Efficient Constraint Generation for Stochastic Shortest Path Problems

Johannes Schmalz[1,2] and Felipe Trevizan[1]

[1]Australian National University, Australia
[2]Saarland University, Germany

Abstract

Stochastic Shortest Path problems (SSPs) are traditionally solved by computing each state's cost-to-go by applying Bellman backups. A Bellman backup updates a state's cost-to-go by iterating through every applicable action, computing the cost-to-go after applying each one, and selecting a minimal action's cost-to-go. State-of-the-art algorithms use heuristic functions; these give an initial estimate of costs-to-go, and lets the algorithm apply Bellman backups only to promising states, determined by low estimated costs-to-go. However, each Bellman backup still considers all applicable actions, even if the heuristic tells us that some of these actions are too expensive, with the effect that such algorithms waste time on unhelpful actions. To address this gap we present a technique that uses the heuristic to avoid expensive actions, by reframing heuristic search in terms of linear programming and introducing an efficient implementation of constraint generation for SSPs. We present CG-iLAO*, a new algorithm that adapts iLAO* with our novel technique, and considers only 40% of iLAO*'s actions on many problems, and as few as 1% on some. Consequently, CG-iLAO* computes on average 3.5× fewer costs-to-go for actions than the state-of-the-art iLAO* and LRTDP, enabling it to solve problems faster an average of 2.8× and 3.7× faster, respectively.

1 Introduction

Planning problems are abstract tasks where an agent must move through a system of states. To do so, the agent is given a set of actions that move it through the states according to some rules. This type of problem is ubiquitous throughout theory and practice, and has been used to encode many real world problems, e.g., finding the shortest route on a map, coordinating warehouse robots to fulfil orders in an efficient and collaborative way [Koenig et al., 2023], and producing fuel-efficient flight routes that take into account bad weather and complex aviation restrictions [Geißer et al., 2020]. In the classical planning problem, actions move the agent between states in a deterministic way. This paper focuses on probabilistic planning, in particular Stochastic Shortest Path Problems (SSPs) [Bertsekas and Tsitsiklis, 1991]. SSPs generalise classical planning by making the effect of applying an action probabilistic: when the agent applies an action, it moves into its next state according to a probability distribution that is known a priori. These have been used to model many practical problems, e.g., penetration testing (pentesting) [Hoffmann, 2015], ecological management [Péron et al., 2017], and management of hydroelectric and thermal power generation [White, 1985].

All state-of-the-art algorithms for optimally solving SSPs compute the optimal costs-to-go of relevant states. The cost-to-go of a state s can be understood as the expected cost that the agent incurs by starting at s and repeatedly applying actions until it reaches a goal. Respectively, the optimal cost-to-go gives the expected cost to reach a goal with an optimal policy, which minimises the expected costs. The purpose of computing the optimal costs-to-go is that they induce an optimal policy: if the agent chooses its action for state s greedily, i.e., it picks an action that minimises the optimal cost-to-go, then this induces an optimal policy. The state-of-the-art algorithms are all based on one particular algorithm for computing optimal costs-to-go, called Value Iteration (VI) [Bellman, 1957]. VI is a dynamic programming

Email addresses: johannes.schmalz@anu.edu.au (Johannes Schmalz), felipe.trevizan@anu.edu.au (Felipe Trevizan).

This manuscript has been accepted for publication in Artificial Intelligence. The final version of record is available at: https://doi.org/10.1016/j.artint.2026.104505

algorithm that iteratively applies *Bellman backups* to converge to the optimal costs-to-go. A Bellman backup applied to state s

- evaluates the cost-to-go after applying each applicable action a,
- selects an action with minimal cost-to-go, and then
- sets the cost-to-go of s to the minimal action's cost-to-go.

The two state-of-the-art algorithms for solving SSPs, iLAO* [Hansen and Zilberstein, 2001] and LRTDP [Bonet and Geffner, 2003b], build on VI and can be orders of magnitude faster because they use heuristics to guide their search. VI is a blind-search algorithm, which means it considers all states in the problem. In contrast, iLAO* and LRTDP have access to heuristic functions that estimate the optimal costs-to-go for all states, which lets the algorithms focus on promising states with small heuristic values, and avoid unpromising states with large values. Thus, heuristic-search algorithms apply Bellman backups only to a promising subset of states. In practice, heuristic-search algorithms scale to much larger problems than blind-search algorithms because the promising subset of states is usually orders of magnitude smaller than the total reachable state space.[1] So, these heuristic-search algorithms enjoy a significant performance advantage by restricting the number of states considered, but there is a gap because they do not restrict the actions to be considered. These state-of-the-art algorithms are fundamentally built on Bellman backups, so they always consider all applicable actions for their states, but it is not clear that heuristic-search algorithms need to do this — they can use their heuristics to avoid not only expensive states, but also actions that lead to expensive states. This is an important issue because considering unneeded actions is not a one-time expense: in SSP planning, states must often be backed up many times before the algorithm converges. In particular, cycles require many backups before converging to the correct cost-to-go. Also, the search may reconsider the same states in many of their iterations, for example, s_0 is backed up in every iteration of LRTDP and iLAO*. Each time the state is backed up we compute all its actions' costs-to-go, thus incurring many unneeded cost-to-go computations.

To make this concept of unneeded actions that waste computation more concrete, consider the grid-world example in figure 1a, which was used as a demonstration by Hansen and Zilberstein [2001]. The agent must navigate from the starting cell to the goal cell using the movement actions North (N), East (E), and South (S), each of which, assuming that the destination is unblocked, has a probability 0.5 of moving the agent to the expected cell, and a 0.5 probability of remaining in the same cell. Figure 1b shows which states and actions are considered by iLAO* after 2 iterations.[2] Observe that there are unneeded actions:

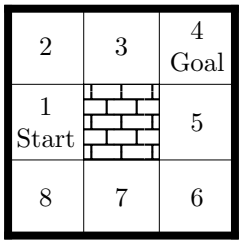
- the heuristic (shown in the bottom of each node) determines that cell 8 is too expensive and it will never be expanded, but nevertheless, the action S from cell 1 needs to be considered and its cost-to-go re-evaluated in each Bellman backup for cell 1; and
- actions such as S and N in cell 2 do not progress the agent and will never have the lowest costs-to-go, but their costs-to-go are still evaluated in each Bellman backup on cell 2.

These actions are not useful for finding the optimal solution, and the heuristic values make this clear — iLAO* is wasting computational resources by considering these actions and re-evaluating their costs-to-go each time the relevant states are backed up.

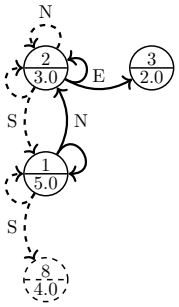
There is an existing approach for removing such unneeded actions called *action elimination* [Bertsekas, 1995]. Action elimination enables an algorithm to prove that an action can never be part of the optimal solution, so that the action can be subsequently permanently removed from search, i.e., eliminated. Action elimination requires a lower bound and an upper bound on an SSP's costs-to-go, and then determines that an action $\tilde{a}$ may be eliminated if the lower bound on $\tilde{a}$'s cost-to-go is larger than the upper bound on another action a's cost-to-go, i.e., $\text{LB}(\tilde{a}) > \text{UB}(a)$. Domain-independent lower bounds (a.k.a. admissible heuristics) can be automatically computed efficiently; however, we are not aware of any efficient ways to automatically derive informative domain-independent upper bounds. Consequently, action elimination has not found much use in domain-independent planning, and FTVI [Dai, Mausam, and Weld, 2009] is the only algorithm we know that uses action elimination in the context of optimal heuristic planning for SSPs. There are other algorithms that use upper bounds, but they do not explicitly use

[1] There are situations where heuristic search is not faster, e.g., if the heuristic gives misleading estimates of the optimal costs-to-go. However, with the heuristics and problems we consider, heuristic search is always significantly faster.

[2] In the original paper [Hansen and Zilberstein, 2001] this figure is used as a demonstration of LAO*, but in this case LAO* behaves the same as iLAO*.



(a) Probabilistic navigation problem.



(b) iLAO*'s states and actions after 2 iterations. The top number describes the cell index and the bottom number is the corresponding state's heuristic value.

Figure 1: A gridworld probabilistic navigation problem and the states and actions that iLAO* considers on this problem after 2 iterations. This example was taken from Hansen and Zilberstein [2001].

action elimination to remove actions from their search, e.g., BRTDP [McMahan, Likhachev, and Gordon, 2005], FRTDP [Smith and Simmons, 2006], VPI-RTDP [Sanner et al., 2009], and IBLAO* [Warnquist, Kvarnström, and Doherty, 2010].

So, the state-of-the-art algorithms for SSPs suffer from considering all applicable actions, even when it is clear that some actions are not needed for finding the optimal solution, and action elimination does not satisfactorily address this gap. We address this open problem by exploiting connections between planning and operations research, and use the technique of constraint generation to intelligently select promising actions. More concretely: our algorithm initially considers none of the actions, and only adds actions lazily when they are deemed promising by the current costs-to-go, which is determined with theory from constraint generation. This contrasts with action elimination, which initially considers all actions and only removes them when it can prove they are suboptimal. Moreover, action elimination requires lower and upper bounds, whereas we only need lower bounds. Our approach improves upon state-of-the-art algorithms by saving on cost-to-go computations, which action elimination has not been able to do. This is an important step in heuristic search, which lets algorithms use heuristics to select actions as well as states.

We introduce the novel algorithm CG-iLAO*, which uses our technique of ignoring actions and adding them as required to generalise iLAO*. We confirm experimentally that CG-iLAO* often considers only 40% of iLAO*'s actions, and in some cases considers as little as 1%. Consequently, CG-iLAO* computes 3.5× fewer Q-values on average, and solves problems 2.8× and 3.7× faster than iLAO* and LRTDP, respectively. In specific problems, the savings in Q-values reach up to 80× resulting in a speedup of over 50×.

We give a breakdown of this paper's contents by section:

- Section 2 gives the background for SSPs, value functions, Bellman backups, and Value Iteration.
- Section 3 describes iLAO*, a state-of-the-art heuristic search algorithm for SSPs upon which we build. We also present a novel way to view iLAO* under the lens of linear programming.
- Section 4 introduces our novel algorithm CG-iLAO*, and give proofs of its correctness.
- Section 5 describes different expansions methods that can be used for CG-iLAO*.
- Section 6 discusses action elimination as an alternative way to cope with unneeded actions. It also gives the context of existing constraint generation approaches in planning and relates CG-iLAO* to the Partial Expansion A* algorithm.
- Section 7 describes the setup and results of our experiments.
- Section 8 and section 9 are the conclusion and future work.

2 Background

Stochastic Shortest Path problems (SSPs) [Bertsekas and Tsitsiklis, 1991] are used to model problems where an agent must navigate through a state space, using actions with probabilistic effects that are known a priori:

Definition 1 (Stochastic Shortest Path problem (SSP) [Bertsekas and Tsitsiklis, 1991])**.** An SSP is defined by the tuple $\langle \mathsf{S}, s_0, \mathsf{G}, \mathsf{A}, P, C \rangle$ where:

- S is the finite set of states;
- $s_0 \in \mathsf{S}$ is the agent's starting state, called the initial state;
- $\mathsf{G} \subsetneq \mathsf{S}$ is the set of goal states – the agent finishes its task successfully if it reaches one of these. We assume $\mathsf{G} \neq \emptyset$ and $s_0 \notin \mathsf{G}$;
- A is the finite set of actions available to the agent and $\mathsf{A}(s) \subseteq \mathsf{A}$ denotes the actions applicable in state s;
- $P(s'|s,a)$ indicates the probability of reaching s' after applying action a to state s;
- $C(s,a) \in \mathbb{R}_{>0}$ gives the cost incurred when the agent applies action a in state s.

The set of states that may be reached after applying action a to state s are called the successors of s and a, and are given by $\mathrm{succ}(s,a) \stackrel{\text{def}}{=} \{s' \in \mathsf{S} : P(s'|s,a) > 0\}$. SSPs are solved by policies, which are potentially partial functions $\pi : \mathsf{S} \to \mathsf{A}$ that map states s that the agent may encounter onto actions $\pi(s)$ that the agent should apply there. $\mathsf{S}^{\pi,s} \subseteq \mathsf{S}$ denotes the set of states that can be reached by following π from s, and is called the policy envelope of π rooted at s. In this work we only consider policy envelopes rooted at s_0, so we write $\mathsf{S}^\pi = \mathsf{S}^{\pi,s_0}$ and call S^π the policy envelope of π. It is desirable that the agent knows what to do in each state it encounters, and eventually reaches the goal, which leads to the following definitions:

Definition 2 (Closed and Open Policies)**.** A policy π is *closed* w.r.t. s_0 if for all states in the policy envelope $s \in \mathsf{S}^\pi$, either $\pi(s)$ is defined or s is a goal state. It is open otherwise.

Definition 3 (Proper and Improper Policies)**.** A policy π is *proper* w.r.t. s_0 if, by following π from s_0, the agent will reach a goal with probability 1. It is improper otherwise.

Properness implies closedness. For closed policies π, their expected cost is the cumulative action cost incurred by the agent following π from s_0, over expectation. An optimal policy π^* is any proper policy that minimises the expected cost. In this paper, we make two standard assumptions for SSPs:

Assumption 1 (Reachability)**.** There exists at least one proper policy w.r.t. s_0.

Assumption 2. All improper policies have infinite expected cost.

Note that $\mathsf{A}(s) \neq \emptyset$ for all $s \in \mathsf{S} \setminus \mathsf{G}$ as a consequence of assumption 2. If this were not the case, then there could be a state $s \in \mathsf{S} \setminus \mathsf{G}$ with $\mathsf{A}(s) = \emptyset$ and we could construct an example where a policy π reaches s with probability 1 — such π's expected cost is finite even though π is improper. In our experiments, we consider SSPs that violate assumption 1, i.e., SSPs with dead ends; we address this by applying the fixed-penalty transformation of SSPs [Trevizan, Teichteil-Königsbuch, and Thiébaux, 2017], which yields a new SSP that has no dead ends and is equivalent to the original SSP, provided that the penalty term is large enough. Other approaches for relaxing our assumptions on SSPs, such as $\mathrm{S}^3\mathrm{P}$ [Teichteil-Königsbuch, 2012], can also be implemented without significant changes.

Value Function. A value function $V : \mathsf{S} \to \mathbb{R}_{\geq 0}$ tries to encode the costs-to-go of each state, and we will use V and costs-to-go interchangeably. We use the following notations:

- Given V, the Q-value of s and a is a common shorthand $Q(s,a) \stackrel{\text{def}}{=} C(s,a) + \sum_{s' \in \mathsf{A}(a)} P(s'|s,a)V(s)$, which can be read as the expected cost-to-go from s if the agent applies a.
- Whenever we have a value function V^x or V_y we will refer to its associated Q-values as Q^x or Q_y respectively.
- We will write $V \leq V'$ to compare value functions element-wise, i.e., $V \leq V'$ iff $V(s) \leq V'(s)\ \forall s \in \mathsf{S}$, and similarly for other comparators, e.g., $=, \geq$.

The optimal value function V^* gives optimal costs-to-go for each state, and is the unique fixed-point solution to the following set of equations, called the Bellman equations:

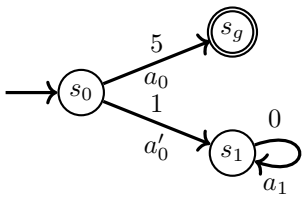
$$V(s) = \begin{cases} 0 & \text{if } s \in \mathsf{G} \\ \min_{a \in \mathsf{A}(s)} Q(s,a) & \text{if } s \in \mathsf{S} \setminus \mathsf{G}. \end{cases} \tag{1}$$

Although an SSP may have multiple optimal policies, the optimal value function is unique.

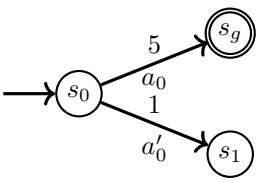
Given a value function V, its greedy policy is $\pi_V(s) \stackrel{\text{def}}{=} \operatorname{argmin}_{a \in \mathsf{A}(s)} Q(s,a)$, where any ties can be broken arbitrarily. The set of all greedy policies for V^*, obtained by breaking ties in all possible ways, is precisely the set of optimal policies. To avoid dealing with multiple greedy policies we make the following tie-breaking assumption:

Assumption 3 (Tie-breaking for Greedy Policies)**.** Given V, we ensure the greedy policy π_V is unique by breaking ties in $\operatorname{argmin}_{a \in \mathsf{A}(s)} Q(s,a)$ with arbitrary but reproducible tie-breaking rules. An example of such a rule is lexicographical tie breaking.

Now that we have defined value functions and greedy policies, we briefly justify assumption 2. In figure 2 we present two SSPs that violate assumption 2. Notably, in figure 2a the state s_1 has a zero-cost action to itself, and in figure 2b the state s_1 has no applicable actions. Both SSPs have optimal value functions V^* with $V^*(s_0) = 1, V^*(s_1) = 0, V^*(s_g) = 0$ (arguably $V^*(s_1) = -\infty$ for figure 2b, depending on how we define the min operator over the empty set), and the greedy policies π_{V^*} have $\pi_{V^*}(s_0) = a'_0$, making them improper, even though proper policies exists. Thus, the optimal value function V^* does not induce an optimal policy. In general, for SSPs that violate assumption 2, an optimal value function V^* need not induce an optimal policy with π_{V^*}, and moreover, V^* need not even be unique, e.g., $V^*(s_0) = 3, V^*(s_1) = 2, V^*(s_g) = 0$ satisfies the Bellman equations for figure 2a. Assumption 2 avoids such issues, and prohibits both examples in figure 2. In practice, it is difficult to ensure an SSP satisfies assumption 2, so it is common to make stronger assumptions that are easy to check. In particular, it is common to require all states s to have $\mathsf{A}(s) \neq \emptyset$ and restricting to strictly positive action costs to prohibit zero-cost cycles (and their generalisation, zero-cost end components), which is sufficient, but not necessary to ensure assumption 2 holds. We take this practical approach, except when stated otherwise.



(a) SSP with zero-cost cycle.



(b) SSP with $\mathsf{A}(s_1) = \emptyset$.

Figure 2: SSPs that violate assumption 2, and consequently have optimal value functions that induce suboptimal policies. One SSP violates it with a zero-cost cycle, and the other SSP violates it by having a state with no applicable actions.

Value Iteration (VI) [Bellman, 1957] is the foundational algorithm for finding a solution to the Bellman equations (1), and thereby solving an SSP, that modern algorithms build on. VI starts with an arbitrary value function V^0, and computes a sequence of value functions $V^1, V^2, \ldots$ which in the limit converge to V^*. These value functions are computed by applying *Bellman backups*:

Definition 4 (Bellman Backup)**.** Given value functions V and V', a Bellman backup applied to state s populates $V'(s)$ with

$$V'(s) \leftarrow \min_{a \in \mathsf{A}(s)} Q(s,a)$$

and does not update $V'(s')$ for any other state $s' \neq s$.

To compute V^{t+1} for $t \in \mathbb{N}_{>0}$, the classical version of VI applies Bellman backups on V^t and V^{t+1} for each $s \in \mathsf{S}$. Asynchronous VI (also called Gauss-Seidel VI) computes V^{t+1} by initialising it with $V^{t+1} = V^t$ and then it only performs a Bellman backup for a single state in each t; but importantly, it ensures that all states are backed up fairly, i.e., if the sequence is infinite then each state is updated infinitely often. In both versions, for any choice of V^0, the sequence of value functions produced by VI will asymptotically converge to the optimal value function, i.e., $\lim_{t \to \infty} V^t = V^*$ [Bertsekas, 1995]. In practice, VI requires

a termination condition that is satisfied in finite time. VI considers the difference between successive value functions, measured by the *Bellman residual*, defined as $\text{RES}(s) \stackrel{\text{def}}{=} |V^t(s) - \min_{a \in \mathsf{A}(s)} Q^t(s,a)|$ and terminates when the residual between successive value functions is small over all states. Formally, for a user-selected value $\epsilon \in \mathbb{R}_{>0}$, VI terminates upon the following condition:

Definition 5 (Global ϵ-consistency)**.** V is globally ϵ-consistent for an SSP $\mathbb{S}$ when $\text{RES}(s) \leq \epsilon \; \forall s \in \mathsf{S}$.

Importantly, when VI is stopped with this condition its value function is only an approximate solution and for any ϵ it is possible to construct an SSP so that VI terminates with V whose greedy policy is not optimal. In practice, for reasonably small ϵ, e.g., 0.0001, the greedy policy for globally ϵ-consistent V will be optimal or close to optimal. Moreover, there is a bound on how much such a V can deviate from V^*:

Definition 6. Let $N(s, V, \mathbb{S}) = \max\{N^*(s), N^{\pi_V}(s)\}$ where $N^{\pi_V}(s)$ denotes the expected number of steps to reach some goal from s by following the greedy policy π_V, and $N^*(s)$ gives the maximum expected number of steps to reach a goal from s over $\mathbb{S}$'s optimal policies.

Theorem 1 (VI Error [Mausam and Kolobov, 2012])**.** Consider globally ϵ-consistent V. Then,

$$|V(s) - V^*(s)| \leq \epsilon N(s, V, \mathbb{S}) \; \forall s \in \mathsf{S}.$$

Note that the error term $\epsilon N(s, V, \mathbb{S})$ can get significantly larger than ϵ. Nevertheless, $N(s, V, \mathbb{S})$ can be bounded from above, and as $\epsilon \to 0$ the error disappears and VI's V converges to V^*.

A critical shortcoming of VI is that it applies Bellman backups to every single state $s \in \mathsf{S}$. This means VI can not scale to large problems, which is especially problematic for compactly encoded problems, such as Probabilistic PDDL [Younes et al., 2005], where the state space can grow exponentially with respect to the problem size. An important insight is that not all states are needed to define an optimal policy, and we only need V^* for the optimal policy's envelope. Then, instead of considering global ϵ-consistency, we consider ϵ-*consistency*:

Definition 7 (ϵ-consistency [Bonet and Geffner, 2003a])**.** A value function V is ϵ-consistent for an SSP $\mathbb{S}$ if $\text{RES}(s) \leq \epsilon \; \forall s \in \mathsf{S}^{\pi_V}$.

Note that ϵ-consistency relies on π_V, which is ambiguous when V has multiple greedy policies; so, we fall back on assumption 3 and assume that ties are broken such that V has a single greedy policy. This definition of ϵ-consistency requires the residual to be small for states in the greedy policy's envelope, and states outside can have arbitrarily large residual. Importantly, to ensure that ϵ-consistent value functions produce optimal policies as $\epsilon \to 0$, we need additional properties over the value functions. In particular, we require *admissibility*:

Definition 8 (Admissible Value Function)**.** V is admissible if $V \leq V^*$.

An admissible value function is simply a lower bound on the optimal value function. Without admissibility, ϵ-consistency does not produce optimal policies. For example, consider figure 3: the value function shown in the bottom of nodes is ϵ-consistent w.r.t. the greedy policy with $\pi_V(s_0) = a_0$, but it is inadmissible, and fails to capture the optimal policy with $\pi^*(s_0) = a'_0$.

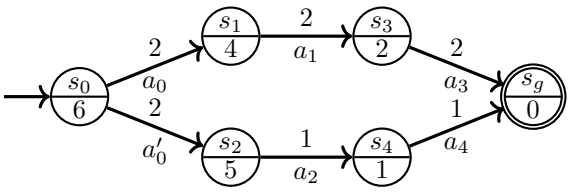


Figure 3: An SSP with a value function that is ϵ-consistent but inadmissible, and its greedy policy is suboptimal.

Another useful condition for value functions, which is stronger and implies admissibility, is *monotonicity*:

Definition 9 (Monotonic Value Function)**.** V is monotonic if $V(s) = 0 \; \forall s \in \mathsf{G}$ and

$$V(s) \leq \min_{a \in \mathsf{A}(s)} Q(s,a) \; \forall s \in \mathsf{S} \setminus \mathsf{G}.$$

An alternative name for monotonic value functions is consistent value functions, and it generalises the notion of consistent heuristics in deterministic planning. Monotonic value functions get their name from the property that Bellman backups will never decrease a value function, i.e., the sequence $V^0, V^1, V^2, \ldots$ obtained by applying Bellman backups is monotonically non-decreasing with $V^i \leq V^{i+1} \; \forall i \in \mathbb{N}$. This property holds for any V^{i+1} obtained from V^i via Bellman backups, for any arbitrary number of Bellman backups applied to an arbitrary set of states. Conveniently, Bellman backups preserve both admissibility and monotonicity. So, an admissible value function with arbitrarily many Bellman backups applied to an arbitrary set of states will yield another admissible value function, and similarly for monotonicity. Thus, an algorithm that is initialised with an admissible value function and only uses Bellman backups to modify V, has $V \leq V^*$ as an invariant.

Heuristic-search algorithms that are initialised with an admissible value function $V^0 \leq V^*$ and apply Bellman backups until ϵ-consistency is satisfied, produce optimal policies. Note that this contrasts with VI, which does not have any requirements over its initial value function V^0. We present this result formally, via the FIND-AND-REVISE framework:

Definition 10 (FIND-AND-REVISE Algorithm [Bonet and Geffner, 2003a])**.** A FIND-AND-REVISE algorithm applied to SSP $\mathbb{S}$ uses a value function V and consists of the steps

- FIND a state s in the greedy policy envelope S^{π_V} s.t. $\text{RES}(s) > \epsilon$,
- REVISE the found state by applying a Bellman backup to s,
- repeat these steps until no such state is found and V is ϵ-consistent w.r.t. $\mathbb{S}$.

The FIND-AND-REVISE schema describes many heuristic search algorithms, including the state-of-the-art iLAO* and LRTDP. FIND-AND-REVISE algorithms are optimal under the following conditions:

Theorem 2 (Optimality of FIND-AND-REVISE [Bonet and Geffner, 2003a; Mausam and Kolobov, 2012])**.** Consider a FIND-AND-REVISE algorithm initialised with an admissible heuristic, and terminated when its V is ϵ-consistent for any given $\epsilon \in \mathbb{R}_{>0}$. As $\epsilon \to 0$, its output $V(s)$ approaches $V^*(s)$ for $s \in \mathsf{S}^{\pi_V}$.

Other termination conditions are possible in place of ϵ-consistency, e.g., ϵ-optimality [Hansen and Zilberstein, 2001; Hansen and Abdoulahi, 2015]. We do not consider these alternatives because they build on top of ϵ-consistency, and are essentially automatic methods for selecting the parameter for ϵ-consistency so that stronger guarantees can be provided. Consequently, our ϵ-consistent algorithms can be adapted to their termination conditions.

3 iLAO*

In this section, we explain iLAO* [Hansen and Zilberstein, 2001] in detail, since it will be necessary for the insights and definition of CG-iLAO*. Then, we conclude the section by expressing iLAO* in terms of linear programming, which is a novel way to look at the algorithm and opens the door to define CG-iLAO* in section 4.

In each step, iLAO* considers a subproblem called a *partial SSP*, which is a copy of the original SSP with a subset of states and actions.[3] To define them, it is convenient to use SSPs with terminal costs:

Definition 11 (SSP with Terminal Costs)**.** An SSP with terminal costs is given by the tuple $\mathbb{S}_T = \langle \mathsf{S}, s_0, \mathsf{G}, \mathsf{A}, P, C, T \rangle$. All terms except T appear in the definition of SSPs, and are defined identically. The new term $T : \mathsf{G} \to \mathbb{R}_{\geq 0}$ is a terminal-cost function that applies the cost $T(g)$ once, whenever the agent reaches the goal $g \in \mathsf{G}$.

SSPs with terminal costs are not more expressive than regular SSPs, and they are in fact equivalent.[4] We use SSPs with terminal cost only because they are more convenient for the definition of partial SSPs:

[3] Partial SSPs are called explicit graphs in the original paper [Hansen and Zilberstein, 2001].

[4] SSPs with terminal costs can be converted to regular SSPs thus: add a new artificial goal state $\overline{g}$, make all old goal states non-goals, and encode the one-time terminal cost for each old goal state g with a deterministic action that leads to $\overline{g}$ with cost $T(g)$.

Algorithm 1: iLAO*

Function *iLAO** $(\mathbb{S}, H, \epsilon)$
  $\widehat{\mathbb{S}} \leftarrow$ partial SSP $\langle \{s_0\}, s_0, \{s_0\}, \emptyset, P, C, H \rangle$
  $V \leftarrow$ Value Function initialised by H
  $\widehat{\pi}_{\text{curr}} \leftarrow$ candidate policy initialised as undefined everywhere (used to approximate $\widehat{\pi}_V$)
  repeat
    $\mathcal{E} \leftarrow$ post-order DFS traversal of $\widehat{\pi}_{\text{curr}}$ from s_0
    $\widehat{\mathbb{S}}, \widehat{\pi}_{\text{curr}} \leftarrow$ Expand-Fringes$(\mathbb{S}, \widehat{\mathbb{S}}, \widehat{\pi}_{\text{curr}}, \mathcal{E})$
    $\mathsf{F} \leftarrow \mathsf{S}^{\widehat{\pi}_{\text{curr}}} \cap (\widehat{\mathsf{G}} \setminus \mathsf{G})$
    $V, \text{RES}, \widehat{\pi}_{\text{curr}}, \widehat{\pi}_{\text{old}} \leftarrow$ Backups$(\widehat{\mathbb{S}}, \widehat{\pi}_{\text{curr}}, \mathcal{E}, V, \mathsf{F}, \epsilon)$
  until $\mathsf{F} = \emptyset$ *and* $\widehat{\pi}_{old} = \widehat{\pi}_{curr}$ *and* RES $\leq \epsilon$
  return V

Function Expand-Fringes$(\mathbb{S}, \widehat{\mathbb{S}}, \widehat{\pi}_{curr}, \mathcal{E})$
  $\mathsf{F} \leftarrow \mathcal{E} \cap (\widehat{\mathsf{G}} \setminus \mathsf{G})$
  for $s_f \in \mathsf{F}$ **do**
    $\widehat{\mathbb{S}} \leftarrow$ Add-Actions$(\mathbb{S}, \widehat{\mathbb{S}}, s_f, \mathsf{A}(s_f))$
    $\widehat{\pi}_{\text{curr}}(s_f) \leftarrow \text{argmin}_{a \in \widehat{\mathsf{A}}(s)} Q(s, a)$
  return $\widehat{\mathbb{S}}$, $\widehat{\pi}_{\text{curr}}$

Function Add-Actions$(\mathbb{S}, \widehat{\mathbb{S}}, s, \mathsf{A}')$
  $\widehat{\mathsf{G}} \leftarrow \widehat{\mathsf{G}} \setminus \{s_f\}$
  $\widehat{\mathsf{A}}(s) \leftarrow \widehat{\mathsf{A}}(s) \cup \mathsf{A}'$
  for $a \in \mathsf{A}'$ **do**
    $\widehat{\mathsf{G}} \leftarrow \widehat{\mathsf{G}} \cup (\text{succ}(s, a) \setminus \widehat{\mathsf{S}})$
    $\widehat{\mathsf{S}} \leftarrow \widehat{\mathsf{S}} \cup \text{succ}(s, a)$
  return $\widehat{\mathbb{S}}$

Function Backups$(\widehat{\mathbb{S}}, \widehat{\pi}_{curr}, \mathcal{E}, V, \mathsf{F}, \epsilon)$
  $\widehat{\pi}_{\text{old}} \leftarrow \widehat{\pi}_{\text{curr}}$
  repeat
    RES $\leftarrow 0$
    for $s \in \mathcal{E} \setminus \widehat{\mathsf{G}}$ **do**
      $Q_{\min} \leftarrow \min_{a \in \widehat{\mathsf{A}}(s)} Q(s, a)$
      RES $\leftarrow \max(|V(s) - Q_{\min}|, \text{RES})$
      $V(s) \leftarrow Q_{\min}$
    $\widehat{\pi}_{\text{curr}} \leftarrow \widehat{\pi}_V$
  until $\mathsf{F} \neq \emptyset$ *or* $\widehat{\pi}_{curr} \neq \widehat{\pi}_{old}$ *or* RES $\leq \epsilon$
  return $V, \text{RES}, \widehat{\pi}_{\text{curr}}, \widehat{\pi}_{\text{old}}$

Definition 12 (Partial SSP)**.** Consider an SSP $\mathbb{S} = \langle \mathsf{S}, s_0, \mathsf{G}, \mathsf{A}, P, C \rangle$ and a heuristic H. A partial SSP for $\mathbb{S}$ with H is an SSP with terminal costs $\widehat{\mathbb{S}} = \langle \widehat{\mathsf{S}}, s_0, \widehat{\mathsf{G}}, \widehat{\mathsf{A}}, P, C, H \rangle$ where

- $\widehat{\mathsf{S}}$ and $\widehat{\mathsf{A}}$ are subsets of their counterparts in $\mathbb{S}$, i.e., $\widehat{\mathsf{S}} \subseteq \mathsf{S}$ and $\widehat{\mathsf{A}}(s) \subseteq \mathsf{A}(s)\ \forall s \in \widehat{\mathsf{S}}$,
- s_0 is identical to $\mathbb{S}$,
- $\widehat{\mathsf{G}}$ is a set of goals that satisfies $\widehat{\mathsf{G}} \subseteq \widehat{\mathsf{S}}, \mathsf{G} \cap \widehat{\mathsf{S}} \subseteq \widehat{\mathsf{G}}$,
- P and C are the same as $\mathbb{S}$ but restricted to $\widehat{\mathsf{S}}$ and $\widehat{\mathsf{A}}$ (we abuse notation and reuse the same symbols as in $\mathbb{S}$ to emphasise that these functions are otherwise unchanged), and
- H is the terminal cost.

In $\widehat{\mathbb{S}}$, whenever the agent reaches an artificial goal $\widehat{g} \in \widehat{\mathsf{G}}$, it incurs a cost of $H(\widehat{g})$ once, and then terminates.

The Bellman equations for partial SSPs are similar to (1), but the equations only consider the partial SSP's subset of states and actions, and the goals incur terminal costs:

$$V(s) = \begin{cases} H(s) & \text{if } s \in \widehat{\mathsf{G}} \\ \min_{a \in \widehat{\mathsf{A}}(s)} Q(s, a) & \text{if } s \in \widehat{\mathsf{S}} \setminus \widehat{\mathsf{G}}. \end{cases} \tag{2}$$

We assume that $s \in \widehat{\mathsf{S}}, a \in \widehat{\mathsf{A}}(s)$ implies that $\text{succ}(s,a) \subseteq \widehat{\mathsf{S}}$, and therefore $Q(s,a)$ is always defined in (2). For a partial SSP $\widehat{\mathbb{S}}$, we will use the following terminology

- $\widehat{\mathsf{G}} \setminus \mathsf{G}$ is called the set of *artificial goals*,
- $\widehat{\mathsf{S}} \setminus \widehat{\mathsf{G}}$ are called *internal states*,
- $\widehat{\mathsf{A}}$ are *internal actions*,
- $\widehat{\pi}_V$ is the greedy policy over V restricted to $\widehat{\mathsf{S}} \setminus \widehat{\mathsf{G}}$, the non-goal states of $\widehat{\mathbb{S}}$.

iLAO* (algorithm 1) is an iterative algorithm. In each iteration, it works towards two opposing objectives:

1. **Optimise:** iLAO* tries to find the optimal policy for $\widehat{\mathbb{S}}$ by working towards solving the partial SSP $\widehat{\mathbb{S}}$'s Bellman equations (2).
2. **Close:** iLAO* expands any artificial goals reachable by its candidate policy $\widehat{\pi}_{\text{curr}}$ (called *fringe states*) so that eventually $\widehat{\pi}_{\text{curr}}$ has no artificial goals and is closed w.r.t. the original SSP.

To achieve a closed candidate policy (objective 2), iLAO* performs a Depth-First Search (DFS) on $\widehat{\pi}_{\text{curr}}$ to find all fringe states (algorithm 1 line 6) and then expands them (algorithm 1 line 7). To expand fringe state s, iLAO* adds $\mathsf{A}(s)$ to the partial SSP, and any states reachable from s, i.e., $\bigcup_{a \in \mathsf{A}(s)} \text{succ}(s,a)$ are added to the partial SSP as new artificial goals, if they are not already in $\widehat{\mathbb{S}}$. Expanding the fringe states in each iteration ensures that the candidate policy $\widehat{\pi}_{\text{curr}}$ will eventually be closed w.r.t. s_0 on the original SSP.

Simultaneously, iLAO* works towards finding an optimal policy on its partial SSP (objective 1), by applying Bellman backups to all states in $\widehat{\pi}_{\text{curr}}$'s envelope $\mathcal{E}$ (algorithm 1 line 9). The aim is to make V ϵ-consistent in order to show that $\widehat{\pi}_{\text{curr}}$ is optimal (w.r.t. ϵ). There is a subtlety regarding how $\widehat{\pi}_{\text{curr}}$ approximates $\widehat{\pi}_V$: during Expand-Fringes and Backups an update to $\widehat{\pi}_{\text{curr}}$ can cause it to reach internal states from which $\widehat{\pi}_{\text{curr}}$ leads to unaccounted fringe states, and these fringes are not tracked in $\mathcal{E}$. iLAO* recomputes F in algorithm 1 line 8, immediately after Expand-Fringes, so there are no unaccounted fringe states there. Backups addresses the issue by recomputing $\widehat{\pi}_{\text{curr}} \leftarrow \widehat{\pi}_V$ after each pass of Bellman backups; recall that the issue only appears if $\widehat{\pi}_{\text{curr}}$ is updated so that it reaches unaccounted fringe states, so the main loop of Backups stops as soon as $\widehat{\pi}_{\text{curr}} \neq \widehat{\pi}_{\text{old}}$, and F will be recomputed in the main loop at algorithm 1 line 8. Thus, this implementation of iLAO* handles the issue by ensuring $\widehat{\pi}_V$'s fringe states are in fact tracked in F, and requires the policy to be recomputed in each loop of Backups (algorithm 1 line 33). Our algorithm CG-iLAO* deviates from this, as we explain in section 4.

The objectives of optimising and closing interfere with each other, in the sense that expanding fringe states can break V's ϵ-consistency and create more work for Backups; and applying backups to the policy envelope can make $\widehat{\pi}_{\text{curr}}$ change in a way that introduces new fringe states that need to be expanded. Arguably, this is what makes iLAO* so efficient, because it is able to abandon working towards objective 1 if it conflicts with objective 2 and avoid redundant work, and vice versa. Once iLAO* achieves both objectives, it has a value function V that is ϵ-consistent w.r.t. the original SSP, since $\widehat{\pi}_{\text{curr}}$ has no artificial goals and $\text{res} \leq \epsilon$. We require that iLAO* is initialised with an admissible heuristic, and since it only modifies V with Bellman backups it preserves the invariant $V \leq V^*$. Consequently, we can refer to theorem 2, and conclude that iLAO* is optimal.

3.1 iLAO* as Linear Program

Now, we present a new way to interpret iLAO* in terms of Linear Programs (LPs) and constraint and variable generation. As a starting point, it is well-known that SSPs can be solved by the LP presented in LP 1.

$$\begin{aligned}
\max_{\boldsymbol{\mathcal{V}}} \quad & V_{s_0} && && \text{(LP 1)}\\
\text{s.t.} \quad & V_s \leq C(s,a) + \sum_{s' \in \mathsf{S}} P(s'|s,a) V_{s'} && \forall s \in \mathsf{S} \setminus \mathsf{G}, a \in \mathsf{A}(s) && \text{(C1)}\\
& V_g = 0 && \forall g \in \mathsf{G} && \text{(C2)}
\end{aligned}$$

Each variable $V_s \in \boldsymbol{\mathcal{V}}$ can be interpreted as $V(s)$, and then the constraints C1 are requiring that $V(s) \leq \min_{a \in \mathsf{A}(s)} Q(s,a)$ for each $s \in \mathsf{S}$. To emphasise this connection we write, when unambiguous, $V(s)$

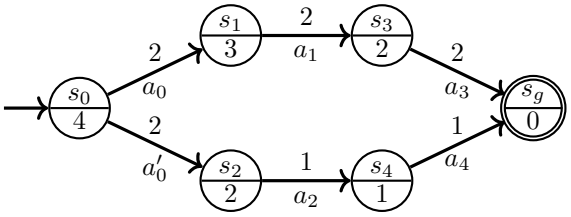


Figure 4: An SSP where LP 1's solution encodes an optimal policy, but does not give the optimal value function at s_1.

instead of V_s, and $Q(s,a)$ instead of the right-hand side of s and a's constraint C1. This LP encodes the Bellman equations (1): its constraints together with the maximisation objective search for $V(s)$ that is a maximal lower bound (infimum) on all $Q(s,a)$, which is a way to express the minimum for a set of values (provided the infimum is in the set), i.e., we are encoding $V(s) = \min_{a \in \mathsf{A}(s)} Q(s,a)$. Interestingly, the Bellman equations only need to be satisfied for states in the optimal policy envelope $\mathsf{S}^{\pi *}$. In other words, given an optimal LP solution $\boldsymbol{\mathcal{V}}^*$ that induces the policy π^*, the LP's constraints are active (tight) for the pairs $\big(s, \pi^*(s)\big)$ for all $s \in \mathsf{S}^{\pi *}$ and may be inactive (slack) everywhere else. We give an example of this in figure 4, where an optimal solution $\boldsymbol{\mathcal{V}}^*$ is shown in the bottom of all nodes; this solution encodes the optimal policy π^* with $\pi^*(s_0) = a'_0$, but does not give the optimal cost-to-go for s_1 (it should be 4).

For an alternative interpretation of LP 1, observe that its constraints are requiring $\boldsymbol{\mathcal{V}}$ to be a monotonic value function. Monotonicity implies admissibility, so LP 1 is looking for an admissible value function that maximises the value at s_0, i.e., V^*. Again, the objective only pushes against the constraints for states and actions encountered by π^*.

Similar to VI, LP 1 can be initialised with any vector. An LP solver will make the vector feasible and then improve its quality until it is optimal. LP 1's constraints require $\boldsymbol{\mathcal{V}}$ to be a monotonic value function, so an admissible value function may not describe a feasible solution, e.g., the following admissible heuristic is not monotonic and therefore infeasible:

$$H(s) = \begin{cases} V^*(s) & \text{if } s = s_0 \\ 0 & \text{if } s \neq s_0. \end{cases}$$

So, given a non-monotonic value function, the LP solver will first make it monotonic, and then work towards making the value function ϵ-consistent w.r.t. s_0.

We now present our key insight that iLAO* and its incremental growing of partial SSPs can be reinterpreted as an algorithm that solves LP 1 with *variable and constraint generation* [Bertsimas and Tsitsiklis, 1997].[5]

Variable generation enables the solving of LPs with prohibitively many variables. It works by considering a small LP with a subset of variables, and iteratively adding variables that are needed for the small LP to have the same solution as the original. The original LP is called the Master Problem (MP), and the smaller LP with a subset of MP's variables is called the Reduced Master Problem (RMP). An optimal solution for the RMP must also be feasible for the MP, but, assuming it is a maximisation problem, the RMP's solution may have a lower objective than the MP's optimal solution. Variable generation iteratively adds variables to the RMP that have the potential to increase its objective, so that eventually the RMP's optimal solution is also optimal for the MP. When solving an LP directly (with the simplex algorithm), the solver explicitly enumerates all unused (non-basic) variables and evaluates whether using it (making it basic) can improve the objective. Variable generation also does this, but instead of enumerating all variables, it determines what variables to add by solving a *pricing problem*. Thus, variable generation is guaranteed to eventually terminate such that its RMP's optimal solution is also optimal for the MP. Variable generation is often more efficient than solving the MP directly because the RMP often induces the optimal solution with significantly fewer variables than the original problem. Note the similarity to heuristic search, where the problem can be solved by considering only a small subset of states.

Constraint generation is similar to variable generation, but solves LPs with excessively many constraints rather than variables. It analogously works by considering small LPs with a subset of constraints, and adding constraints as needed to obtain an optimal solution for the MP. The smaller LPs have fewer constraints and thereby relax the MP, so they are called *relaxed LPs*. For a maximisation problem, the

[5] Variable and constraint generation are also known column generation and the cutting plane method, respectively.

solution to a relaxed LP must have its objective greater or equal to the MP's optimal objective, but, since some constraints are missing, the relaxed solution may not be feasible for the MP. Constraint generation uses a *separation oracle* to detect which of the MP's constraints are violated by the relaxed solution, and adds these to relaxed LP, so that its next solution is closer to being feasible for MP. Once the separation oracle finds no violations and the relaxed solution is feasible for the MP, the solution is immediately optimal for the MP.

Simultaneous variable and constraint generation operates over LPs that are simultaneously RMPs with a subset of the MP's variables, and relaxations with a subset of the MP's variables. Respectively, this requires a pricing problem to determine which variables to add, as well as a separation oracle that detects which of the MP's constraints are violated. Solutions for intermediate LPs can be difficult to interpret, because they need not give lower bounds on the optimal solution (since variables may be missing) and they need not give upper bounds (because constraints may be missing). However, as soon as we have a solution where the pricing problem does not flag any new variables and the separation oracles does not detect any violations, we can be sure that the solution is optimal for the MP. To justify this fact, consider a MP with variables X and constraints $\mathcal{C}$, and a relaxed RMP with variables $\widehat{X} \subseteq X$ and constraints $\widehat{\mathcal{C}}$ with an optimal solution $\widehat{\boldsymbol{x}}$. If the pricing problem does not add any variables to the relaxed RMP, then we know that $\widehat{\boldsymbol{x}}$ is an optimal solution for the LP with all variables X but the subset of constraints $\widehat{\mathcal{C}} \subseteq \mathcal{C}$. If $\widehat{\boldsymbol{x}}$ additionally does not violate any constraints in $\mathcal{C}$, then it must immediately be an optimal solution for the MP.

We can interpret iLAO* as a variable and constraint generation algorithm over LP 1. Its partial SSP induces LP 2 which is simultaneously an RMP and a relaxation of for LP 1, because it has the subset of variables $\{V_s : s \in \widehat{\mathsf{S}} \setminus \widehat{\mathsf{G}}\}$ and only requires monotonicity constraints for actions in $\widehat{\mathsf{A}}$ (C3). Recall that iLAO*'s partial SSPs are SSPs with terminal costs of H at $\widehat{\mathsf{G}}$ (definition 12), so we include the constraints C4 for artificial goals, in order to capture the terminal costs. Observe that V_g are constants for $g \in \widehat{\mathsf{G}}$, similarly to LP 1's $V_g = 0$ for $g \in \mathsf{G}$. Strictly, due to these artificial goal constraints, LP 2 is not a true relaxation of LP 1, since these constraints do not appear in LP 1. However, it can still be considered a relaxation because these constraints only affect the "boundary condition" of constants, and since we use an admissible heuristic H, we know that an optimal solution for LP 2 remains a lower bound an optimal solution for LP 1.

$$\begin{aligned}
\max_{\boldsymbol{V}} \quad & V_{s_0} && && \text{(LP 2)}\\
\text{s.t.} \quad & V_s \leq C(s,a) + \sum_{s' \in \mathsf{S}} P(s'|s,a) V_{s'} && \forall s \in \widehat{\mathsf{S}} \setminus \widehat{\mathsf{G}}, a \in \widehat{\mathsf{A}}(s) && \text{(C3)}\\
& V_g = H(g) && \forall g \in \widehat{\mathsf{G}} && \text{(C4)}
\end{aligned}$$

In terms of LP 2, iLAO* generates new variables for its RMP (partial SSP) by expanding $\widehat{\pi}_{\text{curr}}$'s fringe states s_f, and inserting them as variables V_{s_f}, replacing the previous constant $V_{s_f} = H(s_f)$. Note that this procedure is guided by the heuristic H in the sense that $\widehat{\pi}_{\text{curr}}$ takes H into account. The separation oracle for iLAO*'s constraint generation, i.e., the mechanism for detecting constraints in the MP that are violated, is very simple: whenever a variable V_s is added, all constraints associated with s are immediately added. Adding all constraints trivially ensures that the relaxed LP will not violate any of the MP's constraints, but iLAO* does not get any savings from leaving out non-violated constraints.

We illustrate iLAO*'s variable and constraint generation with a small example. Consider the SSP in figure 5 with the heuristic values shown in the bottom of each node. Then, iLAO*'s iterations will look like this:

Iter. 1 $\widehat{\mathsf{S}} = \{s_0\}, \widehat{\mathsf{G}} = \{s_0\}$ — $\widehat{\pi}_{\text{curr}} = \{\}$ — expand s_0;

Iter. 2 $\widehat{\mathsf{S}} = \{s_0, s_1, s_2, s_3\}, \widehat{\mathsf{G}} = \{s_1, s_2, s_3\}$ — $\widehat{\pi}_{\text{curr}} = \{s_0 \mapsto a_0'\}$ — expand s_2;

Iter. 3 $\widehat{\mathsf{S}} = \{s_0, s_1, s_2, s_3, s_g\}, \widehat{\mathsf{G}} = \{s_1, s_3, s_g\}$ — $\widehat{\pi}_{\text{curr}} = \{s_0 \mapsto a_0\}$ — expand s_1;

Iter. 4 $\widehat{\mathsf{S}} = \{s_0, s_1, s_2, s_3, s_g\}, \widehat{\mathsf{G}} = \{s_3, s_g\}$ — $\widehat{\pi}_{\text{curr}} = \{s_0 \mapsto a_0, s_1 \mapsto a_1, s_2 \mapsto a_2\}$ — done.

The LPs for iLAO*'s partial SSPs at the starts of iterations 2–4 are shown in figure 6. For each partial SSP $\widehat{\mathbb{S}}$, its LP has variables for each internal state $s \in \widehat{\mathsf{S}} \setminus \widehat{\mathsf{G}}$, and the goals $\widehat{\mathsf{G}}$ correspond to constants, e.g., $V_{s_g} = 0$. Thus, when iLAO* expands a state, we generate the corresponding variable for the LP, as well as the constraints for its outgoing actions. Note that the constants for $s \in \widehat{\mathsf{G}}$ disappear when s gets expanded, and they are replaced with variables.

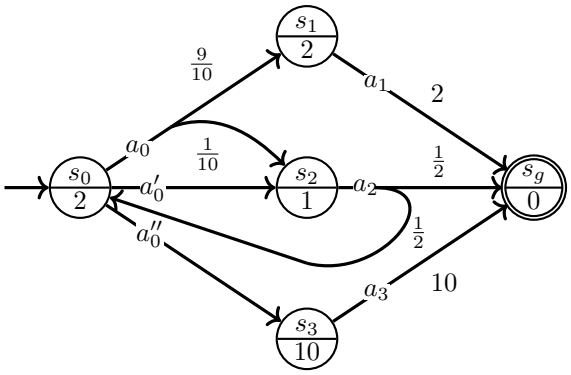


Figure 5: Example SSP that we solve with iLAO* under the lens of constraint and variable generation. $H(s)$ are given in the bottom of each node.

Iter. 2	Iter. 3	Iter. 4
$\max_{V_{s_0}} V_{s_0}$ s.t.	$\max_{V_{s_0}, V_{s_2}} V_{s_0}$ s.t.	$\max_{V_{s_0}, V_{s_1}, V_{s_2}} V_{s_0}$ s.t.
$V_{s_0} \leq Q(s_0, a_0)$	$V_{s_0} \leq Q(s_0, a_0)$	$V_{s_0} \leq Q(s_0, a_0)$
$V_{s_0} \leq Q(s_0, a'_0)$	$V_{s_0} \leq Q(s_0, a'_0)$	$V_{s_0} \leq Q(s_0, a'_0)$
$V_{s_0} \leq Q(s_0, a''_0)$	$V_{s_0} \leq Q(s_0, a''_0)$	$V_{s_0} \leq Q(s_0, a''_0)$
		$V_{s_1} \leq Q(s_1, a_1)$
	$V_{s_2} \leq Q(s_2, a_2)$	$V_{s_2} \leq Q(s_2, a_2)$
$V_{s_1} = H(s_1) = 2$	$V_{s_1} = H(s_1) = 2$	
$V_{s_2} = H(s_2) = 1$		
$V_{s_3} = H(s_3) = 10$	$V_{s_3} = H(s_3) = 10$	$V_{s_3} = H(s_3) = 10$
	$V_{s_g} = 0$	$V_{s_g} = 0$

Figure 6: RMPs corresponding to iLAO*'s partial SSPs at the start of iterations 2–4 from the example in figure 5.

We point out that the solution to the final partial SSP has

- $V_{s_0} = 3.05$
- $V_{s_1} = 2.0$
- $V_{s_2} = 2.52$.

Importantly, notice that the constraint $V_{s_0} \leq Q(s_0, a''_0)$ is trivially satisfied because $3.05 \leq 11$; i.e., the constraint $V_{s_0} \leq Q(s_0, a''_0)$ is not needed, and corresponds to an unneeded action that could be left out of the partial SSPs because $H(s_3)$ is large and deems s_3 unpromising. This is a consequence of iLAO*'s naïve separation oracle that adds constraints for all outgoing actions as states get expanded.

This interpretation extends the currently understood connections between planning algorithms and operations research, and highlights the shortcoming of Bellman backups that consider all actions because these correspond to a naïve separation oracle that adds all constraints. In the next section, we address this shortcoming, and introduce an efficient separation oracle that only adds violated constraints.

4 CG-iLAO*

All previous algorithms based on VI update a state's value function $V(s)$ by considering all applicable actions actions. In iLAO*, this happens because all its states are either unexpanded and have no available actions $\widehat{\mathsf{A}}(s) = \emptyset$, or fully expanded with all applicable actions available $\widehat{\mathsf{A}}(s) = \mathsf{A}(s)$; iLAO* is not able to partially expand states. We address this shortcoming by allowing states to be partially expanded so

Algorithm 2: CG-iLAO*

Function *CG-iLAO** $(\mathbb{S}, H, \epsilon, \eta)$
  $\widehat{\mathbb{S}} \leftarrow$ partial SSP $\langle \{s_0\}, s_0, \{s_0\}, \emptyset, P, C, H \rangle$
  $V \leftarrow$ value function initialised by H
  $\widehat{\pi}_{\text{curr}} \leftarrow$ candidate policy initialised as undefined everywhere (used to approximate $\widehat{\pi}_V$)
  $\Gamma \leftarrow \emptyset$
  repeat
    $\mathcal{E} \leftarrow$ post-order DFS traversal of $\widehat{\pi}_{\text{curr}}$ from s_0
    $\widehat{\mathbb{S}}, \widehat{\pi}_{\text{curr}}, \mathcal{E} \leftarrow$ PARTLY-EXPAND-FRINGES$(\mathbb{S}, \widehat{\mathbb{S}}, \widehat{\pi}_{\text{curr}}, \mathcal{E}, V)$
    $\mathsf{F} \leftarrow \mathsf{S}^{\widehat{\pi}_{\text{curr}}} \cap (\widehat{\mathsf{G}} \setminus \mathsf{G})$
    $V, \text{RES}, \widehat{\pi}_{\text{curr}}, \widehat{\pi}_{\text{old}}, \Gamma \leftarrow$ CG-BACKUPS$(\mathbb{S}, \widehat{\mathbb{S}}, \widehat{\pi}_{\text{curr}}, \mathcal{E}, V, \mathsf{F}, \Gamma, \epsilon, \eta)$
    $V, \text{RES}, \Gamma, \widehat{\mathbb{S}} \leftarrow$ FIX-CONSTRS$(\mathbb{S}, \widehat{\mathbb{S}}, \widehat{\pi}_{\text{curr}}, V, \Gamma, \text{RES}, \epsilon)$
  until $\mathsf{F} = \emptyset$ *and* $\widehat{\pi}_{old} = \widehat{\pi}_{curr}$ *and* RES $\leq \epsilon$
  return V

Function PARTLY-EXPAND-FRINGES$(\mathbb{S}, \widehat{\mathbb{S}}, \widehat{\pi}_{curr}, \mathcal{E}, V)$
  while $\exists s_f \in \mathcal{E} \cap (\widehat{\mathsf{G}} \setminus \mathsf{G})$ **do**
    $\mathsf{A}' \leftarrow \{a \in \mathsf{A}(s_f) : Q(s_f, a) = \min_{a' \in \mathsf{A}(s_f)} Q(s_f, a')\}$ `/* Select greedy actions argmin_{a∈A} Q(s,a) */`
    $\widehat{\mathbb{S}} \leftarrow$ ADD-ACTIONS$(\mathbb{S}, \widehat{\mathbb{S}}, s_f, \mathsf{A}')$ `/* Note: s_f ∉ Ĝ after ADD-ACTIONS */`
    $\widehat{\pi}_{\text{curr}}(s_f) \leftarrow$ some greedy action $a \in \widehat{\mathsf{A}}(s)$
    for $s_{int} \in succ(s_f, \widehat{\pi}_{curr}(s_f)) \setminus \widehat{\mathsf{G}}$ **do**
      $\mathcal{E} \leftarrow \mathcal{E}$ updated with post-order DFS traversal of $\widehat{\pi}_{\text{curr}}$ from s_{int} (no duplicate states)
  return $\widehat{\mathbb{S}}$, $\widehat{\pi}_{\text{curr}}$, $\mathcal{E}$

Function CG-BACKUPS$(\mathbb{S}, \widehat{\mathbb{S}}, \widehat{\pi}_{curr}, \mathcal{E}, V, \mathsf{F}, \Gamma, \epsilon, \eta)$
  $\widehat{\pi}_{\text{old}} \leftarrow \widehat{\pi}_{\text{curr}}$
  repeat
    RES $\leftarrow 0$
    for $s \in \mathcal{E} \setminus \widehat{\mathsf{G}}$ **do**
      $Q_{\min} \leftarrow \min_{a \in \widehat{\mathsf{A}}(s)} Q(s, a)$
      if $Q_{\min} - V(s) > \eta$ **then**
        $\Gamma \leftarrow \Gamma \cup$ EXT-SUCCS$(s, \mathbb{S}, \widehat{\mathbb{S}})$
      else if $V(s) - Q_{\min} > \eta$ **then**
        $\Gamma \leftarrow \Gamma \cup$ PREDS$(s, \mathbb{S}, \widehat{\mathbb{S}})$
      RES $\leftarrow \max(|V(s) - Q_{\min}|, \text{RES})$
      $V(s) \leftarrow Q_{\min}$
      $\widehat{\pi}_{\text{curr}}(s) \leftarrow$ some greedy action $a \in \widehat{\mathsf{A}}(s)$ s.t. $Q(s, a) = Q_{\min}$
  until $\mathsf{F} \neq \emptyset$ *or* $\widehat{\pi}_{curr} \neq \widehat{\pi}_{old}$ *or* RES $\leq \epsilon$
  return $V, \text{RES}, \widehat{\pi}_{\text{curr}}, \widehat{\pi}_{\text{old}}, \Gamma$

Function FIX-CONSTRS$(\mathbb{S}, \widehat{\mathbb{S}}, \widehat{\pi}_{curr}, V, \Gamma, \text{RES}, \epsilon)$
  $\Gamma' \leftarrow \emptyset$
  for $(s, a) \in \Gamma$ *s.t.* $V(s) > Q(s, a) + \epsilon$ **do**
    if $a \notin \widehat{\mathsf{A}}(s)$ **then**
      $\widehat{\mathsf{A}}(s) \leftarrow \widehat{\mathsf{A}}(s) \cup \{a\}$
      $\widehat{\mathsf{G}} \leftarrow \widehat{\mathsf{G}} \cup (\text{succ}(s, a) \setminus \widehat{\mathsf{S}})$
      $\widehat{\mathsf{S}} \leftarrow \widehat{\mathsf{S}} \cup \text{succ}(s, a)$
    RES $\leftarrow \max(V(s) - Q(s, a), \text{RES})$
    $V(s) \leftarrow Q(s, a)$
    $\widehat{\pi}_{\text{curr}}(s) \leftarrow a$
    $\Gamma' \leftarrow \Gamma' \cup$ PREDS$(s, \mathbb{S}, \widehat{\mathbb{S}})$
  return $V, \text{RES}, \Gamma', \widehat{\mathbb{S}}$

that backups in the partial SSP only consider a subset of $\mathsf{A}(s)$. To this end, we first define which actions can be left out of the partial SSP without affecting optimality.

Definition 13 (Inactive Action)**.** Consider an SSP $\mathbb{S}$, its partial SSP $\widehat{\mathbb{S}}$, a value function V, and a state $s \in \widehat{\mathsf{S}}$. An action $a \in \mathsf{A}(s) \setminus \widehat{\mathsf{A}}(s)$ is inactive in state s if $\min_{\widehat{a} \in \widehat{\mathsf{A}}(s)} Q(s, \widehat{a}) < Q(s, a)$.

If an action a is inactive for s, then $Q(s, a)$ can not reduce $V(s)$, and we can still obtain V^* if a is ignored by our Bellman backups. Of course a may become active later, but while it is inactive, it can be safely left out of the partial SSP. To understand inactive actions in terms of linear programming, consider the LP 2 associated with $\widehat{\mathbb{S}}$ and let V be a solution for it. Each state-action pair (s, a) corresponds to the consistency constraint C3. An inactive action a for state s does not have an associated constraint in the LP since $a \notin \widehat{\mathsf{A}}(s)$, and importantly, if the constraint were added it would be inactive (also called loose). Therefore, adding the constraint does not affect the solution and only introduces redundant work.

To exploit this insight, we generalise iLAO* so it can ignore inactive actions in each of its partial SSPs and partially expand states, i.e., states may have a non-empty strict subset of applicable actions available in the partial SSP, $\emptyset \neq \widehat{\mathsf{A}}(s) \subsetneq \mathsf{A}(s)$. Then, we introduce a mechanism that efficiently adds actions that are not inactive as they appear. In terms of linear programming, we are using the constraint generation framework to leave unneeded constraints out of the LP, and use a separation oracle to identify constraints that may be needed to encode the optimal solution. This yields a new dynamic programming algorithm, which, in honour of this tight connection to constraint generation, we call Constraint-Generation iLAO* (CG-iLAO*).

CG-iLAO* (algorithm 2) is similar to iLAO* (algorithm 1), but with the key difference that in its expansion phase (algorithm 2 line 8), CG-iLAO* calls Partly-Expand-Fringes and expands a state with only the greedy actions w.r.t. V, and not all applicable actions. This introduces two issues that can affect CG-iLAO*'s correctness.

1. Partial expansion may ignore an optimal action if V is inaccurate. This is clearly problematic because CG-iLAO* can not possibly find the optimal policy if its partial SSP does not contain the optimal actions. So, CG-iLAO* requires a mechanism that can detect and add optimal actions if the relevant state's partial expansion missed them.

2. V is evaluated w.r.t. the partial SSP, so if an optimal action a is missing, then it is not taken into account by V (this can cause $V \geq V^*$). If the missing action is later added, then V does not automatically reflect that a has been made available, so we must ensure that a's improvements to V are fully propagated.

Elegantly, it turns out that both of these problems are instances of constraint violation, and can therefore be addressed by detecting constraint violations with a separation oracle and then enforcing that the constraints become satisfied.

How can we find constraint violations efficiently? A naïve separation oracle considers all constraints missing in the partial SSP, i.e., the constraints for $s \in \widehat{\mathsf{S}}, a \in \mathsf{A}(s) \setminus \widehat{\mathsf{A}}(s)$, and evaluates whether they are violated by checking if $V(s) > Q(s, a)$. This means we compute Q-values for each external action and therefore save no work compared to adding them to the partial SSP in the first place. The key realisation that lets us save Q-value computations is that non-violated constraints remain non-violated unless V is updated in a particular way. Our efficient separation oracle exploits this by computing a set of constraints that might be violated, and then checking violations only in this set. Suppose there are currently no constraint violations and $V(s)$ is assigned $\min_{a \in \widehat{\mathsf{A}}(s)} Q(s, a)$. Then, the separation oracle determines that the constraints associated with s may become violated according to the following three cases:

1. ***V*(*s*) stays the same.** There can be no new constraint violations.

2. ***V*(*s*) increases.** The constraints $V(s) \leq Q(s, a')$ may be violated for $(s, a') \in \textsc{Ext-Succs}(s, \mathbb{S}, \widehat{\mathbb{S}})$ where
$$\textsc{Ext-Succs}(s, \mathbb{S}, \widehat{\mathbb{S}}) \stackrel{\text{def}}{=} \left\{ (s, a) : a \in \mathsf{A}(s) \setminus \widehat{\mathsf{A}}(s) \right\}.$$
Note that we only have to consider external successor actions, and not internal ones (s, a') with $a' \in \widehat{\mathsf{A}}(s)$, because $V(s) \leftarrow \min_{a \in \widehat{\mathsf{A}}(s)} Q(s, a) \leq Q(s, a')$, and therefore the constraint associated with (s, a') must be satisfied already.

3. **$V(s)$ decreases.** The only constraints that may be violated are $V(s') \leq Q(s', a')$ for $(s', a') \in \textsc{Preds}(s, \mathbb{S}, \widehat{\mathbb{S}})$ where

$$\textsc{Preds}(s, \mathbb{S}, \widehat{\mathbb{S}}) \stackrel{\text{def}}{=} \left\{ (s', a') : s' \in \widehat{\mathsf{S}}, a' \in \mathsf{A}(s'), s \in \text{succ}(s', a') \right\}.$$

   In this case, we must consider internal actions as well as external ones. However, we do not consider external predecessor states $s' \in \mathsf{S} \setminus \widehat{\mathsf{S}}$, because such states will eventually be expanded and handled that way if their constraint violations persist.

In the pseudocode, these checks are performed whenever V is updated, and all potential violations are stored in Γ:

- if $V(s)$ increases then $\Gamma \leftarrow \Gamma \cup \textsc{Ext-Succs}(s, \mathbb{S}, \widehat{\mathbb{S}})$ (algorithm 2 line 29);
- if $V(s)$ decreases, then $\Gamma \leftarrow \Gamma \cup \textsc{Preds}(s, \mathbb{S}, \widehat{\mathbb{S}})$ (algorithm 2 line 31 and line 47).

At the end of each iteration of its main loop, CG-iLAO* calls Fix-Constrs to check all potentially violated constraints Γ. If a constraint in Γ is not violated, it is simply discarded. If a constraint (s, a) is indeed violated, then Fix-Constrs fixes it by setting $V(s) \leftarrow Q(s, a)$ (algorithm 2 line 45), and then removes (s, a) from Γ. Importantly, this update to V may create new constraint violations, so we must apply the same rules as before and track any new potential constraint violations in Γ for the next iteration. This ensures that Γ is always a superset of CG-iLAO*'s constraint violations, and the algorithm only terminates once they have all been addressed.

A subtle technical difference that follows from this construction is that CG-iLAO* requires the additional error parameter $\eta \in \mathbb{R}_{>0}$ that is used to decide whether a state's value function has changed sufficiently to warrant updating Γ. For now, it is convenient to use $\eta = \epsilon$, and we explain the effects of this parameter in page 20.

There are also two subtle differences in the implementations of iLAO* and CG-iLAO* as presented in algorithm 1 and algorithm 2, which will turn out to be significant in our experiments (section 7). These changes are orthogonal to CG-iLAO*'s partial expansion mechanism, so it is possible to define four similar algorithms: (i) full expansion with implementation 1, (ii) full expansion with implementation 2, (iii) partial expansion with implementation 1, (iv) partial expansion with implementation 2. In previous work we only considered (i) and (iv) [Schmalz and Trevizan, 2024], which is problematic for comparing the effect of using partial expansions versus full expansions. We address this by comparing (ii) with (iv), and we also include (i) as a way to compare to the previous work. The two subtle implementation differences are:

1. Partly-Expand-Fringes deviates from Expand-Fringes in the case where the expansion adds an internal state $s \in \widehat{\mathsf{S}}$ where $\widehat{\pi}_{\text{curr}}$ is defined: iLAO* stops there and does not update $\mathcal{E}$; CG-iLAO* follows $\widehat{\pi}_{\text{curr}}$ from s, updating $\mathcal{E}$, and expands any new fringe states that are encountered this way. Our pseudocode does this by updating $\mathcal{E}$ during the `while` loop, and in practice it is implemented with recursion. iLAO* (algorithm 1)'s implementation follows Hansen and Zilberstein [2001], whereas CG-iLAO* (algorithm 2)'s implementation follows the recursive definition from Hansen and Abdoulahi [2016]. Thus, per call, CG-iLAO*'s Partly-Expand-Fringes can potentially expand more states than iLAO*'s Expand-Fringes if it keeps expanding states that lead to internal states where $\widehat{\pi}_{\text{curr}}$ is defined.

2. The second change concerns how $\widehat{\pi}_{\text{curr}}$ is tracked, and how to handle its deviation from $\widehat{\pi}_V$. Recall from section 3, that iLAO*'s Expand-Fringes or Backups may encounter an internal state whose policy leads to a fringe state that is not accounted for in $\mathcal{E}$. There, it is handled by making sure $\widehat{\pi}_{\text{curr}} = \widehat{\pi}_V$ during Backups. CG-iLAO* embraces the property that $\widehat{\pi}_{\text{curr}}$ approximates $\widehat{\pi}_V$, and only updates $\widehat{\pi}_{\text{curr}}$ at states s where $V(s)$ changes (algorithm 2 line 34 and line 46). This is more efficient, but can have the effect that the inaccurate $\widehat{\pi}_{\text{curr}}$ has no fringe states, even though $\widehat{\pi}_V$ does. This is not an issue because in such situations $\widehat{\pi}_{\text{curr}}$ must be updated at some point, setting $\widehat{\pi}_{\text{curr}} \neq \widehat{\pi}_{\text{old}}$, and thereby ensuring that CG-iLAO* does not terminate until it has indeed closed all fringes. Thus, CG-iLAO* saves the overhead of computing $\widehat{\pi}_{\text{curr}}$ explicitly, but may require additional iterations to discover that its greedy policy still contains fringes.

We will see in our experiments that these implementation changes can have a significant impact on performance.

In summary, CG-iLAO* generalises iLAO* by allowing actions to be left out of its partial SSPs. CG-iLAO* ignores inactive actions and only adds actions as they are required. Under the lens of linear programming, the inactive actions correspond to inactive constraints, and CG-iLAO* refines iLAO* by making the separation oracle more selective, so that only constraints that are actively violated are added to the partial SSP. We note that CG-iLAO*'s separation oracle incurs an overhead of Q-value computations, since it must determine whether previously non-violated constraints have become violated. In our experiments (section 7), we show that the Q-value savings from ignoring expensive actions outweigh the separation oracle's Q-value overhead, and CG-iLAO* reliably computes fewer Q-values than the state-of-the-art. These savings let CG-iLAO* outperform the state-of-the-art on our benchmarks. CG-iLAO* represents an important step for heuristic search, because it enables a heuristic to guide our algorithm away from expensive actions as well as expensive states, letting the algorithm save on Q-value computations for those actions.

4.1 Properties of CG-iLAO* and Proof of Correctness

In this section, we show some properties of CG-iLAO* that distinguish it from previous approaches. Then, we prove that CG-iLAO* is correct.

CG-iLAO* is not monotonic In all previous algorithms based on Bellman backups, such as LRTDP and iLAO*, if they are initialised with a monotonic heuristic, their backups are guaranteed to be monotonically non-decreasing, i.e., $V^{t+1} \geq V^t$ for all timesteps t. CG-iLAO* does not have this guarantee because it ignores inactive actions and may evaluate a suboptimal policy before recognising and inserting the missing optimal action, which decreases the value function. As an example of this, consider the SSP in figure 7 where $H(s)$ is given in the bottom of the respective node. H is monotonic, but as we step through the iterations of CG-iLAO*, a cost decrease occurs:

Iter. 1 Expands s_0.

Iter. 2 Partly expands s_1 with a_1.

Iter. 3 Expands s_2 with a_2 and, after CG-Backups, we have $V(s_2) = 3$, $V(s_1) = 4$, $V(s_0) = 5$ and $\Gamma = \{(s_1, a'_1)\}$. Since $\Gamma \neq \emptyset$, Fix-Constrs verifies that (s_1, a'_1) is currently better than the existing action a_1 for s_1, so a'_1 is added to $\widehat{\mathsf{A}}(s_1)$ and $V(s_1)$ is changed from 4 to 3. Recall that $V(s_0) = 5$, so when $V(s_1)$ is updated to 3, $\{s_0, a_0\}$ is inserted into Γ, and no further changes are made in this iteration.

Iter. 4 Expands s_3 and CG-Backups reduces $V(s_0)$ from 5 to 4, so V has been decreased by CG-Backups.

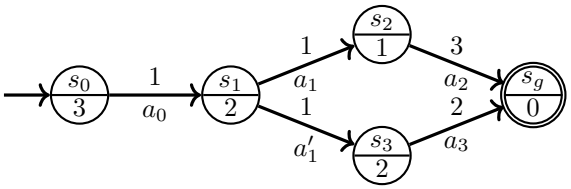


Figure 7: An SSP equipped with a monotonic heuristic where CG-iLAO*'s value function is not monotonically non-decreasing.

CG-iLAO* is ϵ-consistent on $\widehat{\mathbb{S}}$, not $\mathbb{S}$ Recall the definition of ϵ-consistency as given in definition 7. For Find-and-Revise algorithms, the output V is ϵ-consistent w.r.t. the original SSP. This is not the case for CG-iLAO*: it guarantees ϵ-consistency over its partial SSP by construction, but may produce a solution that is not ϵ-consistent w.r.t. $\mathbb{S}$. This is not an issue and does not prevent CG-iLAO* from finding optimal solutions, as we explain later, but it is important to understand. Consider CG-iLAO*'s steps on the pathological SSP in figure 8:

Iter. 1 Expands s_0.

Iter. 2 Expands s_3 and partly expands s_1 with a'_1 since $Q(s_1, a'_1) = 4$ and $Q(s_1, a_1) = 6$. After CG-Backups, $V(s_3) = 6, V(s_1) = 10, V(s_0) = 9$. Importantly, $V(s_1)$ did not increase, so its successors are not added to Γ.

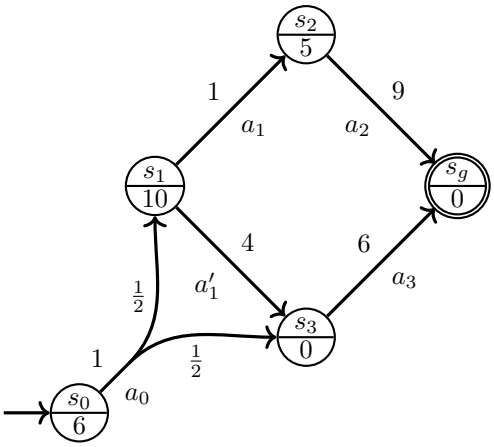


Figure 8: An SSP where CG-iLAO* does not return an ϵ-consistent w.r.t. the whole SSP.

Iter. 3 There are no fringes left to expand, so CG-iLAO* does one pass of BACKUPS which leaves V unchanged and exits with RES $= 0$ and $\Gamma = \emptyset$. The algorithm terminates.

The algorithm has terminated, but $V(s_1) - Q(s_1, a_1) = 4 > \epsilon$, so V is not ϵ-consistent. Nevertheless, the algorithm has found an optimal policy. The upshot is that we must be careful and specify the SSP for which the algorithm is ϵ-consistent. This is analogous to FIND-AND-REVISE algorithms not being globally ϵ-consistent, and requiring that the greedy policy is specified.

Thus, CG-iLAO* has different behaviours to previous algorithms based on Bellman backups. Consequently, we can not immediately apply previous proof techniques, and we must be careful with our definition of optimality. We take a moment to formalise the notion of an optimal algorithm.

Definition 14 (Optimal Algorithm)**.** An algorithm parameterised by the error term $\epsilon \in \mathbb{R}_{>0}$ is called optimal if, as $\epsilon \to 0$, it outputs an optimal policy.

Observe that this definition was used implicitly in the background (section 2), and we are just making it explicit now. Theorem 1 and theorem 2 demonstrate that VI and FIND-AND-REVISE algorithms are optimal, respectively.

It turns out that ϵ-consistency over partial SSPs $\widehat{\mathbb{S}}$ is sufficient to prove optimality over the original SSP $\mathbb{S}$, provided that V is admissible w.r.t. $\mathbb{S}$, and the greedy policy $\widehat{\pi}_V$ does not terminate in one of $\widehat{\mathbb{S}}$'s artificial goal states $\widehat{\mathsf{G}} \setminus \mathsf{G}$. The first condition is inherited from regular ϵ-consistency, and the second condition ensures that the greedy policy $\widehat{\pi}_V$ induces a closed policy for $\mathbb{S}$. To formalise and prove this claim, we start by introducing the following upper bound over the expected number of steps to reach the goal.

Definition 15. Let $\mathbb{S}_{\mathsf{G}'}$ denote an SSP that is identical to $\mathbb{S}$ but has different goal states $\mathsf{G}' \subseteq \mathsf{S}$, and let $N(s, \pi, \mathbb{S}_{\mathsf{G}'})$ denote the expected number of steps to reach the goals G' by following the proper policy π from s. Then, $\overline{N}(s, \mathbb{S})$ is the smallest number that satisfies $\overline{N}(s, \mathbb{S}) \geq N(s, \pi, \mathbb{S}_{\mathsf{G}'})$ for all $\mathbb{S}_{\mathsf{G}'}$ and all policies π that are proper for $\mathbb{S}_{\mathsf{G}'}$ from s, omitting any $\mathbb{S}_{\mathsf{G}'}$ that have no proper policies from s. Importantly, for a fixed $\mathbb{S}$, the number of ways to pick $\mathbb{S}_{\mathsf{G}'}$ is finite, and for each $\mathbb{S}_{\mathsf{G}'}$ the number of proper policies from s is also finite. Therefore, we can indeed pick $\overline{N}(s, \mathbb{S})$ to be the minimal number that satisfies our constraints, so it is well-defined and finite.

This term will be useful because it lets us use the same term over many variants of the SSP $\mathbb{S}$. Without it, we would have a different $N(s, V, \mathbb{S}')$ for each SSP $\mathbb{S}'$, and we would have to be careful how they relate. The fact that $\overline{N}(s, \mathbb{S})$ can get large is unimportant for our proofs as long as $\epsilon\overline{N}(s, \mathbb{S}) \to 0$ as $\epsilon \to 0$. It is possible to sharpen our proofs with tighter upper bounds, but this does not contribute towards proving that CG-iLAO* is correct.

We are now equipped to formalise and prove our claim that ϵ-consistency over partial SSPs $\widehat{\mathbb{S}}$ is sufficient to prove optimality over the original SSP $\mathbb{S}$, provided that V is admissible w.r.t. $\mathbb{S}$, and the greedy policy $\widehat{\pi}_V$ does not terminate in one of $\widehat{\mathbb{S}}$'s artificial goal states $\widehat{\mathsf{G}} \setminus \mathsf{G}$. Note that we weaken the requirement of $V \leq V^*$, which will be useful for later proofs.

Theorem 3. Consider an algorithm that for each ϵ outputs $\widehat{\mathbb{S}}$ and V such that

- V is ϵ-consistent w.r.t. $\widehat{\mathbb{S}}$,
- V satisfies $V(s) \leq V^*(s) + \epsilon\overline{N}(s, \mathbb{S})\ \forall s \in \mathsf{S}^{\widehat{\pi}_V}$, and
- $\widehat{\pi}_V$ does not reach any artificial goal states, i.e., $\mathsf{S}^{\widehat{\pi}_V} \cap (\widehat{\mathsf{G}} \setminus \mathsf{G}) = \emptyset$.

As $\epsilon \to 0$, the greedy policy $\widehat{\pi}_V$ becomes an optimal policy, i.e., the algorithm is optimal.

Proof. First, ϵ-consistency over $\widehat{\mathbb{S}}$ implies that $\widehat{\pi}_V$ is accurately evaluated by V, i.e.,

$$|V(s) - V_{\widehat{\pi}_V}(s)| \leq \epsilon\overline{N}(s, \mathbb{S})\ \forall s \in \mathsf{S}^{\widehat{\pi}_V}.$$

To get this bound, we construct a copy of $\mathbb{S}$ called $\mathbb{S}'$ with the actions changed thus: $\mathsf{A}'(s) = \{\pi_V(s)\}\ \forall s \in \mathsf{S} \setminus \mathsf{G}$. The optimal value function of $\mathbb{S}'$ is $V_{\widehat{\pi}_V}$ (defined over $\mathsf{S}^{\widehat{\pi}_V}$), where $V_{\widehat{\pi}_V}(s)$ is the cost-to-go of $\widehat{\pi}_V$ from s. Our ϵ-consistent V must be globally ϵ-consistent for $\mathbb{S}'$, and we can apply theorem 1 to obtain $|V(s) - V_{\widehat{\pi}_V}(s)| \leq \epsilon N(s, V, \mathbb{S}')\ \forall s \in \mathsf{S}^{\widehat{\pi}_V}$. We get the desired bound by observing that $N(s, V, \mathbb{S}') \leq \overline{N}(s, \mathbb{S})$ for all $s \in \mathsf{S}^{\widehat{\pi}_V}$ by definition 15.

Second, we have $V_{\widehat{\pi}_V}(s) \geq V^*(s)\ \forall s \in \mathsf{S}^{\widehat{\pi}_V}$ because $\widehat{\pi}_V$ is a closed policy for $\mathbb{S}$ which can not be cheaper than an optimal one. We combine this with the previous observation to get

$$V(s) + \epsilon\overline{N}(s, \mathbb{S}') \geq V_{\widehat{\pi}_V}(s) \geq V^*(s)\ \forall s \in \mathsf{S}^{\widehat{\pi}_V}$$

which can be rearranged into

$$V(s) \geq V^*(s) - \epsilon\overline{N}(s, \mathbb{S})\ \forall s \in \mathsf{S}^{\widehat{\pi}_V}.$$

On the other hand, we have assumed the upper bound $V(s) \leq V^*(s) + \epsilon\overline{N}(s, \mathbb{S})\ \forall s \in \mathsf{S}^{\widehat{\pi}_V}$, which lets us squeeze V thus:

$$V^*(s) - \epsilon\overline{N}(s, \mathbb{S}) \leq V(s) \leq V^*(s) + \epsilon\overline{N}(s, \mathbb{S})\ \forall s \in \mathsf{S}^{\widehat{\pi}_V}.$$

As $\epsilon \to 0$, the errors disappear, so $V(s) = V^*(s)\ \forall s \in \mathsf{S}^{\widehat{\pi}_V}$ and $V(s) \leq V^*(s)\ \forall s \in \mathsf{S}$. Thus, the greedy policy $\widehat{\pi}_V$ is greedy w.r.t. V^* since $V(s) = V^*(s)$ over $\widehat{\pi}_V$'s envelope, i.e., $\widehat{\pi}_V$ must be an optimal policy. □

Thus, theorem 3 lets us relax the assumption that algorithms must be ϵ-consistent w.r.t. the original SSP, and shows that ϵ-consistency w.r.t. partial SSPs still provides optimality guarantees, provided that some additional requirements are satisfied. We briefly point out that ϵ-consistency w.r.t. partial SSPs can be relaxed even further by specifying a closed policy π, and inferring a partial SSP from π's envelope. Algorithms that are in this sense ϵ-consistent w.r.t. a specified policy are still optimal under similar assumptions to ϵ-consistency w.r.t. partial SSPs by similar arguments. The notion of ϵ-consistency w.r.t. partial SSPs accommodates CG-iLAO*, and equips us to prove that CG-iLAO* is sound and complete, i.e., it always returns some solution and it is optimal. We start by showing that CG-iLAO* always terminates:

Lemma 1. Partly-Expand-Fringes terminates.

Proof. In each pass of Partly-Expand-Fringes's main loop (algorithm 2 lines 15–20), a new state is removed from the artificial goals $\widehat{\mathsf{G}} \setminus \mathsf{G}$ and added as an internal state. But the SSP $\mathbb{S}$ has finitely-many states, so eventually Partly-Expand-Fringes must run out of new states to add, and will therefore exit its main loop and terminate. □

Lemma 2. CG-Backups terminates.

Proof. The main loop of CG-Backups (algorithm 2 lines 24–35) is running VI on $\mathcal{E} \setminus \widehat{\mathsf{G}}$ until ϵ-consistency, with additional bookkeeping (algorithm 2 lines 28–31) that does not affect VI. Given that $\widehat{\mathbb{S}}$ and $\mathcal{E}$ are fixed, this VI eventually reduces the residual to $\textsc{res} \leq \epsilon$, so CG-Backups exits the main loop and terminates. Note that the other conditions for termination $\mathsf{F} \neq \emptyset$ or $\widehat{\pi}_{\text{curr}} \neq \widehat{\pi}_{\text{old}}$ may cause CG-Backups to terminate sooner. □

Theorem 4. CG-iLAO* terminates.

Proof. For contradiction, suppose CG-iLAO* does not terminate. By lemma 1 and lemma 2 we know that the functions that CG-iLAO* calls must terminate, so if CG-iLAO* does not terminate, it must be because it gets stuck in its main loop (algorithm 2 lines 6–12). Suppose that $\mathsf{F} \neq \emptyset$ for infinitely-many steps. In each pass of the main loop where $\mathsf{F} \neq \emptyset$, Partly-Expand-Fringes changes at least one of $\widehat{\pi}_{\text{curr}}$'s fringe states to an internal state. But this is impossible, because $\mathbb{S}$ only has finitely-many states and we only allow fringe states to be expanded once, so CG-iLAO* must eventually run out of states to partially expand, and after finitely-many steps F remains empty. Moreover, after finitely-many steps $\widehat{\mathbb{S}}$ remains constant by a similar argument, since $\mathbb{S}$ only has finitely-many actions that can be added to $\widehat{\mathbb{S}}$. Then there must be a finite set of states $X \subseteq \widehat{\mathsf{S}}$ that are updated with Bellman backups infinitely often by CG-Backups or Fix-Constrs. But X induces a new partial SSP, and applying Bellman backups infinitely often to all X solves this new partial SSP with VI. Consequently, V converges to a fixed point and the residual will be less than ϵ in finite time. Since V will not be updated, $\widehat{\pi}_{\text{curr}}$ will also no longer be updated. At this point all the termination conditions are satisfied (algorithm 2 line 12), forcing CG-iLAO* to terminate, giving us the desired contradiction. □

We now work towards proving that CG-iLAO* is optimal by showing that it satisfies ϵ-consistency w.r.t. its partial SSPs, and that it satisfies the extra conditions required by theorem 3. To help with this, we prove some simpler properties of CG-iLAO* first:

Lemma 3. At the end of CG-iLAO*'s main loop (algorithm 2 after line 11), if there is $s \in \widehat{\mathsf{S}} \setminus \widehat{\mathsf{G}}$ and $a \in \mathsf{A}(s)$ such that $V(s) > Q(s,a) + \epsilon$, then $(s,a) \in \Gamma$ or $V(s) \leq V^*(s)$.

Proof. We prove by induction over n, the number of iterations of CG-iLAO*. The base case is $n = 0$, i.e., the partial SSP $\widehat{\mathbb{S}}$ has been initialised and never updated, so $\widehat{\mathsf{S}} \setminus \widehat{\mathsf{G}} = \emptyset$, making the claim is vacuously true. Our inductive hypothesis is that the lemma holds after n iterations, and we now prove that this makes the lemma hold after $n+1$ iterations. We step through the possible ways that the lemma's precondition can be made true, and how in those cases the algorithm ensures that the relevant column is in Γ:

1. If $V(s) > Q(s,a) + \epsilon$ at the start of iteration $n+1$, then by the inductive hypothesis, $(s,a) \in \Gamma$ or $V(s) \leq V^*(s)$. So, we only concern ourselves with any constraint violations that are introduced in this iteration.

2. The post-order DFS traversal (algorithm 2 line 7) does not affect $\widehat{\mathsf{S}} \setminus \widehat{\mathsf{G}}$ nor V, and therefore can not introduce any new constraint violations.

3. Partly-Expand-Fringes (algorithm 2 line 8) does not affect V, but may introduce new states into $s \in \widehat{\mathsf{S}} \setminus \widehat{\mathsf{G}}$. It expands fringe states s in $\mathsf{S}^{\widehat{\pi}_{\text{curr}}}$ by bringing s into $\widehat{\mathsf{S}} \setminus \widehat{\mathsf{G}}$, adding some action $\widehat{a} \in \mathsf{A}(s)$ into $\widehat{\mathsf{A}}(s)$, and setting $\widehat{\pi}_{\text{curr}}(s) \leftarrow \widehat{a}$. This may introduce constraint violations but s was until now a fringe state, so $V(s) = H(s) \leq V^*(s)$.

4. In CG-Backups (algorithm 2 line 10), a new $V(s) > Q(s,a) + \epsilon$ may occur if $V(s)$ is increased or $Q(s,a)$ is decreased. CG-Backups tracks the appropriate Ext-Succs or Preds for V increases and decreases respectively.

5. Fix-Constrs (algorithm 2 line 11) ensures for each $(s,a) \in \Gamma$ that $V(s) \leq Q(s,a) + \epsilon$ before removing (s,a) from Γ. $V(s)$ can not increase in Fix-Constrs, since we are setting $V(s) \leftarrow \min\{V(s), Q(s,a)\}$ for $(s,a) \in \Gamma$. However, new instances of $V(s) > Q(s,a) + \epsilon$ can be introduced if some $Q(s,a)$ is decreased, which is tracked by Fix-Constrs.

So indeed, any constraint violations $V(s) > Q(s,a) + \epsilon$ introduced by CG-iLAO* in lines 7–11 are tracked in Γ or $V(s) \leq V^*(s)$. □

Importantly, this lemma applies only to internal states $s \in \widehat{\mathsf{S}} \setminus \widehat{\mathsf{G}}$, but in terms of actions applies to all applicable actions $a \in \mathsf{A}(s)$, not only internal ones. Equipped with this invariant, we prove two important properties of CG-iLAO* when $\Gamma = \emptyset$.

Lemma 4. At the end of CG-iLAO*'s main loop (algorithm 2 after line 11), $\Gamma = \emptyset$ implies V is admissible up to an error, i.e., $V(s) \leq V^*(s) + \epsilon \overline{N}(s, \mathbb{S})\ \forall s \in \mathsf{S}$.

Proof. All fringe, external, and true goal states $s \in \widehat{\mathsf{G}} \cup (\mathsf{S} \setminus \widehat{\mathsf{S}})$, are initialised with $V(s) \leftarrow H(s) \leq V^*(s)$ and never have their $V(s)$ updated. For the remaining states, i.e., the internal states $\widehat{\mathsf{S}} \setminus \widehat{\mathsf{G}}$, we can use lemma 3. We ignore the internal states s with $V(s) \leq V^*(s)$, because they immediately satisfy the

requirement of the lemma, and we focus on the internal states with $V(s) > V^*(s)$, which we call $\tilde{\mathsf{S}}$. We assume that $\Gamma = \emptyset$, so by the contrapositive of lemma 3 it must be the case that $V(s) \leq Q(s,a) + \epsilon \; \forall s \in \tilde{\mathsf{S}}$. It follows that $V(s) \leq V^*(s) + \epsilon \overline{N}(s, \mathbb{S}) \; \forall s \in \tilde{\mathsf{S}}$. To see this: if we apply VI to $\tilde{\mathbb{S}}$, the $\mathbb{S}$ restricted to the states $\tilde{\mathsf{S}}$, V will remain unaffected because $\textsc{res}(s) \leq \epsilon$ for these states, and then we can apply theorem 1 to get $V(s) \leq V^*(s) + \epsilon N(s, V, \tilde{\mathbb{S}}) \; \forall s \in \tilde{\mathsf{S}}$. Then, $N(s, V, \tilde{\mathbb{S}}) \leq \overline{N}(s, \mathbb{S}) \; \forall s \in \tilde{\mathsf{S}}$ (definition 15), giving the desired bound. Thus, we have addressed all states in S, proving our claim. □

Lemma 5. Upon termination, CG-iLAO* has $\Gamma = \emptyset$.

Proof. For contradiction, suppose CG-iLAO* does terminate with $\Gamma \neq \emptyset$. Then, Γ must have been populated by the final call to Fix-Constrs, but constraints are only added to Γ in Fix-Constrs when $V(s) > Q(s,a) + \epsilon$, and therefore $\textsc{res} \geq V(s) - Q(s,a) > \epsilon$. This yields the desired contradiction because CG-iLAO* can not terminate until $\textsc{res} \leq \epsilon$. □

Finally, we are ready to prove that CG-iLAO* is optimal.

Theorem 5. CG-iLAO* is optimal.

Proof. Suppose that CG-iLAO* terminates with the value function V, the partial SSP $\widehat{\mathbb{S}}$, and the candidate policy $\widehat{\pi}_{\text{curr}}$. Upon termination, we know that $\widehat{\pi}_{\text{curr}} = \widehat{\pi}_V$, otherwise a policy change must have occurred. Then, V satisfies ϵ-consistency w.r.t. $\widehat{\mathbb{S}}$, since $\textsc{res} \leq \epsilon$ is one of CG-iLAO*'s termination conditions (algorithm 2 line 12). Now, it suffices to show $V(s) \leq V^*(s) + \epsilon \overline{N}(s, \mathbb{S}) \; \forall s \in \mathsf{S}$ and that $\widehat{\pi}_V$ does not encounter any artificial goals $\widehat{\mathsf{G}} \setminus \mathsf{G}$, and then we can apply theorem 3 to conclude that CG-iLAO* is optimal. Indeed, upon termination, CG-iLAO* has $\Gamma = \emptyset$ by lemma 5, so we can apply lemma 4 to get $V(s) \leq V^*(s) + \epsilon \overline{N}(s, \mathbb{S}) \; \forall s \in \mathsf{S}$. We also know that $\widehat{\pi}_V$ does not encounter any artificial goals thanks to CG-iLAO*'s termination condition $\mathsf{F} \neq \emptyset$ (algorithm 2 line 12). □

In these proofs, we have assumed that $\eta = 0$ in order to guarantee that all cost increases and decreases are tracked. In practice, we use $\eta > 0$ due to the inherent numerical instability issues of floating-point number representations, and in particular it is convenient to choose $\eta = \epsilon$, to minimise the number of parameters. This is not problematic in our benchmarks, but one must be careful because $\eta > 0$ can introduce arbitrarily large errors into the value function. For example, consider the SSPs in figure 9, with initial value functions shown in the bottom of each node. For the SSP on the left, CG-iLAO* with $\eta = \epsilon$ does the following:

Iter. 1 Partially expands s_0 with a_0.

Iter. 2 Expands s_1 with a_1 and updates $V(s_1) = 3 + \frac{2\epsilon}{3}, V(s_0) = 4 + \frac{2\epsilon}{3}$. The increases to $V(s_1)$ and $V(s_0)$ are $\frac{2\epsilon}{3} < \eta = \epsilon$, so nothing is added to Γ.

Iter. 3 Expands s_2 with a_2 and updates $V(s_2) = 2 + \frac{2\epsilon}{3}, V(s_1) = 3 + \frac{4\epsilon}{3}, V(s_0) = 4 + \frac{4\epsilon}{3}$. Again, the increases to V are $\frac{2\epsilon}{3} < \eta = \epsilon$, so nothing is added to Γ.

Iter. 4 Expands s_3 with a_3 and updates $V(s_3) = 1 + \frac{2\epsilon}{3}, V(s_2) = 2 + \frac{4\epsilon}{3}, V(s_1) = 3 + \frac{6\epsilon}{3}, V(s_0) = 4 + \frac{6\epsilon}{3}$. For the last time, the increases to V are $\frac{2\epsilon}{3} < \eta = \epsilon$, so nothing is added to Γ.

Iter. 5 There are no fringes left to expand, so CG-iLAO* does one pass of Backups which leaves V unchanged and exits with $V(s_0) = 4 + 2\epsilon$. The algorithm terminates.

Thus, CG-iLAO* finds a policy for the SSP on the left, which is suboptimal by ϵ. CG-iLAO* behaves similarly on the right SSP, accumulating an error of $(2k-1)\epsilon$ which gets arbitrarily large as the number of states n grows. Clearly, as $\eta \to 0$ this error disappears, so in particular $\eta = \epsilon$ works in that sense, but one has to be careful. Regardless of the choice of ϵ, we now have the additional theoretical issue that for any η it is possible to construct a pathological SSP that makes CG-iLAO*'s policy arbitrarily bad.

We note that although CG-iLAO* is not ϵ-consistent w.r.t. $\mathbb{S}$ in general, it becomes ϵ-consistent w.r.t. $\mathbb{S}$ if H is monotonic. Also, we can make CG-iLAO* ϵ-consistent w.r.t. $\mathbb{S}$ by modifying Partly-Expand-Fringes so that for each expanded fringe $s_f \in \mathcal{E} \cap (\widehat{\mathsf{G}} \setminus \mathsf{G})$, we add $\mathsf{A}(s_f)$ to Γ. This addresses the edge case in step 3. of lemma 3's proof. As explained before, it is not important that CG-iLAO* is ϵ-consistent w.r.t. $\mathbb{S}$ for the normal use-case of CG-iLAO*.

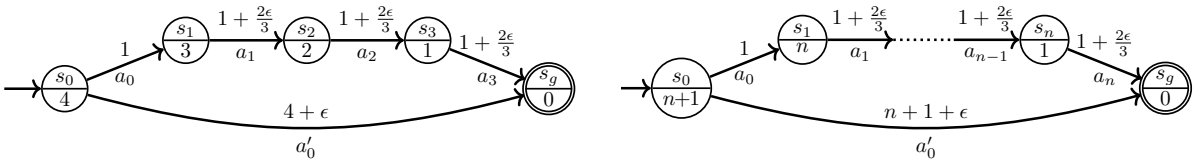


Figure 9: SSPs where CG-iLAO* with a large η ($\eta \geq \epsilon$) accumulates a large error. The left SSP ends up with an error of ϵ and $V(s_0) = 4 + 2\epsilon$, and the right SSP can have an arbitrarily large error of $(2k-1)\epsilon$ with $V(s_0) = n + 1 + 2k\epsilon$ for $k = 3n$.

5 Different Expansion Methods for CG-iLAO*

In this section we explore different methods that CG-iLAO* can use to expand fringe states. For the correctness of CG-iLAO*, Partly-Expand-Fringes$(\mathbb{S}, \widehat{\mathbb{S}}, \widehat{\pi}_V, \mathcal{E}, V)$ has two requirements:

1. at least one fringe state $s \in \mathcal{E} \cap (\widehat{\mathsf{G}} \setminus \mathsf{G})$ is expanded, i.e., afterwards $s \notin \widehat{\mathsf{G}} \setminus \mathsf{G}$ and $\widehat{\mathsf{A}}(s) \neq \emptyset$ or $s \in \mathsf{G}$,
2. after expansion, $\mathcal{E}$ is up to date and new fringe states are accurately recorded in $\widehat{\mathsf{G}}$.

In this work, Partly-Expand-Fringes expands all fringe states $\mathcal{E} \cap (\widehat{\mathsf{G}} \setminus \mathsf{G})$, i.e., we do not consider a method that expands a subset of fringe states. This is based on the empirical results of Hansen and Zilberstein [2001] that suggest expanding all reachable fringes is more efficient than expanding subsets.

We now describe different methods for expanding each fringe state.

5.1 One-step-look-ahead

Here, we consider three methods for expanding a state s that only look one step ahead and greedily select actions based on their Q-values.

Single-Greedy-Action expands s by computing the Q-values of all applicable actions, and selects a single greedy action, breaking ties arbitrarily but in a reproducible way (assumption 3). Formally, Single-Greedy-Action(s) sets

$$\widehat{\mathsf{A}}(s) \leftarrow \{\widehat{a}\} \text{ for some } \widehat{a} \in \mathsf{A}(s) \text{ s.t. } Q(s,\widehat{a}) = \min_{a \in \mathsf{A}(s)} Q(s,a). \tag{3}$$

In the best case, this method adds precisely the actions that define an optimal policy, and consequently CG-iLAO*'s partial SSP is minimal in the sense that it only contains the states and actions that are necessary to define the relevant optimal policy.

Theorem 6. If initialised with V^*, CG-iLAO* with Single-Greedy-Action as its expansion mechanism will have a minimal partial SSP. That is, given CG-iLAO*'s output π^*, its partial SSP will consist precisely of $\widehat{\mathsf{S}} = \mathsf{S}^{\pi *}$ and $\widehat{\mathsf{A}}(s) = \{\pi^*(s)\}$ for each $s \in \widehat{\mathsf{S}} \setminus \mathsf{G}$.

Proof. In each expansion, CG-iLAO* will insert the greedy action according to $V = V^*$. Then, Bellman updates will not change V since the greedy actions' $Q(s,a)$ are precisely $V^*(s)$ and $V = V^*$. Therefore Γ remains empty and the greedy policy is never updated (it breaks ties in favour of its current actions). So, CG-iLAO* will perform a greedy best-first search using $V = V^*$, which results in expanding precisely the states in the greedy policy envelope S^{π_V} and the corresponding actions. □

Note that if there are multiple optimal policies, there is no guarantee that CG-iLAO* returns the one with the smallest envelope, i.e., it does not guarantee the fewest possible expansions over all optimal policies.

In practice, it is usually worthwhile to add all tied-greedy actions upon expansion. So, we define the expansion All-Greedy-Actions(s) which sets

$$\widehat{\mathsf{A}}(s) \leftarrow \left\{ \widehat{a} \in \mathsf{A}(s) : Q(s,\widehat{a}) = \min_{a \in \mathsf{A}(s)} Q(s,a) \right\}. \tag{4}$$

To motivate that it often pays off to add tied-greedy actions, suppose CG-iLAO* expands state s with the single greedy action $a_{\min}$. If $V(s)$ increases, then CG-iLAO* is forced to add all remaining external successor actions $\mathsf{A}(s) \setminus \{a_{\min}\}$ to Γ, and from these must add all actions with $Q(s,a) = Q(s,a_{\min})$ to

$\widehat{\mathbb{S}}$. Such increases to $V(s)$ are common, e.g., an increase is guaranteed when $H(s) < \min_{a \in \mathsf{A}(s)} Q(s,a)$, which is typical for practical heuristics. Thus, in this common situation all tied-greedy actions get added to $\widehat{\mathbb{S}}$ anyway (assuming their own Q-values did not increase significantly), so it makes sense to avoid the overhead and add them immediately. This is the expansion presented in CG-iLAO*'s pseudocode (algorithm 2).

The final option we consider is to simply add all applicable actions. With this expansion, CG-iLAO* implements iLAO*, noting the minor differences in implementation discussed in section 4.[6]

5.2 Rollout

We now consider expansion methods that use rollouts. A rollout from state s is obtained by simulating the execution of a *base policy*: we start in s, apply the base policy, randomly select the next state according to P, apply the base policy to the new state, and so on. This produces a trajectory through the SSP from s, which we can use to inform our expansion. Rollouts have a long history in value-iteration style algorithms, and they have enjoyed success in planning, e.g., PROST [Keller and Eyerich, 2012], LRTDP [Bonet and Geffner, 2003b], and Robust-FF [Teichteil-Königsbuch, Kuter, and Infantes, 2010]. The primary benefit of rollouts is that they enable information from the goal to be propagated sooner. Observe that the value functions of CG-iLAO* with one-step-lookahead expansions are informed exclusively by the heuristic for most of their lifetimes, and only start propagating information from goal states when a goal enters the partial SSP. To try to address this, we implement rollouts into CG-iLAO* during the expansion phase: when expanding state s, we perform a rollout from s to generate a trajectory $\langle s^0, a^0, s^1, a^1, \ldots, s^n \rangle$, and then add all the state-action pairs (s^i, a^i) to the partial SSP. Although this may insert suboptimal actions into the partial SSP, this expansion satisfies the requirements of Partly-Expand-Fringes, and therefore does not affect CG-iLAO*'s correctness. We do not consider adding rollouts to iLAO*'s expansion, because iLAO* would have to add all applicable actions for all the states in the trajectory, which makes its partial SSP significantly larger than it needs to be. We now present two base policies to use for CG-iLAO*'s rollout expansions: trials and FF.

Trial rollouts Following RTDP [Barto, Bradtke, and Singh, 1995], we roll out with a trial by selecting the greedy action w.r.t. V at each step. The pseudocode for our trials is given in algorithm 3.

SSPs can have cycles, and a naïve implementation of trial is susceptible to getting stuck inside a cycle indefinitely. RTDP deals with cycles by updating $V \leftarrow \min_{a \in \mathsf{A}(s)} Q(s,a)$ for each state s it encounters; suppose RTDP's simulation is stuck inside a cycle, then the relevant V values increase until the Q-values to remain inside the cycle get so large that the simulation leaves the cycle. In practice, to handle SSPs that violate reachability (assumption 1), we use the fixed-penalty transform [Trevizan, Teichteil-Königsbuch, and Thiébaux, 2017] that allows the agent to "give-up" with a large cost $d \in \mathbb{R}_{>0}$ — consequently, if RTDP is stuck within a cycle, the simulation "gives up" as soon as $V(s) \geq d$. Similarly to RTDP, we update V in our trial, but, importantly, in order for CG-iLAO* to remain correct, it must preserve that $V(s) \leq V^*(s)$ for all external states $s \in \mathsf{S} \setminus \widehat{\mathsf{S}}$. Recall that CG-iLAO* may have $V(s) \not\leq V^*(s)$ for internal states $s \in \widehat{\mathsf{S}}$. To preserve $V(s) \leq V^*(s)$ for external states, we force the trial to terminate whenever it can reach an internal state from its current state s, i.e., the trial is stopped whenever there is an action $a \in \mathsf{A}(s)$ such that $\mathrm{succ}(s,a) \cap \widehat{\mathsf{S}} \neq \emptyset$. Consequently, $V(s)$ is only updated by the trial if all effects of all applicable actions are external. External states are in this way only updated with full Bellman backups using other external states. Recall that external states are initialised with $V(s) = H(s) \leq V^*(s)$, so by updating $V(s)$ for external states only by applying Bellman backups to other external states, we guarantee that $V(s) \leq V^*(s)$ for external states. Terminating the trial whenever an internal state is reachable also has the intuitive interpretation that we stop the rollout as soon as it reaches a state where "we know what to do already."

A trial from s can either "succeed" by reaching a goal or an internal state, or it can "fail" by reaching a dead end or by its number of steps exceeding $t_{\max}$.

- If the trial succeeded: we first remove any cycles from the trial, then we expand the remaining acyclic trajectory by adding all its state-action pairs to $\widehat{\mathbb{S}}$.[7]

[6] There is a minor exception apart from the implementation changes where CG-iLAO* with Expand-Fringes does not behave like iLAO*: if H is not monotonic, then a decrease to V can occur, causing constraints to be added to Γ, which CG-iLAO* has to deal with. Though possible, we have not observed this edge case in our experiments.

[7] To remove cycles from the trial we identify the first and last occurrence of each state in the trial, and if these are not equal, remove the subsequence between them (as well as one of the copies of the state).

Algorithm 3: trial

```
1  Function trial(s_init, V, t_max)
2      s ← s_init
3      trajectory ← ⟨s⟩
4      t ← 0
5      while s ∉ Ŝ and s ∉ G do
           // Return failure if the trial reaches a dead end or times out
6          if t > t_max or V(s) > d or A(s) = ∅ then
7              return failure
           // If there is an action that reaches Ŝ:  take the cheapest one
8          if ∃a ∈ A(s) : succ(s, a) ∩ Ŝ ≠ ∅ then
9              a_min ← argmin_{a∈A(s):succ(s,a)∩Ŝ≠∅} Q(s, a)
10             s ← some state ∈ succ(s, a_min) ∩ Ŝ
           // Otherwise continue RTDP trial as normal
11         else
12             a_min ← argmin_{a∈A(s)} Q(s, a)
13             V(s) ← Q(s, a_min)
14             s ← state randomly chosen from P(·|s, a_min)
15         append ⟨a_min, s⟩ to trajectory
16         t ← t + 1
17     remove cycles from trajectory
18     return trajectory
```

- If the trial failed: it does not make sense to expand the entire trial; the only fact we know about this sequence of states and actions is that they do not result in a goal or an internal state. Instead, we expand s with ALL-GREEDY-ACTIONS on the updated V, with the idea that the updated V will inform the one-step-lookahead to avoid the action that lead to failure.

Our trials are parameterised by $t_{\max}$. A low value of $t_{\max}$ serves as another mechanism to escape cycles, and more generally as a mechanism to avoid wasting time on a simulation that ends up in an expensive part of the state space that CG-iLAO* does not benefit from expanding. However, this is at the cost of finding fewer successful trajectories that end in goals or internal states.

FF rollouts Inspired by the success of FF-Replan [Yoon, Fern, and Givan, 2007] and Robust-FF [Teichteil-Königsbuch, Kuter, and Infantes, 2010], we use the deterministic algorithm FF [Hoffmann and Nebel, 2001] to find a plan for the SSP's determinisation, and use the plan as a rollout. A determinisation removes the SSP's probabilistic effects, and produces a deterministic shortest path problem. We consider two determinisations:

1. **All-outcomes determinisation.** The all-outcomes determinisation splits each probabilistic action $a \in \mathsf{A}(s)$ into multiple deterministic ones. For each $s' \in \text{succ}(s, a)$, the determinisation produces the actions $a_{s \to s'}$ that lead to s' deterministically. This is proper relaxation of the SSP: if a proper policy exists for the SSP, then a plan must exist for the all-outcomes determinisation.

2. **Most-likely-outcome determinisation.** The most-likely-outcomes determinisation transforms each probabilistic action $a \in \mathsf{A}(s)$ into a single deterministic action. It selects s' for some $s' \in \text{argmax}_{s''} P(s''|s, a)$ (breaking ties arbitrarily), and then produces the single action $a_{s \to s'}$ that leads to s' deterministically. This determinisation has the benefit that it has fewer actions and maintains some information about probability distributions by picking the most likely; however, it is not a proper relaxation, because it can happen that the single most-likely outcomes that we selected do not allow a plan to be constructed.

We run FF rollouts by running FF on the respective determinisation. Then, FF returns the plan $\langle s^0, a^0, \ldots, s^n \rangle$ where $s^0 = s$ and $s^n \in \mathsf{G}$, or returns that no plan exists. If it finds a plan, we expand it. If no plan exists on the all-outcomes determinisation, then it is proof that the state we are expanding is a dead end and we mark it as such. If no plan exists on the most-likely-outcomes determinisation we can not conclude that it is a dead end, so we run FF again, but on the all-outcomes determinisation, to construct a plan or prove a dead end.

6 Related Work

In this section, we present existing work that is related to CG-iLAO*. First, we present and discuss action elimination, which has the same goal as CG-iLAO* of avoiding Q-value computations for unneeded actions. We explain how it is different, and has various disadvantages compared to our approach. Afterwards, we discuss existing work in planning that uses constraint generation, motivating that our use of constraint generation is novel. Finally, we compare CG-iLAO* to the Partial Expansion A* algorithm, which uses the same paradigm of ignoring actions, but in the deterministic setting.

6.1 Action Elimination

Action elimination lets us prove that certain actions can not be part of the optimal policy, and can therefore be permanently removed from search without affecting optimality. This technique, similarly to CG-iLAO*'s mechanism for ignoring actions, allows us to avoid computing Q-values for actions that will not help with the solution. However, action elimination approaches the issue from the opposite direction, in the sense that it starts with all actions and permanently removes them when they are proved to be outside optimal policies, compared to CG-iLAO*'s mechanism which initially ignores all actions and adds them as required.

Action elimination requires upper and lower bounds on V^*, called V_{ub} and V_{lb} respectively, and determines which actions can be eliminated using the following theorem.

Theorem 7 (Action Elimination [Bertsekas, 1995])**.** Given a state $s \in \mathsf{S}$ and action $\tilde{a} \in \mathsf{A}(s)$, if $\exists a \in \mathsf{A}(s)$ such that $Q_{\text{lb}}(s, \tilde{a}) > Q_{\text{ub}}(s, a)$, then action $\tilde{a}$ cannot be optimal for state s, and can therefore be eliminated.

We have already discussed how to maintain V_{lb} as a lower bound. Note that outside this section we call such value functions V, but here we call it V_{lb} to emphasise the distinction between upper and lower bounds. The main drawback of action elimination, and the reason its use is not widespread in planners, is that obtaining the upper bound V_{ub} is difficult. There are methods for deriving an upper bound V_{ub} from only a lower bound and its Bellman residual [Hansen and Zilberstein, 2001; Hansen and Abdoulahi, 2015], but they require the expensive computation of the expected number of steps to reach the goal or that the policy is proper, which is impractical for our purposes. The standard way to track V_{ub} is to initialise V_{ub} s.t. $V_{\text{ub}} \geq V^*$, and then apply Bellman backups to it, which are guaranteed to preserve $V_{\text{ub}} \geq V^*$. This is done in addition to tracking V_{lb}, so the algorithm tracks two value functions, and each partial backup must be applied both to V_{lb} and V_{ub}. Thus, if all else is the same, then tracking the two value functions exactly doubles the number of computed Q-values. An additional difficulty for tracking V_{ub}, on top of the additional maintenance cost, is that we are not aware of any efficient and domain-independent algorithms to compute an initial upper bound. The only domain-independent upper bound we are aware of, is the trivial upper bound

$$V_{\text{ub}}(s) = \begin{cases} 0 & \text{if } s \in \mathsf{G} \\ d & \text{if } s \notin \mathsf{G} \end{cases}$$

where $d = \infty$ or the penalty term, if we are using a fixed-penalty transformation.

Action elimination has not been used by optimal heuristic-search algorithms to remove actions during search. There are variants of RTDP [Barto, Bradtke, and Singh, 1995] that track both V_{ub} and V_{lb}, such as BRTDP [McMahan, Likhachev, and Gordon, 2005], FRTDP [Smith and Simmons, 2006], VPI-RTDP [Sanner et al., 2009]; and variants for LAO* [Hansen and Zilberstein, 2001] such as IBLAO* [Warnquist, Kvarnström, and Doherty, 2010]. However, none of these use action elimination: they use lower and upper bounds to define different ways to prioritise expansion and backups, and to define alternative termination conditions.

The only algorithm we are aware of that explicitly uses action elimination is FTVI [Dai et al., 2011]. FTVI consists of a computation stage where it runs Topological Value Iteration (TVI), and a preprocessing stage, where it tries to make the search space more amenable to TVI. TVI works well when the state-space has small strongly connected components (SCCs), and degenerates to standard VI as the SCCs get bigger. The preprocessing stage attempts to break any large SCCs into smaller ones by removing suboptimal actions with action elimination. This step iteratively performs a Depth-First Search on the state-space, applies Bellman backups on V_{lb} and V_{ub} in post-order, and uses action elimination to remove actions when it can. Note that FTVI is not a heuristic search algorithm because its search step does not use a heuristic and considers the whole state-space.

Action elimination has another disadvantage w.r.t. to CG-iLAO*: it initially considers all actions, and only eliminates them later in its lifetime. In comparison, CG-iLAO* starts with no actions, and adds them as they are deemed necessary. This means that CG-iLAO* starts with a small partial SSP, and grows it over time, enjoying a relatively small partial SSP during its entire execution; whereas action elimination has a large partial SSP for most of its execution, only shrinking it when V_{lb} and V_{ub} become sufficiently accurate to prove actions are suboptimal.

To summarise, action elimination has two key disadvantages compared to CG-iLAO*'s mechanism for ignoring actions:

1. Action elimination requires both a lower bound V_{lb} and an upper bound V_{ub}, whereas CG-iLAO* only needs V_{lb}. Maintaining V_{ub} incurs an overhead of Q-value computations.

2. Action elimination is only able to shrink its partial SSPs late in its lifetime, whereas CG-iLAO* enjoys smaller partial SSPs from the start.

We show experimentally in section 6.1 that CG-iLAO*'s mechanism for ignoring actions is indeed more efficient than action elimination.

6.2 Constraint Generation in Planning

The paradigm of constraint generation lends itself well to complex planning problems where a relaxation can be solved efficiently. An example that demonstrates this very clearly is Multi-Agent Path-Finding (MAPF) [Stern et al., 2021], where the aim is to coordinate multiple agents to achieve their objectives as efficiently as possible, making sure that the agents do not interfere with each, e.g., they should not crash into each other. This problem is notoriously difficult, especially as the number of agents increases; finding optimal solutions is NP-hard [Yu and LaValle, 2013], and in some cases even finding a feasible solution is NP-hard [Nebel, 2020]. On the other hand, if we focus on each agent in isolation, then finding their individual optimal paths is a classical planning problem, which can be solved efficiently. If the isolated optimal paths do not interfere, then this immediately gives an optimal solution for the MAPF problem, but in general there will be interferences that must be addressed. We can impose constraints that block such interferences; for example, suppose that following the isolated paths causes agents A and B to crash into each other in location (x, y) at time t, then we can prevent the crash by disallowing A from entering (x, y) at time t. Then, a viable algorithmic loop is to find paths for the agents in isolation, determine where they interfere, add constraints that block the undesired behaviour, then find updated plans for the agents in isolation that respect the added constraints, and repeat until no interferences remain. The MAPF community calls this technique Conflict Based Search (CBS) [Sharon et al., 2012], and it has seen a reasonable amount of success [Boyarski et al., 2015]. Constraint generation has also been used explicitly (by name) in MAPF with Mixed Integer Programs [Calliess and Roberts, 2021]. The same paradigm applies similarly to other complex planning problems, e.g., in metric hybrid factored planning in nonlinear domains [Say and Sanner, 2019].

Counter-Example Guided Abstraction Refinement (CEGAR) [Seipp and Helmert, 2013] can also be considered a form of constraint generation, which has seen success in constructing heuristics [Rovner, Sievers, and Helmert, 2019] and policy verification [Vinzent, Sharma, and Hoffmann, 2023]. In the context of classical planning, the idea is to use abstraction to compress the original problem's state-space, so that the ensuing problem is smaller and easier to solve, but still captures some of the original problem's structure. Then, we can solve the smaller problem to obtain a plan. A plan for the smaller problem may immediately be a solution for the original problem, in which case the problem is solved; more likely, the plan has some *flaw* when we try to apply it to the original problem. This is called a counter-example, and is used to refine the abstraction so that this particular flaw can not occur again, effectively adding constraints prohibiting it. Consequently, we get the algorithmic loop that solves an abstraction, finds flaws, refines the abstraction to prohibit these flaws, resolves the abstraction, and then repeats until no more flaws can be found. Again, this fits the framework of constraint generation.

These constraint-generation approaches differ quite significantly from CG-iLAO*, because their separation oracle determines how a candidate solution violates the constraint of the original planning problem, whereas CG-iLAO*'s separation oracle determines violations of monotonicity in the value function. There is also existing work in the planning literature that applies constraint generation to various value functions.

For Partially Observable MDPs (POMDPs) the value function is encoded by a set of vectors, and an LP with constraint generation can prune unneeded vectors [Walraven and Spaan, 2017]. Although this LP is related to the value function, pruning vectors is a fundamentally different task from the one we

are addressing. Similarly, a linear value function can give a compact approximation of the value function for factored MDPs, and if we replace the value function in LP 1 with this approximation, we obtain the approximate LP (ALP) [Mausam and Kolobov, 2012] with a significant reduction in the number of variables. Schuurmans and Patrascu [2001] apply constraint generation to the ALP, and the separation oracle has a similar condition for adding constraints as CG-iLAO*; however, it checks for the condition naïvely, which is practical with the compact representation of factored MDPs with a linear value function approximation, but not for SSPs.

In all these works, to find constraint violations the separation oracle either naïvely checks all possible constraints or relies on sampling, compared to CG-iLAO*, which efficiently tracks potential violations via changes in the value function.

6.3 Partial Expansion A*

The concept of ignoring unneeded actions has been considered in the deterministic shortest path setting by the Partial Expansion A* (PEA*) algorithm [Yoshizumi, Miura, and Ishida, 2000]. Similarly to CG-iLAO*, states are partially expanded: the algorithm only considers a greedy subset of applicable actions, and efficiently tracks the other actions to be reconsidered when they may become needed. We assume that the reader is familiar with A* [Hart, Nilsson, and Raphael, 1968; Russell and Norvig, 2016], and we only give a brief reminder of how the algorithm works. We use similar notation to Yoshizumi, Miura, and Ishida [2000], but talk about states instead of nodes. Then, for a state s

- $g(s)$ denotes the lowest path cost to s, i.e., the currently best-found cost to move from s_0 to s;
- $h(s)$ is an admissible heuristic function evaluated at s (defined similarly to heuristic functions in our setting);
- $f(s) = g(s) + h(s)$ is the evaluation function.

A* maintains an *open list* of states (also called the frontier), from which the algorithm iteratively pops a state s with lowest $f(s)$ and then expands it. We assume that the open list is implemented as a priority queue ordered by f. To expand s, A* iterates over s's successors and places them into the open list with their respective g, h, and f values. A key feature of A* is that it will stop as soon as it pops a goal from the open list, and any states that remain in the open list are discarded, not needing to be considered for constructing the optimal plan. For problems with large branching factors the number of such discarded states can get very large, which makes it expensive maintain the priority queue, and in extreme cases causes the algorithm to run out of memory. PEA* addresses this by using partial expansions. A partial expansion for s is defined as follows: let SUCC denote the successor states of s, then the partial expansion only adds the greedy successor states

$$\mathrm{SUCC}_{\leq 0} = \left\{ s_{\min} \in \mathrm{SUCC} : f(s_{\min}) = \min_{s' \in \mathrm{SUCC}} f(s') \right\}$$

and all non-greedy successors are captured by placing the parent state s back into the open list with an updated priority

$$F(s) = \min_{s' \in \mathrm{SUCC} \setminus \mathrm{SUCC}_{\leq 0}} f(s').$$

This $F(s)$ can be read as the smallest f-value of s's non-greedy successors. Note that PEA* uses F to sort its priority queue, and we use $F(s) = f(s)$ if a state has not been partially expanded before. The key idea is that, if search determines that the states we have ignored become relevant, then s will be popped again as a proxy for its successors, and the partial re-expansion of s will place the newly greedy successors into the open list, and progress as though these states were always on the open list. Thus, the algorithm can maintain a smaller open list by ignoring unpromising states, at the cost of requiring additional re-expansion steps.

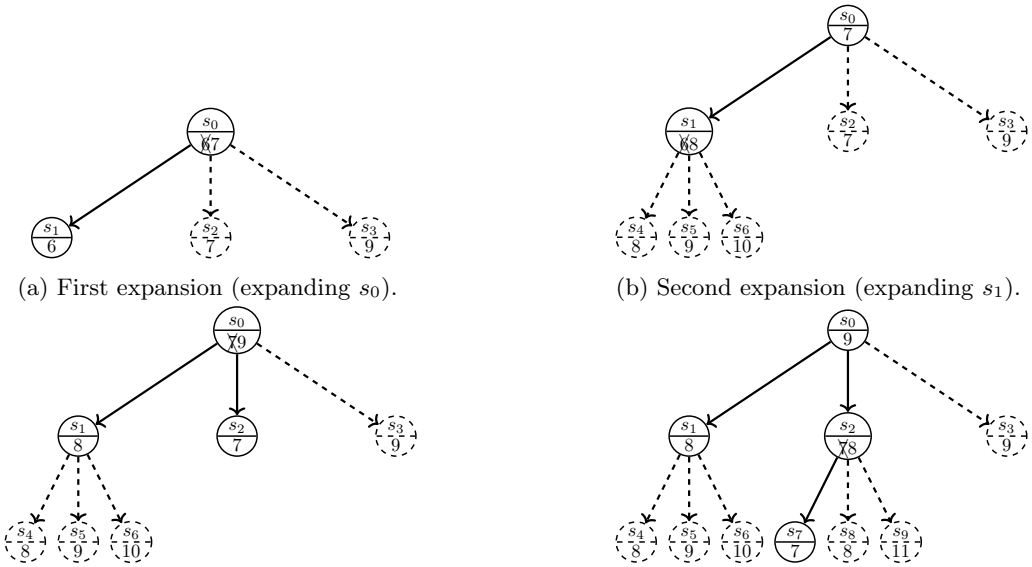
If iLAO* is the generalisation of A* to SSPs, then CG-iLAO* is the generalisation of PEA*, and CG-iLAO* shares the key behaviours of PEA* when applied to deterministic problems. It is awkward to compare CG-iLAO* and PEA* directly, because their settings are different: CG-iLAO* computes the states' costs-to-go in V and tracks the current greedy policy $\widehat{\pi}_{\mathrm{curr}}$, whereas PEA* is computing the lowest path costs g and implicitly tracking the best plan by recording each state's parent. Nevertheless, the order in which CG-iLAO* and PEA* handle states and actions is comparable. We illustrate this in figure 10 and figure 11. Figure 10 is the example from Yoshizumi, Miura, and Ishida [2000] that demonstrates the expansion order of PEA*. Solid and dotted circles indicate that the state has or has not been added to

the open list, respectively. The number at the bottom of each circle represents the corresponding state's priority on the open list, i.e., its $F(s)$, and when a value is crossed out this indicates that $F(s)$ was updated in this step. Figure 11 shows how CG-iLAO* behaves on the same problem. Importantly, the numbers at the bottom of each node have a different semantic: they now represent $V(s)$, and the solid and dotted circles represent states that are either inside or outside the current partial SSP. We use the same heuristic for both algorithms, namely h with:

$$h(s_0) = 7, h(s_1) = 5, h(s_2) = 6, h(s_3) = 9, h(s_4) = 6, h(s_5) = 7, h(s_6) = 8, h(s_7) = 5, h(s_8) = 6, h(s_9) = 9.$$

At an intuitive level, we see that the expansions of CG-iLAO* behave similarly to PEA*'s first-time expansions, and CG-iLAO*'s fixing of violated constraints lines up with PEA*'s re-expansions, so that steps (a), (b), (c), and (d) match between the two figures. Within these steps, the candidate plans are the same, with the exception that in step (b) CG-iLAO* considers an extra action and state, since PEA* immediately recognises that s_0 needs to be re-expanded, and CG-iLAO* needs to compute the policy first. To understand the relationship between CG-iLAO* and PEA*, we start by pointing out that iLAO* behaves similarly to A*: in each step of iLAO* in a deterministic problem, its current policy is a plan p in the partial SSP from s_0 to $s \in \widehat{\mathsf{G}}$ such that $C(p) + h(s)$ is minimal; and A* considers the state in its open list with minimal $f(s) = g(s) + h(s)$. With consistent tie breaking, it turns out that in matching steps this s is the same, and $C(p) = g(s)$, so that $V(s_0) = f(s)$. CG-iLAO* and PEA* inherit this relationship, and it only remains to understand how CG-iLAO*'s fixing of violated constraints corresponds to PEA*'s re-expansions. This does not line up exactly, as we see in our example in step (b), because PEA* immediately places a state s back into the open list if $f(s) > f(s')$ for its parent s', whereas CG-iLAO* does not take such shortcuts and computes the greedy policy. Nevertheless, the steps roughly correspond: CG-iLAO* recognises a violated constraint when a single action can be added to improve its candidate policy; and PEA*, re-expands s and brings in a new successor state s' with $F(s') = F(s)$, effectively improving the candidate plan by adding the missing action from s to s'. Thus, it seems clear that CG-iLAO* and PEA* have near-identical behaviour on deterministic problems.

We note that the definition of Yoshizumi, Miura, and Ishida [2000] is more general than what we have presented, because they allow $\text{SUCC}_{\leq C}$ for any cutoff $C \in \mathbb{R}_{\geq 0}$ and we only presented the special case $C = 0$. With the special case $\text{SUCC}_{\leq 0}$, PEA* is analogous to CG-iLAO* with the All-Greedy-Actions expansion (section 5.1). Larger values of C would translate to a CG-iLAO* expansion that adds actions that are within C of greedy. We investigated such expansions, but we did not find them theoretically insightful, and preliminary experiments were unpromising, so we have left them out of this paper.



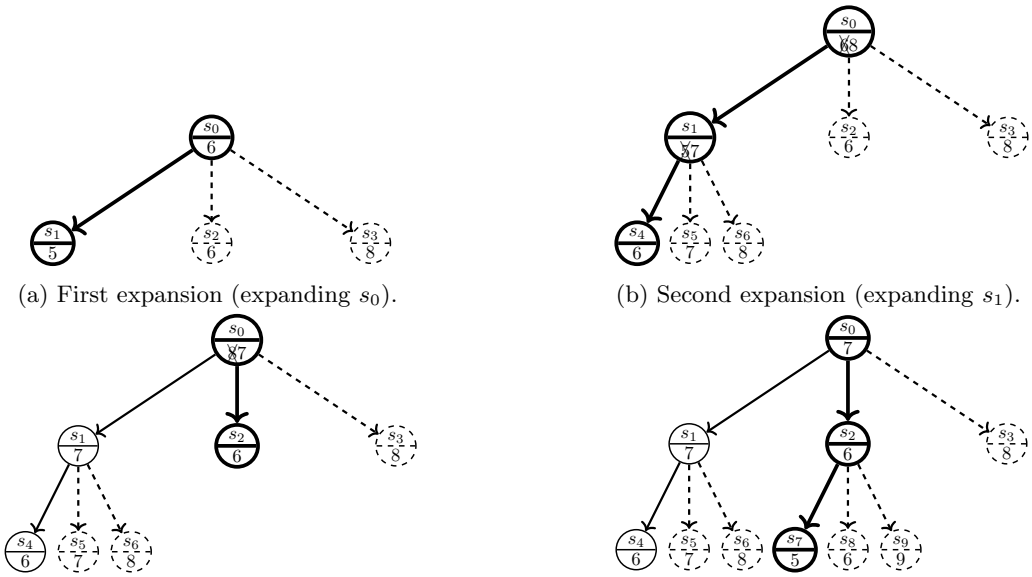
(a) First expansion (expanding s_0).

(b) Second expansion (expanding s_1).

(c) Third expansion (re-expanding s_0).

(d) Fourth expansion (expanding s_2).

Figure 10: PEA*'s behaviour (comparing CG-iLAO* and PEA*).



(a) First expansion (expanding s_0).

(b) Second expansion (expanding s_1).

(c) After fixing violated constraints from second expansion.

(d) Third expansion (expanding s_2).

Figure 11: CG-iLAO*'s behaviour on deterministic problem (comparing CG-iLAO* and PEA*). Thick lines indicate the greedy policy envelope.

7 Experiments

In this section, we experimentally determine the most efficient variants of CG-iLAO*, demonstrate how these variants compete with the state-of-the-art, and analyse these results in detail. We start by presenting the methodology of our experiments (section 7.1), describing the benchmark domains that we consider (section 7.2), and we explain the tables and plots that we use to present the results (section 7.3). Then, we compare the versions of CG-iLAO* to identify the best performers (section 7.4), and show that CG-iLAO*'s best performers are competitive with the state-of-the-art (section 7.5). We investigate CG-iLAO* further and show how many actions it is able to ignore (section 7.6), and how its mechanism for ignoring actions compares to action elimination (section 7.7).

7.1 Methodology

We compare different versions of CG-iLAO* with the state-of-the-art optimal heuristic-search algorithms iLAO* [Hansen and Zilberstein, 2001] and LRTDP [Bonet and Geffner, 2003b]. We will give more details about the algorithms we use in the relevant sections. Heuristics have a major impact on the performance of heuristic-search algorithms, so we consider the following suite of admissible heuristics from the literature: LMCut [Helmert and Domshlak, 2009]; ROC [Trevizan, Thiébaux, and Haslum, 2017]; and PDB [Klößner et al., 2021; Klößner and Hoffmann, 2021].

Following the original authors of ROC, on problems with dead ends we use ROC with $h^{\max}$ [Bonet and Geffner, 2000] as a dead-end detection mechanism. We only consider PDBs generated up to size 2, following the results of Klößner and Hoffmann [2021]. We convert SSPs into dead-end free SSPs with the fixed-penalty transformation [Trevizan, Teichteil-Königsbuch, and Thiébaux, 2017] with a penalty of

- $D = 10^4$ for Elev, $\square$TW, and Sched;
- $D = 10^7$ for PARC-R, PARC-N, and Sys;
- $D = 500$ for all other domains.

The domains with $D > 500$ require large penalties for different reasons. PARC-N and PARC-R have very large action costs. On problems such as $\square$TW, the optimal policy π^*'s probability of reaching a goal (without using give-up actions) is very low, with the effect that the cost of π^* is within ϵ of 500, so D needs to be increased. On problems such as Elev, an optimal policy requires the agent to repeat a sequence of actions many times before it reaches a goal, so that the optimal policy's cost exceeds 500. As the parameter for ϵ-consistency, we use $\epsilon = 0.0001$, and we set CG-iLAO*'s parameter $\eta = \epsilon$.

For each triple of algorithm, heuristic, and problem, we run it 5 times with different random seeds. These triples coupled with a random seed are called **instances**. We run multiple instances per problem because, even if all other parameters are fixed, the random seed can cause a significant amount of variance: for algorithms that rely on random choices this is obvious (e.g., LRTDP), but even for seemingly deterministic algorithms, random seeds have a significant impact on behaviour in practice (e.g., on tie breaking). To emphasise this point, we have observed a difference of over 1000 seconds on the same problem, algorithm, and heuristic, only changing the random seed. For each instance, the execution is cut off at 30 minutes of CPU time and 8GB of memory. The experiments were conducted in a cluster of Intel Xeon 3.2 GHz CPUs and each run used a single CPU core. The LP solver used for computing ROC was CPLEX version 20.1.

Our code, benchmarks, and results are available online [Schmalz and Trevizan, 2023]. We now give descriptions of our benchmark domains.

7.2 Benchmark Domains

We use the benchmark problems from Klößner et al. [2021], which is a compilation of domains from the International Probabilistic Planning Competitions (IPPC) that were run in 2004 [Younes et al., 2005], 2006 [Bonet and Givan, 2006], and 2008 [Bryce and Buffet, 2008]. These IPPCs encoded problems in Probabilistic PDDL [Younes et al., 2005]; we do not consider later competitions because they use RDDL [Sanner, 2010], which is incompatible with our heuristics. We consider three additional domains, which we also describe in this section. To reduce wasted compute time, we eliminate problems that are too difficult to solve. For a domain with problems enumerated from 1 to n, we consider the problems i to n too difficult if none of these were solved by LRTDP, iLAO*, or CG-iLAO* with any heuristics, given an extended deadline of 45 minutes. We now describe each domain and list each IPPC where the

domain appeared. We omit the domain "Boxworld" because all of its problems are too difficult, and we omit "Drive" and "Tireworld" because their problems are too easy: all algorithm configurations solved all problems in under 30 seconds (with a few exceptions).

Blocks World (BW) [IPPC 2004, 2006, 2008]. Deterministic Blocks World has featured in many of the (deterministic) International Planning Competitions (IPCs). A Blocks World problem consists of a table that has n unique blocks on it which can be stacked on top of each other, i.e., a block can be placed on top of any other, as long as there is no block on top of it already. The agent is presented with an initial configuration of blocks, and its task is to rearrange them into a specified configuration, e.g., block 1 should be on block 2, and block 2 should be on the table. To achieve this, the agent controls a "gripper" that can pick up one block at a time that has no other blocks on top of it, and then the gripper can put this block down on the table or on top of another block. The probabilistic version of Blocks World, which we consider, introduces to each of these actions a 0.25 probability that the block being handled slips out of the gripper and falls on the table. The probabilistic version also adds high-risk high-reward actions that enable the agent to handle a tower of two blocks, i.e., it can pick up a tower of two and then place it on the table or on top of a block, but it only succeeds with probability 0.1. In this domain, any action's effects can be reversed, e.g., if the agent tries to place block 1 but drops it, it can repeat the pick-up action until it is holding block 1, returning it to its previous state. Consequently, the problems have no dead ends.

Core Security Pentesting (Coresec) [Steinmetz, Hoffmann, and Buffet, 2016; Klößner et al., 2021]. This domain represents a penetration testing (pentesting) scenario, where the agent aims to exploit vulnerabilities in a network to compromise specific targets in the fewest possible steps. Each exploit can be used once, succeeding or failing with some probability. Thus, Coresec problems violate reachability (assumption 1), because any exploit that is necessary to breach the network may fail, and since the exploit can not be used again, the agent has no way to achieve its goal. Pentesting is a real-world application of SSPs, where real-world networks are modelled and presented to the agent; the agent's policies for breaching the network help administrators identify and address vulnerabilities. This particular domain is indirectly based on a test scenario from Core Security (http://www.coresecurity.com/). This domain has not featured in any IPPC.

Elevators (Elev) [IPPC 2006]. The agent is tasked with collecting all coins distributed within a building. The building can be visualised as a grid with y floors and x horizontal positions. Within a floor, the agent can freely walk between the horizontal positions. To move vertically between floors, the agent must use an elevator, which are available at particular horizontal positions. If the agent is at the same floor and horizontal position as an elevator, it can enter and move between floors as expected. The coins that the agent must collect are distributed throughout locations of the building, and the agent must try to navigate between them in the minimum number of steps. The probabilistic factor is that all locations except the ground floor have a "gate", and whenever the agent walks past such a gate it has a 0.5 probability of falling to the left-most part of the ground floor. Thus, the agent should avoid walking past gates unnecessarily. This domain has no dead ends because the agent can simply try to move to the next coin repeatedly, regardless of how many times it gets sent back to the left-most part of the ground floor. Note that policies can get very expensive, particularly if there is a coin to which the closest elevator is far away horizontally, since it forces the agent to walk past many gates.

Exploding Blocks World (ExBW) [IPPC 2004, 2006, 2008]. This domain, like BW, is also based on the deterministic Blocks World from IPC. Instead of the block slipping out of the agent's gripper, the probabilistic element is that each block is rigged with an explosive that can detonate once, instantly destroying the table or the block directly beneath it. When a block is placed, it detonates with probability 0.4 if placed on the table and 0.1 if placed on another block. Once the table or a block is destroyed, the agent can no longer interact with them, and if they were not in their goal position, the goal becomes unreachable. This means ExBW problems can have unavoidable dead ends: there may be a non-zero probability of reaching a state where a necessary block or the table have been destroyed. Under the fixed-penalty transformation, planners must avoid policies that incur the give-up penalty with high probability. Earlier versions of this domain had a bug where a block can be placed on top of itself; this bug is fixed in all of the problems we consider.

Probabilistic PARC Printer (PARC) [Trevizan, Thiébaux, and Haslum, 2017]. The deterministic PARC Printer domain from the IPC models a modular printer consisting of various components, where each page scheduled for printing must pass through multiple components in a specific order. The agent's task is to print a document of s pages in as few steps as possible, where each page can have different requirements which require different components in the printer. For example, the printer has coloured and black-and-white printing components, so pages in the document must be routed accordingly.

In the probabilistic variant, c components have a 0.1 probability of jamming, rendering the component unusable and requiring the affected page to be reprinted. The domain has two variants:

- PARC-R (with repair): Jammed components can be repaired and reused with a high cost.
- PARC-N (without repair): Jammed components remain permanently unusable.

This domain does not come from Klößner et al. [2021], and has not featured in any IPPC.

Random (Rand) [IPPC 2006]. In this domain, problems are generated randomly with no associated semantics. Rand is exceptional among the domains we consider, because it can have significantly more applicable actions in each state. The other domains have a number applicable actions on the order of 10^1, occasionally 10^2; whereas Rand has on the order of 10^4 (see table 6 for the number of states and actions considered in a particular Rand problem).

Schedule (Sched) [IPPC 2006, 2008]. This is a network scheduling scenario. There are m distinct packets being sent through the network, and n "classes" that can receive them. The agent's task is to ensure that each class receives and serves one or more packets, with the network staying alive. In each timestep, each class may randomly receive an incoming packet, which sets the packet's "time-to-live" to k timesteps. Then, the agent must choose whether the class should serve the packet, or hold onto it until the time-to-live expires, which causes the packet to be reclaimed, i.e., the current class no longer has access to the packet, and the packet gets randomly sent to one of the classes, as before. A complicating factor is that, whenever a packet gets reclaimed, the previous owner class has a small probability of locking up, which makes the network not alive, i.e., the agent fails its task. Different classes have different probabilities of locking up, so the agent should use the low-probability ones for reclaiming packets, and avoid reclaiming on the high-probability ones.

Search and Rescue (S&R) [IPPC 2008]. An autonomous helicopter must find a safe spot to land in order to save a survivor. There are n many locations, and it is not known a priori whether it is possible to land there or not. The agent must explore the locations – each explore action probabilistically determines whether it is possible to land at that location, or not. Once the agent finds a safe location to land, it can pick up and deliver the survivor to safety. The more actions the agent applies, the less likely the human survives. The agent's task is to maximise the human's chance of survival.

Sysadmin (Sys) [IPPC 2008]. The agent must manage a network of servers. In each timestep, a server has a probability of failing. If the server's neighbouring servers are all operational, then this server's probability of failing is low (probability 0.05), but if any of its neighbours have failed, then this server's probability of failing is significantly higher (probability 0.25). The only decision available to the agent is to reboot one problematic server per timestep, fixing it with probability 0.9. In the version we consider, all servers are initially broken, and the agent must make all servers operational while minimising the number of reboots over expectation. Note that Sys has zero-cost actions to handle the bookkeeping of which servers fail, which violates our assumption that costs are strictly positive. However, this is not an issue because we manually verified that there are no zero-cost cycles and assumption 2 holds.

Triangle Tire World (ΔTW) [IPPC 2008] The agent is presented with a network of locations arranged in a triangular layout with corners A, B, C, as in figure 12, and the agent's task is to drive from corner A to C. Each time the agent moves, the car gets a flat tyre with probability 0.5. If the car gets a flat tyre the agent must replace the flat with a spare tyre if it has one; when the agent has no spare tyre, the car is stuck, i.e., it is in a dead end with no actions available. The agent can only load spare tyres in specific locations (circles in figure 12), and can only keep one spare tyre at a time. This problem was introduced by Little and Thiebaux [2007] to be probabilistically interesting: if the probabilistic effects are ignored, the agent will take the direct path from A to C, which maximises the probability of encountering a dead end; whereas an optimal policy navigates from A to B to C, so that spare tyres are always available and it can reach the goal with probability 1. Thus, ΔTW has dead ends, but satisfies the reachability assumption (assumption 1) because a proper policy exists. ΔTW problems are specified by $n \in \mathbb{N}$. We show the layouts of ΔTW with $n = 1, 2, 3$ in figure 12; the problem with $n = 1$ can be considered a base case, and the problems with $n > 1$ are constructed by connecting n copies of the $n = 1$ problem along each edge of the triangle. With this construction, the state-space grows exponentially with respect to n, which is inconvenient for running experiments: it can happen that problem n is solved easily by all algorithms, but all algorithms time-out for $n + 1$. To address this shortcoming, we introduced ΔTW with Head-start [Schmalz and Trevizan, 2024], which gives a finer granularity of problem difficulty. This extension was not taken from Klößner et al. [2021], and has not featured in any IPPCs. A ΔTW with Head-start problem is defined by $\Delta\text{TW}(n, d)$, where $n \in \mathbb{N}$ denotes the problem size as before, and $d \in \mathbb{N}$ denotes that the agent is given a head-start along the edge AB, thus the agent has the initial location x-y with $x = (d + 1)$ and $y = 1$ in figure 12. For fixed n, d can take the values $0 \leq d \leq 2n$ where

- $d = 0$ defines the original ΔTW problem, with no head-start, where the agent starts at corner A,
- $d = 2n$ is the easiest variant of the problem where the agent is given the biggest head-start starting at corner B.

So, by starting at $d = 0$ we have the original ΔTW problem, and by increasing d we make the problem easier, giving us problems of intermediate difficulty between the standard ΔTW problems.

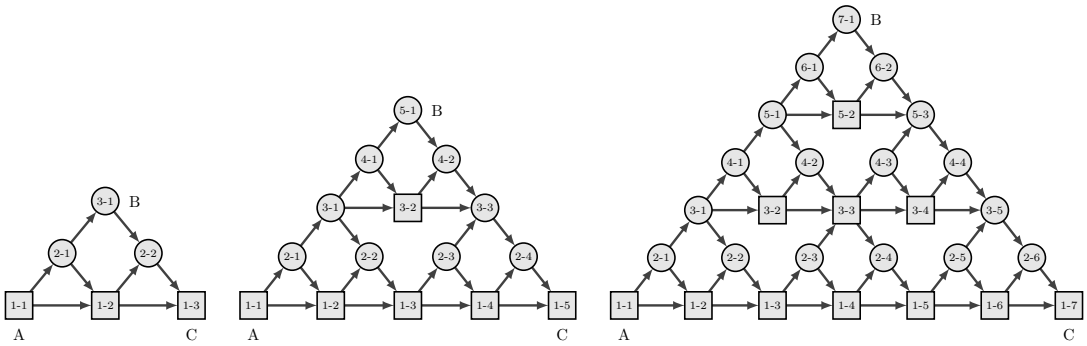


Figure 12: Triangle Tire World problems 1, 2, 3 (left to right).

Rectangle Tire World (□TW) [IPPC 2008]
The agent must drive from the bottom-left corner of a rectangular network of roads to the top-right, avoiding dangerous areas as much as possible. This domain is inspired by ΔTW [Little and Thiebaux, 2007], but with two key changes. First, as the name suggests, the road network is rectangular, rather than triangular, which makes it easier to encode large road systems compactly. On this network, the agent can drive to orthogonally adjacent cities, as well as diagonally adjacent ones. Second, the agent can no longer get a flat tyre, instead there are dangerous locations. If the agent enters a dangerous location, it crashes with certainty, killing the agent, so it can not achieve any goal. Entire rows and columns of the rectangular grid are specified to be unsafe. Driving diagonally is "somewhat dangerous," so anytime the agent performs this action it crashes with probability 0.2. The agent should minimise the number of drives between cities, while minimising the probability of crashing. Unsafe locations may be unavoidable in this domain, so □TW does not satisfy the reachability assumption (assumption 1). Moreover, randomly generated instances of this problem may be unsolvable in the sense that there is no way for the agent to reach the goal with probability > 0.

Zenotravel (Zeno) [IPPC 2004, 2006, 2008]. The agent must route aeroplanes to move people between cities. When grounded, aeroplanes board and debark passengers as expected. Aeroplanes can fly at a normal speed or "zoom" at a faster speed, allowing the vehicle to reach its destination faster at the cost of more fuel consumption. Once an aeroplane's fuel is exhausted, it must refuel. The boarding, debarking, flying to a destination (at any speed), and refuelling actions all succeed only with a specified probability, and upon failure nothing happens. This means the domain has no dead ends, since any action can be repeated until it succeeds.

7.3 Presentation of Results

In the following sections, we will be using similar tables and plots to summarise the results of our experiments:

- **Rank tables.** To obtain these tables, we start by creating a ranking of algorithms' runtimes for each instance, where the fastest algorithm on an instance gets rank 1, the second-fastest gets 2, and so on. If multiple algorithms tie with identical runtimes, then these algorithms receive the average rank of the group, e.g., if algorithm a is strictly fastest, algorithms b, c, d are tied second, and algorithm e is strictly slowest, then a receives rank 1, b, c, d all receive $\frac{2+3+4}{3} = 3$, and e gets 5. If an algorithm times out on an instance, it receives the largest rank for that instance, tied among all algorithms that timed out. Our rank tables present each algorithm's average (mean) rank over all relevant instances. Rankings are an intuitive way to determine which algorithms are fastest overall, and abstract away from the concrete times taken. Different rank tables *are not comparable*.
- **Coverage tables.** This is a standard way to summarise results in the planning community. For each domain, we present the number of instances that an algorithm and heuristic were able to solve

before timing out. This gives a good indication of how well an algorithm scales, again without worrying about concrete timings. Different coverage tables *are comparable*.

- **Cumulative plots.** These plots show how many instances an algorithm and heuristic was able to solve with a budget over time, number of Q-values, or number of calls to the heuristic. In the cumulative plot over time, every point x, y on a curve denotes that the relevant algorithm and heuristic solved y-many instances within x seconds, and the other plots are similar w.r.t. their respective budgets. Note that these plots effectively show total coverage, as in the coverage table, but over slices of time (or Q-values, or heuristic calls). Different cumulative plots *are comparable*.

- **Partial SSP size percentage tables.** These tables show how large an algorithm and heuristic's final partial SSP is, as a percentage of a baseline's final partial SSP sizes. Thus, the $|\widehat{\mathsf{S}}|\%$ and $|\widehat{\mathsf{A}}|\%$ give the number of states $|\widehat{\mathsf{S}}|$ and actions $|\widehat{\mathsf{A}}| = \sum_{s \in \widehat{\mathsf{S}}} |\widehat{\mathsf{A}}(s)|$ in the relevant algorithm's final partial SSP, as a percentage of the baseline's counterparts. We use a percentage w.r.t. the baseline rather than the entire SSP because we want to emphasise the difference from the baseline, rather than how much smaller the partial SSPs are than the original SSP.[8] Note that these tables can only feature iLAO* and CG-iLAO* type algorithms, because we have not defined partial SSPs for others. Different partial SSP size tables *are only comparable if the baseline is the same*. We consider a single baseline across all these tables, so they are indeed comparable.

- **Per-problem tables.** We present more detailed tables that give the coverage, average runtime, and average Q-values for each algorithm and heuristic per problem (rather than per domain) at https://github.com/schmlz/cgilao/tree/main/results and [Schmalz and Trevizan, 2023]. These are only given externally, but we will occasionally refer to them in this section.

7.4 What are the best versions of CG-iLAO*?

We investigate which settings of CG-iLAO* perform best, in order to eliminate unsuccessful variants and focus on only successful ones in the following sections. In particular, we are comparing CG-iLAO* with different expansion settings:

- CG-iLAO$^*_{\text{single}}$: expand with a single greedy action (Single-Greedy-Action from section 5.1).
- CG-iLAO$^*_{\text{tied}}$: expand with all tied-greedy actions (All-Greedy-Actions from section 5.1).
- CG-iLAO$^*_{\text{trial}}$: expand with rollout using trial (section 5.2). We use $t_{\max} = 100$, because preliminary results suggested this was a reasonable choice.
- CG-iLAO$^*_{\text{FF-AO}}$: expand with rollout using FF on all-outcomes determinisation (section 5.2).
- CG-iLAO$^*_{\text{FF-MLO}}$: expand with rollout using FF on most-likely-outcomes determinisation (section 5.2).

Table 1 is a rank table that presents the average ranks and 95% Confidence Interval (CI) for the variants of CG-iLAO*. For all heuristics, the ordering of ranks is identical: from 1 to 5, we have CG-iLAO$^*_{\text{tied}}$, CG-iLAO$^*_{\text{single}}$, CG-iLAO$^*_{\text{trial}}$, CG-iLAO$^*_{\text{FF-AO}}$, CG-iLAO$^*_{\text{FF-MLO}}$. The 95% CI shows that our experiments were unable to find a statistical difference between the average rank of the top two (CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$), but there is a statistically significant difference between the average ranks of the top two and CG-iLAO$^*_{\text{trial}}$, and between CG-iLAO$^*_{\text{trial}}$ and the variants with FF expansion. In the coverage table table 2, we see that CG-iLAO$^*_{\text{tied}}$, CG-iLAO$^*_{\text{single}}$, and CG-iLAO$^*_{\text{trial}}$ have similar total coverage, and the FF expansions are clustered together with significantly lower total coverage. Observe that with ROC, CG-iLAO$^*_{\text{single}}$ has a slightly higher total coverage than CG-iLAO$^*_{\text{tied}}$ by 7 instances, even though it has a lower average rank. This is not a contradiction, and happens because CG-iLAO$^*_{\text{tied}}$ tends to be faster on the other instances. Furthermore, the instances that CG-iLAO$^*_{\text{single}}$ solves and CG-iLAO$^*_{\text{tied}}$ fails to solve are not solved by the other algorithms either, so CG-iLAO$^*_{\text{tied}}$'s tied-last rank remains small there. Overall, the coverage supports that CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$ are the fastest variants. Finally, we consider the cumulative plot over time in figure 13. This clearly shows that CG-iLAO$^*_{\text{tied}}$, CG-iLAO$^*_{\text{single}}$, and CG-iLAO$^*_{\text{trial}}$ outperform the FF expansions with a significant margin over all heuristics. This performance gap becomes clear for easy problems that take around 50 seconds to solve, and is carried through

[8] Our SSPs are so large, that it is often impractical to compute the reachable states to calculate its size — recall that we use Probabilistic PDDL to represent our problems, and the state-space can be exponentially large w.r.t. the encoding.

ROC		PDB		LMCut	
CG-iLAO$^*_{\text{tied}}$	2.12±0.08	CG-iLAO$^*_{\text{tied}}$	2.24±0.08	CG-iLAO$^*_{\text{tied}}$	2.25±0.07
CG-iLAO$^*_{\text{single}}$	2.17±0.07	CG-iLAO$^*_{\text{single}}$	2.39±0.07	CG-iLAO$^*_{\text{single}}$	2.30±0.07
CG-iLAO$^*_{\text{trial}}$	2.68±0.09	CG-iLAO$^*_{\text{trial}}$	2.78±0.09	CG-iLAO$^*_{\text{trial}}$	2.63±0.08
CG-iLAO$^*_{\text{FF-AO}}$	3.96±0.08	CG-iLAO$^*_{\text{FF-AO}}$	3.75±0.08	CG-iLAO$^*_{\text{FF-AO}}$	3.80±0.07
CG-iLAO$^*_{\text{FF-MLO}}$	4.08±0.08	CG-iLAO$^*_{\text{FF-MLO}}$	3.85±0.08	CG-iLAO$^*_{\text{FF-MLO}}$	4.02±0.08

Table 1: Runtime ranking of CG-iLAO* variants within a specified heuristic (mean and 95% CI over all instances).

		BW	Corsec	Elev	ExBW	PARC-N	PARC-R	Rand	□TW	S&R	Sched	Sys	ΔTW	Zeno	total
	# of instances	110	35	75	105	30	30	75	70	25	45	20	40	45	705
ROC	CG-iLAO$^*_{\text{tied}}$	**105**	25	**75**	**105**	**30**	**25**	36	60	20	**45**	**20**	35	**40**	621
ROC	CG-iLAO$^*_{\text{single}}$	**105**	25	**75**	**105**	**30**	**25**	**43**	60	20	**45**	**20**	35	**40**	**628**
ROC	CG-iLAO$^*_{\text{trial}}$	**105**	25	70	**105**	**30**	24	35	60	21	**45**	**20**	35	**40**	615
ROC	CG-iLAO$^*_{\text{FF-AO}}$	100	20	59	95	**30**	20	35	60	20	35	10	25	30	539
ROC	CG-iLAO$^*_{\text{FF-MLO}}$	85	20	61	95	**30**	20	40	60	20	35	15	26	26	533
PDB	CG-iLAO$^*_{\text{tied}}$	90	**30**	**75**	90	0	0	30	60	**24**	30	**20**	**38**	**40**	527
PDB	CG-iLAO$^*_{\text{single}}$	90	**30**	**75**	90	0	0	30	60	20	30	**20**	37	**40**	522
PDB	CG-iLAO$^*_{\text{trial}}$	90	**30**	70	90	0	0	30	60	**24**	30	**20**	35	**40**	519
PDB	CG-iLAO$^*_{\text{FF-AO}}$	85	**30**	61	90	5	0	30	60	20	35	10	21	35	482
PDB	CG-iLAO$^*_{\text{FF-MLO}}$	85	**30**	61	90	5	0	30	60	20	35	15	22	30	483
LMCut	CG-iLAO$^*_{\text{tied}}$	45	25	70	101	**30**	20	20	**65**	**24**	**45**	**20**	30	25	520
LMCut	CG-iLAO$^*_{\text{single}}$	45	25	71	100	**30**	20	20	**65**	20	**45**	**20**	30	25	516
LMCut	CG-iLAO$^*_{\text{trial}}$	45	25	70	100	**30**	20	20	**65**	22	**45**	**20**	30	25	517
LMCut	CG-iLAO$^*_{\text{FF-AO}}$	45	20	63	95	25	15	25	**65**	20	35	10	20	20	458
LMCut	CG-iLAO$^*_{\text{FF-MLO}}$	45	20	62	95	22	15	25	**65**	20	35	15	20	15	454

Table 2: Coverage for each CG-iLAO* variant and considered heuristic over the benchmark domains. The highest coverage for each problem is marked with boldface.

all larger problems. Among the top three algorithms (CG-iLAO$^*_{\text{tied}}$, CG-iLAO$^*_{\text{single}}$, CG-iLAO$^*_{\text{trial}}$), it is difficult to determine a clear winner, but we observe that CG-iLAO$^*_{\text{trial}}$'s curve is underneath the curves of CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$ most of the time. Combining these observations, we declare that overall, CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$ are the tied-best variants of CG-iLAO*, CG-iLAO$^*_{\text{trial}}$ is a close second, and the FF expansions are last by a significant margin. We now compare the variants in more detail.

We have seen that CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$ are the best-performing variants of CG-iLAO*, with no significant difference between them, overall. However, we highlight that their behaviours are domain dependent, i.e., they have different strengths and weaknesses, and are not simply duplicates of each other. In particular, we focus on Rand with ROC, where CG-iLAO$^*_{\text{single}}$ has a higher coverage than CG-iLAO$^*_{\text{tied}}$ by 7 instances. Rand is an outlier among our domains, because it is the only one that is not inspired by real-world problems, and it has many more applicable actions per state than our other domains, on the order of 10^4 where the others have 10^1 or 10^2 (as we will see in table 6). The key difference is that CG-iLAO$^*_{\text{single}}$ has significantly fewer actions in its final partial SSP, by up to two orders of magnitude (we will see this later in section 7.6). Interestingly, this does not translate to savings in Q-value computations nor heuristic calls, but rather, the difference in performance comes from the overhead of managing the partial SSP's data-structures. Clearly, CG-iLAO$^*_{\text{tied}}$ is adding many tied-greedy actions to its partial SSP that are not necessary for finding an optimal policy, and CG-iLAO$^*_{\text{single}}$ avoids the issue in this case by adding only one greedy action in its expansion. CG-iLAO$^*_{\text{single}}$ also produces smaller partial SSPs on Rand with the other heuristics, but neither of the algorithms are able to solve the large problems with these heuristics, so the effect of differently sized partial SSPs is not noticeable. Over other domains, CG-iLAO$^*_{\text{single}}$ generally produces smaller partial SSPs than CG-iLAO$^*_{\text{tied}}$ (we investigate this further in section 7.6), but the size difference is not as extreme, and does not result in a significant performance difference. On the other hand, CG-iLAO$^*_{\text{tied}}$ has a higher coverage on S&R with LMCut

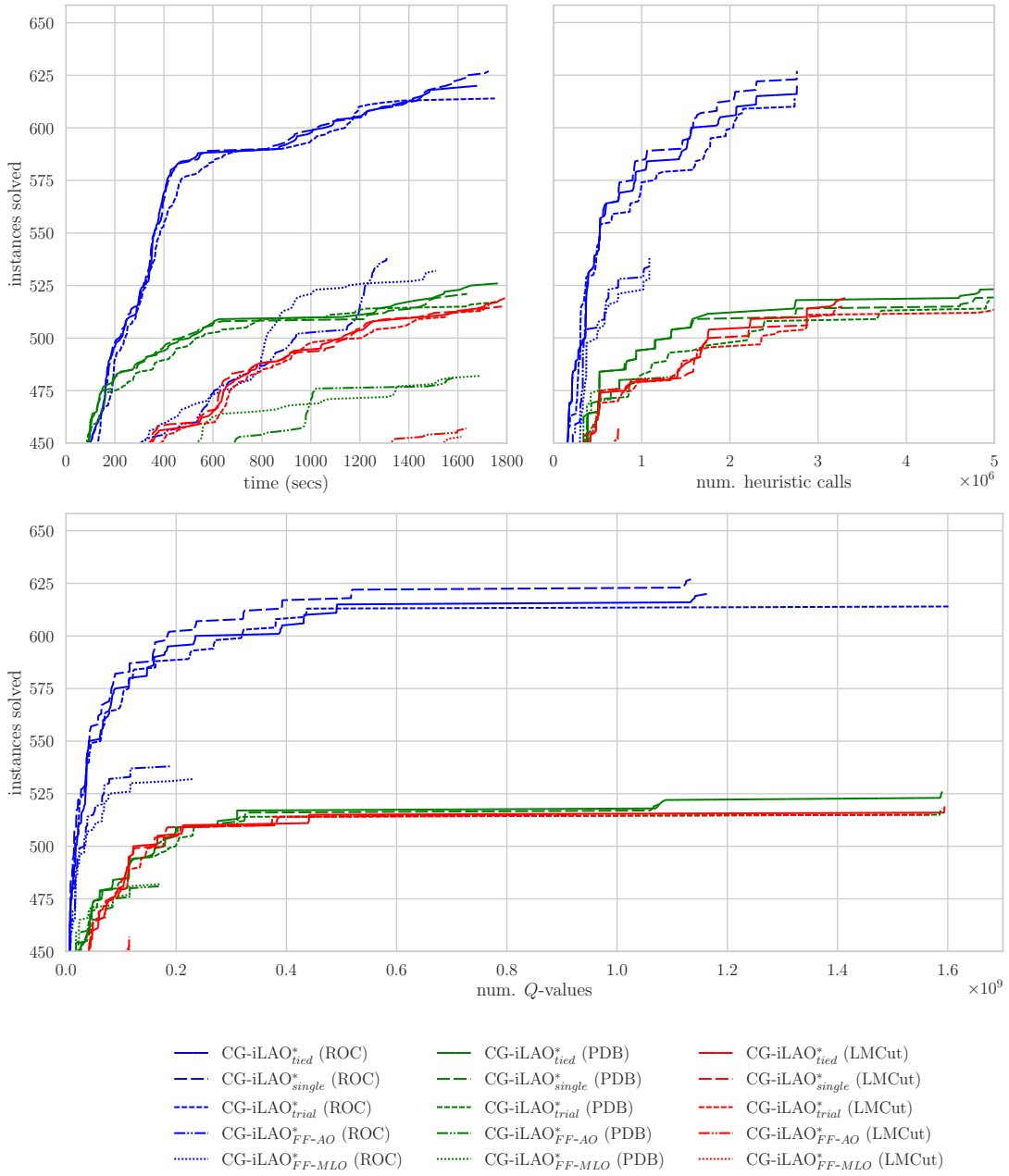


Figure 13: For each algorithm and heuristic, the cumulative plot of how many problems and seeds were solved w.r.t. time in seconds (top left), number of Q-values (top right), and number of calls to heuristic (bottom). To focus on where the algorithms are different we start the y-axis at 450 instances, and end the x-axis at 5×10^6 heuristic calls.

and PDB. As we will see in section 7.5, these are problems where iLAO* performs well, i.e., adding all applicable actions pays off. In these problems, the partial SSP sizes are similar, but CG-iLAO$^*_{\text{tied}}$ has fewer Q-values, presumably because it avoids the overhead of inserting missing tied-greedy actions. Outside these configurations, there is no significant difference between the two. Overall, this comparison shows that CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$, while similar, have different strengths and weaknesses that make them more or less suitable to specific problems.

Now, we investigate CG-iLAO$^*_{\text{trial}}$. Using the cumulative plots (figure 13), it is difficult to distinguish CG-iLAO$^*_{\text{trial}}$ from CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$ in terms of time and Q-values, but we can clearly see that it is less performant in terms of heuristic calls from 0.53×10^6 calls for ROC and PDB, and for LMCut it is again difficult to distinguish. The similarity in Q-values makes it clear that CG-iLAO$^*_{\text{trial}}$'s trials fail to add all useful actions in a way that saves Q-values, but it is also not adding a significant number of

unnecessary actions that would inflate the number of Q-values. The larger number of heuristic calls tells us that CG-iLAO$^*_{\text{trial}}$ considers more states than CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$, which makes sense because its trials deliberately probe states beyond the policy envelope. This is arguably a weakness because CG-iLAO$^*_{\text{trial}}$ needs to perform more heuristic computations without savings in Q-values. However, with the less informative LMCut heuristic, this weakness disappears and CG-iLAO$^*_{\text{trial}}$ has a similar number of heuristic calls to the others. This is because CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$ need to eventually expand the same states to prove optimality. In this case, CG-iLAO$^*_{\text{trial}}$ is more competitive, and even has a slightly higher total coverage than CG-iLAO$^*_{\text{single}}$ by 1 instance. This suggests that the trials do not pay off with informative heuristics: it is more efficient to expand greedily; whereas with less informative heuristics, it can be beneficial to run a trial before deeming an action useful.

Overall, the FF expansions performed poorly over our benchmarks, which can be attributed to two factors: the FF expansion often adds unneeded states and actions, and calling FF for each expansion is expensive. In the cumulative plots in figure 13, the FF expansions incur significantly more Q-value computations and heuristic calls, which confirms that more states and actions are being considered than necessary. Since Q-value computations and heuristic calls are the main computation expenses of CG-iLAO*, it is clear that the FF expansion variants can not keep up. This increase in calls happens because FF's plans do not coincide with an optimal policy, which is to be expected, given that it completely ignores probabilistic effects with the determinisations. These issues are only exacerbated by the additional cost of running FF for each expansion, rather than a single Bellman backup. The only case where FF expansions pay off in our experiments is in PARC-N with PDB. This is because the PDB heuristic is very uninformative on the PARC-N domain, and FF happens to expand states and actions that are sufficiently close to the optimal policy.

7.5 Comparing CG-iLAO* to the State-of-the-art

We compare CG-iLAO* with the state-of-the-art optimal heuristic-search algorithms iLAO* [Hansen and Zilberstein, 2001] and LRTDP [Bonet and Geffner, 2003b]. We only consider CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$, the best-performing variants of CG-iLAO* according to section 7.4. We consider two implementations of iLAO* in this experiment: iLAO* as implemented in the original paper [Schmalz and Trevizan, 2024], and CG-iLAO$^*_{\text{all}}$, which is CG-iLAO* with the complete Bellman expansion (see section 5.1). Although CG-iLAO$^*_{\text{all}}$ implements iLAO*, there are some subtle differences in implementation (discussed in section 3 and section 4) with large impacts on performance.

Using table 3, we can see that the ordering of the first three average ranks is consistent over all heuristics, from 1 to 3: CG-iLAO$^*_{\text{tied}}$, CG-iLAO$^*_{\text{single}}$, CG-iLAO$^*_{\text{all}}$. The remaining orderings are not consistent across heuristics: LRTDP and iLAO* alternate. The 95% CI reveals that our experiments can not distinguish between CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$ (consistent with section 7.4), nor can we distinguish between LRTDP and iLAO*. For the other ordering of rankings there is no overlap between 95% CIs: CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$ are ranked better than CG-iLAO$^*_{\text{all}}$, which is itself ranked better than iLAO* and LRTDP.

In terms of total coverage (table 4), CG-iLAO$^*_{\text{single}}$(ROC) is the leader with 628 instances solved, then with 621 instances solved, CG-iLAO$^*_{\text{tied}}$(ROC) and CG-iLAO$^*_{\text{all}}$(ROC) are close second. The next best is iLAO*(ROC) with 580 instances solved, 41 behind. We have observed that CG-iLAO$^*_{\text{tied}}$(ROC) has a lower rank than CG-iLAO$^*_{\text{all}}$(ROC), even though they have the same coverage. This is because for the instances that both algorithms solved, CG-iLAO$^*_{\text{tied}}$(ROC) tends to be faster, which can be seen in the cumulative plot over time (figure 14): the curve of CG-iLAO$^*_{\text{all}}$(ROC) lies underneath the curve of CG-iLAO$^*_{\text{tied}}$(ROC), i.e., CG-iLAO$^*_{\text{tied}}$(ROC) can solve more instances in the same time. Thus, CG-iLAO$^*_{\text{tied}}$(ROC) has better performance than CG-iLAO$^*_{\text{all}}$(ROC), because it has the same coverage but a lower rank over the solved instances. By similar argument CG-iLAO$^*_{\text{single}}$(ROC) also has better performance than CG-iLAO$^*_{\text{all}}$(ROC), strengthened by the fact that it has slightly higher coverage. Thus, CG-iLAO$^*_{\text{tied}}$(ROC) and CG-iLAO$^*_{\text{single}}$(ROC) are the best-performing algorithms.

This trend is similar over the other heuristics: CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$ are ranked first, and have the highest coverage; in this case, with strictly larger coverage than the next-best CG-iLAO$^*_{\text{all}}$, strengthening the arguments from before. The cumulative plot over time (figure 14) confirms this, because the curves for CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$ lie above the other algorithms' curves within the same heuristic. Thus, we declare CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$ as the best algorithms overall.

Now, we analyse the different performances in more detail. The fundamental operations that take up most of an algorithm's runtime are Q-values computations and calls to the heuristic. There are other operations that may take significant runtime, but they can be considered overhead, e.g., tracking the

ROC		PDB		LMCut	
CG-iLAO$^*_{\text{tied}}$	2.13±0.08	CG-iLAO$^*_{\text{tied}}$	2.27±0.08	CG-iLAO$^*_{\text{tied}}$	2.32±0.07
CG-iLAO$^*_{\text{single}}$	2.23±0.08	CG-iLAO$^*_{\text{single}}$	2.43±0.08	CG-iLAO$^*_{\text{single}}$	2.40±0.08
CG-iLAO$^*_{\text{all}}$	2.96±0.08	CG-iLAO$^*_{\text{all}}$	2.86±0.07	CG-iLAO$^*_{\text{all}}$	2.78±0.07
iLAO*	3.84±0.09	LRTDP	3.63±0.10	iLAO*	3.68±0.09
LRTDP	3.84±0.11	iLAO*	3.82±0.09	LRTDP	3.81±0.09

Table 3: Runtime ranking of CG-iLAO* and state-of-the-art within a specified heuristic (mean and 95% CI over all instances).

		BW	Coresec	Elev	ExBW	PARC-N	PARC-R	Rand	□TW	S&R	Sched	Sys	ΔTW	Zeno	total
	# of instances	110	35	75	105	30	30	75	70	25	45	20	40	45	705
ROC	LRTDP	55	25	**75**	**105**	**30**	20	33	55	**25**	**45**	**20**	35	20	543
ROC	iLAO*	**105**	25	69	100	**30**	25	38	48	20	35	**20**	30	35	580
ROC	CG-iLAO$^*_{\text{all}}$	**105**	25	70	**105**	**30**	**27**	35	60	**25**	**45**	**20**	35	39	621
ROC	CG-iLAO$^*_{\text{tied}}$	**105**	25	**75**	**105**	**30**	25	36	60	20	**45**	**20**	35	**40**	621
ROC	CG-iLAO$^*_{\text{single}}$	**105**	25	**75**	**105**	**30**	25	**43**	60	20	**45**	**20**	35	**40**	**628**
PDB	LRTDP	84	30	**75**	90	0	0	30	60	**25**	35	**20**	35	30	514
PDB	iLAO*	85	**35**	70	85	0	0	30	57	20	30	**20**	30	35	497
PDB	CG-iLAO$^*_{\text{all}}$	85	30	70	90	0	0	30	60	**25**	30	**20**	37	**40**	517
PDB	CG-iLAO$^*_{\text{tied}}$	90	30	**75**	90	0	0	30	60	24	30	**20**	**38**	**40**	527
PDB	CG-iLAO$^*_{\text{single}}$	90	30	**75**	90	0	0	30	60	20	30	**20**	37	**40**	522
LMCut	LRTDP	45	20	**75**	100	**30**	5	20	59	**25**	40	**20**	25	15	479
LMCut	iLAO*	45	25	70	95	**30**	20	20	55	20	35	**20**	25	25	485
LMCut	CG-iLAO$^*_{\text{all}}$	45	25	70	100	**30**	20	20	**65**	**25**	40	**20**	30	25	515
LMCut	CG-iLAO$^*_{\text{tied}}$	45	25	70	101	**30**	20	20	**65**	24	**45**	**20**	30	25	520
LMCut	CG-iLAO$^*_{\text{single}}$	45	25	71	100	**30**	20	20	**65**	20	**45**	**20**	30	25	516

Table 4: Coverage for CG-iLAO* and state-of-the-art with each considered heuristic over the benchmark domains. The highest coverage for each problem is marked with boldface.

current partial SSP and ensuring the applicable actions are up-to-date. It is important to recognise that timings are dependent on many factors, e.g., how optimised the implementation is for the hardware in use, whereas the number of Q-values and heuristic calls is a more robust measure. In that sense, the following discussions are more insightful than discussing the fastest algorithm.

We start by looking at the number of Q-values. Focusing on ROC, we see in the cumulative plot over Q-values in figure 14 that CG-iLAO$^*_{\text{single}}$(ROC) and CG-iLAO$^*_{\text{tied}}$(ROC) require the fewest Q-values to solve their instances. Between CG-iLAO$^*_{\text{single}}$(ROC) and the third-best CG-iLAO$^*_{\text{all}}$(ROC), the lead is clear almost immediately, and it maintains a large gap over all problems. From 0.04×10^9 to 0.64×10^9 Q-values, CG-iLAO$^*_{\text{tied}}$(ROC) leads over CG-iLAO$^*_{\text{all}}$(ROC) with a gap of around 5-15 instances to CG-iLAO$^*_{\text{all}}$(ROC). From 0.64×10^9 to 1.13×10^9 Q-values CG-iLAO$^*_{\text{all}}$(ROC) catches up to CG-iLAO$^*_{\text{tied}}$(ROC), and then CG-iLAO$^*_{\text{tied}}$(ROC) makes a jump of 4 instances; this is happening because two problems (Elev 04-13 and S&R 05) require significantly more Q-values to solve than the rest. In fact, this emphasises how many fewer Q-values CG-iLAO$^*_{\text{tied}}$(ROC) requires, because it solves these large problems with 1.16×10^9 Q-values, and CG-iLAO$^*_{\text{all}}$(ROC) requires 1.67×10^9. The gap between the CG-iLAO* variants and other algorithms is even more significant. Thus, CG-iLAO$^*_{\text{tied}}$(ROC) computes the fewest Q-values. For the other heuristics, we observe a similar trend, but more extreme in favour of CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$. Thus, the top two CG-iLAO* variants are the algorithms that compute the fewest Q-values.

To investigate the number of heuristic calls, we use the cumulative plot over heuristic calls (figure 14). We see that CG-iLAO$^*_{\text{tied}}$, CG-iLAO$^*_{\text{single}}$, and CG-iLAO$^*_{\text{all}}$ are decisively ahead of iLAO* and LRTDP over all heuristics, in as few calls as 0.5×10^6. Between these top three, there is no consistent trend over heuristics.

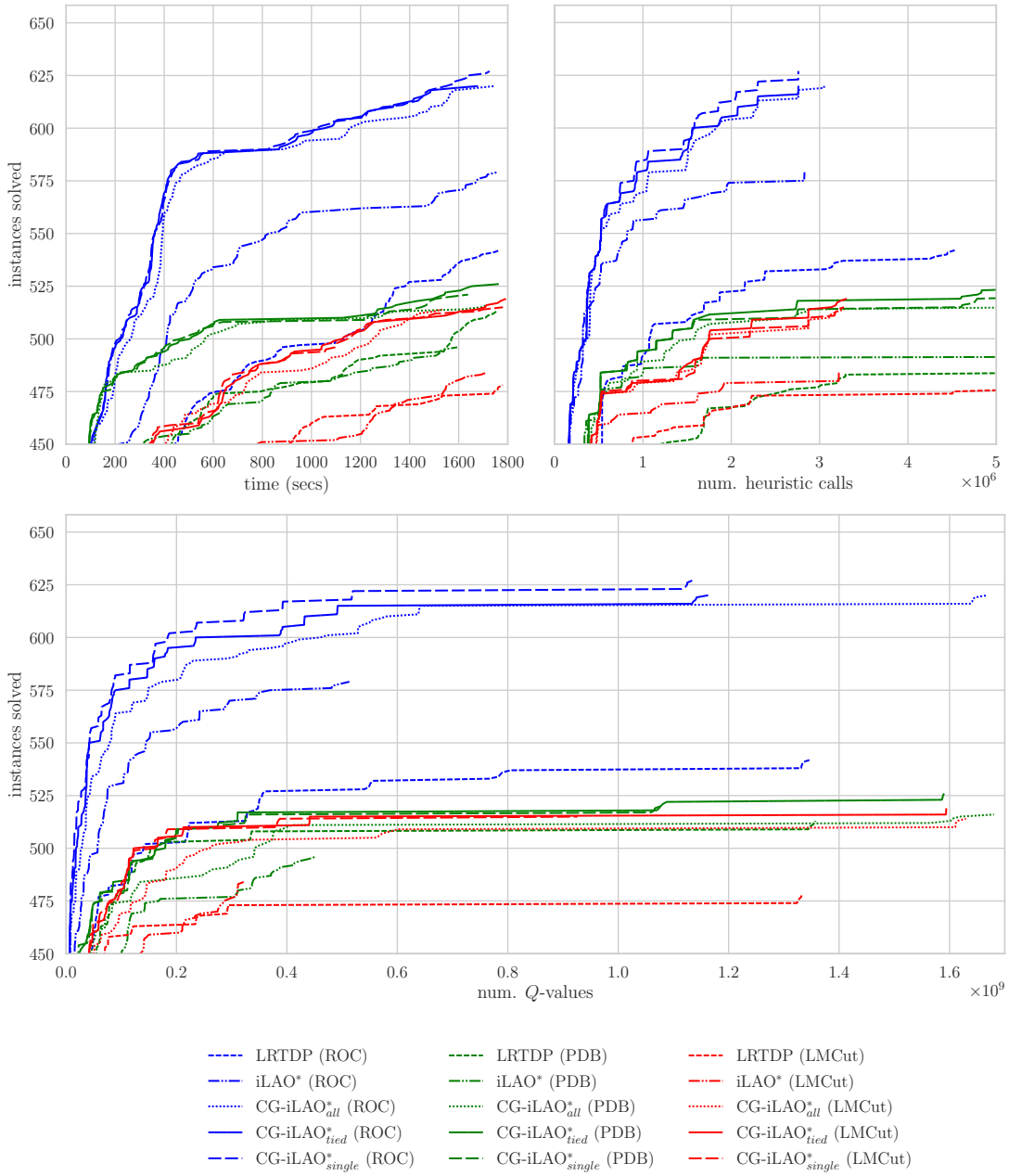


Figure 14: For each algorithm and heuristic, the cumulative plot of how many problems and seeds were solved w.r.t. time in seconds (top left), number of Q-values (top right), and number of calls to heuristic (bottom). To focus on where the algorithms are different we start the y-axis at 450 instances, and end the x-axis at 5×10^6 heuristic calls.

7.6 How many actions does CG-iLAO* ignore?

The key idea of CG-iLAO* is that it ignores inactive actions and only adds actions when they are needed, with the aim of producing a smaller partial SSP. In this section, we confirm experimentally that indeed CG-iLAO* has smaller partial SSPs than iLAO*, and quantify by how much, i.e., how many actions CG-iLAO* ignores.

For CG-iLAO$^*_{\text{tied}}$, CG-iLAO$^*_{\text{single}}$, and iLAO*, we present the number of states and actions in their final partial SSPs as a percentage of the number of states and actions in CG-iLAO$^*_{\text{all}}$'s final partial SSP in table 5. The table gives for each domain, the percentages averaged over instances where the algorithm being considered and the baseline both terminate. Since we require the baseline to terminate, we use CG-iLAO$^*_{\text{all}}$ rather than iLAO*, because it solves more instances, letting us consider more instances. In

		CG-iLAO$^*_{\text{tied}}$		CG-iLAO$^*_{\text{single}}$		iLAO*			
		$\lvert\widehat{\mathsf{S}}\rvert$%	$\lvert\widehat{\mathsf{A}}\rvert$%	$\lvert\widehat{\mathsf{S}}\rvert$%	$\lvert\widehat{\mathsf{A}}\rvert$%	$\lvert\widehat{\mathsf{S}}\rvert$%		$\lvert\widehat{\mathsf{A}}\rvert$%	
BW	ROC	91.8±1.3	40.8±1.2	98.7± 3.5	33.2±2.8	86.8±	1.5	85.3±	1.5
	LMCut	97.5±0.6	45.5±1.4	97.6± 0.5	42.5±2.0	88.3±	0.8	84.7±	1.2
	PDB	90.9±1.4	39.7±0.9	90.2± 3.0	30.9±1.6	85.9±	1.4	84.1±	1.5
Coresec	ROC	100.2±0.3	33.6±1.9	100.2± 0.4	20.5±1.1	85.7±	0.6	142.0±	1.8
	LMCut	100.0±0.1	39.0±1.9	99.9± 0.1	24.9±1.1	95.8±	0.7	193.4±	2.2
	PDB	99.6±0.5	29.0±1.6	100.0± 0.4	22.2±1.6	87.1±	0.5	113.9±	1.5
Elev	ROC	97.6±1.1	60.5±1.6	98.1± 1.3	53.3±2.2	94.9±	1.1	94.9±	1.1
	LMCut	93.1±3.0	45.0±3.1	96.0± 2.1	40.4±3.0	90.2±	1.4	90.3±	1.4
	PDB	97.0±1.5	56.9±1.7	98.2± 2.0	50.1±2.2	92.3±	1.6	92.5±	1.6
ExBW	ROC	95.8±1.4	44.8±1.5	99.9± 3.2	39.6±1.7	97.9±	0.7	103.5±	1.8
	LMCut	93.9±2.0	39.2±2.1	99.2± 3.3	37.1±1.8	97.8±	0.9	102.2±	2.1
	PDB	95.7±1.9	53.9±3.4	98.7± 3.4	49.9±3.7	97.6±	1.1	101.7±	2.0
PARC-N	ROC	96.6±0.6	44.7±0.9	97.0± 0.6	32.7±0.9	87.5±	1.1	92.7±	1.1
	LMCut	93.5±0.9	44.0±1.2	94.5± 0.8	35.2±1.3	75.3±	1.5	80.9±	1.8
PARC-R	ROC	99.6±0.2	44.1±1.0	99.6± 0.2	43.0±1.0	86.9±	1.7	86.6±	1.6
	LMCut	101.1±0.6	51.9±1.0	101.9± 0.9	50.1±1.2	95.7±	1.0	95.8±	0.9
Rand	ROC	89.8±8.6	9.1±1.8	93.7±11.3	0.9±0.3	96.3±	3.1	96.3±	3.1
	LMCut	100.0	9.9±3.9	100.0	1.8±0.7	100.0		100.0	
	PDB	95.1±5.3	17.0±8.1	62.5±16.7	1.0±0.5	2369.7±3093.0		3987.8±5296.5	
□TW	ROC	100.0±0.2	31.5±0.8	100.0± 0.2	33.1±1.6	100.0		126.0±	12.9
	LMCut	99.9±0.2	30.9±0.9	100.0± 0.1	32.8±1.8	99.7±	0.3	125.8±	12.9
	PDB	97.9±1.0	31.0±1.5	98.8± 1.1	31.5±1.8	100.0±	1.4	127.0±	11.3
S&R	ROC	100.0±0.1	91.6±0.8	100.2± 0.1	91.4±0.7	101.0±	0.2	104.5±	0.4
	LMCut	100.1±0.1	92.0±0.8	100.0± 0.1	91.3±0.7	100.9±	0.2	104.5±	0.4
	PDB	100.3±0.1	92.2±0.8	100.5± 0.2	91.4±0.7	101.2±	0.2	104.2±	0.3
Sched	ROC	100.7±0.3	78.1±0.7	100.9± 0.5	50.7±1.1	99.0±	0.8	100.0±	0.1
	LMCut	99.0±0.5	73.8±3.2	101.4± 0.8	50.2±1.2	99.3±	0.6	100.2±	0.1
	PDB	100.0	79.6±0.5	100.0	51.1±1.3	100.0		100.0	
Sys	ROC	100.0±0.1	99.5±0.3	100.0	97.4±0.3	100.0±	0.1	100.0	
	LMCut	100.0	99.5±0.4	100.0	97.3±0.4	100.1±	0.1	100.1	
	PDB	100.0	100.0	100.0	98.4±0.4	100.0		100.0	
ΔTW	ROC	98.0±0.8	64.4±0.8	97.2± 1.0	63.8±0.9	99.7±	0.4	99.8±	0.4
	LMCut	100.7±0.6	68.6±1.5	100.0± 0.9	67.0±1.5	98.0±	0.9	98.3±	0.9
	PDB	98.2±0.8	65.7±0.7	98.0± 0.8	65.5±0.7	99.6±	0.4	99.8±	0.5
Zeno	ROC	88.4±0.5	37.1±0.6	86.7± 0.3	30.1±0.8	80.5±	1.1	77.2±	1.0
	LMCut	89.0±1.2	34.3±0.4	88.5± 1.3	31.6±0.5	78.0±	1.4	74.2±	1.1
	PDB	89.9±1.0	44.0±1.3	88.0± 0.9	34.0±1.1	82.1±	1.6	79.1±	1.4

Table 5: Size of each algorithm's final partial SSP, as a percentage of CG-iLAO$^*_{\text{all}}$'s final partial SSP. These values are means over instances where the considered algorithm and CG-iLAO$^*_{\text{all}}$ both terminated.

table 5 we see that the number of states in the final partial SSPs $|\widehat{\mathsf{S}}|$%, is around 100% for all algorithms and heuristics. This is what we expect, because CG-iLAO* ignores actions, and has no mechanism for ignoring more states than iLAO*. We acknowledge the notable outliers at Rand with PDB, where CG-iLAO$^*_{\text{single}}$ has less than 60% and iLAO* has over 2000% of CG-iLAO$^*_{\text{all}}$'s states, but this is a unique exception which does not contribute much.

More importantly, the number of actions in the final partial SSPs $|\widehat{\mathsf{A}}|$%, is always smaller than 101% for both CG-iLAO* variants, and below 50% in most problems. This confirms that indeed CG-iLAO* is able to leave out many actions that iLAO* considers. Exceptions to this are S&R and Sys, where the CG-iLAO* variants have over 90% of CG-iLAO$^*_{\text{all}}$'s actions. Note that this is reflected by CG-iLAO$^*_{\text{all}}$ having higher coverage on S&R than the CG-iLAO* variants. It is not entirely clear why CG-iLAO* has such large partial SSPs on these problems, but we conjecture that it is because the heuristics are not informative enough, and with more informative heuristics CG-iLAO* would be able to ignore more actions. Between CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$, we see that CG-iLAO$^*_{\text{single}}$ tends to have fewer actions in its final partial SSP. This is unsurprising, because CG-iLAO$^*_{\text{tied}}$ can add unnecessary actions in its expansion, whereas CG-iLAO$^*_{\text{single}}$ only adds actions that are necessary at some point. Note, that this difference in partial SSP size is not reflected in the number of Q-values computed, as discussed in section 7.4, because

CG-iLAO$^*_{\text{single}}$ incurs additional overhead in checking which actions should be added. The exception is again Rand where we have seen CG-iLAO$^*_{\text{single}}$(ROC) obtains highest coverage, precisely because its final partial SSP is significantly smaller.

To make the final partial SSP sizes more concrete, we present the actual final partial SSP sizes for select problems in table 6. This gives a sense of scale of how many actions are being saved, bearing in mind that we picked smaller problems where all algorithms had close-to-complete coverage.

In summary, the CG-iLAO* variants often have significantly smaller final partial SSPs than the iLAO* variants, confirming that our mechanism for ignoring actions has a significant impact.

To gain a better insight of how many actions are ignored by CG-iLAO*, we investigate how dense CG-iLAO*'s partial SSPs are, i.e., how many actions were added to CG-iLAO*'s partial SSP, out of the possible actions that could have been added. We define the density of state s as $|\widehat{\mathsf{A}}(s)|/|\mathsf{A}(s)|$. To start, we restrict our attention to a single representative problem from each domain, solved by CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$ with ROC. The representative problems were chosen to be the largest problem that was still solved in time (or some large problem that was solved in time, if there is no clear hierarchy). We give cumulative plots over density for these problems in figure 15, where each point (x, y) represents that a proportion x of states in the final partial SSP has actions with density y or less, e.g., $(0.5, 0.3)$ tells us that 50% of the partial SSP's states have a density of 0.3 or less. Note that the proportion x of states for CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$ are evaluated w.r.t. to their respective final partial SSP sizes, i.e., they may be referring to a different number of states. However, we have seen in section 7.6 that the number of states in their partial SSPs is similar, so the curves are roughly comparable. A plot that grows quickly at the start, with the extreme case $y = 1$ for $x \geq 0$ (shaped ⌜), indicates that all states have high density. Conversely, if the plot grows slowly and then jumps up for large x, with the extreme case $y = 0$ for $x < 1$ (shaped ⌟), it means that most states have low density. Another way to understand this curve is that, if the area under the curve is large then most states are dense, and if the area under the curve is small then most states are not dense. Variants of iLAO* have the extreme ⌜-shaped curve because iLAO* adds all actions, resulting in density 1 for all states.

Looking at figure 15, we see that the density profiles vary dramatically across problems. For problems such as Coresec, Rand, and □TW, both CG-iLAO* variants have mostly low-density states, which is reflected by their small final partial SSPs in table 5. In contrast, Elev, S&R, and Sys have mostly high-density states, which is respectively reflected by relatively large final partial SSPs in table 5. This is particularly extreme in Sys, where the CG-iLAO* variants have over 99% of the number of actions as CG-iLAO$^*_{\text{all}}$. CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$ have significantly different profiles for BW, Coresec, ExBW, PARC-N, Rand, and Sched; and nearly identical profiles for the other problems.This is generally reflected by the differences in partial SSP sizes for ROC in table 5. This confirms, again, that CG-iLAO$^*_{\text{single}}$ is generally able to ignore more actions than CG-iLAO$^*_{\text{tied}}$. Otherwise, we can only conclude that CG-iLAO* equipped with ROC is able to ignore many actions on some problems, and very few on others.

Recall theorem 6, which tells us that CG-iLAO$^*_{\text{single}}$ with the perfect heuristic adds only the actions necessary to define its optimal policy, i.e., with the perfect heuristic its density plots would have $y = 1$ for $x \geq 0$. In that sense, the density of CG-iLAO$^*_{\text{single}}$ is heuristic-dependent, rather than problem-dependent. Consequently, figure 15 can be interpreted as a measure of heuristic quality. This motivates that CG-iLAO* fails to produce small partial SSPs on some problems due to uninformative heuristics, rather than any property of the problems themselves.

To get a higher-level picture we now present a summary of density over entire domains in figure 16. Here, we generate the density curves for each algorithm, heuristic, and problem triple (averaged over instances in the problem), and then compute the area under these curves (AUC); we call this value the density AUC. An instance with mostly dense states and therefore a ⌜-shaped curve will have AUC close to 1, and conversely an instance with few dense states and a ⌟-shaped curve will have AUC close to 0. We aggregate density AUC over domains with box-and-whisker plots to show the distribution over different problems within the domain. These results are generally consistent with the previous discussion and let us make similar conclusions: the density varies a lot over different domains, and generally CG-iLAO$^*_{\text{single}}$ tends to have final partial SSPs with lower density that CG-iLAO$^*_{\text{tied}}$. We note that in some domains the densities are very similar across problems within the domain, e.g., PARC-N, PARC-R, S&R; but in others the densities are varied, e.g., Elev, ExBW, Sched. This does not give any deep insight, only that some of our benchmarks have problems of similar structure within a domain, and others do not. One must be careful comparing the behaviour over different heuristics, because they are associated with different coverages and therefore have different datasets; nevertheless, it seems that the behaviour does not change significantly over different heuristics.

			CG-iLAO$^*_{\text{tied}}$			CG-iLAO$^*_{\text{single}}$			iLAO*		CG-iLAO$^*_{\text{all}}$	
			$\lvert\widehat{\mathsf{S}}\rvert$	$\lvert\widehat{\mathsf{A}}\rvert$	$\lvert\widehat{\mathsf{A}}^{\max}\rvert$	$\lvert\widehat{\mathsf{S}}\rvert$	$\lvert\widehat{\mathsf{A}}\rvert$	$\lvert\widehat{\mathsf{A}}^{\max}\rvert$	$\lvert\widehat{\mathsf{S}}\rvert$	$\lvert\widehat{\mathsf{A}}\rvert$	$\lvert\widehat{\mathsf{S}}\rvert$	$\lvert\widehat{\mathsf{A}}\rvert$
BW	04-10	ROC	38 197±187	107 485± 619	264 000± 1337	38 041±160	69 964± 384	263 097± 1068	35 601± 245	249 229± 1769	42 269± 332	310 921± 2560
		PDB	38 604±308	112 018± 782	266 356± 2200	38 141±274	70 045± 740	263 724± 1758	35 514± 198	248 260± 1479	42 229± 398	310 213± 3083
	08-11	ROC	12 961±776	42 333±2215	107 413± 6639	12 530±402	22 106± 549	104 209± 3392	12 172± 797	102 326± 6804	14 114± 907	116 851± 7586
	08-12	ROC	12 961±776	42 333±2215	107 413± 6639	12 530±402	22 106± 549	104 209± 3392	12 172± 797	102 326± 6804	14 114± 907	116 851± 7586
Elev	04-15	ROC	511 935±243	2 236 293±8866	3 063 040± 1474	512 158±286	2 143 683±7449	3 064 324± 1739	513 039± 171	3 069 532± 959	515 568± 988	3 084 272± 5788
		LMCut	441 272±717	1 652 518±6622	2 649 696± 4449	441 162±810	1 551 021±8734	2 648 887± 4855	425 933±3021	2 557 899±18 032	452 195±3016	2 714 150±17 807
		PDB	506 141±399	2 134 977±8328	3 028 903± 2315	506 320±222	2 045 625±6661	3 029 960± 1276	504 455± 808	3 018 725± 4709	510 224±1157	3 052 950± 6776
Rand	04-07	ROC	8± 5	40± 31	707± 582	8± 5	12± 9	707± 582	8± 5	707± 582	8± 5	707± 582
		LMCut	3	54± 49	264± 52	3	4± 1	264± 52	3	264± 52	3	264± 52
		PDB	4	16± 2	265± 52	4± 1	5± 1	265± 52	4	265± 52	4	265± 52
	04-09	ROC	4± 1	506± 33	3690± 954	5± 2	6± 5	4731± 2508	4± 1	3690± 954	4± 1	3690± 954
		PDB	10± 2	468± 50	9712± 2326	10± 5	15± 9	9790± 4901	10± 2	9448± 2300	10± 2	9448± 2300
	04-13	ROC	25± 5	7634±1621	46 300± 9419	25± 6	42± 11	45 070±10 729	26± 4	48 143± 7394	26± 4	48 143± 7394
		LMCut	7	845± 3	11 992± 137	7	10	11 992± 137	7	12 001± 42	7	12 001± 42
		PDB	128± 12	37 172±4838	208 000±20 635	98± 23	171± 41	162 899±39 785	197± 29	325 885±48 159	197± 29	325 905±48 161
S&R	08-03	ROC	9910± 8	35 309± 32	39 577± 4	9921± 13	35 132± 30	39 583± 7	10 022± 15	39 633± 7	9904± 5	38 081± 2
		LMCut	9914± 8	35 152± 12	39 580± 4	9913± 4	35 088± 27	39 579± 2	10 024± 24	39 635± 12	9919± 14	38 088± 7
		PDB	9890± 10	35 316± 34	39 573± 5	9894± 6	35 153± 20	39 575± 3	9993± 19	39 625± 9	9863± 2	38 102± 1
	08-04	ROC	113 810± 17	527 263± 203	578 498± 8	113 820± 47	525 092± 167	578 502± 23	114 249± 76	578 717± 38	113 769± 30	560 917± 15
		LMCut	113 817± 22	525 083± 43	578 501± 11	113 828± 13	524 661± 99	578 506± 7	114 234± 78	578 709± 39	113 799± 26	560 932± 13
		PDB	113 763± 32	527 146± 311	578 481± 16	113 766± 44	524 952± 158	578 483± 22	114 178± 78	578 689± 39	113 624± 14	560 916± 7
	08-05	ROC	–	–	–	–	–	–	–	–	1 251 364± 52	7 485 099± 26
		LMCut	1 251 642± 57	7 078 350± 557	7 682 167± 29	–	–	–	–	–	1 251 455± 60	7 485 144± 30
		PDB	1 251 444± 65	7 109 673± 538	7 682 078± 33	–	–	–	–	–	1 250 771± 40	7 484 911± 20
Sys	08-02	ROC	1541	1725	1729	1541	1697± 4	1729	1539	1727	1541	1729
		LMCut	1580	1768	1768	1580	1738± 2	1768	1584	1772	1580	1768
		PDB	1592	1780	1780	1592	1760± 3	1780	1592	1780	1592	1780
	08-03	ROC	4676	5231	5231	4676	5130± 7	5231	4677	5232	4676	5231
		LMCut	4692	5247	5247	4692	5135± 6	5247	4693± 1	5248± 1	4692	5247
		PDB	4699± 1	5254± 1	5254± 1	4699± 1	5159± 8	5254± 1	4703± 2	5258± 2	4699± 1	5254± 1
	08-04	ROC	33 016± 1	37 041± 1	37 041± 1	33 017	35 730± 29	37 042	33 038± 2	37 063± 2	33 016± 1	37 041± 1
		LMCut	33 021± 4	37 046± 4	37 046± 4	33 023± 4	35 680± 19	37 048± 4	33 042± 2	37 067± 2	33 022± 3	37 047± 3
		PDB	33 074± 4	37 099± 4	37 099± 4	33 071± 6	36 030± 28	37 096± 6	33 096± 1	37 121± 1	33 074± 4	37 099± 4

Table 6: The final partial SSP sizes of CG-iLAO* and iLAO*. The number of states in the final partial SSP $|\widehat{\mathsf{S}}|$, the number of actions in the final partial SSP $|\widehat{\mathsf{A}}| = \sum_{s\in\widehat{\mathsf{S}}} |\widehat{\mathsf{A}}(s)|$, and for CG-iLAO* the number of actions available to its partial states $|\widehat{\mathsf{A}}^{\max}| = \sum_{s\in\widehat{\mathsf{S}}} |\mathsf{A}(s)|$. We do not count give-up actions from the fixed-penalty transformation. We restrict the results to the larger problems of each domain.

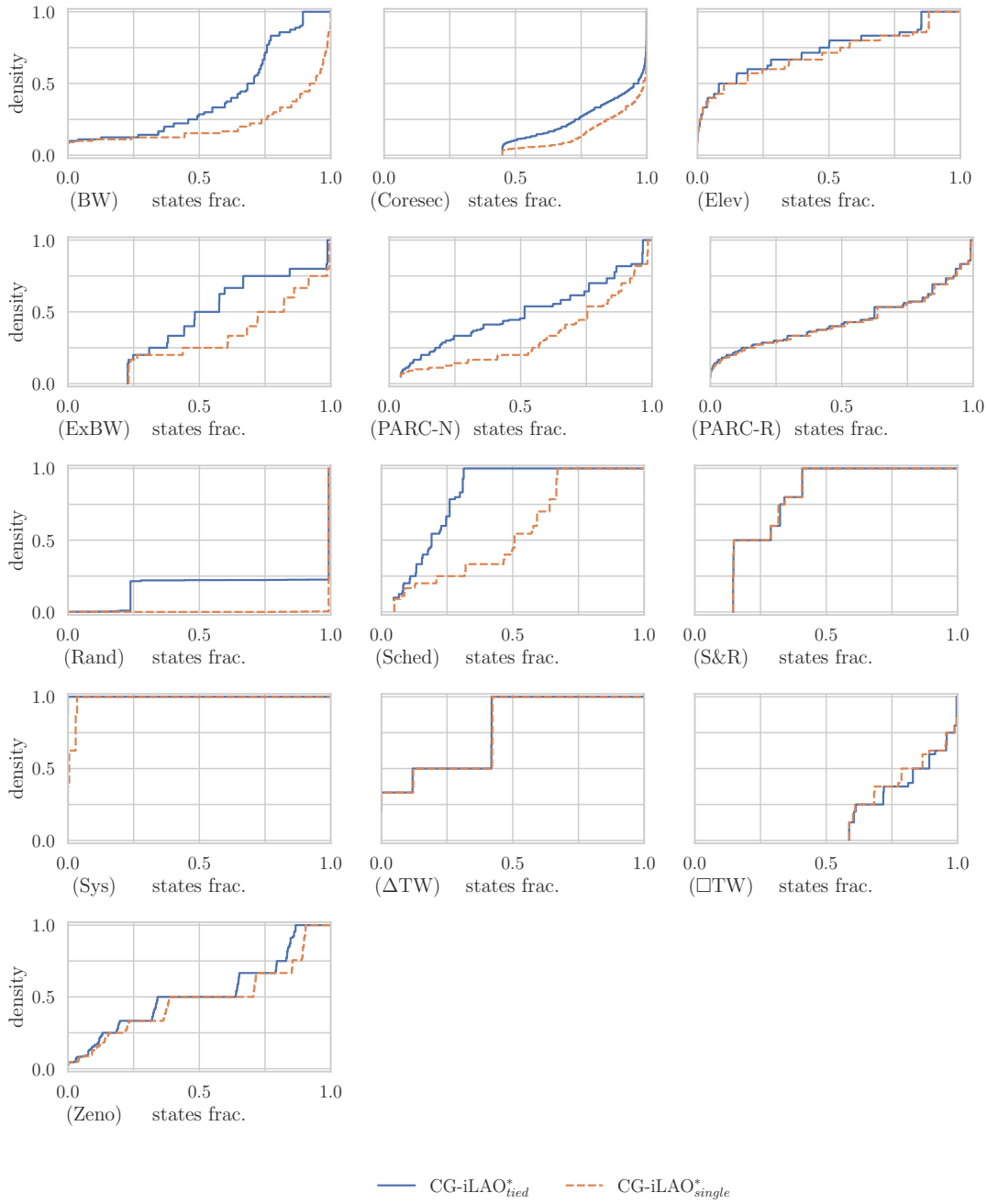


Figure 15: Cumulative plot of state density w.r.t. the fraction of states in the final partial SSP over 5 instances for one representative problem per domain.

Figure 16: Box-and-whisker plots of area under density curve (density AUC) aggregated over problems within each domain.

7.7 How does CG-iLAO* compare to action elimination?

We have discussed that CG-iLAO*'s mechanism for ignoring actions has two key advantages over action elimination: it does not require an upper bound, and it starts with a minimal set of actions and only adds more when they are required. By adding actions as they are required, CG-iLAO* enjoys small partial SSPs from the start and keeps them relatively small; compared to action elimination, which starts with all actions and only removes them when they are proved suboptimal, which is towards the end of the algorithm's lifetime. These advantages result in CG-iLAO* being more effective than action elimination. This section confirms our claim experimentally, and furthermore shows that action elimination is able to remove only few actions that CG-iLAO* inserts into its partial SSPs, demonstrating that CG-iLAO* adds few provably suboptimal actions. In order to compare CG-iLAO*'s mechanism with action elimination we now describe how to augment our algorithms with action elimination.

Adding Action Elimination to iLAO* In general, Bellman-backup-based algorithms can be easily modified to use action elimination with the following modifications:

1. Rename the lower bound V to V_{lb}, and additionally track an upper bound V_{ub}. We initialise V_{ub} with the trivial upper bound $V_{\text{ub}}(g) = 0$ for all goals $g \in \mathsf{G}$ and $V_{\text{ub}}(s) = d$ for all other states $s \in \mathsf{S} \setminus \mathsf{G}$, where $d = \infty$ or the penalty term if we are using the fixed-penalty transformation.

2. For each backup to V_{lb}, also apply a backup to V_{ub} (ignoring its residual).

3. After computing $Q_{\text{lb}}(s, a)$ and $Q_{\text{ub}}(s, a)$ for each $a \in \widehat{\mathsf{A}}(s)$, permanently remove (eliminate) any action $\tilde{a} \in \widehat{\mathsf{A}}$ from the partial SSP if it has $Q_{\text{lb}}(s, \tilde{a}) > \min_{a \in \mathsf{A}(s)} Q_{\text{ub}}(s, a)$.

If $V_{\text{lb}} \leq V^* \leq V_{\text{ub}}$, then these modifications preserve the correctness of the algorithm, because we are only removing suboptimal actions with action elimination (theorem 7). Indeed, if iLAO* is initialised with $V_{\text{ub}} \geq V^*$, then it preserves $V_{\text{ub}} \geq V^*$ as an invariant, by using similar arguments to show iLAO* has the invariant $V_{\text{lb}} \leq V^*$ when initialised with an admissible heuristic [Hansen and Zilberstein, 2001]. Thus, this definition of iLAO* with action elimination behaves as we expect and remains optimal.

Adding Action Elimination to CG-iLAO* We can apply the same modifications to CG-iLAO*, but with a subtle yet important change. Recall that CG-iLAO* does not guarantee $V_{\text{lb}} \leq V^*$ in general, and if we apply action elimination when $V_{\text{lb}} \not\leq V^*$ then optimal actions may be permanently removed, breaking the optimality of the algorithm. So, we only allow CG-iLAO* to apply action elimination when $\Gamma = \emptyset$, at which point $V_{\text{lb}} \leq V^*$ holds (see lemma 4), and action elimination can be performed safely.[9] There is no analogous concern for V_{ub}, and it is always a valid upper bound because the absence of actions can only increase V_{ub}, since optimal actions may be missing.

Thus, we define CG-iLAO*$_{\text{tied-elim}}$, CG-iLAO*$_{\text{single-elim}}$, and CG-iLAO*$_{\text{all-elim}}$ by adding action elimination to the relevant variant of CG-iLAO*. We make note of two technical details:

1. Initially, the only values of V_{ub} with $V_{\text{ub}}(s) < d$ are goal states, so Bellman backups can not decrease V_{ub} until goal states enter the partial SSP. Thus, in our implementation, we only start computing Q-values for V_{ub} after a goal state enters the partial SSP, avoiding needless computation of Q-values fixed at d.

2. We still use ϵ-consistency as the termination condition. With access to V_{ub}, other termination conditions are possible, e.g., $V_{\text{ub}}(s) - V_{\text{lb}}(s) \leq \epsilon \; \forall s \in \mathsf{S}$ [McMahan, Likhachev, and Gordon, 2005], which have slightly different guarantees, and can in some cases be satisfied sooner.

In these experiments, we consider the following algorithms:

- CG-iLAO*$_{\text{tied}}$ and CG-iLAO*$_{\text{tied-elim}}$
- CG-iLAO*$_{\text{single}}$ and CG-iLAO*$_{\text{single-elim}}$
- CG-iLAO*$_{\text{all}}$ and CG-iLAO*$_{\text{all-elim}}$
- FTVI [Dai, Mausam, and Weld, 2009].

[9] Actually, lemma 4 gives us $V_{\text{lb}} \leq V^* + \epsilon \overline{N}(s, \mathbb{S})$, i.e., there is an error term. This error term is insignificant in practice, but can theoretically affect optimality.

ROC		PDB		LMCut	
CG-iLAO$^*_{\text{tied}}$	3.01±0.12	CG-iLAO$^*_{\text{tied}}$	3.01±0.12	CG-iLAO$^*_{\text{tied}}$	3.22±0.11
CG-iLAO$^*_{\text{single}}$	3.14±0.13	CG-iLAO$^*_{\text{single}}$	3.30±0.13	CG-iLAO$^*_{\text{single}}$	3.38±0.12
CG-iLAO$^*_{\text{tied-elim}}$	3.47±0.12	CG-iLAO$^*_{\text{tied-elim}}$	3.63±0.11	CG-iLAO$^*_{\text{single-elim}}$	3.42±0.12
CG-iLAO$^*_{\text{single-elim}}$	3.49±0.13	CG-iLAO$^*_{\text{single-elim}}$	3.86±0.12	CG-iLAO$^*_{\text{tied-elim}}$	3.79±0.11
CG-iLAO$^*_{\text{all}}$	4.33±0.13	CG-iLAO$^*_{\text{all}}$	4.17±0.12	CG-iLAO$^*_{\text{all}}$	4.08±0.12
CG-iLAO$^*_{\text{all-elim}}$	4.52±0.13	CG-iLAO$^*_{\text{all-elim}}$	4.46±0.12	CG-iLAO$^*_{\text{all-elim}}$	4.45±0.11
FTVI	6.05±0.13	FTVI	5.57±0.13	FTVI	5.67±0.13

Table 7: Runtime ranking of CG-iLAO* and action elimination methods within a specified heuristic (mean and 95% CI over all instances).

		BW	Coresec	Elev	ExBW	PARC-N	PARC-R	Rand	□TW	S&R	Sched	Sys	ΔTW	Zeno	total
	# of instances	110	35	75	105	30	30	75	70	25	45	20	40	45	705
ROC	FTVI	55	20	50	65	0	10	33	54	20	30	**20**	30	5	392
	CG-iLAO$^*_{\text{all}}$	**105**	25	70	**105**	**30**	**27**	35	60	**25**	**45**	**20**	35	39	621
	CG-iLAO$^*_{\text{all-elim}}$	**105**	25	70	**105**	**30**	25	35	60	**25**	**45**	**20**	35	35	615
	CG-iLAO$^*_{\text{tied}}$	**105**	25	**75**	**105**	**30**	25	36	60	20	**45**	**20**	35	**40**	621
	CG-iLAO$^*_{\text{tied-elim}}$	**105**	25	74	**105**	**30**	25	37	60	20	**45**	**20**	35	38	619
	CG-iLAO$^*_{\text{single}}$	**105**	25	**75**	**105**	**30**	25	**43**	60	20	**45**	**20**	35	**40**	**628**
	CG-iLAO$^*_{\text{single-elim}}$	**105**	25	72	**105**	**30**	25	**43**	60	20	**45**	**20**	35	38	623
PDB	FTVI	45	20	50	57	0	0	22	55	20	30	**20**	30	5	354
	CG-iLAO$^*_{\text{all}}$	85	**30**	70	90	0	0	30	60	**25**	30	**20**	37	**40**	517
	CG-iLAO$^*_{\text{all-elim}}$	85	**30**	70	90	0	0	30	60	**25**	30	**20**	35	**40**	515
	CG-iLAO$^*_{\text{tied}}$	90	**30**	**75**	90	0	0	30	60	24	30	**20**	**38**	**40**	527
	CG-iLAO$^*_{\text{tied-elim}}$	90	**30**	74	90	0	0	30	60	**25**	30	**20**	35	**40**	524
	CG-iLAO$^*_{\text{single}}$	90	**30**	**75**	90	0	0	30	60	20	30	**20**	37	**40**	522
	CG-iLAO$^*_{\text{single-elim}}$	90	**30**	72	90	0	0	30	60	20	30	**20**	35	**40**	517
LMCut	FTVI	45	20	50	70	0	1	18	60	20	30	**20**	25	5	364
	CG-iLAO$^*_{\text{all}}$	45	25	70	100	**30**	20	20	**65**	**25**	40	**20**	30	25	515
	CG-iLAO$^*_{\text{all-elim}}$	45	25	70	100	**30**	20	20	**65**	**25**	40	**20**	30	25	515
	CG-iLAO$^*_{\text{tied}}$	45	25	70	101	**30**	20	20	**65**	24	**45**	**20**	30	25	520
	CG-iLAO$^*_{\text{tied-elim}}$	45	25	70	100	**30**	20	20	**65**	21	40	**20**	30	25	511
	CG-iLAO$^*_{\text{single}}$	45	25	71	100	**30**	20	20	**65**	20	**45**	**20**	30	25	516
	CG-iLAO$^*_{\text{single-elim}}$	45	25	70	101	**30**	20	20	**65**	20	42	**20**	30	25	513

Table 8: Coverage for action elimination experiment with each considered heuristic over the benchmark domains. The highest coverage for each problem is marked with boldface.

To verify that CG-iLAO* is more effective than action elimination, we compare the performance of CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$ with their action-elimination extended counterparts, and additionally compare them to CG-iLAO$^*_{\text{all}}$ and its action-eliminated counterpart, which does not use the mechanism for ignoring inactive actions and implements iLAO*. We also include FTVI, as a baseline of algorithms that use action elimination. The ranks (table 7), coverage (table 8), and cumulative plots (figure 17) paint a consistent picture:

- algorithms with action elimination are slower than their counterparts without action elimination,
- CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$ are faster than any of the algorithms with action elimination that we consider.

The issue with action elimination is that it must track and compute Bellman backups for the additional value function V_{ub}, incurring an additional overhead of Q-value computations, and this can only pay off if it eliminates sufficiently many actions to save more Q-value computations. By the cumulative plot over Q-values (figure 17), we see that this is not the case.

We now investigate how many actions are removed by action elimination, compared to how many actions are ignored by CG-iLAO*. Table 9 shows the number of actions in the final partial SSPs of the

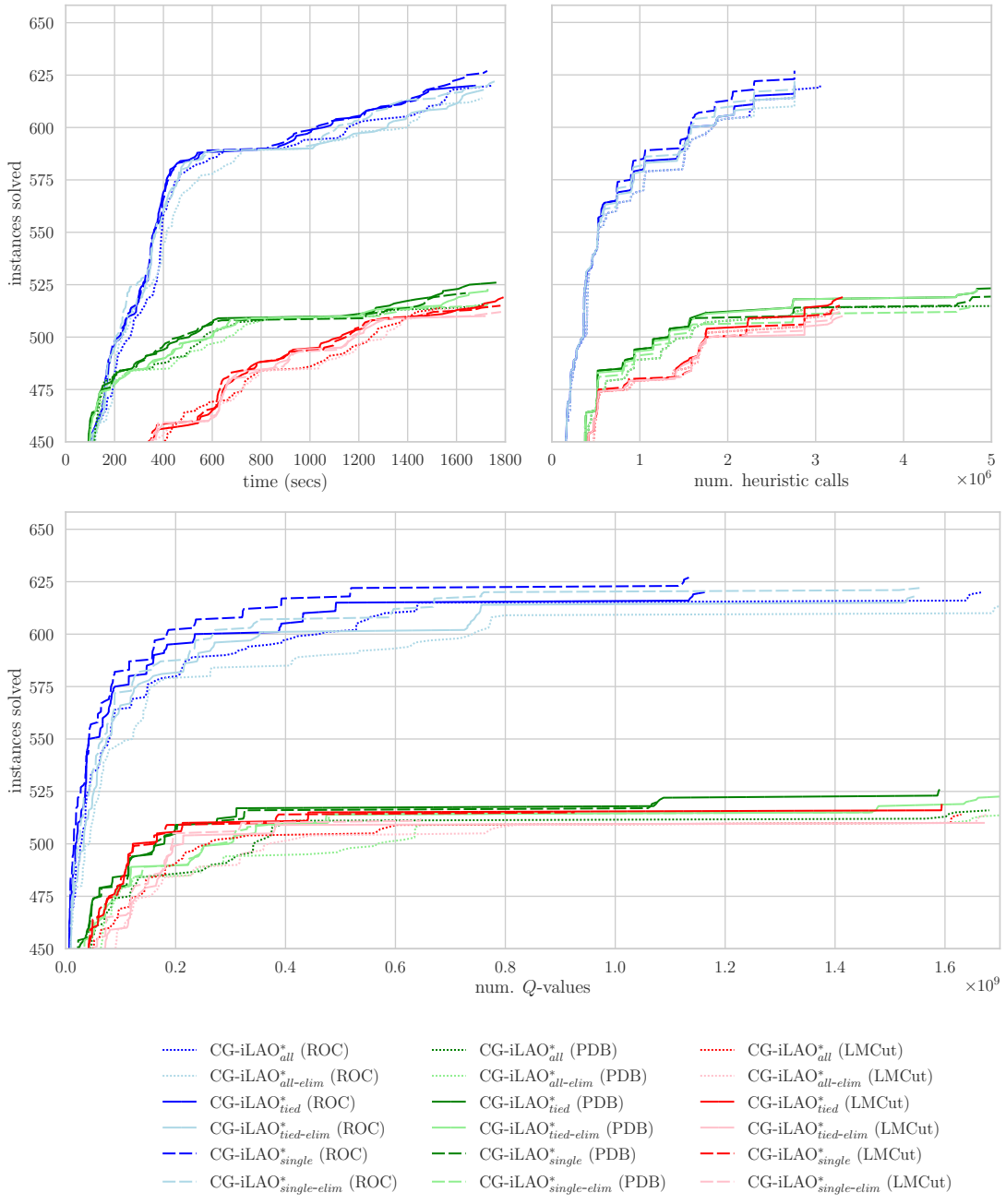


Figure 17: For each algorithm and heuristic, the cumulative plot of how many problems and seeds were solved w.r.t. time in seconds (top left), number of Q-values (top right), and number of calls to heuristic (bottom). To focus on where the algorithms are different we start the y-axis at 450 instances, and end the x-axis at 5×10^6 heuristic calls.

CG-iLAO* variants. We do not include states in this table – action elimination has no way to reduce the number of states, so the percentage of states is equal between algorithms and their counterpart with action elimination (excluding some noise). In terms of actions, we see that CG-iLAO$^*_{\text{all-elim}}$ never has significantly fewer actions in its final partial SSP than CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$, and often has significantly more actions. This tells us that CG-iLAO* is more effective at reducing partial SSP sizes than action elimination. Moreover, if we compare CG-iLAO$^*_{\text{tied}}$ and CG-iLAO$^*_{\text{single}}$ with their action elimination counterparts, we see that action elimination is able to shrink the partial SSP slightly, but never significantly, i.e., CG-iLAO* adds very few actions that can be removed by action elimination. This is a strong argument in favour of CG-iLAO*: its mechanism for ignoring actions induces similarly sized final partial SSPs to action elimination, but it does not require upper bounds, and it avoids these actions

		CG-iLAO$^*_{\text{tied}}$	CG-iLAO$^*_{\text{tied-elim}}$	CG-iLAO$^*_{\text{single}}$	CG-iLAO$^*_{\text{single-elim}}$	CG-iLAO$^*_{\text{all-elim}}$
BW	ROC	40.8±1.2	40.7±1.3	33.2±2.8	33.1±2.8	98.8± 0.3
	LMCut	45.5±1.4	44.6±1.3	42.5±2.0	41.5±1.8	97.6± 0.3
	PDB	39.7±0.9	39.3±0.9	30.9±1.6	30.6±1.6	97.4± 0.5
Coresec	ROC	33.6±1.9	33.5±1.8	20.5±1.1	20.4±1.0	77.6± 4.0
	LMCut	39.0±1.9	37.4±1.5	24.9±1.1	23.8±0.9	82.6± 3.1
	PDB	29.0±1.6	28.4±1.5	22.2±1.6	22.0±1.5	73.0± 3.3
Elev	ROC	60.5±1.6	60.1±1.5	53.3±2.2	53.0±2.1	98.0± 0.6
	LMCut	45.0±3.1	44.5±3.0	40.4±3.0	40.0±2.8	97.0± 1.1
	PDB	56.9±1.7	56.5±1.6	50.1±2.2	49.8±2.2	97.5± 0.8
ExBW	ROC	44.8±1.5	44.1±1.5	39.6±1.7	38.8±1.7	85.6± 2.2
	LMCut	39.2±2.1	38.6±2.0	37.1±1.8	36.5±1.7	88.3± 2.4
	PDB	53.9±3.4	53.3±3.4	49.9±3.7	49.4±3.7	93.4± 1.5
PARC-N	ROC	44.7±0.9	34.8±1.0	32.7±0.9	23.9±0.9	44.8± 1.6
	LMCut	44.0±1.2	37.1±1.3	35.2±1.3	29.0±1.1	62.0± 1.7
PARC-R	ROC	44.1±1.0	42.9±0.7	43.0±1.0	41.9±0.8	92.7± 2.0
	LMCut	51.9±1.0	50.8±1.1	50.1±1.2	49.1±1.4	89.8± 1.8
Rand	ROC	9.1±1.8	9.1±1.8	0.9±0.3	0.9±0.3	100.0
	LMCut	9.9±3.9	9.9±3.9	1.8±0.7	1.8±0.7	76.1±18.6
	PDB	17.0±8.1	17.0±8.0	1.0±0.5	1.0±0.5	75.0±14.8
□TW	ROC	31.5±0.8	26.6±1.1	33.1±1.6	27.9±1.7	66.2± 4.2
	LMCut	30.9±0.9	26.4±1.2	32.8±1.8	27.9±1.8	67.4± 4.2
	PDB	31.0±1.5	26.3±1.3	31.5±1.8	26.7±1.8	66.2± 3.9
S&R	ROC	91.6±0.8	83.3±2.3	91.4±0.7	83.0±2.3	91.1± 1.9
	LMCut	92.0±0.8	83.5±2.3	91.3±0.7	82.9±2.3	90.8± 2.1
	PDB	92.2±0.8	85.1±2.5	91.4±0.7	82.7±2.4	91.3± 2.0
Sched	ROC	78.1±0.7	77.6±0.9	50.7±1.1	50.2±1.1	81.4± 0.9
	LMCut	73.8±3.2	73.2±3.5	50.2±1.2	49.8±1.3	79.0± 0.5
	PDB	79.6±0.5	79.6±0.5	51.1±1.3	51.1±1.3	79.6± 0.5
Sys	ROC	99.5±0.3	95.8±0.8	97.4±0.3	93.6±0.8	95.9± 0.8
	LMCut	99.5±0.4	95.4±0.9	97.3±0.4	93.3±0.8	95.5± 0.9
	PDB	100.0	95.6±1.0	98.4±0.4	93.8±0.7	95.6± 1.0
ΔTW	ROC	64.4±0.8	63.0±0.9	63.8±0.9	62.3±0.9	86.3± 2.9
	LMCut	68.6±1.5	67.8±1.8	67.0±1.5	66.7±1.6	88.5± 3.9
	PDB	65.7±0.7	64.2±0.8	65.5±0.7	64.0±0.8	86.2± 3.2
Zeno	ROC	37.1±0.6	37.0±0.6	30.1±0.8	30.1±0.8	99.9± 0.1
	LMCut	34.3±0.4	34.3±0.4	31.6±0.5	31.6±0.5	99.9± 0.1
	PDB	44.0±1.3	44.0±1.3	34.0±1.1	34.0±1.1	99.8± 0.1

Table 9: Number of actions in each algorithm's final partial SSP, as a percentage of CG-iLAO$^*_{\text{all}}$'s final partial SSP. These values are means over instances where the considered algorithm and CG-iLAO$^*_{\text{all}}$ both terminated.

from the start, rather than eliminating them towards the end of the algorithm's lifetime.

8 Conclusion

In this paper, we addressed an open problem in optimal heuristic search for SSPs: existing methods consider all applicable actions, even when the heuristic makes it clear that some are not needed. To do this, we built on existing connections between operations research and planning to reframe iLAO*, a state-of-the-art optimal heuristic-search algorithm, as constraint and variable generation for LPs. Under this lens, we were able to refine the existing method's separation oracle, which enables the algorithm to ignore inactive actions, and only add actions when they are deemed necessary. Bringing this back into a dynamic programming implementation yields our optimal heuristic-search algorithm CG-iLAO*. CG-iLAO*'s novel ability to ignore inactive actions gives it different theoretical properties from existing methods, e.g., it does not guarantee that its value function remains admissible, and the algorithm is not monotonic. Nevertheless, we have proved that it is an optimal algorithm. We showed experimentally that CG-iLAO*'s mechanism for ignoring actions pays off, and it outperforms the state-of-the-art on our benchmarks. Going into more detail, we showed that CG-iLAO* is able to ignore a significant proportion of actions, resulting in significantly smaller search spaces than iLAO*, which in turn results in fewer Q-value computations and savings in time. In comparison, the existing technique of action elimination, which proves that actions are suboptimal in order to permanently remove them, fails to pay off. We

showed that CG-iLAO*'s mechanism for ignoring actions is faster, generally ignores the same actions that action elimination removes, and in addition CG-iLAO* not require an upper bound. Thus, CG-iLAO* successfully addresses the open problem, and represents an important step in heuristic-search that lets us use heuristics to select promising actions, not only states.

9 Future Work

As a direction for future research, we aim to generalise CG-iLAO* to more complex models. We have seen that CG-iLAO*'s mechanism for ignoring inactive actions and adding them as necessary yields in savings of Q-value computations, which is a promising benefit for problems where Q-values are expensive to compute. For example, consider models with imprecise parameters, such as MDPIPs and MDPSTs [White III and Eldeib, 1994; Trevizan, Cozman, and Barros, 2007]. These models have *minimax* semantics for the Bellman equations, which means that their value function minimises the expected cost-to-go while accounting for an adversary who selects the values of the imprecise parameters in a way that maximises the cost-to-go. Thus, computing each Q-value is an expensive operation that requires solving a maximisation problem. Consequently, CG-iLAO*'s mechanism for avoiding actions and saving on Q-values could potentially improve performance significantly.

Other candidate models include SSPs with *PLTL constraints* [Baumgartner, Thiébaux, and Trevizan, 2018; Mallett, Thiebaux, and Trevizan, 2021]. Typically, these models are solved by extending the state and action space of the original SSP with information about the relevant PLTL formulae, in order to track whether the PLTL constraints are satisfied. It may be possible to use the concept of inactive actions to avoid actions that lead to constraint violations in their partial problems. In fact, the methods presented in this paper may be applicable to model checking more broadly. For example, previous work has explored the use of heuristics to guide the search for probabilistic reachability [Brázdil et al., 2014], where action elimination can be directly applied.

Acknowledgements

This research was undertaken with the assistance of resources and services from the National Computational Infrastructure (NCI), which is supported by the Australian Government. Johannes Schmalz received funding from DFG grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science).

References

Barto, A. G.; Bradtke, S. J.; and Singh, S. P. 1995. Learning To Act Using Real-Time Dynamic Programming. *Artificial Intelligence*, 72(1): 81–138. [Cited on pages 22 and 24.]

Baumgartner, P.; Thiébaux, S.; and Trevizan, F. 2018. Heuristic Search Planning With Multi-Objective Probabilistic LTL Constraints. In *Proc. of Int. Conf. on Principles of Knowledge Representation and Reasoning (KR)*, 415–424. [Cited on page 48.]

Bellman, R. E. 1957. *Dynamic Programming.* Princeton University Press. [Cited on pages 1 and 5.]

Bertsekas, D. 1995. *Dynamic Programming and Optimal Control.* Athena Scientific. [Cited on pages 2, 5, and 24.]

Bertsekas, D.; and Tsitsiklis, J. 1991. An Analysis of Stochastic Shortest Path Problems. *Mathematics of Operations Research*, 16(3): 580–595. [Cited on pages 1 and 4.]

Bertsimas, D.; and Tsitsiklis, J. 1997. *Introduction to Linear Optimization.* Athena Scientific. [Cited on page 10.]

Bonet, B.; and Geffner, H. 2000. Planning as Heuristic Search: New Results. In *Proc. of European Conf. on Planning (ECP)*, 360–372. [Cited on page 29.]

Bonet, B.; and Geffner, H. 2003a. Faster Heuristic Search Algorithms for Planning with Uncertainty and Full Feedback. In *Proc. of Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 1233–1238. [Cited on pages 6 and 7.]

Bonet, B.; and Geffner, H. 2003b. Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming. In *Proc. of Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 12–21. [Cited on pages 2, 22, 29, and 36.]

Bonet, B.; and Givan, R. 2006. 5th International Planning Competition: Non-deterministic track. Call for Participation. [Cited on page 29.]

Boyarski, E.; Felner, A.; Stern, R.; Sharon, G.; Tolpin, D.; Betzalel, O.; and Shimony, E. 2015. ICBS: Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding. In *Proc. of Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 740–746. [Cited on page 25.]

Brázdil, T.; Chatterjee, K.; Chmelík, M.; Forejt, V.; Křetínský, J.; Kwiatkowska, M.; Parker, D.; and Ujma, M. 2014. Verification of Markov Decision Processes Using Learning Algorithms. In *Automated Technology for Verification and Analysis*, 98–114. [Cited on page 48.]

Bryce, D.; and Buffet, O. 2008. International Planning Competition Uncertainty Part: Benchmarks and Results. [Cited on page 29.]

Calliess, J.-P.; and Roberts, S. 2021. Multi-Agent Planning with Mixed-Integer Programming and Adaptive Interaction Constraint Generation (Extended Abstract). In *Proc. of Int. Symp. on Combinatorial Search (SoCS)*, 207–208. [Cited on page 25.]

Dai, P.; Mausam; and Weld, D. 2009. Focused Topological Value Iteration. In *Proc. of Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 82–89. [Cited on pages 2 and 44.]

Dai, P.; Mausam; Weld, D. S.; and Goldsmith, J. 2011. Topological Value Iteration Algorithms. *Journal of Artificial Intelligence Research*, 181–209. [Cited on page 24.]

Geißer, F.; Povéda, G.; Trevizan, F.; Bondouy, M.; Teichteil-Königsbuch, F.; and Thiébaux, S. 2020. Optimal and Heuristic Approaches for Constrained Flight Planning under Weather Uncertainty. In *Proc. of Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 384–393. [Cited on page 1.]

Hansen, E.; and Abdoulahi, I. 2015. Efficient Bounds in Heuristic Search Algorithms for Stochastic Shortest Path Problems. In *Proc. of AAAI Conf. on Artificial Intelligence*, 3283–3290. [Cited on pages 7 and 24.]

Hansen, E.; and Abdoulahi, I. 2016. General Error Bounds in Heuristic Search Algorithms for Stochastic Shortest Path Problems. In *Proc. of AAAI Conf. on Artificial Intelligence*, 3130–3137. [Cited on page 15.]

Hansen, E. A.; and Zilberstein, S. 2001. LAO*: A Heuristic Search Algorithm That Finds Solutions With Loops. *Artificial Intelligence*, 129(1): 35–62. [Cited on pages 2, 3, 7, 15, 21, 24, 29, 36, and 44.]

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107. [Cited on page 26.]

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? *Proc. of Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 162–169. [Cited on page 29.]

Hoffmann, J. 2015. Simulated Penetration Testing: From "Dijkstra" to "Turing Test++". In *Proc. of Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 364–372. [Cited on page 1.]

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14: 253–302. [Cited on page 23.]

Keller, T.; and Eyerich, P. 2012. PROST: Probabilistic Planning Based on UCT. In *Proc. of Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 119–127. [Cited on page 22.]

Klößner, T.; and Hoffmann, J. 2021. Pattern Databases for Stochastic Shortest Path Problems. In *Proc. of Int. Symp. on Combinatorial Search (SoCS)*, 131–135. [Cited on page 29.]

Klößner, T.; Hoffmann, J.; Steinmetz, M.; and Torralba, Á. 2021. Pattern Databases for Goal-Probability Maximization in Probabilistic Planning. In *Proc. of Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 201–209. [Cited on page 29.]

Klößner, T.; Hoffmann, J.; Steinmetz, M.; and Torralba, Á. 2021. Code and Benchmarks of ICAPS 2021 Paper "Pattern Databases for Goal-Probability Maximization in Probabilistic Planning". [Cited on pages 29, 30, and 31.]

Koenig, S.; Chan, S.-H.; Li, J.; and Zheng, Y. 2023. *Artificial Intelligence and Automation*, 205–231. Springer International Publishing. [Cited on page 1.]

Little, I.; and Thiebaux, S. 2007. Probabilistic Planning vs. Replanning. In *ICAPS Workshop on IPC: Past, Present and Future*, –. [Cited on pages 31 and 32.]

Mallett, I.; Thiebaux, S.; and Trevizan, F. 2021. Progression Heuristics for Planning with Probabilistic LTL Constraints. In *Proc. of AAAI Conf. on Artificial Intelligence*, 11870–11879. [Cited on page 48.]

Mausam; and Kolobov, A. 2012. *Planning with Markov Decision Processes: An AI Perspective*. Morgan & Claypool Publishers. [Cited on pages 6, 7, and 26.]

McMahan, H. B.; Likhachev, M.; and Gordon, G. J. 2005. Bounded Real-Time Dynamic Programming: RTDP with Monotone Upper Bounds and Performance Guarantees. In *Proc. of Int. Conf. on Machine Learning (ICML)*, 569–576. [Cited on pages 3, 24, and 44.]

Nebel, B. 2020. On the Computational Complexity of Multi-Agent Pathfinding on Directed Graphs. In *Proc. of Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 212–216. [Cited on page 25.]

Péron, M.; Jansen, C. C.; Mantyka-Pringle, C.; Nicol, S.; Schellhorn, N. A.; Becker, K. H.; and Chadès, I. 2017. Selecting Simultaneous Actions of Different Durations to Optimally Manage an Ecological Network. *Methods in Ecology and Evolution*, 1332–1341. [Cited on page 1.]

Rovner, A.; Sievers, S.; and Helmert, M. 2019. Counterexample-Guided Abstraction Refinement for Pattern Selection in Optimal Classical Planning. In *Proc. of Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 362–367. [Cited on page 25.]

Russell, S.; and Norvig, P. 2016. *Artificial Intelligence A Modern Approach, Global Edition*. Pearson Deutschland, third edition. [Cited on page 26.]

Sanner, S. 2010. Relational Dynamic Influence Diagram Language (RDDL): Language Description. [Cited on page 29.]

Sanner, S.; Goetschalckx, R.; Driessens, K.; and Shani, G. 2009. Bayesian Real-Time Dynamic Programming. In *Proc. of Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 1784–1789. [Cited on pages 3 and 24.]

Say, B.; and Sanner, S. 2019. Metric Hybrid Factored Planning in Nonlinear Domains with Constraint Generation. In *Proc. of Conf. on Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*, 502–518. [Cited on page 25.]

Schmalz, J.; and Trevizan, F. 2023. Code, Benchmarks, and Technical Report for AAAI 2024 Paper "Efficient Constraint Generation for Stochastic Shortest Path Problems". [Cited on pages 29 and 33.]

Schmalz, J.; and Trevizan, F. 2024. Efficient Constraint Generation for Stochastic Shortest Path Problems. In *Proc. of AAAI Conf. on Artificial Intelligence*, 20247–20255. [Cited on pages 15, 31, and 36.]

Schuurmans, D.; and Patrascu, R. 2001. Direct Value-Approximation for Factored MDPs. In *Advances in Neural Information Processing Systems (NIPS)*, 1579–1586. [Cited on page 26.]

Seipp, J.; and Helmert, M. 2013. Counterexample-Guided Cartesian Abstraction Refinement. In *Proc. of Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 347–351. [Cited on page 25.]

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. 2012. Conflict-Based Search For Optimal Multi-Agent Path Finding. In *Proc. of AAAI Conf. on Artificial Intelligence*, 563–569. [Cited on page 25.]

Smith, T.; and Simmons, R. G. 2006. Focused Real-Time Dynamic Programming for MDPs: Squeezing More Out of a Heuristic. In *Proc. of AAAI Conf. on Artificial Intelligence*, 1227–1232. [Cited on pages 3 and 24.]

Steinmetz, M.; Hoffmann, J.; and Buffet, O. 2016. Goal Probability Analysis in Probabilistic Planning: Exploring and Enhancing the State of the Art. *Journal of Artificial Intelligence Research*, 57: 229–271. [Cited on page 30.]

Stern, R.; Sturtevant, N.; Felner, A.; Koenig, S.; Ma, H.; Walker, T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K.; Barták, R.; and Boyarski, E. 2021. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *Proc. of Int. Symp. on Combinatorial Search (SoCS)*, 151–158. [Cited on page 25.]

Teichteil-Königsbuch, F. 2012. Stochastic Safest and Shortest Path Problems. In *Proc. of AAAI Conf. on Artificial Intelligence*, 1825–1831. [Cited on page 4.]

Teichteil-Königsbuch, F.; Kuter, U.; and Infantes, G. 2010. Incremental Plan Aggregation for Generating Policies in MDPs. In *Proc. of Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, 1231–1238. [Cited on pages 22 and 23.]

Trevizan, F.; Cozman, F. G.; and Barros, L. N. 2007. Planning under Risk and Knightian Uncertainty. In *Proc. of Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 2023–2028. [Cited on page 48.]

Trevizan, F.; Teichteil-Königsbuch, F.; and Thiébaux, S. 2017. Efficient Solutions for Stochastic Shortest Path Problems with Dead Ends. In *Proc. of Int. Conf. on Uncertainty in Artificial Intelligence (UAI)*, –. [Cited on pages 4, 22, and 29.]

Trevizan, F.; Thiébaux, S.; and Haslum, P. 2017. Occupation Measure Heuristics for Probabilistic Planning. In *Proc. of Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 306–315. [Cited on pages 29 and 30.]

Vinzent, M.; Sharma, S.; and Hoffmann, J. 2023. Neural Policy Safety Verification via Predicate Abstraction: CEGAR. *Proc. of AAAI Conf. on Artificial Intelligence*, 15188–15196. [Cited on page 25.]

Walraven, E.; and Spaan, M. 2017. Accelerated Vector Pruning for Optimal POMDP Solvers. In *Proc. of AAAI Conf. on Artificial Intelligence*, 3672–3678. [Cited on page 25.]

Warnquist, H.; Kvarnström, J.; and Doherty, P. 2010. Iterative Bounding LAO*. In *Proc. of European Conf. on Artificial Intelligence (ECAI)*, 341–346. [Cited on pages 3 and 24.]

White, D. J. 1985. Real Applications of Markov Decision Processes. *Interfaces*, 15(6): 73–83. [Cited on page 1.]

White III, C. C.; and Eldeib, H. K. 1994. Markov Decision Processes with Imprecise Transition Probabilities. *Operations Research*, 42(4): 739–749. [Cited on page 48.]

Yoon, S. W.; Fern, A.; and Givan, R. 2007. FF-Replan: A Baseline for Probabilistic Planning. In *Proc. of Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 352–359. [Cited on page 23.]

Yoshizumi, T.; Miura, T.; and Ishida, T. 2000. A* with Partial Expansion for Large Branching Factor Problems. In *Proc. of AAAI Conf. on Artificial Intelligence*, 923–929. [Cited on pages 26 and 27.]

Younes, H.; Littman, M.; Weissman, D.; and Asmuth, J. 2005. The First Probabilistic Track of the International Planning Competition. *Journal of Artificial Intelligence Research*, 851–887. [Cited on pages 6 and 29.]

Yu, J.; and LaValle, S. 2013. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. In *Proc. of AAAI Conf. on Artificial Intelligence*, 1443–1449. [Cited on page 25.]