal{S}$ of its environment is the Cartesian product of the natural numbers and some finite set R (the first coordinate of a state $s = \langle s_n, s_r \rangle$ is called the *planning parameter* at s, and is denoted by $(s)_n := s_n$);
- The state transition function and observations only depend on whether the planning value is zero or nonzero, not on the exact state: if $s_{n,1}, s_{n,2} > 0$ then $\Omega(\langle s_{n,1}, s_r \rangle) = \Omega(\langle s_{n,2}, s_r \rangle)$ and $\Delta(\langle s'_{n,1}, s'_r \rangle | \langle s_{n,1}, s_r \rangle, a) = p$ implies $\Delta(\langle s'_{n,2}, s'_r \rangle | \langle s_{n,2}, s_r \rangle, a) = p$, with $s_{n,1} - s'_{n,1} = s_{n,2} - s'_{n,2}$;
- An action at some state either decreases the planning parameter by 1 or leaves it unchanged: if $\Delta(\langle s'_n, s'_r \rangle \,|\, \langle s_n, s_r \rangle, a) > 0$ then $s'_n \in \{s_n, s_n - 1\}$.
- The initial states are $S_0 = \{\langle s^i_0, s_{0,r} \rangle \,|\, i \in \mathbb{N}\}$ for a given $s_{0,r}$.
- For every goal state, the planning parameter is 0.

The possible dependence on the planning value being zero or nonzero is required to allow for the controller to notice when the problem is solved, and act accordingly (for example terminate). Without it, there would be no difference in observations of goal states and non-goal states.

In a one-dimensional problem, we can prove that if a plan is correct for some finite set of basic problems, then that plan is correct for all instances of the generalised problem.

Theorem 2. *Let $\overline{\mathcal{P}} = \langle \mathcal{E}, \langle s_0^i \rangle_{i\in\mathbb{N}}, \mathcal{G} \rangle$ be a one-dimensional probabilistic planning problem, and C a finite state controller for $\mathcal{E}$. There is a natural number n_0, such that if $\langle \mathcal{E}, \mathcal{C} \rangle$ is* **TER** + **PC** *for every initial value $n \leq n_0$ of the planning parameter, then $\mathcal{C}$ is* **TER** + **PC** *for $\overline{\mathcal{P}}$.*

The intuition of the proof is that for a high enough initial value of the planning parameter, there will be two time steps during execution at which the controller state is identical and the environment state is similar but non-identical. As the next action of a finite state controller depends only on its controller state and the *last* observation, we can freely remove or repeat the action sequence between two such states.

Notation. We say that two environment states s_1 and s_2 are *similar*, and write $s_1 \approx s_2$, whenever $(s_1)_r = (s_2)_r$ and $(s_1)_n > 0$ and $(s_2)_n > 0$. Obviously, $\approx$ is an equivalence relation, i.e. it's reflexive, transitive and symmetric.

We denote a combined state $\langle q, s \rangle$ by qs. For a combined state, $\approx$ extends as such: $qs_1 \approx qs_2$ whenever $q_1 = q_2$ and $s_1 \approx s_2$.

For the proof we use the following lemma.

Lemma 3. *If $h^{(0:t)} = qs^{(0:t)}$ is a non-empty history of the system $\langle \mathcal{E}, C \rangle$ for an FSC C, and $qs'^{(0)} \approx qs^{(0)}$, $(s^{(t)})_n > 0$ and $(s'^{(0)})_n > (s^{(0)})_n - (s^{(t)})_n$, then there is a history h' s.t. $h'^{(0)} = qs'^{(0)}$, and for all $i \leq t$, $h^{(i)} \approx h'^{(i)}$ and $(s'^{(i)})_n - (s'^{(0)})_n = (s^{(i)})_n - (s^{(0)})_n$.*

Proof. By induction on the length of h.

Suppose $len(h) = 1$. Then the history $h' = \langle \langle q^{(0)}, s'^{(0)} \rangle \rangle$ trivially fulfils the criteria.

Suppose the statement is true for every h of length t. We want to construct a $h'^{(0:t)}$ for $h^{(0:t)}$ that fulfils the criteria. By the induction hypothesis, there is a $h'^{(0:t-1)}$ s.t. at the end of this history the system is in controller state $q'^{(t-1)} = q^{(t-1)}$, and environment state $s'^{(t-1)} \approx s^{(t-1)}$. The environment is one-dimensional, so the observation is the same for these two environment states: $\Omega(s'^{(t-1)}) = \Omega(s^{(t-1)})$. The next state of the controller is $\delta(q'^{(t-1)}, \Omega(s'^{(t-1)}) = \delta(q^{(t-1)}, \Omega(s^{(t-1)}) = q^{(t)}$. The next action of the controller is $\gamma(q'^{(t-1)}, \Omega(s'^{(t-1)}) = \gamma(q^{(t-1)}, \Omega(s^{(t-1)}) = a^{(t-1)}$. Let $s'^{(t)}$ be such that $(s'^{(t)})_n -$

$(s'^{(t-1)})_n = (s^{(t)})_n - (s^{(t-1)})_n$ and $(s'^{(t)})_r = (s^{(t)})_r$. So we have:

$$(s'^{(t)})_n - (s'^{(t-1)})_n = (s^{(t)})_n - (s^{(t-1)})_n \quad \text{(definition of } (s'^{(t)})_n) \qquad (\spadesuit)$$
$$(s'^{(t-1)})_n - (s'^{(0)})_n = (s^{(t-1)})_n - (s^{(0)})_n \quad \text{(the ind. hypothesis)} \qquad (\clubsuit)$$
$$(s'^{(t)})_n - (s'^{(0)})_n = (s^{(t)})_n - (s^{(0)})_n \qquad (\spadesuit) + (\clubsuit)$$

Together with the conditions on the planning parameters of the lemma, we can again appeal to the one-dimensionality of $\mathcal{E}$: if $\Delta(s^{(t)} \mid s^{(t-1)}, a^{(t-1)}) > 0$ then $\Delta(s'^{(t)} \mid s'^{(t-1)}, a'^{(t-1)}) > 0$, so h' is a valid history that fulfils the criteria. □

We will use the fact that the planning parameter decreases monotonically and doesn't skip values during a run.

Lemma 4. *Let $\sigma = s^{(0:t_2)}$ be the run of a system for some FSC C and one-dim. problem $\overline{\mathcal{P}}$. If $t_1 < t_2$, $n_1 := (s^{(t_1)})_n$ and $n_2 := (s^{(t_2)})_n$, then for all n' s.t. $n_1 \geq n' \geq n_2$ there is some t' s.t. $t_1 \leq t' \leq t_2$ and $(s^{(t')})_n = n'$.*

Proof sketch. This is easily proven by induction on $t_2 - t_1$, using that an action can leave the planning parameter unchanged or decrease it by one. □

We need one more result, which says that during a long enough goal history, there is a repeated but non-identical combined state.

Lemma 5. *Let C be an FSC for a one-dimensional problem $\overline{\mathcal{P}}$ (with state space $\mathbb{N} \times R$), and $n_0 = |R| \cdot |Q| + 2$. Then any goal history of C in P_{n_0}, there exist t_1 and t_2 s.t. $t_1 < t_2$, $q^{(t_1)} = q^{(t_2)}$, $s^{(t_1)} \approx s^{(t_2)}$, and $(s^{(t_1)})_n > (s^{(t_2)})_n > 0$.*

Proof. From Lemma 4 it follows that for any n, if h is a goal history for C and $\mathcal{P}_n$, then $len(h) \geq n$.

If a history h from $s_0^{n_0}$ is a goal run, then it has at least $n_0 - 1$ *non-identical* combined states $qs^{(t)}$, for which $(s^{(t)})_n > 0$. There are $|R| \cdot |Q|$ possible pairs of $\langle q^{(t)}, (s^{(t)})_r \rangle$. By the pigeonhole principle (Herstein, 1964), there exists a pair of non-identical combined states in h (at time steps t_1 and t_2 for which $q^{(t_1)} = q^{(t_2)}$ and $(s^{(t_1)})_r = (s^{(t_2)})_r$, i.e. $s^{(t_1)} \approx s^{(t_2)}$. As $s^{(t_1)} \neq s^{(t_2)}$ and the planning parameter is non-increasing, we must have $(s^{(t_1)})_n > (s^{(t_2)})_n$. □

Now we are ready to prove Theorem 2.

Proof of Theorem 2. Let N_{fail} denote the set of initial planning parameters n for which C is not **TER**+**PC** for $\mathcal{P}_n$, and let $n_0 = |R| \cdot |Q| + 2$.

↯ Proof by contradiction. Suppose N_{fail} is non-empty. As a non-empty set of natural numbers, N_{fail} has a smallest element (Halmos, 1974); let this smallest element be n_{fail}. We construct an $n''_{fail} < n_{fail}$ for which C is not **TER**+**PC**.

If C is not **TER**+**PC** for $\mathcal{P}_{n_{fail}}$, then it has a history $h^{(0:t)} = \langle qs^{(i)} \rangle_{i=0}^{t}$ that either cannot be extended into a terminating history or terminates in a non-goal state. There must be such a history such that $n_{fail} - (s^{(t)})_n \geq 0$; otherwise by Lemma 5 there would be a history h' from $s_0^{n_0}$ with similar attributes, making C not **TER**+**PC** on $\mathcal{P}_{n_0}$.

By Lemma 3, there is a history $h' = \langle qs'^{(i)} \rangle_{i=0}^{t}$ from $\langle q_0, s_0^{n_0} \rangle$, for which $h^{(i)} \approx h'^{(i)}$ for all $i \leq t$.

By Lemma 5, h' has a pair of repeated combined states, at time steps t_1 and t_2 (with $t_1 < t_2$). This is to say that $h'^{(t_1)} \approx h'^{(t_2)}$; from the previous paragraph we have $h^{(t_1)} \approx h'^{(t_1)}$ and $h^{(t_2)} \approx h'^{(t_2)}$. As $\approx$ is an equivalence relation, $h^{(t_1)} \approx h^{(t_2)}$.

Let $n''_{fail} := n_{fail} - \left((s^{(t_1)})_n - (s^{(t_2)})_n\right)$. Note that as $(s^{(t_1)})_n > (s^{(t_2)})_n$, we have $n''_{fail} < n_{fail}$.

Again by using Lemma 3, we construct a history $h'' = \langle qs^{(i)} \rangle_{i=0}^{t_1}$ starting at $\langle q_0, s_0^{n''_{fail}} \rangle$, such that $h^{(i)} \approx h''^{(i)}$ for all $i \leq t_1$. By the construction of h'' we have $(s''^{(t_1)})_n = (s^{(t_2)})_n$, and from $h^{(t_1)} \approx h^{(t_2)}$, it follows that $h^{(t_2)} = h''^{(t_1)}$.

Suppose C was not **TER** in $\mathcal{P}_{n_{fail}}$. By the Markov property of FSCs (Lemma 1), h can be extended into a terminating history *if and only if* h'' can be extended into one. By definition, this makes C not **TER** in $\mathcal{P}_{n''_{fail}}$, thus n_{fail} is not the smallest element of N_{fail}.

Suppose C was **TER** but not **PC** in $\mathcal{P}_{n_{fail}}$. By the construction of h'', history h terminates in a non-goal state *if and only if* h'' terminates in a non-goal state. This makes C not **PC** in $\mathcal{P}_{n''_{fail}}$. Contradiction. ↯ □

3.8 Discussion

Unexplored runs. Due to the early termination at line 12 of Alg. 2, the total likelihood of unexplored runs is $\overline{LTERPC} - \underline{LTERPC}$ when a successful plan is returned. Among these runs there can be undefined controller actions or unbounded runs. The exact **LTERPC** of the controller is also not known. If one is interested in either of these properties, the early termination can be removed.

Another approach to *approximate* **LPC** is by doing Monte Carlo simulations, in which case one can support the approximations with confidence intervals, as described in (Younes and Musliner, 2002). These approximation methods fall outside the scope of our work, but we think this is a promising avenue for further research.

Limitations. If the number of absorbing non-goal states and goal states together is relatively small in comparison with the size of the state space, our planner suffers from an exponential blowup. In environments where the average number of possible action outcomes is close to one, the runtime only grows exponentially with the average number of *observations* per explored history, with the multiplying factor the number of actions. This is because even on a 100 step long history, if the observations are binary, then the there are at most $|\mathcal{O}| \cdot |Q|$ choices to be made. The runtime grows asymptotically *linearly* in the length of trajectories. (And the history length is at most $|Q| \times |\mathcal{S}|$, as simulations end with repeated combined states.)

If most actions have multiple outcomes and $LTERPC^\star \ll 1$, then PANDOR explores most of the action outcomes before failing on a controller (and its extensions). Each next state brings about potentially multiple next actions, which means a number of histories exponential in their average *length*. The likelihood of these histories diminishes with their length, and as $\overline{LTERPC}$ needs to fall below $LTERPC^\star$, it takes many failed histories before a controller is found inadequate.

Applicability. On some probabilistic planning problems the planner can ignore the probabilities without losing performance. Indeed, this is the approach taken by some successful probabilistic planners, such as FF-replan (Yoon et al., 2007) (which was hence declared to be a "baseline for probabilistic programming"). The investigation of Little and Thiébaux (2007) revealed the circumstances in which this approach is suboptimal, and they called these planning problems *probabilistically interesting*.

Besides probabilistically interesting problems, there is another case where the method of Hu and De Giacomo (2013) fails but ours does not. Specifically, when there are state action pairs for which $\Delta(s|s,a) > 0$, then there are histories with repeated combined states for any controller that takes this action. Often such an action is the optimal one: for example in robotics, noisy action effects are usually modelled as leaving the environment unchanged by chance. This failure of the predecessor algorithm is demonstrated in the Probabilistic WalkThroughFlap

problem in section 4.2.

3.9 Summary

In this chapter, we proposed multiple variants of the PANDOR algorithm, for synthesising finite state controllers that are correct on at least a user-defined likelihood of runs. The PANDOR-BASIC algorithm introduced how to keep track of the multiple runs of a single controller, treating any looping history as failed execution, resulting in a controller whose emulated correct runs are bounded in length. An improvement is the PANDOR-LOOPY algorithm, which correctly counted looping histories by taking into account the probability of looping and the probability of reaching a goal state. Furthermore, we introduced the class of one-dimensional probabilistic planning problems, and proved a theorem under what conditions would a controller generalise to an infinite generalised planning problem.

Our planner comes with theoretical guarantees, which is further strengthened by our result on one-dimensional probabilistic planning problems. Our theorem states that on such a generalised planning problem, a finite state controller – such as one synthesised by PANDOR – need only be correct for some finite subset of basic problems in order to be correct on a countably infinite problems.

In the next chapter we apply our planner to new planning problems, some of them one-dimensional, to see the benefits of this likelihood-based approach.

Chapter 4

Empirical results

While there are examples of iteratively interesting problems in the literature (Bonet et al., 2009; Hu and Levesque, 2009), and there are noisy domains for probabilistic planners (Little and Thiébaux, 2007), most of the work on the topic of planning with loops in noisy environments has been theoretical (Belle and Levesque, 2016; Belle, 2018). This chapter describes the new domains we designed for evaluating PANDOR, and the performance and output of the planner on the first one.

4.1 The BridgeWalk domain

Initially, the agent is standing on the handrail of a bridge: on its left the sidewalk, on its right the river, in front of it n steps away is the goal. With every step taken on the handrail, the agent has a 0.1 probability of falling into the river (which is an absorbing state), and 0.9 probability of moving forward one step. Every other move is deterministic, which include climbing on or off the handrail, or stepping forward on the sidewalk. (Trying to move off the grid leaves the agent in the same state.) The observation space is $\{\text{AtGoal}, \neg\text{AtGoal}\}$, indicating whether the agent is in line with the goal (but potentially on the sidewalk).

The state space of the BridgeWalk(n) environment is an $(n+1)\times 3$ grid world, with the initial state $(n,0)$, the only goal state $(0,0)$. The actions are `(0,1)`, `(0,-1)`, `(-1,0)` (in the figure $\uparrow$, $\downarrow$, $\rightarrow$, respectively), with transition probabilities as described above. (The actions generally act as vector addition.) The observation at state $\langle x,y\rangle$ is AtGoal if $x = n$, and $\neg$AtGoal otherwise. This is illustrated in Fig. 4.1.

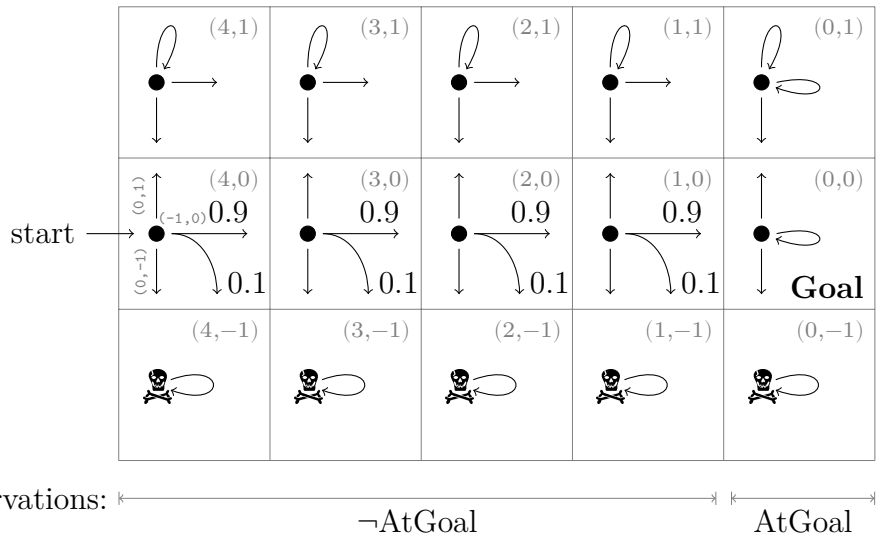


Figure 4.1: The BridgeWalk(4) environment. ☠ indicates a dead end state; other states are marked with ●. For clarity, most action names are omitted.

In this problem, there are multiple goal runs, not every **ONE** plan is **PC**. One run is the result of attempting n forward steps: due to the noisy actions, this plan has 0.9^n probability of ending in the goal, and with $1-0.9^n$ probability it ends in a dead end state (in the river). Another plan moves up on the first step, moving forward n times and moving down when the observation is AtGoal. All runs of these two plans are illustrated in Fig. 4.2.

This figure also shows two FSCs that result in the above runs. Note that the smallest controller with $\mathbf{LTERPC} > 0$ has a single controller state; the smallest controller with $\mathbf{LTERPC} > 0.9^4 \approx 0.66$ has two controller states.

When executing our planner on the problem with the bound on the number of controller states $N = 1$, it returns the simpler controller when the desired **LTERPC** is 0.5 (Fig. 4.2, left), and fails when $\mathbf{LTERPC} \geq 0.9$. For $N = 2$, the synthesised controller depends on multiple factors. The controller and its single run for $\mathbf{LTERPC} \geq 0.99$ are shown in Fig. 4.3 (left). The controller takes the safe path, but it takes an extra step towards the boundary to change state from q_1 to q_0. This is one drawback of our planning method: it does not have a preference for shorter paths, as long as the plan meets the desired correctness.

When planning for $\mathbf{LTERPC} > 0.7$, and the possible actions are enumerated in the order $\rightarrow$, $\uparrow$, $\downarrow$, we see yet another solution (Fig. 4.3, right). The first step is taken towards the goal and changes state, and then proceeds towards the safe path. The agent cannot stay in the initial state on the first step, because it

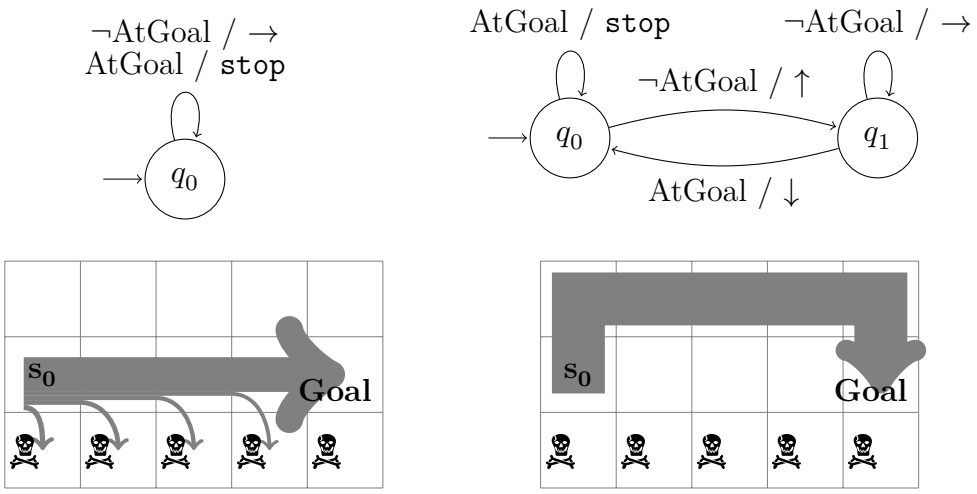


Figure 4.2: Two finite state controllers for the BridgeWalk(4) problem, with all of their runs. Left: controller with $\mathbf{LTERPC} = 0.9^4 \approx 0.66$. Right: controller with $\mathbf{LTERPC} = 1$.

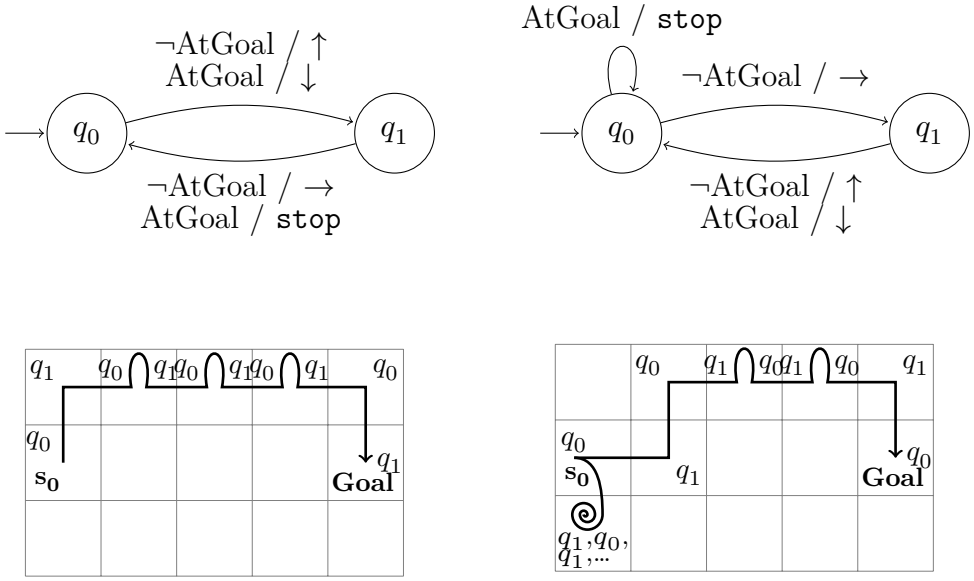


Figure 4.3: Two controllers for the BridgeWalk domain and their runs. Left: $\mathbf{LTERPC} = 1$. Right: $\mathbf{LTERPC} = 0.9$.

receives the same observation on its next step too, so it is bound to take the same action. After enumerating all actions a for the first transition $\langle q_0, \neg\text{AtGoal}\rangle \to \langle q_0, a\rangle$ and seeing that all of them result in too low correctness, the transition on the list is $\langle q_0, \neg\text{AtGoal}\rangle \to \langle q_1, \to\rangle$. While this plan ends in a dead end state with 10% probability, the failing run is not even explored, due to the early termination: the execution ends with $\overline{LTERPC} = 1.0$ and $\underline{LTERPC} = 0.9$.

There is no combination of settings that would result in the "aesthetically pleasing" plan on the right of Fig. 4.2. This would be possible either with a heuristic that prefers actions that bring the environment closer the goal according to some distance metric, or with randomised action and next state selection in the OR steps. As of yet, none of these techniques have been implemented in our planner.

Generalisation of plans. BridgeWalk is a one-dimensional problem, by the definition in section 3.7: the planning parameter is the first coordinate of the state, and states for which the planning parameter is non-zero are "similar". As such, Theorem 2 holds: if 2-state FSC C is correct ($\mathbf{LTERPC} = 1$) for all BridgeWalk(n) with $n \leq |Q| \cdot |R| + 2 = 8$, then it is correct for any BridgeWalk problem. The plan returned for $N \geq 10$ below is of course the same as that synthesised for $N = 4$ below, meaning they generalise to arbitrary initial values of the planning parameter.

Clock time required. We varied the value of n with $\mathbf{LTERPC} \geq .99$ and $N = 2$, and measured the number of OR-steps and backtracks; the results are shown in Table 4.1. The number of explored runs grows superlinearly, but the number of backtracks is sublinear with n, and the latter even plateaus between $n = 50$ and $n = 100$. While the number of backtracks made cannot be more than the number of possible non-isomorphic controllers, which is $(|Q| \cdot |\mathcal{O}|)^{|Q|\cdot(|\mathcal{A}|+1)}/|Q|! = 32768$, we see that in practice it is orders of magnitude smaller. This is thanks to discarding a controller together with all of its extensions, once it has been found inadequate. The time needed to synthesise a plan was under a second even for $N = 100$. We can see that the order in which the actions are enumerated make a noticeable difference. Execution time with the less advantageous action order was 0.8 second for $n = 100$, and 4.2 ms for $n = 4$, on a 2.7 GHz Intel Core i5 machine from 2014.

Similarity to domains from the literature. Our domain looks similar to the Cliff Walking domain in (Sutton and Barto, 2018, Example 6.6), which

	Action order: $\rightarrow,\uparrow,\downarrow$		Action order: $\uparrow,\rightarrow\downarrow$	
n	# backtracks	# steps	# backtracks	# steps
4	270	323	96	115
10	300	389	102	133
20	350	499	112	163
50	617	3990	168	1207
100	717	24756	168	7415

Table 4.1: The number of backtracks and steps as a function of n on BridgeWalk(n).

was used to demonstrate the differences between the on-policy control method Sarsa and the off-policy method Q-learning in reinforcement learning. The key difference is that in Cliff Walking the environment is deterministic while the action selection is stochastic (for Sarsa), while in BridgeWalk the environment is noisy and the action selection is deterministic. In Cliff Walking, the agent receives a -1 reward per step, so the optimal path is the shorter straightforward one, in contrast with our setting.

4.2 Probabilistic WalkThroughFlap

The Probabilistic WalkThroughFlap problem was designed to test how the planner reacts to the presence of repeating combined states.

This problem is shown in Fig. 4.4. The state space of the environment is a 4×1 grid world with states $\{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}\}$, plus an initial state $\star$. The observation space is $\{\star, \mathsf{false}, \mathsf{true}\}$, with $\Omega(\star) = \star$, $\Omega(\mathbf{1}) = \Omega(\mathbf{2}) = \mathsf{false}$, $\Omega(\mathbf{0}) = \Omega(\mathbf{3}) = \mathsf{true}$. State $\star$ has a single action possible, `start`, which leads to state **1** with probability 0.6 and to **2** with probability 0.4. Executing `left` (or `right`) in **1** changes state deterministically to **0** (or **2**), and `right` in **2** moves to **3**. Executing `left` in **2** leads back to **2** with 0.1 probability and to **1** with 0.9 probability. Goal states are **0** and **3**.

This problem has a solution with single-state controllers, which executes `start` in $\star$, either `left` or `right` when the observation is false, and executes `stop` when the observation is true (when the agent is in a goal state).

Our planner was able to synthesise the correct plan that attempts the action `left`, and the α values during execution were as prescribed in section 3.4, resulting in $\underline{LTERPC} = 1.0$, despite the outcome of `left` from **2** leaves the environment

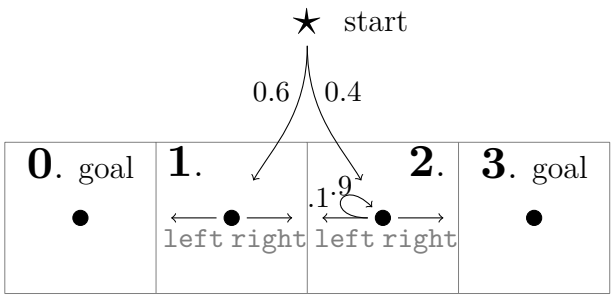


Figure 4.4: The Probabilistic WalkThroughFlap problem.

in place.

4.3 The Probabilistic Halls domains

The Halls problem suite was introduced by Bonet et al. (2009), for the purpose of evaluating their planner that synthesised finite state controllers. They described the planning problems Hall-A($1 \times n$), Hall-A($n \times m$), which we adapted to include probabilistic action effects.

ProbHall-A($1 \times n$). The $1 \times n$ variant of the ProbHall-A problem is a $1 \times n$ grid world. Initially, the agent starts in the leftmost cell (marked with A), and its goal is to visit the rightmost cell marked B and go back to cell A. The observation space is $\{A, B, -\}$, indicating whether it is currently in cell A, cell B, or neither. The two possible actions are $\rightarrow$ and $\leftarrow$, which move the agent in the desired direction with 0.9 probability, and leave it in place with 0.1 probability. (Originally both actions had deterministic effects.) Moving against the boundary of the world leaves the agent in place.

The state space for ProbHall-A($1 \times n$) is thus $\{1, 2, 3, \ldots, n\} \times \{\neg\text{visB}, \text{visB}\}$, where the first coordinate is the current position of the agent, and the second is whether it has visited cell B. The single goal state is thus $\langle 1, \text{visB} \rangle$.

ProbHall-A($n \times n$). The $n \times n$ version is a combination of four $1 \times n$ environments joined at the corners, where the cells form a square shape. The four observables now are the four corners A, B, C, D, and the goal is to visit all four

of these corners. Formally, the state space is

$$\begin{aligned}\mathcal{S} = \{\text{top}, \text{right}, \text{bottom}, \text{left}\} \times \{1, 2, \ldots, n-1\} \times \\ \{\neg\text{visA}, \text{visA}\} \times \{\neg\text{visB}, \text{visB}\} \times \\ \{\neg\text{visC}, \text{visC}\} \times \{\neg\text{visD}, \text{visD}\} \times\end{aligned}$$

where the first coordinate indicates which sub-hall the agent is occupying currently, the second is the position inside this sub-hall, and the third to sixth is whether the agent has visited cells A, B, C or D. The observation is $\{A, B, C, D, -\}$, according to whether the agent is currently in any of the corner cells (or in neither of them).

The initial state is $s_0 = \langle \text{top}, 1, \neg\text{visA}, \neg\text{visB}, \neg\text{visC}, \neg\text{visD} \rangle$. In this variant we have multiple goal states: it doesn't matter where exactly the agent terminates. The set of goal states is thus

$$\mathcal{G} = \{s \in \mathcal{S} \,|\, (s)_3 = \text{visA}, (s)_4 = \text{visB}, (s)_5 = \text{visC}, (s)_6 = \text{visD}\}.$$

The minimal number of states required for this problem is $N = 2$: the observation in a given cell when the agent is moving from A to B is the same as the observation in that cell when it is moving in the opposite direction. With $N = 1$, the controller state and observation doesn't contain sufficient information to decide which action to execute next.

Hu and De Giacomo's planner is not able to solve any of the ProbHall-A problems. This is due the noisy action effects: executing any action in cell A will leave the agent in cell A in one outcome, and the planner cannot tell that retrying this execution brings down the failure of probability to exactly zero. No matter which next state it chooses on the first step, there will be a history which fails due to a repeated combined state in at most two steps.

Chapter 5

Conclusion

The central topic of this thesis was the synthesis of plans with loops for probabilistic planning problems. We proposed a new planning algorithm, PANDOR, which uses finite state controllers to represent such a loopy plan. The algorithm is the probabilistic extension of the AND-OR planner of (Hu and De Giacomo, 2013), which works by doing a depth-first search in the space of possible finite state controllers, and simulating the runs of these controllers parallel to extending them. In contrast to its predecessor, our planner is capable of synthesising finite state controllers for finite discrete stochastic domains, by keeping an account of the likelihoods of the simulated runs. The controllers synthesised by PANDOR are not necessarily optimal, but their likelihood of correctness is guaranteed to exceed some user-defined limit. This property is strengthened by our result which says that in a one-dimensional probabilistic planning problem, if an FSC is adequate only for some finite number of basic problems, then that controller is adequate for an infinite basic problems. We also proposed new probabilistic planning problems, and evaluated our planner on some of these problems. We saw that PANDOR was able to produce plans for problems for which its predecessor could not, thanks to its ability to track the likelihoods of plans and thus account for looping histories correctly.

5.1 Further work

Until automated planners can synthesise generalised plans as efficiently and reliably as humans can, even in the presence of noisy actions, our work is not finished. To advance research in this field, we see three broad avenues for further research.

Correctness guarantees. We showed that correct plans generalise to certain infinite generalised planning problems, if the problem is one-dimensional. We are interested in making stronger claims along this line: either relaxing the criterion that every run must be a terminating goal run, or to extend the requirements posed on the domain to generalise. We conjecture the following for relaxing 100% correctness:

Conjecture. *Let C be an FSC and $\overline{\mathcal{P}}$ a one-dimensional problem. There is a natural number n_0 such that if C is* $\mathbf{LTERPC} \geq \lambda$ *for every problem $\mathcal{P}_n$ with $n \leq n_0$, and C is* $\mathbf{LTERPC} = \lambda$ *on both $\mathcal{P}_{n_0}$ and $\mathcal{P}_{n_0-1}$, then C is* $\mathbf{LTERPC} \geq \lambda$ *for $\overline{\mathcal{P}}$.*

For relaxing the definition of one-dimensional problems, we expect that similar results could be proven even if some actions can increase the planning parameter, or if the definition allows for situation-dependent fluents through the use of situation calculus, similar to the original definition of Hu and Levesque (2011). For this work, strong theoretical foundations of planning with loops and noise, such as the results of Belle (2018), will be necessary.

More complex planning domains. In order to test the boundaries of planners like ours, we call for designing more interesting and complex planning problems, that are both iteratively and probabilistically interesting. PANDOR has its limitations, and while it solved some of the problems from the previous chapter efficiently, it might fail on many domains that are trivial to solve for other planners.

More efficient planning methods. Finally, we could extend PANDOR with techniques that have already proven efficient for probabilistic planning or generalised planning. For probabilistic planning, we could guide the search process with the use of a symbolic probabilistic planner (Rintanen, 2014), or instead of maintaining precise estimates on the lower and upper bound of correctness, we could maintain confidence intervals through statistical approaches and sampling trajectories (Younes and Musliner, 2002). For generalised planning, state abstraction techniques could automatically recognise "similar" states: the planners of ARANDA (Srivastava et al., 2008) is a planner that utilised this method successfully, and its plans are guaranteed to generalise in so-called Extended-LL domains. Such methods could allow PANDOR to exploit the structure of the planning problem more efficiently.

Appendix A

Proof about correctness

Lemma 6. *At any point during the execution of* PANDOR-LOOPY *(Alg. 4),* $\lambda^{(0)}_{win} \leq$ **LTERPC** $\leq 1 - \lambda^{(0)}_{fail} - \lambda^{(0)}_{noter}$ *for the current controller and planning problem, for the* λ *values returned by* CALCLAMBDA. *The inequalities are strict when not every run of* C *has been simulated.*

Proof. The algorithm simulates the runs of an FSC and those of its extensions. (As per the convention that an undefined controller transition equals to a `stop` action, every FSC C has an equivalent extension C', where if $C(q,o) = \mathsf{undef}$ then $C'(q,o) = (q, \texttt{stop})$.) Let C denote the current controller, $h \in (Q \times \mathcal{S})^{<\omega}$ the current history, and $n = len(h)$.

Define the sets $H^{(k)}_{win}$, $H^{(k)}_{loop}$ as follows:

$$
\begin{aligned}
H_{all} &= \{h' \in (Q \times \mathcal{S})^{<\omega} \,|\, h' \text{ is a valid history of } C \text{ in } \mathcal{E} \text{ from } \langle q_0, s_0 \rangle\} \\
H_{win} &= \{h' \in H_{all} \quad\quad |\, \mathcal{C}(h') = \texttt{stop} \text{ and } (end(h'))_s \in \mathcal{G} \\
H^{(k)}_{win} &= \{h' \in (Q \times \mathcal{S})^{<\omega} \,|\, h^{(:k-1)} \cdot h' \in H_{win} \text{ and } h' \cap h^{(:k-1)} = \emptyset\} \\
H^{(k)}_{loop} &= \{h' \in (Q \times \mathcal{S})^{<\omega} \,|\, h^{(:k)} \cdot h' \in H_{all} \text{ and } end(h') = h^{(k)} \text{ and} \\
&\qquad\qquad h^{(:k)} \cap h'^{(:len(h')-1)} = \emptyset\}
\end{aligned}
$$

$H^{(k)}_{win}$ is the set of histories from $h^{(k)}$ that terminate in the goal and do not repeat any combined state in $h^{(:k)}$; and $H^{(k)}_{loop}$ is the set of histories from $h^{(k)}$ that end in $h^{(k)}$ but otherwise do not contain any combined state from $h^{(:k)}$. Furthermore, let $H_{expd} \subset H_{all}$ denote the histories of C that have already been explored by the algorithm.

We prove by induction that at any time,

$$\lambda_{win}^{(k)} \leq \sum_{h_{win} \in H_{win}^{(k+1)}} \ell(\langle h^{(k)} \rangle \cdot h_{win})$$

for any k s.t. $0 \leq k \leq n$. (Where we treat $h^{(:-1)}$ as the empty sequence $\langle\rangle$.)

This is true for the base case, $k = n$:

$$\begin{aligned}\lambda_{win}^{(n)} \overset{def}{=} \alpha_{win}^{(n)} &= \sum_{\substack{\{h' \in H_{win}^{(n)} \cap H_{expd} \mid \\ h'^{(0)} \neq h^{(n+1)}\}}} \ell(\langle h^{(n)} \rangle \cdot h') \\ &= \sum_{\{h' \in H_{win}^{(n)} \cap H_{expd}\}} \ell(\langle h^{(n)} \rangle \cdot h') \leq \sum_{\{h' \in H_{win}^{(n)}\}} \ell(\langle h^{(n)} \rangle \cdot h'),\end{aligned}$$

together with the definition of $H_{win}^{(k)}$. Suppose the statement holds for $k+1$.

By examining the program state when $\boldsymbol{\alpha}$ is changed, it is easy to see that the following statements always hold:

$$\begin{aligned}\alpha_{loop}^{(k)} &= \sum_{\substack{\{h' \in H_{loop}^{(k)} \cap H_{expd} \mid \\ h'^{(0)} \neq h^{(k)}\}}} \ell(\langle h^{(k)} \rangle \cdot h'), \\ \alpha_{win}^{(k)} &= \sum_{\substack{\{h' \in H_{win}^{(k)} \cap H_{expd} \mid \\ h'^{(0)} \neq h^{(k+1)}\}}} \ell(\langle h^{(k)} \rangle \cdot h').\end{aligned}$$

For each $-1 \leq k \leq len(h)$, the following is true:

$$\sum_{\substack{\{h' \in H_{win} \mid \\ h'^{(:k-1)} = h^{(:k-1)}, \\ h'^{(k:)} \cap h^{(:k-1)} = \emptyset\}}} \ell(h') = \sum_{\substack{\{h' \in H_{win} \mid \\ h'^{(:k-1)} = h^{(:k-1)}, \\ h'^{(k:)} \cap h^{(:k-1)} = \emptyset \\ h'^{(k)} \neq h^{(k)}\}}} \ell(h') + \sum_{m=0}^{\infty} \sum_{\substack{\{h' \in H_{win} \mid \\ h'^{(:k)} = h^{(:k)}, \\ h'^{(k:)} \cap h^{(:k-1)} = \emptyset \\ h'^{(k:)} \text{ contains } h^{(k)} \ m+1 \text{ many times}\}}} \ell(h'), \qquad (\star)$$

as from $h^{(:k)}$ the controller can continue either without $h^{(k)}$ as its next state, or going through $h^{(k)}$ exactly $m+1$ times, for $m = 0, 1, 2, ...$

We use Corollary 1 for the inner sum on the right side of ($\star$) (calculation for

$m = 1$):

$$\begin{aligned}
\sum_{\substack{\{h' \in H_{win} \mid \\ h'^{(:k)} = h^{(:k)}, \\ h'^{(k:)} \cap h^{(:k-1)} = \emptyset \\ h'^{(k:)} \text{ contains } h^{(k)} \text{ twice}\}}} \ell(h') &= \sum_{h_{loop} \in H_{loop}^{(k)}} \sum_{h_{win} \in H_{win}^{(k+1)}} \ell(h^{(:k)} \cdot h_{loop} \cdot h_{win}) \\
&= \sum_{h_{loop} \in H_{loop}^{(k)}} \sum_{h_{win} \in H_{win}^{(k+1)}} \Big(\ell(h^{(:k)})\ \ell(\langle h^{(k)} \rangle \cdot h_{loop})\ \ell(\langle h^{(k)} \rangle \cdot h_{win}) \Big) \\
&= \ell(h^{(:k)}) \sum_{h_{loop} \in H_{loop}^{(k)}} \ell(\langle h^{(k)} \rangle \cdot h_{loop}) \sum_{h_{win} \in H_{win}^{(k+1)}} \ell(\langle h^{(k)} \rangle \cdot h_{win}) \\
&\geq \ell(h^{(:k)})\, \alpha_{loop}^{(k)} \lambda_{win}^{(k+1)}.
\end{aligned}$$

The first equality holds because the elements of $H_{loop}^{(k)}$ and $H_{win}^{(k)}$ do not contain repeated combined states (even when prepended with $h^{(:k)}$), so if h_1 and h_2 are elements of either of these sets, then $h_1 \cdot h_2$ is not. In the second line we used the fact that $end(h_{loop}) = h^{(k)}$.

The last inequality holds because of the induction hypothesis for k and

$$\sum_{h_{loop} \in H_{loop}^{(k)}} \ell(\langle h^{(k)} \rangle \cdot h_{loop}) \geq \sum_{\substack{h_{loop} \in \\ H_{loop}^{(k)} \cap H_{expd}}} \ell(\langle h^{(k)} \rangle \cdot h_{loop}) = \alpha_{loop}^{(k)}.$$

For arbitrary m, the right side generalises to

$$\ell(h^{(:k)}) \left(\alpha_{loop}^{(k)}\right)^m \alpha_{win}^{(k+1)} = \ell(h^{(:k-1)})\, p^{(k)} \left(\alpha_{loop}^{(k)}\right)^m \alpha_{win}^{(k+1)}.$$

Now we have the following inequality from equation ($\star$):

$$\begin{aligned}
(\star) &\geq \ell(h^{(:k-1)})\, \alpha_{win}^{(k)} + \sum_{m=0}^{\infty} \ell(h^{(:k-1)})\, p^{(k)} \left(\alpha_{loop}^{(k)}\right)^m \lambda_{win}^{(k+1)} \\
&= \ell(h^{(:k-1)}) \left(\alpha_{win}^{(k)} + p^{(k)} \frac{1}{1 - \alpha_{loop}^{(k)}} \lambda_{win}^{(k+1)} \right) \overset{def}{=} \ell(h^{(:k-1)})\, \lambda_{win}^{(k)}.
\end{aligned}$$

By Corollary 1, the left-hand side of equation ($\star$) is equal to

$$\ell(h^{(:k-1)}) \cdot \sum_{h_{win} \in H_{win}^{(k)}} \ell(\langle h^{(k-1)} \rangle \cdot h_{win}),$$

which proves the inductive statement for $\lambda_{win}^{(k)}$.

Finally, apply this result for $k = 0$. Notice that $H^{(0)}_{win}$ is the set of histories that terminate in the goal if the system is started in the initial state $\langle q_0, s_0 \rangle$, and

$$\lambda^{(0)}_{win} \leq \sum_{h' \in H^{(0)}_{win}} \ell(\langle\rangle \cdot h') \stackrel{def}{=} \mathbf{LTERPC}.$$

The proofs for $\lambda^{(k)}_{noter}$ and $\lambda^{(k)}_{fail}$ are almost identical. They lead to

$$\mathbf{LTERPC} \leq 1 - \lambda^{(0)}_{fail} - \lambda^{(0)}_{noter},$$

which completes the proof. □

Bibliography

Babcsányi, I. (2000). Equivalence of Mealy and Moore Automata. *Acta Cybernetica*, 14:541–552.

Belle, V. (2018). On Plans With Loops and Noise. In André, E., Koenig, S., Dastani, M., and Sukthankar, G., editors, *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018*, pages 1310–1317. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM.

Belle, V. and Levesque, H. J. (2016). Foundations for Generalized Planning in Unbounded Stochastic Domains. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016*, pages 380–389. AAAI Press.

Bonet, B., Palacios, H., and Geffner, H. (2009). Automatic Derivation of Memoryless Policies and Finite-State Controllers Using Classical Planners. In Gerevini, A., Howe, A. E., Cesta, A., and Refanidis, I., editors, *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*. AAAI.

Boolos, G. S., Burgess, J. P., and Jeffrey, R. C. (2007). *Computability and Logic*. Cambridge University Press, 5 edition.

Fikes, R. and Nilsson, N. J. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In Cooper, D. C., editor, *Proceedings of the 2nd International Joint Conference on Artificial Intelligence. London, UK, September 1-3, 1971.*, pages 608–620. William Kaufmann.

Ghallab, M., Nationale, E., Aeronautiques, C., Isi, C. K., (sri, D. W., Barrett, A., Christianson, D., Friedman, M., Kwok, C., Golden, K., Penberthy, S., Smith,

D. E., Sun, Y., Weld, D., and Mcdermott, C. D. (1998). PDDL – The Planning Domain Definition Language.

Halmos, P. R. (1974). *Naive Set Theory*. Springer.

Herstein, I. N. (1964). *Topics in Algebra*. Waltham: Blaisdell Publishing Company.

Hoffmann, J. and Nebel, B. (2001). The FF Planning System: Fast Plan Generation Through Heuristic Search. *J. Artif. Intell. Res.*, 14:253–302.

Hu, Y. and De Giacomo, G. (2013). A Generic Technique for Synthesizing Bounded Finite-State Controllers. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013*. AAAI.

Hu, Y. and Levesque, H. (2009). Planning with Loops: Some New Results. In *Workshop on Generalized Planning, ICAPS 2009*.

Hu, Y. and Levesque, H. J. (2011). A Correctness Result for Reasoning about One-Dimensional Planning Problems. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, 2011*, pages 2638–2643. IJCAI/AAAI.

Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1998). Planning and Acting in Partially Observable Stochastic Domains. *Artificial Intelligence*, 101(1-2):99–134.

Levesque, H. J. (2005). Planning with Loops. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, UK, 2005*, pages 509–515. Professional Book Center.

Levesque, H. J., Pirri, F., and Reiter, R. (1998). Foundations for the Situation Calculus. *Electron. Trans. Artif. Intell.*, 2:159–178.

Little, I. and Thiébaux, S. (2007). Probabilistic planning vs. replanning. *Workshop, ICAPS 2007*.

Mealy, G. H. (1955). A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079.

Moore, E. F. (1956). Gedanken-Experiments on Sequential Machines. In *Automata Studies*, pages 129–153. Princeton U.

Parr, R. and Russell, S. J. (1997). Reinforcement Learning with Hierarchies of Machines. In Jordan, M. I., Kearns, M. J., and Solla, S. A., editors, *Advances in Neural Information Processing Systems 10, [NIPS Conference, Denver, Colorado, USA, 1997]*, pages 1043–1049. The MIT Press.

Poupart, P. and Boutilier, C. (2003). Bounded Finite State Controllers. In Thrun, S., Saul, L. K., and Schölkopf, B., editors, *Advances in Neural Information Processing Systems 16 [Neural Information Processing Systems, NIPS 2003, December 8-13, 2003, Vancouver and Whistler, British Columbia, Canada]*, pages 823–830. MIT Press.

Rintanen, J. (2014). Madagascar: Scalable planning with SAT. In *Proceedings of the 8th International Planning Competition (IPC-2014)*.

Russell, S. J. and Norvig, P. (2010). *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education.

Srivastava, S. (2010). *Foundations and Applications of Generalized Planning*. PhD thesis, University of Massachusetts Amherst.

Srivastava, S., Immerman, N., and Zilberstein, S. (2008). Learning Generalized Plans Using Abstract Counting. In Fox, D. and Gomes, C. P., editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 991–997. AAAI Press.

Srivastava, S., Zilberstein, S., Gupta, A., Abbeel, P., and Russell, S. J. (2015). Tractability of Planning with Loops. In Bonet, B. and Koenig, S., editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 3393–3401. AAAI Press.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction (second edition)*. The MIT Press.

Yoon, S. W., Fern, A., and Givan, R. (2007). FF-Replan: A Baseline for Probabilistic Planning. In Boddy, M. S., Fox, M., and Thiébaux, S., editors, *Proceedings of the Seventeenth International Conference on Automated Planning and*

Scheduling, ICAPS 2007, Providence, Rhode Island, USA, September 22-26, 2007, page 352. AAAI.

Younes, H. L. and Littman, M. L. (2004). PPDDL1.0: An extension to PDDL for expressing planning domains with probabilistic effects. In *Proc. International Planning Competition*.

Younes, H. L. S. and Musliner, D. J. (2002). Probabilistic Plan Verification through Acceptance Sampling. In *In Proceedings of the AIPS 2002 Workshop on Planning via Model Checking*, pages 81–88. AAAI Press.