# Replanning with Guarantees for Planning under Uncertainty

**Johannes Schmalz**

A thesis presented for the degree of
Bachelor of Adv. Computing (Honours)

College of Engineering & Computer Science
Australian National University

# Declaration

This thesis is an account of research undertaken between January 2020 and January 2021 at the ANU College of Engineering & Computer Science.

Except where acknowledged in the customary manner, the material presented in this thesis is, to the best of my knowledge, original and has not been submitted in whole or part for a degree in any university.

---

Johannes Schmalz

January 2021

# Acknowledgements

I want to thank first and foremost my supervisor Felipe Trevizan for his guidance, support, patience, and flexibility throughout the challenges that were presented by the project, and the numerous difficulties of 2020. His hand as supervisor ensured an interesting and enjoyable research experience.

I would also like to thank Jochen Renz for organising an extension for honours research projects that were in progress during 2020.

# Abstract

Probabilistic planning problems describe systems where an agent must traverse a set of states by selecting actions with stochastic outcomes. Stochastic shortest path problems additionally have goal states that should be reached with expected minimal cost. Some of the most successful approaches for solving these problems convert the probabilistic problem into a deterministic relaxation, and solve the determinisation with existing efficient deterministic solvers. Due to this approximation, these algorithms can not give guarantees of optimality, or even that following the solution will necessarily lead to a goal.

Operations research has developed algorithms for dealing with large, complex problems with mathematical guarantees. In particular, the column generation technique enables us to reduce problems into simpler subproblems that can be solved efficiently. For instance, vehicle routing problems can be solved by iteratively solving shortest path problems as subproblems within the column generation framework.

This thesis aims to simplify probabilistic planning problems into deterministic planning problems akin to the existing approaches from the planning community, but in the disciplined framework of column generation; thereby leveraging the performance of the former, and the mathematical guarantees of the latter.

Our contribution is the PBColgen algorithm, which relies on column generation to construct deterministic planning problems. These deterministic subproblems are solved to either improve the current stochastic solution in the next iteration, or to prove that the current stochastic solution is optimal. PBColgen guarantees that it will converge to the optimal solution in finite time, and exhibits reasonable anytime behaviour.

# Contents

# Chapter 1

# Introduction

Consider problems in which an agent must find its way through a maze, or stack blocks on top of each other, or construct a schedule for a Mars rover such that the rover remains safe but explores as much as possible; these are all instances of planning problems. This thesis focuses on a particular flavour of planning problems which allows actions to have probabilistic effects. We explore how operations research techniques can be combined with approaches from the planning community, and present an algorithm that can combine desirable traits of both.

## 1.1 Planning

Planning problems consist of a discrete, finite set of states, which an agent must navigate by applying actions. Typically, we are interested in the agent optimising its path through the state space by some criteria – for instance, we might be interested in the shortest path through a maze. This enables us to model many practical problems, ranging from GPS navigation to deriving a schedule for autonomous robots.

From the start, the planning community has been interested in deterministic planning problems which have the property that applying an action will always yield a unique, predictable outcome. However, many problems have an element of chance, and thus do not satisfy this property. Therefore, it is natural to generalise deterministic planning problems into stochastic planning problems – the focus of this thesis.

The addition of probabilistic effects at first seems innocuous, but in fact it introduces a few complications. First of all, an optimal solution needs to

consider all possible effects of each action, which quickly covers a much larger state space than the solution of a deterministic problem. More than that, we can no longer rely on properties that have made deterministic planners so effective, such as assuming a state will never be revisited.

## 1.2 Stochastic Planning with Deterministic Planners

One approach that has shown to be effective for dealing with the complexity of stochastic planning problems is by relaxing them into deterministic planning problems – which we can deal with more easily. There is a large choice of efficient deterministic solvers to choose from to solve these relaxations, and this family of solvers delivers impressive performance. However, deterministic relaxations inherently lose information, so that the planner can no longer account for probabilistic effects. The result is that such solvers do not take probabilities in consideration, and tend to prefer solutions that are cheap regardless of their likelihood of failing.

## 1.3 Linear Programming in Stochastic Planning

Linear programming is a powerful and flexible tool for optimising variables under linear constraints. This thesis is particularly concerned with a linear programming technique called column generation. Column generation allows us to solve large and complex problems by breaking them down into subproblems, and then recombining these into a solution for the original problem.

Linear programming has already been used by the planning community to develop solvers for stochastic planning problems. However, these formulations work by considering the relationships between individual states and their applicable actions. There is no work that we are aware of at the time of writing, that uses plans for deterministic problems to construct an optimal solution for stochastic problems.

## 1.4 Goals and Contributions

The goal of this project was to review the relevant techniques in planning and operations research, and combine this knowledge into an elegant linear program to encode stochastic planning problems in a way that is amenable to column generation – so that the subproblems reduce to deterministic planning problems. Once this was in place, we aimed to turn the theoretical formulation into a practical algorithm. Our goals can be summarised in the following research question:

*Can column generation be used to build an effective replanner with optimality guarantees?*

Our contribution is the PBColgen algorithm, which is built on an intuitive and concise linear program that drives a column generation algorithm. This algorithm reduces stochastic shortest path problems to relatively simple subproblems, and guarantees optimality.

## 1.5 Thesis Outline/Structure

- Chapter 2. The purpose of this chapter is to introduce the terminology and models of the planning community, which will be used throughout this thesis.
- Chapter 3. In this chapter we first introduce and discuss some classic algorithms for stochastic planning problems. Then, we introduce additional terminology that helps us discuss the remaining algorithms and our contribution. The remaining algorithms are especially pertinent to this thesis, as they have fundamental similarities to our approach.
- Chapter 4. We give a high level overview of linear programming, and the techniques most relevant to our algorithm. Then, we present the existing linear programs for solving stochastic planning problems, and a linear program for evaluating the quality of a given policy.
- Chapter 5. This chapter develops the PBColgen algorithm – our contribution. We start by developing a system of linear equations for evaluating a policy in a way that is amenable to column generation. Using this as motivation we derive the linear program that is the backbone of PBColgen, and discuss how column generation lets us create deterministic subproblems. Then, we introduce some ways to solve these particular subproblems.

- Chapter 6. In this chapter we evaluate the performance of our novel algorithm against existing approaches in formal experiments. Then, we explain the observed behaviours, and discuss them.
- Chapter 7. Finally, we summarise the contents of this thesis, and address some interesting avenues of investigation that went beyond the scope of this thesis.

# Chapter 2

# Background for Planning

In this chapter, we present the background in planing for this thesis. We begin with deterministic planning (section 2.1), also known as classical planning. Then, we present stochastic planning problems (section 2.2), the probabilistic counterpart of deterministic planning and the main focus of this thesis. This chapter ends by reviewing determinisation (section 2.3), a common technique to relax stochastic planning problems into deterministic planning problems.

## 2.1 Deterministic Planning

First, we introduce the classic planning model. The associated problem describes a finite set of discrete states, wherein an agent must navigate from a starting state to a goal by applying actions that move the agent from its current state to the next.

**Definition 1 (Deterministic Planning Problem)** We define deterministic planning problems as the following tuple $\mathbb{D} = \langle S, s_0, G, A, T, C \rangle$ [Hector Geffner and Bonet 2013, p. 15] where

- $S$ is a non-empty, finite, discrete set of states
- $s_0$ is the initial state (note $s_0 \in S$)
- $G$ is a non-empty set of goal states (note $G \subsetneq S$ and $s_0 \notin G$)
- $A$ is the set of all actions, but we overload the symbol and define a function $A(\cdot) : S \to \mathcal{P}(A)$ such that $A(s)$ denotes the set of applicable actions in state $s$

- $T : S \times A \to S$ is a *deterministic transition function*, that is, $T(s, a) = s'$ where $s'$ is the state resulting from applying action $a$ to state $s$ – this is only defined if $a \in A(s)$.
- $C : S \times A \to \mathbb{R}_{>0}$ where $C(s, a)$ indicates the cost of applying action $a$ while in state $s$.

We assume that goal states have no applicable actions, i.e. $A(g) = \emptyset$ for any goal $g \in G$.

We often represent deterministic planning problems as weighted directed graphs where $S$ corresponds to the vertices, and transitions in $T$ correspond to edges, weighted according to $C$. Such graphs are also associated with a starting or entry state at $s_0$ and goal or target states at $G$ [Hector Geffner and Bonet 2013, p. 16].

The solution to such problems is a sequence of actions that leads the agent from the starting state to a goal.

**Definition 2 (Plan)** The solution for deterministic planning problems is a path (in the graph theoretic sense) from $s_0$ to some state in $g \in G$, which we call a *plan*. A plan can be described by a sequence of actions $\langle a^0, \ldots, a^{n-1} \rangle$ called the *action trace*, and a corresponding sequence of states $\langle s^0, \ldots, s^n \rangle$ called the *state trace* [Hector Geffner and Bonet 2013, pp. 15–16], such that

- $s^0$ is the initial state $s_0$
- $s^n \in G$
- $a^i \in A(s^i)$
- $s^i = T(s^{i-1}, a^{i-1}) \quad \forall i \in \{1, \ldots, n\}$.

Notice that, by definition of a path, states may not be revisited – in terms of the state trace that means $i \neq j \implies s^i \neq s^j \quad \forall i, j \in \{0, \ldots, n\}$

Since deterministic actions uniquely determine states, a plan can be exactly represented by the sequence of actions. Note that a sequence of states is not strictly sufficient, since there may be multiple actions between two states.

A concise way of expressing a plan with its states is

$$s^0 \xrightarrow{a^0} \ldots \xrightarrow{a^{n-1}} s^n.$$

The cost of a plan $\phi = s^0 \xrightarrow{a^0} \ldots \xrightarrow{a^{n-1}} s^n$ is given by

$$C(\phi) = \sum_{i \in \{0,\ldots,n-1\}} C(s^i, a^i).$$

A plan is optimal if its cost is minimal, i.e. plan $\phi$ is optimal if $C(\phi) \leq C(\phi')$ for all other plans $\phi'$.

At times, we must iterate over all state $s^i$, action $a^i$, effect $s^{i+1}$ triples in a plan $\phi$ where

$$\phi = s^0 \xrightarrow{a^0} \ldots s^i \xrightarrow{a^i} s^{i+1} \ldots \xrightarrow{a^{n-1}} s^n.$$

For summation, we use the shorthand

$$\sum_{s^i \xrightarrow{a^i} s^{i+1} \in \phi}$$

which is equivalent to

$$\sum_{s^i, a^i, s^{i+1} \forall i \in \{0,\ldots,n-1\}}$$

given that the plan consists of state trace $\langle s^0, \ldots, s^n \rangle$ and action trace $\langle a^0, \ldots, a^{n-1} \rangle$.

## 2.2 Stochastic Shortest Path Problems

We generalise the notion of deterministic planning by dropping the requirement that the effects of actions are unique. This allows us to model interesting problems with elements of chance. Now, rather than mapping the transition function of an effect to a single state, an action maps to a probability distribution over possible states.

**Definition 3 (SSP)** A Stochastic Shortest Path Problem (SSP) [Bertsekas and J. Tsitsiklis 1991] is the tuple $\mathbb{S} = \langle S, s_0, G, A, P, C \rangle$ where

- $S$ is a non-empty, finite, and discrete set of states
- $s_0$ is the initial state
- $G \subsetneq S$ is a non-empty set of goals (note $s_0 \notin G$)
- $A$ is the set of all applicable actions – as before we write $A(s)$ to denote the applicable actions in state $s$.
- $P(s'|s, a)$ denotes the probability of reaching state $s'$ after applying

action $a$ in state $s$ for any $s, s' \in S$ and $a \in A(s)$

- $C : S \times A \to \mathbb{R}_{>0}$ is defined as $C(s, a)$ giving the cost of applying action $a$ in state $s$.

We assume that goal states have no applicable actions, i.e. $A(g) = \emptyset$ for any goal $g \in G$.

Similar to deterministic planning problems, we often think of SSPs as directed weighted graphs. To represent the probabilistic effects of an action it is possible to use an AND/OR graph, but in this thesis we introduce hyper-edges to denote the various possible outcomes of applying an action. For example, $\mathbb{S} = \langle S, s_0, G, A, P, C \rangle$ with

- $S = \{s_0, g_0, g_1\}$,
- $G = \{g_0, g_1\}$,
- $A = \Big\{ s_0 \mapsto \{a_0\} \Big\}$,
- $P(g_0|s_0, a_0) = 0.1$, $P(g_1|s_0, a_0) = 0.2$, $P(s_0|s_0, a_0) = 0.7$,
- $C(s_0, a_0) = 1$

will be represented by the graph in figure 2.1. Note that the nodes' names correspond to the state names in $\mathbb{S}$. So, it is clear that $s_0$ is the starting state, and $g_0$ and $g_1$ are the goal states, as in $\mathbb{S}$. Also, note that probabilities of reaching states given an action are printed in black, and the cost of applying an action is printed in red. We use this convention throughout this thesis.

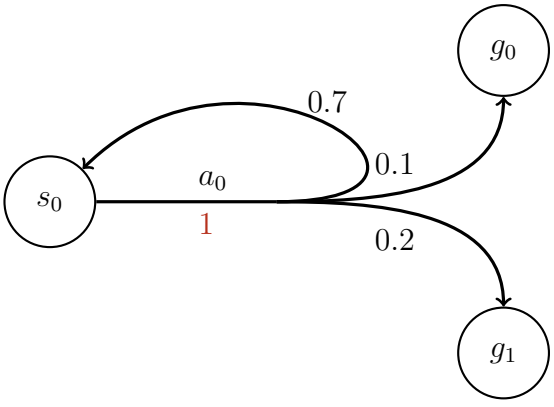



Figure 2.1: Directed weighted graph with hyper-edges representing an SSP.

**Definition 4 (Support of an action)** We refer to the possible effects of applying an action $a$ in state $s$ as the *support* of the probability distribution $P(\cdot|s,a)$. This is denoted by $\text{supp}(s,a)$. Formally,

$$\text{supp}(s,a) = \{s' : P(s'|s,a) > 0\}.$$

To solve an SSP it is not sufficient to consider a plan, since applying an action may have stochastic effects that do not correspond to the next state in the plan. We must generalise the notion of a plan to take into account the various outcomes of each action – this is called a *policy*.

**Definition 5 (Policy)** In general, a *policy* can be either a

- *deterministic policy* which is a function $\pi : S' \to A$ such that $S' \subseteq S$ and $\forall s \in S' \quad \pi(s) \in A(s)$, or a
- *stochastic policy* which is a function $\pi : S' \to$ prob. distribution over $A$.

In this project, policies refer to deterministic policies unless explicitly stated otherwise.

Note that this definition does not require a policy to be defined on all states. It is useful to distinguish between policies that are defined over the entire state space, and those that are not.

**Definition 6 (Complete & Partial Policies)** If a policy $\pi$ is defined for all states, i.e. $S' = S$, then $\pi$ is called a *complete policy*.

Otherwise, if a policy $\pi$ is not defined for all states, i.e. $S' \subsetneq S$, then $\pi$ is called a *partial policy*.

To gain more insight as to what makes a good policy, we must consider the possible paths an agent may take by following a given policy from some starting state $s$. Such potential paths are called *traces*.

**Definition 7 (Trace)** Given a policy $\pi$ and some state $s \in S$, by looking at a possible execution of $\pi$ from $s$, we obtain a potentially infinite sequence of states called a *trace*. To be precise: a finite trace $\mathcal{T} = \langle s^0, \ldots s^n \rangle$ must satisfy that

- $s^0 = s$
- all non-initial states must be reachable from their predecessors, i.e. $P(s^{i+1}|s^i, \pi(s^i)) > 0 \quad \forall i \in \{0, \ldots, n-1\}$

- if $s^i$ is defined on $\pi$, then $s^i$ must have a successor $s^{i+1}$ such that $s^{i+1} \in \text{supp}(s^i, \pi(s^i))$; so, $\pi(s^n)$ must be undefined – note that this may be because $s^n \in G$.

An infinite trace $\mathcal{T} = \langle s^0, \dots \rangle$ must only satisfy

- $s^0 = s$
- $P(s^{i+1}|s^i, \pi(s^i)) > 0 \quad \forall i \in \mathbb{N}$.

Given an arbitrary finite trace $\mathcal{T}$, we define $|\mathcal{T}|$ to be the number of actions in $\mathcal{T}$, and we define the *probability of* $\mathcal{T}$ *succeeding* by

$$P(\mathcal{T}) = \prod_{i=0}^{|\mathcal{T}|-1} P\big(s^{i+1}|s^i, \pi(s^i)\big).$$

The probability of an infinite trace is similarly defined, but the finite product is replaced with an infinite product

$$P(\mathcal{T}) = \prod_{i=0}^{\infty} P\big(s^{i+1}|s^i, \pi(s^i)\big).$$

The cost of a finite trace $\mathcal{T}$ is the sum of the costs of its actions,

$$C(\mathcal{T}) = \sum_{i=0}^{|\mathcal{T}|-1} C\big(s^i, \pi(s^i)\big).$$

The cost of an infinite trace is obtained by replacing the finite sum with an infinite sum

$$C(\mathcal{T}) = \sum_{i=0}^{\infty} C\big(s^i, \pi(s^i)\big).$$

The set of all traces obtainable from $s$ and $\pi$ is called $\mathcal{T}_{\pi,s}$.

We partition $\mathcal{T}_{\pi,s}$ into the set of finite and infinite trace:

$$\mathcal{T}_{\pi,s} = \mathcal{T}_{\pi,s}^{\text{finite}} \cup \mathcal{T}_{\pi,s}^{\text{infinite}}$$

respectively.

Note: since we have assumed that SSPs have finitely many states, infinite traces are only possible with a cycle.

Notice that there is no reason that the entire state space need be reachable by following a particular policy – even if it is complete. We are interested in the subspace of states that are actually reachable in executions of a policy.

**Definition 8 (Policy envelope)** The *policy envelope* of policy $\pi$ from state $s$ is denoted $S^{\pi,s}$, and represents the set of states reachable by following $\pi$ from $s$. Formally, we define this

$$S^{\pi,s} = \bigcup_{\mathcal{T} \in \mathcal{T}_{\pi,s_0}} \text{states}(\mathcal{T}).$$

We write $S^{\pi}$ to denote $S^{\pi,s_0}$.

Notice that there may be states in an SSP from which no goal is reachable, regardless of what actions we apply. Such states are intuitively called *dead ends*.

**Definition 9 (Dead end)** A state $s \in S$ is a *dead end* when no trace $\mathcal{T}$ exists such that $\mathcal{T}$ starts at $s$, is finite, and ends in some goal state $s_g \in G$. Equivalently,

$$\begin{aligned} s \text{ is dead end } \iff & \forall \pi \in \Pi \, \forall \mathcal{T} \in \mathcal{T}_{\pi,s} \\ & \mathcal{T} \text{ is infinite } \vee \mathcal{T} = \langle s^0, s^1 \ldots, s^n \rangle \text{ and } s^n \notin G \end{aligned}$$

If a state $s$ has no applicable actions i.e. $A(s) = \emptyset$ then $s$ is called a *trivial* dead end.

Alternatively, $s$ may be a dead end with applicable actions, in which case we call it a *non-trivial* dead end. The simplest non-trivial dead end is a state $s$ with a loop – i.e. a single action that always goes back to $s$.

If there exists a policy $\pi$ for SSP $\mathbb{S}$ such that all dead ends are avoided, i.e.

$$\mathcal{T} \text{ is finite } \wedge \text{ final state } s^n \in G \quad \forall \mathcal{T} \in \mathcal{T}_{\pi,s}$$

then $\mathbb{S}$ has only *avoidable dead ends* (or none at all).

If there does not exist such a policy for $\mathbb{S}$, then $\mathbb{S}$ has *unavoidable dead ends*.

Dead ends are important features of SSPs. However, they introduce some technical challenges, so we introduce the *give-up transformation*.

**Definition 10 (Give-up transformation)** Given an SSP $\mathbb{S} = \langle S, s_0, G, A, P, C \rangle$, we introduce "give-up" actions on each state $s$, which lets $s$ transition to a goal. First, select an arbitrary goal state $g$. Then, we get an extended set of actions $A'$, probabilities $P'$, and costs $C'$ where

$$A'(s) = A(s) \cup \{s_{\text{give-up}}\} \quad \forall s \in S$$

$$P'(s'|s, s_{\text{give-up}}) = \begin{cases} 1 & \text{if } s' = g \\ 0 & \text{if } s' \neq g \end{cases}$$

$$C'(s,a) = \begin{cases} C(s,a) & \text{if } a \in A(s) \\ D & \text{if } a \text{ is a give-up action} \end{cases} \quad \forall s \in S, a \in A'(s),$$

where $D \in \mathbb{R}_{>0}$ is some finite penalty. This gives us the transformed SSP $\mathbb{S}_{\text{give-up}} = \langle S, s_0, G, A', P', C' \rangle$.

This transformation is equivalent to the *Finite-Penalty* method [Kolobov, Mausam, and Weld 2012], which guarantees that the updated SSP contains no dead ends. Intuitively, this transformation allows any state to reach a goal, but the dead end penalty disincentivises the use of give-up actions, so only dead ends actually use them.

Now, we use this transformation to justify the strong assumption that the SSPs we consider from now contain no dead ends without loss of generality.

**Assumption 1 (Reachability)** We assume that any SSP in this work (unless explicitly stated otherwise) has no dead ends, i.e. that a goal state is reachable from all states $s \in S$. If needed, we apply the give-up transformation to guarantee this property.

Note that we will refer to states that can only reach goals via give-up actions as dead ends to emphasise their semantic, even if they are technically not dead ends.

Now, we have the tools and framework to make more fine-grained distinction between policies. As mentioned before, the envelope of a policy need not –and usually does not– contain the entire state space. So, rather than distinguishing policies by being complete or partial, we can also distinguish policies by whether they can give an action for any state that the policy may lead us to (given that the state has an applicable action).

**Definition 11 (Closed and open policies)** Policy $\pi$ is *closed* if all reachable states with applicable actions have an action defined. That is, if for all states $s \in S^\pi$ one of the following holds:

- $\pi$ is defined for $s$
- $s \in G$.

A policy that is not closed, i.e. its envelope contains a state that does not satisfy any of the criteria above, is called *open*.

A complete policy is closed since it's defined for all $s \in S$, but a closed policy may be partial. Generally we are interested in closed policies rather than complete ones, since unreachable states are of no interest.

Given that policies may be open, we want to know how reliably a policy is able to lead to a goal.

**Definition 12 (Outcomes of a policy)** The probability of an execution of policy $\pi$ from state $s$ reaching a goal state can be given by

$$P(\text{reach goal}|\pi, s) = \sum_{\mathcal{T}=\langle s^0,\ldots,s^g\rangle \in \mathcal{T}_{\pi,s}^{\text{finite}}: s^n \in G} P(\mathcal{T}).$$

Note that there may be infinitely many such traces that reach the goal, and a feasible way to calculate this probability is given by LP 8.

Based on the probability of reaching the goal, we introduce another way to distinguish policies.

**Definition 13 (Proper and Improper Policies)** A policy $\pi$ is *proper* if following it from the initial state $s_0$ reaches a goal with probability 1. That is,

$$P(\text{reach goal}|\pi, s_0) = 1.$$

On the other hand, if $P(\text{reach goal}|\pi, s_0) < 1$ then $\pi$ is *improper*.

Notice that proper policies must be closed, but the opposite need not be true.

Now, we develop a method to distinguish policies by their expected cost. We do this by introducing the *value function*.

**Definition 14 (Value function)** Abstractly, a *value function* $V : S \to \mathbb{R}_{\geq 0}$ is an estimate of the cost to get from state $s$ to some goal on the

SSP $\mathbb{S}$. We assume that $V(s) = 0 \quad \forall s \in G$ since the cost to stay at a goal is zero.

Often a value function is associated with a policy $\pi$, and indicates the cost to reach a goal from $s$ by following the policy. This is given by

$$V_\pi(s) = \begin{cases} 0 & \text{if } s \in G \\ \sum_{\mathcal{T} \in \mathcal{T}_{\pi,s}} P(\mathcal{T}) \cdot C(\mathcal{T}) & \text{else.} \end{cases}$$

Finally, we can specify what makes a policy optimal.

**Definition 15 (Optimal policy)** Given an SSP $\mathbb{S}$, we say that a policy $\pi^*$ is *optimal* if it is closed and

$$V_{\pi^*}(s_0) \leq V_\pi(s_0) \; \forall \text{ policies } \pi \text{ on } \mathbb{S}.$$

Note that $V_\pi(s_0)$ diverges if $\pi$ is improper. Therefore, given that a proper policy exists, the optimal policy will be proper.

It turns out that SSPs will always have an optimal deterministic policy [Puterman 1994, p. 27], which is why we focus on obtaining deterministic policies in this thesis. Note that an SSP may not have a unique optimal deterministic policy.

For the final part of this section we do not use assumption 1, and introduce some additional definitions for dealing with dead ends.

The definition of optimality presented in definition 15 works as long as proper policies exist on the SSP, and the assumption of reachability (assumption 1) holds. If we want to discuss optimal policies on SSPs with unavoidable dead ends, then there are multiple ways we can define it. We introduce only the following two notions of cost on improper policies, as they will be most useful in the construction of our algorithm, and in the empirical evaluation. For both, the give-up transformation will be useful.

The first notion of cost takes into account the dead-end penalty when a policy fails.

**Definition 16 (FP cost)** Given an SSP $\mathbb{S}$, we apply the give-up transformation to obtain $\mathbb{S}_{\text{give-up}}$. The *expected cost with finite penalty* (FP cost) of policy $\pi$ for $\mathbb{S}$ is given by $\mathbb{S}_{\text{give-up}}$'s value function $V_{\pi^*}(s_0)$.

The second notion is called the *Min-Cost given Max-Prob* (MCMP) criterion [Trevizan, F. Teichteil-Königsbuch, and Thiébaux 2017]. This cost includes

the cost of all possible executions of some policy, but without including the dead end cost (be it a finite penalty or infinite).

**Definition 17 (MCMP cost)** Given an SSP $\mathbb{S}$, we apply the give-up transformation to obtain $\mathbb{S}_{\text{give-up}}$. The *MCMP cost* of policy $\pi$ for $\mathbb{S}$ is given by

$$\sum_{\mathcal{T} \in \mathcal{T}_{\pi,s_0}} \sum_{\substack{s \xrightarrow{a} s' \in \mathcal{T} \text{ s.t.} \\ a \text{ is not give-up action}}} C(s,a)$$

where $\mathcal{T}_{\pi,s_0}$ is the set of traces from following $\pi$ from $s_0$ on $\mathbb{S}_{\text{give-up}}$.

Note that both of these costs collapse down to the regular expected cost of a policy if the policy is proper.

A policy is optimal according to either of these definitions of cost if the policy has minimal cost compared to other possible policies.

## 2.3 Determinisation

Some planners, including the one we develop in this thesis, rely on solving deterministic subproblems that help us construct a policy across the SSP. To construct these deterministic subproblems we relax properties of a given SSP to obtain a deterministic problem [Yoon, Fern, and Givan 2007]. Note that this type of transformation inherently loses information (unless the SSP only had deterministic actions to begin with).

The first natural transformation is the *all-outcomes determinisation*, where we take each effect of a probabilistic action, and transform it into a deterministic action in the relaxation. So, we pretend that we can obtain any possible outcome of an action deliberately.

**Definition 18 (All-outcomes determinisation)** The all-outcomes determinisation of $\mathbb{S} = \langle S, s_0, G, A, P, C \rangle$ yields the deterministic planning problem $\mathbb{S}_{\text{det}} = \langle S, s_0, G, A', T, C' \rangle$, where

- the states are completely unchanged, this includes the initial and goal states
- for each action $a \in A(s)$ we generate a set of determinised actions $\det(a) = \{a \triangleright s' : P(s'|s,a) > 0\}$, then $A'(s) = \bigcup_{a \in A(s)} \det(a)$
- $T(a \triangleright s', s) = s' \quad \forall a \triangleright s' \in A'(s)$

- $C'(s, a \triangleright s') = C(s, a) \quad \forall a \triangleright s' \in A'(s)$.

Note that we do not include the probability in the determinised action's cost.

Under assumption 1 that there are no dead ends, the all-outcomes determinisation will always be solveable. If we relax this assumption, and allow unavoidable dead ends this guarantee is weakened: the all-outcomes determinisation is always solveable as long as there is a goal reachable from the initial state.

Alternatively, we can relax an SSP by only considering the most likely effect – rather than all of them. This is called the *most-likely-outcomes determinisation*.

**Definition 19 (Most-likely-outcomes deteriminisation)** The most-likely-outcomes determinisation of $\mathbb{S} = \langle S, s_0, G, A, P, C \rangle$ gives $\mathbb{S}_{\text{det}} = \langle S, s_0, G, A', T, C' \rangle$, where

- the states are completely unchanged, this includes the initial and goal states
- for each action $a \in A(s)$ we generate the singleton

$$\det(a) = \left\{ \underset{s' \in \text{supp}(s,a)}{\text{argmax}} \; P(s'|s, a) \right\}$$

  breaking ties arbitrarily. Then, $A'(s) = \bigcup_{a \in A(s)} \det(a)$.
- $T(a \triangleright s', s) = s' \quad \forall a \triangleright s' \in A'(s)$
- $C'(s, a \triangleright s') = C(s, a) \quad \forall a \triangleright s' \in A'(s)$

Note that the arbitrary breaking of ties means that the most-likely-outcomes determinisation of a problem may not be unique.

The most-likely-outcomes determinisation retains some information about the probabilities of the original SSP, but there is no guarantee that the relaxation is solveable – even if a goal is reachable from $s_0$ in $\mathbb{S}$. For instance, consider the simple SSP with two states and one action, where the action has a 0.1 probability of reaching the goal, and 0.9 probability of looping back to the initial state. Under the most-likely-outcomes determinisation, this becomes a disconnected graph.

# Chapter 3

# Related Work

In this chapter we explore various algorithms developed in the planning community for constructing solutions for SSPs. To start with, we introduce the classic algorithm for solving SSPs – Value Iteration (section 3.1); we move on to RTDP, a more modern variant (section 3.2). Then, we take an interlude from planners to introduce some additional terminology (section 3.3), which makes it easier to talk about planners that use solutions for determinisations to construct policies. This leads into the introduction of two such planners: FF-Replan (section 3.4) and Robust-FF (section 3.5), which are the algorithms that function most closely to our novel approach. Finally, we summarise the properties and categories of introduced planners in the discussion (section 3.6).

## 3.1 Value Iteration

Value Iteration (VI) [Bellman 1957] is the classic algorithm for computing optimal policies for SSPs. It uses a dynamic programming approach to compute a value function which accurately estimates the cheapest costs to get from any state to its nearest goal, which can then be used to compute an optimal policy.

Recall that value functions $V : S \to \mathbb{R}_{\geq 0}$ should give an estimate of how expensive it will be to reach the nearest goal from $s$. Then, the *optimal value function* $V^*$ denotes precisely the cheapest expected cost to get to a goal from any state $s$.

**Definition 20 (Optimal value function)** The optimal value function which indicates the cheapest expected costs to get from any state to a goal is specified by the *Bellman equation*:

$$V^*(s) = \begin{cases} 0 & \text{if } s \in G \\ \min_{a \in A(s)} C(s,a) + \sum_{s' \in S} P(s'|s,a) \cdot V^*(s') & \text{else.} \end{cases}$$

Note that while there may not be a unique optimal policy, there is a unique optimal value function.

Given a value function $V$, we can construct a policy by selecting actions that lead to the cheapest expected cost in a greedy fashion.

**Definition 21 (Greedy policy)** Given a value function $V$, the *greedy policy* $\pi_V(s)$ is given by

$$\pi_V(s) = \operatorname*{argmin}_{a \in A(s)} C(s,a) + \sum_{s' \in S} P(s'|s,a) \cdot V(s') \quad \forall s \in S \setminus G.$$

Ties may be broken arbitrarily.

It turns out that any greedy policy $\pi_{V^*}$ on the optimal value function $V^*$ is optimal.

The upshot is that finding the optimal value function enables us to find all optimal policies. And this is what value iteration does. It computes $V^*$, and then returns $\pi_{V^*}$ which we know to be optimal. To compute $V^*$, value iteration requires some initial value function $V_0$ – the exact estimates for each state do not matter, as long as they are in $\mathbb{R}_{\geq 0}$. Then, VI applies the following bootstrapped variant of the Bellman equation iteratively: for all $s \in S$ and incrementing $i \in \{0, \ldots, n\}$ for some $n \in \mathbb{N}$

$$V_i(s) = \begin{cases} 0 & \text{if } s \in G \\ \min_{a \in A(s)} C(s,a) + \sum_{s' \in S} P(s'|s,a) \cdot V_{i-1}(s') & \text{else.} \end{cases}$$

Applying this equation once is called a *Bellman backup*. Now, as $i \to \infty$ we get that $V_i \to V^*$ – our computed value function approaches the optimal value function; but it need not be the case that we converge in finitely many steps. Consider the SSP in figure 3.1, and suppose the initial value function estimates $V_0(s_0) = 0$.

We see that our algorithm will estimate $V_i(s_0) = 1 + \sum_{k=1}^{i} 0.9^k$ for $i > 0$, which will never give us an exact value for finite $i$. However, there is a

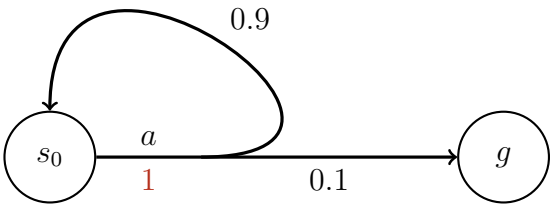



Figure 3.1: Example of an SSP in which VI can not converge to the exact optimal value function in finitely many steps.

point at which our estimate is "good enough" in the sense that the greedy policy derived from our approximation of the optimal value function yields an optimal policy. To gauge this, we determine how much our estimate varies between two Bellman backups, given by the *Bellman residual* defined by

$$\text{res}_{V_i} = \max_{s \in S} \Big| V_{i+1}(s) - V_i(s) \Big|.$$

We say that VI has converged to the optimal solution w.r.t. epsilon consistency when

$$\text{res}_{V_i} < \epsilon$$

for some –usually small– value $\epsilon \in \mathbb{R}_{>0}$. This gives us a definition of optimality for value-based solutions.

**Definition 22 (Optimal w.r.t. epsilon consistency)** A greedy policy $\pi_V$ is optimal w.r.t. epsilon consistency if the value function $V$'s Bellman residual is less than epsilon, i.e.

$$\max_{s \in S} \Big| V'(s) - V(s) \Big| < \epsilon$$

where $V'$ is the Bellman backup applied to $V$, and $\epsilon$ is some value $\in \mathbb{R}_{>0}$.

VI guarantees that the Bellman residual of its values converges to a value less than $\epsilon$ in finitely many steps, and thus the algorithm guarantees an optimal policy w.r.t. epsilon consistency.

The easiest way to think about each iteration of VI is that $V_i(s)$ is computed for all $s \in S$ at once or in parallel. If the algorithm is implemented in this way, it is usually referred to as synchronous VI. Instead of updating all states at once, we can consider only one state each iteration, which is

called *Asynchronous Value Iteration* [Hector Geffner and Bonet 2013, pp. 84–85].

The benefit of selecting a subset of states in each iteration is that we can leverage some properties of the underlying graph so that we don't have to waste time recomputing converged or useless states. A good motivating example is a tree with one goal: if we start VI and update states according to a breadth first search from the goal, we can find the optimal value policy in one pass with $|S|$ many Bellman updates.

In general, asynchronous VI guarantees an optimal value function as long as all states are allowed to be visited infinitely many times – in the sense that running the algorithm indefinitely should not disallow any states; but if we are this general we do not gain the benefits of asynchronous over synchronous VI. It is useful if we can start with an initial value function $V_0$ that underestimates all costs, i.e. $V_0(s) \leq V^*(s) \quad \forall s \in S$ – akin to an admissible heuristic. This ensures that updating any state $s$ will only increase the cost, i.e. $V_i(s) \leq V_{i+1}(s) \quad \forall i \in \mathbb{N}$. A consequence of this property is that we can safely not update states that are not in our current policy, and their underestimated cost is already higher than our value function's estimates, so there is no way that updating them will decrease our value estimates.

## 3.2 Real Time Dynamic Programming

Real Time Dynamic Programming (RTDP) [Barto, Bradtke, and Singh 1995] can be seen as a version of asynchronous VI with an admissible value function, where we add random sampling to focus in on the most likely outcomes.

The principle is that RTDP starts with admissible value function $V$, and implicitly generates the greedy policy $\pi_V$. RTDP then simulates an execution of $\pi_V$, and updates its value function with the costs incurred during simulation. Each of these executions is called a *trial*, see algorithm 1. Remember that the initial $V$ is admissible, so updates on $V$ can only increase cost. These trials are repeated until the Bellman residual over states in the envelope of $\pi_V$ are less than epsilon [Bonet and H. Geffner 2003].

This method will converge to an optimal value function w.r.t
epsilon-consistency, as long as all states in the policy envelope of our candidate policy are explored – as before with asynchronous VI and an admissible value function.

Running trials repeatedly implicitly searches the entire policy envelope of

current $\pi_V$, but gets interrupted when $\pi_V$ is improved. Once $\pi_V$ is optimal, this search will not be interrupted. And so, RTDP guarantees that it will eventually search the entire policy envelope of some optimal policy, and thus converges to an optimal value function.

The problem is that sampling can not tell us when we have explored the entire envelope of our current $\pi_V$, or how long it will take. So, RTDP is only asymptotically optimal, in the sense that the value function will eventually have a Bellman residual less than $\epsilon$, but stopping it after any finite amount of time can not guarantee this.

A more recent algorithm, Labelled Real Time Dynamic Programming (LRTDP) [Bonet and H. Geffner 2003] extends RTDP by labelling states $s$ as solved when all states in the policy tree rooted at that $s$ have a residual less than $\epsilon$.

The theoretical benefit of this approach is that LRTDP converges in a finite number of steps w.r.t. epsilon-consistency. Furthermore, it turns out that the overhead of the labelling mechanism is well worth it, because the computation time for solutions of similar quality is generally reduced.

**Algorithm 1:** RTDP trial

**Input:** SSP $\mathbb{S} = \langle S, s_0, G, A, P, C \rangle$, curr. value function $V$, state $s_{\text{start}}$

current state $s \leftarrow s_{\text{start}}$

**while** $s \notin G$ **do**

  $a_{\text{best}} \leftarrow \operatorname{argmin}_{a \in A(s)} C(s,a) + \sum_{s' \in S} P(s'|s,a) \cdot V(s')$ `// select best action`

  $V(s) \leftarrow C(s, a_{\text{best}}) + \sum_{s' \in S} P(s'|s, a_{\text{best}}) \cdot V(s')$ `// update value function`

  next state $s' \leftarrow$ simulated state from applying $a_{\text{best}}$ to $s$ `// i.e.` $s'$ `is selected with prob.` $P(s'|s, a_{\text{best}})$

  $s \leftarrow s'$ `// update current state`

## 3.3 Interlude: Extended Definitions

To make it easier to talk about algorithms that use deterministic solutions to construct policies, and at various stages may have partial policies, we introduce the following terminology.

When we are considering partial policies, we can distinguish states that are defined in our policy, and those that are not.

**Definition 23 (Open & closed states)** We can partition policy $\pi$'s policy envelope $S^\pi$ into

$$S^\pi = S^\pi_{\text{closed}} \cup S^\pi_{\text{open}}$$

where $S^\pi_{\text{closed}}$ denotes the set of states that are defined on $\pi$, and $S^\pi_{\text{open}}$ denotes those that are not. So,

$$S^\pi_{\text{closed}} = \{s \in S^\pi : \pi(s) \text{ is defined}\}$$
$$S^\pi_{\text{open}} = \{s \in S^\pi : \pi(s) \text{ is undefined}\}.$$

We have assumed that goal states have no applicable actions in SSPs; that is, $A(s_g) = \emptyset \; \forall s_g \in G$. Consequently, goal states are in $S^\pi_{\text{open}}$.

Note that the edge case of trivial dead ends –i.e. non-goal states $s$ with $A(s) = \emptyset$– is avoided by assumption 1.

Now, to discuss how plans on deterministic relaxations can be used to construct policies on the original SSPs, we introduce the concept of casting a plan into a policy.

**Definition 24 (Casted plan)** Suppose we have SSP $\mathbb{S}$ and its determinisation $\mathbb{S}_{\text{det}}$. Given a plan $\phi$ for $\mathbb{S}_{\text{det}}$ we can interpret it as a policy on $\mathbb{S}$. We call such a policy $\pi_\phi$ the *casted plan* of $\phi$. Formally, given plan $\phi$ with state trace $\langle s^0, \ldots, s^n \rangle$ and action trace $\langle a^0, \ldots, a^{n-1} \rangle$, the casted plan is given by

$$\pi_\phi(s^i) = a^i \quad \forall i \in \{0, \ldots, i-1\}.$$

Note that such a policy is only closed if the plan $\phi$ accounts for all effects of the actions it applies, i.e.

$$\bigcup_{i \in \{0,\ldots,n-1\}} \text{supp}(s^i, a^i) \setminus G = \text{states}(\phi) \setminus G$$

where $\langle s^0, \ldots, s^n \rangle$ and $\langle a^0, \ldots, a^{n-1} \rangle$ are $\phi$'s state and action traces. This is typically not the case, so most casted policies are partial. The two main cases when a casted plan is closed, is when the plan uses deterministic actions; and when the actions can only fail by taking the state to another one in the plan. The first of these is clear; for the second, consider figure 3.2.

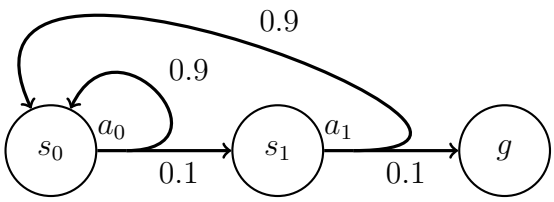



Figure 3.2: Example of an SSP where a casted plan from the all-outcomes-determinisation yields a closed policy.

On the all-outcomes determinisation there is exactly one plan from $s_0$: $\phi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} g$. The casted plan $\pi_\phi$ then has the following mapping:

$$s_0 \mapsto a_0$$
$$s_1 \mapsto a_1.$$

It is very unlikely that $\phi$ would succeed on the SSP, in the sense that applying $a_0$ to reach $s_1$ and then applying $a_1$ to reach $g$ will work with probability 0.01. But, the casted plan $\pi_\phi$ is defined on all states, and is thus complete – and it is even proper.

## 3.4 FF-Replan

*FF-Replan* [Yoon, Fern, and Givan 2007] is an online planner, which introduced the idea of solving deterministic sub-problems to construct solutions for stochastic planning problems. The principle of FF-Replan is to get a plan from the determinisation of a given SSP, and cast that plan into a policy. This policy is not guaranteed to be closed – if FF-Replan is given an undefined state $s$, it simply generates a plan from $s$, and appends this casted plan to its policy. See algorithm 2.

Since deterministic solvers are very efficient even over large state spaces this makes for an effective online solver, and approaches a closed policy as more states are discovered during execution.

Note that the original deterministic planner used was FF (hence FF-Replan) but any deterministic planner can be used – and it need not be optimal.

Although FF-Replan is intended as an online solver, we can use it as an offline solver by simulating executions of the current policy, and updating it with FF-Replan upon failure. This basic iteration has no termination condition, so we would perform this with a time-out. Since the time-out may not give sufficient time, and a closed policy is only guaranteed in the limit, it is best to allow the online solving component as well.

**Algorithm 2:** FF-Replan

**Input:** SSP $\mathbb{S} = \langle S, s_0, G, A, P, C \rangle$

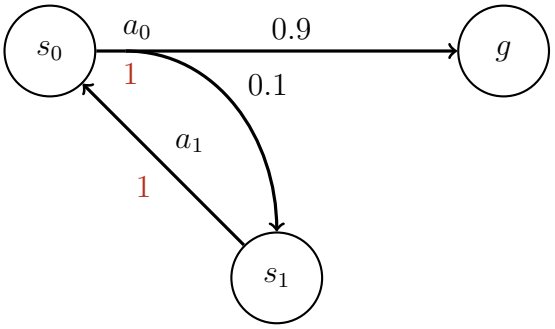
```
// Initial set up
```
$\mathbb{S}_{\text{det}} \leftarrow$ determinisation of $\mathbb{S}$
$\phi \leftarrow$ solve $\mathbb{S}_{\text{det}}$ with deterministic planner
$\pi \leftarrow \pi_\phi$
```
// Function that returns an action for state s
```
**function** `decide_action(`$s$`):`
    **if** $\pi(s)$ *is undefined* **then**
        $\phi \leftarrow$ solve $\mathbb{S}_{\text{det}}$ with deterministic planner
        $\pi(s') \leftarrow \pi_\phi(s') \quad \forall s' : \pi(s')$ undefined $\wedge\, \pi_\phi(s')$ defined
    **return** $\pi(s)$

## 3.5 Robust-FF

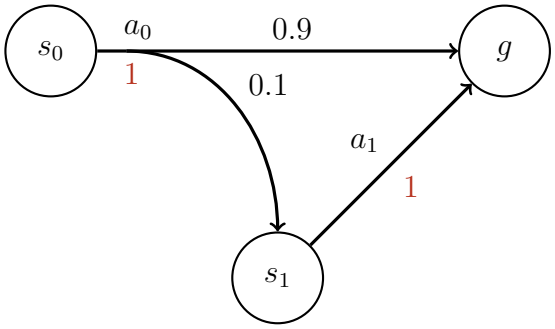
*Robust-FF* [Florent Teichteil-Königsbuch and Infantes 2008] builds on FF-Replan to construct more robust policies, i.e. the policies are less likely to require online replanning. Robust-FF also introduces some techniques to improve the efficiency of the algorithm.

First, Robust-FF prefers the most-likely-outcomes determinisation, in the sense that the deterministic solver is first run on this determinisation. This has the advantage that plans typically have a higher chance of succeeding on the SSP. Note however, that this is heuristic – there are plenty of examples where this is not the case. As discussed in section 3.3, the most-likely-outcomes determinisation does not guarantee a solution. So, if the deterministic solver fails, we must run it again on the all-outcomes-determinisation.

Second, Robust-FF makes the observation that given an undefined state $s$, we don't necessarily need to find a plan to a goal to deal with it effectively, but rather it suffices to find a plan to a closed state – since from there we




(a) If applying action $a_0$ "fails" and goes to state $s_1$, we can "undo" our mistake by applying action $a_1$, and returning to $s_0$ from where we failed.




(b) If applying action $a_0$ "fails" and goes to state $s_1$, we can "fix" our mistake by applying action $a_1$, and going to the intended effect of $a_0$: state $g$.

Figure 3.3: Simple SSPs demonstrating the undo and fix actions.

know how to get to a goal. This optimisation allows Robust-FF to save time in computation.

Finally, Robust-FF enhances the simulation approach we introduced in 3.4 to use FF-Replan as a pseudo-offline solver. Rather than only simulating executions and replanning undefined states, Robust-FF simulates $N$ full executions of the current policy, and provides an estimated probability of success by $\frac{\text{number of executions that did not reach undefined state}}{N}$. If this estimate is sufficiently small –as determined by the user– it terminates and returns its current policy; otherwise it replans. Note that our implementation keeps track of the undefined states in our simulations –up to some cap–, and solves for these states on failure.

We said that Robust-FF may choose to replan to closed states rather than only goal states. This omitted the detail that deterministic solvers tend to underperform when provided a large number of goals. Therefore, a stronger implementation will only select a subset of closed states as artificial goals. The selection method is heuristic – we describe one approach that was used in our implementation.

Across domains there are some common probabilistic "detours" from plans on stochastic problems. Suppose we are in state $s_i$ and apply action $a_i \in A(s_i)$, which takes us to undesired state $s_k$. Then, an *undo action* $a_{\text{undo}} \in A(s_k)$ takes us back to $s_i$, and a *fix action* $a_{\text{fix}} \in A(s_k)$ takes us to the intended effect of $a_i$. See figure 3.3 for a graphical representation.

To motivate why these actions appear, consider a problem where you must give directions to a driver who gets their lefts and rights confused 10% of the time. If the driver makes an incorrect turn we can undo their error by performing a u-turn and returning to the intersection where the mistake was made. Alternatively, we can direct the driver to reach the original destination via an alternative route, and thereby fix their mistake.

If we accept that these cases are common, then, if Robust-FF reaches an undefined state, it is worthwhile to add the previous state we were in and the intended effect of our action, so that undo and fix actions respectively can be found by the deterministic solver.

Another heuristic method for deciding which closed states to add as artificial goals, is to generate a trace by following the state we failed from, and taking the most likely effects. So, suppose we were in state $s^i$ before applying $\pi(s^i)$ took us to an open state. We generate the state trace $\langle s^i, s^{i+1}, \ldots, s^n \rangle$ where

$$s^{i+1} = \underset{s \in \text{supp}(s^i, \pi(s^i))}{\text{argmax}} P(s|s^i, a^i),$$

and $s^n$ is an open state – either a goal or undefined. Due to Robust-FF's sampling nature we expect the most likely effects of each state to be defined often. For this reason we expect that the states in this trace are good candidates for artificial goals. For the same reason the trace can be quite long – if it exceeds the number of artificial goals we desire, it is reasonable to select a subset of states from the start of the trace – as they are the most likely to be closed.

Note that the simplest approach for adding artificial goals is simply by randomly selecting some closed states.

The basic algorithm for Robust-FF is straightforward, but there are a lot of parameters that can be modified. For instance, the number of artificial goals we allow, the maximum depth we allow for our simulations, the number of simulations we perform to compute the probability of failure, etc. These are up to the user to decide, and there is no single best answer.

Note that Robust-FF –like FF-Replan– was originally designed with use of the FF planner, but any deterministic planner works. In our implementation we used FF.

| | Optimal | Replanner |
|---|---|---|
| VI | Yes | No |
| RTDP | Yes* | No |
| LRTDP | Yes | No |
| FF-Replan | No | Yes |
| Robust-FF | No | Yes |

Table 3.1: Summary of discussed planners and their categorisation.
* RTDP is optimal w.r.t. epsilon-consistency in the limit.

## 3.6 Discussion

We have introduced two families of algorithms for constructing policies on SSPs. The first family –encompassing VI and its variants; RTDP, and LRTDP– compute optimal value functions that enable them to construct an optimal policy (w.r.t. epsilon-consistency). The second family –FF-Replan and Robust-FF– relies on deterministic solvers to solve subproblems, and slowly build up a policy. Note that this is of course not an exhaustive list of algorithms nor approaches, but it suffices for the purposes of this thesis. We categorise the second family as *replanners* [Little and Thiébaux 2007]. We summarise the discussed planners in table 3.1.

In thesis, we are primarily concerned with the two replanners. To motivate the differences between FF-Replan, Robust-FF, and an optimal planner, we present the SSP in figure 3.4. We assume an optimal deterministic planner for the following discussion. FF-Replan and Robust-FF will apply their preferred determinisations to the problem, and in both cases they will find the shortest plan $s_0 \xrightarrow{a} s_1 \xrightarrow{b} g$ with a cost of 11. If this partial policy fails and we end up in state $s_2$, then FF-Replan and Robust-FF behave differently. FF-Replan will run the deterministic planner to get from $s_2$ to $g$ in the cheapest way possible, which is $s_2 \xrightarrow{d} g$. On the other hand, Robust-FF will insert the states it already has defined in its policy as additional goal states, so it will find the shortest path from $s_2$ to $s_0$, $s_1$, or $g$. This will return the plan $s_2 \xrightarrow{c} s_1 \xrightarrow{b} g$, since the cheapest way to get to any of the new goal states is with a cost of 999, ignoring the fact that this is not a goal state of the actual problem.

An optimal planner will notice that action $a$ has an expected outcome of $1 + 0.75 \cdot 10 + 0.25 \cdot 1000 = 258.5$, whereas action $a'$ has an expected outcome of 100. So, an optimal planner will choose $a'$.

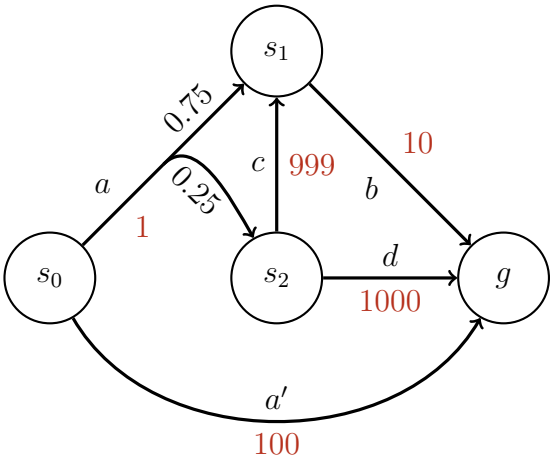



Figure 3.4: SSP that exemplifies the different behaviours of FF-Replan, Robust-FF, and an optimal planner.

# Chapter 4

# Background for Linear Programming

The formal details and development of linear programming is beyond the scope of this thesis, but in this chapter we try to give some intuition and a high level overview of the topic. First, we give a geometric motivation and some notations for linear programming (section 4.1). Then, we introduce the simplex algorithm, classic algorithm for solving linear programs (section 4.2) and the concept of duality (section 4.3) which is indispensable throughout linear programming techniques. This set up allows us to introduce the fundamental technique behind our novel algorithm: column generation (section 4.4). Finally, we discuss how SSPs can be encoded as linear programs (section 4.5) and present a formulation that allows us to evaluate policies in a robust way (section 4.6).

## 4.1 Introduction

Linear programming is an approach for solving minimisation or maximisation problems with linear equalities and/or inequalities over continuous variables. Such a problem may look like

$$\begin{array}{lll} \text{maximise} & x & \text{(LP1)} \\ \text{subject to} & x - 2y \leq 0 & \text{(C1)} \\ & -2x + y \leq 0 & \text{(C2)} \\ & x + y \leq 0 & \text{(C3)} \end{array}$$

where $x, y \in \mathbb{R}$.

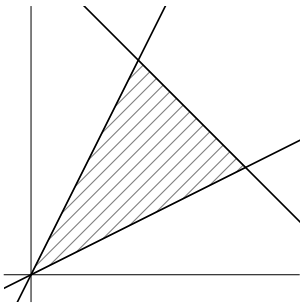


Figure 4.1: Geometric representations of the inequalities presented in LP 1.

Since we have two variables, we can draw these inequalities in 2 dimensional space to get figure 4.1. The shaded triangle represents the *feasible region*, that is, the values that the variables can take such that the inequalities are satisfied. The linear term that we are trying to minimise or maximise, generally referred to as the *objective*, can be represented by a vector. In our case, this vector points directly right along the $x$-axis, and so finding the optimal solution consists of finding the right-most point on the feasible region, i.e. the triangle. The vector may point in any direction, and the optimal solution finds the most extreme point with respect to that vector.

Note that the feasible region does not need to be finite. This can make problems uninteresting by allowing the objective to be infinitely large or small in maximisation and minimisation problems respectively; however, this does not need to be the case. Consider LP 1 if we remove constraint (C2) – the feasible region is unbounded, but the maximum objective remains the same. The direction in which an LP is unbounded is described by an *extreme ray* [Bertsimas and John Tsitsiklis 1998, p. 176].

As we move to linear programs with more variables and constraints, we end up with a feasible region in higher dimensions. The intersection of each constraint's subspace yields a *convex polytope*.

To express general linear programs we use matrix and vector notation

$$\begin{array}{rl} \text{maximise/minimise} & \mathbf{c}^T \cdot \mathbf{x} \\ \text{subject to} & A\mathbf{x} \begin{matrix} \leqslant \\ = \\ \geqslant \end{matrix} \mathbf{b} \end{array} \qquad \text{(LP2)}$$

where $\mathbf{x}$ is the vector of variables, $A$ is a matrix of the coefficients in constraints, $\mathbf{b}$ is the vector of constants on the right-hand side of constraints, and $\mathbf{c}$ is the cost vector – i.e. the variable coefficients in the objective.

## 4.2 Simplex Algorithm

Given that linear programs describe a convex polytope, the simplex algorithm makes the realisation that an optimal solution must lie on a vertex. This becomes intuitive when we consider a polytope in 2 dimensions (as in LP 1 and correspondingly figure 4.1). Note that there may be infinitely many optimal solutions lying on an edge (or a facet in aribtrary dimensions), but then there is still a vertex that shares this edge, and thus gives an optimal solution.

This observation means that we can move along the vertices of the region, and eventually find the optimum. The simplex algorithm does precisely this – it considers one vertex of the feasible region at a time, and intelligently decides to which vertex to move to. The number of constraints in the linear program limit the number of variables that may be non-zero so that the solution lies on a vertex. So, if we want to move from one vertex to another, we need to set a variable that is currently zero to non-zero, and respectively set a variable that is currently non-zero to zero. The decision procedure for deciding which variables to apply this to takes into account how much the new variable contributes to the objective, and how much the old variable contributed to the objective before it was set to zero. One way to do this is with dual variables as we explain in the next section.

## 4.3 Duality

Any linear program has an associated dual LP, which yields the same objective – if it exists. We refer to the original LP as the primal LP. Dual LPs are interesting to us because the variables of dual LPs have semantic meaning in the primal LP with interesting consequences.

To motivate this idea, consider the simple LP 3. By inspection, we see that the optimal solution is $x_1 = 5$ and $x_2 = 0$.

$$\begin{array}{lll} \text{maximise} & 2x_1 + x_2 & \text{(LP3)} \\ \text{subject to} & x_1 \leq 5 & \text{(C1)} \\ & x_2 \leq 5 & \text{(C2)} \\ & x_1 + x_2 \leq 5 & \text{(C3)} \\ & x_1, x_2 \geq 0 & \end{array}$$

If we were not able to solve the problem by inspection, we could combine constraints to obtain an upper bound for the optimal solution. For instance,

we might combine (C1) twice and (C2) once to get the expression $2x_1 + x_2 \leq 15$. We have carefully selected constraints such that the left-hand side is equivalent to the objective, and thereby we can say that the objective is upper bounded by 15. If we combine constraints more intelligently, e.g. (C1) and (C3) we get the expression $2x_1 + x_2 \leq 10$. Now we know that we can do no better than 10. In this case we know that second lower bound is tight, but in general this is not clear. We can automate this procedure, combine all constraints, and look for the lowest upper bound – which turns into another LP.

We omit the details of the procedure here [Bertsimas and John Tsitsiklis 1998, pp. 142–146], but applying this transformation gives us the dual LP presented in LP 4.

$$\begin{array}{rll} \text{minimise} & 5y_1 + 5y_2 + 5y_3 & \text{(LP4)} \\ \text{subject to} & y_1 + y_3 \geq 2 & \text{(C1)} \\ & y_2 + y_3 \geq 1 & \text{(C2)} \\ & y_1, y_2, y_3 \geq 0 & \end{array}$$

We see that LP 4 can not attain a better objective that 10. The strong duality theorem [Bertsimas and John Tsitsiklis 1998, p. 148] tells us that this is true in general – if an optimal solution exists for the dual, then the primal will have an optimal solution with the same value, and vice versa.

There is a strong connection between both formulations of the problem. The variable in the dual problem that is associated with a particular constraint in the primal problem is called the *dual variable* or *shadow cost*. The latter name comes from the property that a dual variable is only non-zero if the associated constraint is active. Therefore, if the dual variable is non-zero, we know that we could improve the primal by relaxing that constraint. In our example the simplex algorithm would give a dual solution of $y_1, y_2 = 0$ and $y_3 = 2$, which tells us that relaxing (C3) in the primal would enable us to improve the objective, which is indeed true.

## 4.4 Column Generation

Column generation is a framework which enables to solve an LP with an infeasibly large number of variables by working with linear programs with a reduced subset of variables, and iteratively adding variables in such a way the optimal solution for the reduced linear program is also the optimal solution for original LP.

The initial LP is called the *Master Problem*. These usually optimise over a set of variables and constraints that explode in terms of cardinality for larger problems. The classic example for this is the "cutting stock" problem [Lübbecke and Desrosiers 2005, p. 24], where the agent has a supply of logs, and is asked to cut out particular lengths, e.g. given a log of length 10m from which we should cut three sections of 1m and two sections of 2m, we can cut 1m out of three logs, and 2m from another two logs; or more sensibly, cut out all sections from one 10m log. The agent should minimise the number of logs that are used. The problem is formulated in terms of "patterns." Assuming that we are only interested in 1m and 2m cuts as before, this means the agent may cut out of a single log the following patterns: ten 1m cuts, eight 1m cuts and one 2m cut, six 1m cuts and two 2m cuts, etc. For small problems this may still be feasible, but as they increase we can not even realistically enumerate all possibilities.

For the remainder of this thesis we assume that any master problems are minimisation problems. This lets us introduce the following terminology in a way that is more relevant to our use cases.

**Assumption 2 (Master problems)** We assume that master problems are minimisation problems, i.e. LPs of the form

$$\begin{aligned} \text{minimise} \quad & \mathbf{c}^T \cdot \mathbf{x} \\ \text{subject to} \quad & A\mathbf{x} \gtreqless \mathbf{b} \end{aligned}$$

The principle of column generation is that we do not optimise over the space of all these variables, but with a subset of sufficiently meaningful variables. This LP with only a subset of values is called the *Reduced Master Problem* (RMP) [Lübbecke and Desrosiers 2005].

The column generation algorithm initialises its RMP with some variables – typically a simple set that makes the RMP feasible. Then, we select a new variable or column that should be included in the next iteration of column generation. One way of looking at this is that all columns are already in the RMP – but only a small number of them are set to non-zero. Then, the method of deciding which column to add next becomes similar to the introduction of variables in the simplex algorithm. We compute the *reduced cost* of a variable to determine how the objective will be affected by introducing it.

**Definition 25 (Reduced cost)** Given dual variables $\Delta^*$ associated with

optimal solution to RMP, the reduced cost of variable $x$ is given by

$$rc(x, \mathbf{\Delta}^*) = c_i - \sum_{j=1}^{m} a_{i,j}\Delta_j^*.$$

The elegance of the column generation method comes from the guarantee that a column with a negative reduced cost will improve the solution; and conversely, if there is no column with a negative reduced cost, we know that the solution optimal, i.e. the solution to the reduced master problem is the same as the solution to the master problem.

Note that RMPs may not always be feasible, even if the master problem is. Consider again the cutting stock example: if the have 10m logs and are asked to cut out five 2m sections, but the patterns in our RMP only consider cuts of 1m, then it is infeasible. Column generation can deal with this scenario with the *Farkas cost*, which lets us determine which column will help to make the solution feasible. Since the primal RMP is infeasible, the dual is either infeasible or unbounded. Then, it does not make sense to use optimal dual variables $\mathbf{\Delta}^*$, but instead we consider the extreme rays on the dual, called the *dual rays* $\mathbf{\Delta}^{\text{ray}}$. Intuitively, these indicate in which direction the dual LP is unbounded. If we can add constraints to make the dual bounded, then we also force the primal to be feasible by the strong duality theorem. Farkas cost uses this idea, and is defined as follows.

**Definition 26 (Farkas cost)** Given dual variables $\mathbf{\Delta}^{\text{ray}}$ associated with optimal solution to RMP, the Farkas cost of variable $x$ is given by

$$fc(x, \mathbf{\Delta}^{\text{ray}}) = \sum_{j=1}^{m} a_{i,j}\Delta_j^{\text{ray}}.$$

## 4.5 SSPs as Linear Programs

### Primal LP

Linear programming can be applied to solving SSPs. The primal LP formulation for solving SSPs computes an optimal value function (see definition 20) [Puterman 1994, p. 223]. We do this by introducing variables $v_s$ for each state $s \in S$, which represent $V^*(s)$ in the optimal solution. These variables are constrained by a variant of the Bellman equation – which defines an optimal value function. Instead of enforcing equality, we set the Bellman equation as an upper bound, and get the LP to maximise $v_s$'s – which in the optimal

solution gives equality. This LP is presented in LP 5.

$$
\begin{array}{lll}
\text{maximise} & \sum_{s \in S} v_s & \text{(LP5)} \\
\text{subject to} & v_s = 0 \qquad\qquad \forall s \in G & \text{(C1)} \\
& v_s \leq C(s,a) + \sum_{s' \in S} \Big( P(s'|s,a) \cdot v_{s'} \Big) \qquad \forall s \in S, a \in A(s) & \text{(C2)}
\end{array}
$$

### Dual LP

As with any LP, we can take the dual of the primal LP for SSPs. Elegantly, this results in an LP that can be interpreted as a network flow [d'Epenoux 1963]. We present the dual LP for SSPs in LP 6, where $in(s)$ and $out(s)$ semantically represent the amount of flow going into and out of state $s$, and the corresponding constraints (C2) and (C4) should be seen as helper constraints which help us express the LP in a more concise and intuitive manner [Trevizan, Thiébaux, Santana, et al. 2016].

$$
\begin{array}{lll}
\text{minimise} & \sum_{s \in S, a \in A(s)} x_{s,a} \cdot C(s,a) & \text{(LP6)} \\
\text{subject to} & x_{s,a} \geq 0 \qquad\qquad \forall s \in S, a \in A(s) & \text{(C1)} \\
& in(s) = \sum_{s' \in S, a \in A(s')} x_{s',a} \cdot P(s|s',a) \qquad \forall s \in S & \text{(C2)} \\
& out(s) - in(s) = 0 \qquad\qquad \forall s \in S \setminus (G \cup \{s_0\}) & \text{(C3)} \\
& out(s_0) - in(s_0) = 1 & \text{(C4)} \\
& out(s) = \sum_{a \in A(s)} x_{s,a} \qquad\qquad \forall s \in S \setminus G & \text{(C5)} \\
& \sum_{s_g \in G} in(s_g) = 1 & \text{(C6)}
\end{array}
$$

As mentioned before, LP 6 can be interpreted as a network flow problem. In line with this semantic, constraints have special names:

- (C1) gives *non-negativity constraints*, which ensures that there is no negative flow
- (C2) and (C5) are *helper constraints*, as we have already explained
- (C3)-(C4) are called *preservation of flow constraints*, because they enforce that the amount of flow coming into a particular node or state

should be equal to the amount of flow exiting that node or state – note that constraint (C4) also "pumps in" a flow of 1 into the initial state $s_0$

- (C6) is the *sink constraint*, which ensures that we get the same amount of flow in the goal as was pumped in initially.

Given the optimised solution $x^*$ to LP 6, we define the stochastic policy $\pi^*(s,a) = \frac{x_{s,a}}{out(s)}$. This policy is optimal. Furthermore, it turns out that $\pi^*$ can be interpreted as a deterministic policy, in the sense that $\pi^*(s,a) \in \{0,1\}$. This is due to the fact that practically all LP solvers are like the simplex algorithm in that they explore solutions that lie on vertices of the feasible region. Constraints (C3)-(C5) ensure that assignments of flow $out(s)$ to one variable lie on a vertex, and a solver will generate a deterministic policy.

So far we have been making use of assumption 1 to construct our LPs. The formulation for LP 6 is flexible enough that we can relax this assumption, and deal with dead ends by introducing variables that allow the network flow to leak with some penalty – or equivalently, they can be interpreted as give-up actions as in the give-up transformation (definition 10) [Trevizan, F. Teichteil-Königsbuch, and Thiébaux 2017]. See LP 7.

$$
\begin{array}{lll}
\text{minimise} & \displaystyle\sum_{s\in s, a\in A(s)} x_{s,a}\cdot C(s,a) + \sum_{s\in S} x_s^D \cdot d & \text{(LP7)} \\
\text{subject to} & x_{s,a} \geq 0 \qquad\qquad \forall s\in S, a\in A(s) & \text{(C1)} \\
& in(s) = \displaystyle\sum_{s'\in S, a\in A(s')} x_{s',a}\cdot P(s|s',a) \qquad \forall s\in S & \text{(C2)} \\
& out(s) - in(s) = 0 \qquad\qquad \forall s\in S\setminus(G\cup\{s_0\}) & \text{(C3)} \\
& out(s_0) - in(s_0) = 1 & \text{(C4)} \\
& out(s) = \displaystyle\sum_{a\in A(s)} x_{s,a} + x_s^D \qquad\qquad \forall s\in S\setminus G & \text{(C5)} \\
& \displaystyle\sum_{s_g\in G} in(s_g) + \sum_{s\in S} x_s^D = 1 & \text{(C6)}
\end{array}
$$

## 4.6 Evaluation LP

Now, rather than investigating how we can construct optimal policies, we present a linear program that takes a policy and evaluates it. For this section we do not use assumption 1, i.e. we allow dead ends. Given a potentially partial or improper policy $\pi$, we are often interested in the probability that following $\pi$ will lead us to a goal – rather than a dead end or an undefined

state; and we are interested in assigning some notion of expected cost to the policy.

To construct this evaluation LP we apply the following modifications to LP 7:

- we do not consider all actions $a \in A$, but rather those that are given by our policy
- we do not consider all states $s \in S$, but only the policy envelope $S^\pi$
- we partition $S^\pi$ into $S^\pi_{\text{open}} \cup S^\pi_{\text{closed}}$
- we allow leakage on undefined states – i.e. non-goal open states
- we remove the costs of actions from the objective – we are not trying to optimise the policy.

These modifications yield LP 8.

$$
\begin{aligned}
&\text{minimise} && \sum_{s \in S^\pi_{\text{closed}}} x^D_s && && \text{(LP8)}\\
&\text{subject to} && x_{s,\pi(s)} \geq 0 && \forall s \in S^\pi_{\text{closed}} && \text{(C1)}\\
& && x^D_s \geq 0 && \forall s \in S^\pi_{\text{closed}} && \text{(C2)}\\
& && in(s) = \sum_{s' \in S^\pi_{\text{closed}}} x_{s',\pi(s')} \cdot P(s|s',\pi(s')) && \forall s \in S^\pi && \text{(C3)}\\
& && out(s) - in(s) = 0 && \forall s \in S^\pi_{\text{closed}} \setminus \{s_0\} && \text{(C4)}\\
& && out(s_0) - in(s_0) = 1 && && \text{(C5)}\\
& && out(s) = x_{s,\pi(s)} + x^D_s && \forall s \in S^\pi_{\text{closed}} && \text{(C6)}\\
& && \sum_{s_g \in G} in(s_g) + \sum_{s_f \in S^\pi_{\text{open}} \setminus G} in(s_f) + \sum_{s \in S^\pi_{\text{closed}}} x^D_s = 1 && && \text{(C7)}
\end{aligned}
$$

Notice that *out* is not defined for open states. This is because –like for goal states– the flow does not get pumped further. These allowances for leakage can be seen as give-up actions, or alternatively as another fake action that leads to a "replan sink." Then, the inflow to this sink, namely the second summation in (C7), represents the likelihood of replanning, given we follow $\pi$ starting at $s_0$.

Note also that we do not include the leakage or give-up actions for undefined states in the objective. This is for the same reason that we do not include the costs of actions in the objective: we are simply following the flow determined

by the given policy, and the leakage is determined by the preservation of flow constraints.

Now, given that we start at $s_0$ and follow policy $\pi$, we can read off the

1. probability of reaching a goal with $\sum_{s_g \in G} in(s_g)$,
2. probability of reaching a dead end with $\sum_{s \in S^\pi_{\text{closed}}} x^D_s$,
3. probability of replanning with $\sum_{s_f \in S^\pi_{\text{open}} \setminus G} in(s_f)$,
4. probability of replanning at a particular undefined state $s_f \in S^\pi_{\text{open}} \setminus G$ as $in(s_f)$.

Furthermore, we can compute the expected cost of a policy where dead-end and replan states incur the finite dead-end penalty (as in definition 16), and the MCMP cost (as given in definition 17). The former is computed by

$$D \cdot \sum_{s \in S^\pi_{\text{closed}}} x^D_s + U \cdot \sum_{s \in S^\pi_{\text{open}} \setminus G} s_f,$$

where $D \in \mathbb{R}_{>0}$ is the dead end penalty, and $U \in \mathbb{R}_{>0}$ is penalty for reaching an undefined state. Note that we typically set $U = D$. The MCMP cost is computed by

$$\sum_{s \in S^\pi_{\text{closed}}} x_{s,\pi(s)} \cdot C(s, \pi(s)).$$

# Chapter 5

# Plan Based Column Generation

In this chapter we introduce our novel algorithm PBColgen, which constructs policies by combining plans in a column generation framework. First, we introduce some additional definitions (section 5.1) before moving on to the construction of a system of linear equations that evaluates a given policy as a combination of plans and cycles (section 5.2). This serves as a motivation and starting point for the core of this thesis: we derive the master problem that computes the optimal policy on an SSP as a combination of plans and cycles, and adapt it to a column generation framework (section 5.3). We then discuss some options for solving the deterministic pricing problem that is crucial to the efficiency of PBColgen (section 5.4). Finally, we summarise the properties of PBColgen (section 5.5).

## 5.1 Definitions

For the following work we find it useful to introduce some more definitions.

**Definition 27 (Set of all plans $\Phi$)** Given an SSP $\mathbb{S}$, we denote by $\Phi$ the set of all plans on the determinisation of $\mathbb{S}$.

We need to discuss cycles, so we formally introduce them now.

**Definition 28 (Cycles)** A cycle is defined as a path (in the graph theoretic sense) with an additional arc that connects the first and last nodes

of the path [Ahuja, Magnanti, and Orlin 1993, p. 26]. In the context of SSPs and their determinisations we say that a cycle consists of a state trace and action trace where there are no duplicates in either – except that the first and last state in the state trace must be the same.

Since cycles are described by state trace $\langle s^0, \ldots, s^n \rangle$ and an action trace $\langle a^0, \ldots, a^{n-1} \rangle$ identically to plans, we use the same notation. Note that we don't distinguish cycles by the first state of their trace, so for example $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_1$ is the same as $s_2 \xrightarrow{a_1} s_1 \xrightarrow{a_1} s_2$.

Now, we also define the set of all cycles.

**Definition 29 (Set of all cycles $\mathscr{W}$)** Given an SSP $\mathbb{S}$, we denote by $\mathscr{W}$ the set of all cycles on the determinisation of $\mathbb{S}$.

Then, do discuss transformations between systems of linear equations and linear programs we use the following notion of equivalence.

**Definition 30 (Equivalence for SLEs and LPs)** We say that two linear programs $L$ and $L'$ are *equivalent* if the optimal solution $x^*$ for $L$ can be transformed into an optimal solution of $L'$, and vice versa. Similarly, we say that two systems of linear equations are equivalent if the solution of one can be transformed into a solution of the other.

## 5.2 Evaluating policies as a sum of plans and cycles

To motivate the approach for finding optimal policies, we first look at how to evaluate policies as a sum of plans and cycles.

The set up of the problem is as follows:

- we are given policy $\pi$ and SSP $\mathbb{S}$, and we want to evaluate the policy with plans and cycles
- we assume that $\mathbb{S}$ has no dead ends
- we assume that $\pi$ is proper (because this is a motivation for finding the optimal policy, and the optimal policy on an SSP without dead ends must be proper).

Our starting point is the policy evaluation LP presented in LP 8, which can be interpreted as a network flow across an SSP. Since $\pi$ is proper and therefore closed, we do not need to distinguish between open and closed

states. Furthermore, since $\pi$ is proper, $\pi$ will not reach any dead ends, which allows us to remove the $x_D$ leakage variables. These simplifications give us the system of linear equations presented in SLE 1. This SLE can also be interpreted as a network flow, so the variables $\mathbf{x}$ represent the flow through $\mathbb{S}$.

(SLE1)

$$
\begin{aligned}
& x_{s,\pi(s)} \geq 0 && \forall s \in S^\pi && \text{(C1)} \\
& in(s) = \sum_{s' \in S} x_{s',\pi(s')} \cdot P(s|s', \pi(s')) && \forall s \in S^\pi && \text{(C2)} \\
& out(s) - in(s) = 0 && \forall s \in S^\pi \setminus (G \cup \{s_0\}) && \text{(C3)} \\
& out(s_0) - in(s_0) = 1 && && \text{(C4)} \\
& out(s) = x_{s,\pi(s)} && \forall s \in S^\pi \setminus G && \text{(C5)} \\
& \sum_{s_g \in G} in(s_g) = 1 && && \text{(C6)}
\end{aligned}
$$

We now apply the all-outcomes determinisation to $\mathbb{S}$, to get $\mathbb{S}_{\text{det}}$. We want to transform the flow $\mathbf{x}$ on $\mathbb{S}$ into an equivalent flow $\mathbf{y}$ on $\mathbb{S}_{\text{det}}$. The issue is that the flow described by $\mathbf{x}$ becomes invalid on actions with probabilistic effects, i.e. given $s \in S^\pi$ and $\pi(s) \in A(s)$ such that $\pi(s)$ has multiple effects, we can no longer talk about $x_{s,\pi(s)}$ since the action $\pi(s)$ does not exist in $\mathbb{S}_{\text{det}}$– it has been broken up into $\pi(s) \triangleright s' \quad \forall s' \in \text{supp}(s, \pi(s))$.

We deal with this by constructing a new flow across $\mathbb{S}_{\text{det}}$ with additional constraints on probabilistic actions, such that the flow on the determinised actions corresponds to the probabilities in $\mathbb{S}$; these are called *regrouping constraints* [Trevizan, Thiébaux, and Haslum 2017]. For example, given the SSP in figure 5.1, we want to enforce that the flow across $a_0 \triangleright s_0$ and $a_0 \triangleright s_1$ retains the probabilistic relation of $\mathbb{S}$. We enforce this by adding the constraint

$$\frac{y_{s_0,a_0 \triangleright s_1}}{\rho} = \frac{y_{s_0,a_0 \triangleright s_1}}{1-\rho}.$$

We now generalise regrouping to work for an arbitrary number of effects. It does not make sense to introduce regrouping constraints for actions with a single effect, since these are already deterministic. So, we introduce the notation $A_{>1}$ where $A_{>1}(s) = \{a \in A : |\text{supp}(s,a)| > 1\}$ to distinguish actions that require regrouping and those that do not.

There are multiple ways to express that probabilistic ratio between arbitrarily many effects, we present three of them. For the following, suppose we are considering state $s$ with action $a$, which has $|\text{supp}(s,a)| = n$ with $n > 1$.

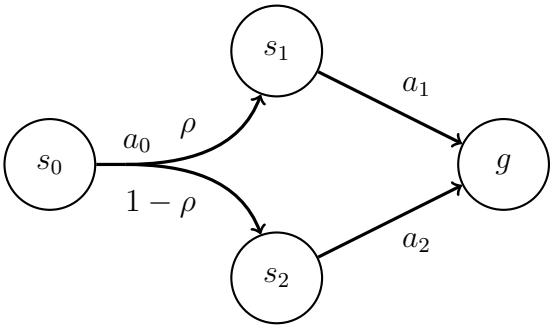



Figure 5.1: SSP with probabilistic effects that we must "regrouped" in a determinisation to avoid a loss of information.

- (one against all) With this approach, we add a constraint for each effect $s'$, where the constraint expresses $s'$ as a fraction of the total flow on $s$ and $a$, i.e.

$$\frac{y_{s,a\triangleright s'}}{P(s'|s,a)} = \sum_{s''\in \mathrm{supp}(s,a)} y_{s,a\triangleright s''} \quad \forall s' \in \mathrm{supp}(s,a).$$

- (hub and spoke) We select one effect as a "hub," and use it to express all other effects – the "spokes." We require a function $\mathrm{hub} : S \times A(s) \to \mathrm{supp}(s,a)$ that deterministically selects a unique representative from the effects of $a$. Conversely, we also define a function that gives us all non-hub effects: $\mathrm{spoke}(s,a) = \mathrm{supp}(s,a) \setminus \{\mathrm{hub}(s,a)\}$. Then, we add the following constraints

$$\frac{y_{s,a\triangleright \mathrm{hub}(s,a)}}{P(\mathrm{hub}(s,a)|s,a)} = \frac{y_{s,a\triangleright s'}}{P(s'|s,a)} \quad \forall s' \in \mathrm{spoke}(s,a).$$

- (chain) Suppose there is an index function $\iota : \{1,\dots,n\} \to \mathrm{supp}(s,a)$ which is bijective. In practice, this will simply be the order in which we encounter each effect. We define pairwise constraints in a chain-like fashion as follows:

$$\frac{y_{s,a\triangleright \iota(i)}}{P(\iota(i)|s,a)} = \frac{y_{s,a\triangleright \iota(i+1)}}{P(\iota(i+1)|s,a)} \quad \forall i \in \{1,\dots,n-1\}$$

The one-against-all formulation introduces $n$ constraints, whereas the hub-and-spoke and chain formulations introduce $n-1$. So, the only practical difference between these constraints is in how convenient it is to construct them.

For now, we use the hub-and-spoke method, as it is will turn out to be convenient for column generation. This gives us SLE 2.

(SLE2)

$$
\begin{aligned}
& y_{s,a\triangleright s'} \geq 0 && \forall s \in S^\pi, a \in A(s), s' \in \text{supp}(s,a) && \text{(C1)}\\
& in(s) = \sum_{s' \in S} \sum_{s'' \in \text{supp}(s',\pi(s'))} y_{s',pi(s')\triangleright s''} && \forall s \in S^\pi && \text{(C2)}\\
& out(s) - in(s) = 0 && \forall s \in S^\pi \setminus (G \cup \{s_0\}) && \text{(C3)}\\
& out(s_0) - in(s_0) = 1 && && \text{(C4)}\\
& out(s) = \sum_{s' \in \text{supp}(s,\pi(s))} y_{s,\pi(s)\triangleright s'} && \forall s \in S^\pi \setminus G && \text{(C5)}\\
& \tfrac{y_{s,a\triangleright \text{hub}(s,a)}}{P(\text{hub}(s,a)|s,a)} = \tfrac{y_{s,a\triangleright s'}}{P(s'|s,a)} && \forall s \in S^\pi, a \in A_{>1}(s), s' \in \text{spoke}(s,a) && \text{(C6)}\\
& \sum_{s_g \in G} in(s_g) = 1 && && \text{(C7)}
\end{aligned}
$$

**Lemma 1** *SLE 1 is equivalent to SLE 2*

**Proof 1** *A solution for* $\mathbf{x}$ *can be transformed into a solution for* $\mathbf{y}$ *by*

$$y_{s,\pi(s)\triangleright s'} = x_{s,\pi(s)} \cdot P(s'|s,\pi(s)) \quad \forall s \in S, s' \in supp(s,a).$$

*This transformed solution satisfies the regrouping constraints by construction, and satisfies the goal constraint trivially. The preservation-of-flow constraints are satisfied because SLE 1 has*

$$x_{s,\pi(s)} = \sum_{s' \in S} x_{s',\pi(s')} \cdot P(s|s',\pi(s'))$$

*which by our transformation gives*

$$\sum_{s' \in supp(s,\pi(s))} y_{s,\pi(s)\triangleright s'} = x_{s,\pi(s)} = \sum_{s' \in S} x_{s',\pi(s')} \cdot P(s|s',\pi(s')) = \sum_{s' \in S} y_{s',\pi(s')\triangleright s}$$

*as desired.*

*A solution for* $\mathbf{y}$ *can be transformed into a solution for* $\mathbf{x}$ *by*

$$x_{s,\pi(s)} = \sum_{s' \in supp(s,\pi(s))} y_{s,\pi(s)\triangleright s'}.$$

*That the transformed solution satisfies all constraints of SLE 1 is given by similar arguments as before.*

Now that we are dealing with arc flow, i.e. flow over individual actions in a deterministic shortest path problem $\mathbb{S}_{\text{det}}$, we can decompose it into flow over plans and cycles [Ahuja, Magnanti, and Orlin 1993, p. 80]. So, given the set of all plans $\Phi$ and the set of all cycles $\mathscr{W}$ on $\mathbb{S}_{\text{det}}$, we can express any flow $y_{s,a\triangleright s'}$ as

$$y_{s,a\triangleright s'} = \sum_{f\in\Phi\cup\mathscr{W}} \lambda_f \cdot \delta_{s,a\triangleright s'}(f)$$

where $\lambda_f$ is the variable that represents the amount of flow we route through plan/cycle $f$, and the indicator function $\delta_{s,a}(f)$ is a constant denoting how many times $s \xrightarrow{a\triangleright s'} s'$ occurs in $f$. Since we are considering plans and cycles –which do not allow the repetition of actions– this is

$$\delta_{s,a}(f) = \begin{cases} 1 & \text{if } s \xrightarrow{a\triangleright s'} s' \text{ occurs in } f \\ 0 & \text{else.} \end{cases}$$

This equivalence allows us to get a new SLE by substituting any occurrence of the equivalence in SLE 2. We notice however, that the flow constraints in SLE 2 are redundant in our new SLE, since plans and cycles must inherently obey flow constraints. But now, we must fix the amount of flow coming into the network, which we can do by adding a convexity constraint to the plans. Note that the convexity constraint does not include cycles. This is because we do not explicitly want flow to pass through cycles, they are only present to make the regrouping constraints feasible, since plans cannot have cycles but policies can. After this process, we get SLE 3.

(SLE3)

$$\lambda_f \geq 0 \qquad \forall f \in \Phi\cup\mathscr{W} \quad \text{(C1)}$$

$$\sum_{\phi\in\Phi} \lambda_\phi = 1 \quad \text{(C2)}$$

$$\frac{\sum_{f\in\Phi\cup\mathscr{W}} \lambda_f \cdot \delta_{s,a\triangleright \text{hub}(s,a)}(f)}{P(\text{hub}(s,a)|s,a)} = \frac{\sum_{f\in\Phi\cup\mathscr{W}} \lambda_f \cdot \delta_{s,a\triangleright s'}(f)}{P(s'|s,a)}$$

$$\forall s \in S, a \in A_{>1}(s), s' \in \text{spoke}(s,a) \quad \text{(C3)}$$

We claim that this transformation retains equivalence.

**Lemma 2** *SLE 2 is equivalent to SLE 3*

**Proof 2** *We rely on theorem 3.5 from [Ahuja, Magnanti, and Orlin 1993, p. 80], which gives us the property that arc flow can in general be decomposed*

*into flow over paths and cycles. We particularise the theorem to arc flow on SSPs. The initial state $s_0$ is a deficit node, and the remaining closed states are bound by flow preservation constraints. We assume $\pi$ to be proper, which means only goal states may be excess nodes. Subsequently, the paths generated by the algorithm are in fact plans. Therefore, the arc flow solution of SLE 2 can be expressed in terms of plans and cycles, and is thus a solution for SLE 3.*

*The same theorem immediately gives us that plans and cycles can be expressed in terms of arc flow, and so a solution for SLE 3 gives us an optimal solution for SLE 2.*

So, a solution for SLE 3 can be transformed into a solution for SLE 2, which in turn can be transformed into a solution for SLE 1. By itself, this new formulation is impractical, since it becomes infeasible to compute $\Phi$ and $\mathscr{W}$ for larger problems. However, this formulation is amenable to column generation, as we shall discuss in finding the optimal policy.

## 5.3 Optimal policy as sum of plans and cycles

By setting up a system of linear equations that evaluates a policy, we have laid the groundwork for a linear program capable of finding the optimal policy. We only need to slightly modify SLE 3 to obtain the master problem. Then, we discuss how column generation can be used to solve it.

### 5.3.1 Master Problem

The set up for our problem is as follows:

- given an SSP $\mathbb{S}$, we want to find the best policy in terms of plans and cycles
- we assume that $\mathbb{S}$ has no dead ends, and so the optimal policy must be proper.

Knowing that the optimal policy must be proper, we can use very similar arguments to the ones we used to transform the network flow based evaluation SLE 1 into the plan and cycle based evaluation SLE 3, to now transform the dual LP 6 which finds optimal policies with network flow into an LP that finds optimal policies in terms of plans and cycles.

As the arguments are virtually identical, we omit them. However, we do need to consider the difference that the dual LP has an objective. It gets transformed as such

$$\sum_{s\in S, a\in A(s)} x_{s,a}\cdot C(s,a)$$
$$\downarrow$$
$$\sum_{\substack{s\in S, a\in A(s),\\ s'\in \mathrm{supp}(s,a)}} y_{s,a\triangleright s'}\cdot C(s, a\triangleright s')$$
$$\downarrow$$
$$\sum_{f\in\Phi\cup\mathscr{W}}\sum_{\substack{s\in S, a\in A(s),\\ s'\in \mathrm{supp}(s,a)}} \lambda_f\cdot\delta_{s,a}(f)\cdot C(s,a)$$
$$\downarrow$$
$$\sum_{f\in\Phi\cup\mathscr{W}}\lambda_f\cdot C(f)$$

where $C(f)$ is the cost of plan or cycle $f$, as given by the sum of the action costs, i.e.

$$C(f)=\sum_{s_i\xrightarrow{a_i\triangleright s_{i+1}}s_{i+1}\in f} C(s_i, a_i\triangleright s_{i+1})=\sum_{\substack{s\in S, a\in A(s)\\ s'\in\mathrm{supp}(s,a)}}\delta_{s,a}(f)\cdot C(s,a).$$

Note that the cost of a cycle considers the cost of each action only once.

The ensuing linear program is given in LP 9.

$$\begin{aligned}
&\text{minimise} && \sum_{f\in\Phi\cup\mathscr{W}}\lambda_f\cdot C(f) && && \text{(LP9)}\\
&\text{subject to} && \lambda_f\geq 0 && \forall f\in\Phi\cup\mathscr{W} && \text{(C1)}\\
& && \textstyle\sum_{\phi\in\Phi}\lambda_\phi = 1 && && \text{(C2)}\\
& && \frac{\sum_{f\in\Phi\cup\mathscr{W}}\lambda_f\cdot\delta_{s,a\triangleright\mathrm{hub}(s,a)}(f)}{P(\mathrm{hub}(s,a)|s,a)} = \frac{\sum_{f\in\Phi\cup\mathscr{W}}\lambda_f\cdot\delta_{s,a\triangleright s'}(f)}{P(s'|s,a)} && &&\\
& && \forall s\in S, a\in A_{>1}(s), s'\in\mathrm{spoke}(s,a) && && \text{(C3)}
\end{aligned}$$

This formulation can be interpreted as a network flow optimisation problem, where we route flow through plans and cycles – the amount of flow routed through plan or cycle $f$ is indicated by the variable $\lambda_f$. Constraints have the following names and semantics:

- (C1) gives *non-negativity constraints*, since a plan or action can not have negative flow routed through it
- (C2) is the *convexity constraint*, which ensures that our plans are a convex sum – this encodes that there should be a flow of 1 pumped from start to end
- (C3) are called the *regrouping constraints* which ensure that flow is routed through plans and cycles in accordance to the probabilities on the SSP; as discussed earlier.

This allows us to refer to particular constraints, e.g. the regrouping constraint on $s, a, s'$ is called $\text{regroup}(s, a, s')$.

### 5.3.2 Column Generation

LP 9 optimises over the entire set of plans $\Phi$ and cycles $\mathscr{W}$, which becomes impossible to even explicitly enumerate as the problem increases in size. This is precisely the kind of problem for which column generation is useful, so we apply it. To obtain the reduced master problem we do not change any constraints, but allow the set of plans and cycles to be incomplete, i.e. we set up the reduced sets $\hat{\Phi} \subseteq \Phi$ and $\hat{\mathscr{W}} \subseteq \mathscr{W}$. Then, we get LP 10.

$$\begin{aligned}
&\text{minimise} && \sum_{f \in \hat{\Phi} \cup \hat{\mathscr{W}}} \lambda_f \cdot C(f) && && \text{(LP10)}\\
&\text{subject to} && \lambda_f \geq 0 && \forall f \in \hat{\Phi} \cup \hat{\mathscr{W}} && \text{(C1)}\\
& && \textstyle\sum_{\phi \in \hat{\Phi}} \lambda_\phi = 1 && && \text{(C2)}\\
& && \frac{\sum_{f \in \hat{\Phi} \cup \hat{\mathscr{W}}} \lambda_f \cdot \delta_{s,a \triangleright \text{hub}(s,a)}(f)}{P(\text{hub}(s,a)|s,a)} = \frac{\sum_{f \in \hat{\Phi} \cup \hat{\mathscr{W}}} \lambda_f \cdot \delta_{s,a \triangleright s'}(f)}{P(s'|s,a)} && && \\
& && \forall s \in S, a \in A_{>1}(s), s' \in \text{spoke}(s,a) && && \text{(C3)}
\end{aligned}$$

In order to determine which plans or cycles to add as columns we need to solve the pricing problem at each iteration of column generation. Suppose that the reduced master problem has been solved, and we have the vector of dual variables $\boldsymbol{\Delta}^*$ from the optimal solution. Note that we denote dual variables associated with a constraint $c$ by $\Delta^*_c$. For example, the regrouping constraint for $s_i, a_i \in A(s_i), s_j \in \text{supp}(s_i, a_i)$ is given by $\Delta^*_{\text{regroup}(s_i,a_i,s_j)}$. We want to find a plan or cycle with negative reduced cost, which is in our problem given by

$$rc(f, \boldsymbol{\Delta}^*) = C(f) - \text{rc-hubs}(f, \boldsymbol{\Delta}^*) + \text{rc-spokes}(f, \boldsymbol{\Delta}^*) - \text{rc-convexity}(f, \boldsymbol{\Delta}^*)$$

where

$$\text{rc-hubs}(f, \boldsymbol{\Delta}^*) = \sum_{s \xrightarrow{a \triangleright s'} s' \in f : s' = \text{hub}(s,a)} \left( \sum_{s'' \in \text{spoke}(s,a)} \frac{\lambda_f \cdot \delta_{s,a}(f)}{P(s'|s,a)} \cdot \Delta^*_{\text{regroup}(s',a,s'')} \right)$$

$$\text{rc-spokes}(f, \boldsymbol{\Delta}^*) = \sum_{s \xrightarrow{a \triangleright s'} s' \in f : s' \in \text{spoke}(s,a)} \frac{\lambda_f \cdot \delta_{s,a}(f)}{P(s'|s,a)} \cdot \Delta^*_{\text{regroup}(\text{hub}(s,a),a,s')}$$

$$\text{rc-convexity}(f, \boldsymbol{\Delta}^*) = \begin{cases} \Delta^*_{\text{convexity}} & \text{if } f \in \Phi \\ 0 & \text{if } f \in \mathscr{W}. \end{cases}$$

Observe the structure of $rc(f, \boldsymbol{\Delta}^*)$ – we can factor out summations over states and actions in the plan or cycle $f$ to obtain

$$rc(f, \boldsymbol{\Delta}^*) = \sum_{s \xrightarrow{a \triangleright s'} s' \in f} \Big( C(s, a \triangleright s') + \text{rc-spoke}(s, a \triangleright s', \boldsymbol{\Delta}^*) - \text{rc-hub}(s, a \triangleright s', \boldsymbol{\Delta}^*) \Big) + \text{rc-convexity}(f, \boldsymbol{\Delta}^*)$$

where

$$\text{rc-hub}(s, a \triangleright s', \boldsymbol{\Delta}^*) = \begin{cases} \sum_{s'' \in \text{spoke}(s,a)} \frac{\lambda_f \cdot \delta_{s,a}(f)}{P(s'|s,a)} \cdot \Delta^*_{\text{regroup}(s',a,s'')} & \text{if } s' = \text{hub}(s,a) \\ 0 & \text{else} \end{cases}$$

$$\text{rc-spoke}(s, a \triangleright s', \boldsymbol{\Delta}^*) = \begin{cases} \frac{\lambda_f \cdot \delta_{s,a}(f)}{P(s'|s,a)} \cdot \Delta^*_{\text{regroup}(\text{hub}(s,a),a,s')} & \text{if } s \in \text{spoke}(s,a) \\ 0 & \text{else} \end{cases}$$

$$\text{rc-convexity}(f, \boldsymbol{\Delta}^*) = \begin{cases} \Delta^*_{\text{convexity}} & \text{if } f \in \Phi \\ 0 & \text{if } f \in \mathscr{W}. \end{cases}$$

With this new cost, we can create a new deterministic planning problem that reflects the pricing cost. To do this, first consider the all-outcomes determinisation of $\mathbb{S}$ to get $\mathbb{S}_{\text{det}} = \langle S, s_0, G, A, T, C \rangle$. Then, we construct a new negative-weighted deterministic planning problem $\mathbb{P} = \langle S, s_0, G, A, T, C' \rangle$ which is identical to $\mathbb{S}_{\text{det}}$ except in the cost. The cost of $\mathbb{P}$ is given by

$$C'(s, a \triangleright s') = C(s, a \triangleright s') + \text{rc-spoke}(s, a \triangleright s', \boldsymbol{\Delta}^*) - \text{rc-hub}(s, a \triangleright s', \boldsymbol{\Delta}^*).$$

This cost is not complete yet, because we must also apply a shift of $\text{rc-convexity}(f, \boldsymbol{\Delta}^*)$ to plans and cycles – noting that this expression is 0 for cycles.

Now, if we find a plan or cycle $f$ such that its cost on $\mathbb{P}$ scaled by the convexity constraint is negative, i.e. if

$$C'(f) + \text{rc-convexity}(f, \mathbf{\Delta}^*) < 0,$$

then we know that $f$ has a negative reduced cost, and thus will improve the objective of our reduced master problem if we add it. We say that a solution for the pricing problem $\mathbb{P}$ is such a plan or cycle.

So far we have not addressed the issue that using a reduced set of plans and cycles means there is no guarantee that the RMP is feasible. Consider the first iteration of our algorithm – we propose that $\hat{\Phi} \cup \hat{\mathscr{W}} = \{\phi\}$ where $\phi$ is some plan. If the casted plan $\pi_\phi$ is not closed, our RMP is infeasible. There are multiple ways we can deal with this case, but for now we use Farkas cost which –in a similar way to the reduced cost– lets us find plans and cycles that help to make the problem feasible. The reduced and Farkas costs are structurally nearly identical, so we omit the derivation. The Farkas cost $fc$ is given by

$$\begin{aligned} fc(f, \mathbf{\Delta}^*) = &\sum_{s \xrightarrow{a \triangleright s'} s' \in f} \Big( \text{fc-hub}(s, a \triangleright s', \mathbf{\Delta}^*) - \text{fc-spoke}(s, a \triangleright s', \mathbf{\Delta}^*) \Big) \\ &- \text{fc-convexity}(f, \mathbf{\Delta}^*) \end{aligned}$$

where

$$\text{fc-hub}(s, a \triangleright s', \mathbf{\Delta}^{\text{ray}}) = \begin{cases} \displaystyle\sum_{s'' \in \text{spoke}(s,a)} \frac{\lambda_f \cdot \delta_{s,a}(f)}{P(s'|s,a)} \cdot \Delta^{\text{ray}}_{\text{regroup}(s',a,s'')} & \text{if } s' = \text{hub}(s,a) \\ 0 & \text{else} \end{cases}$$

$$\text{fc-spoke}(s, a \triangleright s', \mathbf{\Delta}^{\text{ray}}) = \begin{cases} \frac{\lambda_f \cdot \delta_{s,a}(f)}{P(s'|s,a)} \cdot \Delta^{\text{ray}}_{\text{regroup}(\text{hub}(s,a),a,s')} & \text{if } s \in \text{spoke}(s,a) \\ 0 & \text{else} \end{cases}$$

$$\text{fc-convexity}(f, \mathbf{\Delta}^{\text{ray}}) = \begin{cases} \Delta^{\text{ray}}_{\text{convexity}} & \text{if } f \in \Phi \\ 0 & \text{if } f \in \mathscr{W}, \end{cases}$$

where $\Delta^{\text{ray}}_c$ refers to the dual ray associated with constraint $c$.

We construct the Farkas problem $\mathbb{F} = \langle S, s_0, G, A, T, C'' \rangle$. As with $\mathbb{P}$, all elements of the tuple except the cost are identical to $\mathbb{S}_{\text{det}}$. The cost $C''$ is now given by

$$C''(s, a \triangleright s') = \text{fc-hub}(s, a \triangleright s', \mathbf{\Delta}^{\text{ray}}) - \text{fc-spoke}(s, a \triangleright s', \mathbf{\Delta}^{\text{ray}}).$$

Now, we know that a plan or cycle $f$ that satisfies

$$C''(f) - \text{fc-convexity}(f, \mathbf{\Delta}^{\text{ray}}) < 0$$

corresponds to a column that should be added to $\hat{\Phi} \cup \hat{\mathscr{W}}$ – we call such plans and cycles solutions to $\mathbb{F}$.

Finally, we have all components for our column generation algorithm. We populate our initial reduced set $\hat{\Phi} \cup \hat{\mathscr{W}}$ with some plan, try to solve the RMP, solve the pricing or Farkas cost if the RMP was feasible or infeasible, respectively, and repeat until the pricing problem yields no solution. See algorithm 3.

### 5.3.3 Give-up action

An alternative way to deal with infeasible reduced master problems is by introducing a single *give-up action*, which allows our planner to "give up" rather than have an infeasible problem. This is implemented by adding an artificial plan $\phi_{\text{give-up}}$ to $\hat{\Phi}$, where $\phi_{\text{give-up}}$ goes from starting state $s_0$ straight to some goal, and the cost of this plan is the dead-end penalty, i.e. $C(\phi_{\text{give-up}}) = D$. This is essentially a big-M approach for feasibility [Lübbecke and Desrosiers 2005, pp. 4–5].

This modification means that our RMP is never infeasible, since $\lambda_{\phi_{\text{give-up}}} = 1$ is always a solution. So, we can remove the steps for Farkas pricing from our algorithm.

If column generation terminates and our optimal solution has $\lambda_{\phi_{\text{give-up}}} = 0$ then the solution is clearly optimal for our original formulation in LP 9, and thus yields an optimal policy for $\mathbb{S}$. This is guaranteed as long as we select our dead-end penalty to be sufficiently large, such that our algorithm prefers to construct a closed proper policy rather than giving update [Kolobov, Mausam, and Weld 2012].

### 5.3.4 Allowing leakage for unavoidable dead ends

Now, we generalise a single give-up action into give-up actions for all effects. This modification has benefits to our algorithm as an online solver, and enables it to solve problems in the absence of the reachability assumption (assumption 1).

The generalisation of the single give-up action can also be interpreted as "leakage" in the network flow. So, we allow leakage on all actions, but such

**Algorithm 3:** Basic PBColgen

**Input:** SSP $\mathbb{S} = \langle S, s_0, G, A, P, C \rangle$

**Output:** Optimal usage for plans and cycles

```
// Set up initial RMP
```

$\phi \leftarrow$ some plan on all-outcomes determinisation $\mathbb{S}_{\text{det}}$

$\hat{\Phi} \cup \hat{\mathscr{W}} \leftarrow \{\phi\}$

update RMP with $\hat{\Phi} \cup \hat{\mathscr{W}}$

```
// Run column generation iteration
```

**while** *true* **do**

  try to solve RMP

  **if** *RMP is infeasible* **then**

    construct Farkas problem $\mathbb{F}$ with $\mathbf{\Delta}^{\text{ray}}$

    plan/cycle $f \leftarrow$ solution of $\mathbb{F}$

    **if** $f$ *is a plan* **then**

      $\hat{\Phi} \leftarrow \hat{\Phi} \cup \{f\}$

    **if** $f$ *is a cycle* **then**

      $\hat{\mathscr{W}} \leftarrow \hat{\mathscr{W}} \cup \{f\}$

  **if** *RMP is feasible* **then**

    construct pricing problem $\mathbb{P}$ with $\mathbf{\Delta}^{*}$

    **if** $\mathbb{P}$ *has no solution* **then**

      **return** *solution for RMP*

    plan/cycle $f \leftarrow$ solution of $\mathbb{P}$

    **if** $f$ *is a plan* **then**

      $\hat{\Phi} \leftarrow \hat{\Phi} \cup \{f\}$

    **if** $f$ *is a cycle* **then**

      $\hat{\mathscr{W}} \leftarrow \hat{\mathscr{W}} \cup \{f\}$

leakage is penalised by a dead-end penalty $D$ in the objective as before. We implement this by adding variables for each possible effect of each applicable action, introducing them to regrouping constraints in such a way that give-up actions can be selected if the problem would otherwise be infeasible, and finally adding these new variable's cost to the objective as a disincentive.

The hub-and-spoke formulation of regrouping constraints becomes unwieldy for this. If $\hat{\Phi} \cup \hat{\mathscr{W}} \subsetneq \Phi \cup \mathscr{W}$ then in

$$\frac{\sum_{f \in \hat{\Phi} \cup \hat{\mathscr{W}}} \lambda_f \cdot \delta_{s,a \triangleright \mathrm{hub}(s,a)}(f)}{P(\mathrm{hub}(s,a)|s,a)} = \frac{\sum_{f \in \hat{\Phi} \cup \hat{\mathscr{W}}} \lambda_f \cdot \delta_{s,a \triangleright s'}(f)}{P(s'|s,a)}$$

for some $s \in S, a \in A(s), s' \in \mathrm{spoke}(s,a)$, it is not clear which side of the expression is larger. This makes it impossible to express the difference with one non-negative variable, so instead we must introduce a non-negative leakage variable to both sides. (Note that the variables must be non-negative, otherwise their contribution to the objective wouldn't make sense.) In short, the hub-and-spoke formulation would force us to add two constraints for each leakage variable, which is undesirable in terms of performance.

It is more convenient to use the one-against-all formulation. Consider the following constraint schema

$$\frac{\sum_{f \in \hat{\Phi} \cup \hat{\mathscr{W}}} \lambda_f \cdot \delta_{s,a \triangleright s'}(f) + \text{give-up}_{s,a \triangleright s'}}{P(s'|s,a)} = \sum_{s'' \in \mathrm{supp}(s,a)} \left( \text{give-up}_{s,a \triangleright s''} + \sum_{f \in \hat{\Phi} \cup \hat{\mathscr{W}}} \lambda_f \cdot \delta_{s,a \triangleright s'}(f) \right)$$

for all $s \in S, a \in A(s), s' \in \mathrm{supp}(s,a)$. As the give-ups occur on both sides, these constraints are always satisfiable.

By allowing the planner to give up on any state we gain stronger anytime performance compared to the other formulations, since it is able to present partial solutions immediately. Even in the first iteration with a single plan $\phi$ in $\hat{\Phi} \cup \hat{\mathscr{W}}$, this formulation is able to return the casted plan $\pi_\phi$ – whereas the Farkas cost version would potentially need to solve multiple Farkas problems to make the policy closed and the RMP feasible; and the version with a single give-up action would return the useless policies that use the give-up action until the real policy can be closed.

For the final advantage of this formulation, we drop assumption 1 – i.e. we allow dead ends both avoidable and unavoidable. The other two formulations

are able to deal with avoidable dead ends in the sense that they will be able to terminate and find an optimal solution that avoids them. If we consider problems with unavoidable dead ends, this is no longer true. This formulation however, is able to provide a policy given unavoidable dead ends – in particular, setting up finite cost penalties for give-ups and dead ends suggests that this formulation will optimise improper policies with respect to the expected cost with finite penalty metric given in definition 16.

### 5.3.5 Extracting a policy

Now we show how a solution for the final iteration can be transformed into an optimal policy, and in some cases a non-final iteration can be transformed into some policy; however, the amount of flow being routed through plans and cycles does not give this immediately.

To extract a policy, we set $\pi(s) = a$ if $\lambda_f > 0$ for some $f$ where $s \xrightarrow{a \triangleright s'} s'$ occurs in $f$, for some $s' \in \text{supp}(s, a)$. Note that the policy being represented by the RMP solution may be stochastic. This is not a problem, as an optimal stochastic policy $\pi_{\text{stoch}}$ can be transformed into an optimal deterministic policy $\pi_{\text{det}}$ (on SSPs) by arbitrarily picking one of the stochastic policies non-zero probability actions, i.e. $\pi_{\text{det}}(s) =$ some $a$ where $\pi_{\text{stoch}}(s, a) > 0$. To justify this, recall that the original dual LP (LP 6) finds a deterministic policy because we will receive a vertex as a solution (section 5). Stochastic policies lie on an edge (or facet in higher dimensions) rather than a vertex, which means that they are convex combinations of deterministic policies. By selecting only one action with non-zero flow we select one of the vertices associated with the edge on which the optimal stochastic policy lies, and we are thereby guaranteed an optimal solution.

To make this method more efficient we only consider the policy envelope by performing a depth-first search where successor states of $s$ are the possible effects of newly defined $\pi(s)$. See algorithm 4 for details.

## 5.4 Solving the pricing problem

The column generation algorithm relies on a solver that guarantees completeness on the reduced cost and Farkas problems, and it is clear that the performance of our algorithm depends on this algorithm being efficient. Here, we will not distinguish between the reduced cost pricing problem and the Farkas pricing problem since the problem structures are identical. So, given a pricing problem where negative costs and negative cycles are possible, we

**Algorithm 4:** Policy extraction algorithm

**Input:** SSP $\mathbb{S} = \langle S, s_0, G, A, P, C \rangle$, PBColgen solution over $\lambda$

**Output:** Corresponding policy $\pi$

```
// Determine which state action pairs have positive flow
```

has-flow$(s, a) \leftarrow$ `false` $\forall s \in S, a \in A(s)$

**for** plan or cycle $s^0 \xrightarrow{a_0} \cdots s^n = f \in \hat{\Phi} \cup \hat{\mathscr{W}}$ **do**

  **if** $\lambda_f > 0$ **then**

    has-flow$(s^i, a_i) \leftarrow$ `true` $\forall i \in [0, n-1]$

```
// Traverse through policy envelope, and define open states (if we
   can)
```

frontier $\leftarrow \{s_0\}$

seen $\leftarrow \{s_0\}$

**while** frontier $\neq \emptyset$ **do**

  pop frontier into state $t$

  `// Define policy by first action with positive flow`

  **for** $a \in A(t)$ **do**

    **if** has-flow$(t, a)$ **then**

      $\pi(t) \leftarrow a$

      break out of for loop

  `// Add effects of` $\pi(t)$ `to the frontier`

  **for** $s' \in \text{supp}(t, \pi(t)) \setminus$ seen **do**

    frontier $\leftarrow$ frontier $\cup \{s'\}$

    seen $\leftarrow$ seen $\cup \{s'\}$

**return** $\pi$

must find a negative plan or cycle, if it exists. Note that we do not care about the most negative solution, we only need to guarantee that we find a negative solution if it exists.

### 5.4.1 Bellman-Ford

The classic algorithm for finding shortest paths on graphs with negative weighted edges and detecting negative cycles is the Bellman-Ford algorithm [Cormen et al. 2009, p. 651]. This is a dynamic programming algorithm that iteratively updates the cheapest path to a goal. Note that this algorithm is in a sense the deterministic equivalent of the Value Iteration algorithm (section 3.1).

See the basic version of our adaptation of Bellman-Ford to pricing problems in algorithm 5. Since we are not given a graph in the classic sense, rather than iterating over edges we traverse all possible outcomes of each action. This is in essence a traversal over the all-outcomes determinisation, but we save computation by avoiding the explicit transformation.

To make this algorithm a bit more efficient for our particular problem we added the following modifications.

We do not iterate over all $s \in S$, but rather we iterate over all $s$ that have best-cost$(s) \neq \infty$. This makes sense because any successor of $s$ with best-cost$(s) = \infty$ will not be updated, and this saves us a lot of computation in large problems, since we implicitly only consider the reachable space.

Another simple optimisation is to check in each iteration of the outer loop if any states have had their best-cost updated. If not, we know that we have converged and there are no negative cycles, and so we can terminate.

Upon termination our algorithm returns a negative cycle if it exists, and otherwise the cheapest plan $\phi$. If the $\phi$ does not satisfy $C(\phi)$+rc-convexity$(\phi, \mathbf{\Delta}^*) < 0$ (or the equivalent Farkas cost), then we know that there is no solution.

### 5.4.2 Modified Bellman-Ford

The Bellman-Ford algorithm searches for the optimal plan, but we only require some negative plan or cycle. This difference in problem specification enables us to modify Bellman-Ford in a way that provides substantial speed-up.

First, we make the observation that $\Delta^*_{\text{convexity}} > 0$ for any RMP. Remember that dual variables represent shadow costs, i.e. how much we can improve

**Algorithm 5:** Bellman-Ford on SSP with Cycle Extraction

**Input:** SSP $\mathbb{S} = \langle S, s_0, G, A, P, C \rangle$

**Output:** Shortest plan or a negative cycle

```
// Initialise maps
parent(s) = ∅   ∀s ∈ S
causal-action(s) = ∅   ∀s ∈ S
best-cost(s) = ∞   ∀s ∈ S \ {s_0}
best-cost(s_0) = 0
// Find minimal plans
do |S| times
    for s ∈ S do
        for a ∈ A(s) do
            for s' ∈ supp(s, a) do
                if best-cost(s) + C(s, a) < best-cost(s') then
                    parent(s') ← s
                    causal-action(s') ← a
                    best-cost(s') ← best-cost(s) + C(s, a)
// Check for negative cycles
for s ∈ S do
    for a ∈ A(s) do
        for s' ∈ supp(s, a) do
            if best-cost(s) + C(s, a) < best-cost(s') then
                return reconstructed cycle from s'
cheapest-goal ← argmin_{s∈G} best-cost(s)
return reconstructed plan from cheapest-goal
```

the solution by relaxing the associated constraint. The constraint associated with convexity is what forces us to pump any flow through plans, so relaxing it enables the cheapest possible solution of pumping zero flow through all plans and cycles. As we require the base SSP $\mathbb{S}$ to have positive costs, the objective associated with pumping flow through any plans or cycles must be positive, and so the convexity constraint must be active, giving us that indeed $\Delta^*_{\text{convexity}}$ must be positive.

This observation makes it safe to set the initial cost of $s_0$ to $\Delta^*_{\text{convexity}}$, i.e. best-cost$(s_0) = \Delta^*_{\text{convexity}}$. This means that plans that detected by our algorithm already have the plan shift included – and so, if a plan is negative we know that it is a solution to the pricing problem.

The upshot of this change is that we can terminate as soon as a goal state $g$ has best-cost$(g) < 0$. When best-cost$(g) < 0$ we know that there is either a negative plan that leads to $g$, or a negative cycle has formed, and there is a plan from it to $g$. We can determine which case we are dealing with by following the parents of $g$, and either case we can return a solution to the pricing problem.

This early termination allows our algorithm to terminate before iterating for $|S|$ times. In practice, we found this to improve solving time dramatically.

Note however, that Bellman-Ford even with our modification has a major bottle-neck. In particular, the main bottleneck is deriving the upper bound for the number of iterations $|S|$. In PPDDL formulations, the number of states in the grounded SSP is exponential with respect to original problem, but a lot of these are unreachable. We are only interested in the number of states in the reachable space, but computing this becomes infeasibly expensive in terms of time and memory for large problems.

### 5.4.3 A* and Admissible Heuristics

A* is the go-to planner for solving deterministic problems optimally [Russell and Norvig 2010, p. 93]. Given an admissible heuristic A* guarantees that it will find an optimal plan, and with sufficiently informative heuristics A* has very strong performance.

However, A* is not able to detect cycles, and there is no work that we are aware of that provides admissible and informative heuristics on problems with negative costs.

Our solution to the former problem is to run A* as the sole deterministic

solver in the absence of cycles; if our problem does contain cycles, we run A* to try to find any cheap plans, and then run Bellman-Ford if A* failed to find a plan with negative reduced cost. This approach guarantees completeness, and can be substantially faster. Note that A* assigns a best cost to each state, as determined by the shortest path to that state so far. This means that we don't have to restart Bellman-Ford from scratch, but we can "warm start" it with the best-costs from A*. Such a warm start has no effect on the classical Bellman-Ford algorithm since we need to iterate $|S|$ times regardless, but our modified version may benefit.

For the latter issue – the absence of admissible heuristics for problems with negative costs, we introduce a simple heuristic. To construct an admissible heuristic we want to find a lower bound for the cost of the cheapest plan. The cheapest possible plan on the pricing problem would take all negative cost actions, and ignore the positive cost actions, i.e.

$$m = \sum_{s \in S, a \in A(s), s' \in \text{supp}(s,a)} \min\{C_{\text{pricing}}(s, a \triangleright s'), 0\}.$$

However, we are considering plans, so we know that each state can only be visited once. So, for each action, we only need to consider one effect rather than all – in particular the most negative effect:

$$n = \sum_{s \in S, a \in A(s)} \min\Big\{ \min_{s' \in \text{supp}(s,a)} \{C_{\text{pricing}}(s, a \triangleright s')\}, 0 \Big\}.$$

It is clear that the heuristic

$$h^{\text{zero}}_{-}(s) = \begin{cases} 0 & \text{if } s \in G \\ n & \text{else} \end{cases} \qquad \forall s \in S$$

is admissible.

Note that there is an additional difficulty in applying A* to problems that allow negative cycles. The A* algorithm typically does not assume a consistent heuristic, and therefore can not be sure that it has the cheapest path to any particular state. So, if A* detects an expanded state $s$ at a cheaper cost than when $s$ was expanded, we must re-add $s$ to the frontier and re-expand it to guarantee optimality. In the presence of negative cycles we can not re-expand nodes, otherwise negative cycles will trap A* in an infinite loop. This issue can be fixed by setting a cap on the number of times a state may be re-expanded, but we add a cycle detection mechanism to A*. Now, when A* finds an expanded state $s$ with cheaper cost, it traverses $s$'s parents to check

if we have detected a cycle. This becomes expensive on larger problems, but has the benefit of being able to detect and return cycles before termination of the main algorithm.

### 5.4.4 Combining A* Heuristics

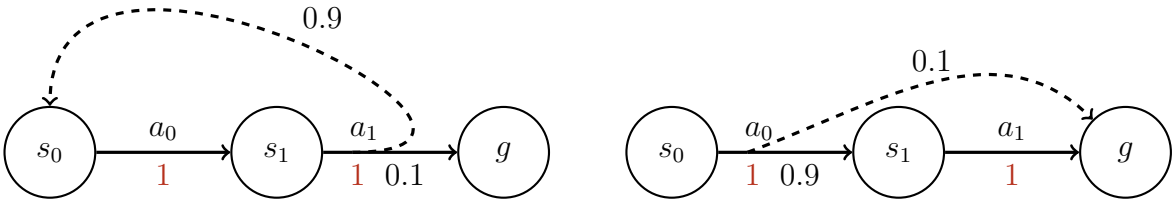
Weighted A* [Wilt and Ruml 2012] is a technique that iterates through heuristics from most informative but not admissible, to less informative but eventually admissible. This is achieved by scaling an admissible heuristic $h$ by $w \in (1, \infty)$, and running A* with this scaled non-admissible heuristic. Then, we reduce $w$ and run A* again – until we reach $w = 1$, at which stage we guarantee the optimal solution. Note that solutions for a previous heuristic are not thrown away, but rather the shortest paths are kept – since they actual cost does not change, and the frontier is updated with the new cost. This approach is able to provide a not-necessarily-optimal solution quickly due to the informative heuristic, and eventually converges to optimal as we use admissible heuristics.

This behaviour is desirable for our problem, as we want some plan of negative cost – ideally as quickly as possible, and at the end we want a guarantee that we have not missed any. So, we apply a similar technique to weighted A*: we iterate through various heuristics from most informative but non-admissible, to admissible. This allows us to compute strong heuristics like $h_{\text{add}}$ and $h_{\text{max}}$ for the determinisation of our SSP, and apply these as non-admissible heuristics to the pricing problem.

### 5.4.5 Depth Probe Optimisation

In all interesting problems, the shortest plan through the determisation will have probabilistic effects that branch off and lead to effects that the plan did not account for. There are certain patterns of "detours" from the plan caused by probabilistic effects which occur frequently. We have introduced and motivated the fix and undo actions in section 3.5, and we introduce two more cases that are important for our algorithm but not Robust-FF. These are *self-cycles* and *skips*.

Given a plan or a cycle, we call it the *pivot*. If there is a probabilistic effect in one of the pivot's actions that takes us to an effect that was previously in the pivot, we call it a self-cycle. If we are looking at a pivot plan, and a probabilistic effect takes us to a later state of the plan, we call it a skip. See figure 5.2.




(a) The probabilistic effect of $a_1$ denoted by a dashed line generates a self-cycle.

(b) The probabilistic effect of $a_0$ denoted by a dashed line is a skip.

Figure 5.2: Simple SSPs demonstrating self-cycles and skips.

Now, we introduce the concept of *outside length*, which indicates how many states that are not part of the pivot must be visited before returning to a state in the pivot. So, self-cycles and skips have outside length 0 because the detour immediately takes us back to the pivot without any extra states. Undo and fix actions have an outside length of 1, as they visit one state that was not in the pivot.

These components are very likely to be useful in the reduced master problem. So, each time we add a new plan or cycle to the reduced master problem we can treat it as a pivot, and perform a graph search to discover components of outside length $n$, extrapolate the entire plans and cycles that belong to these components, and then add these as columns also.

To give some more intuition for the notion of outside length, consider figure 5.3. Suppose the pivot plan is

$$\phi = s_0 \xrightarrow{a_0 \triangleright s_1} s_1 \xrightarrow{a_1 \triangleright g} g.$$

If we check for an outside length of 0, we will discover

- the deterministic action $a_0 \triangleright g$ which corresponds to a skip, and introduces the plan $\phi_{\text{skip}} = s_0 \xrightarrow{a_0 \triangleright g} g$
- deterministic action $a_1 \triangleright s_1$ which corresponds to the self-loop $W_{\text{self-loop}} = s_1 \xrightarrow{a_1 \triangleright s_1} s_1$
- deterministic action $a_1 \triangleright s_0$ which corresponds to the self-cycle $W_{\text{self-cycle}} = s_0 \xrightarrow{a_0 \triangleright s_1} s_1 \xrightarrow{a_1 \triangleright s_0} s_0$.

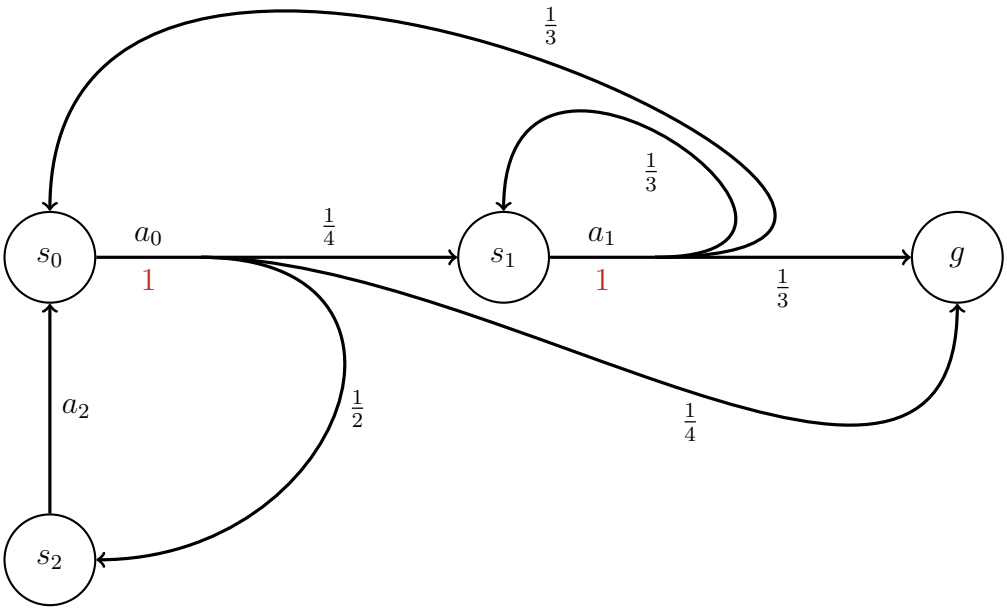



Figure 5.3: SSP that demonstrates which plans and cycles are added with various outside lengths.

If we check for outside length of 1, we will additionally discover

- the undo action $a_2 \triangleright s_0$ which corresponds to the new cycle $s_0 \xrightarrow{a_0 \triangleright s_2} s_2 \xrightarrow{a_2 \triangleright s_0} s_0$.

### 5.4.6 SSPs without Cycles

We bring to attention an important subset of SSPs which allows our algorithm to make improvements to performance. Since in general our pricing problem may have negative cycles, this unfortunately eliminates all current deterministic shortest path planners. If we only consider acyclic SSPs, we no longer have to worry about negative cycles in the determinisation, and then running A* with an admissible heuristic will necessarily find the cheapest plan – so PBColgen finds the optimal policy without requiring the addi-

tional runs of Bellman-Ford. Furthermore, this allows us to transform the solution into a regular deterministic planning problem, which in turn means we can use strong deterministic solvers – which is what makes FF-Replan and Robust-FF so effective. This is left as future work.

It is important to note that the subset of acyclic SSPs remains interesting because any problem with a monotonically strictly increasing or decreasing resource or counter is acyclic. For instance, any routing problem that models fuel consumption without refuelling actions, such as commercial flight routing [Geisser et al. 2020]. Another example is the class of finite-horizon SSPs. These are modelled with a time-step, which increases after each action is applied, and so they are acyclic.

## 5.5 Properties of PBColgen

Here we summarise the properties of PBColgen.

- We consider PBColgen a replanner because it uses deterministic planners to construct a policy, and is able to return open policies if not enough time is given to find a closed policy [Little and Thiébaux 2007].
- Assuming that the SSP has no unavoidable dead ends, and the deterministic solver returns a negative plan or cycle if it exists, then PBColgen will find the optimal solution, and upon termination of column generation we know that the solution is optimal.
- If the SSP has unavoidable dead ends and we use the generalised give-up formulation (section 5.3.4), and the deterministic planner is complete, then PBColgen will find the policy that is optimal according to the expected outcomes with finite penalty cost (definition 16).
- Since PBColgen is an iterative algorithm, it is possible to extract a policy before column generation has terminated. In this sense we can use PBColgen as an online solver. Note that the generalised give-up formulation is best-suited for this, as it can return partial policies, whereas the others must make their policy proper before giving a meaningful solution.
- Note that the linear programming framework is quite flexible, and allows us to adapt our algorithm to slightly different problems. For instance, if we are trying to find a policy which simply maximises the probability of reaching a goal, regardless of the cost, we can with easily modify the objective to minimise give-ups.

# Chapter 6

# Empirical Evaluation

## 6.1 Methodology

We investigate the online performance of PBColgen with respect to the other replanners we have introduced, namely FF-Replan (section 3.4) and Robust-FF (section 3.5).

To do this, we fixed each planner's maximum runtime and maximum memory usage, as well as an interval of $t$ seconds. Each time that $t$ seconds passed we recorded the best current policy of each planner. We achieved this by the following procedure:

1. start timer
2. run planner with a $t$ second soft deadline
3. stop timer
4. record the current policy and timestamp
5. if planner has terminated, reached max. time, or reached max. memory usage: stop – otherwise, loop back to 1.

We used a soft deadline because there are subroutines of planners which can not be interrupted, in the sense that terminating them will require us to run it again from the start, e.g. solving a deterministic problem with FF, or solving an LP with a commercial solver. Such interrupts cause two issues: first, a solver may be disadvantaged because it has to perform an expensive computation from scratch; second, a solver may get stuck in a single iteration, because its subroutine is not given enough time to terminate. Giving

planners leniency with soft deadlines means that observed timestamps need not match up with our desired "regular" time intervals. To resynchronise them as much as possible we kept track of the amount of time that a planner exceeded its soft deadline, and reduced its next time slot accordingly. To process the data we made sure that each collected data point's solution quality was only attributed to time intervals that occurred after that measurement's timestamp. Time intervals $\tau$ that had no data because of overtime were filled in with the value of the data point with the largest timestamp less than $\tau$.

We investigated three measures of policy quality:

- probability of reaching goal
- FP cost (definition 16)
- MCMP cost (definition 17).

Each of these was computed with the policy evaluation LP 8. The optimal values on problems without unavoidable dead ends were computed by running LRTDP. On problems with unavoidable dead ends, we first computed the maximal value of reaching a goal $p_{\max}$, and then solved a modified version of the dual LP (LP 6), where the sink constraint uses this value of $p_{\max}$ rather than 1, and the flow preservation constraints are relaxed to be $in(s) \geq out(s)$. The MCMP cost was then computed by summing the flow on actions scaled by their costs. The FP was then given by the following equality

$$\text{FP cost} = \text{MCMP cost} + D \cdot (1 - p_{\max})$$

where $D$ is the dead end penalty [Trevizan, F. Teichteil-Königsbuch, and Thiébaux 2017]. Note that in general this is an inequality, but on our problems we know by observation that our choice of $D = 500$ is large enough so that the expected cost given finite penalty does not truncate traces that exceed $D$ in cost, and so we have equality.

All experiments were performed on an Intel i5-8600k (3.60GHz). To make better use of resources we parallelised experiments. So, experiments were either distributed among 5 worker processes, each with 1 dedicated core and 2.6GB RAM; among 3 worker processes, each with 1 dedicated core and 5GB RAM; or among 2 worker processes, each with 1 dedicated core and 7GB RAM, depending on the size of the problem.

Note that the size of time intervals, maximum planner time, and maximum cutoff time was selected according to each problem.

## 6.2 PBColgen parameters

In our discussion of PBColgen in chapter 5, we introduced numerous alternative formulations and optimisations. Due to time restrictions we could not feasibly explore all combinations, so we explore over the following parameters.

- (formulation) In all runs we used the generalised give-up formulation presented in section 5.3.4. This is because of this formulation's benefits as an online solver, and ability to produce solutions on problems with unavoidable dead ends.
- (depth probe optimisation) Overall, we tested runs with the depth probe optimisation (as presented in section 5.4.5) set to
  - off
  - on with outside length set to 0
  - on with outside length set to 1
  - on with outside length set to 2.

  For the sake of brevity, we refer to the depth probe optimisation setting as D.P.O. throughout the results.
- (deterministic solver stack) We discussed in section 5.4.3 that PBColgen can use our Bellman-Ford variant as the sole determistic solver, or we can run A* first –with a variety of combinations of heuristics– or in problems without cycles it makes sense to run A* by itself. We will represent the list of solvers by the expression `s1 [(h1)], s2 [(h2)]...` where `s1` is the first deterministic solver we use, `s2` is the second, etc. and where applicable, i.e. when using A*, `hn` refers to the heuristic that was used by the $n$-th deterministic solver. Note that in this sequence of solvers each solver uses the partial solution of its predecessor, as described in section 5.4.4.

So, all runs may be assumed to be with the generalised give-up formulation, and the depth probe optimisation and deterministic solver settings will be provided.

Note that different experiments were performed with different combinations of parameters, as we used domain knowledge to find settings that yielded most interesting behaviours. As a reference point we used the basic version of PBColgen in each experiment – that is, PBColgen with the depth probe

optimisation disabled, and only our modified Bellman-Ford as the deterministic solver.

## 6.3 Test Problems

We performed experiments on the following domains from the international probabilistic planning competition (IPPC): tireworld, blocksworld, and exploding blocksworld [Younes et al. 2005].

### 6.3.1 Blocksworld

Blocksworld encodes the following problem: given a set of blocks on a table, the agent is tasked to put the blocks on top of each other in a particular configuration. Note that the blocks may already be partially stacked on top of each other at the start. For example, suppose we have blocks A and B, and to begin with B is on top of the table, and A is on top of B. Our agent may be tasked to construct the following configuration: B is on top of A. The agent must then pick up A, place it on the table, pick up B, and place it on top of A. The probabilistic variant we are considering introduces a probability 0.25 of the block "slipping" out of the agent's grasp and falling on the table. In addition, our variant allows the agent to pick up a tower of two blocks at once, and place this tower on the table or on top of another block. However, picking up a tower fails with 0.9 probability in which case nothing happens, and when placing a tower on top of another block there is a 0.9 probability of it "slipping" and falling to the table as a tower, i.e. the two blocks stay in the same configuration to each other, but are now on the table.

The difficulty of the problem depends on the number of blocks, and how many actions we need to perform to get from the initial to the goal configuration.

Note that blocksworld has no dead ends.

### 6.3.2 Tireworld

Tireworld's scenario is this: the agent must drive a car from a starting city to a destination city. There are numerous cities on the way, and the agent's available actions are to drive to one of the neighbouring cities. The catch is that each time the agent drives, there is a 0.5 probability of getting a flat tyre. In some –not all– cities the agent may load a spare tyre. If the agent

gets a flat tyre and has a spare loaded, it may replace its flat, otherwise it is stuck.

The difficulty of a tireworld problem increases exponentially as we increase the size of the problem.

Note that tireworld has dead ends, as the agent may get a flat tyre without having a spare. These dead ends are avoidable (by loading a spare).

Note also that tireworld has no cycles. This is because roads only go one way towards the goal such that the agent can not backtrack, and a tyre can only be loaded in a city once.

This domain was designed to be a hard problem for determinisation based planners, in particular replanners. The shortest path from the starting city to the goal city contains no cities in which the agent can pick up a spare tyre, and the cities with tyres available are laid out in such a way that the agent needs to take a detour. This arrangement is intended to trick planners that use determinisation into the cheapest policy which is least likely to succeed.

### 6.3.3 Exploding blocksworld

Exploding blocksworld is a modification of blocksworld, where the possibility of "slipping" is removed, and instead blocks can irreversibly explode. That is, each block has a "detonator" attached, which has a particular probability of exploding each time that block is placed down. Such an explosion destroys the block or table beneath it, and once a detonator has exploded it can no longer explode.

As with blocksworld, the difficulty of the problem depends on the number of blocks, and how many actions are needed to move from the starting arrangement to the goal arrangement.

The original formulation of our exploding blockworld domain had the probability of detonation set to $\frac{2}{5}$ and $\frac{1}{10}$ depending on whether the agent places the block on a table or another block. Due to issues with numerical instability, this made the problem infeasible for our LPs. So, we had to modify the exploding blocks-world domain to use fractional values that are easier to represent with floating point arithmetic. For this purpose we changed all probabilities to 0.5.

Note that exploding blocksworld can have unavoidable dead ends – we always risk exploding a block that is necessary in the goal configuration. So, some

instances only have improper policies. This makes it more difficult to compare the quality of solutions, which is why we present MCMP cost, expected cost given finite penalty, and the probability of reaching a goal.

## 6.4 Results

Here we present the results of our experiments organised according to the problem domain. In addition, we provide the domain specific planner settings, and some initial observations.

### 6.4.1 Presentation

Note that the data presented in tables is complete for that particular experiment, whereas for clarity, plots typically only show results from a subset of planners.

In each table, we present information about the planner's ability to construct an initial policy, to converge to the optimal, and to terminate on the given problem. Concretely, we tested each planner over 10 runs, and present the following information about the planner's construction of an initital policy:

- (count) the number of runs for which the planner was able to construct an initial policy without exceeding its time and memory restrictions
- (avg. time) the mean average time to construct an initial policy over the runs that were able to do so, with the 95% confidence interval
- (max. time) the maximum time to construct an initial policy over the runs that were able to do so.

Information about convergence and termination is presented in the same fashion, where a run terminates if the associated algorithm reaches its termination criterion, and a run converges if it outputs a policy with optimal probability and MCMP cost until termination or the end of the experiment. Note that PBColgen terminates when it has proved that its solution is optimal, whereas Robust-FF terminates when 5000 simulations of the candidate policy do not reach any undefined states. FF-Replan has no termination criterion.

All plots were generated using the mean values of 10 runs. We use crosses to denote the maximum termination time of each planner over all runs.

### 6.4.2 Blocksworld

We performed experiments on two blocksworld problems, `bw_5` and `bw_6` with 5 and 6 blocks respectively. The associated information of each tested problem is presented in table 6.1.

| | max. time | time intervals | max. RAM | optimal MCMP cost | optimal prob. of reaching goal | optimal FP cost |
|---|---|---|---|---|---|---|
| `bw_5` | 30 secs | 0.1 secs | 2.6 GB | 15.9444 | 1.0 | 15.9444 |
| `bw_6` | 15 mins | 1 sec | 2.6 GB | 14.5833 | 1.0 | 14.5833 |

Table 6.1: Settings for blocksworld experiments.

For the results of `bw_5` see figure 6.1 and table 6.2; and for `bw_6` see figure 6.2 and table 6.3.

Blocksworld has the property that plans based on relaxations tend to perform quite well, since the consequence of a block slipping or failing to move a tower is minor. Indeed, FF-Replan and Robust-FF are able to provide near optimal solutions almost instantaneously. PBColgen tends to be slower to generate its first policy – while FF-Replan and Robust-FF already have a policy and are working on closing it. However, we see that PBColgen does not take substantially longer to catch up to the other replanners in terms of quality. More importantly, PBColgen is the only planner able to find the optimal solution. This shows the shortcomings of previous replanners even in problems without dead ends.

### 6.4.3 Tireworld

We performed experiments on three tireworld problems, namely, `tire_2`, `tire_3`, and `tire_5` where the number of cities is 25, 48, and 120 respectively. The settings for each experiment are provided in table 6.4.

| | max. time | time intervals | max. RAM | optimal MCMP cost | optimal prob. of reaching goal | optimal FP cost |
|---|---|---|---|---|---|---|
| `tire_2` | 60 secs | 0.25 secs | 2.6 GB | 11.8594 | 1.0 | 11.8594 |
| `tire_3` | 30 mins | 10 secs | 5 GB | 19.2178 | 1.0 | 19.2178 |
| `tire_5` | 10 mins | 0.5 secs | 7 GB | 35.0137 | 1.0 | 35.0137 |

Table 6.4: Settings for tireworld experiments.

As this problem is acyclic, we can guarantee an optimal solution by running

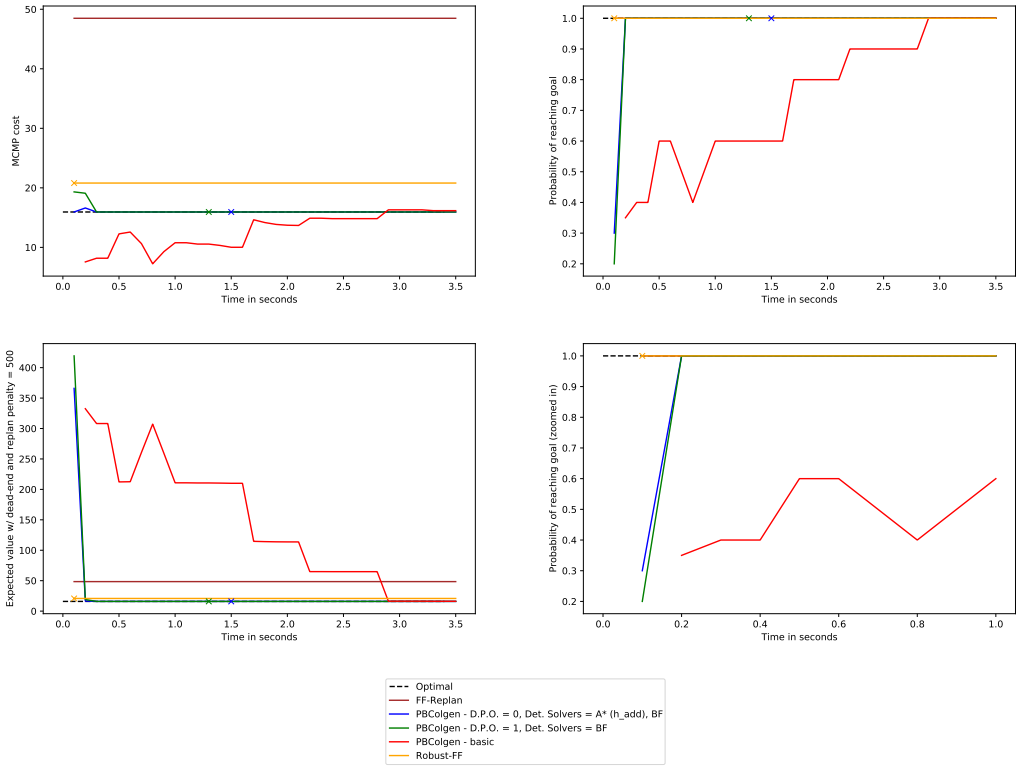



Figure 6.1: Solution quality as a function of time on problem `bw_5` (averaged over 10 runs). Measurements were recorded each $\frac{1}{10}$ seconds, so each half second interval on the plot contains 5 measurements. Crosses indicate the max. termination time across runs – in the case of PBColgen this is a guarantee of optimality. This plot considers only the first 3.5 seconds of the 30 second experiment, as all depicted planners plateau after that time. Note that Robust-FF obscures FF-Replan.

| Planner | | | Initial Policy | | | Convergence | | | Termination | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | count | avg. time (secs) | max. time (secs) | count | avg. time (secs) | max. time (secs) | count | avg. time (secs) | max. time (secs) |
| | depth probe opt. | det. solvers | | | | | | | | | |
| PBColgen | 0 | A* ($h_{\text{add}}$), A* ($h_{\text{max}}$), A* ($h^{\text{zero}}$), BF | 10/10 | 0.18 ± 0.02 | 0.2 | 10/10 | 0.21 ± 0.04 | 0.3 | 10/10 | 2.13 ± 0.31 | 2.9 |
| | 0 | A* ($h_{\text{add}}$), BF | 10/10 | 0.17 ± 0.03 | 0.2 | 10/10 | 0.20 ± 0.05 | 0.3 | 10/10 | 1.15 ± 0.18 | 1.6 |
| | 0 | BF | 10/10 | 0.20 ± 0.00 | 0.2 | 10/10 | 0.97 ± 0.39 | 1.9 | 10/10 | 2.22 ± 0.98 | 4.6 |
| | 1 | A* ($h_{\text{add}}$), A* ($h_{\text{max}}$), A* ($h^{\text{zero}}$), BF | 10/10 | 0.19 ± 0.02 | 0.2 | 10/10 | 0.19 ± 0.02 | 0.2 | 10/10 | 2.20 ± 0.28 | 3.0 |
| | 1 | A* ($h_{\text{add}}$), BF | 10/10 | 0.19 ± 0.02 | 0.2 | 10/10 | 0.19 ± 0.02 | 0.2 | 10/10 | 1.30 ± 0.14 | 1.6 |
| | 1 | BF | 10/10 | 0.18 ± 0.02 | 0.2 | 10/10 | 0.30 ± 0.00 | 0.3 | 10/10 | 1.13 ± 0.10 | 1.4 |
| | 2 | A* ($h_{\text{add}}$), A* ($h_{\text{max}}$), A* ($h^{\text{zero}}$), BF | 10/10 | 0.20 ± 0.00 | 0.2 | 10/10 | 0.20 ± 0.00 | 0.2 | 10/10 | 1.93 ± 0.19 | 2.3 |
| | 2 | A* ($h_{\text{add}}$), BF | 10/10 | 0.17 ± 0.03 | 0.2 | 10/10 | 0.17 ± 0.03 | 0.2 | 10/10 | 1.20 ± 0.11 | 1.4 |
| | 2 | BF | 10/10 | 0.19 ± 0.02 | 0.2 | 10/10 | 0.30 ± 0.00 | 0.3 | 10/10 | 1.09 ± 0.06 | 1.3 |
| | off | A* ($h_{\text{add}}$), A* ($h_{\text{max}}$), A* ($h^{\text{zero}}$), BF | 10/10 | 0.13 ± 0.03 | 0.2 | 10/10 | 0.29 ± 0.02 | 0.3 | 10/10 | 2.78 ± 0.47 | 3.9 |
| | off | A* ($h_{\text{add}}$), BF | 10/10 | 0.13 ± 0.03 | 0.2 | 10/10 | 0.26 ± 0.03 | 0.3 | 10/10 | 1.69 ± 0.26 | 2.4 |
| | off | BF | 10/10 | 0.20 ± 0.00 | 0.2 | 10/10 | 1.88 ± 0.57 | 3.8 | 10/10 | 3.25 ± 1.22 | 7.9 |
| FF-Replan | | | 10/10 | 0.10 ± 0.00 | 0.1 | 0/10 | - | - | 0/10 | - | - |
| Robust-FF | | | 10/10 | 0.10 ± 0.00 | 0.1 | 0/10 | - | - | 10/10 | 0.10 ± 0.00 | 0.1 |

Table 6.2: Summary of experimental results for `bw_5`. All presented averages are taken over the runs that were successful –as indicated by count– and are shown with the 95% confidence interval. For more information about parameter settings for PBColgen see section 6.2, and for more information about the way data is presented see section 6.4.1.

| Planner | | | Initial Policy | | | Convergence | | | Termination | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | count | avg. time (secs) | max. time (secs) | count | avg. time (secs) | max. time (secs) | count | avg. time (secs) | max. time (secs) |
| | depth probe opt. | det. solvers | | | | | | | | | |
| PBColgen | 0 | A* ($h_{\text{add}}$), A* ($h_{\text{max}}$), A* ($h^{\text{zero}}$), BF | 10/10 | 1.20 ± 0.25 | 2.0 | 10/10 | 69.30 ± 40.94 | 183.0 | 9/10 | 206.67 ± 110.58 | 633.0 |
| | 0 | A* ($h_{\text{add}}$), BF | 10/10 | 1.20 ± 0.25 | 2.0 | 10/10 | 4.50 ± 1.08 | 8.0 | 10/10 | 65.40 ± 12.76 | 97.0 |
| | 0 | BF | 10/10 | 1.90 ± 0.19 | 2.0 | 10/10 | 17.00 ± 9.51 | 52.0 | 10/10 | 99.10 ± 28.35 | 204.0 |
| | 1 | A* ($h_{\text{add}}$), A* ($h_{\text{max}}$), A* ($h^{\text{zero}}$), BF | 10/10 | 1.70 ± 0.28 | 2.0 | 9/10 | 5.44 ± 4.56 | 22.0 | 8/10 | 91.38 ± 13.73 | 125.0 |
| | 1 | A* ($h_{\text{add}}$), BF | 10/10 | 1.60 ± 0.30 | 2.0 | 10/10 | 4.40 ± 3.05 | 18.0 | 10/10 | 64.50 ± 17.52 | 130.0 |
| | 1 | BF | 10/10 | 2.00 ± 0.00 | 2.0 | 10/10 | 4.80 ± 2.14 | 13.0 | 10/10 | 46.10 ± 6.03 | 64.0 |
| | 2 | A* ($h_{\text{add}}$), A* ($h_{\text{max}}$), A* ($h^{\text{zero}}$), BF | 10/10 | 2.00 ± 0.00 | 2.0 | 10/10 | 16.60 ± 9.47 | 44.0 | 10/10 | 78.40 ± 11.36 | 105.0 |
| | 2 | A* ($h_{\text{add}}$), BF | 10/10 | 1.20 ± 0.25 | 2.0 | 10/10 | 2.50 ± 0.57 | 4.0 | 10/10 | 40.40 ± 6.29 | 58.0 |
| | 2 | BF | 10/10 | 1.90 ± 0.19 | 2.0 | 10/10 | 6.20 ± 3.69 | 21.0 | 10/10 | 53.00 ± 10.55 | 86.0 |
| | off | A* ($h_{\text{add}}$), A* ($h_{\text{max}}$), A* ($h^{\text{zero}}$), BF | 10/10 | 1.30 ± 0.28 | 2.0 | 7/10 | 157.43 ± 119.14 | 525.0 | 6/10 | 202.17 ± 53.31 | 334.0 |
| | off | A* ($h_{\text{add}}$), BF | 10/10 | 1.20 ± 0.25 | 2.0 | 10/10 | 7.60 ± 3.35 | 19.0 | 10/10 | 71.90 ± 10.16 | 101.0 |
| | off | BF | 10/10 | 2.00 ± 0.00 | 2.0 | 10/10 | 117.20 ± 67.78 | 369.0 | 5/10 | 335.40 ± 89.02 | 532.0 |
| FF-Replan | | | 10/10 | 1.00 ± 0.00 | 1.0 | 0/10 | - | - | 0/10 | - | - |
| Robust-FF | | | 10/10 | 1.00 ± 0.00 | 1.0 | 0/10 | - | - | 10/10 | 1.00 ± 0.00 | 1.0 |

Table 6.3: Summary of experimental results for `bw_6`. All presented averages are taken over the runs that were successful –as indicated by count– and are shown with the 95% confidence interval. For more information about parameter settings for PBColgen see section 6.2, and for more information about the way data is presented see section 6.4.1.

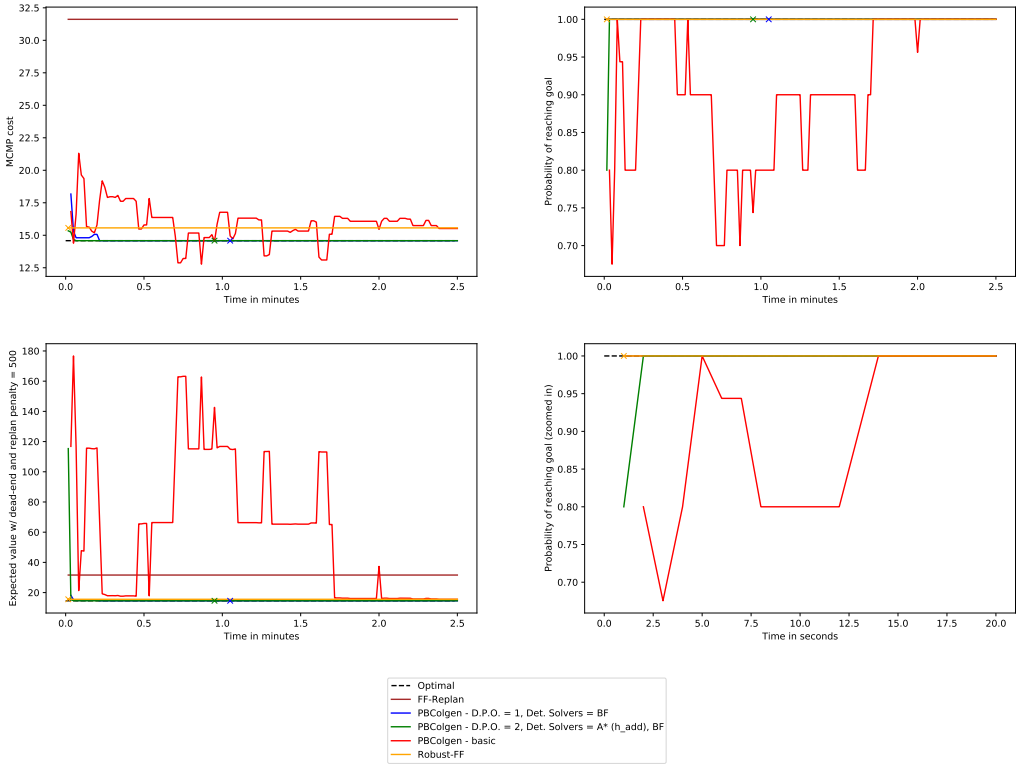



Figure 6.2: Solution quality as a function of time on problem `bw_6` (averaged over 10 runs). Measurements were recorded each second, so each half minute interval on the plot contains 30 measurements. Crosses indicate the max. termination time across runs – in the case of PBColgen this is a guarantee of optimality. This plot considers only the first 2.5 minutes 15 minute experiment, as all depicted planners plateau after that time except PBColgen-basic, which does not reliably converge within the experiment limits (see table 6.3). Note that Robust-FF obscures FF-Replan.

| Planner | | | Initial Policy | | | Convergence | | | Termination | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | count | avg. time (secs) | max. time (secs) | count | avg. time (secs) | max. time (secs) | count | avg. time (secs) | max. time (secs) |
| PBColgen | depth probe opt. | det. solvers | | | | | | | | | |
| | 0 | A* ($h_{\text{add}}$), BF | 10/10 | 0.25 ± 0.00 | 0.25 | 10/10 | 10.35 ± 2.44 | 15.50 | 10/10 | 13.07 ± 1.40 | 16.25 |
| | 0 | A* ($h_{\text{add}}$), A* ($h_{\_}^{\text{zero}}$) | 10/10 | 0.25 ± 0.00 | 0.25 | 10/10 | 12.12 ± 3.58 | 23.50 | 10/10 | 16.35 ± 1.79 | 23.50 |
| | 0 | A* ($h_{\_}^{\text{zero}}$) | 10/10 | 0.25 ± 0.00 | 0.25 | 10/10 | 27.48 ± 4.28 | 39.25 | 10/10 | 28.85 ± 4.68 | 45.00 |
| | 0 | BF | 10/10 | 0.25 ± 0.00 | 0.25 | 10/10 | 14.97 ± 5.43 | 37.25 | 10/10 | 19.00 ± 9.32 | 55.00 |
| | off | BF | 10/10 | 0.25 ± 0.00 | 0.25 | 7/10 | 7.64 ± 0.87 | 9.75 | 7/10 | 8.07 ± 0.84 | 10.25 |
| FF-Replan | | | 10/10 | 0.25 ± 0.00 | 0.25 | 0/10 | - | - | 0/10 | - | - |
| Robust-FF | | | 10/10 | 0.25 ± 0.00 | 0.25 | 0/10 | - | - | 10/10 | 0.25 ± 0.00 | 0.25 |

Table 6.5: Summary of experimental results for `tire_2`. All presented averages are taken over the runs that were successful –as indicated by count– and are shown with the 95% confidence interval. For more information about parameter settings for PBColgen see section 6.2, and for more information about the way data is presented see section 6.4.1.

| Planner | | | Initial Policy | | | Convergence | | | Termination | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | count | avg. time (secs) | max. time (secs) | count | avg. time (secs) | max. time (secs) | count | avg. time (secs) | max. time (secs) |
| PBColgen | depth probe opt. | det. solvers | | | | | | | | | |
| | 0 | A* ($h_{\text{add}}$), BF | 10/10 | 10.00 ± 0.00 | 10.0 | 0/10 | - | - | 0/10 | - | - |
| | 0 | A* ($h_{\text{add}}$), A* ($h_{\_}^{\text{zero}}$) | 10/10 | 10.00 ± 0.00 | 10.0 | 0/10 | - | - | 0/10 | - | - |
| | 0 | A* ($h_{\_}^{\text{zero}}$) | 3/10 | 23.33 ± 5.33 | 30.0 | 0/10 | - | - | 0/10 | - | - |
| | 0 | BF | 10/10 | 10.00 ± 0.00 | 10.0 | 0/10 | - | - | 0/10 | - | - |
| | off | BF | 10/10 | 12.00 ± 2.48 | 20.0 | 0/10 | - | - | 0/10 | - | - |
| FF-Replan | | | 10/10 | 10.00 ± 0.00 | 10.0 | 0/10 | - | - | 0/10 | - | - |
| Robust-FF | | | 10/10 | 10.00 ± 0.00 | 10.0 | 0/10 | - | - | 10/10 | 10.00 ± 0.00 | 10.0 |

Table 6.6: Summary of experimental results for `tire_3`. All presented averages are taken over the runs that were successful –as indicated by count– and are shown with the 95% confidence interval. For more information about parameter settings for PBColgen see section 6.2, and for more information about the way data is presented see section 6.4.1.

A* with an admissible heuristic as the sole deterministic solver, which we have included among our experiments.

Note that `tire_5` was given a shorter period of time with smaller time intervals than `tire_3` to investigate how long it takes for each planner to construct an initial solution.

This problem was designed to be hard for greedy replanning algorithms [Little and Thiébaux 2007]. This can be seen in the results, as FF-Replan and Robust-FF reach the goal with very small probability. The experiment also highlights the advantages of PBColgen, namely that our search over the plan and cycle space is complete, and we can improve our solution over time, making sure that dead ends are avoided when possible.

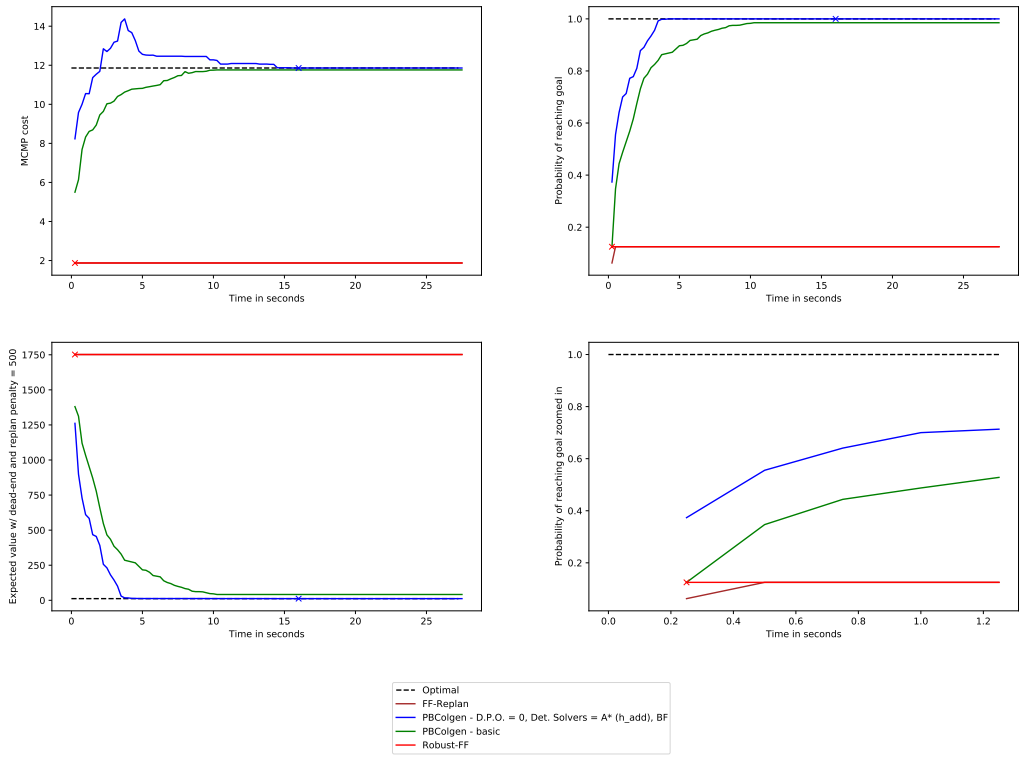



Figure 6.3: Solution quality as a function of time on problem `tire_2` (averaged over 10 runs). Measurements were recorded every $\frac{1}{4}$ seconds, so each 5 second interval on the plot contains 20 measurements. Crosses indicate the max. termination time across runs – in the case of PBColgen this is a guarantee of optimality. This plot considers only the first 30 seconds of the 60 second experiment, as all depicted planners plateau after that time. Note that Robust-FF obscures FF-Replan.

| Planner | | | Initial Policy | | | Convergence | | | Termination | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | count | avg. time (secs) | max. time (secs) | count | avg. time (secs) | max. time (secs) | count | avg. time (secs) | max. time (secs) |
| PBColgen | depth probe opt. | det. solvers | | | | | | | | | |
| | 0 | A* ($h_{\text{add}}$), BF | 10/10 | 5.55 ± 0.22 | 6.0 | 0/10 | - | - | 0/10 | - | - |
| | 0 | A* ($h_{\text{add}}$), A* ($h^{\text{zero}}_{-}$) | 10/10 | 0.55 ± 0.09 | 1.0 | 0/10 | - | - | 0/10 | - | - |
| | 0 | BF | 10/10 | 4.55 ± 0.09 | 5.0 | 0/10 | - | - | 0/10 | - | - |
| | off | BF | 10/10 | 5.50 ± 0.00 | 5.5 | 0/10 | - | - | 0/10 | - | - |
| FF-Replan | | | 10/10 | 0.50 ± 0.00 | 0.5 | 0/10 | - | - | 0/10 | - | - |
| Robust-FF | | | 10/10 | 0.50 ± 0.00 | 0.5 | 0/10 | - | - | 10/10 | 0.50 ± 0.00 | 0.5 |

Table 6.7: Summary of experimental results for `tire_5`. All presented averages are taken over the runs that were successful –as indicated by count– and are shown with the 95% confidence interval. For more information about parameter settings for PBColgen see section 6.2, and for more information about the way data is presented see section 6.4.1.

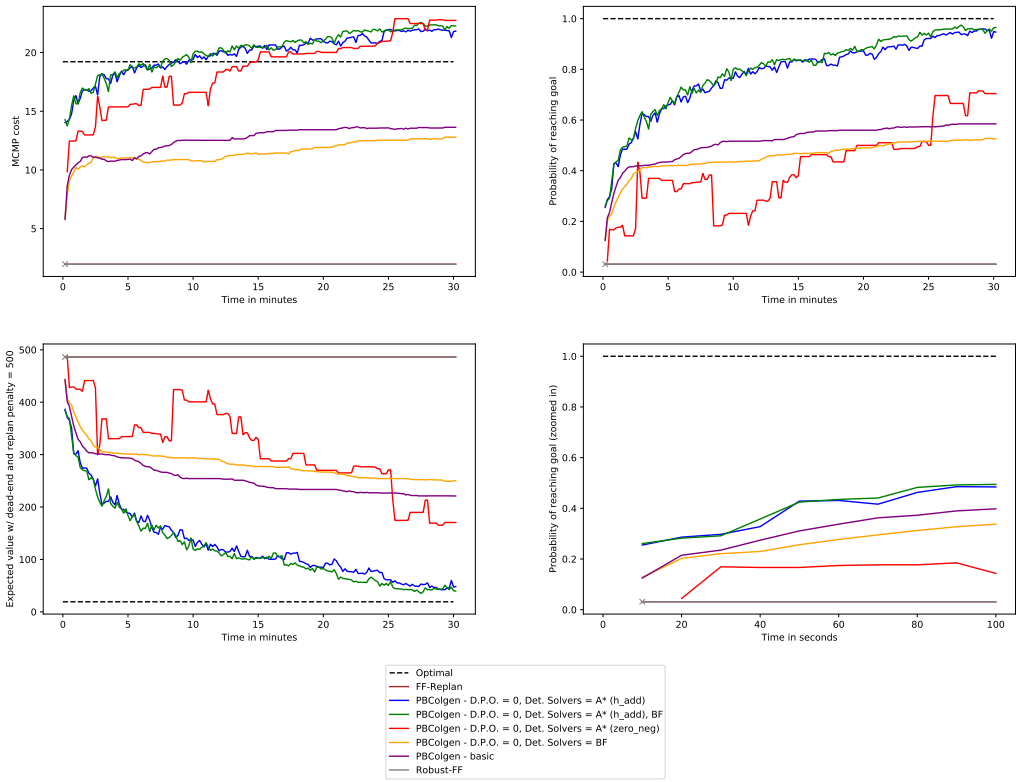



Figure 6.4: Solution quality as a function of time on problem `tire_3` (averaged over 10 runs). Measurements were recorded every 10 seconds, so every 5 minute interval on the plot contains 30 measurements. Crosses indicate the max. termination time across runs – in the case of PBColgen this is a guarantee of optimality. Note that Robust-FF obscures FF-Replan.

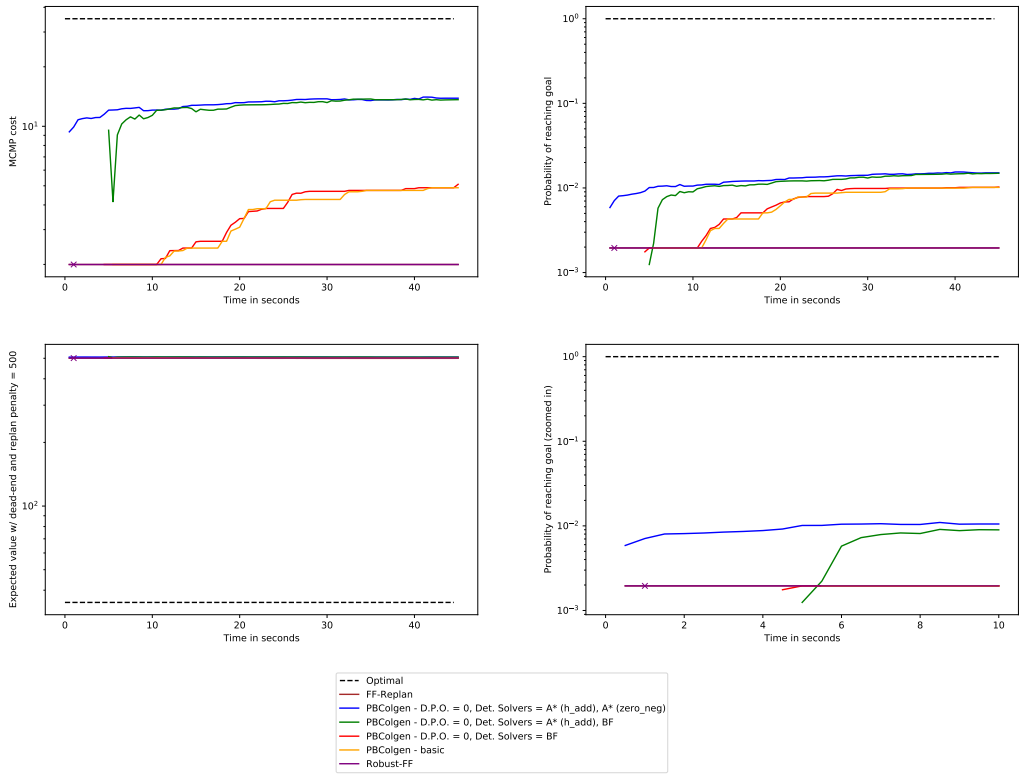



Figure 6.5: Solution quality as a function of time on problem `tire_5` (averaged over 10 runs). Measurements were recorded every $\frac{1}{2}$ seconds, so every 10 second interval on the plot contains 20 measurements. This plot presents the $y$-axis on a logarithmic scale as no planners are able to come close to the optimal probability of reaching goal. This plot only considers the first 50 seconds of the 10 minute experiment, as it focuses on the time it takes for a planner to produce its first policy. Crosses indicate the max. termination time across runs – for Robust-FF this only means that its simulations indicate a closed policy. Note that Robust-FF obscures FF-Replan.

| Planner | | | Initial Policy | | | Convergence | | | Termination | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | count | avg. time (secs) | max. time (secs) | count | avg. time (secs) | max. time (secs) | count | avg. time (secs) | max. time (secs) |
| | depth probe opt. | det. solvers | | | | | | | | | |
| PBColgen | 0 | A* ($h_{\text{add}}$), A* ($h_{\text{max}}$), A* ($h^{\text{zero}}$), BF | 10/10 | 0.20 ± 0.00 | 0.2 | 10/10 | 2.62 ± 0.27 | 3.1 | 5/10 | 12.28 ± 3.87 | 19.6 |
| | 0 | A* ($h_{\text{add}}$), BF | 10/10 | 0.21 ± 0.02 | 0.3 | 10/10 | 2.72 ± 0.26 | 3.2 | 5/10 | 14.72 ± 4.97 | 19.7 |
| | 0 | BF | 10/10 | 0.20 ± 0.00 | 0.2 | 10/10 | 1.93 ± 0.48 | 3.5 | 10/10 | 5.71 ± 1.26 | 8.5 |
| | 1 | A* ($h_{\text{add}}$), A* ($h_{\text{max}}$), A* ($h^{\text{zero}}$), BF | 10/10 | 0.20 ± 0.00 | 0.2 | 10/10 | 1.95 ± 0.18 | 2.2 | 10/10 | 12.99 ± 4.69 | 29.3 |
| | 1 | A* ($h_{\text{add}}$), BF | 10/10 | 0.21 ± 0.02 | 0.3 | 10/10 | 2.06 ± 0.16 | 2.3 | 5/10 | 7.14 ± 4.50 | 17.3 |
| | 1 | BF | 10/10 | 0.20 ± 0.00 | 0.2 | 10/10 | 1.32 ± 0.35 | 2.0 | 10/10 | 4.79 ± 0.83 | 6.5 |
| | 2 | A* ($h_{\text{add}}$), A* ($h_{\text{max}}$), A* ($h^{\text{zero}}$), BF | 10/10 | 0.20 ± 0.00 | 0.2 | 10/10 | 1.98 ± 0.20 | 2.2 | 10/10 | 13.32 ± 4.75 | 29.9 |
| | 2 | A* ($h_{\text{add}}$), BF | 10/10 | 0.20 ± 0.00 | 0.2 | 10/10 | 2.06 ± 0.18 | 2.3 | 5/10 | 7.12 ± 4.55 | 17.4 |
| | 2 | BF | 10/10 | 0.20 ± 0.00 | 0.2 | 10/10 | 1.26 ± 0.34 | 1.8 | 10/10 | 4.70 ± 0.81 | 6.4 |
| | off | A* ($h_{\text{add}}$), A* ($h_{\text{max}}$), A* ($h^{\text{zero}}$), BF | 10/10 | 0.20 ± 0.00 | 0.2 | 10/10 | 2.58 ± 0.29 | 3.0 | 8/10 | 8.72 ± 1.05 | 10.0 |
| | off | A* ($h_{\text{add}}$), BF | 10/10 | 0.20 ± 0.00 | 0.2 | 10/10 | 2.65 ± 0.27 | 3.1 | 4/10 | 12.32 ± 6.60 | 19.2 |
| | off | BF | 10/10 | 0.20 ± 0.00 | 0.2 | 10/10 | 1.56 ± 0.32 | 2.2 | 10/10 | 5.13 ± 0.87 | 6.9 |
| FF-Replan | | | 10/10 | 0.13 ± 0.03 | 0.2 | 0/10 | - | - | 0/10 | - | - |
| Robust-FF | | | 10/10 | 0.10 ± 0.00 | 0.1 | 0/10 | - | - | 10/10 | 0.20 ± 0.00 | 0.2 |

Table 6.9: Summary of experimental results for `exbw_p02-n3-N5-s2`. All presented averages are taken over the runs that were successful –as indicated by count– and are shown with the 95% confidence interval. For more information about parameter settings for PBColgen see section 6.2, and for more information about the way data is presented see section 6.4.1.

### 6.4.4 Exploding Blocksworld

For the exploding blocksworld domain, we performed experiments on problems `exbw_p02-n3-N5-s2` with 5 blocks and `exbw_p03-n3-N6-s3` with 6 blocks. Settings for each experiment are provided in table 6.8.

| | max. time | time intervals | max. RAM | optimal MCMP cost | optimal prob. of reaching goal | optimal FP cost |
|---|---|---|---|---|---|---|
| `exbw_p02-n3-N5-s2` | 30 secs | 0.1 secs | 2.6 GB | 5 | 0.25 | 380 |
| `exbw_p03-n3-N6-s3` | 30 mins | 5 secs | 5 GB | 8 | 0.5 | 258 |

Table 6.8: Settings for exploding blocksworld experiments.

The exploding blocks world domain tests a planner's ability to "think ahead," and avoid actions that are more likely to trap the agent in a dead end. Classical replanners have no mechanism to consider the unintended side effects of moving blocks, i.e. exploding another block or table and making them unusable. Thus, they are unable to reason about the probability of reaching a dead end. We see that FF-Replan and Robust-FF are tricked by the problem to minimise the MCMP cost, but perform poorly on the other criteria. PBColgen does not allow itself to be mislead in the same manner, and prioritises avoiding dead ends.

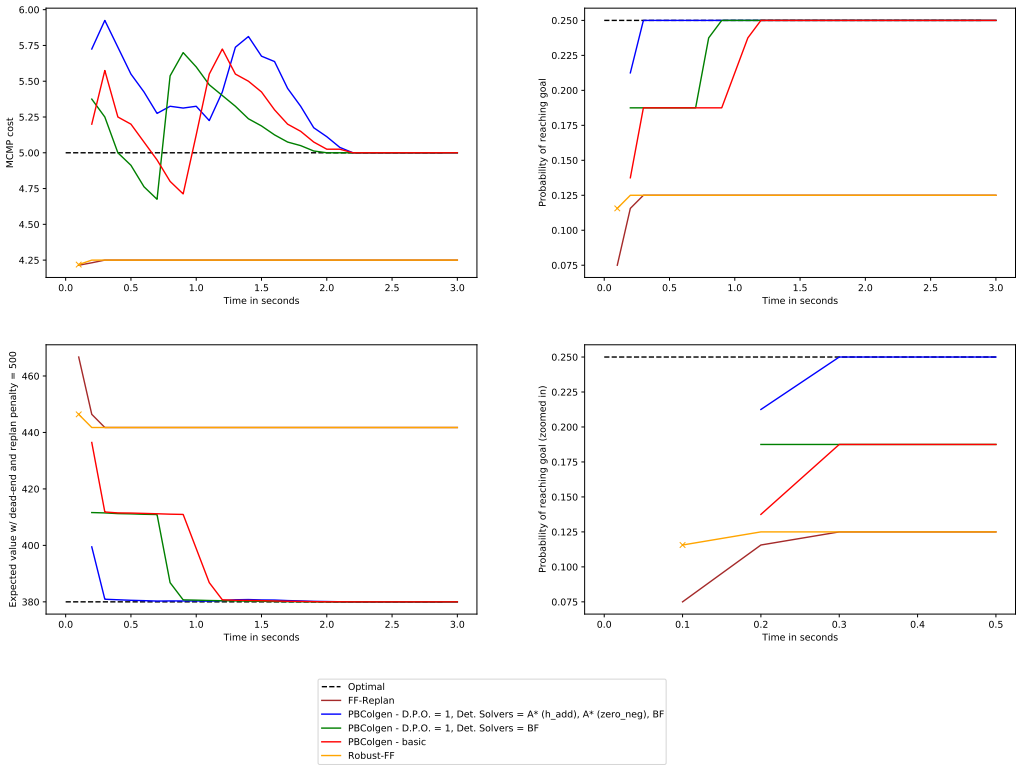



Figure 6.6: Solution quality as a function of time on problem `exbw_p02-n3-N5-s2` (averaged over 10 runs). Measurements were recorded every $\frac{1}{10}$ seconds, so each half-second interval on the plot contains 5 measurements. Crosses indicate the max. termination time across runs – in the case of PBColgen this is a guarantee of optimality, in the case of Robust-FF this occurs when 5000 simulations indicate a closed policy. On this problem we observe that Robust-FF "terminates" before it has a closed policy, and so the solution improves after "termination" due to Robust-FF's online functionality. This plot considers only the first 3 seconds of the 30 second experiment, as all depicted planners plateau after that time.

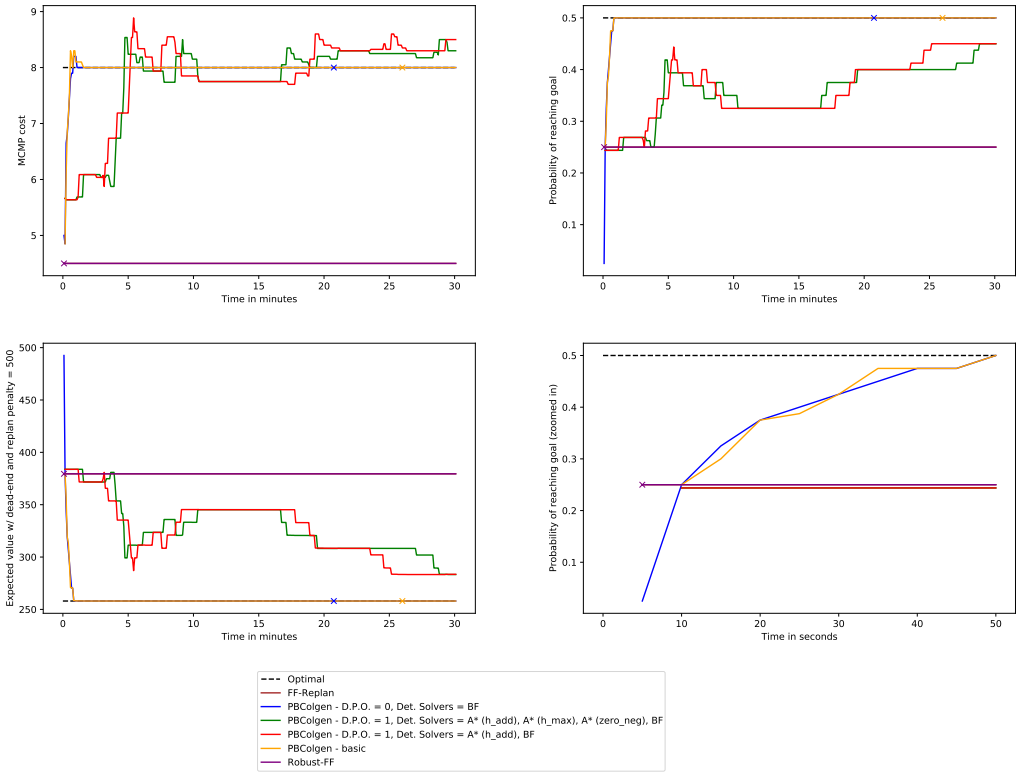



Figure 6.7: Solution quality as a function of time on problem `exbw_p03-n3-N6-s3` (averaged over 10 runs). Measurements were recorded every 5 seconds, so each 5 minute interval on the plot contains 60 measurements. Crosses indicate the max. termination time across runs – for Robust-FF this only means that its simulations indicate a closed policy. Note that Robust-FF obscures FF-Replan.

| Planner | | | Initial Policy | | | Convergence | | | Termination | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | count | avg. time (secs) | max. time (secs) | count | avg. time (secs) | max. time (secs) | count | avg. time (secs) | max. time (secs) |
| | depth probe opt. | det. solvers | | | | | | | | | |
| PBColgen | 0 | A* ($h_{\text{add}}$), A* ($h_{\text{max}}$), A* ($h^{\text{zero}}_{-}$), BF | 10/10 | 10.00 ± 0.00 | 10.0 | 2/10 | 1535.00 ± 62.37 | 1580.0 | 0/10 | - | - |
| | 0 | A* ($h_{\text{add}}$), BF | 10/10 | 10.00 ± 0.00 | 10.0 | 2/10 | 1502.50 ± 190.57 | 1640.0 | 0/10 | - | - |
| | 0 | BF | 10/10 | 9.50 ± 0.93 | 10.0 | 10/10 | 40.50 ± 9.24 | 65.0 | 7/10 | 1079.29 ± 101.35 | 1250.0 |
| | 1 | A* ($h_{\text{add}}$), A* ($h_{\text{max}}$), A* ($h^{\text{zero}}_{-}$), BF | 10/10 | 10.00 ± 0.00 | 10.0 | 2/10 | 1450.00 ± 436.57 | 1765.0 | 0/10 | - | - |
| | 1 | A* ($h_{\text{add}}$), BF | 10/10 | 10.00 ± 0.00 | 10.0 | 1/10 | 1275.00 ± 0.00 | 1275.0 | 0/10 | - | - |
| | 1 | BF | 10/10 | 10.00 ± 0.00 | 10.0 | 10/10 | 28.50 ± 5.72 | 45.0 | 8/10 | 1084.38 ± 114.33 | 1400.0 |
| | off | A* ($h_{\text{add}}$), A* ($h_{\text{max}}$), A* ($h^{\text{zero}}_{-}$), BF | 10/10 | 9.50 ± 0.93 | 10.0 | 1/10 | 1590.00 ± 0.00 | 1590.0 | 0/10 | - | - |
| | off | A* ($h_{\text{add}}$), BF | 10/10 | 10.00 ± 0.00 | 10.0 | 1/10 | 1510.00 ± 0.00 | 1510.0 | 0/10 | - | - |
| | off | BF | 10/10 | 10.00 ± 0.00 | 10.0 | 10/10 | 47.00 ± 12.72 | 95.0 | 5/10 | 1122.00 ± 235.50 | 1565.0 |
| FF-Replan | | | 10/10 | 5.00 ± 0.00 | 5.0 | 0/10 | - | - | 0/10 | - | - |
| Robust-FF | | | 10/10 | 5.00 ± 0.00 | 5.0 | 0/10 | - | - | 10/10 | 5.00 ± 0.00 | 5.0 |

Table 6.10: Summary of experimental results for `exbw_p03-n3-N6-s3`. All presented averages are taken over the runs that were successful –as indicated by count– and are shown with the 95% confidence interval. For more information about parameter settings for PBColgen see section 6.2, and for more information about the way data is presented see section 6.4.1.

## 6.5 Discussion

In essence there are two questions we are interested in to determine the effectiveness of PBColgen as a replanner:

1. How long does it take for PBColgen to return its first policy?
2. How quickly does PBColgen's initial solution improve?

To answer how long it takes for PBColgen to return its first policy, it is most useful to consider the plots that depict the zoomed in probability of reaching goal. It is clear that the answer to this question depends on the size of the problem and our parameters. In particular, we know that versions of PBColgen with Bellman-Ford as a deterministic solver need to compute the reachable space before starting, which has heavy performance impact on large problems. The data for `tire_05` in figure 6.5 and table 6.7 shows this clearly: FF-Replan and Robust-FF give a solution within the first recorded time interval of 0.5 seconds, and our versions of PBColgen that use Bellman-Ford take up to more than 5 seconds. However, our variant that did not run Bellman-Ford and therefore did not have to compute the reachable space, was able to generate a policy within 0.5 seconds – on par with the other replanners. In all our tests variants of PBColgen with amenable parameters were able to match FF-Replan and Robust-FF in the sense that all returned their first policy within the time it took to take the first measurement.

How quickly does PBColgen's solution improve? First, we observe that PB-Colgen's quality does not improve monotonically. This is because the algorithm's reduced master problems do not correspond to the quality of the

current solution, but rather to the progress of proving optimality. This is not necessarily intuitive, so we elaborate in appendix A.

If we want PBColgen to exhibit monotonic increase in quality we can achieve this by keeping a "best policy so far" which is only updated once the policy improves. Clearly with this technique we need to specify by which criterion we are optimising, and evaluate the policy with respect to this criterion at each step. Note that this evaluation process adds overhead.

Regardless of PBColgen's non-monotonicity, we observe throughout all problems that variants of PBColgen with amenable parameters are able to provide a better quality policy than those of FF-Replan and Robust-FF very quickly. For instance of this, consider `exbw_p02-n3-N5-s2` (figure 6.6 and table 6.9). PBColgen takes marginally longer to produce its first policy, but already overtakes FF-Replan and Robust-FF in terms of quality by the next time 0.1 seconds. This is clearly not true for all parameter settings, as can be seen in e.g. figure 6.1, so this result should treated cautiously.

A common phenomenon in optimisation is that the optimal solution might be reached early on in the search, but most of the time is spent proving that this solution is infact the optimal. It is interesting to observe problems like `exbw_3` (figure 6.7 and table 6.10) confirm this entirely: the solvers with effective parameters take almost 50 times longer to terminate and prove optimality than to converge to the optimal policy; whereas other problems like `tire_2` (figure 6.3 and table 6.5) do not corroborate this, as most planners converge only slightly before terminating. This occurs because the former problem has unavoidable dead ends. So, before terminating and guaranteeing optimality, PBColgen attempts to make its policy proper by adding plans and cycles that deviate from the current policy, only to realise that they generate an inferior solution. On the second problem, tireworld, PBColgen finds the optimal proper policy, and can quickly exhaust the set of plans and cycles that leave the policy proper.

To summarise: FF-Replan and Robust-FF are able to construct policies very quickly, but do not improve over time, and almost always get trapped in suboptimal solutions. In problems like blocksworld where there are no dead ends and the penalty of not taking probabilistic effects into account is minor, these planners are able to perform well. In constrast, on problems with dead ends, these planners perform poorly in terms of quality.

PBColgen is slower to construct its initial policies – keeping in mind that variants that did not use Bellman-Ford (as seen in the tireworld domain) delivered promising results on this front. If we consider the strongest candidates

from PBColgen, once PBColgen has its initial policy, it quickly overtakes FF-Replan and Robust-FF in terms of quality.

# Chapter 7

# Conclusion

## 7.1 Summary

The focus of this thesis was to combine the efficiency of replanners and the guarantees provided by a column generation framework into a replanner that can construct an optimal solution – according to some criteria which can be easily adapted; as was summarised in our research goal:

*Can column generation be used to build an effective replanner with optimality guarantees?*

Our contribution is PBColgen, which is developed throughout chapter 5. It embodies a replanner that uses column generation to construct optimal solutions. That is, PBColgen takes in an SSP and, within the column generation framework, it constructs pricing problems, solves these subproblems as negative-weighted deterministic planning problems, and combines these solutions into a policy. Column generation guarantees that the algorithm will terminate, and the constructed policy will be optimal.

Unfortunately our subproblems are not classical deterministic planning problems, as they allow negative cycles. We introduce some algorithms to solve this unusual problem in section 5.4 – noting that we are not interested in the most negative path, but rather in some negative path or cycle. Furthermore we point out that on an interesting subset of SSPs, namely acyclic SSPs, we can adapt the problem into a standard deterministic planning problem – which will allow us to use state-of-the-art deterministic solvers. Note that due to time restrictions this latter approach was left for future work.

To quantify PBColgen's effectiveness, we compared its performance to other

successful replanners, namely, FF-Replan and Robust-FF in chapter 6. The results confirmed PBColgen's theoretical properties, i.e. convergence to an optimal solution given enough time, and showed promising behaviour as an online solver.

So, PBColgen demonstrates that it is certainly possible to build a replanner with optimality guarantees in the column generation framework, and our tests indicate that in the context of replanners, it can be effective.

## 7.2 Future Work

### 7.2.1 Implementation Improvements

Due to time restrictions and my inexperience with c++ and the programming framework, some aspects of the implementation were left incomplete.

One particularly disruptive issue is numerical instability. That is, the numerical imprecision of floating point arithmetic can cause issues throughout multiple stages of the algorithm. In some domains, this becomes an issue as column generation gets stuck in a loop, because the deterministic solver finds plans or cycles that do not improve the solution, and thus have non-negative cost; however, due to numerical instability, it gets assigned a very slight negative cost, and thus gets added. As this column does not improve the solution, the pricing problem remains the same, and we find the same column in the next iteration. We deal with this by introducing a small $\epsilon \in \mathbb{R}_{>0}$ value, and stop column generation once we are unable to find a plan or cycle with cost less than $\epsilon$. With carefully selected $\epsilon$ this is not a problem, but if $\epsilon$ is too large we may not find the optimal solution.

Numerical instablility also introduces issues in domains with "difficult" probabilities. We discussed this particular problem in 6.3.3.

In the future, we will look into ways to reduce the error of floating point arithetic, and introduce methods that allow the algorithm to deal with small deviations. For instance, in linear programs we can add slack variables that take up the small error.

Furthermore, there are parts of the code which are likely not implemented in the most efficient manner, e.g. we had to use `std::list` structures rather than `std::vector` due to a quirk in the framework – it may not make a large difference, but generally vectors are more efficient as they are contiguous in memory, and thus have better performance with respect to cache lines.

### 7.2.2 Deterministic Solvers

Profiling PBColgen revealed that the majority of CPU time was taken up by the deterministic solvers – typically in the range of 60% to 90%. We anticipate that there are more efficient solvers for the pricing problems than we presented.

In the case of acyclic SSPs this is obvious. With some modifications to the pricing problem we can use state-of-the-art deterministic solvers and heuristics to solve them, which is almost certain to deliver stronger performance. As discussed in section 5.4.6, this is an interesting category of problems, and so we consider it worthwhile to investigate this approach.

For cyclic SSPs, it is likely that there are more efficient algorithms than those we discussed. A preliminary investigation did not yield many relevant results for the unusual problem of finding some negative plan or cycle on negative-weighted deterministic planning problems, so a more in-depth literature review of existing algorithms is required.

### 7.2.3 Extensions to Column Generation

There is a technique in column generation that allows us to get a lower bound for the optimal policy cost in PBColgen. Clearly, given a policy $\pi$ from a reduced master problem of PBColgen, the cost of $\pi$ gives us an upper bound for the cost of an optimal policy $\pi^*$ – since $\pi$ can minimally cost as much as $\pi^*$ if it is itself optimal. For column generation algorithms with a minimisation master problem (and respectively reduced master problems) we can obtain a lower bound for the optimal objective of the master problem [Lübbecke and Desrosiers 2005, pp. 8–9]. In future work, we are interested in applying this result to PBColgen. This would allow for some extra possibilities, e.g. it would let PBColgen terminate when its solution is sufficiently close to the lower bound, as defined by the user.

### 7.2.4 Further Experimentation

Unfortunately, we did not have the time to answer all our questions with our experiments.

We would like to determine which of PBColgen's optimisation parameters are effective in which circumstances. This information would allow us to enhance those that are effective, remove those that are not, and provide a cleaner algorithm for the user in the sense that a user should not have to tweak parameters manually.

We limited our experiments to compare replanners. As PBColgen is also an optimal planner, it would be interesting to compare it to state-of-the-art optimal planners like LRTDP and ILAO$^*$.

## 7.3 Final Remarks

We believe the PBColgen algorithm is an interesting contribution to the planning community. Amongst replanners it is novel that an algorithm can give guarantees in terms of optimality, and the flexibility of linear programming allows the algorithm to be adapted to various optimality criteria.

# Bibliography


Ahuja, Ravindra K., Thomas L. Magnanti, and James B. Orlin (1993). *Network flows: theory, algorithms, and applications*. Englewood Cliffs, N.J: Prentice Hall. ISBN: 9780136175490.

Barto, Andrew G., Steven J. Bradtke, and Satinder P. Singh (1995). "Learning to act using real-time dynamic programming". In: *Artificial Intelligence* 72.1, pp. 81–138. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(94)00011-O. URL: http://www.sciencedirect.com/science/article/pii/000437029400011O.

Bellman, Richard (1957). "A Markovian Decision Process". In: *Journal of Mathematics and Mechanics* 6.5, pp. 679–684. ISSN: 00959057, 19435274. URL: http://www.jstor.org/stable/24900506.

Bertsekas, D. and J. Tsitsiklis (1991). "An Analysis of Stochastic Shortest Path Problems". In: *Math. Oper. Res.* 16, pp. 580–595.

Bertsimas, Dimitris and John Tsitsiklis (Jan. 1998). *Introduction to Linear Optimization*.

Bonet, Blai and H. Geffner (Jan. 2003). "Labeled RTDP: Improving the convergence of real-time dynamic programming". In: *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS'03)*, pp. 12–21.

Cormen, Thomas H. et al. (2009). *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press. ISBN: 0262033844.

d'Epenoux, Francois (1963). "A probabilistic production and inventory problem". In: *Management Science* 10.1, pp. 98–108.

Geffner, Hector and Blai Bonet (2013). *A concise introduction to models and methods for automated planning*. eng. OCLC: 867140422. San Rafael, Calif: Morgan & Claypool. ISBN: 9781608459698.

Geisser, F. et al. (2020). "Optimal and Heuristic Approaches for Constrained Flight Planning under Weather Uncertainty". In: *Proc. of 30th Int. Conf. on Automated Planning and Scheduling (ICAPS)*. URL: http://felipe.trevizan.org/papers/geisser20:flight.pdf.

Kolobov, Andrey, Mausam, and Daniel Weld (2012). *A Theory of Goal-Oriented MDPs with Dead Ends*. arXiv: 1210.4875 [cs.AI].

Little, I. and S. Thiébaux (2007). "Probabilistic planning vs replanning". In:

Lübbecke, Marco E. and Jacques Desrosiers (Dec. 2005). "Selected Topics in Column Generation". en. In: *Operations Research* 53.6, pp. 1007–1023. ISSN: 0030-364X, 1526-5463. DOI: 10.1287/opre.1050.0234. URL: http://pubsonline.informs.org/doi/abs/10.1287/opre.1050.0234 (visited on 06/27/2020).

Puterman, Martin L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. 1st. USA: John Wiley & Sons, Inc. ISBN: 0471619779.

Russell, Stuart and Peter Norvig (2010). *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall.

Teichteil-Königsbuch, Florent and Guillaume Infantes (Jan. 2008). "RFF: A Robust, FF-Based MDP Planning Algorithm for Generating Policies with Low Probability of Failure". In:

Trevizan, F., F. Teichteil-Königsbuch, and S. Thiébaux (2017). "Efficient Solutions for Stochastic Shortest Path Problems with Dead Ends". In: *Proc. of 33rd Int. Conf. on Uncertainty in Artificial Intelligence (UAI)*. URL: http://felipe.trevizan.org/papers/trevizan17:mcmp.pdf.

Trevizan, F., S. Thiébaux, and P. Haslum (2017). "Occupation Measure Heuristics for Probabilistic Planning". In: *Proc. of 27th Int. Conf. on Automated Planning and Scheduling (ICAPS)*. URL: http://felipe.trevizan.org/papers/trevizan17:hpom.pdf.

Trevizan, F., S. Thiébaux, P. Santana, et al. (2016). "Heuristic Search in Dual Space for Constrained Stochastic Shortest Path Problems". In: *Proc. of 26th Int. Conf. on Automated Planning and Scheduling (ICAPS)*. URL: http://felipe.trevizan.org/papers/trevizan16:idual.pdf.

Wilt, Christopher and Wheeler Ruml (Jan. 2012). "When does weighted A* fail?" In: *Proceedings of the 5th Annual Symposium on Combinatorial Search, SoCS 2012*, pp. 137–144.

Yoon, Sung, Alan Fern, and Robert Givan (Jan. 2007). "FF-Replan: A Baseline for Probabilistic Planning". In: pp. 352–.

Younes, Håkan et al. (July 2005). "The First Probabilistic Track of the International Planning Competition". In: *J. Artif. Intell. Res. (JAIR)* 24, pp. 851–887. DOI: 10.1613/jair.1880.

# Appendix A

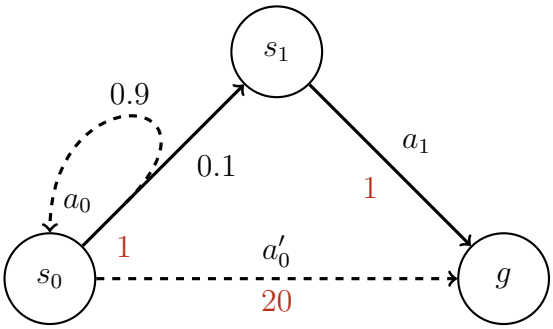
# Non-monotonicity of PBColgen

At first it may be surprising that PBColgen's policies are not monotonic in terms of quality, as we expect the objective of PBColgen's reduced master problems to improve monotonically – but, the reduced master problem's objective does not align with the policy's quality until the algorithm terminates. In short, this is because PBColgen (with the generalised give-ups) must pump flow through give-up actions for all effects that are not accounted by some plan or cycle – even if in the SSP these effects are already accounted for by the policy.

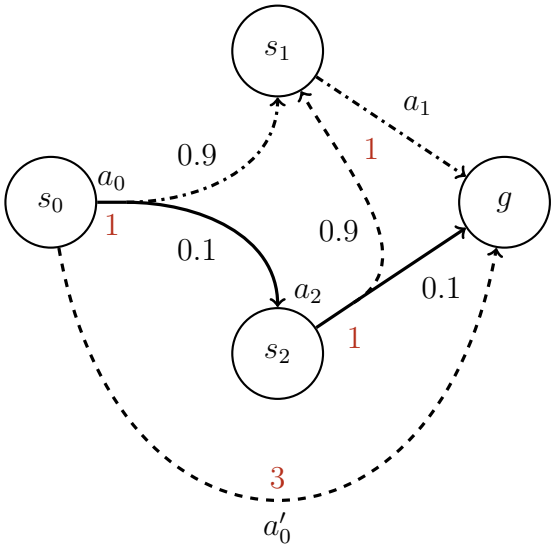
To illustrate this, consider the SSPs presented in figure A.1. In the first example (figure A.1a), we assume that PBColgen's RMP contains plan $\phi = s_0 \xrightarrow{a_0 \triangleright s_1} s_1 \xrightarrow{a_1 \triangleright g} g$ (as denoted by the solid line) in $\hat{\Phi}$. It turns out that the casted plan $\pi_\phi$ is already optimal, since it can only fail by looping back to itself, but PBColgen does not recognise this, and assigns the current solution a value of $0.1 \cdot 1 + 0.9 \cdot D$ where $D$ is the dead-end penalty. PBColgen computes the objective of the solution with $s_0 \xrightarrow{a'_0 \triangleright g} g$ to be 20, which reduces the objective in the reduced master problem (assuming $D > \frac{19.9}{0.9} \approx 22.1$) – even though we know it replaces the optimal policy with a suboptimal one. This scenario can not occur with the depth probe optimisation, so we present the second example (figure A.1b) which can still occur.

In example presented in figure A.1b PBColgen combines

$$\phi_{\text{top}} = s_0 \xrightarrow{a_0 \triangleright s_1} s_1 \xrightarrow{a_1 \triangleright g} g$$




(a) The plan indicated by the solid line can fail by looping back to itself.




(b) The plan indicated by solid lines may fail by ending up on the plan indicated by dash-dotted lines.

Figure A.1: Simple SSPs to demonstrate why PBColgen's solution quality is non-monotonic.

as indicated by the dash-dotted lines, and

$$\phi_{\text{bot}} = s_0 \xrightarrow{a_0 \triangleright s_2} s_2 \xrightarrow{a_2 \triangleright g} g$$

as indicated by the solid lines. The only way that $\phi_{\text{bot}}$ can fail is by ending up in a state in $\phi_{\text{top}}$, which means that the policy which combines these plans is closed, and in this case optimal. Nevertheless, PBColgen has no mechanism to realise this, and may in the next iteration add plan $s_0 \xrightarrow{s'_0 \triangleright g} g$, as this reduces the objective of the reduced master problem.

With a slight modification to the example in figure A.1a, we can obtain the SSP in figure A.2. Here, by similar argument as before, the current objective is $0.1 \cdot 1 + 0.9 \cdot D$, and so PBColgen may introduce the plan $s_0 \xrightarrow{a'_0 \triangleright g} g$ with an objective of $0.2 \cdot 2 + 0.8 \cdot D$ (assuming that dead-end penalty $D > 3$), which decreases the generated policy's probability of reaching a goal from 1 to 0.2.

Note that the objective of the reduced master problems' linear programs exhibits (non-strict) monotonic decrease, and it is this which guarantees that

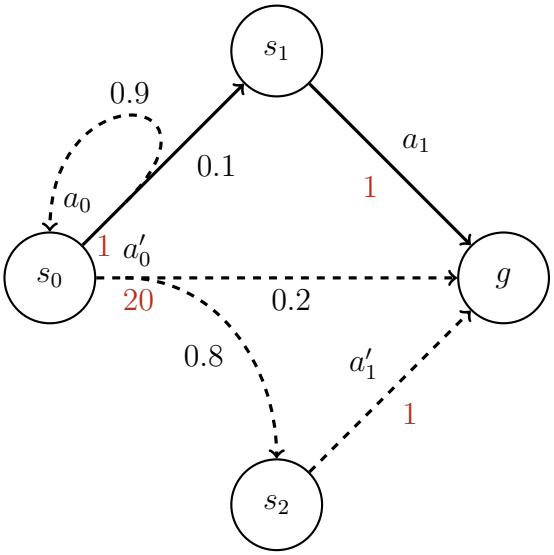



Figure A.2: This SSP demonstrates that PBColgen may select a combination of plans and cycles that decrease the probability of reaching a goal.

PBColgen eventually converges to an optimal solution.