Learning-Based Search Control for Policy Tree Search

by

Jake Tuero

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

University of Alberta

Abstract

Policy tree search combines the generalization of learned policies with the deliberation of systematic search, offering a practical route to solving challenging deterministic single-agent problems where either component alone can be insufficient. This thesis focuses on first-solution search in sparse-feedback domains, where the aim is to find any feasible solution efficiently rather than to optimize solution cost. In this setting, a central challenge in policy tree search is decision-making about computation: determining, from limited experience and partial information, where additional search effort is most likely to reduce time-to-solution or search effort. Rather than treating search control as a fixed set of heuristics, the thesis positions it as an adaptive capability that should respond to the properties of the instance being solved.

This thesis develops learning-based principles and methods for allocating and redirecting search effort in a structure-aware way. The resulting algorithms use distributional runtime prediction to make instance-sensitive restart decisions, learned subgoals to concentrate computation on informative intermediate states, and learned rerooters to exploit regularities in the underlying state space without requiring explicit subgoal generation. Across the structured benchmark domains studied, these contributions improve first-solution efficiency, yielding policy tree search procedures that adapt their behaviour to the instance at hand while reducing reliance on brittle hand-designed control rules.

Preface

This thesis is original work done under supervision of Professor Michael Buro and Professor Levi Lelis. Chapter 7 was done jointly with Dr. Laurent Orseau, who provided insights and design considerations into the proposed methods.

The majority of the chapters in this thesis are based on prior work that has been published. Chapter 4 was published as "Bayes DistNet — A Robust Neural Network for Algorithm Runtime Distribution Predictions" in the Proceedings of the AAAI Conference on Artificial Intelligence (2021). Chapter 6 was published as "Subgoal-Guided Policy Heuristic Search with Learned Subgoals" in the Proceedings of the 42nd International Conference on Machine Learning (2025). Chapter 7 was published as "Structure-Induced Information for Rerooting Levin Tree Search" in the Proceedings of the 43rd International Conference on Machine Learning (2026).

This research was supported by Canada's NSERC and the CIFAR AI Chairs program. This research was enabled in part by support provided by the Digital Research Alliance of Canada.

We can solve any problem by introducing an extra level of indirection.

– David J. Wheeler

...except for the problem of too many layers of indirection.

– Kevlin Henney

Acknowledgements

There are many people without whom this journey would not have been possible.

First and foremost, I would like to thank my supervisors, Professor Michael Buro and Professor Levi Lelis. I began graduate school as a curious student without a clear sense of direction, and they provided patient and generous mentorship throughout this journey. Their guidance, encouragement, and thoughtful feedback helped me develop both as a researcher and as a scholar. I am deeply grateful for the time they invested in my work and for the many lessons I learned from them.

I owe a special thanks to Dr. Laurent Orseau for his collaboration on my recent work and for the insight he shared on policy tree search algorithms. Our discussions contributed greatly to the development of this research and to my own understanding of the subject.

I would also like to thank Professor Martin Mueller for his helpful feedback and guidance during our annual supervisory committee meetings. I am grateful as well to Professor Nathan Sturtevant, Professor Jonathan Schaeffer, and Dr. Sven Koenig for serving on my supervisory committee and for the thoughtful questions and feedback they provided during my candidacy examination and thesis examination.

Several people had an important influence on me early in my academic career. I would like to thank Professors Angèle M. Foley, Chính T. Hoàng, and Zilin Wang for recognizing my potential and giving me the opportunity to become involved in research as an undergraduate student. Those experiences introduced me to academic research, shaped the direction of my career, and ultimately opened the door to graduate school.

Finally, I would like to thank my family for their unconditional support, patience, and love. Their encouragement sustained me through the challenges of this journey, and this accomplishment would not have been possible without them.

Contents

List of Tables

List of Figures

List of Algorithms

List of Symbols

$\mathcal{S}$	Set of all states
$\mathcal{S}^g$	Set of goal states
$\mathcal{A}$	Set of actions
$\mathcal{A}(s)$	Set of actions available in state s
$\text{Succ}(s, a)$	Deterministic transition function
s_1	Initial state of the search problem
$a_{1:t}$	A plan (sequence of actions)
$\ell(s, s')$	Step-cost of transitioning from state s to s'
$g(a_{1:t})$	$\sum_{i=1}^{t} \ell(s_i, s_{i+1})$, cost of plan $a_{1:t}$
$\pi(a \mid s)$	Probability of action a in state s from policy π
$\mathcal{N}$	Set of nodes in the search tree
$\mathcal{N}^g$	Set of goal nodes in the search tree
n	A node in $\mathcal{N}$, representing a sequence of actions
$s(n)$	The state represented by node n
$\varphi(n)$	Evaluation function for best-first search
n_1	Root node, representing initial state s_1
na	The resulting child node of n after applying action a
$d(n)$	Node depth, or number of actions from the root that n represents
$g(n)$	Cumulative transition cost of the partial path of node n
$\mathcal{C}(n)$	Child nodes of n
$\text{anc}(n)$	Set of ancestors of n
$\text{anc}_+(n)$	$\text{anc}(n) \cup \{n\}$
$\text{desc}(n)$	Descendants of n
$\text{desc}_+(n)$	$\text{desc}(n) \cup \{n\}$
$n \prec n'$	$n \in \text{anc}(n')$
$n \preceq n'$	$n \in \text{anc}_+(n')$
$\pi(n)$	Path probability of node n, equivalently $\pi(n \mid n_1)$
$\pi(n' \mid n)$	Probability of the suffix path from n to n'; equals $\pi(n')/\pi(n)$ when $n \preceq n'$ and $\pi(n) > 0$
$\ell(n)$	Expansion loss incurred when node n is expanded
$L(S, n)$	Search loss of algorithm S up to and including expansion of n
$\mathcal{T}$	A search tree (see Section 3.2)
$\mathcal{L}(\mathcal{T})$	The leaf nodes of $\mathcal{T}$
$g_\ell(n)$	Cumulative expansion cost along the path from n_1 to n

$\frac{d}{\pi}(n)$	Levin cost function, $d(n)/\pi(n)$
$\frac{d}{\pi}(n_k \prec n)$	Rooted Levin cost function
$\frac{\lambda}{\pi}(n)$	Slenderness cost function
$\frac{\lambda}{\pi}(n \mid n_k)$	Rooted slenderness cost function
w_t	Rerooting weight for LTS rooted at node n_t
$w_{<t}$	Cumulative rerooting weight $w_{<t} = \sum_{i<t} w_i$
$\tilde{w}$	Reparameterization of w

Chapter-Specific Symbols (Chapters 4–7)

$\mathcal{F}$	Parametric runtime or trajectory-length distribution family (Chapters 4–5)
$\boldsymbol{\beta}$	Parameters of the distribution family $\mathcal{F}$ (Chapters 4–5)
$\boldsymbol{\theta} = (\boldsymbol{\mu}, \boldsymbol{\rho})$	Variational posterior parameters of Bayes DistNet (Chapter 4)
ξ	Problem instance (Chapters 4–5)
$\delta_{\xi,j}$	Indicator that runtime observation j for instance ξ is uncensored (Chapter 4)
$\hat{y}_\xi^{(r)}$	r-th Monte Carlo prediction for instance ξ (Chapters 4–5)
T_ξ, F_ξ	Successful trajectory length and its distribution for instance ξ (Chapter 5)
$\hat{F}_\xi$	Predicted trajectory-length distribution for instance ξ (Chapter 5)
τ	Quantile level used to select a LubyTS shift (Chapter 5)
ρ	Multiplier for a sampled-maximum shift (Chapter 5)
k	VQVAE codebook size and number of generated subgoals (Chapter 6)
e_i	ith VQVAE codebook entry (Chapter 6)
π^{hi}, π^{low}	High-level subgoal policy and subgoal-conditioned low-level policy (Chapter 6)
γ	Geometric factor for the Leiden clustering-update schedule (Chapter 7)
τ	Clustering-update index (Chapter 7)
k	Selected hierarchy level of the cluster graph (Chapter 7)
α	Inverse temperature of the heuristic rerooter (Chapter 7)
u_a, u_b	Mixing coefficients for the hybrid rerooter (Chapter 7)
c	Cluster colour assigned to a tree node (Chapter 7)
$M_{\tau,c}$, $\delta_{\tau,c}$	Cluster size and post-update expansion counts for colour c (Chapter 7)

Chapter 1

Introduction

We all search, constantly, without thinking much about it. You search for your keys when you are late, for the right file in a cluttered folder, for a route that avoids traffic, or for the right words to explain an idea. Even when the "space of possibilities" is large, we rarely examine everything. Instead, we follow cues, backtrack when a lead goes cold, and shift attention toward where progress seems most likely.

Many of the problems we want AI systems to solve have the same characteristics. A solution is not presented directly; it must be found by exploring alternatives. In its most basic form, this can be modelled as moving through a space of situations, starting from an initial one, applying rules that transform the current situation into a new one, and repeating until some goal condition is satisfied. The result is a *path* representing a sequence of decisions from start to goal. In many settings, we do not want just any path, but one that is best under some cost measure, such as shortest, cheapest, safest, or most reliable. The principal objective of this thesis is narrower: first-solution efficiency, or finding any feasible solution with low search effort. Solution cost is measured in selected experiments, but is not generally optimized or guaranteed.

A large class of AI methods formalize this view through state-space search [115], where problem solving is reduced to exploring possible action sequences under a transition model and goal condition. In many applications, the state space is too large to represent explicitly, so a solver must generate successor states on demand. As a result, the computational burden is not in defining

the problem, but in deciding how to explore it efficiently.

Tree search provides a general mechanism for this exploration. Starting from an initial state, a tree search algorithm incrementally expands partial solution paths, constructing a search tree whose nodes represent states reachable by action sequences. The defining feature of tree search is not that it enumerates possibilities, since exhaustive enumeration is rarely feasible, but that it structures deliberation. At each step, the algorithm chooses a frontier node to expand and allocates additional computation to that part of the tree. This makes tree search broadly applicable across single-agent tasks that require planning or combinatorial reasoning, including classical planning problems and puzzle-like domains, where solutions must satisfy hard constraints and often require long sequences of correct decisions.

The effectiveness of tree search depends critically on *search control*: the mechanisms that determine which nodes are expanded, which are deferred, and when effort is redirected [74]. Search control can take many forms such as priority functions [42, 83], pruning rules [56], depth or cost limits [55]. In all cases, its purpose is to concentrate limited computational resources on regions of the search tree that are most likely to yield progress. This becomes increasingly important as problem difficulty grows, since the number of candidate paths can increase exponentially with depth.

These challenges are especially pronounced in single-agent "needle-in-the-haystack" settings. In such problems, solutions may exist but are sparse relative to the number of plausible trajectories. Many partial plans may appear locally reasonable yet fail to lead to a goal. Progress may require discovering and committing to narrow bottlenecks in the state space, and errors can be irreversible, producing deep dead ends that are costly to detect. Consequently, performance is governed less by the ability to generate successors and more by the ability to allocate and redirect search effort effectively as evidence accumulates during exploration.

Scope This thesis studies deterministic, single-agent search problems in this first-solution setting, with sparse or uninformative intermediate feedback. The

formal results additionally require the assumptions stated with each algorithm, including finite branching or finite low-cost contours and positive policy probability for at least one solution path. The empirical evaluation focuses on structured, predominantly grid-based benchmark domains. The resulting claims do not extend directly to stochastic or partially observable settings, to objectives requiring optimal or bounded-suboptimal solutions, or to problems in which the policy assigns zero probability to every solution path. As with other learned search-control methods, the models may also fail to generalize under distribution shift.

These considerations motivate the central question of this thesis: *how should search effort be allocated and redirected effectively in challenging single-agent domains?*

Research Questions This thesis addresses the following questions:

1. Can predictions of randomized search outcomes improve instance-specific restart decisions?

2. Can failed search trees provide useful training signal for policies that use intermediate structure?

3. Can state-space structure guide the redirection of search effort without explicit subgoal generation?

4. Under what conditions do these mechanisms improve coverage, expansion efficiency, or time efficiency?

1.1 Contributions and Thesis Outline

This thesis focuses on *policy tree search*, a family of tree search algorithms that use a *policy* — that is, a probability distribution over actions — to guide the search. This thesis argues that a central challenge in the setting studied here is not only learning useful guidance, but deciding how that guidance should be used to control computation: when search should continue, when it should restart, and where additional effort should be redirected. It develops

learning-based methods for making these decisions in an instance-sensitive and structure-aware way, enabling policy tree search to adapt its focus to the problem being solved and reduce reliance on brittle hand-designed control rules. The methods are evaluated on harder benchmark variants within the setting described above.

This thesis makes three primary contributions:

1. It develops distributional prediction methods for randomized-algorithm outcomes and applies them to predict instance-dependent successful-rollout depths for restart control.

2. It extends policy-guided search with learned subgoals that induce a hierarchical decomposition of the task, allowing search to concentrate effort on informative intermediate states and improving training efficiency in sparse-reward domains.

3. It introduces learned rerooters that redirect policy tree search using structural signals derived from the underlying state space. This provides a complementary, lower-overhead mechanism for exploiting structure, capturing many of the benefits of decomposition without requiring explicit subgoal generation or reasoning. Rather than generating and evaluating a set of explicit subgoals and their associated low-level policies at each node, rerooting assigns a single weight from structural information available during search.

The rest of this thesis is organized as follows:

- Chapter 2 reviews the background on which policy tree search builds in practice. It introduces the core concepts of best-first search, heuristics, and policies, and discusses how these forms of guidance are learned and used in search algorithms.

- Chapter 3 lays the theoretical and conceptual foundation for policy tree search and surveys the main algorithmic landscape, including LubyTS, multiTS, LevinTS, PHS*, and $\sqrt{\text{LTS}}$.

- Chapter 4 introduces Bayes DistNet, a robust neural model for algorithm runtime-distribution prediction. In the reported low-data comparisons, the model improves on prior methods, can learn from censored observations, and provides uncertainty estimates over its predictions. This chapter is joint work with Michael Buro. It was previously published at the AAAI Conference on Artificial Intelligence [102].

- Chapter 5 applies Bayes DistNet to LubyTS by using predicted trajectory-length distributions to make instance-specific restart decisions. The resulting method reduces wasted search effort relative to baseline restart modifications, especially when the training data are censored.

- Chapter 6 introduces subgoal-guided extensions of LevinTS and PHS* in which learned subgoals are used to define a hierarchical policy. The resulting policy formulation can be trained even when search fails to find a solution, leading to improved training efficiency. This chapter is joint work with Michael Buro and Levi Lelis. It was previously published at the Forty-second International Conference on Machine Learning [103].

- Chapter 7 introduces structure-induced rerooters for $\sqrt{\text{LTS}}$ and provides an automated method for learning rerooting policies from state-space structure. It shows how both local and global structural information can be used to guide rerooting, and it provides a new bound for their additive combination. This chapter is joint work with Michael Buro, Levi Lelis, and Laurent Orseau. It was previously published at the Forty-third International Conference on Machine Learning [104].

- Chapter 8 concludes the thesis by revisiting its central theme of learning-based search control in policy tree search, summarizing the main contributions, and outlining directions for future work.

Chapter 2

Background

The contributions of this thesis address decision-making about computation in policy tree search. In particular, they attempt to answer the question of how search effort should be allocated and redirected to reduce the computational effort required to reach a solution or improve the quality of the solution found. Before presenting these contributions, this chapter reviews the components on which policy tree search builds in practice, including best-first search, heuristic and policy guidance, and it highlights how each shapes search control in single-agent state spaces.

2.1 Single-Agent Search Problems

This thesis considers single-agent deterministic search problems. We define a single-agent deterministic search problem as the tuple $(\mathcal{S}, \mathcal{A}, \mathrm{Succ}, s_1, \mathcal{S}^g, \ell)$, where $\mathcal{S}$ denotes a set of states, $\mathcal{A}$ is a finite set of action labels, and Succ is a deterministic partial transition function. For each state $s \in S$, let

$$\mathcal{A}(s) \coloneqq \{a \in \mathcal{A} : \mathrm{Succ}(s, a) \text{ is defined}\} \tag{2.1}$$

denote the set of actions applicable in state s. Thus, if $a \in \mathcal{A}(s)$, then $\mathrm{Succ}(s, a) \in \mathcal{S}$ is the state obtained by applying action a from state s. The initial state is given by $s_1 \in \mathcal{S}$, and $\mathcal{S}^g$ is the set of goal states. The function $\ell : \mathcal{S} \times \mathcal{S} \to \mathbb{R}_{\geq 0}$ assigns a non-negative cost $\ell(s, s')$ to transitions from state s to state s'. A (finite) *plan* is an action sequence $a_{1:t} \triangleq (a_1, \ldots, a_t)$ such that, starting from s_1, $a_i \in \mathcal{A}(s_i)$ and $s_{i+1} = \mathrm{Succ}(s_i, a_i)$ for all $i = 1, \ldots, t$. The plan

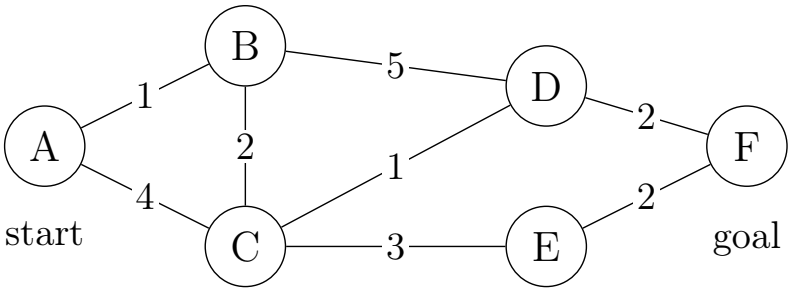


Figure 2.1: A simple single-agent search problem of a toy road network. Each city is represented by a node, with weighted edges showing the cost to travel between neighbouring cities.

$a_{1:t}$ is a *solution* if the terminal state satisfies the goal condition, *i.e.*, $s_{t+1} \in \mathcal{S}^g$. The cost of $a_{1:t}$ is the cumulative transition cost $g(a_{1:t}) = \sum_{i=1}^{t} \ell(s_i, s_{i+1})$, which is also referred to as the *path cost* or *g-cost*. Unless stated otherwise, the *objective* of single-agent search problems is to find a minimum-cost solution, *i.e.*, a solution with minimal cost among all solutions.

Example 2.1.1. A single-agent search problem is shown in Figure 2.1, which depicts a small road network. The set of states is $\mathcal{S} = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}\}$, where each state corresponds to a city (we use "state" and "city" interchangeably in this example). Travel is only possible along direct routes. An edge $(s, s') \in E$ indicates that the agent can move directly from s to s'. Each such move incurs a non-negative cost given by the edge weight, and we denote this by $\ell(s, s')$. For example, traversing the edge from A to C has cost $\ell(\mathrm{A}, \mathrm{C}) = 4$. Unlike domains with a fixed action set (e.g., grid worlds with $\{\textsc{Up}, \textsc{Down}, \textsc{Left}, \textsc{Right}\}$), the available actions here depend on the current state. Specifically, $\mathcal{A}(s) = \{a_{s \to s'} : (s, s') \in E\}$, where action $a_{s \to s'}$ deterministically transitions the agent to s'. The initial state is $s_1 = \mathrm{A}$, and the goal set is $\mathcal{S}^g = \{\mathrm{F}\}$.

In many domains of interest, the state space is too large to enumerate explicitly. We therefore assume an *implicit* representation in which successor states are generated on demand via the transition model. Search algorithms differ primarily in how they allocate computation across the resulting (implicitly defined) set of partial solution paths.

2.2 Best-First Search

We now review best-first search as a general search procedure for exploring such implicitly defined state spaces. Its central organizing principle is the ordering of frontier nodes by an evaluation function φ, which determines which node is selected at each iteration. This viewpoint is useful for policy tree search, where learned guidance typically affects behaviour through the design of φ and the expansion priorities it induces.

2.2.1 State-Space Graphs and Induced Search Trees

While the underlying problem is defined over a *state-space graph* whose vertices are states and whose edges correspond to actions, many domains of interest have no explicit graph representation available a priori. This setting is often referred to as *implicit graph search*, since the reachable portion of the state space is revealed only through successor generation during search.

It is useful to distinguish a *state* from a *search node*. A state represents a configuration of the environment, whereas a search node represents a particular partial path (*i.e.*, a sequence of actions) from the initial state. We often use the phrase "the state represented by node n" to denote the state reached by executing the action sequence associated with n from the initial state. This distinction matters because the same state may be reachable via multiple action sequences, leading to multiple distinct search nodes that correspond to the same underlying state. This also motivates the distinction between the

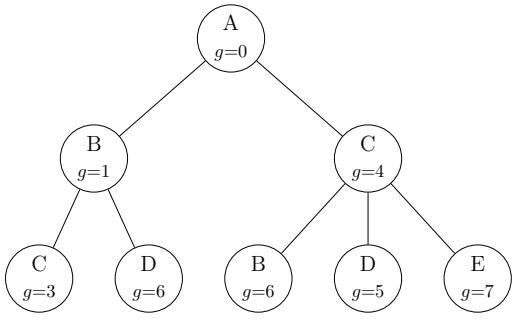


Figure 2.2: A search tree representation corresponding to the toy road network search problem.

state-space graph and the *search tree*. The state-space graph contains a single vertex per state and edges induced by the transition dynamics. In contrast, the problem induces a conceptual search tree whose nodes correspond to action sequences: the root corresponds to the empty sequence at the initial state, and each tree edge corresponds to appending one applicable action to a partial path. Consequently, the same underlying state can appear multiple times in the search tree at different nodes. Figure 2.2 illustrates this distinction for the toy road-network example.

In an implicit setting, the full search tree is not constructed explicitly. Instead, it is a conceptual object induced by the problem: its nodes are the finite applicable action sequences starting from the initial state. A search algorithm generates only a finite subset of this underlying tree during exploration. At any point in time, let $\mathcal{N}_t \subseteq \mathcal{N}$ denote the set of search nodes that have been generated so far. We refer to generated but unexpanded nodes as the frontier (or OPEN list), and expanded nodes as the CLOSED set. Thus, OPEN is the active boundary of the generated portion of the underlying search tree, rather than necessarily the set of all leaves of $\mathcal{N}_t$. In particular, an expanded node in CLOSED may also be a leaf of $\mathcal{N}_t$ if it has no applicable actions, or if no children of that node have been generated. In each iteration, a frontier node $n \in$ OPEN is expanded by generating its children: for each applicable action, the algorithm applies the transition function to the state represented by n and creates a successor node corresponding to the extended action sequence.

We write $\mathcal{N}$ for the set of search nodes in the underlying search tree induced by the problem, and n_1 for the root node corresponding to the empty action sequence at the initial state. Thus, $\mathcal{N}$ denotes the conceptual set of all finite applicable action sequences from the initial state, not necessarily the set of nodes generated by a particular run of a search algorithm. For a node $n \in \mathcal{N}$, let $s(n)$ denote the state reached by executing the partial action sequence represented by n. The children of n, written $\mathcal{C}(n)$, are the successor nodes obtained by appending one applicable action $a \in \mathcal{A}(s(n))$ to that sequence. This notation is used in general domains, where the set of actions need not be the same for every state, as is the case in Example 2.1.1. For the domains

Algorithm 1 `BestFirstSearch`

Input: Root node n_1, evaluation function φ, B budget, goal set $\mathcal{N}^g$
Output: Solution path, or failure
OPEN $\leftarrow \{n_1\}$, CLOSED $\leftarrow \emptyset$, expansions $\leftarrow 0$
while OPEN is not empty **and** expansions $< B$ **do**
    expansions $\leftarrow$ expansions $+ 1$
    $n \leftarrow \arg\min_{m \in \text{OPEN}} \varphi(m)$ ▷ *Select node*
    OPEN $\leftarrow$ OPEN$\backslash\{n\}$
    CLOSED $\leftarrow$ CLOSED $\cup \{n\}$ ▷ *Mark expanded*
    if $n \in \mathcal{N}^g$ **then**
        return n ▷ *Solution state found*
    for each child node $n' \in \mathcal{C}(n)$ **do**
        $\tilde{g} \leftarrow g(n) + \ell\big(s(n), s(n')\big)$ ▷ *Proposed cost to n'*
        if CONSIDER$(n', \tilde{g})$ **then**
            $g(n') \leftarrow \tilde{g}$ ▷ *Commit improved path-cost*
            OPEN $\leftarrow$ OPEN $\cup \{n'\}$ ▷ *Add to frontier*
return `failure`
procedure CONSIDER(node n', proposed cost $\tilde{g}$)
    ▷ *return true if the node is to be considered, false otherwise* ◁
    $m \leftarrow$ `FindByState`(OPEN, $s(n')$) ▷ *node with same state, if any*
    if $m \neq \perp$ **then** ▷ *Previously generated but not expanded*
        if $g(m) \leq \tilde{g}$ **then**
            return `false` ▷ *Existing OPEN node is no worse*
        OPEN $\leftarrow$ OPEN$\backslash\{m\}$ ▷ *Replace with better path*
    $m \leftarrow$ `FindByState`(CLOSED, $s(n')$) ▷ *node with same state, if any*
    if $m \neq \perp$ **then** ▷ *Previously expanded*
        if $g(m) \leq \tilde{g}$ **then**
            return `false` ▷ *Existing expanded node is no worse*
        CLOSED $\leftarrow$ CLOSED$\backslash\{m\}$ ▷ *Reopen on improvement*
    return `true`

considered in this thesis, the set of actions is a fixed set, where all actions are applicable in all states. We thus use a simplified notation $a \in \mathcal{A}$, where the dependency of the particular state is dropped. Finally, $\mathcal{N}^g := \{n \in \mathcal{N} : s(n) \in \mathcal{S}^g\}$ denotes the set of goal nodes in the underlying search tree.

2.2.2 The Best-First Search Framework

The `BestFirstSearch` procedure [81] provides a general template for implicit graph search and is summarized in Algorithm 1. It maintains OPEN as a priority-ordered set of frontier nodes and repeatedly selects the node with min-

imum evaluation value under an ordering rule given by an evaluation function $\varphi : \mathcal{N} \to \mathbb{R}$, where $\mathcal{N}$ denotes the set of search nodes. In each iteration, the selected node $n \in$ OPEN is removed from OPEN and placed into CLOSED, being marked as expanded, then checked for goal satisfaction. If the node n is in the goal set $\mathcal{N}^g$, then the corresponding solution path is returned. This *goal-on-expansion* convention is useful for algorithms such as uniform-cost search and A*, where returning a goal when it is selected for expansion supports the usual optimality argument. In other first-solution settings, the goal test can instead be performed when a node is generated; for example, an implementation of Levin Tree Search can return immediately when a generated child is a goal node, rather than waiting for that child to be selected from OPEN (see Chapter 3). For a search node n, we write $g(n)$ for the cumulative transition cost of the partial path represented by n. Thus, $g(n_1) = 0$, and for any child $n' \in \mathcal{C}(n)$, $g(n') = g(n) + \ell(s(n), s(n'))$.

To expand n, the algorithm generates its children $\mathcal{C}(n)$ by applying the transition function for each applicable action. A newly generated child node n' may correspond to a state that has been encountered previously. How such duplicate states are handled is algorithm-dependent; in Algorithm 1, this logic is abstracted by the procedure CONSIDER, which determines whether a generated node should be inserted into OPEN. For algorithms that maintain best known path costs to states, such as A*, CONSIDER can check whether a node representing $s(n')$ already appears in OPEN or CLOSED. If there already exists a node in OPEN representing $s(n')$ with path cost no greater than the newly discovered cost, then n' is discarded. Otherwise, the incumbent node in OPEN is removed and replaced by n'. An analogous check can be performed for nodes in CLOSED. This often involves a FINDBYSTATE check to determine whether OPEN or CLOSED already contains a node representing $s(n')$. In practice, this can be implemented efficiently by maintaining auxiliary hash maps keyed by a canonical representation of the underlying state, giving expected $\mathcal{O}(1)$-time lookup of previously generated nodes with the same state. This instantiation of CONSIDER is given in the procedure in Algorithm 1. Finally, the retained child node is added to the frontier by inserting it into OPEN

if the generated node is to be considered. This replacement-based duplicate handling is an illustrative graph-search implementation; later policy tree search algorithms may instead use lazy pruning or other duplicate-handling semantics.

2.2.3 Uninformed Best-First Search

Algorithm 1 defines a general best-first search template in which the next node to expand is chosen by minimizing an evaluation function $\varphi(n)$. In the *uninformed* setting, φ depends only on information available from the path constructed so far (*e.g.*, depth or accumulated path cost). Two standard uninformed instantiations are breadth-first search and uniform-cost search.

Breadth-first search. Breadth-first search (BFS) [18, 59, 67] expands nodes in order of increasing depth from the root. Let $d(n)$ denote the depth of node n, which also corresponds to the number of actions in its associated action sequence. Note that this is also the canonical notion of depth, where the root node has a depth of 0. Breadth-first search is obtained from Algorithm 1 by choosing

$$\varphi_{\mathrm{BFS}}(n) = d(n), \tag{2.2}$$

together with a FIFO (First-In, First-Out) tie-breaking rule among nodes with equal depth. Breadth-first search can also move the goal-check into the CONSIDER procedure. Intuitively, breadth-first search explores the search tree level by level. When all step costs are identical (equivalently, when ℓ induces unit-cost transitions), breadth-first search is *complete* (*i.e.* is guaranteed to find a solution if one exists) for finite branching factor and returns a minimum-depth solution, which coincides with a minimum-cost solution under unit costs.

Uniform-cost search (UCS). Uniform-cost search [88] generalizes breadth-first search to settings with non-uniform transition costs by prioritizing nodes by their accumulated path cost. UCS is obtained by choosing

$$\varphi_{\mathrm{UCS}}(n) = g(n), \tag{2.3}$$

so that nodes are expanded in order of increasing path cost. Under non-negative edge costs, once a node representing a state is selected for expansion with minimum g-cost among frontier nodes, its g-cost is final, so reopening expanded states is not required. UCS is equivalent to Dijkstra's algorithm applied to an implicitly defined state-space graph [23]. For finite reachable state-space graphs, this graph-search version is complete and returns a minimum-cost solution under non-negative transition costs. In unbounded state spaces, a standard sufficient condition for completeness for finite-cost solutions is that the branching factor is finite and all transition costs are bounded below by some $\epsilon > 0$.

Finally, breadth-first search can be seen as the unit-cost specialization of UCS at the level of the expansion ordering: if $\ell(s, s') \equiv 1$ for all applicable transitions, then $g(n) = d(n)$, so ordering nodes by increasing path cost is equivalent to ordering them by increasing depth, up to tie-breaking. The two procedures may still differ in implementation details. For example, BFS is often implemented with a goal check when a node is generated, which is safe in the unit-cost setting, whereas UCS is usually stated with the goal check when a node is selected for expansion.

2.3 Heuristics as Guidance

Algorithm 1 highlights that search control is fundamentally a decision about computation, in that at each iteration, the algorithm allocates effort by selecting a node from OPEN according to an evaluation function $\varphi(n)$. In uninformed search, φ depends only on quantities accrued along the current partial plan (*e.g.*, depth or path cost). In contrast, *heuristic* search augments this template with additional information about goal proximity, typically via a function that estimates remaining cost or effort from a state. This section introduces heuristics, surveys how they are traditionally constructed, and then instantiates the best-first template to recover common heuristic search algorithms.

2.3.1 Heuristic Function

A *heuristic* $h : \mathcal{N} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ is a node evaluation function that provides guidance about which parts of the search space are more promising to explore. It most commonly represents a *cost-to-go* estimate that approximates the optimal remaining cost from a state s to any goal state. Heuristics differ from the path cost $g(n)$ in that $g(n)$ measures the cost already incurred along the partial plan that produced n, whereas $h(n)$ estimates the additional solution cost still required to reach a goal. This separation is useful because it allows search control to be expressed as different ways of trading off *progress so far* versus *predicted effort remaining*.

While not used directly in this thesis, there are some properties of heuristic functions which are useful for particular algorithms. Let $h^*(n)$ be the true minimal cost to a goal node starting in node n, with $h^*(n) = \infty$ if no goal node is reachable from n. In practice, learned heuristics usually produce finite values and therefore approximate this extended-valued target. A heuristic h is called *admissible* if it never overestimates the true cost-to-go, *i.e.*, $h(n) \leq h^*(n)$ for all n. A stronger condition is *consistency* (or monotonicity), which requires that $h(n^*) = 0$ for all goal nodes n^* and $h(n) \leq \ell(s(n), s(n')) + h(n')$ for every node n and its child n'. It can be shown that consistency implies admissibility (see [26], Theorem 1.2).

2.3.2 Designing Heuristics

Heuristics are designed to be cheap because their purpose is to inform *how* computation is allocated, not to replace computation entirely. Since h is queried at scale during search, its evaluation cost should remain small, otherwise the algorithm spends more time *predicting* which nodes are promising than exploring them. A perfect heuristic would coincide with the optimal cost-to-go h^*, but obtaining h^* generally requires solving a problem of comparable difficulty, eliminating the computational advantage of heuristic guidance. There are a few broad design paradigms for designing heuristics:

Rules of thumb and domain knowledge. Heuristic functions can be hand-crafted by leveraging domain knowledge, such as the number of misplaced tiles in the sliding-tile puzzle, or simple deadlock rules in Sokoban (e.g., treating a box pushed into a non-goal corner as effectively unsolvable). Such heuristics can be effective, but they require substantial expertise and may be brittle when problem structure changes.

Abstractions and relaxations. A systematic approach is to derive heuristics from simplified versions of the problem that can be solved efficiently. This includes: (i) *abstractions*, which map the original state-space into a smaller space, using exact distances there as estimates, and (ii) *relaxations*, which remove constraints so that the goal distances become easier to compute. Pattern databases [20, 29] are a prominent example of abstraction-based heuristics, which precompute goal distances for an abstracted subproblem and look up the resulting value during search time. As a simple relaxation example, in 2D grid pathfinding one can ignore obstacles (*i.e.*, treat blocked cells as traversable), yielding Manhattan distance to the goal as an efficient estimate.

Learned heuristics. Modern systems increasingly learn heuristic guidance from data. A learned heuristic typically approximates $h^*(s)$ or another proxy signal (e.g., expected remaining steps, probability of solvability, or a bounded suboptimal cost-to-go), using supervised learning from solved instances [30], bootstrapping from search traces [6, 7], or reinforcement learning [2, 3]. Learned heuristics can provide strong guidance in complex domains where handcrafted abstractions are hard to design. However, they usually do not satisfy admissibility by default, so their role is often to improve efficiency rather than to preserve strict optimality guarantees.

2.3.3 The Bootstrap Process

This subsection gives a more thorough treatment of the Bootstrap process, which will be used extensively to learn heuristics and policies in later chapters.

Algorithm 2 The Bootstrap Process

Input: Initial budget B_0, training problems $\mathcal{N}^T$, validation problems $\mathcal{N}^V$, search algorithm S, neural policy/heuristic models Θ
`solved` $\leftarrow \emptyset$
for $t = 0, 1, \ldots$ **do**
    for each $n_1 \in \mathcal{N}^T$ **do**
        `result` $\leftarrow S(n_1, \Theta, B_t)$ ▷ *Perform budgeted search*
        if `result` $\neq$ `timeout` **and** `result` $\neq$ `no_solution` **then**
            `solved` $\leftarrow$ `solved` $\cup \{n_1\}$
            update Θ using solution trajectory from `result`
    if all $n_1 \in \mathcal{N}^V$ solved by $S(n_1, \Theta, B_t)$ **then**
        break ▷ *Bootstrap complete*
    `choose budget` B_{t+1}

The Bootstrap algorithm [7] learns a neural policy and/or heuristic function from search. The initial policy/heuristic is represented by a randomly initialized neural network, and they are improved in each iteration as follows. A search algorithm using the current neural model runs on a subset of the training problem instances with a bound on the number of node expansions allowed for each problem. Problems that are solved have their solution trajectory used to update the models, and the process continues. After every sweep through a set of training problems, a separate validation set is used to determine when to stop. If the search can solve all problems from the validation set, then the training is complete and the neural model is returned. If the search cannot make any *meaningful progress* after a sweep of the training set, the expansion budget is increased. Naïvely doubling the budget only when no new problems are solved is one possible update rule, but it can be slow when the current budget solves a small number of additional problems on each sweep. In that case, the algorithm is making progress, so the budget is not increased, but the progress may be too slow to efficiently solve the remaining training instances. Another option, which is what is used throughout this thesis, is to follow the budget update schedule in [76] where if less than a factor $(1 + b)$ of problems are solved at any iteration t compared to the previous iteration, then the budget is updated to $B_{t+1} = 2B_t + T_t^+ / s_t^-$, where T_t^+ is the total number of expansions of solved problems at iteration t and s_t^- is the number of remaining

unsolved problems at iteration t. See Algorithm 2 for its pseudocode.

The motivation for this update rule is that making any progress is not always enough evidence that the current budget is appropriate. For example, suppose $B_t = 1{,}000$, the training set currently has 100 unsolved problems, and one sweep solves only one additional problem. A rule that increases the budget only when no new problems are solved would keep the budget fixed at 1,000, even though at this rate many sweeps may be needed before the remaining problems become solvable. By contrast, the update above treats such slow progress as a signal that the search budget should increase. If the solved problems in the sweep required $T_t^+ = 99{,}000$ total expansions and $s_t^- = 99$ problems remain unsolved, then the update gives

$$B_{t+1} = 2B_t + \frac{T_t^+}{s_t^-} = 2{,}000 + 1{,}000 = 3{,}000.$$

Thus, the budget is increased even though some progress was made. The term $2B_t$ provides a coarse increase in search depth, while T_t^+/s_t^- uses the observed expansion cost of recently solved problems to scale the increase according to the number of remaining unsolved instances.

2.3.4 Heuristic Search

In Algorithm 1, heuristic guidance is incorporated by choosing evaluation functions $\varphi(n)$ that depend on a cost-to-go estimate $h(n)$ in addition to path information such as $g(n)$ or depth.

Greedy best-first search (GBFS). Greedy best-first search [25] prioritizes nodes that *appear closest to the goal* under the heuristic:

$$\varphi_{\mathrm{GBFS}}(n) = h(n). \tag{2.4}$$

GBFS is often effective when h strongly correlates with true goal proximity, but it can commit to misleading regions of the state space when the heuristic is inaccurate.

A* search. A* search [42] balances the cost accumulated thus far with the predicted remaining cost:

$$\varphi_{\mathrm{A}^*}(n) = g(n) + h(n). \tag{2.5}$$

With a goal test on expansion, an admissible h, and the standard finiteness conditions stated above for UCS, A* returns an optimal finite-cost solution [42]. When h is consistent, A* only expands nodes it has found an optimal path to, and therefore never needs to reopen a closed node. Consistency eliminates reopening but does not dictate how duplicates in OPEN are represented. In a keyed OPEN implementation such as Algorithm 1, an unexpanded node is replaced when a better path is generated. Alternatively, a lazy implementation may retain multiple OPEN entries and discard an entry when it is selected if its state has already been expanded.

Weighted variants. When solution quality can be traded for speed, a common relaxation is Weighted A* search (or WA*) [83]:

$$\varphi_{\mathrm{WA}^*}(n) = g(n) + w \cdot h(n), \tag{2.6}$$

where $w > 1$, which biases the search more strongly toward heuristic guidance. WA* does not guarantee an optimal solution; instead, with an admissible heuristic it provides a bounded-suboptimality guarantee where the solution found is no more than w times the optimal solution cost [83]; see also Thayer and Ruml [100] for a concise modern presentation of this argument.

2.4 Policies as Guidance

Algorithm 1 likewise helps clarify the role of policies in search. As with heuristics, the expansion decision depends on the information used to prioritize candidates, but policies provide a different kind of signal. Rather than estimating how far a state is from a goal, a policy expresses preferences over which actions appear most promising from the current state. This section introduces policies, reviews how they are designed or learned, and then discusses the main ways they can be incorporated into search.

2.4.1 Policy Function

A *policy* $\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a mapping that assigns preferences to actions available in a state, which is written as $\pi(a \mid s)$ where $a \in \mathcal{A}(s)$, $\pi(a \mid s) \geq 0$, and $\sum_{a \in \mathcal{A}(s)} \pi(a \mid s) = 1$. Thus, $\pi(\cdot \mid s)$ is a probability distribution over the actions applicable in state s.

Unlike a heuristic or cost-to-go function, which estimates how much cost remains before reaching a goal, a policy assigns local preferences to the actions available in the current state. Its purpose is therefore not to predict the remaining solution cost, but to bias computation toward actions that are more likely to lead to progress. Action ranking is important, but for Levin-style search the usefulness of a policy also depends on probability scale and support: path probabilities multiply along a trajectory, so equally ranked policies can induce very different search orders and bounds. Later, when working directly in the induced search tree, we will equivalently view the policy as a distribution over child nodes, writing $\pi(n' \mid n) := \pi(a \mid s(n))$ where $n' = na$ is the result of applying action a to node n.

2.4.2 Designing Policies

As with heuristics, the practical goal in designing a policy is not to recover an ideal decision rule exactly, but to obtain a source of guidance that is substantially cheaper to evaluate than the search effort it helps direct. The policy serves as a compact rule for assigning local preferences to available actions, ideally capturing enough structure about promising decisions.

In reinforcement learning, one common route is *value-based* learning, in which action values are estimated and the policy is induced by favouring actions with high estimated return, as in Q-learning [106]. Another is *direct policy optimization*, in which the policy itself is parameterized and updated so as to increase expected return, as in REINFORCE and related policy-gradient methods [107]. These two perspectives differ in whether one first learns action values and derives a policy from them, or instead optimizes the policy directly, but both provide general mechanisms for constructing action-selection

rules from experience.

Policies can also be learned from demonstrations. In *imitation learning*, the policy is trained to reproduce expert decisions [9, 84]. In *inverse reinforcement learning*, demonstrations are used to infer a reward or cost representation, from which a policy is obtained by solving or approximating the induced control problem [72]. For the purposes of search, these approaches provide a way to train policies that encode useful action preferences without manually specifying them by hand.

2.4.3 Policies in Search

Policies are used in a variety of search procedures beyond policy tree search. One important example is Monte Carlo Tree Search (MCTS), where simulations from a frontier node are completed using a *default policy* [15, 54]. In the simplest case this rollout policy may be random, but a more informative policy can steer simulations toward more promising action sequences [93]. A second example is beam search in sequence modelling. There, a model defines a distribution over the next symbol given the current partial output, and beam search repeatedly extends partial hypotheses while retaining only the highest-scoring continuations [97]. In this sense, the model acts as a policy over local continuations, and search uses those policy scores to decide which branches of the exponentially large sequence space remain under consideration.

Taken together, these examples illustrate a broader point: a policy need not define a standalone reactive controller. It can also act as a computational guide inside a larger search procedure, shaping successor ordering, sampling, rollout behaviour, or frontier prioritization. Moreover, because a policy induces probabilities over action sequences, it can support search guarantees stated in terms of the probability mass assigned to solution paths, such as upper bounds on the amount of the search tree that must be explored before reaching a solution. This computational role of policies is the one most relevant for the present thesis and motivates the policy-guided search methods developed in later chapters.

Chapter 3

Foundations of Policy Tree Search

Chapter 2 introduced the background needed for the present thesis, including the role of policies as a source of learned guidance in search. This chapter turns to policy tree search itself. It develops the perspective adopted in this thesis, explains the setting in which policy tree search arises, and surveys the major forms these methods take.

3.1 A Place for Policy Tree Search

Policy tree search emerged from the convergence of several lines of work. It draws on the use of policies as local guidance, on search procedures built for making good decisions under limited computation, and on the practical difficulty of deterministic domains where useful feedback is scarce and solutions may lie deep in the tree. This section traces that convergence and shows why it opens space for a distinct family of methods.

3.1.1 The Road to Policy Tree Search

Policy tree search emerged from combining ideas that had largely developed separately. Monte Carlo Tree Search showed that policies could be integrated effectively into tree search, particularly in stochastic and adversarial settings with complete information. UCT-like rules have asymptotic guarantees under their stated tabular tree-search assumptions [54, 87]. Practical PUCT-based systems such as AlphaGo [94] and MuZero [90] combine policies with learned value estimates and, in the latter case, learned models; the corresponding

guarantees do not automatically extend to function approximation, learned models, or finite search. These distinctions become especially important in the deterministic *first-solution* setting considered here, where the goal is to find any solution (not necessarily the optimal solution) and value estimates may remain uninformative until a solution is reached.

3.1.2 Why Sampling is not Enough

Deterministic single-agent search, by contrast, has been shaped primarily by heuristic methods for goal-directed search. This thesis is concerned with a particularly difficult subset of these problems often referred to as *needle-in-the-haystack* domains, in which finding even a single solution is challenging. For this setting, the sampling procedure used in MCTS is often poorly matched to the task. In sparse-reward problems with little informative intermediate feedback, unsuccessful partial trajectories may all receive similarly uninformative evaluations. MCTS can then spend substantial effort reallocating samples among many competing branches without obtaining the signal needed to commit search deeply along the rare trajectory that leads to a solution [68]. Incorporating a policy prior, as in PUCT, can improve the initial ranking of actions, but it does not fundamentally resolve this issue in that the search still relies on repeated sampling to separate branches by estimated value. When those estimates remain uninformative until a full solution is reached, even a strong prior may only bias how effort is spread across shallow alternatives rather than induce the depth-focused behaviour needed for rapid first-solution discovery. This limitation points toward the need for search-control mechanisms that are more directly aligned with first-solution search in deterministic domains. What is needed, then, is a way to use learned guidance without waiting for value estimates to become informative.

3.1.3 Following Probability Mass

It is in this context that policy tree search was introduced. Rather than using a policy primarily to sample rollouts or support value estimation, policy tree search treats the policy itself as the central source of search guidance. In this

sense, it can be viewed as a form of heuristic search in which a policy replaces the heuristic as the main learned input to the search procedure. The key advantage of this perspective is that, at each fixed depth, a policy defines a probability distribution over action-sequence prefixes. This makes it possible to reason directly about the search effort required to discover a solution: one can derive bounds on the number of node expansions in terms of the probability assigned to solution trajectories. These guarantees are qualitatively different from the classical guarantees associated with value-based guidance. In heuristic search, theoretical analysis is usually expressed in terms of the ordering induced by estimated costs, together with properties such as completeness, optimality, or bounded suboptimality. For example, methods such as A* are characterized by which states must be expanded under the f-value ordering relative to the cost of an optimal solution, so the effectiveness of the guidance is understood through how well estimated cost-to-go values prioritize states. Policy tree search offers a different perspective. Because a policy assigns probability mass to action-sequence prefixes, search effort can instead be related directly to the first-hitting probability mass of trajectories that reach a goal. Thus, rather than analyzing guidance through the accuracy of state-based value estimates, policy tree search analyzes guidance through how probability mass is allocated over solution paths.

3.2 Notation

Before introducing the main policy tree search algorithms, we briefly fix the node and tree notation used throughout this chapter and the remaining thesis. We build on the search-tree notation introduced in Chapter 2 and restate only the parts needed here for readability. As in Chapter 2, a node $n \in \mathcal{N}$ represents a partial action sequence from the root n_1, and $d(n)$ denotes its depth with $d(n_1) = 0$. The children of n are $\mathcal{C}(n) := \{na \mid a \in \mathcal{A}(s(n))\}$, where na denotes the node obtained by appending action a to the sequence represented by n. The set of goal nodes is given by $\mathcal{N}^g$. For a node n, $\mathrm{anc}(n)$ denotes its set of ancestors and $\mathrm{anc}_+(n) := \mathrm{anc}(n) \cup \{n\}$. Similarly, $\mathrm{desc}(n)$ denotes its

set of descendants and $\mathrm{desc}_+(n) := \mathrm{desc}(n) \cup \{n\}$. We also use the notation $n' \prec n$ for $n' \in \mathrm{anc}(n)$, and $n' \preceq n$ for $n' \in \mathrm{anc}_+(n)$.

The policy introduced in Chapter 2 can equivalently be viewed as a distribution over child nodes. Specifically, for a child $n' \in \mathcal{C}(n)$, define $\pi(n' \mid n) := \pi(a \mid s(n))$, where $n' = na$ is the result of applying action a to node n. This induces *probabilities over paths* in the search tree. For nodes $n \preceq n'$, let $\pi(n' \mid n)$ denote the probability of generating the suffix path from n to n' under π, with $\pi(n \mid n) = 1$. Equivalently, if $n = n_i \preceq n_{i+1} \preceq \cdots \preceq n_j = n'$ is the path from n to n', then

$$\pi(n' \mid n) = \prod_{k=i}^{j-1} \pi(n_{k+1} \mid n_k). \tag{3.1}$$

If $n \not\preceq n'$, define $\pi(n' \mid n) = 0$. For convenience, we write $\pi(n) := \pi(n \mid n_1)$ for the probability of the full root-to-n path. When $\pi(n) > 0$, the suffix probability also satisfies

$$\pi(n' \mid n) = \frac{\pi(n')}{\pi(n)}. \tag{3.2}$$

We allow policies to assign zero probability to some actions. A node n' is said to be *policy-reachable* from n if $\pi(n' \mid n) > 0$, and n' is policy-reachable from the root if $\pi(n') > 0$.

A set of nodes is *prefix-closed* if, whenever it contains a node, it also contains every ancestor of that node. Since nodes correspond to partial action sequences, this is equivalent to requiring that every prefix of a represented node sequence is also contained in the set. A *search tree* $\mathcal{T} \subseteq \mathcal{N}$ can now be characterized as any prefix-closed set of nodes such that, if $n \in \mathcal{T} \cap \mathcal{N}^g$, then no descendant of n belongs to $\mathcal{T}$. The leaves of $\mathcal{T}$ are denoted by $\mathcal{L}(\mathcal{T})$, *i.e.*, the nodes in $\mathcal{T}$ with no descendants in $\mathcal{T}$. We overload $\ell(n)$ slightly: as in Chapter 2, $\ell(s, s')$ denotes the transition cost in the underlying state space, while in this chapter $\ell(n)$ denotes the cost incurred when expanding node n. For a search algorithm S, define $L(S, n)$ as the cumulative expansion cost incurred up to and including the expansion of n. In the unit-cost case $\ell(n) \equiv 1$, $L(S, n)$ reduces to the number of node expansions.

The thesis uses the resource-accounting conventions in Table 3.1. They

Table 3.1: Resource-accounting conventions used throughout this thesis.

Measure	**Counted quantity**
Sampled transition	One policy-selected action during a rollout.
Generated node	Creation of a child search-tree node.
Node expansion	Selection of a search-tree node for expansion and successor processing.
Node visit	A node expansion in the $\sqrt{\text{LTS}}$ literature.
Search loss $L(S, n)$	Sum of the specified expansion losses; it equals the number of node expansions only when $\ell(n) \equiv 1$.
Aggregate computation time	Sum of computation time across problem runs and worker threads.
Wall-clock time	Elapsed real time for an experiment; it is not summed across worker threads.

distinguish work performed by rollout-based methods from work performed by best-first methods and from elapsed experiment time.

3.3 Sampling-Based Policy Tree Search

The most direct way to use a policy for first-solution search is to sample trajectories from it. If the policy assigns substantial probability mass to goal-reaching trajectories, then repeated sampling already provides a plausible search strategy. Unlike MCTS-style methods, the policy is no longer used merely to bias rollouts whose values must later be estimated and compared. Instead, the policy itself determines which trajectories are likely to be explored. This makes it possible to analyze search effort in terms of the cumulative probability mass assigned to solutions.

For the sampling-based algorithms in this section, we count search effort in units of sampled transitions. Thus, a call to `sample_trajectory` with depth parameter d has budget at most d: the initial check of the root node is treated as bookkeeping and is not charged as a sampled transition. This convention matches the depth parameter used by the restart schedules below. In the best-first algorithms introduced later, search effort will instead be counted in the usual way by expanded search-tree nodes. Throughout this section, $L(S, n^*)$

Algorithm 3 `sample_trajectory`

Input: Root node n_1, policy π, depth d, goal set $\mathcal{N}^g$
Output: Solution path, or failure
$n \leftarrow n_1$
if $n \in \mathcal{N}^g$ **then**
    return n
for $i = 1, \ldots, d$ **do**
    if $\mathcal{A}(s(n)) = \emptyset$ **then**
        return `failure`
    $a \sim \pi(\cdot \mid n)$ ▷ *Sample action according to policy*
    $n \leftarrow na$ ▷ *Extend sampled path by action a*
    if $n \in \mathcal{N}^g$ **then**
        return n
return `failure`

is interpreted under the sampled-transition cost convention described above. The sampling procedure is outlined in Algorithm 3. Since nodes correspond to sequences of actions from the root, we use the notation na to represent the child node of n as a result of applying action a.

Before introducing two sampling-based policy tree search algorithms, we first introduce some shared notation that will be used to provide bounds on the expected number of sampled transitions required to find a solution. Let

$$G_d := \{n \in \mathcal{N}^g \ : \ d(n) \leq d \text{ and } \mathrm{anc}(n) \cap \mathcal{N}^g = \emptyset\}, \tag{3.3}$$

$$\pi_d^+ := \sum_{n \in G_d} \pi(n). \tag{3.4}$$

If $\pi_d^+ = 0$, then no first-hitting goal node in G_d is reached with positive probability by a policy-generated trajectory of depth at most d. Thus, G_d is the set of first-hitting goal nodes reachable in at most d steps, and π_d^+ is the probability that a policy-generated trajectory reaches a goal within at most d steps.

3.3.1 Fixed-Depth Sampling: multiTS

A simple form of this idea is to sample trajectories up to a fixed depth bound and repeat until a solution is found. When a useful upper bound on a solution depth $d_{\max}$ is known, such repeated sampling can exploit the cumula-

Algorithm 4 multiTS

Input: Root node n_1, policy π, budget B, depth $d_{\max}$, goal set $\mathcal{N}^g$
Output: Solution path, or failure
while $B > 0$ **do**
    $d \leftarrow \min(d_{\max}, B)$ ▷ *rollout length budget allows*
    $r \leftarrow$ `sample_trajectory`$(n_1, \pi, d, \mathcal{N}^g)$
    if r **is not** `failure` **then**
        return r ▷ *Solution path*
    $B \leftarrow B - d$ ▷ *Update remaining budget*
return `failure`

tive probability mass of all goal-reaching trajectories within that horizon. In particular, this strategy can be effective when many different solution paths exist, each with modest individual probability. This strategy is captured by the multiTS algorithm [78], and is given in Algorithm 4. It repeatedly calls `sample_trajectory`, so long as the total budget permits.

From this simple algorithm, we can already start to provide bounds on the search effort in terms of sampled transitions. Due to the sampling nature of multiTS, the bound is given in expectation.

Theorem 3.3.1 (Orseau et al. [78], Theorem 4, adapted)**.** *If* $\pi^+_{d_{\max}} > 0$*, the expected number of sampled transitions multiTS requires before returning a (random) first-hitting goal node* $n^* \in G_{d_{\max}} \subseteq \mathcal{N}^g$ *is bounded by*

$$\mathbb{E}\left[L(\text{multiTS}, n^*)\right] \leq \frac{d_{\max}}{\pi^+_{d_{\max}}}. \tag{3.5}$$

If $\pi^+_{d_{\max}} = 0$*, no finite bound of this form is possible.*

The proof is given in Appendix A.1. The difficulty is that in most search problems, an appropriate depth bound is not known in advance. If the bound is too small, all solutions may be excluded; if it is too large, computation may be wasted on trajectories far longer than necessary.

3.3.2 Unknown Depth and Universal Restarts: LubyTS

Luby Tree Search (LubyTS) [78] addresses this difficulty by combining policy-guided trajectory sampling with a *universal restart schedule*. Rather than

Algorithm 5 LubyTS

Input: Root node n_1, policy π, budget B, goal set $\mathcal{N}^g$
Output: Solution path, or failure
$i \leftarrow 1$
while $B > 0$ **do**
    $d \leftarrow \min(\texttt{A6519}(i), B)$ ▷ *rollout length budget allows*
    $r \leftarrow \texttt{sample\_trajectory}(n_1, \pi, d, \mathcal{N}^g)$
    if r **is not** `failure` **then**
        return r ▷ *Solution path*
    $B \leftarrow B - d$ ▷ *Update remaining budget*
    $i \leftarrow i + 1$
return `failure`

committing to a single sampling depth, it allocates trials across depths in a way that remains effective when the relevant solution depth is unknown.

To motivate LubyTS, consider a randomized procedure ρ whose halting time is random and distributed according to some unknown distribution p, where halting corresponds to solving the underlying problem. The objective is to design a restart schedule that runs ρ multiple times with varying time limits so as to minimize the total expected computation before halting. If ρ were known, then the best strategy would be to restart ρ repeatedly with a fixed cutoff t_p chosen specifically for that distribution: each run is allowed to continue for at most t_p steps, after which it is terminated and restarted if it has not already halted. Luby et al. [64] showed that this fixed-cutoff strategy is optimal (up to constant factors). Luby et al. also introduced a *universal restart strategy* based on a sequence of running times known as A182105[1]:

1 1 2 1 1 2 4 1 1 2 1 1 2 4 8 1 1 2 1 1 2 4 1 1 2 1 1 2 4 8 16 1 1 2 ...

Universal restart strategies repeatedly return to small cutoffs, but also periodically try larger powers of two. It can be seen as a strategy which hedges across all possible solution depths. Orseau et al. [78] proposed to use a sequence A6519[2] instead:

1 2 1 4 1 2 1 8 1 2 1 4 1 2 1 16 1 2 1 4 1 2 1 8 1 2 1 4 1 2 1 32 1 2...

[1] https://oeis.org/A182105
[2] https://oeis.org/A006519

noting that it is simpler to compute using $\text{A6519}(n) := ((n \text{ XOR } (n-1))+1)/2$. LubyTS, which is given in Algorithm 5, follows this strategy, repeatedly calling `sample_trajectory` using depths from A6519, so long as the budget permits. Similar to multiTS, a bound on the search effort in terms of sampled transitions for LubyTS is given as an expectation.

Theorem 3.3.2 (Orseau et al. [78], Theorem 6, adapted)**.** *The expected number of sampled transitions LubyTS requires before reaching a (random) first-hitting goal node $n^* \in G_d \subseteq \mathcal{N}^g$ is bounded by*

$$\mathbb{E}[L(\text{LubyTS}, n^*)] \leq \min_{\substack{d \in \mathbb{N}_1 \\ \pi_d^+ > 0}} d + \frac{d}{\pi_d^+} \left(\log_2 \frac{d}{\pi_d^+} + 6.1 \right). \tag{3.6}$$

If there is no d such that $\pi_d^+ > 0$, then the policy assigns zero probability to every finite-depth goal-reaching trajectory, and the theorem provides no finite guarantee.

The proof is given in Appendix A.2. The bound for LubyTS incurs an extra logarithmic overhead compared to multiTS as a result of adapting to the unknown depth. This makes LubyTS attractive in settings where solutions are numerous and their collective probability mass is large, even if no single solution trajectory is especially likely.

Sampling methods show that a policy can be used directly to guide first-solution search, without relying on value estimates to become informative. Their strength is that they can exploit the cumulative probability mass of many solution trajectories, and LubyTS in particular avoids committing to a single depth bound. At the same time, they allocate computation at the level of complete sampled trajectories, so unsuccessful trials provide only limited reusable structure. This suggests a second family of policy tree search methods in which the policy is used not to sample trajectories, but to determine which frontier node should be expanded next.

3.4 Best-First Policy Tree Search

This leads to the best-first view of policy tree search. Rather than drawing complete trajectories from the policy, one can use the policy to order nodes in

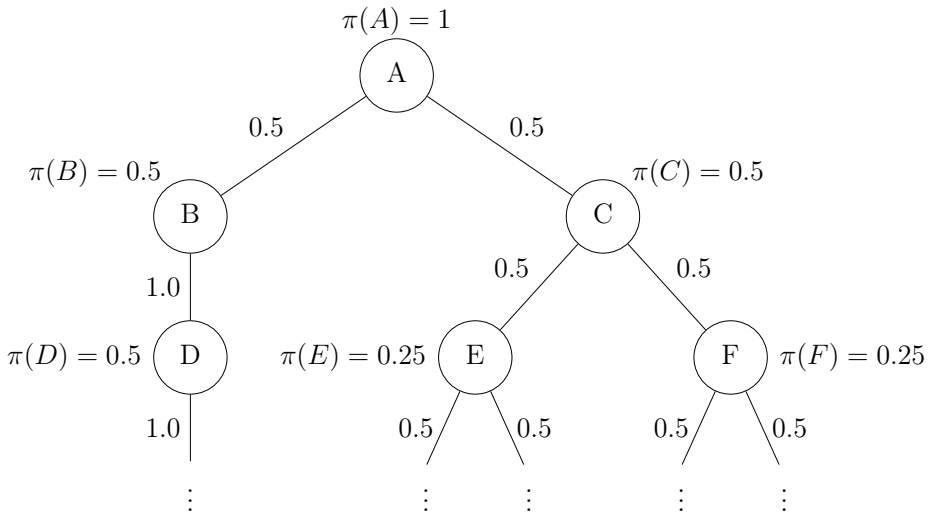


Figure 3.1: A search tree showcasing pitfalls of using path probabilities for the cost function in best-first search.

an explicit search tree. A natural first attempt is to expand nodes in decreasing order of path probability, or equivalently in increasing order of the cost $\varphi_\pi(n) = 1/\pi(n)$. However, path probability alone is not sufficient, because it ignores the amount of search effort already required to reach a node. As a result, highly probable prefixes can dominate the search indefinitely even when they do not lead efficiently toward a goal.

Example 3.4.1. Consider the example tree in Figure 3.1. The subtree rooted in B is an infinite chain in which each edge has a conditional probability of 1, so every node n on that branch has a path probability of $\pi(n) = 1/2$, and therefore cost $\varphi_\pi(n) = 2$. The subtree rooted in C, by contrast, is an infinite binary tree in which each child has conditional probability $1/2$. Hence a node n in that subtree at depth $d(n)$ has path probability $\pi(n) = 2^{-d}$, so every node below depth 1 in the right subtree has cost strictly greater than 2. Thus, after expanding B and C, a best-first search ordered by $\varphi_\pi(n)$ will continue selecting nodes from the left chain, since they always have a lower cost than any deeper node in the right subtree. If the goal nodes $n \in \mathcal{N}^g$ all lie in the right subtree below depth 1, then a search never expands a goal node, despite those nodes having non-zero probability.

3.4.1 Levin Tree Search

Levin tree search (LevinTS or LTS) [78] addresses this failure of pure probability ordering by balancing path probability against path length. Rather than ordering nodes solely by $\pi(n)$, it also incorporates the node depth to account for nodes which increase in depth but remain constant in path probability.

Definition 3.4.2. The *Levin cost* or *LTS cost* is given by

$$\varphi_{\mathrm{LTS}}(n) = \tfrac{d}{\pi}(n) = \frac{d(n)}{\pi(n)}. \tag{3.7}$$

When $\pi(n) = 0$, we define $\tfrac{d}{\pi}(n) = +\infty$. Thus, only policy-reachable nodes have finite Levin cost.

LTS is a best-first search algorithm using φ_{LTS} as the evaluation function, and is given in Algorithm 6. In the example in Figure 3.1, the added depth term ensures that the infinite left branch does not retain constant priority forever. Its nodes become progressively less favourable with depth, and nodes in the right subtree are therefore eventually expanded. Furthermore, the Levin cost φ_{LTS} is a *monotone* cost function.

Theorem 3.4.3 (Orseau et al. [78], Theorem 2, adapted)**.** *Let $c(n) = d(n)/\pi(n)$ when $\pi(n) > 0$, and $c(n) = +\infty$ when $\pi(n) = 0$. Then c is a monotone cost function,* i.e.*, if $n' \in \mathcal{C}(n)$ then $c(n') \geq c(n)$.*

Proof. Let n' be a child of n. Then, $d(n') = d(n)+1$ and $\pi(n') = \pi(n)\pi(n' \mid n)$, with $0 \leq \pi(n' \mid n) \leq 1$. If $\pi(n) = 0$, then $\pi(n') = 0$, so both $c(n)$ and $c(n')$ are $+\infty$ under the convention above. If $\pi(n) > 0$ but $\pi(n' \mid n) = 0$, then $\pi(n') = 0$, so $c(n') = +\infty \geq c(n)$. Finally, if $\pi(n) > 0$ and $\pi(n' \mid n) > 0$, then

$$c(n') = \frac{d(n)+1}{\pi(n)\pi(n' \mid n)} \geq \frac{d(n)+1}{\pi(n)} \geq \frac{d(n)}{\pi(n)} = c(n).$$

Hence, the cost cannot decrease from a node to any of its children, and so c is monotone. ■

Intuitively, LTS prefers nodes that are both likely under the policy and reachable by short prefixes, preventing the search from being dominated by

Algorithm 6 Levin Tree Search (LTS)

1: **Input:** Root node n_1, budget B, goal set $\mathcal{N}^g$
2: **Output:** Solution path, or failure
3: **return** `BestFirstSearch`$(n_1, \varphi_{\text{LTS}}, B, \mathcal{N}^g)$

highly probable but unproductive branches. In deterministic domains, this yields a systematic best-first policy-guided search procedure with a bound on the number of node expansions before reaching a goal.

Theorem 3.4.4 (Orseau et al. [78], Theorem 3, adapted)**.** *The number of node expansions LTS requires before reaching the goal node with minimal Levin cost $n^* \in \mathcal{N}^g$ is bounded by*

$$L(\text{LTS}, n^*) \leq 1 + \frac{d(n^*)}{\pi(n^*)}. \tag{3.8}$$

The bound is finite only when the selected goal node n^* is policy-reachable, *i.e.*, when $\pi(n^*) > 0$. The proof is given in Appendix A.3.1. Theorem 3.4.4 shows that the Levin cost is not merely a priority rule. For a goal node n^*, the quantity $\varphi_{\text{LTS}}(n^*)$ upper-bounds the amount of search effort required to reach that node. In this sense, the evaluation function serves as a proxy for first-solution search effort rather than an unrelated priority score. This perspective will be important in later sections in this chapter which derive other cost functions and bounds for various extensions of policy tree search.

State Pruning The state pruning checks in `BestFirstSearch` (Algorithm 1) need to be modified for LTS to ensure nodes are expanded in best-first order and their lowest cost first.

Theorem 3.4.5 (Orseau et al. [78], Theorem 2, adapted)**.** *Assume the search problem is deterministic and the policy is state-based. Let $n, m \in \mathcal{N}$ be two nodes that represent the same state s. Suppose that*

$$d(m) \leq d(n) \qquad \textit{and} \qquad \pi(m) \geq \pi(n).$$

Then n can be safely pruned for LTS: for every descendant $n' \in \text{desc}(n)$, there exists a corresponding descendant $m' \in \text{desc}(m)$, obtained by following

the same action suffix from m that leads from n to n', such that m' and n' represent the same state and $\frac{d}{\pi}(m') \leq \frac{d}{\pi}(n')$. In particular, if $n', m' \in \mathcal{N}^g$, pruning n cannot remove a solution whose $\frac{d}{\pi}$ cost is smaller than all solutions still reachable through m.

The proof is in Appendix A.3.2, along with Algorithm 15 which provides the *lazy* state pruning check which is done when nodes are removed from OPEN. The overhead for state pruning is a map where the keys are the previously expanded states and the values are a retained (d, π)-pair for that state, where pruning is permitted only when the retained pair has no greater depth and no smaller path probability.

The given pruning rule allows the search to *prune* redundant nodes while keeping the LTS guarantees. In practice, however, it is common to prune nodes during expansion if a node representing the same state has been previously expanded. While the theoretical guarantees no longer hold, the implementation is simpler and has a smaller computation cost per expansion, and is what is used throughout the thesis.

Numerical Stability The path probabilities $\pi(n)$ may decrease exponentially with depth, which can lead to numerical underflow. Since applying a monotone transformation preserves the expansion order induced by the evaluation function, it is often preferable to compute Levin-style costs in log-space:

$$\ln \varphi_{\text{LTS}}(n) = \ln d(n) - \ln \pi(n).$$

The root is handled separately and expanded first; the log-space expression applies only to non-root nodes, for which $d(n) > 0$. Moreover, if n' is a child of n, then the log-path probability can be updated incrementally as $\ln \pi(n') = \ln \pi(n) + \ln \pi(n' \mid n)$. If $\pi(n) = 0$, the node is assigned infinite Levin cost; the log-space formula is used only for nodes with $\pi(n) > 0$.

3.4.2 Policy-Guided Heuristic Search

LTS shows that a policy can do more than guide sampling: it can induce a systematic best-first ordering over explicit search nodes, with guarantees

stated directly in terms of the policy mass assigned to a solution path. Yet LTS remains policy only. In many deterministic domains, however, heuristic estimates of remaining difficulty can also be highly informative, as discussed in Chapter 2. This suggests a natural extension: rather than choosing between policy guidance and heuristic guidance, one can ask whether both can be combined within a single best-first policy tree search procedure, while still analyzing performance in terms of search effort rather than solution cost. This is the perspective taken by Policy-Guided Heuristic Search (PHS) [79], of which PHS* is the more aggressive variant.

Policy-guided Heuristic Search extends LTS by augmenting the LTS cost function with an explicit heuristic factor. To help in this discussion, we introduce a new term for the cost of expanding nodes along a path. Note that this is distinct from the depth $d(n)$ and the transition cost along a path $g(n)$.

Definition 3.4.6. The *cumulative expansion cost* along the path from n_1 to n, including the expansion of n itself, is given by

$$g_\ell(n) := \sum_{n' \in \mathrm{anc}_+(n)} \ell(n') \tag{3.9}$$

In its general form, PHS evaluates nodes using

$$\varphi_{\mathrm{PHS}}(n) = \eta(n)\frac{g_\ell(n)}{\pi(n)}, \tag{3.10}$$

where $g_\ell(n)/\pi(n)$ plays the same role as in the Levin cost with $g_\ell(n)$ generalizing to non-unit expansion losses, and $\eta(n) \geq 1$ uses heuristic information to rescale this quantity toward the search effort required to reach a descendant solution. As with LTS, nodes with $\pi(n) = 0$ are assigned infinite PHS cost whenever $g_\ell(n) > 0$. In the unit expansion-loss setting, this means that every zero-probability node has infinite PHS cost. When $\ell(n) \equiv 1$ and $\eta(n) \equiv 1$ for all nodes n, this reduces to the standard LTS form

$$\frac{d(n)+1}{\pi(n)}.$$

This differs slightly from the root-separated adaptation used in Theorem 3.4.4, where the root expansion is accounted for outside the ratio.

Algorithm 7 PHS

1: **Input:** Root node n_1, budget B, goal set $\mathcal{N}^g$
2: **Output:** Solution path, or failure
3: **return** `BestFirstSearch`$(n_1, \varphi_{\mathrm{PHS}}, B, \mathcal{N}^g)$

This yields a best-first policy-guided search procedure depicted in Algorithm 7, with a bound on the cumulative expansion loss before reaching a goal. In the unit-loss case $\ell(n) \equiv 1$, this bound becomes a bound on the number of node expansions, recovering the usual Levin-style interpretation. While the LTS cost function φ_{LTS} is monotone, φ_{PHS} need not be, due to the heuristic factor η. Following Orseau and Lelis [79], define the monotone non-decreasing *envelope*

$$\varphi^+(n) := \max_{n' \in \mathrm{anc}_+(n)} \varphi_{\mathrm{PHS}}(n'). \tag{3.11}$$

To write this in the same multiplicative form, define

$$\eta^+(n) := \varphi^+(n) \frac{\pi(n)}{g_\ell(n)} \geq \eta(n), \tag{3.12}$$

with $\eta^+(n) = \eta(n)$ when $g_\ell(n) = 0$, such that

$$\varphi^+(n) = \eta^+(n) \frac{g_\ell(n)}{\pi(n)}.$$

To provide a bound on the search, define $\mathcal{N}_\varphi(n) = \{n' \in \mathrm{desc}_+(n_1) \ : \ \varphi^+(n') \leq \varphi^+(n)\}$ as the set of nodes of value at most $\varphi^+(n)$, and $\mathcal{L}_\varphi(n) = \{n' \in \mathcal{N}_\varphi(n) \ : \ \mathcal{C}(n') \cap \mathcal{N}_\varphi(n) = \emptyset\}$ as the set of leaf nodes in $\mathcal{N}_\varphi(n)$. The definition of $\eta^+(n)$ is used only for finite-cost nodes. Nodes with $\varphi^+(n) = +\infty$ do not belong to $\mathcal{N}_\varphi(n^*)$ when n^* has finite cost, and therefore do not affect the finite bound below.

Theorem 3.4.7 (Orseau and Lelis [79], Theorem 1, adapted)**.** *Let ℓ be a non-negative expansion-loss function and let $\eta(\cdot) \geq 1$ be any heuristic factor. Let $n^* \in \arg\min_{n \in \mathcal{N}^g} \varphi^+(n)$ be a finite-cost goal node. Then the cumulative expansion loss incurred by PHS before returning a solution is bounded by*

$$L(\mathrm{PHS}, n^*) \leq \frac{g_\ell(n^*)}{\pi(n^*)} \eta^+(n^*) \sum_{n' \in \mathcal{L}_\varphi(n^*)} \frac{\pi(n')}{\eta^+(n')}. \tag{3.13}$$

The proof is given in Appendix A.4. While the PHS cost function φ_{PHS} generalizes the one used by LTS, one may wonder if the LTS bound can be recovered from the PHS bound. And the answer is yes!

Corollary 3.4.8 (Orseau and Lelis [79], Corollary 2, adapted)**.** *From Theorem 3.4.7, if* $\eta(n) \equiv 1$*, then*

$$L(\mathrm{PHS}, n^*) \leq \frac{g_\ell(n^*)}{\pi(n^*)}. \tag{3.14}$$

The proof is given in Appendix A.4. In the unit-cost case $\ell(n) \equiv 1$, this becomes

$$L(\mathrm{PHS}, n^*) \leq \frac{d(n^*) + 1}{\pi(n^*)}. \tag{3.15}$$

This recovers the standard LTS bound. By contrast, Theorem 3.4.4 uses a root-separated adaptation and obtains the slightly tighter expression $1 + d(n^*)/\pi(n^*)$. Since $\pi(n^*) \leq 1$, the root-separated bound from Theorem 3.4.4 satisfies

$$1 + \frac{d(n^*)}{\pi(n^*)} \leq \frac{d(n^*) + 1}{\pi(n^*)},$$

so the two expressions are consistent, with Theorem 3.4.4 giving a slightly tighter bound under the notation adopted earlier in the chapter. Both accounting conventions appear in the policy-tree-search literature: this thesis uses the root-separated convention for its LTS analysis, while PHS uses root-inclusive expansion loss.

Practical Heuristics

Many problems in single-agent deterministic search already have readily available heuristic functions. For the present search-effort analysis, let $h(n)$ estimate the remaining expansion loss from n to a solution, analogous to how an A* heuristic estimates remaining transition cost. The heuristic factor can then be defined as

$$\eta_h(n) = \frac{g_\ell(n) + h(n)}{g_\ell(n)}, \tag{3.16}$$

which results in the evaluation function

$$\varphi_h(n) = \eta_h(n)\,\frac{g_\ell(n)}{\pi(n)} = \frac{g_\ell(n) + h(n)}{g_\ell(n)}\,\frac{g_\ell(n)}{\pi(n)} = \frac{g_\ell(n) + h(n)}{\pi(n)}. \tag{3.17}$$

The heuristic factor $\eta_h(n)$ extends the LTS cost $g_\ell(n)/\pi(n)$ by incorporating an estimate of the remaining expansion loss. However, this construction inherits only the additive part of the future solution cost. In particular, if n^* is a least-expansion-loss descendant solution of n and h is exact, then $g_\ell(n) + h(n) = g_\ell(n^*)$, and therefore $\varphi_h(n) = g_\ell(n^*)/\pi(n)$. But the LTS cost associated with the descendant solution itself is $g_\ell(n^*)/\pi(n^*)$. Thus, even with an exact heuristic for the remaining cost, $\varphi_h(n)$ still misses the ratio

$$\frac{\pi(n)}{\pi(n^*)} = \frac{1}{\pi(n^* \mid n)},$$

which accounts for the additional decay in probability along the suffix from n to n^*. This distinction matters because, in policy tree search, search effort depends not only on how much path cost remains, but also on how much probability mass must be preserved along the remaining trajectory. When $\pi(n^*) \ll \pi(n)$, the quantity $\varphi_h(n)$ can substantially underestimate the Levin-style search effort associated with actually reaching the descendant solution.

Accurately Predicting Remaining Search Effort

The motivation for PHS* is precisely to compensate for this missing probability term by extrapolating the future policy decay from the prefix observed so far. The key idea is to estimate not only the remaining additive loss, but also the additional decay in policy probability along the suffix from n to a descendant solution n^*. Suppose $h(n)$ estimates the remaining loss from n to n^*. Then the desired solution search effort is

$$\frac{g_\ell(n^*)}{\pi(n^*)} \approx \frac{g_\ell(n) + h(n)}{\pi(n^*)}.$$

The difficulty is therefore to estimate $\pi(n^*)$. The following extrapolation is defined for nodes with $\pi(n) > 0$ and $g_\ell(n) > 0$. Nodes with $\pi(n) = 0$ are assigned infinite PHS* cost. Define the geometric extrapolation statistic

$$p(n) := \pi(n)^{1/g_\ell(n)}.$$

This statistic approximates the geometric mean policy probability per unit of expansion loss along the prefix. Under the root-inclusive convention, the

Algorithm 8 PHS*

1: **Input:** Root node n_1, budget B, goal set $\mathcal{N}^g$
2: **Output:** Solution path, or failure
3: **return** `BestFirstSearch`$(n_1, \varphi_{\text{PHS}^*}, B, \mathcal{N}^g)$

approximation includes the root expansion even though it has no corresponding policy factor. The following approximation assumes that the log policy probability per unit expansion loss remains approximately stationary along the remaining suffix:

$$\pi(n^*) \approx p(n)^{g_\ell(n)+h(n)} = \left(\pi(n)^{1/g_\ell(n)}\right)^{g_\ell(n)+h(n)} = \pi(n)^{1+h(n)/g_\ell(n)}.$$

Substituting this estimate into the LTS search effort yields

$$\varphi_{\text{PHS}^*}(n) := \frac{g_\ell(n) + h(n)}{\pi(n)^{1+h(n)/g_\ell(n)}} = \eta_{\text{PHS}^*}(n)\frac{g_\ell(n)}{\pi(n)}, \tag{3.18}$$

where the corresponding heuristic factor is

$$\eta_{\text{PHS}^*}(n) := \frac{1 + h(n)/g_\ell(n)}{\pi(n)^{h(n)/g_\ell(n)}}. \tag{3.19}$$

This produces a more aggressive PHS* evaluation function. In contrast to $\varphi_h(n) = (g_\ell(n) + h(n))/\pi(n)$, which only accounts for the additive increase in loss, $\varphi_{\text{PHS}^*}(n)$ also attempts to account for the multiplicative decrease in policy mass expected along the remaining path to a solution.

The resulting algorithm is shown in Algorithm 8, and can be viewed as using the same best-first search PHS framework but with a different choice of heuristic factor. Importantly, the bound in Theorem 3.4.7 continues to apply to PHS* as the result does not depend on any particular form for η, as it only requires $\eta(n) \geq 1$. Since η_{PHS^*} is simply another such choice of heuristic factor, the same general upper bound holds when PHS is instantiated with η_{PHS^*}.

State Pruning The state pruning checks in `BestFirstSearch` (Algorithm 1) need to be modified for PHS to ensure nodes are expanded in best-first order and their lowest cost first.

Theorem 3.4.9 (Orseau and Lelis [79], Appendix G, adapted)**.** *Assume the search problem is deterministic, the policy is state-based, and both the expansion-loss function $\ell(\cdot)$ and heuristic factor $\eta(\cdot)$ are state-based: for all nodes $u, v \in \mathcal{N}$, if u and v represent the same state s, then $\ell(u) = \ell(v)$ and $\eta(u) = \eta(v)$. Let $m, n \in \mathcal{N}$ be two nodes that represent the same state s. Suppose that*

$$\varphi_{\text{PHS}}(m) \leq \varphi_{\text{PHS}}(n) \qquad \text{and} \qquad \pi(m) \geq \pi(n).$$

Then n is safely pruned for PHS: for every descendant $n' \in \text{desc}(n)$, there exists a corresponding descendant $m' \in \text{desc}(m)$ such that m' and n' represent the same state s' and $\varphi_{\text{PHS}}(m') \leq \varphi_{\text{PHS}}(n')$.

The proof is in Appendix A.4.2, along with Algorithm 16 which provides the *lazy* state pruning check which is done when nodes are removed from OPEN. The overhead for state pruning is a map where the keys are the previously expanded states and the values are retained $(\varphi_{\text{PHS}}, \pi)$-pairs. The pruning condition above applies to PHS instantiations whose heuristic factor is state-based. It should not be applied automatically to PHS*, since η_{PHS^*} depends on path-dependent quantities such as $g_\ell(n)$ and $\pi(n)$.

The given pruning rule allows the search to *prune* redundant nodes while keeping the PHS guarantees. In practice, however, it is common to prune nodes during expansion if a node representing the same state has been previously expanded. While the theoretical guarantees no longer hold, the implementation is simpler and has a smaller computation cost per expansion, and is what is used throughout the thesis.

Numerical Stability The path probabilities $\pi(n)$ may decrease exponentially with depth, which can lead to numerical underflow. Since applying a monotone transformation preserves the expansion order induced by the evaluation function, it is often preferable to compute the PHS* cost in log-space:

$$\ln \varphi_{\text{PHS}^*}(n) = \ln(g_\ell(n) + h(n)) - \left(1 + \frac{h(n)}{g_\ell(n)}\right) \ln \pi(n).$$

Moreover, if n' is a child of n, then the log-path probability can be updated incrementally as $\ln \pi(n') = \ln \pi(n) + \ln \pi(n' \mid n)$. The log-space expression is

used for finite-cost nodes with $\pi(n) > 0$; zero-probability nodes are assigned infinite cost.

3.4.3 Rerooting Levin Tree Search

The progression from LTS to PHS and PHS* shows one way that additional information can be incorporated into policy tree search. In these methods, the extra information is attached to the current frontier node and is used to estimate how difficult it will be to complete the path to a solution. PHS*, in particular, strengthens this idea by trying to account not only for the remaining additive loss, but also for the future decay in policy mass along the suffix to a solution.

Rerooting Levin Tree Search ($\sqrt{\text{LTS}}$, read as "root-LTS") [77] considers a different possibility. Sometimes the most informative signal available during search is not an estimate of the remaining distance to a goal, but the appearance of an intermediate node that marks meaningful progress. Such a node may indicate that a bottleneck in the state-space has been crossed, that an important subtask has been solved, or that the search has entered a promising region of the tree. This role is played by *clue nodes*, which we use informally to make explicit the information that guides the search rather than as a separate formally defined node type. Rather than estimating future effort directly, clues identify places from which it may be advantageous to redirect search.

Example 3.4.10. The intuition behind $\sqrt{\text{LTS}}$ and clue nodes is well illustrated in the 1,000 clue node example from Orseau et al. [77], depicted in Figure 3.2. Consider a binary tree of depth 100 with a single solution leaf. In the absence of any additional information, solving such a problem may require search through the full depth of the tree, which may, in the worst case, require searching through the 2^{100} leaves. Now suppose that at depth 50 there are 1,000 clue nodes, and the solution lies below exactly one of them. These clues are not visible in advance, but are revealed only when the corresponding nodes are reached. Even though 999 of the 1,000 clues are misleading, the existence of clue nodes fundamentally changes the structure of the search problem. A

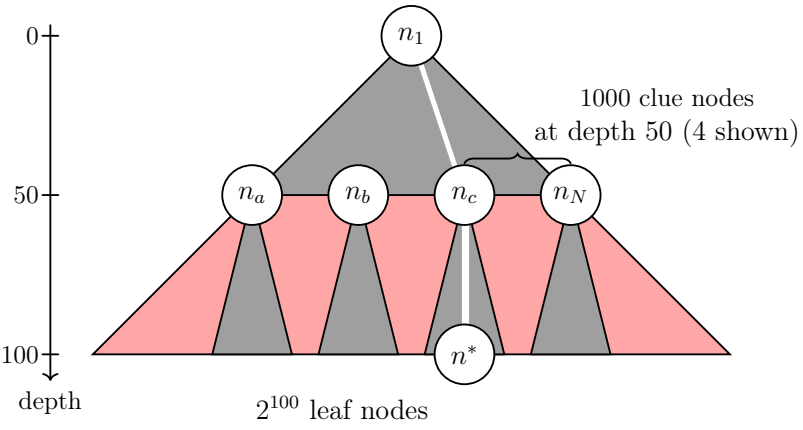


Figure 3.2: A binary tree of depth 100, with 1000 clue nodes at depth 50.

clue-aware search algorithm can simply search from the root until it finds all clue nodes at depth 50, then start a search under each clue node until it finds a solution. This simple clue-aware algorithm would require approximately $1{,}001 \times 2^{51}$ node expansions, which yields exponential savings.

A natural question to ask is whether this information is special in any way such that a new algorithm needs to be designed to take advantage of it, or whether using LTS is enough here. As it turns out, LTS struggles to make any substantial gains using clue nodes. Continuing from Example 3.4.10, under a clue-aware policy one may assign $\pi(n' \mid n) = 1$ when a generated child is recognized as either a clue node or the goal node, and otherwise split the policy uniformly. The solution node n^* would then have $\pi(n^*) = 2^{-98}$, and the LTS cost would be $\frac{d}{\pi}(n^*) = 100 \times 2^{98}$. Every node at depth 98 that does not descend from a clue node has a cost of $\frac{d}{\pi}(n) = 98 \times 2^{98} < \frac{d}{\pi}(n^*)$, and nodes n at depth 98 descending from clue nodes have cost $\frac{d}{\pi}(n) = 98 \times 2^{97}$. Thus, LTS using clue nodes in this example would need to expand up to and including depth 98, resulting in at least 2^{99} nodes.

This example is useful because it separates the value of clues from the mechanism used to exploit them. Even a simple clue-aware strategy can benefit by first locating all clue nodes and then searching beneath them, which is already dramatically better than ignoring the clues altogether. At the same time, the example also shows why the problem is nontrivial. Since most clues

are misleading, it is not enough to react greedily to the first clue encountered, nor is it enough to assume that every clue should be treated equally. What is needed is a principled way to share search effort among multiple candidate rerooting points while retaining search-effort guarantees. The next step is therefore to introduce rooted Levin-style cost functions and a way of composing them into a single best-first search procedure through rerooting weights. This is the key idea behind $\sqrt{\text{LTS}}$.

Self-Counting Cost Functions

To obtain a search-effort bound for rerooting methods, we need a cost function whose value is more than a priority score. Ideally, the numerical value assigned to a node should already control how many nodes can have cost no larger than it. This motivates the notion of a self-counting cost function.

Definition 3.4.11 (Orseau et al. [77])**.** A cost function $c : \mathcal{N} \to \mathbb{R}_{\geq 0} \cup \{+\infty\}$ is *self-counting* if, for all $\theta \geq 0$, the number of nodes of cost at most θ is itself at most θ:

$$|\{n \in \mathcal{N} : c(n) \leq \theta\}| \leq \theta. \tag{3.20}$$

In other words, the cost value already controls the size of its own low-cost region. By itself, however, this is only a counting statement. To turn it into a statement about the progress of best-first search, monotonicity is required.

Remark 3.4.12. For a monotone self-counting cost function c, `BestFirstSearch` enumerates nodes in order of increasing costs. For all t, we have

$$t = |\{n_1, \ldots, n_t\}| \leq |\{n : c(n) \leq c(n_t)\}| \leq c(n_t). \tag{3.21}$$

When the cost is monotone, `BestFirstSearch` visits nodes in non-decreasing cost order, and the self-counting property then implies that the node reached at step t must have cost at least t. Thus, self-counting provides the counting argument, monotonicity provides the ordering argument, and together they yield a direct link between cost values and search effort.

To make this concrete, we previously showed in the proof of Theorem 3.4.4 for LTS that $|\{n : \frac{d}{\pi}(n) \leq \theta\}| \leq 1 + \theta$. `BestFirstSearch` with cost function

$\frac{d}{\pi}(\cdot)$ is equivalent to `BestFirstSearch` with $1+\frac{d}{\pi}(\cdot)$, and since $|\{n : 1+\frac{d}{\pi}(n) \leq 1+\theta\}| \leq 1+\theta$, we have that $1+\frac{d}{\pi}(\cdot)$ is a self-counting cost function, which matches the LTS bound given in Theorem 3.4.4.

Weighted Composition

The previous subsection introduced self-counting cost functions as a way to relate cost values directly to search effort. The rerooting setting now raises a new question. If several candidate rerooting points are available, how should search effort be shared among them within a single best-first procedure? This is the role of rerooting weights.

This interpretation is most naturally understood through a computation-sharing analogy. Suppose N algorithms are run on the same CPU, with algorithm i receiving a fraction w_i of total computation. If that algorithm would solve the problem in τ_i steps on its own, then under the shared schedule it requires roughly T total steps, where

$$\lfloor w_i T \rfloor = \tau_i \qquad \Leftrightarrow \qquad T \leq \frac{\tau_i + 1}{w_i}.$$

Since this holds for every i, a solution is found after

$$T \leq \min_i \frac{\tau_i + 1}{w_i}.$$

The weight therefore acts as a budget share, and dividing by the weight accounts for the slowdown caused by sharing computation. The same idea applies to search costs. If $c_i(n)$ denotes the cost of reaching node n under the i-th component search, and that component receives a fraction w_i of the total search effort, then its effective cost is inflated to approximately $c_i(n)/w_i$. A natural combined cost is therefore

$$c(n) = \min_i \frac{c_i(n)}{w_i}.$$

The minimum selects the rooted search which is *quickest* to reach n after accounting for that search's allotted share of computation.

Rooted Levin Costs

To instantiate the composition idea, we next need a cost function that measures search effort relative to a candidate rerooting point rather than only relative to the original root. Suppose that, after visiting an ancestor n_k, we were to start a fresh LTS search that treats n_k as its local root and ignores everything outside the subtree below n_k. The Levin-style search effort required to reach a descendant n from that local root is then

$$\tfrac{d}{\pi}(n_k \prec n) := \frac{d(n) - d(n_k)}{\pi(n \mid n_k)}, \tag{3.22}$$

with $\tfrac{d}{\pi}(n_k \prec n) = \infty$ if $n_k \not\prec n$. This is simply the usual LTS cost applied to the suffix from n_k to n rather than to the full path from the original root. If $\pi(n \mid n_k) = 0$, then $\tfrac{d}{\pi}(n_k \prec n)$ is defined to be $+\infty$. Thus, a rooted cost is finite only when the suffix from n_k to n is policy-reachable.

This rooted quantity captures the local subproblem induced by n_k. Once the search has already paid the price to reach n_k, the remaining question is how difficult it is to continue from there. The rerooted Levin cost is meant to answer exactly that question. It measures the effort of solving the problem below a candidate local root, rather than the effort of reaching the same node from the start of the search.

Slenderness Cost Function

While the rooted LTS cost given in Equation 3.22 is certainly a valid option to use for a cost and bound on a rooted local search, using this cost function can lead to loose bounds on the search effort required. This is most evident in the following example.

Example 3.4.13. Consider the search tree in Figure 3.3, which depicts a long chain from the root to a node n_a, that has two leaves n_b and n_c with $\pi(n_b \mid n_a) = \pi(n_c \mid n_a) = 1/2$. Using the self-counting cost function $1 + \tfrac{d}{\pi}$, the number of nodes with cost at most the cost of n_a is $1 + d(n_a)$, which matches exactly the number of nodes from n_1 to n_a. However, when we consider the two children of n_a, they both have costs $1 + \tfrac{d}{\pi}(n_b) = 1 + \tfrac{d}{\pi}(n_c) = 2d(n_a) + 3$.

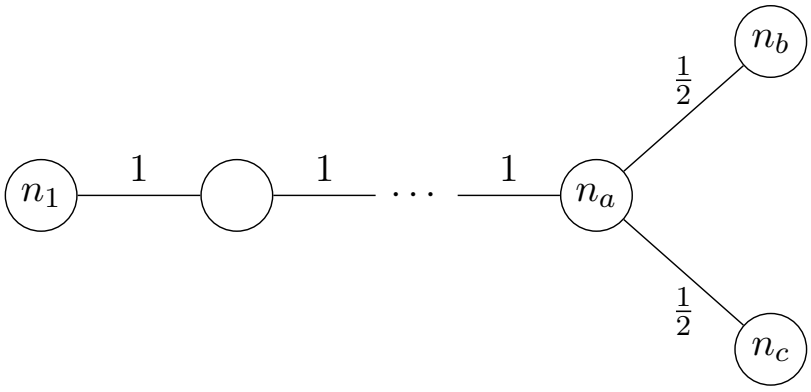


Figure 3.3: A chain with two leaf nodes at the end.

While this is indeed an upper bound on the number of nodes of cost at most n_b, the actual count is $d(n_a)+3$, which results from double-counting the ancestors shared by n_b and n_c.

A tighter alternative to the LTS cost function is what Orseau et al. [77] refer to as the slenderness cost function.

Definition 3.4.14 (Slenderness, Orseau et al. [77])**.** The function $\lambda : \mathcal{N} \to [1, \infty)$ is the *slenderness* of a node, where $\lambda(n_1) = 1$ and for all nodes n,

$$\lambda(n') = \lambda(n)\pi(n' \mid n) + 1, \qquad \forall n' \in \mathcal{C}(n). \tag{3.23}$$

The intuition behind the slenderness of a node and how it helps with the double counting problem is the following: if a node n holds a weighted share $\lambda(n)$ of its ancestors (including itself), then a child n' holds a share $\lambda(n)\pi(n' \mid n)$ of the ancestors of n, and thus a weighted share $\lambda(n)\pi(n' \mid n) + 1$ of the ancestors of n'.

It also follows that for every node n, $1 \leq \lambda(n) \leq 1 + d(n)$. An important quantity used throughout the previous proofs on search bounds has been to sum a value over leaf nodes in the tree. In Theorem 3.4.4, it was the sum of depths $d(n)$ of the leaf nodes. In Theorem 3.4.7, it was the sum of cumulative expansion costs along the path from root to the leaf nodes $g_\ell(n)$. Summing the slenderness $\lambda(n)$ values of the leaf nodes should provide a tighter bound on the number of nodes than summing the depth as done previously.

Definition 3.4.15 (Slenderness cost function, Orseau et al. [77])**.** The *slenderness cost function* $\frac{\lambda}{\pi} : \mathcal{N} \to \mathbb{R}_{\geq 1} \cup \{+\infty\}$ is defined as

$$\frac{\lambda}{\pi}(n) = \frac{\lambda(n)}{\pi(n)} = \sum_{n' \preceq n} \frac{1}{\pi(n')}, \tag{3.24}$$

which is a monotone cost function. If $\pi(n) = 0$, then $\frac{\lambda}{\pi}(n) = +\infty$. Equivalently, if any prefix $n' \preceq n$ has $\pi(n') = 0$, then the corresponding term in the sum is assigned value $+\infty$.

Definition 3.4.16 (Rooted slenderness cost function, Orseau et al. [77])**.** The *rooted slenderness cost function* $\frac{\lambda}{\pi} : \mathcal{N} \times \mathcal{N} \to \mathbb{R}_{\geq 1} \cup \{+\infty\}$ is defined as

$$\frac{\lambda}{\pi}(n \mid n') = \sum_{n' \preceq n_i \preceq n} \frac{1}{\pi(n_i \mid n')}, \tag{3.25}$$

and $\frac{\lambda}{\pi}(n \mid n') = \infty$ if $n' \not\preceq n$. If any denominator $\pi(n_i \mid n')$ appearing in the sum is zero, the corresponding term is assigned value $+\infty$, and hence the rooted slenderness cost is infinite. For a fixed root n', $\frac{\lambda}{\pi}(\cdot \mid n')$ is self-counting and monotone along descendant paths within the subtree rooted at n'. However, the root-excluded component used by $\sqrt{\text{LTS}}$, and the resulting weighted minimum over candidate roots, should not be treated as a globally monotone best-first cost [77].

Orseau et al. [77] then use the following as the rooted cost function, taking place of $\frac{d}{\pi}(n_k \prec n)$:

$$c_t^r(n) = \begin{cases} \frac{\lambda}{\pi}(n \mid n_t) - 1 & \text{if } n_t \prec n, \\ \infty & \text{otherwise.} \end{cases} \tag{3.26}$$

The proofs and analysis that follow in this thesis which use $\frac{d}{\pi}(n_k \prec n)$ for the rooted-cost function have analogous proofs using $c_t^r(n)$. For details related to $c_t^r(n)$, see Orseau et al. [77]. This chapter retains $\frac{d}{\pi}$ for the rerooting analysis and examples; Chapter 7 explicitly uses the slenderness-cost implementation.

The $\sqrt{\text{LTS}}$ Algorithm

$\sqrt{\text{LTS}}$ can now be understood as combining all these local LTS searches into a single best-first search procedure, given in Algorithm 9. Every visited node

Algorithm 9 $\sqrt{\text{LTS}}$

1: **Input:** Root node n_1, budget B, goal set $\mathcal{N}^g$
2: **Output:** Solution path, or failure
3: **return** `BestFirstSearch`$(n_1, \varphi_{\sqrt{\text{LTS}}}, B, \mathcal{N}^g)$

acts as a candidate local root. The rerooter assigns a weight to each such root, and the search scores a node n by the cheapest weighted local search through which n can be reached:

$$\varphi_{\sqrt{\text{LTS}}}(n) = \begin{cases} 0, & n = n_1, \\ \min\limits_{n_t \prec n} \dfrac{1}{w_t} \frac{d}{\pi}(n_t \prec n), & n \neq n_1 \end{cases} \tag{3.27}$$

where $w_t \geq 0$ is the rerooting weight for the LTS search anchored at the ancestor n_t of n. The value assigned to the root is only a convention ensuring that the initial node is expanded first. The rooted costs are meaningful only for non-root nodes, since they measure the cost of reaching n from a strict ancestor used as a local root. The analysis below therefore counts the nodes below the root using $\varphi_{\sqrt{\text{LTS}}}$, and then accounts for the root expansion by the leading 1 appearing in the search-effort bounds. Ancestors with $w_t = 0$ are ignored in the minimum, equivalently their weighted rooted cost is treated as $+\infty$. If all such ancestors have zero weight, then $\varphi_{\sqrt{\text{LTS}}}(n) = +\infty$. In practice, we set the root rerooting weight $w_1 = 1$.

Thus, a node is preferred when it can be reached cheaply from some promising rerooting point after accounting for the share of computation allocated to that rooted search. In this sense, rerooting does not merely rescore frontier nodes. It reallocates search effort across multiple candidate subtasks discovered during the search itself.

Having defined the rerooted evaluation function, we can now ask what search-effort guarantee it provides. The answer is naturally expressed in terms of a subtask decomposition of the path to the target node. Rather than charging the entire root-to-target trajectory to a single Levin-style search, the analysis allows the path to be partitioned across intermediate rerooting points. The cost of each segment is then measured locally from its chosen root and scaled by the share of computation assigned to that rooted search.

Definition 3.4.17. A *subtask decomposition* of the path from n_1 to n^* is an ordered set $D = \{n_{T_1}, n_{T_2}, \ldots, n_{T_m}\} \subseteq \mathrm{anc}_+(n^*)$ such that $n_1 = n_{T_1} \prec n_{T_2} \prec \cdots \prec n_{T_m} = n^*$. The nodes in D are viewed as subtask boundaries.

For example, in the game of Sokoban, one subtask decomposition is all the ancestors where the agent has pushed a box on a goal spot. Orseau et al. [77] also provide theoretical guarantees on the number of node expansions required until the first solution node has been found, which depend on the quality of the policy and the rerooter. Suppose $\sqrt{\text{LTS}}$ finds the solution node n^* at some step T.

Theorem 3.4.18 ($\sqrt{\text{LTS}}$ subtask bound, Orseau et al. [77], Corollary 12, adapted)**.** *Let $\mathcal{D}(n^*)$ be the set of all such subtask decompositions of n^*. For $D \in \mathcal{D}(n^*)$, the selected ancestors of n^* are expanded at steps $T_1, T_2, \ldots, T_{|D|}$, where necessarily $T_1 = 1$ and $T_{|D|} = T$. Then the number of search steps T that $\sqrt{\text{LTS}}$ takes before visiting n^* is bounded by*

$$T \leq 1 + \min_{D \in \mathcal{D}(n^*)} \max_{i<|D|} \frac{w_{<T}}{w_{T_i}} \, \tfrac{d}{\pi}(n_{T_i} \prec n_{T_{i+1}}) \,, \tag{3.28}$$

where $w_{<T} = \sum_{j<T} w_j$ is the cumulative rerooting weight of the nodes expanded up to step T (excluded). The bound is finite for decompositions in which each selected segment $n_{T_i} \prec n_{T_{i+1}}$ has positive suffix probability $\pi(n_{T_{i+1}} \mid n_{T_i}) > 0$ and positive rerooting weight $w_{T_i} > 0$.

The proof is in Appendix A.5.2. The way to interpret the $\sqrt{\text{LTS}}$ subtask bound is that $w_{T_i}/w_{<T}$ is the time share allocated to the LTS instance started at the ith (shallowest) ancestor of n^* in the subtask decomposition, $\frac{d}{\pi}(n_{T_i} \prec n_{T_{i+1}})$ is the bound on the number of search steps this LTS instance takes to reach the next subtask boundary $n_{T_{i+1}}$, and $\max_{i<|D|}$ corresponds to the most "difficult" subtask in the decomposition. This subtask decomposition can allow $\sqrt{\text{LTS}}$ to expand exponentially fewer nodes than LTS. Because $w_{<T}$ depends on the unknown stopping step T, this is an implicit a posteriori guarantee rather than a directly computable a priori bound.

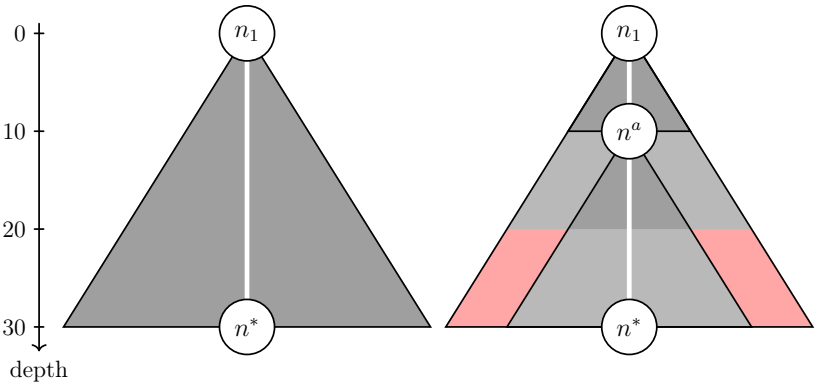


Figure 3.4: Difference in search effort between LTS and $\sqrt{\text{LTS}}$. On the left, LTS searches the entire tree in grey. On the right, $\sqrt{\text{LTS}}$ utilizes a subtask decomposition and only searches the nodes within the grey region, skipping the nodes in the red region.

Example 3.4.19. Consider a perfect binary tree with a solution node n^* at depth 30, shown in Figure 3.4. Under a uniform policy, the LTS bound (Equation 3.8) tells us that LTS would take at most $30 \times 2^{30} + 1$ node visits to find n^*. Let $n^a \prec n^*$ such that $d(n^a) = 10$. $\sqrt{\text{LTS}}$ visits n^a at step T_a and n^* at step T^*. For this oracle rerooter, set $w_1 = w_{T_a}$, and 0 everywhere else. Equation 3.28 tells us that $\sqrt{\text{LTS}}$ is competitive with the best subtask decomposition. Therefore, it is also competitive with any subtask decomposition that is meaningful for analysis. Let us compare it with the subtask decomposition $n_1 \prec n^a = n_{T_a} \prec n^*$, which happens to be the best one, in this case. The node n^* is visited at T^* with

$$\begin{aligned} T^* - 1 &\leq w_{<T^*} \max\left\{ \frac{\frac{d}{\pi}(n_1 \prec n^a)}{w_1}, \frac{\frac{d}{\pi}(n^a \prec n^*)}{w_{T_a}} \right\} \\ &= 2 \max\left\{ \frac{10 \times 2^{10}}{1}, \frac{20 \times 2^{20}}{1} \right\} = 40 \times 2^{20} \,. \end{aligned}$$

This is an exponential improvement over the leading 30×2^{30} term in the LTS bound. The corresponding rooted slenderness-cost variant gives a bound of $2^{22} - 4$, which is approximately a factor of ten tighter than the $\frac{d}{\pi}$ bound above. This illustrates why slenderness is a useful refinement and why Chapter 7 uses it in the implementation, while this chapter retains $\frac{d}{\pi}$ for analytic clarity.

State Pruning The state pruning checks in `BestFirstSearch` (Algorithm 1) need to be modified for $\sqrt{\text{LTS}}$ to ensure nodes are expanded in best-first order and their lowest cost first.

Definition 3.4.20 (Safely prunable node)**.** A node n_t visited at step t is safely prunable if there exists a node n_j with $j < t$ such that $s(n_t) = s(n_j)$ and for each non-empty action sequence y such that $n_{\underset{\sim}{t}}$ (resp. $n_{\underset{\sim}{j}}$) is the descendant of n_t (resp. n_j) following y, with $j < \underset{\sim}{j}$ and $t < \underset{\sim}{t}$, then $\varphi_{\sqrt{\text{LTS}}}(n_{\underset{\sim}{j}}) \leq \varphi_{\sqrt{\text{LTS}}}(n_{\underset{\sim}{t}})$.

Theorem 3.4.21. *Assume the search problem is deterministic, the policy is state-based, and the rerooting weights are state-based: for all nodes $n_i, n_j \in \mathcal{N}$, if $s(n_i) = s(n_j)$, then $w_i = w_j$. Let n_t be a node visited at step t. Suppose that there exists a node n_j with $j < t$ such that $s(n_j) = s(n_t)$ and that the following holds: for every ancestor $n_{\widetilde{t}} \prec n_t$ with $\widetilde{t} < t$, there exists an ancestor $n_{\widetilde{j}} \prec n_j$ with $\widetilde{j} < j$ satisfying*

$$d(n_j) - d(n_{\widetilde{j}}) \leq d(n_t) - d(n_{\widetilde{t}})\,,$$
$$\textit{and} \qquad w_{\widetilde{j}}\,\pi(n_j \mid n_{\widetilde{j}}) \geq w_{\widetilde{t}}\,\pi(n_t \mid n_{\widetilde{t}}).$$

Then, n_t is safely prunable for $\sqrt{\text{LTS}}$. That is, for every descendant $n_t \prec n_{\underset{\sim}{t}}$, if $n_{\underset{\sim}{j}}$ is the descendant of n_j obtained by following the same action suffix y from n_j as $n_{\underset{\sim}{t}}$ follows from n_t, then $n_{\underset{\sim}{j}}$ and $n_{\underset{\sim}{t}}$ represent the same state, and

$$\varphi_{\sqrt{\text{LTS}}}(n_{\underset{\sim}{j}}) \leq \varphi_{\sqrt{\text{LTS}}}(n_{\underset{\sim}{t}}).$$

The proof is in Appendix A.5.3, along with Algorithm 17 which provides the *lazy* state pruning check which is done when nodes are removed from OPEN. Unlike LTS and PHS, safe duplicate pruning for $\sqrt{\text{LTS}}$ is not generally characterized by a single-node dominance test. The reason is that $\varphi_{\sqrt{\text{LTS}}}(n)$ depends on the set of rerooting points available along the ancestor chain of n. Accordingly, a node can be pruned only when its entire ancestor chain is covered, in the sense of Definition 3.4.20, by the ancestor chains of the retained equal-state nodes. As a result, the overhead for state pruning can be quite costly.

The given pruning rule allows the search to *prune* redundant nodes while keeping the $\sqrt{\text{LTS}}$ guarantees. We note that this is one possible pruning rule; there may be cases for additional pruning opportunities. In practice, however, it is common to prune nodes during expansion if a node representing the same state has been previously expanded. While the theoretical guarantees no longer hold, the implementation is simpler and has a smaller computation cost per expansion, and is what is used throughout the thesis.

Numerical Stability The path probabilities $\pi(n)$ may decrease exponentially with depth, which can lead to numerical underflow. Since applying a monotone transformation preserves the expansion order induced by the evaluation function, for non-root nodes it is often preferable to compute the $\sqrt{\text{LTS}}$ cost in log-space:

$$\ln \varphi_{\sqrt{\text{LTS}}}(n) = \min_{n_k \prec n} \ln(d(n) - d(n_k)) - \ln w_k + \ln \pi(n_k) - \ln \pi(n).$$

Moreover, if n' is a child of n, then the log-path probability can be updated incrementally as $\ln \pi(n') = \ln \pi(n) + \ln \pi(n' \mid n)$. The log-space expression is used only for finite weighted rooted costs, where the suffix probability and the corresponding rerooting weight are both positive. Components with zero suffix probability or zero rerooting weight are ignored in the minimum, equivalently treated as having infinite component cost.

Robustness

From the subtask decomposition bound given in Equation 3.28, for subtask i rooted at n_{T_i}, at global step T the *share* of the total compute going towards that subtask is approximately $w_{T_i}/w_{<T}$. Finishing subtask i requires approximately $\frac{d}{\pi}(n_{T_i} \prec n_{T_{i+1}})$ local steps, but because total compute is shared, the total number of steps required to finish subtask i is approximately

$$\frac{w_{<T}}{w_{T_i}} \frac{d}{\pi}(n_{T_i} \prec n_{T_{i+1}}).$$

If $w_{<T}$ grows linearly with T, the time share for the ith subtask keeps shrinking as T grows. The consequence of this is that the longer a subtask takes, the worse the bound becomes for that same subtask.

Definition 3.4.22 (Orseau et al. [77])**.** If the rerooter produces weights $\tilde{w}$, then define the *robust* rerooting weights w as

$$w_t = \frac{\tilde{w}_t}{\tilde{w}_{\leq t}}. \tag{3.29}$$

This construction assumes $\tilde{w}_t \geq 0$ and $\tilde{w}_{\leq t} > 0$ for every t, as ensured for example by a positive root weight.

One can then show a related subtask decomposition bound similar to Theorem 3.4.18.

Corollary 3.4.23 (Robust $\sqrt{\text{LTS}}$ guarantee, Orseau et al. [77], Corollary 17, adapted)**.** *For every subtask decomposition, where the selected ancestors of n^* are expanded at steps $T_1, T_2, \ldots, T_{|D|}$, where necessarily $T_1 = 1$ and $T_{|D|} = T$, the number of search steps T that $\sqrt{\text{LTS}}$ takes to visit n^* when using robust rerooting weights is bounded by*

$$T \leq 1 + \left[1 + \ln\left(\frac{\tilde{w}_{<T}}{\tilde{w}_1}\right)\right] \max_{i<m} \frac{\tilde{w}_{\leq T_i}}{\tilde{w}_{T_i}} \tfrac{d}{\pi}(n_{T_i} \prec n_{T_{i+1}}). \tag{3.30}$$

The statement assumes the relevant reparameterized weights in the denominators are positive. If $\tilde{w}_{T_i} = 0$ for a selected subtask root, that decomposition gives an infinite contribution to the bound.

The proof is in Appendix A.5.2. The subtask decomposition bound using robust weights prevents the linear blowup in two subtle but important ways. First, the global normalizer $w_{<T}$ becomes logarithmic. Second, the remaining linear factor for subtask i depends on the cumulative weight *before* that subtask, $\tilde{w}_{\leq T_i}$, not on everything that happens afterwards. Finally, in the worst case $\sqrt{\text{LTS}}$ is only a log-factor away from LTS when using robustness. Take $m = 2$, $n_1 = n_{T_1} \prec n_{T_2} = n_T$. Then, if $T_1 = 1$ and $\tilde{w}_1 = 1$,

$$\frac{\tilde{w}_{\leq T_i}}{\tilde{w}_{T_i}} = \frac{\tilde{w}_{\leq 1}}{\tilde{w}_1} = 1.$$

So the bound becomes

$$T \leq 1 + \left[1 + \ln\left(\frac{\tilde{w}_{<T}}{\tilde{w}_1}\right)\right] \tfrac{d}{\pi}(n_1 \prec n_T).$$

For additional details on robustness, including additional reparameterizations, see [77].

A Connection Between Best-First Evaluation Functions and $\sqrt{\text{LTS}}$

As a small observation, there is a neat connection between general best-first evaluation functions and the rerooting framework. $\sqrt{\text{LTS}}$ requires only a policy and a rerooting weight function, and these need not be specified independently. Given any node-evaluation function with $0 < \varphi(n') < \infty$ for every child and finite branching, one may instead induce both quantities directly from φ by defining

$$Z_\varphi(n) = \sum_{n' \in \mathcal{C}(n)} \frac{1}{\varphi(n')}, \qquad \pi_\varphi(n' \mid n) = \frac{\varphi(n')^{-1}}{Z_\varphi(n)}, \qquad w_\varphi(n) = Z_\varphi(n).$$

This choice assigns greater policy mass to children preferred by φ, while using the same normalizing quantity as the rerooting weight at n. The resulting construction has a simple local interpretation: for every child $n' \in \mathcal{C}(n)$,

$$\frac{1}{w_\varphi(n)} \frac{d(n') - d(n)}{\pi_\varphi(n' \mid n)} = \varphi(n').$$

Hence the parent-based rooted Levin term reproduces exactly the one-step ordering induced by φ. In this way, any positive best-first evaluation function may be viewed as inducing a particular Levin-style rerooting rule. Since this construction does nothing more than specify a policy and rerooting weights, all of the previously established guarantees for the $\sqrt{\text{LTS}}$ algorithm continue to apply immediately under the substitution $\pi = \pi_\varphi$ and $w = w_\varphi$. The only caveat is that global equivalence need not hold in general, because the rerooted evaluation minimizes over all ancestor rooting points rather than only the parent, so the overall expansion order may differ from that of the original best-first search. If φ is path-dependent, then the induced policy is node-based rather than state-based, so state-based duplicate-pruning results do not automatically apply.

3.4.4 Learning the Policy: The Levin Loss

The search procedures considered in this chapter are guided by a policy, so an important question is how that policy should be learned. A natural objective is to train the policy so as to reduce the *search loss*, rather than merely to

imitate locally preferred actions. In this section, we show the derivation of the *Levin loss objective* [79].

Let π_θ denote a differentiable policy with parameters θ, and let S_θ denote a policy-guided search algorithm such as LTS, PHS*, or $\sqrt{\text{LTS}}$ instantiated with π_θ. For a collection of training tasks indexed by $k \in [K]$, the ideal objective is

$$\theta^* = \arg\min_\theta \sum_{k=1}^{K} \min_{n^* \in \mathcal{N}_k^g} L_k(S_\theta, n^*), \tag{3.31}$$

where $L_k(S_\theta, n^*)$ is the cumulative search loss incurred on task k before the algorithm expands a solution node n^*. This objective is appealing because it directly matches the quantity we ultimately wish to minimize: the amount of computation required to solve the training tasks.

In practice, however, this objective is not convenient to optimize directly. The search process is discrete, the solution node returned by search depends on the current policy, and even if one introduces a differentiable approximation to the search loss, its gradient is generally difficult to obtain. For this reason, Orseau and Lelis [79] propose optimizing a surrogate derived from the PHS* upper bound instead.

For Levin tree search, and more generally for PHS* when the heuristic-dependent terms are treated separately, the dominant quantity in the upper bound (Equation 3.13) is of the form

$$\frac{g_\ell(n^*)}{\pi_\theta(n^*)}.$$

A natural surrogate objective for a solution node n^* is

$$\mathcal{L}_{\text{Levin}}(\theta; n^*) = \frac{g_\ell(n^*)}{\pi_\theta(n^*)}.$$

When $\ell(n) \equiv 1$ and $\eta(n) \equiv 1$, we have

$$g_\ell(n^*) = d(n^*) + 1,$$

so this becomes

$$\mathcal{L}_{\text{Levin}}(\theta; n^*) = \frac{d(n^*) + 1}{\pi_\theta(n^*)}.$$

Thus, in the unit-cost case, the surrogate recovers the standard Levin-style quantity in which the root expansion is counted inside the numerator. This differs slightly from the root-separated convention sometimes used for Levin tree search, where the root expansion is accounted for outside the ratio and the priority is written in terms of $d(n)/\pi(n)$. The distinction is purely a matter of accounting convention and does not change the basic learning principle.

Following the derivation of Orseau and Lelis [79], suppose that for task k the search loss is approximated as

$$L_k(S_\theta, n_k^*) \approx c_k \frac{g_\ell(n_k^*)}{\pi_\theta(n_k^*)}, \tag{3.32}$$

where c_k is an unknown factor treated as approximately independent of θ. Intuitively, this isolates the leading inverse-probability term in the upper bound and absorbs the remaining factors into a multiplicative constant. Thus, the Levin loss is a search-motivated surrogate: it retains the dominant inverse path-probability term while ignoring the policy dependence of the remaining search dynamics, including the returned solution, competing paths, and any rerooting or pruning behaviour. Using the identity

$$\nabla_\theta f(\theta) = f(\theta)\, \nabla_\theta \log f(\theta),$$

we obtain

$$\begin{aligned} \nabla_\theta \widetilde{L}_k &= \widetilde{L}_k \nabla_\theta \log \widetilde{L}_k \\ &\approx \widetilde{L}_k \nabla_\theta \log \left(c_k \frac{g_\ell(n_k^*)}{\pi_\theta(n_k^*)} \right). \end{aligned}$$

Here, $\widetilde{L}_k$ denotes a differentiable surrogate for the true loss L_k, where we use Equation 3.32. Since both c_k and $g_\ell(n_k^*)$ are treated as constants with respect to θ, this simplifies to

$$\nabla_\theta \widetilde{L}_k \approx \widetilde{L}_k\, \nabla_\theta \log \frac{1}{\pi_\theta(n_k^*)}.$$

Replacing $\widetilde{L}_k$ by the observed search loss gives the practical gradient estimate

$$\nabla_\theta \widetilde{L}_k \approx L_k(S_\theta, n_k^*)\, \nabla_\theta(-\log \pi_\theta(n_k^*)).$$

Let $n_1 = n_1, n_2, \ldots, n_{d(n^*)+1} = n^*$ denote the nodes on the path from the root to n^*, indexed in path order, so that $\text{parent}(n_i) = n_{i-1}$ for each $i \geq 2$. Because the path probability factors along the solution trajectory,

$$\pi_\theta(n^*) = \prod_{i=2}^{d(n^*)+1} \pi_\theta(n_i \mid n_{i-1}) \,,$$

it follows that

$$-\log \pi_\theta(n^*) = \sum_{i=2}^{d(n^*)+1} -\log \pi_\theta(n_i \mid n_{i-1}) \,.$$

Hence, for a solved training instance, the policy objective can be written as

$$\mathcal{J}^{(k)}_{\text{policy}}(\theta) = \widehat{L}_k \sum_{i=2}^{d(n_k^*)+1} -\log \pi_\theta(n_i \mid n_{i-1}) \,,$$

where

$$\widehat{L}_k := L_k(S_\theta, n_k^*)$$

is the observed search loss, which is treated as a constant (stop-gradient) during the policy update. In this form, the update is a weighted negative log-likelihood over the successful root-to-goal trajectory.

This objective can be viewed as a search-aware variant of the usual cross-entropy loss. The logarithmic term is the same negative log-probability that appears in standard policy classification or behaviour cloning, but here it is applied to an entire successful trajectory and weighted by the search effort required to discover that trajectory. As a result, trajectories that were expensive to find induce larger updates than those found almost immediately. This distinction is important because a naïve cross-entropy objective would treat all actions appearing on successful trajectories as equally informative, regardless of how difficult the corresponding trajectory was for the search to uncover. In the present setting, however, the aim is not simply to match action frequencies, but to reduce future search effort. A solution trajectory found with little search is already comparatively easy for the current policy-guided search procedure to recover, so increasing its probability may have only limited effect on subsequent search. By contrast, a trajectory found only after many node expansions is a potentially valuable target for learning, although high search

effort can also arise from competing branches, heuristic guidance, rerooting, duplicate structure, or tie-breaking. Weighting by observed search loss directs larger updates toward trajectories that required substantial search effort, encouraging the policy to shift probability mass toward solution paths whose increased probability may reduce future search effort.

The implementations discussed in this thesis do not apply additional variance-control mechanisms specific to the Levin loss, such as loss clipping, normalization, logarithmic transformation, or a baseline.

Chapter 4

Bayes DistNet: A Robust Neural Network for Algorithm Runtime Distribution Predictions

This chapter is joint work with Michael Buro. It was previously published at the AAAI Conference on Artificial Intelligence [102].

Within this thesis, this chapter provides the distributional prediction component used in Chapter 5 for instance-specific restart control.

4.1 Introduction

Many of the algorithmic solvers for NP-complete problems, such as CSP and SAT, rely on backtracking methods. These solvers can have runtimes that vary substantially, depending on whether they make mistakes caused by sub-optimal heuristics during the recursive backtrack calls. Adding randomization and restarts to the backtrack algorithms has been shown to help alleviate some of the heavy-tail nature that these runtimes exhibit, decreasing the algorithm's runtime by many orders of magnitude in some cases [39, 43].

If the underlying *runtime distribution* (RTD) is known for a particular algorithm, then an optimal fixed cutoff-time restart strategy can be formulated [64]. Knowing the underlying RTD can also lead to efficient use of algorithm portfolios, which can dynamically assign algorithms to input instances based on the predictive RTD. These are just a few reasons why it is important to have models which are robust and can give accurate RTD predictions for unseen

instances.

The current state-of-the-art RTD prediction models are designed to give exact parametric distributions as output, with the model predicting the particular distribution parameters [27]. However, these models make the assumption that the particular algorithms on the given instances follow a particular parametric distribution. These models also tend to have their predictive power hindered in the presence of a small number of observations per input instance or with censored examples (only a lower bound for the runtime is known). The goal of this chapter is to develop a robust model under these conditions and to lift some of the restrictions imposed by existing models.

The remainder of this chapter first reviews randomized algorithm runtime prediction and Bayesian deep learning. It then describes the Bayesian extension and evaluates it in several scenarios. The following chapter applies this predictive model to instance-specific restart control for policy tree search.

4.2 Background and Related Work

In this section, we give an overview of randomized algorithm runtime prediction and Bayesian learning in neural networks.

4.2.1 Randomized Algorithm Runtime Prediction

Problems such as CSP and SAT are NP-complete, meaning that there is currently no guarantee there is a polynomial time solution technique. The dominant solver technique for these types of problems is backtrack search, in which heuristics pick an unbound variable, assign a value to it, and the search proceeds recursively. If an inconsistency is detected, the algorithm backtracks and tries another assignment to that variable.

A consequence of using backtrack-based solvers is that early mistakes can cause long searches down branches of the search tree which eventually need to be backtracked. This leads to the so-called "heavy-tail" phenomenon, in which the runtime of these algorithms exhibits tails which are not exponentially bounded. Harvey [43] was the first to show that adding randomization

and restarts to the backtracking algorithm can alleviate the issue of wasting time on eventual dead-ends. Gomes et al. [39] presented general methods for introducing controlled randomization, and showed that speedups of several orders of magnitude could be achieved for state-of-the-art algorithms.

Introducing randomness into algorithms means that a given algorithm run on the same input problem instance has a runtime which follows some initially unknown distribution. There are many reasons why one would want the ability to predict such runtime distributions (RTDs). Luby et al. [64] showed that if one knows the underlying RTD, then it is possible to construct a fixed cutoff-time restart strategy that is optimal among all possible universal strategies, up to a constant factor. Other popular CSP and SAT solvers use a portfolio of various algorithms, such as SATzilla [108] and ArgoSmArT [73]. Knowing the RTDs of each algorithm in the portfolio allows such solvers to choose the *best* algorithm for a given problem instance.

The majority of the early work for predicting algorithm runtimes involved predicting mean instance runtimes, given the instance's features. Many of these methods were based on regression variants, such as ridge regression used in SATzilla [108].

Gagliolo and Schmidhuber [32] investigated using a neural network to predict the time remaining before an algorithm reaches the solution on a given problem instance, in the context of algorithm portfolio time allocation. Previous methods for predicting RTDs had separate models for each distribution parameter. DistNet [27] established a new state-of-the-art RTD prediction model which jointly learns the parameters of the RTD by using a neural network. Each output node of the neural network corresponds to a parameter for the given distribution. The neural network's loss function directly minimizes the negative log-likelihood (NLLH) of the distribution parameters, given the observed runtimes.

4.2.2 Bayesian Neural Networks

Traditional neural networks can be viewed as a probabilistic model: given a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$, the parameterized model $P(\mathbf{y} \mid \mathbf{x}, \mathbf{w})$ takes in a

d-dimensional input $\mathbf{x} \in \mathbb{R}^d$ and computes a point estimate for each output $\mathbf{y} \in \mathcal{Y}$. Training most commonly involves finding the set of weights $\mathbf{w}$ which maximizes the likelihood of the data.

These networks can be prone to overfitting, especially when the number of model parameters is sufficiently greater than the number of training samples. Regularization techniques like using weight priors and dropout [96] have been proposed to mitigate overfitting.

Bayesian neural networks (BNNs) [65, 70] extend traditional neural networks by replacing the network's deterministic weights with a prior distribution over these weights, and using posterior inference. The full posterior distribution of the model's parameters $\mathbf{w}$ is used when making predictions of unseen data. Prediction for this type of probabilistic model involves taking the expectation over the optimized posterior distribution $P(\mathbf{w} \mid \mathcal{D})$. The *posterior predictive distribution* of unseen data $(\hat{\mathbf{x}}, \hat{\mathbf{y}})$ is given by

$$\begin{aligned} P(\hat{\mathbf{y}} \mid \hat{\mathbf{x}}) &= \mathbb{E}_{P(\mathbf{w}|\mathcal{D})}\left[P(\hat{\mathbf{y}} \mid \hat{\mathbf{x}}, \mathbf{w})\right] \\ &= \int P(\hat{\mathbf{y}} \mid \hat{\mathbf{x}}, \mathbf{w}) P(\mathbf{w} \mid \mathcal{D})\, d\mathbf{w}. \end{aligned}$$

Bayesian neural networks can help with overfitting as the model implicitly involves using an ensemble of an infinite number of neural networks by averaging over all possible weight values, while adding a constant multiple number of network parameters, determined by how many parameters the parametric distribution has. Bayesian neural networks primarily represent *epistemic* uncertainty through uncertainty over weights. *Aleatoric* uncertainty must be represented through the likelihood or predictive distribution used for the observations. Aleatoric uncertainty captures uncertainties which are inherent to running statistical trials, like the outcome of a dice toss. This type of uncertainty *cannot* be reduced, no matter how much data is collected. Epistemic uncertainty captures the uncertainty in the model being used, which is due to limited data and/or knowledge. This type of uncertainty *can* be reduced with more data.

Both inference and prediction for BNNs require access to the posterior distribution over weights,

$$P(\mathbf{w} \mid \mathcal{D}) = \frac{P(\mathcal{D} \mid \mathbf{w})P(\mathbf{w})}{P(\mathcal{D})} = \frac{P(\mathcal{D} \mid \mathbf{w})P(\mathbf{w})}{\int P(\mathcal{D} \mid \mathbf{w}')P(\mathbf{w}')\, d\mathbf{w}'}.$$

Although the prior $P(\mathbf{w})$ can be chosen, the normalization term $P(\mathcal{D})$ requires integration over the full weight space. For neural networks, this integral is generally computationally intractable, making exact posterior inference impractical. Sampling-based methods such as Markov Chain Monte Carlo (MCMC) methods [36] can be used to approximate this posterior, but they typically scale poorly as the number of parameters and data points grows, which limits their practicality for modern neural networks [10].

A common alternative, suggested by Hinton and Van Camp [45] and Graves [40], is to use a variational approximation for the posterior $P(\mathbf{w} \mid \mathcal{D})$. *Variational methods* construct a new distribution $q(\mathbf{w} \mid \boldsymbol{\theta})$, parameterized by $\boldsymbol{\theta}$, that approximates the true posterior $P(\mathbf{w} \mid \mathcal{D})$ by minimizing the Kullback-Leibler (KL) divergence between the two:

$$\begin{aligned}
\boldsymbol{\theta}_{\text{VI}} &= \arg\min_{\boldsymbol{\theta}} KL\left[q(\mathbf{w} \mid \boldsymbol{\theta}) || P(\mathbf{w} \mid \mathcal{D})\right] \\
&= \arg\min_{\boldsymbol{\theta}} \int q(\mathbf{w} \mid \boldsymbol{\theta}) \log \frac{q(\mathbf{w} \mid \boldsymbol{\theta})}{P(\mathbf{w})P(\mathcal{D} \mid \mathbf{w})}\, d\mathbf{w} \\
&= \arg\min_{\boldsymbol{\theta}} KL\left[q(\mathbf{w} \mid \boldsymbol{\theta}) || P(\mathbf{w})\right] - \mathbb{E}_{q(\mathbf{w}|\boldsymbol{\theta})}\left[\log P(\mathcal{D} \mid \mathbf{w})\right].
\end{aligned}$$

We denote $\boldsymbol{\theta}_{\text{VI}}$ as the parameters found from variational inference methods. Bayes by Backprop [12] uses the above to form the cost function

$$J_{\text{BBB}}(\mathcal{D}, \boldsymbol{\theta}) = KL\left[q(\mathbf{w} \mid \boldsymbol{\theta}) \;||\; P(\mathbf{w})\right] - \mathbb{E}_{q(\mathbf{w}|\boldsymbol{\theta})}\left[\log P(\mathcal{D} \mid \mathbf{w})\right]. \tag{4.1}$$

The intuition for the above cost function is that there is a tradeoff between having a variational posterior that is *close* to the prior we choose yet also able to explain the likelihood of the data.

However, while this avoids exact computation of the true posterior, the objective is still not available in closed form for general neural networks, since it contains an expectation over a high-dimensional distribution of weights. Moreover, because the weights are sampled from $q(\mathbf{w} \mid \boldsymbol{\theta})$, the loss involves

stochastic nodes through which ordinary backpropagation cannot be applied naïvely. Bayes by Backprop [12] addresses this by using Monte Carlo sampling together with the reparameterization trick [52, 75] to obtain a differentiable stochastic approximation of the objective. This results in the approximation to Equation 4.1 as

$$J_{\text{BBB}}(\mathcal{D}, \boldsymbol{\theta}) \approx \frac{1}{M} \sum_{r=1}^{M} \left[\log q(\mathbf{w}^{(r)} \mid \boldsymbol{\theta}) - \log P(\mathbf{w}^{(r)}) - \log P(\mathcal{D} \mid \mathbf{w}^{(r)}) \right], \quad (4.2)$$

where M is the number of Monte Carlo samples and $w^{(r)}$ is the set of weights drawn from the r-th Monte Carlo sample of the variational posterior $q(\mathbf{w} \mid \boldsymbol{\theta})$. Having a computationally efficient way of training BNNs has allowed extensions to other network architectures like RNNs [31] and CNNs [92], as well as studies into utilizing the uncertainty measures the BNNs provide [4, 49].

4.3 Problem Formulation

We follow the same problem statement introduced by Eggensperger et al. [27]: A randomized algorithm A is run on a set of n problem instances $\Xi_{\text{train}} = \{\xi_1, \ldots, \xi_n\}$. Each instance $\xi \in \Xi_{\text{train}}$ has m instance features $\mathbf{f}(\xi) = [f(\xi)_1, \ldots, f(\xi)_m]$. Since algorithm A is randomized, k runtime observations $\mathbf{t}(\xi) = [t(\xi)_1, \ldots, t(\xi)_k]$ are gathered by executing A on problem instance ξ, with k different random number generator seeds. The goal is to learn a model that can predict the RTD for unseen instance ξ_{n+1}, given features $\mathbf{f}(\xi_{n+1})$.

Because it is common during the data generation stage to terminate the algorithm on *hard* problem instances which take a long time to solve, the exact runtime may not be known. Ideally, we would like to make use of the measured runtime lower bound without discarding the data and wasting time. From this motivation, we also consider cases where there is censoring, i.e., stopping the algorithm execution if it exceeds a cutoff time t_c. We define the level of censoring as the fraction of runtime observations which are censored due to exceeding a cutoff time t_c. If there are N runtimes in total, with a predetermined level of censoring of c, then t_c is defined as the u-th fastest

runtime where $u = \lfloor N(1-c) \rfloor$ for $0 \leq c < 1$. The adjusted runtimes $t'(\xi)_i$ are thus set as $\min(t(\xi)_i, t_c)$ for all i.

4.4 A Bayesian Approach to Predicting RTDs

Previous methods for predicting RTDs would fit RTD parameters to the set of runtimes for each training instance, then measure loss in the space of RTD parameters $\boldsymbol{\beta}$. Eggensperger et al. [27] introduced DistNet, a new state-of-the-art algorithm runtime prediction model, which is the first of its kind that jointly learns all RTD parameters (as opposed to having independent models for each RTD parameter) by directly minimizing the NLLH loss function.

In this section, we extend DistNet with variational weight sampling to obtain a more robust model in both low sample settings and for handling censored observations.

4.4.1 DistNet - A State-of-the-Art Algorithm RTD Prediction Model

Before we introduce our extensions, we give an overview of the DistNet model [27]. DistNet is a simple feed-forward network for a given distribution $\mathcal{F}$ with distribution parameter vector $\boldsymbol{\beta}$. There is one input neuron for each instance feature, and one output neuron for each distribution parameter β_i. The novelty of DistNet is that it directly minimizes the NLLH of the chosen distribution $\mathcal{F}$. The loss function used in DistNet is

$$J_{\text{DN}}(\mathbf{w}) = -\sum_{\xi \in \Xi_{\text{train}}} \sum_{i=1}^{k} \log \mathcal{L}_{\mathcal{F}}(\hat{\boldsymbol{\beta}}_{\mathbf{w}, f(\xi)} \mid t(\xi)_i), \tag{4.3}$$

where $\mathcal{L}_{\mathcal{F}}$ is the likelihood function of the parametric distribution $\mathcal{F}$, and $\hat{\boldsymbol{\beta}}_{\mathbf{w}, f(\xi)}$ are the parameters of the distribution from the network output, with current weights $\mathbf{w}$.

4.4.2 Extending DistNet with a Bayesian Neural Network

To extend DistNet with a variational distribution over network weights, which we will refer to as *Bayes DistNet*, a few changes need to be made to the network. We start from the same base network, which has one input neuron for each input feature, and uses simple feed-forward layers.

One possible Bayesian formulation would have each sampled network output the parameters $\boldsymbol{\beta}$ of the chosen distribution $\mathcal{F}$, evaluate the corresponding observation likelihood, and form a Monte Carlo mixture for prediction. Instead, the implementation used here represents the predictive distribution through a stochastic ensemble of scalar runtime outputs.

The network directly outputs predicted runtimes $\hat{y}$. By sampling the network, a sequence of runtimes is produced which is assumed to follow $\mathcal{F}$. The parameters of $\mathcal{F}$ can then be approximated by computing their maximum likelihood estimates (MLEs), denoted $\hat{\boldsymbol{\beta}}^{\text{MLE}}$. The choice of $\mathcal{F}$ can ensure that we have analytical formulations of $\hat{\boldsymbol{\beta}}^{\text{MLE}}$, which are fast to compute using deep-learning framework intrinsics, and whose gradients can be tracked from the loss function.

The model uses a variational distribution $q(\mathbf{w} \mid \boldsymbol{\theta})$ over weights together with a prior $P(\mathbf{w})$. Following Blundell et al. [12], we assume that the variational posterior $q(\mathbf{w} \mid \boldsymbol{\theta})$ is a diagonal Gaussian distribution with mean $\boldsymbol{\mu}$ and standard deviation $\boldsymbol{\sigma}$. While it is possible to use a multivariate Gaussian, a diagonal Gaussian allows for simple computations without major numerical issues. To be able to use backpropagation, we use the reparameterization given by Blundell et al. [12], which obtains a sample of weights $\mathbf{w}$ for each layer by the following procedure: Sample the parameter-free noise $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, and let $\boldsymbol{\sigma} = \log(\mathbf{1} + \exp(\boldsymbol{\rho}))$. We can then denote $\boldsymbol{\theta} = (\boldsymbol{\mu}, \boldsymbol{\rho})$, and the parameterization of $\boldsymbol{\sigma}$ ensures that the values are always greater than 0. The weights $\mathbf{w}$ can then be sampled by $\mathbf{w} = \boldsymbol{\mu} + \boldsymbol{\sigma} \circ \boldsymbol{\epsilon}$, where $\circ$ is point-wise multiplication. Once a particular set of point-wise weights is sampled for each of the layers of the network, those weights are then used as in the traditional feed-forward

layers.

Since the network's task is to predict runtimes directly, it has a single output neuron. The network output $\hat{y}$ represents the predicted runtime for a particular set of point-wise weights sampled from each layer. To get a predictive distribution for a particular instance ξ, Monte Carlo sampling is used: the same instance feature vector $f(\xi)$ is evaluated under multiple weight samples from the variational posterior. Let M denote the number of Monte Carlo samples. For $r = 1, \ldots, M$, let $\boldsymbol{w}^{(r)} \sim q(\boldsymbol{w} \mid \boldsymbol{\theta})$ and let $\hat{y}_\xi^{(r)}$ denote the runtime predicted for instance ξ by evaluating the Bayesian network on $f(\xi)$ using the sampled weights $\boldsymbol{w}^{(r)}$. The predicted runtimes $\hat{y}_\xi^{(1)}, \ldots, \hat{y}_\xi^{(M)}$ are then treated as samples from the chosen distribution family $\mathcal{F}$, and the corresponding distribution parameters are estimated by maximum likelihood:

$$\hat{\boldsymbol{\beta}}_\xi^{\mathrm{MLE}} := \mathrm{MLE}_{\mathcal{F}}\left(\hat{y}_\xi^{(1)}, \ldots, \hat{y}_\xi^{(M)}\right). \tag{4.4}$$

Thus, as in DistNet, the distribution parameters remain instance-dependent. The difference is that Bayes DistNet obtains these parameters indirectly from Monte Carlo samples of predicted runtimes rather than directly from the network output.

Combining the Monte Carlo weight regularization term with the likelihood of the MLE-induced runtime distribution yields the following ensemble-derived training objective:

$$\begin{aligned} J_{\mathrm{BDN}}(\mathcal{D}, \boldsymbol{\theta}) = \frac{1}{M} \sum_{r=1}^{M} \left[\log q(\mathbf{w}^{(r)} \mid \boldsymbol{\theta}) - \log P(\mathbf{w}^{(r)})\right] \\ - \sum_{\xi \in \Xi_{\mathrm{train}}} \sum_{j=1}^{k} \log \mathcal{L}_{\mathcal{F}}\big(\hat{\boldsymbol{\beta}}_\xi^{\mathrm{MLE}} \mid t(\xi)_j\big), \end{aligned} \tag{4.5}$$

where $\hat{\boldsymbol{\beta}}_\xi^{\mathrm{MLE}}$ is computed separately for each training instance ξ using the M predicted runtimes obtained by evaluating the Bayesian network on $f(\xi)$. The dependence of $\hat{\boldsymbol{\beta}}_\xi^{\mathrm{MLE}}$ on the sampled weights $\boldsymbol{w}^{(1)}, \ldots, \boldsymbol{w}^{(M)}$ and variational parameters $\boldsymbol{\theta}$ is suppressed for notational convenience. Because the likelihood depends jointly on the full set of sampled weights through $\hat{\boldsymbol{\beta}}_\xi^{\mathrm{MLE}}$, this objective is not the standard Bayes-by-Backprop ELBO for a per-weight observation model. Here, the variational weight distribution is used to generate

the stochastic ensemble of scalar runtime predictors, while Equation 4.5 is the ensemble-derived objective used to train that representation.

4.4.3 Observing Censored Runtimes

In our application domain, censoring occurs when an algorithm needs to be stopped before completion, i.e., we only know a lower bound on the time it takes to actually complete the algorithm. There is a rich history in survival analysis on handling different types of censored data, and we recommend Klein and Moeschberger [53] for background reading. Gagliolo and Schmidhuber [33] first introduced handling *Type I censored sampling* in the context of algorithm runtime prediction, and we give the explanation for clarity.

If censoring time of t_c is used, then t_c is the maximum time an algorithm can run before being terminated. The dataset now consists of triples

$$\mathcal{D} = \{(\boldsymbol{f}(\xi), t'(\xi)_j, \delta_{\xi,j}) \ : \ \xi \in \Xi_{\text{train}}, \ j = 1, \ldots, k\},$$

where

$$t'(\xi)_j = \min(t(\xi)_j, t_c), \qquad \delta_{\xi,j} = \mathbb{I}\left[t(\xi)_j \leq t_c\right].$$

Thus, $\delta_{\xi,j} = 1$ indicates that the runtime observation was not censored, while $\delta_{\xi,j} = 0$ indicates that the observed value is only a lower bound.

Let $f_{\mathcal{F}}$ denote the density of distribution $\mathcal{F}$ with parameters $\boldsymbol{\beta}$, and let $\mathcal{S}_{\mathcal{F}}$ denote the corresponding survival function. If the runtime is not censored, then its contribution to the likelihood is the density evaluated at the observed runtime. If the runtime is censored, then all we know is that the true runtime exceeds the cutoff, so its contribution is the survival probability. Combining these two cases, the likelihood contribution of observation (ξ, j) is

$$\begin{aligned}\mathcal{L}_{\mathcal{F}} &\left(\hat{\boldsymbol{\beta}}_{\xi}^{\text{MLE}} \mid t'(\xi)_j, \delta_{\xi,j}\right) \\ &= f_{\mathcal{F}}\left(t'(\xi)_j \mid \hat{\boldsymbol{\beta}}_{\xi}^{\text{MLE}}\right)^{\delta_{\xi,j}} \mathcal{S}_{\mathcal{F}}\left(t'(\xi)_j \mid \hat{\boldsymbol{\beta}}_{\xi}^{\text{MLE}}\right)^{1-\delta_{\xi,j}}, \end{aligned} \tag{4.6}$$

which is then used to replace the likelihood term in the Bayesian cost function (Equation 4.5) from above.

4.4.4 Applying Bayes DistNet to RTD Prediction

For comparative results, we follow the same preprocessing procedure from Eggensperger et al. [27]. Input instance features are standardized to mean 0 and standard deviation 1, and observed runtimes are scaled into the range of $[0, 1]$. For a level of censoring c, where c is the fraction of censored runtimes from a total of N, the cutoff time t_c is the u-th fastest runtime, where $u = \lfloor N(1-c) \rfloor$. The runtimes are then set to $\min(t(\xi)_i, t_c)$, and the observations include whether censoring occurred or not.

Both the reference DistNet model and our new Bayesian variant share the majority of the network architecture proposed by Eggensperger et al. [27]. The input layer has a neuron for each instance feature. This is then followed by two hidden fully connected layers, each of which has 16 neurons. DistNet has a fully connected output layer with one neuron per parameter for the specific parametric distribution, whereas Bayes DistNet has a single output.

The Bayes DistNet architecture also has a prior $P(\mathbf{w})$ and a variational posterior $q(\mathbf{w} \mid \boldsymbol{\theta})$ that needs to be specified. As previously noted, we use a diagonal Gaussian variational posterior, with initial parameters $\boldsymbol{\theta}_{\text{init}} = (\boldsymbol{\mu}_{\text{init}}, \boldsymbol{\rho}_{\text{init}})$, $\boldsymbol{\mu}_{\text{init}} \sim \mathcal{N}(0, 0.1^2)$ and $\boldsymbol{\rho}_{\text{init}} \sim \mathcal{N}(-3, 0.1^2)$, where 0.1 denotes the standard deviation in both initializations. The chosen $\boldsymbol{\mu}_{\text{init}}$ ensures weights are initialized around 0 (just as we would for a standard neural network), and $\boldsymbol{\rho}_{\text{init}}$ is initialized to a small value, as we found learning is not as stable otherwise [92]. For the prior, we follow the proposal from Blundell et al. [12] of using a mixture of two Gaussian distributions

$$P(\mathbf{w}) = \prod_i \left[\alpha \mathcal{N}(\mathbf{w}_i; 0, \sigma_1^2) + (1-\alpha)\mathcal{N}(\mathbf{w}_i; 0, \sigma_2^2) \right],$$

where $\mathcal{N}(x; \mu, \sigma^2)$ is the Gaussian density evaluated at x with mean μ and variance σ^2, with $\alpha = 0.5$, $\sigma_1 = 0.3$, and $\sigma_2 = 0.01$. Changing these parameters did not make a noticeable difference so long as $\sigma_1 > \sigma_2$ and $\sigma_2 \ll 1$, as per Blundell et al. [12]. When performing inference and prediction, we use 16 Monte Carlo samples. As with previous studies [31, 92], a large number of samples gives marginally better results at the cost of computation time. We

did not find much improvement beyond 16 samples.

DistNet uses the *tanh* activation function, with the exception of the *exponential* activation function on the output layer to ensure the output distribution parameters are > 0. Bayes DistNet, on the other hand, uses *softplus* [38] on all layers as an architectural choice; the positive final activation ensures positive scalar runtime predictions. The positivity of variational posterior standard deviations is enforced separately by $\boldsymbol{\sigma} = \log(\mathbf{1} + \exp(\boldsymbol{\rho}))$. We then largely follow the hyperparameter choices from Eggensperger et al. [27], whose DistNet model we are basing our Bayes DistNet model from. Both network architectures use stochastic gradient descent (SGD), with batch normalization [46], L_2-regularization of $1e^{-4}$, a learning rate of $1e^{-3}$ which exponentially decays to $1e^{-5}$ over 500 expected epochs, and gradient clipping of $1e^{-2}$. Early stopping [85] is used on a separate validation set to reduce overfitting.

4.5 Experiments

The focus of our experiments is to see how our Bayesian model compares with the current state-of-the-art in settings with few observations, and with various levels of censoring. When gathering data to train the models, using fewer observations requires less time for the data gathering process. Moreover, being able to handle censored examples also means that one can still use instances which stop prematurely as training data, instead of throwing them away. There are also many domains in which much of the data is censored, such as survival data in medical trials, or nodes that remain in the open list when heuristic search methods are used.

For both experiments, we look at two different algorithms run on different problem instances, with the data gathered by Eggensperger et al. [27]:

- **Clasp-factoring**: The *Clasp* [35] CDCL solver running on SAT-encoded factorization problems.
- **LPG-Zenotravel**: The *LPG* [37] local search solver for planning graphs running on the *Zenotravel* planning domain [82].

Both DistNet and Bayes DistNet require a parametric distribution: DistNet directly outputs parameters for this distribution, and both use the distribution in the loss function. For both of the above scenarios, Eggensperger et al. [27] considered different parametric distributions to use for DistNet. By using the Kolmogorov-Smirnov (KS) goodness-of-fit test, they chose the top two distributions to use in their comparison. For both *Clasp-factoring* and *LPG-Zenotravel*, the inverse Gaussian and lognormal distributions were found to have the closest fit to the empirical distributions as compared to the other considered distributions. In our experiments, we also use these two distributions. The training process uses Python PyTorch 1.5 [80], and we make use of an open source Bayesian Layers PyTorch library [92]. All experiments were run on an Intel i7-7820X and Nvidia GTX 1080 Ti, with 64GB of memory running Ubuntu 16.04.

4.5.1 Low Sample Count per Instance

We evaluated the performance of our Bayes DistNet model by varying the number of observed runtimes per instance. Every instance starts with 100 observed runtimes, and a random subset of those 100 is selected. Having a smaller number of observed runtimes per instance means that the total training set is reduced. For testing, we keep all 100 observed runtimes for every instance in the test set. The evaluation is performed using a 10-fold cross-validation, with one run under each of 10 distinct seeds, and the results are averaged across folds.

Figure 4.1 reports several metrics to evaluate the goodness-of-fit of each model, compared to the empirical observed runtimes of the test set. The likelihood is a way to measure how probable a model is, given the observed data. A lower negative log likelihood (NLLH) thus indicates that one particular model gives the observed data a higher probability of occurring. For both scenarios of *Clasp-factoring* and *LPG-Zenotravel*, Bayes DistNet achieves a lower negative log likelihood across various levels of samples per instance. At the largest sample count tested, both DistNet and Bayes DistNet have similar likelihood scores. The full tabulated results corresponding to Figure 4.1 are

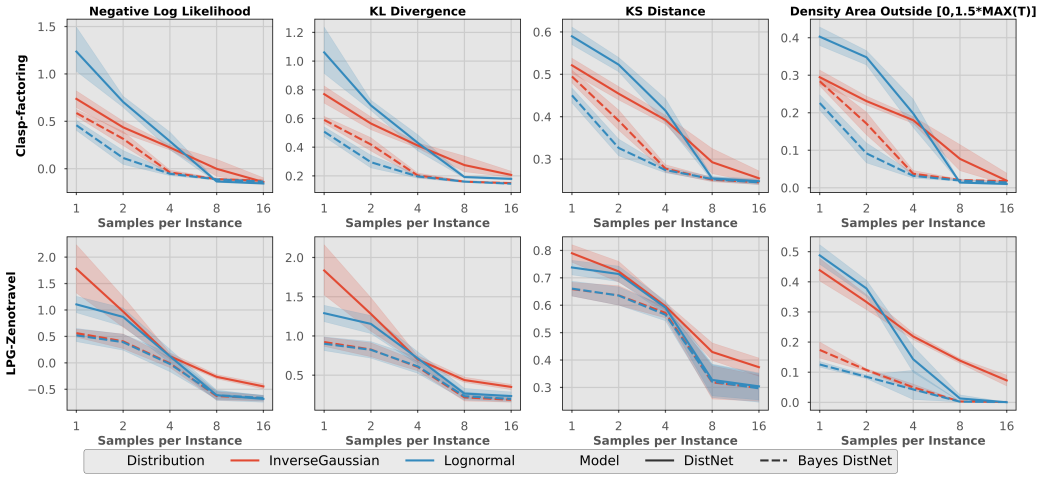


Figure 4.1: Comparison of DistNet and Bayes DistNet, for various numbers of observed runtimes per instance. Metrics are averaged across 10 cross-validation folds, each run with a distinct seed. From left to right: NLLH, KL-Divergence, KS-Distance, and percentage of the density area outside the diagnostic range of 0 to 1.5 times the maximum observed runtime for each instance.

given in Table C.1.

While the likelihood is a way to measure how probable a model is given the data, it does not indicate the shape of the predicted runtime distribution. We thus looked at two metrics to quantify how dissimilar the model's predictive distribution shape is compared to the empirical distribution, namely KL-Divergence and the Kolmogorov-Smirnov statistic (KS-Distance). For KL-Divergence, we estimate the empirical RTD using a Gaussian kernel density estimate with Scott's bandwidth rule[1], and report $D_{\mathrm{KL}}(P_{\mathrm{KDE}} \| P_{\mathrm{model}})$. The KS-Distance measures the maximal distance in height from the model's CDF to the empirical CDF. For both metrics, lower is better.

Our Bayesian model achieves lower KL-Divergence and lower KS-Distance with respect to the empirical distribution, as compared to the DistNet model across various levels of samples per instance. This indicates that, under these estimators, the Bayesian model produces predictive RTDs that are closer in shape to the empirical RTD. At the largest sample count tested, both models predict RTDs that are similarly close to the empirical RTDs on average.

[1] as implemented by `scipy.stats.gaussian_kde`

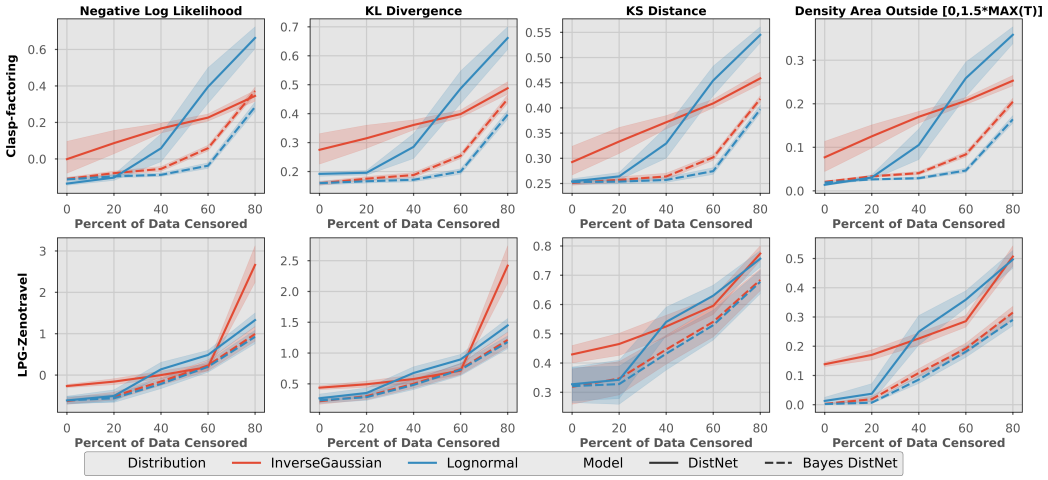


Figure 4.2: Comparison of DistNet and Bayes DistNet, for various levels of censoring, each with 8 observations per instance. All metrics are averaged across 10 cross-validation folds, each run with a distinct seed. From left to right: NLLH, KL-Divergence, KS-Distance, and percentage of density area outside the diagnostic range of 0 to 1.5 times the maximum observed runtime for each instance.

4.5.2 Handling Censored Observations

We also evaluated our Bayes DistNet model by keeping the number of samples per instance constant, but varying the percentage of censoring in the training data. We decided to use 8 samples per instance, as this was the number of samples at which we started to see both DistNet and our Bayesian variant perform similarly without censoring. To have an accurate comparison, we modified the DistNet loss function to include censored samples as well, following Equation 4.6. Figure 4.2 reports metrics similar to those used before to evaluate the goodness-of-fit for both DistNet and our Bayesian variant, as compared to the empirical RTDs. The full tabulated results corresponding to Figure 4.2 are given in Table C.2.

The NLLH of both the best DistNet model and Bayes DistNet is identical under no censoring. As the censoring percentage increases, both models are trained on less fully observable data, and we expect the NLLH to increase. Interestingly, we see that while both models have their NLLH increase, Bayes DistNet increases at a slower rate than DistNet. This indicates that Bayes DistNet has lower NLLH under the reported censoring protocol.

Using the distance metrics again, we can see how similar our Bayesian model is to the empirical RTD, as compared to DistNet. We expect both the KL-Divergence and the KS-Distance to increase as the percentage of censored data increases. Both the best DistNet model and our Bayesian model start with roughly similar distance metrics. As the percentage of censored data increases, our Bayesian model's distance metrics increase at a slower rate compared to DistNet. Under this evaluation, our Bayesian model produces distributions that are more similar to the empirical RTDs under censoring than DistNet.

4.5.3 Predictive Uncertainty Under Synthetic Feature Shifts

A known issue with standard neural networks, which use point estimates for weights, is that they tend to be overconfident in regions which had little or no data during training. Neural networks fit a function to training data which minimizes the chosen loss function. If deterministic weights are used, then a single particular function of many possible ones is chosen for the data. This function is then extrapolated to make predictions for inputs which may be *far* from the data used to train the network, resulting in overconfident predictions. As a result, predictions for inputs far from the training data can be overconfident.

Instead of using point estimates for weights, the variational weight ensemble in Bayes DistNet averages predictions across sampled weight values. This model averaging can, but need not, produce broader predictive intervals for inputs far from the training data.

Figure 4.3 illustrates the behaviour of DistNet and Bayes DistNet under a simple synthetic feature shift. Both networks were trained on the *Clasp-factoring* dataset, except for a held-out sample, and both models use the lognormal distribution in the loss function. The held-out sample then had all its features shifted by values in the range [-8,8] to produce a spectrum of shifted inputs.

The predictive distribution means and interquartile ranges were then plotted for these samples. DistNet has small interquartile ranges for these shifted

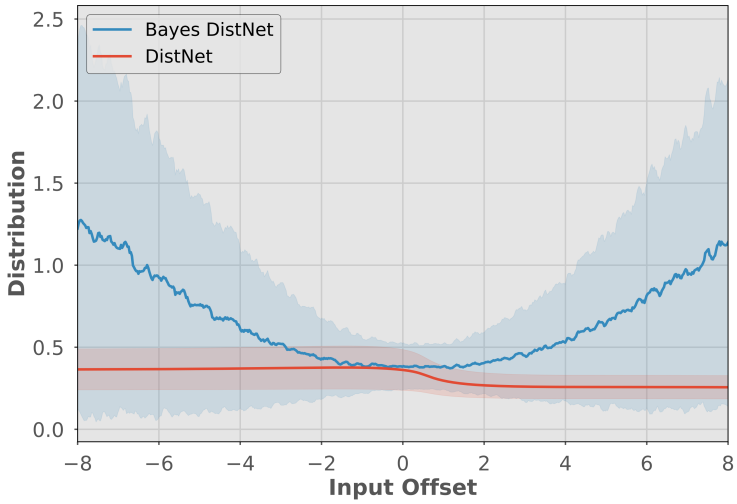


Figure 4.3: Predictive distributions under synthetic feature shifts for DistNet and Bayes DistNet. Both models are trained on the *Clasp-factoring* dataset, using the lognormal distribution loss function. Solid lines are mean predictions, with shaded regions showing interquartile ranges.

inputs, whereas the Bayes DistNet interquartile ranges increase over the displayed shifts. This is an empirical observation for this held-out instance and does not establish that variational weight uncertainty will increase under arbitrary distribution shifts. In sensitive or computationally expensive scenarios, a fallback measure based on predictive uncertainty would require separate calibration and validation.

4.5.4 Discussion

In the reported comparisons, Bayes DistNet gives higher likelihoods of the data and better matches the estimated empirical RTD shape in low-observation and censored settings.

Both DistNet and Bayes DistNet require a parametric distribution. Both use this distribution in the loss function for the likelihood, but DistNet directly outputs its parameters, whereas Bayes DistNet fits them by MLE to the scalar outputs of the weight-sampled ensemble. Thus, Bayes DistNet relaxes the requirement that each network output directly encode the parameters of $\mathcal{F}$, but it retains $\mathcal{F}$ as the distribution used to fit and score the resulting runtime representation. From Figures 4.1 and 4.2, we observe that the choice of para-

metric distribution has a greater impact on DistNet than on Bayes DistNet in the reported comparisons.

We finally give some insight into why Bayes DistNet can perform differently from DistNet even when the same parametric distribution is used. For low numbers of samples per instance and various levels of censoring, Figures 4.1 and 4.2 respectively show where the predicted RTD densities lie. We plot the percentage of total area under the RTDs outside the range $[0, T]$, where T is 1.5 times the maximum observed runtime for each instance. This out-of-range density is a diagnostic of extrapolation relative to the observed sample, not a calibration metric: genuinely heavy-tailed RTDs can assign substantial probability beyond the largest observed runtime. When a low number of samples is used, Figure 4.1 shows that DistNet assigns more density outside this diagnostic range in some cases. Figure 4.2 shows similar behaviour under censoring.

In summary, both DistNet and Bayes DistNet perform similarly at the largest non-censored sample count tested. As the level of censoring increases and as the number of samples per instance decreases, Bayes DistNet achieves better reported fit metrics than DistNet and more closely matches the KDE-estimated empirical RTDs. The experiments compare Bayes DistNet with DistNet; they do not constitute a comprehensive comparison with other uncertainty or survival-modelling approaches.

4.6 Conclusions and Future Work

This chapter has shown how the DistNet architecture can be extended with a variational distribution over weights and an ensemble-derived runtime objective. In the reported comparisons with DistNet, Bayes DistNet achieves better fit metrics in low-observation and censored settings. The model relaxes the requirement that network outputs directly encode an explicit parametric distribution, while retaining a parametric distribution for fitting and scoring the ensemble outputs.

Both DistNet and Bayes DistNet require a parametric distribution $\mathcal{F}$ to

quantify the likelihood in the loss function. As previously mentioned, we chose to have our Bayesian network output predicted runtimes directly instead of $\mathcal{F}$'s parameters. A further direction is to develop an efficient method for having the network output $\mathcal{F}$'s parameters and compare it with the present approach. Future work could develop extensions that do not depend on a restrictive class of parametric distributions. Finally, existing algorithm portfolio methods can be extended to utilize the features of our Bayesian network, such as having access to the uncertainty in predicting runtime distributions. Within this thesis, the immediate application is the use of predicted trajectory-length distributions to guide the restart shifts developed in Chapter 5.

Chapter 5

Beyond the Universal Strategy: Distribution-Guided Luby Tree Search

This chapter presents a novel extension to LubyTS, using the model introduced in Chapter 4. For background on LubyTS, see Section 3.3.2; for the predictive model used here, see Chapter 4.

5.1 Introduction

Chapter 3 introduced Luby Tree Search (LubyTS) as a policy-guided sampling method for the case in which an appropriate rollout depth is not known in advance. Rather than committing to a single depth bound, LubyTS follows a universal restart schedule, which guarantees that deeper rollouts are eventually attempted while remaining competitive with the best fixed cutoff in hindsight. This universality is attractive when little is known about the instance being solved, but it comes at a practical cost: a substantial fraction of the schedule is spent on very short rollouts. When solution trajectories are relatively deep, these short trials can consume a large amount of computation while having little chance of success.

A natural modification is to shift the Luby sequence schedule by a positive base amount, so that every rollout begins deeper in the tree. However, a single global shift is still instance-agnostic. The appropriate shift can vary substantially across problems. On some instances, a small shift is sufficient because

solution trajectories are comparatively short. On others, a much larger shift is preferable because repeated shallow restarts waste effort before the search ever reaches the relevant depth range. At the same time, overly aggressive shifts can also be harmful. In domains such as Sokoban, long rollouts may spend substantial effort inside deadlocked or otherwise unproductive regions of the search tree before restarting. The difficulty is therefore not whether to shift the Luby schedule, but how much to shift it for the particular instance at hand.

Chapter 4 introduced Bayes DistNet, a distributional prediction model that learns from censored observations and produces more reliable predictive distributions when data are limited or truncated. In this chapter, we use that model to predict, for each problem instance, a distribution over solution trajectory lengths under the policy-guided sampling process. These predicted distributions are then converted into instance-specific base shifts for LubyTS. This yields a distribution-guided version of LubyTS that retains the practical benefit of restart-based sampling while adapting its initial rollout depths to the structure of the current problem.

The chapter makes two main points. First, instance-sensitive shifts are more effective than a single global shift because they allow the restart schedule to adapt to heterogeneity across problems. Second, since Bayes DistNet can train from lower-bound observations, it remains informative even when many long trajectories are censored during data collection. This is especially important here, since difficult training problems are precisely the ones most likely to exceed a collection cutoff and therefore produce incomplete observations.

5.2 Method

5.2.1 Distribution-Guided Luby Tree Search

We now describe how Bayes DistNet is used to produce instance-specific shifts for LubyTS. A global shift can improve over the unshifted universal strategy when successful trajectories are typically deep, but it must apply the same

offset to every problem instance. In practice, however, the appropriate rollout depth can vary substantially from one problem to another. The proposed method therefore predicts, for each instance, a distribution over successful trajectory lengths and uses that distribution to determine how much the Luby schedule should be shifted for that particular problem.

Let ξ denote a problem instance, and let $\mathbf{o}(\xi)$ be its initial state observation. For a fixed policy π, let S_ξ denote the event that a collection run returns a solution, and define $T_\xi \in \mathbb{N}_{\geq 1}$ to be the length of its successful trajectory conditional on S_ξ. The quantity of interest is therefore the instance-dependent trajectory length distribution

$$F_\xi(t) := \Pr(T_\xi \leq t \mid S_\xi, \mathbf{o}(\xi)). \tag{5.1}$$

This is a conditional successful-trajectory-length model, not a model of the probability of success or of a possibly infinite hitting time. Unsuccessful collection runs, including runs that encounter dead ends or exhaust the collection budget, are excluded rather than treated as censored trajectory lengths.

Although Bayes DistNet was introduced for runtime-distribution prediction (Chapter 4), the model itself only requires a positive-valued target together with optional censoring indicators. Here, we repurpose it to model successful trajectory lengths instead of wall-clock runtimes. The predictor uses the channel-first Sokoban state observation described in Appendix B, rather than the handcrafted feature vectors used in the Chapter 4 experiments.

Given an instance's state observation, the model produces a predictive distribution $\hat{F}_\xi$ over successful trajectory lengths. A shift value is then extracted from this distribution using a fixed quantile level $\tau \in (0, 1)$. Specifically, the predicted shift for instance ξ is defined as

$$\hat{s}_\tau(\xi) := \lceil Q_{\hat{F}_\xi}(\tau) \rceil, \tag{5.2}$$

where $Q_{\hat{F}_\xi}(\tau)$ denotes the τ-quantile of the predictive distribution. The restriction $\tau \in (0, 1)$ avoids the endpoint ambiguity of continuous distributions with unbounded support for which the upper endpoint quantile is infinite. Because the model predicts positive trajectory lengths and the shift is rounded upward,

the implemented shift is at least one; it is then added to every A6519(i) term. Intuitively, this selects a rollout-length offset that reflects the predicted scale of successful trajectories for the given instance. Smaller values of τ yield more conservative shifts, while larger values produce more aggressive ones.

This predicted value is used to translate the Luby restart schedule. Let A6519(i) denote the i-th term of the Luby sequence (see Section 3.3.2). Then rollout i on instance ξ is assigned length

$$d_i(\xi) = \hat{s}_\tau(\xi) + \text{A6519}(i). \tag{5.3}$$

In this way, the universal schedule is retained, but shifted by an amount that depends on the current problem instance. The proposed method may therefore be viewed as an instance-sensitive generalization of uniformly shifted Luby search: rather than selecting one fixed base offset for all problems, it predicts that offset separately for each instance.

In the experiments below, we also consider more aggressive diagnostic shifts based on the finite set of Monte Carlo predictions produced by Bayes DistNet. If $\hat{y}_\xi^{(1)}, \ldots, \hat{y}_\xi^{(M)}$ are the M predicted trajectory lengths sampled from the model for instance ξ, define the sampled-maximum shift

$$\hat{s}_{\max}(\xi) := \left\lceil \max_{r=1,\ldots,M} \hat{y}_\xi^{(r)} \right\rceil.$$

For a multiplier $\rho \geq 1$, we define the corresponding maximum-multiplier shift as

$$\hat{s}_{\rho\max}(\xi) := \left\lceil \rho \max_{r=1,\ldots,M} \hat{y}_\xi^{(r)} \right\rceil.$$

These shifts are not distributional quantiles. They are included only as diagnostic stress tests to check whether performance improves simply by using larger shifts, rather than by selecting an informative instance-specific scale.

5.2.2 Runtime Bound

The same argument used to bound LubyTS in Section 3.3.2 can be adapted to the distribution-guided setting by conditioning on the shift selected for a particular problem instance (see Theorem 3.3.2). In other words, once the

model has chosen an instance-specific shift $X = \hat{s}_\tau(\xi)$, the resulting search procedure can be analyzed as a fixed shifted restart schedule, yielding the following expected search-effort bound. For a model-generated random shift, the theorem applies after the Monte Carlo prediction has realized a fixed shift $X = x$; it does not require the conditional trajectory-length model to estimate the policy's overall success probability.

Theorem 5.2.1. *Let $X \in \mathbb{N}_0$ be a fixed shift, and let* LubyTS_X *denote the variant of LubyTS which samples rollout i to depth $d_i = X + A6519(i)$. For $u \in \mathbb{N}_1$, define*

$$q_X(u) := \pi^+_{X+u},$$

where π^+_{X+u} is the probability that a policy-generated trajectory reaches a goal within at most $X + u$ steps. Then, under the sampled-transition convention of Section 3.3, the expected number of sampled transitions LubyTS_X *requires before returning some (random) first-hitting goal node $n^* \in G_{X+u} \subseteq \mathcal{N}^g$ is bounded by*

$$\mathbb{E}[L(\mathrm{LubyTS}_X, n^*)] \leq \min_{\substack{u \in \mathbb{N}_1 \\ q_X(u) > 0}} \left[u + \frac{u}{q_X(u)} \left(\log_2 \frac{u}{q_X(u)} + 6.1 \right) + \frac{X\, 2^{\lceil \log_2 u \rceil}}{q_X(u)} \right]. \tag{5.4}$$

Equivalently, since $2^{\lceil \log_2 u \rceil} \leq 2u$,

$$\mathbb{E}[L(\mathrm{LubyTS}_X, n^*)] \leq \min_{\substack{u \in \mathbb{N}_1 \\ q_X(u) > 0}} \left[u + \frac{u}{q_X(u)} \left(\log_2 \frac{u}{q_X(u)} + 6.1 + 2X \right) \right]. \tag{5.5}$$

If there is no u such that $q_X(u) > 0$, then the shifted schedule has no finite success-probability depth to exploit, and the theorem provides no finite guarantee.

The proof is in Appendix D.1. When $X = 0$, the shifted bound reduces to the original LubyTS bound. For $X > 0$, the effect of the shift appears in two competing ways. On one hand, the success probability term becomes π^+_{X+u}, so the bound can improve when shifting the schedule moves rollouts into a depth range with substantially more solution probability mass. On the other hand,

the shift is paid on every rollout, which introduces the additional overhead term

$$\frac{X\, 2^{\lceil \log_2 u \rceil}}{\pi^{+}_{X+u}}.$$

Thus, the theorem does not say that shifting is always beneficial. Rather, it formalizes the tradeoff exploited by distribution-guided LubyTS: a good instance-specific shift can reduce wasted shallow rollouts, while an overly large shift can increase the cost of each failed trial.

5.2.3 Training Data and Censoring

Training data for the predictor are obtained by collecting successful trajectory lengths on the 10,000 problems used for training and validation. We use a fixed policy obtained from the online Bootstrap process with PHS*, and run LubyTS 10 times with distinct random seeds on each problem. Each run has a budget of 20,000 sampled transitions (environment interactions). When LubyTS finds a solution within this budget, its rollout length is recorded; runs that do not find a solution produce no finite trajectory-length observation for that seed and are not included as censored samples. This procedure yields 86,056 observed rollout lengths from 100,000 possible problem-seed runs.

In the censoring experiment below, censoring is applied artificially to these observed successful trajectory lengths. When an observed length exceeds the censoring cutoff, it is recorded at the cutoff and treated as right-censored, meaning that only a lower bound on that observed successful trajectory length is retained.

A key advantage of using Bayes DistNet is that such artificially censored observations can still be incorporated during training through the censoring-aware likelihood given by Equation 4.6 from Chapter 4. Thus, artificially censored successful observations do not need to be discarded entirely. Even when the exact successful trajectory length is hidden by the censoring procedure, the model can still learn that it exceeds the cutoff, which helps preserve information about the upper tail of the trajectory-length distribution. This is crucial for the present application, since the shift applied to the Luby sched-

ule is intended precisely to compensate for cases in which useful rollouts are relatively deep.

5.3 Experiments

This section evaluates whether distribution-guided shifts improve the practical performance of LubyTS. The proposed method is compared against two baselines. The first is standard LubyTS, which uses the unshifted universal schedule from A6519. The second is a uniformly shifted variant of LubyTS, which applies the same global shift to every problem instance. This second baseline isolates the benefit of applying a uniform shift relative to the unshifted schedule, while the proposed method evaluates a problem-specific shift under the selected rule. The reported validation and test curves each use a single fixed LubyTS random seed; they are not averages over repeated search runs and do not include uncertainty bands. For the distribution associated with the Bayes DistNet loss function, we use the lognormal distribution, as Tuero and Buro [102] showed it works well across multiple scenarios. Unlike the fully connected model used in the Chapter 4 experiments, this application uses a convolutional ResNet variant with variationally sampled convolutional weights to process state observations.

5.3.1 Environment

To test our proposed method, we evaluate on the Sokoban environment (see Appendix B.4). The agent is tasked with pushing boxes onto designated goal locations. Since the agent can only push boxes, not pull, the environment state can become deadlocked due to boxes getting stuck along walls. Sokoban is PSPACE-hard [21], and we use the problems from Boxoban [41]. The 10,000 problems used for trajectory-length collection are split into a training set and a validation set. The training set is used to fit the empirical trajectory-length distribution for the uniform-shift baseline and train Bayes DistNet. The validation set is used only to select the shift rule, such as the quantile level or sampled-maximum multiplier. The held-out test set is used only for the fi-

nal evaluation after all shift rules have been fixed. The implementation uses Python 3.12.12 and PyTorch 2.9.0 on an Intel i9-7960X CPU with an NVIDIA RTX 3090 GPU and 128 GB of system memory, running Ubuntu 24.04. For all validation and test experiments, a maximum budget of 256,000 sampled transitions is used. In our experiments, we use the first 10,000 training problems, split into 9,000 training problems and 1,000 validation problems, and the first 1,000 test problems. This is the fixed source ordering used in the original experiments; the results are conditional on this split and do not establish that the source ordering is random.

5.3.2 Selecting the Shift Rule

Both shifted methods require a rule for converting trajectory-length data into a shift value. This rule is selected on a validation set that is disjoint from both the training set and the final test set. The training set is used to fit the empirical trajectory-length distribution for the uniform baseline and to train Bayes DistNet. The validation set is then used to choose the quantile level or sampled-maximum multiplier using an equal-width sorted-effort score, calculated separately for the uniform-shift and Bayes DistNet candidate families. Within either family, let $c_{r,(j)}$ denote the j-th smallest raw sampled-transition count among the validation problems solved by candidate rule r, and let K be the smallest number of validation problems solved by any candidate in that family within the 256,000-transition budget. We select the rule minimizing

$$A(r) := \sum_{j=1}^{K} c_{r,(j)}. \tag{5.6}$$

Thus, every candidate within a family is compared over the same number of solved problems; the criterion measures cumulative search effort over this common solved prefix and does not reward a rule merely for solving more than K problems. The two resulting scores use potentially different values of K and are used only to select a rule within their respective families. Once selected, the shift rules are fixed and evaluated on the held-out test set.

Table 5.1: Validation statistics for selecting the uniform LubyTS shift. The common solved-prefix width K is the minimum value in the Solved column, and $A(r)$ is the raw sampled-transition score from Equation (5.6). Lower $A(r)$ is better.

Shift Rule	Solved	$A(r)$
$\tau = 0.2$	889	9,272,522
$\tau = 0.4$	897	8,551,908
$\tau = 0.6$	906	8,063,994
$\tau = 0.8$ (selected)	912	5,463,424
max	924	7,372,754

Uniform Shift

For the uniform-shift baseline, a single empirical distribution is formed from the successful trajectory lengths observed on the training set. For each candidate quantile level $\tau \in (0, 1)$, the corresponding empirical quantile is used as a global shift value. We also evaluate the empirical maximum observed trajectory length as a finite-sample diagnostic shift, denoted by max in Figure 5.1. LubyTS is then evaluated on the validation set using each candidate global shift, and the quantile level that gives the best validation performance is selected. This produces one global shift rule, which is fixed before evaluating on the held-out test set. Table 5.1 reports the validation statistics used to select this rule. The resulting sampled-transition curves for different quantile levels are shown in Figure 5.1.

Bayes DistNet

The proposed method uses the same validation-selection protocol, but at the level of individual instances. Bayes DistNet is first trained using only the trajectory-length data from the training set. For a validation instance ξ, the model predicts a trajectory-length distribution $\hat{F}_\xi$, from which the instance-specific shift is computed for each candidate shift rule. We then evaluate the corresponding shifted LubyTS procedure on the validation set and select the rule that gives the best validation performance. This selected rule is then fixed and used without further tuning on the held-out test set. The candidate rules

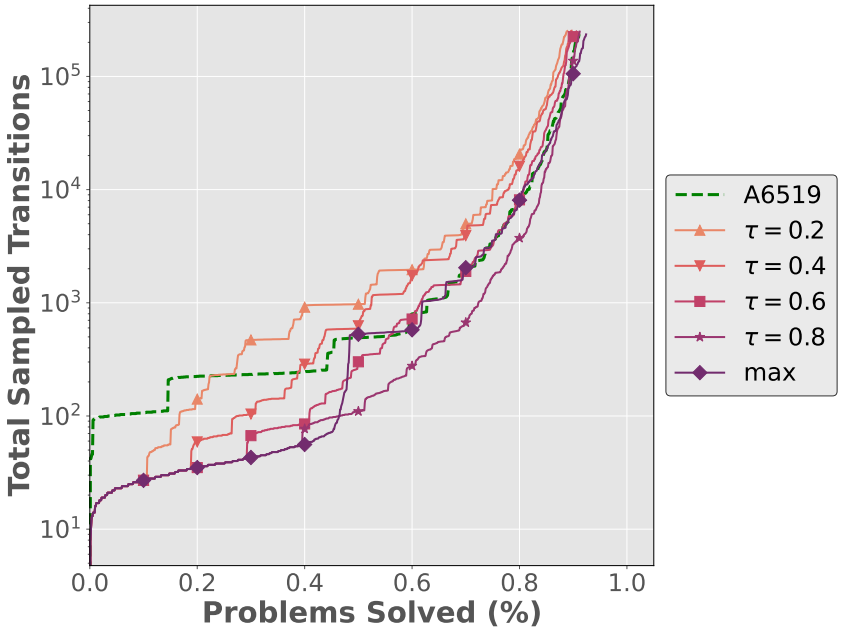


Figure 5.1: Validation-set search efficiency of LubyTS with a uniform depth shift for varying shift rules. Curves labelled by τ use empirical quantile shifts, while the curve labelled max uses the empirical maximum observed trajectory length as the shift. The problems on the x-axis are sorted by the required number of sampled transitions. The total number of sampled transitions on the y-axis is shown in log scale.

include quantile shifts with $\tau \in (0, 1)$, as well as the finite-sample sampled-maximum rules max, $2 \times$ max, and $4 \times$ max.

Table 5.2 reports the validation statistics used to select the instance-specific rule. The resulting sampled-transition curves for different shift rules are shown in Figure 5.2.

Analysis

Across both figures, a similar tradeoff can be observed in the choice of shift rule. For the easier portion of the solved set, all shifted variants outperform the unshifted Luby baseline, indicating that the universal schedule spends substantial effort on shallow rollouts that are unlikely to solve these instances. As the solved fraction increases, however, the more conservative shifts begin to lose this advantage and can eventually underperform the A6519 baseline. This suggests that such shifts are large enough to make failed rollouts more

Table 5.2: Validation statistics for selecting the Bayes DistNet instance-specific shift rule. The common solved-prefix width K is the minimum value in the Solved column, and $A(r)$ is the raw sampled-transition score from Equation (5.6). Lower $A(r)$ is better.

Shift Rule	Solved	$A(r)$
$\tau = 0.2$	895	8,816,779
$\tau = 0.4$	905	8,002,951
$\tau = 0.6$	908	6,514,845
$\tau = 0.8$	932	6,098,823
max (selected)	933	4,475,679
$2 \times$ max	922	4,939,351
$4 \times$ max	926	7,507,953

expensive, but still too small to move the search reliably into the depth range where solutions become likely. More aggressive shifts remain competitive over a wider range because they probe deeper more quickly, but this trend does not continue indefinitely: once the shift becomes too large, performance again degrades as the search overcommits computation to long unsuccessful rollouts. Overall, the figures suggest an intermediate "sweet spot" for the shift rule, balancing the reduction in shallow-rollout waste against the cost of increasingly aggressive failed trajectories.

For the universal shift baseline given in Figure 5.1, the validation-selected quantile is $\tau = 0.8$. For the Bayes DistNet model given in Figure 5.2, the selected rule is the sampled-maximum shift $\hat{s}_{\max}(\xi)$. This is an operational rule based on $M = 16$ Monte Carlo predictions, not an intrinsic quantile of $\hat{F}_\xi$; its behaviour can depend on M and on the prediction samples. We additionally evaluated more aggressive shifts of two and four times the sampled maximum, namely $\hat{s}_{2\max}(\xi)$ and $\hat{s}_{4\max}(\xi)$. These more aggressive variants performed markedly worse. This is consistent with the tradeoff observed: while larger shifts can reduce shallow-rollout waste, overly large shifts overcommit computation to long failed trajectories. These comparisons show only that, under the validation protocol used here, arbitrarily increasing the selected maximum is harmful; they do not establish that the sampled maximum is the optimal instance-specific scale or that it would remain selected for another

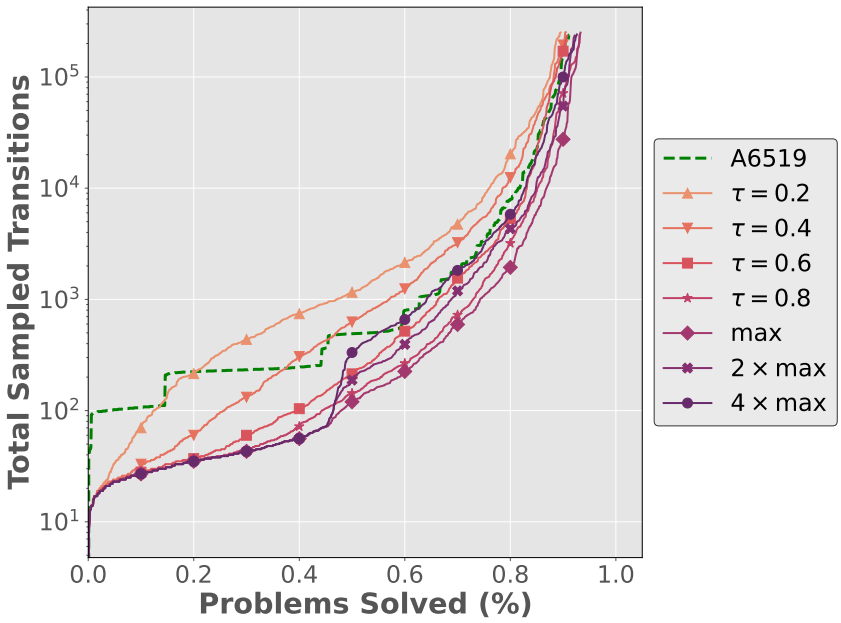


Figure 5.2: Validation-set search efficiency of LubyTS with a problem-specific depth shift from Bayes DistNet for varying shift rules. The problems on the x-axis are sorted by the required number of sampled transitions. The total number of sampled transitions on the y-axis is shown in log scale. Curves labelled max, $2 \times$ max, and $4 \times$ max use the sampled maximum predicted trajectory length, twice the sampled maximum, and four times the sampled maximum, respectively. These are diagnostic maximum-multiplier shifts rather than quantile levels.

value of M. Because the uniform and Bayes DistNet methods select different shift statistics, their comparison does not isolate the effect of instance personalization from differences in shift magnitude. The uniform-shift quantile $\tau = 0.8$ and the Bayes DistNet sampled-maximum shift rule are fixed in the censoring experiments below.

5.3.3 Effect of Censoring

We next examine how the two shifted methods behave when the trajectory-length data used to determine the shift are increasingly censored. In this experiment, censoring is introduced by applying a cutoff to the training trajectory lengths. For a censoring level γ, the cutoff is chosen so that approximately a γ fraction of the training observations exceed the cutoff. Trajectories whose lengths do not exceed the cutoff remain uncensored, while trajectories

whose lengths exceed the cutoff are recorded at the cutoff value and marked as right-censored. Thus, the exact values of the largest trajectory lengths are hidden, but the lower-bound observations are retained. We consider censoring levels of 0%, 20%, 40%, and 60%. For each censoring level, censoring is applied only to the training trajectory-length data. The uniform empirical distribution and Bayes DistNet model are then fit from this censored training data. The uniform baseline does not use a censoring-aware likelihood, whereas Bayes DistNet uses the censoring-aware likelihood from Chapter 4; this comparison therefore assesses the complete proposed pipeline against a naïve global empirical baseline, rather than isolating the effect of instance-specific prediction alone. The shift rules themselves are not reselected on the test set: the uniform-shift baseline uses the validation-selected quantile level, and Bayes DistNet uses the validation-selected sampled-maximum rule. The resulting methods are then evaluated on the held-out test set. Table 5.3 provides a tabular summary of the sorted-effort curves in Figure 5.3. For each censoring level γ, K_γ is the smallest solved count among the compared methods, $A_\gamma(r)$ is the sum of the first K_γ sorted raw sampled-transition counts, and $A_\gamma(r)/K_\gamma$ is the corresponding mean effort.

Figure 5.3 shows that under no censoring, both shifted methods improve over the unshifted LubyTS baseline, with Bayes DistNet performing slightly better overall. As the level of censoring increases, however, the two shifted methods behave quite differently. The Bayes DistNet curve remains consistently below the LubyTS baseline over the solved set, indicating that it largely preserves its advantage even when a substantial fraction of the largest trajectory lengths are censored. In contrast, the uniform-shift baseline degrades progressively as censoring increases. This is visible in the point at which the blue and green curves intersect: as the censoring level rises, that intersection moves steadily to the left, meaning that the uniform-shift method becomes worse than the unshifted Luby baseline over an increasingly large fraction of the solved problems.

This pattern is consistent with how the two methods use the training data, while also reflecting their different treatment of censored observations.

Table 5.3: Tabular summary of the sorted-effort curves under training-data censoring. For each censoring level, the common solved-prefix width K_γ is the minimum value in that block's Solved column. $A_\gamma(r)$ is the raw cumulative sampled-transition effort over this prefix, and $A_\gamma(r)/K_\gamma$ is the mean effort per problem.

Method	Solved	$A_\gamma(r)$	$A_\gamma(r)/K_\gamma$
Censoring Level: 0%			
LubyTS (A6519)	912	6,995,812	7,670.85
Uniform shift ($\tau = 0.8$)	912	5,558,676	6,095.04
Bayes DistNet (max)	933	4,484,676	4,917.41
Censoring Level: 20%			
LubyTS (A6519)	912	6,946,217	7,683.87
Uniform shift ($\tau = 0.8$)	904	6,737,711	7,453.22
Bayes DistNet (max)	928	5,460,964	6,040.89
Censoring Level: 40%			
LubyTS (A6519)	912	6,939,344	7,718.96
Uniform shift ($\tau = 0.8$)	899	8,390,799	9,333.48
Bayes DistNet (max)	926	4,357,028	4,846.53
Censoring Level: 60%			
LubyTS (A6519)	912	6,892,954	7,736.20
Uniform shift ($\tau = 0.8$)	891	7,546,599	8,469.81
Bayes DistNet (max)	922	5,637,223	6,326.85

The uniform-shift baseline relies directly on the observed empirical trajectory-length distribution, so censoring hides precisely the upper-tail values that are most informative for choosing an effective shift. As a result, the estimated global shift becomes increasingly biased toward smaller values, causing the method to drift back toward the behaviour of the original universal schedule while still paying the cost of an imperfect shift. Bayes DistNet, on the other hand, does not discard these partially observed runs. Although the exact trajectory lengths are not available, the model still receives the information that they exceed the cutoff, allowing it to retain a more informative picture of the right tail of the instance-specific trajectory-length distribution. The resulting shifts degrade more gradually under the reported protocol.

Taken together, these results show more favourable behaviour for the proposed pipeline under incomplete training data in this experiment. In the uncensored setting, both shifted methods can improve over the universal strat-

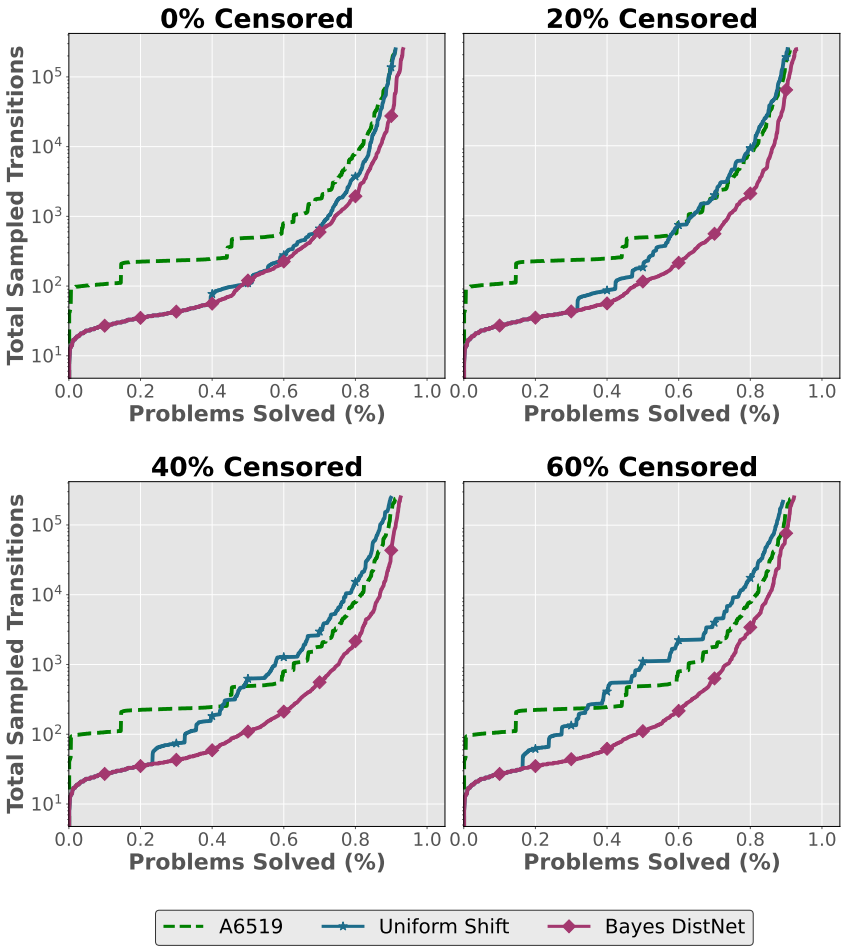


Figure 5.3: Effect of training-data censoring on the quality of shift estimation for LubyTS. The methods compared are unshifted LubyTS using A6519, uniform shift with fixed quantile $\tau = 0.8$ estimated from the censored empirical distribution, and Bayes DistNet with the sampled-maximum shift rule trained on censored and uncensored observations.

egy by reducing effort spent on shallow rollouts. Under increasing censoring, however, only the Bayes DistNet approach continues to show the observed improvement in this single-seed held-out test. This is precisely the regime in which the censoring-aware distributional model from Chapter 4 becomes useful for search control.

5.4 Chapter Summary

This chapter showed that, under the reported experimental protocol, the restart behaviour of LubyTS can be improved by replacing a fixed global shift

with an instance-specific shift predicted from conditional successful-trajectory-length distributions. Using Bayes DistNet for this purpose showed more favourable behaviour under artificial censoring than the naïve empirical global-shift baseline, while preserving the benefit of moving computation away from shallow, unproductive rollouts. The chapter therefore extends the rollout-based policy tree search methods of Chapter 3 by demonstrating how distributional prediction can be used to adapt restart depth to the problem instance being solved.

Chapter 6

Subgoal-Guided Policy Heuristic Search with Learned Subgoals

This chapter is joint work with Michael Buro and Levi Lelis. It was previously published at the Forty-second International Conference on Machine Learning [103]. This chapter introduces extensions to Levin Tree Search and PHS**. Readers are directed to Section 3.4 for a complete overview of those methods.*

Within this thesis, this chapter studies explicit intermediate structure as a form of search control: learned subgoals direct policy-guided search and allow training to use failed search trees.

6.1 Introduction and Overview

This chapter focuses on solving single-agent deterministic search problems, using minimal domain knowledge. In particular, we are interested in "*needle-in-haystack*" problems, where finding any solution can be challenging. Many important problems can be represented as a deterministic single-agent search problem, such as robotic navigation [99] and network routing [63]. A recent line of research to address this class of problems is *policy tree search* algorithms [78, 79], which use a learned policy to guide the search, *i.e.*, a probability distribution over the set of actions available to the agent. Orseau et al. [78] showed that LevinTS (Section 3.4.1) provides an upper bound on the number of node expansions required to find a solution, with the bound depending on the quality of the policy. Orseau and Lelis [79] generalized this view to search

loss, showing how policy-guided search can be analyzed and trained under cumulative expansion losses; in the unit-loss case, these search-loss bounds reduce to bounds on node expansions.

Policy-Guided Heuristic Search PHS* [79] (Section 3.4.2) is a policy-guided search algorithm that combines a learned policy and a heuristic function. While there have been previous approaches to combining a policy and heuristic/value function, such as the PUCT-based search algorithms [54, 87] like AlphaZero [94] and MuZero [90] in adversarial search domains like Go and Chess, Orseau and Lelis [79] show that PHS* is better suited for the single-agent search problems considered here.

Policy tree search methods, including PHS*, are usually trained online using the Bootstrap search-and-learn process [7] (Section 2.3.3). The Bootstrap process initially uses randomly initialized neural models encoding the heuristic and the policy to iteratively solve a subset of the problems. If the search cannot solve problems within a search budget, the resulting trees are discarded; if at least one problem is solved, the models are optimized on the solution trajectories found. A problem with this approach is that much of the search effort is wasted on failed attempts, especially in the early iterations of learning, with untrained neural models.

A key innovation of our approach is that we use these failed search attempts to learn subgoals, which can shorten the planning horizon and ease the learning process. These failed attempts allow us to learn policies that navigate between subgoals. Using subgoals to break down the search horizon into smaller pieces has shown success in complex domains, as demonstrated in kSubS [22], AdaSubS [110], and HIPS [57]. An issue with some of these approaches is that they lack completeness, *i.e.*, they might fail to return a solution even if one exists. HIPS-ε [58] is a variant of HIPS that is complete. However, HIPS-ε, as well as kSubS, AdaSubS, and HIPS, require precomputed datasets of solution trajectories, which may be prohibitively costly in complex environments.

We propose a hierarchical policy for policy-guided tree search algorithms that retains the completeness conditions of the underlying search method and does not require precomputed solution trajectories. This is achieved by learning subgoal-guided low-level policies from the Bootstrap data. In our approach, a subgoal generator produces a set of subgoals for a given state encountered in the search. The low-level policy is then conditioned on each of these subgoals and on the current state, thus providing one probability distribution over actions for each subgoal. A high-level policy produces a distribution over the generated subgoals, which gives an importance weighting for each of the low-level policies. The low-level policies are then mixed, based on the weighting given by the high-level policy, to produce a final policy that guides the search. In addition, we propose a method to learn these subgoals and policies using the data provided by the tree search, even in contexts where a budget for the search causes early termination without a solution.

In experiments, we evaluate training expansion efficiency, measured as the cumulative node expansions required to learn effective policies, relative to other search algorithms trained with the Bootstrap algorithm in a variety of problem domains. We also show that policy tree search algorithms using our subgoal-based policy can learn how to solve problems from domains that HIPS-ε cannot solve.

Our contributions can be summarized as follows. We provide a novel subgoal discovery method for policy-guided tree search algorithms that learns from data collected during search. Our method for learning policies automatically reduces complex problems into easier sub-problems, by automatically learning subgoals. Moreover, it can train neural policies from both successful and failed searches, unlike other algorithms that learn with the Bootstrap process.

6.2 Related Work

Subgoal Search Our work is primarily related to other works that use subgoals in search. kSubS [22] learns a subgoal generator to predict resulting states that are k steps ahead of the current state. AdaSubS [110] extends

kSubS by learning multiple subgoal generators, each for a different look-ahead depth k. Since both kSubS and AdaSubS search at the subgoal level, they are not complete. HIPS [57] learns to segment trajectories through reinforcement learning, then trains a subgoal generator using these trajectories and a low-level policy to reach the generated subgoals. HIPS also searches at the subgoal level, and thus is not complete. HIPS-ε [58] augments the subgoal search by combining the subgoals and the actions of the environment. These approaches require a precomputed dataset of solution trajectories, which our approach does not require. Also, we showed empirically that HIPS-ε might fail to solve problems when reconstructing the state is challenging, while our approach is robust to reconstruction inaccuracies.

Goal-Conditioned Reinforcement Learning Identifying useful subgoal states for policies to navigate between has shown promising results for solving long-horizon tasks. There are many different approaches to how these subgoal states are generated from the history of states the agent has visited. HIGL [50] builds a graph of landmark states, from which a goal is selected. HILL [114] and HESS [62] sample goal states from a learned embedding using distance metrics. L^3P [113] learns a latent space and then learns latent landmarks through clustering, and finally decodes the cluster centroids as goals. DisTop [8] clusters an embedding of the state space and samples a goal from one of the selected clusters. Our method is different in that subgoals are found using only the underlying properties of the state-space graph, without relying on additional metrics to select subgoals such as novelty and reachability measures.

VQVAEs as Subgoal Generators Vector Quantized Variational Autoencoders [105] have become a popular model to use as a subgoal generator. DGRL [47], Choreographer [66], and QPHIL [16] use VQVAEs to generate a target subgoal observation, which is used with the current state as input to policies. VQVAEs are generally trained on an offline dataset, separate from the reinforcement learning task. CQM [60] trains a VQVAE to discretize continuous observations, which is used to create a curriculum over landmarks. Our

method trains a VQVAE to generate subgoal observations, but does so online, while learning the policy. This allows our method to be used in scenarios where generating quality training trajectories is too costly.

6.3 Preliminaries

We follow the same definition of a single-agent deterministic search problem defined in Section 2.1. Readers are pointed towards Section 3.4.1 for the background on Levin Tree Search, and Section 3.4.2 for the background on PHS*, as this chapter builds upon those works. The Bootstrap algorithm [7] is used throughout this chapter to learn a neural policy and/or heuristic functions from search. A detailed explanation of the Bootstrap process is given in Section 2.3.3, and its pseudocode in Algorithm 2.

6.3.1 Problem Statement

Given a set of problem instances K, the objective is to solve them while minimizing the *total search loss*. Concretely, we use the formulation provided by Orseau and Lelis [79]: For a search algorithm S, a loss of $\ell(n) > 0$ is incurred by S expanding node n, and the *search loss* $L(S, n)$ is the sum of individual losses $\ell(n')$ for all nodes n' that have been expanded by algorithm S up to and including n. The *total search loss* is then given by $\sum_{k \in K} L(S, n^*_k)$, where $L(S, n^*_k)$ is the search loss of algorithm S while finding a solution node n^*_k on the k-th problem in K. For example, if $\ell(n) = 1$ for all nodes n, then the objective is to solve the problem instances using as few node expansions as possible.

6.4 Method

We propose learning a subgoal-guided policy for policy search algorithms, which uses learned subgoals from a subgoal generator to guide the search through the use of policies conditioned on subgoals. In this chapter, we define subgoals as states from the underlying state space that the search attempts

to achieve. Unlike previous policy search methods, our policy uses data from budget-bounded searches that fail to find a solution to train its neural policy.

Our method is given in Figure 6.1. Figure 6.1.a shows how the low-level policy, high-level policy, heuristic, and subgoal generator models interact when expanding a node during the search (Sections 6.4.1 and 6.4.2). During the Bootstrap process used to train the policy, the heuristic, and the subgoal generator models, the budgeted search will either terminate without a solution or with a solution trajectory. When the budgeted search terminates without a solution found, Figure 6.1.b shows how the search tree can be used to generate subgoals and trajectories between them for the subgoal generator and low-level policy to learn from, respectively (Section 6.4.3). Finally, Figure 6.1.c shows how we use solution trajectories to train the heuristic model, low-level policy, high-level policy, and subgoal generator (Section 6.4.4).

6.4.1 Subgoal-Guided Policy Search

Instead of a single policy $\pi(\cdot)$ that produces a probability distribution over actions, we employ several subgoal-conditioned low-level policies together with a high-level policy over subgoals. The low-level policy $\pi_\theta^{\text{low}}(a \mid s, \hat{s}_{g_i})$ is conditioned on a state s and a subgoal $\hat{s}_{g_i}$, and represents a probability distribution over actions to achieve $\hat{s}_{g_i}$ from s. For k subgoals, we generate a set of k probability distributions by conditioning π_θ^{low} on each of the k subgoals $\hat{s}_{g_i}$. The high-level policy $\pi_\psi^{\text{hi}}(\hat{s}_{g_i} \mid s)$ gives a distribution over the k subgoals. These low-level and high-level policies are implemented as neural networks parameterized by θ and ψ, respectively.

The *subgoal-guided policy* $\pi^{\text{SG}}(a \mid s)$ is then defined by weighting each subgoal-conditioned low-level policy by the high-level policy and normalizing over the actions applicable in state s. For any $a \in A(s)$, we define

$$\pi^{\text{SG}}(a \mid s) = \frac{\sum_{i=1}^{k} \pi_\psi^{\text{hi}}(\hat{s}_{g_i} \mid s) \cdot \pi_\theta^{\text{low}}(a \mid s, \hat{s}_{g_i})}{\sum_{a' \in \mathcal{A}(s)} \sum_{i=1}^{k} \pi_\psi^{\text{hi}}(\hat{s}_{g_i} \mid s) \cdot \pi_\theta^{\text{low}}(a' \mid s, \hat{s}_{g_i})}. \tag{6.1}$$

In the implementation, every primitive action in the fixed action set is attempted, so the low-level and high-level softmax outputs make the denomi-

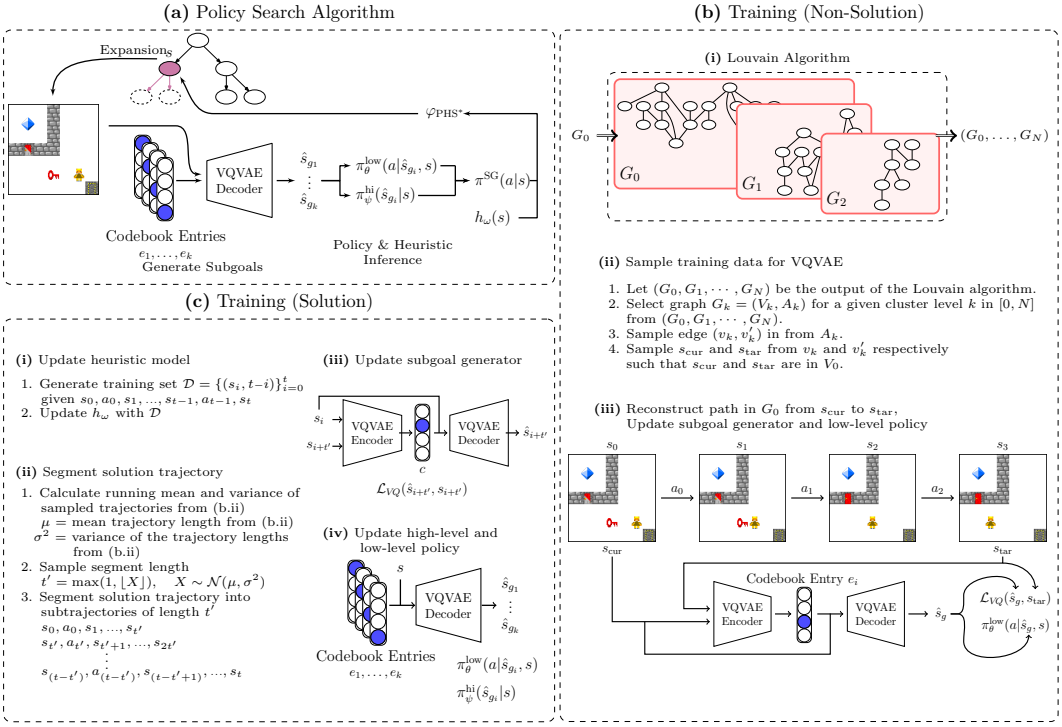


Figure 6.1: **(a) Tree Search**. Policy tree search generates subgoals to use with π^{SG}. **(b) Training (Non-Solution)**. **(i)** The underlying graph that induced the tree search is generated from the parent-child relationships found during search, which is used to create a hierarchy of cluster graphs using the Louvain algorithm. **(ii)** States s_{cur} and s_{tar} are sampled from neighbouring states in G_i from the graphs created by the Louvain Algorithm. **(iii)** The resulting trajectory from s_{cur} to s_{tar} is used to update the subgoal generator and low-level policy. **(c) Training (Solution)**. **(i)** The heuristic is updated using the solution trajectory. **(ii)** The solution trajectory is segmented. **(iii)** The subgoal generator is updated using the partial trajectory. **(iv)** The high-level and low-level policies are updated using the segmented trajectories.

Algorithm 10 LevinTS(π^{SG})

1: **Input:** Root node n_1, policy π^{SG}, budget B, goal set $\mathcal{N}^g$
2: **Output:** Solution path, or failure
3: **return** `BestFirstSearch`$(n_1, \varphi_{\mathrm{SLTS}}, B, \mathcal{N}^g)$

Algorithm 11 PHS*(π^{SG})

1: **Input:** Root node n_1, policy π^{SG}, budget B, goal set $\mathcal{N}^g$
2: **Output:** Solution path, or failure
3: **return** `BestFirstSearch`$(n_1, \varphi_{\mathrm{SPHS}^*}, B, \mathcal{N}^g)$

nator equal to one up to numerical precision. We retain the denominator to make the normalization of the mixture explicit.

We can now define two cost functions which are analogous to φ_{LTS} and φ_{PHS^*} that were given in Equation 3.7 and Equation 3.18, respectively:

$$\varphi_{\mathrm{SLTS}}(n) = \frac{d(n)}{\pi^{\mathrm{SG}}(n)}, \tag{6.2}$$

and

$$\varphi_{\mathrm{SPHS}^*}(n) = \eta_{\mathrm{PHS}^*}(n; \pi^{\mathrm{SG}}) \frac{g_\ell(n)}{\pi^{\mathrm{SG}}(n)}. \tag{6.3}$$

Here, $\eta_{\mathrm{PHS}^*}(n; \pi^{\mathrm{SG}})$ denotes the PHS loss factor with every occurrence of the base path policy replaced by π^{SG}. The two search algorithms that use the subgoal-guided policy, denoted LevinTS(π^{SG}) and PHS*(π^{SG}), are given in Algorithm 10 and Algorithm 11, respectively.

6.4.2 VQVAE Subgoal Generator

We use a Vector Quantized Variational Autoencoder [105] (VQVAE), represented by a neural network parameterized by ϕ, as a subgoal generator. Our design is similar to the one used in HIPS-ε [57].

VQVAE Architecture

VQVAEs utilize a codebook containing k learned D-dimensional embedding vectors $\boldsymbol{e}_i \in \mathbb{R}^D, i = 1, \ldots, k$. Inputs $\boldsymbol{x}$ are encoded by the *encoder*, resulting in an encoded vector $\boldsymbol{z}_e$. The encoding vector is then mapped to a quantized encoding vector $\boldsymbol{z}_q$ which is the *nearest* embedding vector using Euclidean

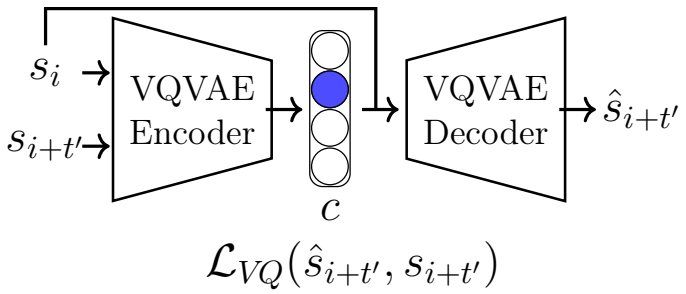


Figure 6.2: The VQVAE subgoal generator architecture.

distance:

$$\boldsymbol{z}_q = \boldsymbol{e}_c, \qquad \text{where } c = \arg\min_i \|\boldsymbol{z}_e - \boldsymbol{e}_i\|_2 . \tag{6.4}$$

The discretized encoding vector is then passed through the *decoder* which produces a reconstructed output $\hat{\boldsymbol{x}}$. The encoder receives a pair of channel-first state observations $(s_{\text{cur}}, s_{\text{tar}})$ and the decoder receives s_{cur} together with the quantized code $\boldsymbol{z}_q$. The state representation is described in Appendix B. The reconstruction loss is the sum of channel-wise cross-entropies between the target and reconstructed binary state observations. The loss used to optimize the encoder and decoder is

$$\mathcal{L}_{VQ} = \mathcal{L}_{\text{rec}}(\hat{\boldsymbol{x}}, \boldsymbol{x}) + \beta \|\boldsymbol{z}_e - sg(\boldsymbol{e}_c)\|_2^2 \tag{6.5}$$

where $sg(\cdot)$ is the stop-gradient operation and $\beta = 0.25$. The codebook is updated separately by exponential moving averages of encoder assignments, rather than by the gradient-updated codebook loss in the original VQVAE objective; implementation parameters are given in Appendix E.3.

The intuition behind the pair of current and target states given to the encoder is that the encoder learns to capture the difference between the two states into the encoding codebook vector. When the current state is given to the decoder at runtime during search, the encoding codebook vector can be used to recover the changes required to be made from the current state to reconstruct the target state. The VQVAE architecture is depicted in Figure 6.2.

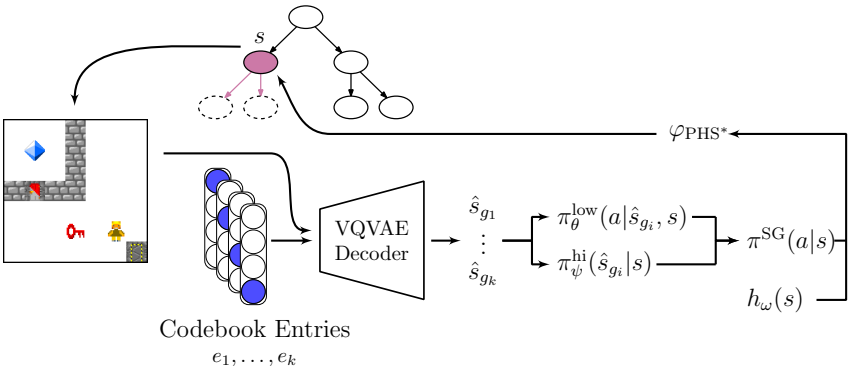
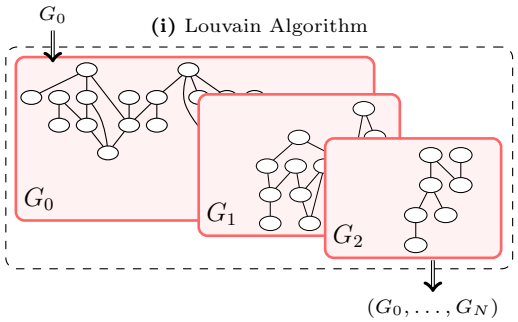


Figure 6.3: VQVAE subgoal generator inference during tree search. The state observations, along with k codebook entries, produce k target subgoal states. Low-level policies corresponding to each subgoal are then mixed, producing a final policy to use in φ_{PHS^*}.

Inference at Search

To generate the i-th subgoal during search, the node's state s and the i-th codebook entry e_i of the VQVAE are given as input to the VQVAE's decoder, which produces a target subgoal state $\hat{s}_{g_i}$. The k generated subgoals are then used as inputs to the low-level and high-level policies, then combined to produce $\pi^{\text{SG}}(\cdot)$, as described in Section 6.4.1.

The difference between our work and HIPS-ε is how the VQVAE subgoal generator is used. We condition low-level policies on the generated subgoal states to guide the search to reach these subgoals, but search is performed at the search problem's action level. HIPS-ε performs search at the subgoal-level, where the resulting state-transition from a node expansion comes from the generated state of the VQVAE. This can be problematic if the generated subgoal states are not accurate reconstructions of actual states because the reconstructed states might not be reachable by the search trying to find the subgoals. We hypothesize that HIPS-ε will not be able to effectively guide the search in complex environments for which state reconstruction is *hard*. By contrast, our policy $\pi^{\text{SG}}(\cdot)$ uses the reconstructed states only to condition the low-level policies; decoded subgoals are not required to be physically valid or reachable states. We further hypothesize that, in this $\pi^{\text{SG}}(\cdot)$ scheme, even imperfect reconstructions of subgoal states can be helpful in guiding the search.



(ii) Sample training data for VQVAE

1. Let $(G_0, G_1, \cdots, G_N)$ be the output of the Louvain algorithm.
2. Select graph $G_k = (V_k, A_k)$ for a given cluster level k in $[0, N]$ from $(G_0, G_1, \cdots, G_N)$.
3. Sample edge (v_k, v'_k) from A_k.
4. Sample s_{cur} and s_{tar} from v_k and v'_k respectively such that s_{cur} and s_{tar} are in V_0.

Figure 6.4: Overview of how the Louvain algorithm produces current and target subgoals for the VQVAE subgoal generator to learn from.

Training the VQVAE Subgoal Generator

To train the VQVAE subgoal generator to generate candidate subgoal states, $(s_{\text{cur}}, s_{\text{tar}})$ are encoded into the continuous latent code z_e, quantized to the *nearest* entry in the *codebook* e_c, then given to the decoder along with s_{cur} to produce a reconstructed target $\hat{s}_{\text{tar}}$. The reconstruction loss $\mathcal{L}_{\text{rec}}(s_{\text{tar}}, \hat{s}_{\text{tar}})$ given in Equation 6.5 is used to encourage the reconstructed target subgoal state to match the given target state, along with a commitment penalty that encourages the encoder output to remain close to its assigned codebook entry. The full loss is denoted as $\mathcal{L}_{VQ}$, and this procedure is used to train the VQVAE in scenarios where the search terminates without a solution being found (Section 6.4.3) and when the search returns a solution trajectory (Section 6.4.4), with the only difference being how s_{cur} and s_{tar} are selected.

6.4.3 Training From Non-Solution Search Trees

We leverage data from search trees that during the online Bootstrap training process would normally be discarded. This allows the subgoal-conditioned low-level policy and subgoal generator networks to be trained on problem instances where LTS and PHS* cannot find solutions. In the empirical section, we show how this can lead to significant savings in the total search loss incurred during the Bootstrap training process.

Different best-first search algorithms potentially generate different trees, which induce different subgraphs of the underlying state space. We denote the directed subgraph containing the generated states as G_0. Each vertex

represents one unique environment state; repeated encounters with a transposition reuse its vertex and add the corresponding directed transition edge, without creating an additional vertex. All edges are unweighted. For Louvain clustering, we treat G_0 as undirected by ignoring transition directions and retaining an edge whenever either directed transition is present. If the search cannot solve a problem instance during the online training process, we use this undirected interpretation of G_0 as the initial graph for the Louvain clustering algorithm [11]. The Louvain algorithm iteratively creates a hierarchy of graphs $(G_0, \ldots, G_N)$ through clustering (Figure 6.4.i). Each iteration of the Louvain algorithm consists of creating a hierarchical graph $G_{i+1} = (V_{i+1}, A_{i+1})$ from the current graph $G_i = (V_i, A_i)$, where V_{i+1} represents a partition of V_i, where each v in V_{i+1} represents a part in this partition. The set A_{i+1} contains edges (v_{i+1}, v'_{i+1}) for v_{i+1} and v'_{i+1} in V_{i+1} only if there are a v_i in v_{i+1} (v_i is in the part v_{i+1}) and a v'_i in v'_{i+1} (v'_i is in the part v'_{i+1}) such that (v_i, v'_i) in A_i. This process continues until a graph consisting of a single node is produced, or if $G_{i+1} = G_i$. The pseudocode of this algorithm is given in Appendix E.1, Algorithm 18.

We use the Louvain algorithm because Evans and Şimşek [28] showed it to find meaningful structure in other state spaces. The difference between our method and that of Evans and Şimşek [28] is that we learn to generalize subgoals across instances, where they assumed the state-transition graph for each individual problem was known a priori.

For a given clustering level k, the graph G_k produced by the Louvain algorithm is selected. Two states s_{cur} and s_{tar} are sampled from adjacent clusters v_k and v'_k, respectively (Figure 6.4.2.ii). Cluster adjacency alone does not guarantee directed reachability. We therefore perform a directed graph traversal over the original transitions in G_0 from s_{cur} to s_{tar}; if no action-direction path exists, the pair is rejected and resampled until a valid directed path is found.

The VQVAE subgoal generator is then trained using $(s_{\text{cur}}, s_{\text{tar}})$ as input to generate the reconstructed target subgoal state $\hat{s}_g$ of s_{tar}, as described in Section 6.4.2. The low-level π_θ^{low} policy is trained using the sequence of states

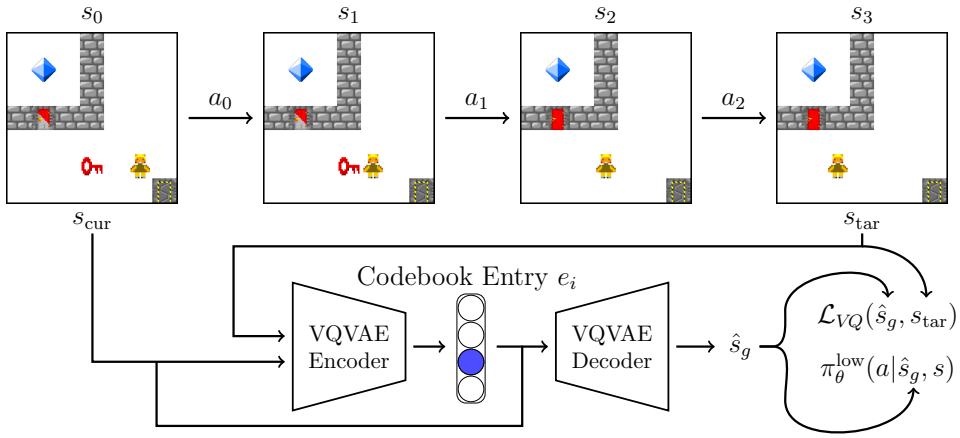


Figure 6.5: Overview of how the subgoal generator learns from a partial trajectory. The start and end of the trajectory are the current and target state respectively. The VQVAE subgoal generator uses this pair to update, and the low-level policy is updated using the partial trajectory.

and actions between s_{cur} and s_{tar} in G_0, while conditioned on $\hat{s}_g$ (Figure 6.5).

6.4.4 Training From Solution Search Trees

If the search finds a solution, then the solution trajectory is used to train the heuristic model h_ω using the costs of the current state along the solution path to the goal state (Figure 6.6.i). Similarly to previous work, the heuristic function is learned by minimizing the mean squared error between the model's predicted value and the actual cost to go for the states along the solution path [7].

Instead of clustering, we segment the solution trajectory $(s_0, s_1, \ldots, s_t)$ into sub-trajectories of length at most t'. For $q = 0, 1, \ldots, \lceil t/t' \rceil - 1$, let $i_q = qt'$ and define the q-th segment as $(s_{i_q}, s_{i_q+1}, \ldots, s_{\min(i_q+t',t)})$. Here, $t' = \max(1, \lfloor X \rfloor)$ where X is sampled once per solution trajectory from a Normal distribution with mean and variance calculated from the lengths of sampled trajectories from the non-solution trees (Figure 6.6.ii). These lengths are maintained in a circular buffer of size 10,000. Before the buffer contains an observation, we use $X \sim \mathcal{N}(5, 1)$ for the initial draw; segment endpoints are always clamped to the remaining solution length by $\min(i_q + t', t)$. Sampling t' ensures that the average number of state transitions between the s_i and $s_{\min(i+t',t)}$ states is

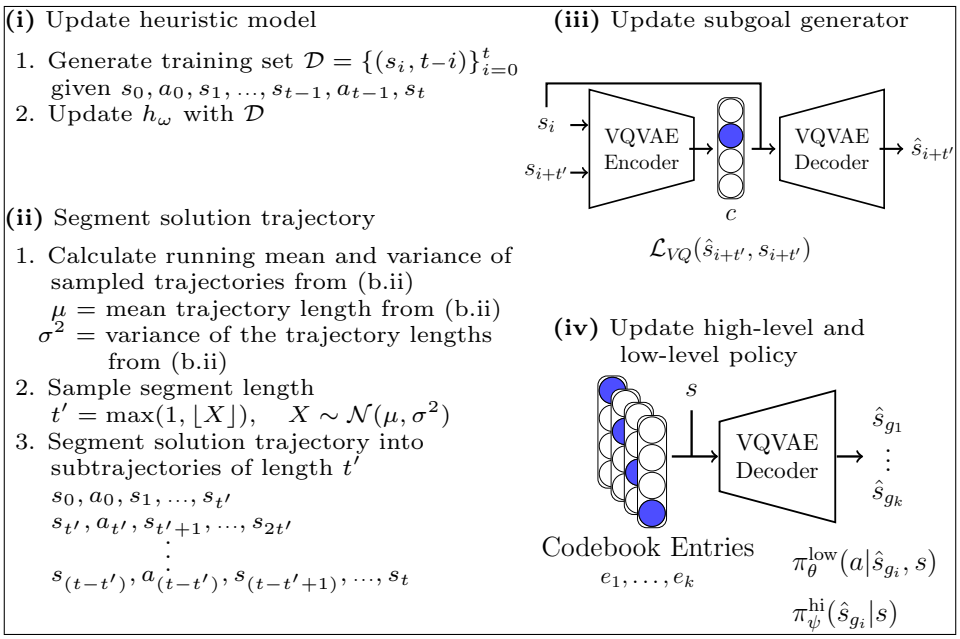


Figure 6.6: Overview of training from a solution trajectory. **(i)** The heuristic is updated using the solution trajectory. **(ii)** The solution trajectory is segmented. **(iii)** The subgoal generator is updated using the partial trajectory. **(iv)** The high-level and low-level policies are updated using the segmented trajectories.

close to s_{cur} and s_{tar} sampled in the non-solution case.

For each sub-trajectory, s_{cur} and s_{tar} are represented by s_i and $s_{\min(i+t',t)}$, respectively. Similarly to Section 6.4.3, the VQVAE subgoal generator is trained using $(s_i, s_{\min(i+t',t)})$ as input to generate the reconstructed target subgoal state $\hat{s}_{\min(i+t',t)}$ (Figure 6.6.iii). The low-level policy π_θ^{low} is trained using the sequence of states and actions $(s_i, s_{i+1}, \ldots, s_{\min(i+t',t)})$ while conditioned on the subgoal $\hat{s}_{\min(i+t',t)}$. The high-level policy π_ψ^{hi} is trained by generating the k subgoals following the process outlined in Section 6.4.2 for the state s_i, of which $\hat{s}_{\min(i+t',t)}$ is one of them, and uses $\hat{s}_{\min(i+t',t)}$ as the target for training the policy π_ψ^{hi} (Figure 6.6.iv). All policies are trained while minimizing the Levin loss (see Section 3.4.4).

6.4.5 Analysis of π^{SG}

Policy-guided search algorithms can be instantiated with the subgoal-guided policy π^{SG} by replacing the original policy π in their evaluation functions.

This substitution does not change the underlying search space: the search still expands primitive actions from $\mathcal{A}(s)$, while the generated subgoals only affect how probability mass is assigned to those actions. Hence, the guarantees from Chapter 3 apply to the corresponding base search algorithm with π replaced by π^{SG}, provided the assumptions of that algorithm are satisfied.

For LevinTS(π^{SG}), the Levin-style bound applies to any goal node n^* such that $\pi^{\mathrm{SG}}(n^*) > 0$. In particular, the finite bound is with respect to solution paths that are policy-reachable under π^{SG}; if all solution paths have zero probability, no finite policy-based bound is obtained. For PHS*(π^{SG}), the PHS bound likewise applies after replacing π by π^{SG}, assuming a non-negative expansion-loss function, $\eta(n) \geq 1$, and a finite-cost solution node. Completeness for the full underlying search problem requires the usual loss assumptions, namely that each expansion has non-negative loss and that no infinite path has finite cumulative path loss, together with sufficient policy support so that at least one solution path is assigned positive probability. This is unlike other subgoal tree search algorithms such as kSubS, AdaSubS, and HIPS, with the exception of HIPS-ε.

6.5 Experiments

The goal of our experiments is to test whether the learning process of π^{SG} improves training expansion efficiency relative to existing approaches to learning policies for search algorithms. We use this term to mean fewer cumulative node expansions during Bootstrap training; wall-clock training efficiency is reported separately where applicable. In the extreme case, this can enable policies to be learned on complex domains in which the comparison methods require prohibitively large expansion or wall-clock budgets before making progress. We also evaluate the quality of the learned policies in terms of expansions and solution cost on a separate test set.

6.5.1 Environment Domains

Below are the environments tested throughout these experiments. For additional details about each environment, see Appendix B.

Box-World A 20 × 20 room with coloured keys and locked boxes [109]. Keys are consumed when used on a lock of the matching colour, with the agent receiving a new key matching the colour of the box. The objective is to open the designated goal box by opening the correct sequence of locks. If the agent opens a corresponding lock on the wrong box, the resulting scenario can become a dead end.

CraftWorld A 14 × 14 room with various raw materials and workbenches [5]. The agent can collect raw materials to create tools. We generate problems with the open-source level generator[1] of the procedure detailed by Andreas et al. [5].

BoulderDash A 14 × 14 room where the agent must gather a number of diamonds to unlock the exit door, and proceed to enter it. Diamonds can be inside locked sub-rooms in which the corresponding key must be gathered first. Various dirt elements exist, which disappear once the agent walks over them, and can significantly increase the state-space size. We generate problems by randomly placing the keys, diamonds, and an exit door.

Sokoban A 10 × 10 room where the agent must push boxes onto specified goal locations. The box can only be pushed; not pulled, so boxes can get stuck on walls of the grid. Sokoban is PSPACE-hard [21]. We use the Boxoban training and test problems [41].

Traveling Salesman Problem (TSP) A 10 × 10 grid-based environment where the agent must visit all of the cities before returning back to the first city it visited. The TSP is NP-hard [34].

[1] https://github.com/jacobandreas/psketch/tree/master

6.5.2 Algorithms Evaluated

Algorithms with π^{SG}

We evaluate our subgoal-guided policy π^{SG} on both PHS* and LevinTS, which we denote PHS*(π^{SG}) and LevinTS(π^{SG}), respectively. Both PHS* and LevinTS were shown to perform well in single-agent deterministic search problems [79].

Baselines

To evaluate the training expansion efficiency provided by our subgoal-guided policy and data generation method from prematurely terminated trees, we compare PHS*(π^{SG}) to PHS* and LevinTS(π^{SG}) to LevinTS using their original single policy formulation, denoted PHS*(π) and LevinTS(π), respectively. Following Orseau and Lelis [79], we also evaluate our policy formulation against Weighted A* [83] with $w = 1.5$, which was reported to require fewer training expansions than PHS* in some environments. We also test our method against HIPS-ε, which is another complete subgoal-guided tree search method.

6.5.3 Experimentation Procedure

For evaluating the training expansion efficiency of different methods to learn an effective policy, we use the setup of Orseau and Lelis [79]. A loss of $\ell(\cdot) = 1$ is used, which corresponds to the search loss $L(S, n_k^*)$ being the number of node expansions that an algorithm S requires to solve problem instance k. PHS*(π), LevinTS(π), and WA* are trained using the Bootstrap process (see Section 2.3.3). PHS*(π^{SG}) and LevinTS(π^{SG}) are trained using the process outlined in Section 6.4. The Bootstrap process starts with a budget of 4,000 expansions and uses the budgeting update schedule noted in Section 2.3.3. After each training sweep, performance is evaluated on the separate validation set; training ends once at least 95% of validation problems are solved. During the online training Bootstrap process, we record the number of outstanding training problems which have yet to be solved, and the total accumulated loss (expansions) incurred after every iteration. For the hard-domain training

experiments, we also record wall-clock time: elapsed real time, which is not summed across worker threads.

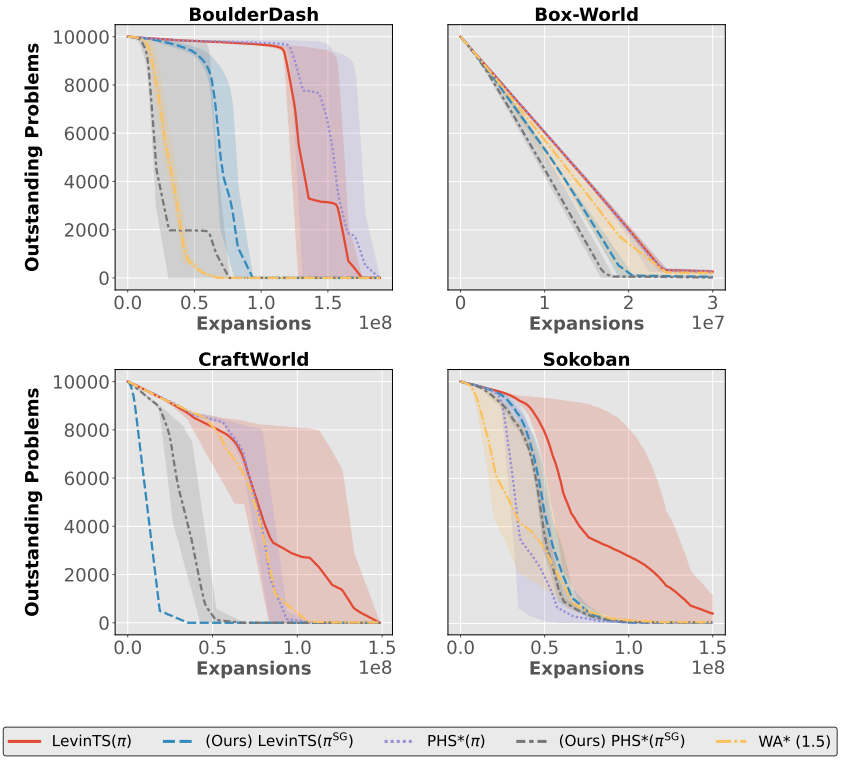
The training procedure outlined for HIPS-ε requires a precomputed dataset of solution trajectories. Since it is not clear how to train HIPS-ε online using the Bootstrap process, we do not include HIPS-ε in our training efficiency results. The policy, heuristic, and subgoal generator models used in HIPS-ε are trained following the procedure of Kujanpää et al. [58]. For Sokoban, we use the offline datasets provided by Kujanpää et al. [58]. For Box-World, CraftWorld, and BoulderDash, we use trajectories found by domain-specific solvers over the same training set used to train the other systems with the Bootstrap process.

Every domain has a disjoint set of 10,000 problem instances to train, 1,000 as validation, and 100 in the test set. Each algorithm is trained with five separate seeds, which determine the model initialization, the train and validate problem instance partitions, and how the problems are shuffled when batched. For the training efficiency experiment, we report the maximum, minimum, and average number of outstanding problems in our plots of results. These five-seed ranges are descriptive and are not confidence intervals or formal uncertainty estimates. For testing, we record the number of problem instances each algorithm was able to solve, the average number of expansions required to solve those instances, and the average solution length. Following the evaluation procedure of Orseau and Lelis [79], the test tables report results from a single selected seed rather than an average across seeds. Among the five seeds, we select the one that solves the most test instances, with ties broken first by fewer node expansions and then by shorter solution lengths. Under the fixed test budget, solved count is the primary outcome. Reported expansions and solution lengths are conditional means over solved instances; when methods solve different numbers of instances, these values describe efficiency among successful runs rather than an overall efficiency ranking. Due to the large runtime costs of HIPS-ε, we limit the budget during testing to 8,000.

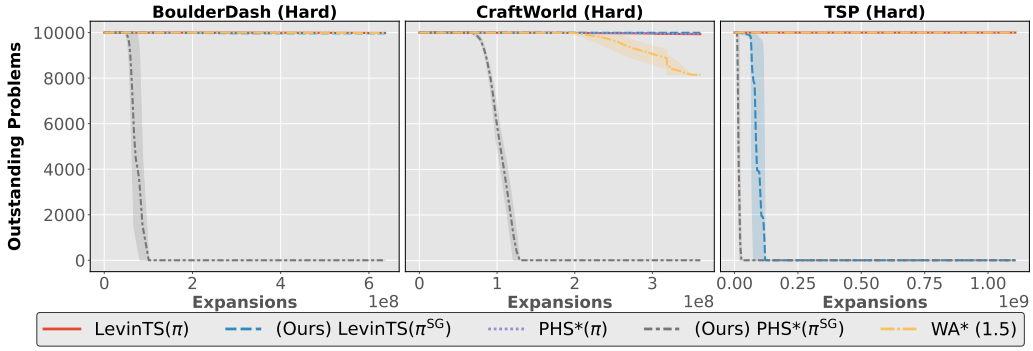
6.5.4 Training Efficiency Results

The first set of training loss experiments focuses on domains that all the baseline methods can solve. The total search loss incurred over the course of training, as measured by the number of node expansions, is shown in Figure 6.7.a. In the Box-World, CraftWorld, and BoulderDash environments, the subgoal-guided policy formulation PHS$^*(\pi^{\text{SG}})$ and LevinTS(π^{SG}) require substantially fewer node expansions than PHS$^*(\pi)$ and LevinTS(π), respectively, to solve all 10,000 training instances. Sokoban is the only environment where PHS$^*(\pi)$ initially can solve more problems using fewer expansions, but both PHS$^*(\pi)$ and PHS$^*(\pi^{\text{SG}})$ converge towards the end of training when only the harder problems remain to be solved. We hypothesize that our method performs worse in Sokoban than in the other problems due to the training data that the clustering algorithm generates. While clustering finds important structures in the other problems, it fails to find helpful structures in Sokoban.

The next set of experiments is similar to the above, but instead showcases whether the training expansion efficiency of our method enables policies to be learned in more challenging problems. For the domain environments with access to level-generators, problem instances were generated to be sufficiently hard, such as adding more diamonds for the agent to collect in BoulderDash. All methods were given a maximum wall-clock time budget of 16 hours for the online bootstrap training process, with the training loss curves given in Figure 6.7.b and summarized in Table 6.1. The baseline algorithms were not able to solve many of the training instances within the time budget, and incurred a large training loss. PHS$^*(\pi^{\text{SG}})$ completed the training process within the allocated wall-clock time on the evaluated hard domains, solving 9,985 of 10,000 CraftWorld instances and all BoulderDash and TSP instances. Despite our method requiring a higher computational cost per node expansion due to the use of several networks, its training expansion efficiency yields a substantially smaller reported cost in node expansions and wall-clock time under this protocol.



(a) Standard problem instances.



(b) Hard problem instances.

Figure 6.7: Training curves for Subgoal-Guided Policy Heuristic Search and baselines. The line is the average number of outstanding problems for the respective number of expansions accumulated during training. Shaded regions show the minimum and maximum outstanding problems across five seeds; they are descriptive ranges, not confidence intervals.

Table 6.1: Training efficiency results on the hard environments. Values are means across five seed-specific training runs. "Expansions" represents the sum over all problems, wall-clock time is measured in hours, and the maximum value for "Solved" is 10,000.

Algorithm	Expansions	Time	Solved
CraftWorld (Hard)			
WA* (1.5)	350,169,632	16.01	1,850
LevinTS(π)	360,454,214	16.02	73
PHS*(π)	324,718,860	16.04	54
LevinTS(π^{SG})	172,188,794	16.02	9
PHS*(π^{SG})	**125,089,420**	**13.74**	**9,985**
BoulderDash (Hard)			
WA* (1.5)	599,818,595	16.01	18
LevinTS(π)	636,718,638	16.04	16
PHS*(π)	591,135,427	16.02	19
LevinTS(π^{SG})	284,044,919	16.02	31
PHS*(π^{SG})	**85,470,564**	**6.25**	**10,000**
TSP (Hard)			
WA* (1.5)	1,019,840,000	16.01	0
LevinTS(π)	1,107,904,000	16.02	0
PHS*(π)	980,928,000	16.02	0
LevinTS(π^{SG})	105,838,613	4.55	**10,000**
PHS*(π^{SG})	**20,593,798**	**1.23**	**10,000**

The results on the easier and more difficult domains support the hypothesis that our subgoal-guided policies can improve Bootstrap-training expansion efficiency under the reported protocol.

6.5.5 Test Results

To test the quality of the learned policies, Table 6.2 reports solved count together with conditional mean expansions and solution lengths on the standard problems. Among the methods that solve all standard test problems, PHS*(π^{SG}) has the lowest conditional mean expansion count on all domains except Sokoban. WA* is able to find solutions that are the closest on average to the optimal cost, but it requires considerably more node expansions. With the exception of BoulderDash, LevinTS(π^{SG}) and PHS*(π^{SG}) have lower conditional mean expansion counts than LevinTS(π) and PHS*(π), respectively, among methods that solve all standard test problems. Taken together, Table 6.2 and Figure 6.7 show that learning π^{SG} can require fewer Bootstrap-training expansions than the non-subgoal-based policies in the reported comparisons, while solving all standard test problems within budget (Table 6.2).

HIPS-ε was unable to solve any problem in BoulderDash, and the majority of problems in CraftWorld within budget constraints. HIPS-ε performs poorly in these reported domains under its separate, offline training regime. Its reliance on subgoal-level search makes reconstruction quality a plausible contributing factor in domains with complex observations, such as BoulderDash and CraftWorld, but this comparison does not isolate that mechanism. In Sokoban, HIPS-ε solved all test instances under the stated budget. Note that expansions reported by HIPS-ε are not directly comparable to the other algorithms, as it only counts the expansions performed in the learned subgoal space, which does not include the costs such as grounding and verifying the actual paths in the original state space. Since PHS*(π^{SG}) and LevinTS(π^{SG}) perform the search in the original space, they are more forgiving with reconstruction errors from the subgoal generator.

Table 6.3 reports similar metrics for each algorithm on the test set of the hard problems. Each problem instance was given a maximum budget

Table 6.2: Results on the test set for each of the standard domain environments. Results are reported for the selected seed described in Section 6.5.3. Solved count is the primary outcome; expansions and solution length are conditional means over solved problems and are not overall efficiency comparisons when solved counts differ. (†) Expansions reported for HIPS-ε are only for the latent-level space, and thus are not directly comparable to the other algorithms.

Algorithm	Solved	Expansions	Length
Box-World (Standard)			
WA* (1.5)	100	2,398.68	**66.19**
LevinTS(π)	100	605.05	67.91
PHS*(π)	100	767.06	68.07
HIPS-ε	100	467.79(†)	67.39
LevinTS(π^{SG})	100	411.38	66.73
PHS*(π^{SG})	100	**378.58**	66.75
CraftWorld (Standard)			
WA* (1.5)	100	2,318.22	**90.01**
LevinTS(π)	100	262.76	93.93
PHS*(π)	100	172.04	93.49
HIPS-ε	19	116.11(†)	92.47
LevinTS(π^{SG})	100	208.28	93.93
PHS*(π^{SG})	100	**103.23**	94.54
BoulderDash (Standard)			
WA* (1.5)	100	1,193.60	**51.44**
LevinTS(π)	100	61.33	52.90
PHS*(π)	100	53.65	52.74
HIPS-ε	0	—	—
LevinTS(π^{SG})	100	65.48	53.30
PHS*(π^{SG})	100	**53.34**	52.68
Sokoban (Standard)			
WA* (1.5)	100	1,091.45	**32.81**
LevinTS(π)	100	1,177.26	41.04
PHS*(π)	100	1,523.38	39.40
HIPS-ε	100	**80.55**(†)	45.24
LevinTS(π^{SG})	100	496.87	41.85
PHS*(π^{SG})	100	808.42	39.61

Table 6.3: Results on the test set for each of the hard domain environments. Results are reported for the selected seed described in Section 6.5.3. Solved count is the primary outcome; expansions and solution length are conditional means over solved problems and are not overall efficiency comparisons when solved counts differ.

Algorithm	Solved	Expansions	Length
CraftWorld (Hard)			
WA* (1.5)	100	313,572.52	**117.03**
LevinTS(π)	0	—	—
PHS*(π)	0	—	—
LevinTS(π^{SG})	100	395,557.94	123.29
PHS*(π^{SG})	100	**3,071.83**	120.77
BoulderDash (Hard)			
WA* (1.5)	16	271,636.94	58.19
LevinTS(π)	0	—	—
PHS*(π)	0	—	—
LevinTS(π^{SG})	22	315,807.91	69.5
PHS*(π^{SG})	100	**172.32**	**84.5**
TSP (Hard)			
WA* (1.5)	1	502,624.0	45.0
LevinTS(π)	0	—	—
PHS*(π)	0	—	—
LevinTS(π^{SG})	100	570.09	**40.89**
PHS*(π^{SG})	100	**41.93**	41.73

of 512,000 node expansions. PHS*(π^{SG}) was able to solve all test problems, while the other comparison methods struggled to solve a majority of them. The exception is WA* on CraftWorld, where both methods solve all test problems but WA* requires over 100 times more node expansions on average than PHS*(π^{SG}). Due to the complexity of the problem instances, HIPS-ε is omitted from these results as it requires a dataset of solution trajectories to train its policies and subgoal generator.

6.5.6 Search Loss Efficiency from Non-Solution Trees

One of the main contributions of our method for learning policies is that it can use data from failed searches to train its low-level policy and subgoal generator. While the previous experiment evaluated the training efficiency that

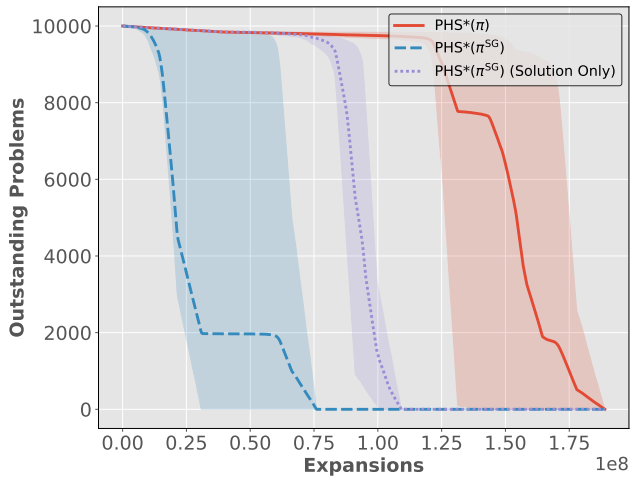


Figure 6.8: Reduction in search loss during training when using data from non-solution trees. The line represents the average and the shaded region spans the minimum and maximum outstanding problems across five seeds; it is a descriptive range, not a confidence interval.

the combination of subgoal-guided policy and the training method provides, this experiment aims to analyze the contribution of each. In Figure 6.8, we show the search loss during training for PHS*(π^{SG}) in the BoulderDash environment when we limit it to only using solution trajectories to update the models. On BoulderDash, these results show that learning π^{SG} from solution trajectories improves the reported training expansion efficiency of PHS*. We see another improvement when the system learns π^{SG} from both successful and failed searches.

6.5.7 Ablation: Impact of Codebook Size

The size of the codebook in the VQVAE determines the number of subgoals parameterized by the model. In Figure 6.9, we show the search loss during training for various codebook sizes on the CraftWorld environment. The chosen codebook size of $k = 4$ incurs the smallest search loss, but we note that there is a fair amount of overlap between the curves. Table 6.4 shows how each of these models performs on the test set. While the solution length is relatively the same, there is quite a large difference in the number of expansions required to solve the problems. If too few subgoals are used, then the

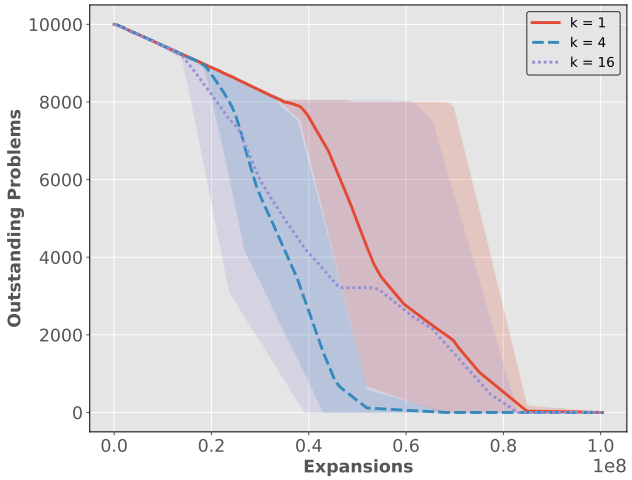


Figure 6.9: Learning curves during training using the Bootstrap process for varying codebook sizes. The line is the average number of outstanding problems for the respective number of expansions accumulated during training. All training runs lie within the shaded regions.

codebook becomes overburdened in trying to represent a single subgoal state. Interestingly, if too many subgoals are used, we also see an increase in the number of expansions required. One possible reason for this could be that if a large number of codebook entries are not being used, then the resulting mixture policy will look more like a uniform policy, which will increase the number of node expansions required. One avenue for future work is to dynamically remove unused codebook entries from being used in the final policy. We note that this CraftWorld-only ablation is diagnostic and does not establish that the same codebook-size behaviour generalizes to other domains.

Table 6.4: Results on CraftWorld test set using various codebook sizes. Expansions and solution length are averaged over solved problems.

Algorithm	Solved	Expansions	Length
$\kappa = 1$	100	263.92	92.52
$\kappa = 4$	100	103.23	94.54
$\kappa = 16$	100	305.17	94.16

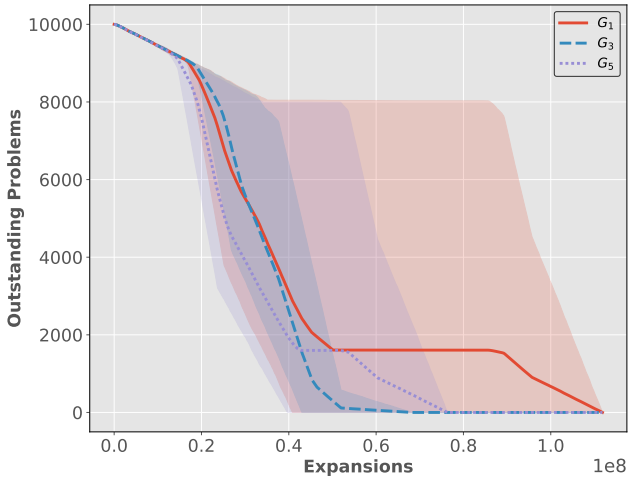


Figure 6.10: Learning curves during training using the Bootstrap process for varying cluster graph levels from which subgoal states were sampled from. The line is the average number of outstanding problems for the respective number of expansions accumulated during training. All training runs lie within the shaded regions.

6.5.8 Ablation: Impact of Cluster Graph Level when Sampling Subgoals

From the base graph G_0, the Louvain algorithm produces a hierarchy of cluster graphs $\{G_1, \ldots, G_N\}$ where earlier graphs contain many smaller clusters with later graphs merging them into fewer clusters. One decision to make when sampling subgoals is which of these cluster graphs in the hierarchy to choose from. Smaller abstraction levels will generally result in pairs of states being sampled which are close in space/time, and larger abstraction levels will generally result in pairs of states further apart, which is depicted in Table 6.5.

Figure 6.10 shows the search loss during training when sampling subgoals from various cluster graph levels, and Table 6.6 shows the test results on the CraftWorld environment. This CraftWorld-only ablation is diagnostic and does not establish the same cluster-level tradeoff in other domains. From the search loss during training and at test time, it appears that one must be careful not to sample subgoals which are too close together from G_1, as the subgoals will not be as meaningful. Sampling subgoals from G_5 which were far apart

as compared to G_3 did lead to a slight loss in performance, both in terms of number of node expansions and solution length.

Table 6.5: Average trajectory length between subgoals for each cluster level on the CraftWorld environment.

Cluster Graph Level	Subgoal Distance
1	1.66
3	4.93
5	17.29

Table 6.6: Results on CraftWorld test set using models which were trained using various cluster graph levels. Expansions and solution length are averaged over solved problems.

Algorithm	Solved	Expansions	Length
G1	100.00	287.24	93.88
G3	100.00	103.23	94.54
G5	100.00	110.10	95.05

6.6 Conclusion

In this chapter, we presented a novel way to combine subgoal discovery with subgoal-guided policies, which can be used as a drop-in replacement for policies used in existing policy tree search algorithms, such as LevinTS and PHS*. The neural network models in our method can be trained online without needing an offline solution dataset. Moreover, they can be trained from both successful and failed searches, which we showed to improve the training expansion efficiency of tree search algorithms substantially. This is due to the fact that a subgoal-conditioned policy can be trained using non-solution trajectories from the search tree. In the domains evaluated, our experiments showed that the large reduction in environment interactions does not reduce the quality of the solutions found, while requiring fewer expansions at test time. As we increased the problem's difficulty, the gap between our subgoal-based approach and other approaches increased—while our learned policies solved all test problems, previous approaches failed to learn effective ones. The following chapter considers

a complementary form of structural search control that seeks to exploit state-space structure through rerooting without explicit subgoal generation.

Chapter 7

Structure-Induced Information for $\sqrt{\text{LTS}}$

This chapter is joint work with Michael Buro, Levi Lelis, and Laurent Orseau. It was previously published at the Forty-third International Conference on Machine Learning [104]. This chapter introduces an automated way to learn re-rooters for $\sqrt{\text{LTS}}$. Readers are guided towards Section 3.4.3 for a complete overview of $\sqrt{\text{LTS}}$.

7.1 Introduction

Complex discrete planning problems remain difficult to solve at scale, driving increasing interest in learning-guided search methods. Levin Tree Search (LTS) [78], a tree search algorithm that uses a learned policy to guide the search, has shown success in addressing these types of problems. A key property of LTS is that it provides an upper bound on the number of search steps required before finding a solution, which depends on the *quality* of the policy. This, in turn, enables policies to be learned with the explicit objective of minimizing search effort. Policy-Guided Heuristic Search (PHS*) [79] extends LTS by combining a learned policy with a learned heuristic function. Orseau and Lelis [79] provide a similar bound for PHS*, and also show that a policy can be learned while minimizing this bound.

Although LTS and PHS* are theoretically well-founded, they rely primarily on the learned policy and heuristic functions and, without additional struc-

tural guidance, can struggle to solve complex problems. A common approach when scaling to complex problem domains, inspired by how humans plan [13, 19, 24], is to decompose the problem into easier subtasks and subgoals. Subgoals help address this limitation by structuring the search and extending its reach beyond what the initial policy can support. Building on this insight, prior work has introduced subgoal-based policy tree search methods, including HIPS-ε [58] and Subgoal-Guided Policy Heuristic Search (SGPS, Chapter 6) [103], which generate intermediate target states and condition low-level policies on these generated subgoals to guide the search. By explicitly reasoning over such subgoals, these approaches can improve early-stage exploration and learning efficiency. However, they also introduce additional modelling complexity and computational overhead, as search performance becomes tightly coupled to the quality of subgoal reconstruction and the policies conditioned on them. As we will show, this issue becomes increasingly pronounced as the domain complexity increases. Chapter 6 addressed this challenge through explicit learned subgoals; this chapter considers the complementary approach of exploiting structure through rerooting without generating subgoal states.

$\sqrt{\text{LTS}}$ [77] (Section 3.4.3), read as "root-LTS", is a policy tree search algorithm that implicitly starts an LTS search at each node in the search tree. The overall search effort is split between each of these searches, and the proportion of time allocated to each is given through a *rerooter*. This mechanism implicitly decomposes the search into subtasks, foregoing the complexity of the subgoal modelling of HIPS-ε and SGPS that makes those methods costly when scaling to complex domains. Orseau et al. [77] showed that the bound on the number of node expansions before $\sqrt{\text{LTS}}$ finds a solution can be exponentially better than that of LTS. In the $\sqrt{\text{LTS}}$ literature, the term "visit" is often used for the event of selecting a node for expansion. In this thesis, "node expansion" is the default term; for $\sqrt{\text{LTS}}$, a search step and a node visit are equivalent to a node expansion unless otherwise stated.

In this chapter, we revisit rerooting as a general mechanism for exploiting structure in the underlying state space in policy tree search and study how to derive rerooting weights in practice to guide search effectively. Within this

framework, we present three instantiations. The first two capture complementary structural information: a global approach that induces structure via state-space clusters identified using Leiden clustering [101], and a lightweight local approach that derives structure from learned heuristic cost information. We then show how an additive rerooter can take advantage of the strengths each rerooter provides, and instantiate this as a hybrid rerooter of the previous two. In contrast to the subgoal baseline methods, which depend on computationally expensive subgoal generation using high-capacity models [103], our approach does not require learning or invoking separate subgoal networks. Instead, we instantiate rerooters from structure already present in the search tree, with the clustering rerooter running on demand during search. Empirical results show that our rerooters substantially improve online training sample efficiency over non-rerooting baselines. These results establish rerooting as a scalable approach for exploiting structure in search, while avoiding explicit subgoal generation of previous work.

Our contributions can be summarized as follows. We provide automated methods for constructing rerooters from the structure of search trees alone and show that even lightweight structural signals can achieve strong performance. Finally, we provide a novel theoretical guarantee on the number of node expansions until the first solution node is found by $\sqrt{\text{LTS}}$ using additive rerooters, which highlights how $\sqrt{\text{LTS}}$ can take advantage of the *synergies* of multiple rerooters. The experiments show that, on the domains and under the training protocol studied, our method can scale to complex environments during online training and achieve strong training efficiency relative to the Bootstrap-based methods evaluated, whereas previous methods that rely on subgoal reconstruction fall short.

7.2 Related Work

Subgoal Search Prior work has represented these through fixed-length subtasks [22], multi-model approaches that represent different subtask lengths by training a separate fixed-horizon model for each length [110], and perform-

ing a high-level search in subgoal-space using a subgoal generator model for varying-length subgoals [57]. While these approaches show promising results, they all suffer from a lack of completeness, *i.e.*, none are guaranteed to find a solution even if one exists. HIPS-ε [58] added completeness guarantees by augmenting the search space to include both subgoals and atomic actions. Subgoal-Guided Policy Heuristic Search (SGPS, Chapter 6, [103]) generates subgoals like HIPS-ε but performs search over atomic actions. The generated subgoals are used to condition low-level policies, which are mixed with a high-level subgoal policy to produce a single policy usable by LTS or PHS*. A novelty of SGPS is that it can be trained online using the data from the search tree, even when a budget for the search causes early termination without finding a solution. Both HIPS-ε and SGPS are reliant on generated subgoals, and the performance of the search becomes tightly coupled to the quality of subgoal reconstruction and the policies conditioned on them. As shown in our experiments, this becomes an issue when scaling to complex domains.

State-Space Structure for Search Control A complementary line of work leverages structural regularities in the state space, often via partitions or region abstractions, to guide how search allocates effort. Cartesian Counterexample Guided Abstraction Refinement [17, 91] guides search in cost-optimal planning through iteratively refining abstractions. Recent work improves refinement strategies [95] and ways of combining multiple abstractions for stronger heuristics [89]. In the options framework [98], methods exploit connectivity and bottleneck structure to create temporally extended actions that reduce planning complexity. This includes using entropy to discover skills that capture key transition patterns in the state-space [111, 112], and state-space clustering [1, 86]. In contrast, we exploit structure only to steer the search effort through rerooting, without constructing abstractions for heuristic computation or learning temporally extended actions.

7.3 Preliminaries

We follow the same definition of a single-agent deterministic search problem defined in Section 2.1. Readers are pointed towards Section 3.4.1 for the background on LevinTS, and Section 3.4.3 for the background on $\sqrt{\text{LTS}}$.

One notational convention is important for this chapter. In Section 3.4.3, the main presentation of $\sqrt{\text{LTS}}$ uses the rooted Levin cost $\frac{d}{\pi}(n_t \prec n)$, since this form makes the search-effort analysis easier to follow. The original $\sqrt{\text{LTS}}$ formulation of Orseau et al. [77], and the algorithms in this chapter, instead use the rooted slenderness cost introduced in Definitions 3.4.15 and 3.4.16. Thus, unless stated otherwise, $\sqrt{\text{LTS}}$ in this chapter refers to the slenderness-cost variant with

$$\varphi_{\sqrt{\text{LTS}}}(n) = \begin{cases} 0, & n = n_1, \\ \min\limits_{n_t \prec n} \dfrac{1}{w_t} c_t^r(n), & n \neq n_1, \end{cases} \tag{7.1}$$

where

$$c_t^r(n) = \begin{cases} \frac{\lambda}{\pi}(n \mid n_t) - 1 & \text{if } n_t \prec n, \\ \infty & \text{otherwise.} \end{cases} \tag{7.2}$$

As in Section 3.4.3, components with zero rerooting weight or zero suffix probability are treated as having infinite cost. This changes only the rooted local cost used inside the $\sqrt{\text{LTS}}$ priority rule; the rerooting weights and the interpretation of $\sqrt{\text{LTS}}$ as sharing computation among local searches remain the same. The $\frac{d}{\pi}$-based bounds stated in Section 3.4.3 have analogous slenderness-cost versions obtained by replacing $\frac{d}{\pi}(n_t \prec n)$ with $c_t^r(n)$, and this is the convention used by the results in this chapter.

This chapter extends the slenderness-cost analogue of the $\sqrt{\text{LTS}}$ search bound from Equation 3.28 when using *additive* rerooters. The Bootstrap algorithm [7] is used throughout this chapter to learn a neural policy and/or heuristic functions from search. See Section 2.3.3, and its pseudocode in Algorithm 2.

7.3.1 Problem Definition

Our objective formulation is the same as given in Chapter 6, Orseau and Lelis [79], and Tuero et al. [103], which is to solve a set K of problem instances as *quickly* as possible. We use the *total search loss* as our objective metric. The *search loss* $L(S, n)$ of algorithm S is the sum of losses $\ell(n')$ incurred for each node n' expanded up to and including n. Here, n could either be a solution node in $\mathcal{N}_g$ or the last node expanded before a given budget is exceeded. The *total search loss* $\sum_{k \in K} L(S, n_k)$ is the sum of individual search losses algorithm S incurs on problem k. We assume $\ell(n) = 1$ everywhere, thus minimizing the search loss corresponds to minimizing the total number of expansions.

7.4 Structure-Induced Rerooters

In this section, we describe how rerooting can be instantiated in practice to exploit structural information during policy tree search. Orseau et al. [77] focused on analyzing the formal properties of $\sqrt{\text{LTS}}$ for a *given* rerooter, and left open the question of how to define or learn the rerooter. We address this gap by presenting two complementary rerooters that derive rerooting weights from signals available during search: a clustering-based rerooter that captures global structure in the state space, and a heuristic-based rerooter that leverages local cost-to-go information. Together, these designs illustrate how rerooting can be automatically induced or learned, as appropriate, without the complexities of explicit subgoal reconstruction, and provide a practical foundation for scalable rerooting in complex domains.

7.4.1 Global Structure–Induced Rerooting

Motivation

Progress in many planning problems is characterized by transitions between distinct regions of the state space. When search reaches a new region, it is often beneficial to concentrate effort there; when it keeps expanding within the same region without making progress, effort should be redirected elsewhere in the tree. Such transitions may correspond to events like entering a new room

or acquiring a key that unlocks additional states. To capture this behaviour, the global rerooter derives rerooting weights from coarse state-space structure by clustering states based on connectivity. It then allocates effort across clusters rather than individual nodes, reflecting global organization while relying only on state observations and a Boolean goal test, without explicit subgoal reconstruction during search.

Global Structure

A key part of our method is the Leiden clustering algorithm [101], which is an improvement over the Louvain clustering algorithm [11], both in terms of runtime complexity and cluster quality. The Leiden algorithm is an iterative algorithm that creates a hierarchy of graphs $(G_1, \ldots, G_N)$ where G_{i+1} is a clustering of graph G_i. In each iteration i, a hierarchical cluster graph $G_{i+1} = (V_{i+1}, E_{i+1})$ is created from $G_i = (V_i, E_i)$, where V_{i+1} is a partition of V_i with each $v \in V_{i+1}$ being a part in the partition, and E_{i+1} containing edges (v_{i+1}, v'_{i+1}) for $v_{i+1}, v'_{i+1} \in V_{i+1}$ if there is a $u \in v_{i+1}$ and $v \in v'_{i+1}$ such that $(u, v) \in E_i$. The process continues until either there is no progress made (i.e., $G_{i+1} = G_i$) or G_{i+1} contains a single node.

Our first rerooter, which we denote $\sqrt{\text{LTS}}$-L, uses the Leiden clustering algorithm to find structures in the induced subgraph of the state space. During the search, the induced subgraph of the underlying state space is constructed iteratively whenever a tree node is generated. Each graph node represents a unique environment state; a newly generated unseen state adds a graph node and an edge from its parent state, while a generated transposition adds only the corresponding graph edge. The search uses simple state-existence pruning to avoid duplicate search representatives, but not the cost-sensitive state-pruning results of Chapter 3, since rerooting weights can depend on discovery time, clustering updates, and colouring history rather than on the underlying state alone.

The Leiden algorithm creates cluster graphs at increasing levels of the hierarchy by maximizing the *modularity*, a metric that quantifies the relation between the edge connectivity within a cluster and the connectivity between clus-

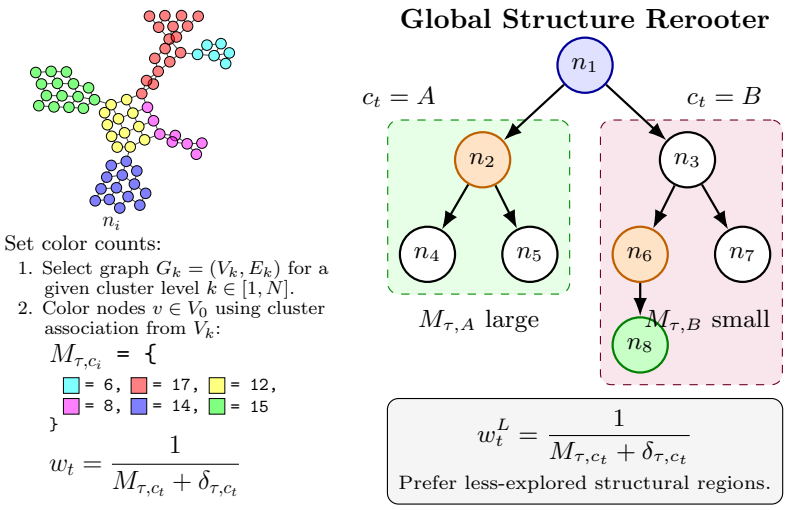


Figure 7.1: $\sqrt{\text{LTS}}$-L Rerooter. Each node in the tree is assigned a colour corresponding to the cluster it is associated with in the cluster graph at level k of the hierarchy. The rerooting weight for the expanded node is then computed using the colour count map.

ters. High modularity results in dense clusters with many connections within each cluster, and sparse connections between clusters. Evans and Şimşek [28] found that the Louvain algorithm using this modularity metric finds meaningful structures in the state spaces they studied, such as two neighbouring clusters corresponding to states of two adjacent rooms in a gridworld environment.

Rerooting Weight

The $\sqrt{\text{LTS}}$-L rerooter is visually depicted in Figure 7.1. We invoke the Leiden algorithm on the incrementally induced subgraph G_0 using the geometric schedule $r_1 = 1$ and $r_{j+1} = \lceil \gamma r_j \rceil$, with hyperparameter $\gamma > 1$. From the resulting hierarchy, we select a cluster graph $G_k = (V_k, E_k)$ at hierarchy level k (a hyperparameter), where each node $U \in V_k$ corresponds to a set of nodes in G_0 (and thus to a set of search-tree nodes). We assign each tree node n a colour $c \in \{1, \ldots, |V_k|\}$ indicating the node U that contains it; we refer to this assignment as a *colouring*. Colours may change across invocations: a node n can have colour i on the τ-th invocation, and colour $j \neq i$ on the $(\tau + 1)$-th invocation.

Let τ denote the index of the most recent clustering invocation, whose schedule step is $r_\tau \leq t$. For the search steps between r_τ and t, some nodes may be expanded which do not have a colour, as they were generated after the most recent clustering invocation. For these nodes, we approximate their cluster membership by assigning their parent's colour. Using a proxy value to defer expensive computations has been used previously in heuristic search, such as when computing the edge cost [69] or the heuristic value [48] is costly.

Let $M_{\tau,c}$ be the number of unique generated environment states with colour c from the (τ)-th colouring, and

$$\delta_{\tau,c} = |\{r_\tau < \ell \leq t : c_\ell = c\}| \tag{7.3}$$

be the count of tree nodes expanded since the τ-th invocation that have colour c. Then, $\sqrt{\text{LTS}}$-L assigns the rerooting weight w_t for the node n_t expanded at search step t as

$$w_t = \frac{1}{M_{\tau,c_t} + \delta_{\tau,c_t}} \tag{7.4}$$

where c_t is the colour associated with n_t. The value of $M_{\tau,c_t}+\delta_{\tau,c_t}$ approximates the size of the current cluster size. It is approximate because parent-colour inheritance is used for nodes generated after the latest clustering update, and because colours and cluster identities are recomputed rather than matched across updates. Counts are reset at each clustering invocation, while weights already assigned to expanded nodes remain fixed.

This gives us the desirable property that a low colour count results in a larger weight w_t and lower search cost. Similarly, a higher colour count results in a smaller weight w_t, and higher cost. As the search expands more nodes with colour c, the count increases and thus subsequent nodes expanded from the same cluster receive lower weights and higher costs. See Algorithm 12 for its pseudocode.

Runtime Complexity

While the induced subgraph of the state space is built incrementally, Leiden cluster graphs are recomputed on the most recent graph under a geometric update schedule to limit runtime overhead. Although cluster assignments (and

sizes) may change across updates, we keep the rerooting weights of previously expanded nodes as fixed: retroactively updating them would make the search tree internally inconsistent, since an ancestor could induce different costs for different descendants depending on when they were generated. Theorem 7.4.1 shows that geometric clustering contributes an amortized near-linear term under its stated implementation assumptions, while naïve evaluation of ancestor-based priorities contributes $O(DN)$ over N expansions at maximum depth D. This is not a constant-factor overhead relative to conventional best-first search. The original $\sqrt{\text{LTS}}$ implementation can often maintain ancestor information more efficiently, but this worst-case analysis does not assume such an optimization. Unlike prior subgoal-based approaches, it avoids queries to compute-intensive models such as subgoal generators [103].

Theorem 7.4.1. *Let N be the number of node expansions performed by $\sqrt{\text{LTS}}$-L, and let D be the maximum depth of any generated node after these expansions. Assume that each expanded node generates at most $b < \infty$ children, that successor generation, policy lookup, colour lookup, and colour count updates take $O(1)$ time per generated node, and that the priority key of a generated node can be computed from stored ancestor information in $O(d(n))$ time. Assume, as an implementation-level runtime model consistent with the Leiden analysis of Traag et al. [101], that a Leiden call on a graph $G = (V, E)$ truncated to a fixed hierarchy level k takes $O(k|E|)$ time. If Leiden clustering is invoked according to a geometric schedule with fixed factor $\gamma > 1$, then the runtime of $\sqrt{\text{LTS}}$-L over N expansions is*

$$O\left(bN\log(bN) + bDN + kbN\frac{\gamma}{\gamma-1}\right).$$

In particular, for fixed b, k, and γ, this is

$$O(N\log N + DN).$$

If k grows with the graph hierarchy depth, the clustering term retains its explicit factor k.

The proof is given in Appendix F.1. The uniform branching-factor assumption is standard in tree-search–based planning and control, where each state has a bounded number of successor states determined by the action set.

7.4.2 Local Structure–Induced Rerooting

Motivation

Global structural cues are not always needed for effective rerooting: local information at individual nodes can distinguish between otherwise similar regions of the search tree. For example, if the root has two structurally identical subtrees, a rerooter based solely on structural expansion history would allocate equal effort to both; yet if one subtree is estimated to be closer to the goal, it should be prioritized. We therefore consider a local rerooter that leverages heuristic cost-to-go estimates to assign weights reflecting the local geometry of the search space.

Rerooting Weight

The rerooter, which we denote by $\sqrt{\text{LTS}}$-H, uses the heuristic value of a node as a goal-proximity signal to define its rerooting weight directly; it is not by itself a certified measure of progress. The rerooting weight is given by

$$w_1 = 1; \quad w_t = \exp\left(-\alpha \frac{h(n_t)}{H_1}\right), \quad H_1 = \max\{1, h(n_1)\}, \tag{7.5}$$

where $\alpha > 0$ controls how strongly rerooting effort is concentrated on nodes with low heuristic value, and $h(n_t)$ is the heuristic value corresponding to node n_t. The heuristic head uses a ReLU output transformation, so $h(n) \geq 0$. The root normalization H_1 avoids a degenerate denominator when the root heuristic is zero or small.

This choice is natural for several reasons. First, it is monotone in the heuristic: if $h(n_i) < h(n_j)$, then $w_i > w_j$. Thus, nodes estimated to be closer to the goal receive a larger rerooting weight and therefore induce a lower rerooted search cost. Normalizing by H_1 makes the rerooting weight depend on heuristic values relative to the root scale. When $h(n_1) \geq 1$, this is exactly a normalization by the root heuristic. This is useful when different environments,

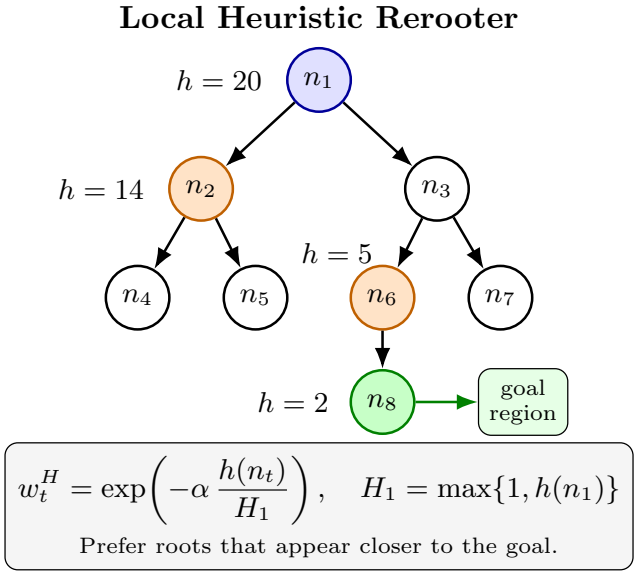


Figure 7.2: $\sqrt{\text{LTS}}$-H Rerooter. The heuristic value is used as a proxy for progress being made. The rerooting weight for the expanded node is then computed using a smoothed normalization where low heuristic values correspond to high rerooting weights.

or different instances within the same environment, exhibit substantially different effective solution-length scales. Second, the exponential map is smooth and strictly positive. Even nodes with relatively poor heuristic values retain non-zero weight, which is desirable when the heuristic is imperfect and should influence, rather than fully determine, the rerooting decision. Third, the parameter α acts as an inverse-temperature parameter. When α is small, the weights are more similar across nodes, so the rerooter behaves conservatively. As α increases, the weight mass becomes more concentrated on nodes whose heuristic values are small relative to the root, causing rerooting to focus more aggressively on regions estimated to be closer to a goal.

A useful consequence is approximate invariance to multiplicative rescaling when the root-normalization floor is inactive before and after rescaling. The exponential form also gives the rerooter a useful normalized interpretation. For non-root candidate rerooting points, the weights are proportional to

$$\exp\left(-\alpha \frac{h(n_i)}{H_1}\right).$$

Therefore, after normalizing over any finite set I of non-root candidate rerooting points,

$$\frac{w_t}{\sum_{i\in I} w_i} = \frac{\exp(-\alpha\, h(n_t)/H_1)}{\sum_{i\in I} \exp(-\alpha\, h(n_i)/H_1)}.$$

Thus, after this explicit finite-set normalization, the weights have a softmax form over negative normalized heuristic values. $\sqrt{\text{LTS}}$ itself uses unnormalized cumulative rerooting weights over nodes discovered through time, so this finite-set interpretation should not be confused with its search priority. See Algorithm 13 for its pseudocode.

7.4.3 Hybrid Structure–Induced Rerooting

Motivation

While our global and local rerooters are each useful in isolation, they capture complementary aspects of the search problem and exhibit distinct limitations. The heuristic-based rerooter provides a lightweight goal-directed signal, but normalizing by the root heuristic can be sensitive to inaccurate estimates: if the root cost-to-go is underestimated, root normalization can make weights concentrate too sharply and blunt guidance. By contrast, the clustering-based rerooter relies on coarse connectivity and is agnostic to goal proximity, making it insensitive to local progress. Consequently, it may spread effort across regions that are structurally distinct but equally distant from the goal.

Rerooter Weight

The hybrid rerooter combines these signals by using global structure to set a stable coarse allocation of effort, and heuristic estimates to refine priorities within each region. This yields a coarse-to-fine rerooting rule that is less sensitive to heuristic miscalibration while retaining goal-directed adaptivity across domains. We denote this hybrid rerooter as $\sqrt{\text{LTS}}$-LH, and use the combined rerooting weights given by

$$w_1 = 1; \qquad w_t = u_a \frac{1}{M_{\tau,c_t} + \delta_{\tau,c_t}} + u_b \exp\left(-\alpha\, \frac{h(n_t)}{H_1}\right) \quad (t > 1), \qquad (7.6)$$

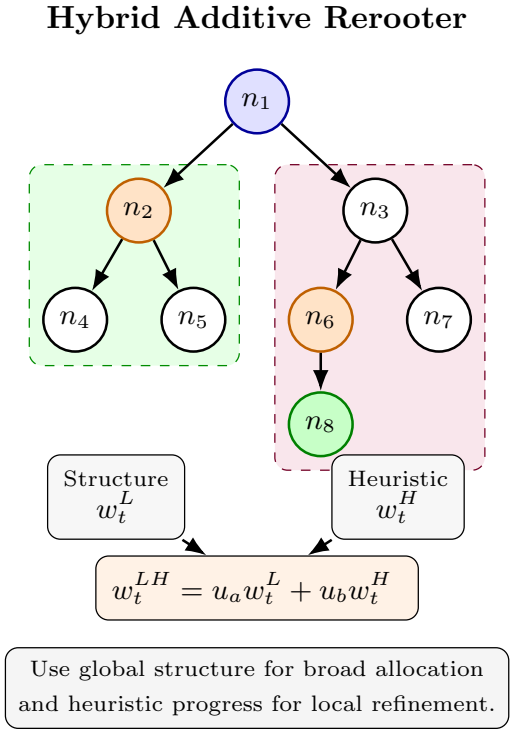


Figure 7.3: $\sqrt{\text{LTS}}$-LH Rerooter. The rerooting weight for the expanded node is computed using a mixture of the $\sqrt{\text{LTS}}$-L rerooter weight and the $\sqrt{\text{LTS}}$-H rerooter weight.

where c_t is the colour associated with n_t, M_{τ,c_t} and δ_{τ,c_t} follow the same definition as in Equation 7.4, and u_a and u_b are mixing coefficients. Unless stated otherwise, $\sqrt{\text{LTS}}$ uses $u_a = u_b = 1$. As before, we set $w_1 = 1$ for the root node. For the additive analysis, define $w_{a,1} = w_{b,1} = (u_a + u_b)^{-1}$, so that $u_a w_{a,1} + u_b w_{b,1} = 1$ agrees with the explicit implementation convention $w_1 = 1$. See Algorithm 14 for its pseudocode.

Additive Rerooter Guarantees

The hybrid rerooter in Equation 7.6 is not only a pragmatic way to blend rerooters, it also has a principled effect on how $\sqrt{\text{LTS}}$ allocates effort across subtasks. Intuitively, adding rerooters allows the search to fall back on whichever signal is informative in a given region of the tree, provided the combination remains balanced so that one component does not dominate the overall time allocation.

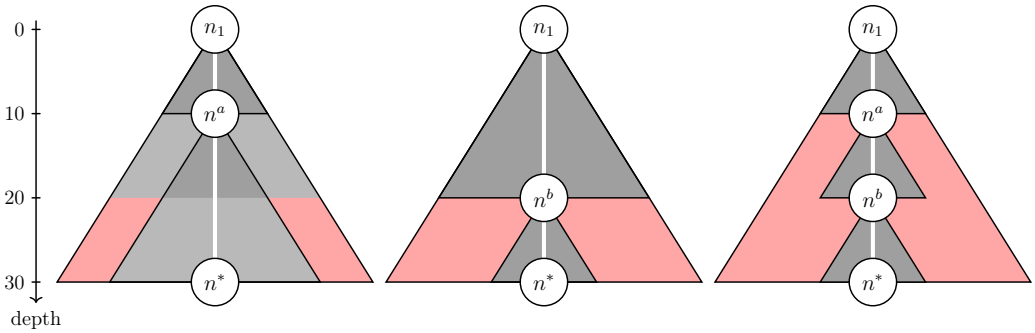


Figure 7.4: An example of how additive rerooters can take advantage of the synergy of sub-rerooters. The grey regions represent nodes in the tree $\sqrt{\text{LTS}}$ visits with each rerooter, skipping the nodes in the red region. On the left and middle are two separate rerooters, each with a subtask decomposition representing two subtasks. On the right is the additive rerooter, which combines the left and middle rerooter and has a subtask decomposition representing three subtasks.

Before stating our theorem, we start with an example that demonstrates how additive rerooters can take advantage not just of the best of the sub-rerooters, but more importantly of the *synergy* of the sub-rerooters.

Example 7.4.2 (Two rerooters)**.** Suppose $d(n^*) = 30$ in a perfect binary tree with a uniform policy. Then the LTS bound tells us that LTS would take at most $(d(n^*)+1)2^{30}$ node visits to find n^*, but the refined algorithm and bound using c_1^r actually takes at most $c_1^r(n^*) = 1+2+4+\ldots 2^{30} = 2^{31}-1$ node visits [77]. Let $n^a \prec n^*$ such that $d(n^a) = 10$. Let $n^b \prec n^*$ such that $d(n^b) = 20$.

Let us consider three rerooters w_a, w_b, w_{ab}. $\sqrt{\text{LTS}}$ with w_a visits n^a at step $T_{a,a}$ and n^* at step T_a^*. $\sqrt{\text{LTS}}$ with w_b visits n^b at step $T_{b,b}$ and n^* at step T_b^*. $\sqrt{\text{LTS}}$ with w_{ab} visits n^a at step $T_{ab,a}$ and n^b at step $T_{ab,b}$ and n^* at step T_{ab}^*. We set $w_{a,1} = w_{b,1} = w_{a,T_{a,a}} = w_{b,T_{b,b}}$, and 0 everywhere else. Furthermore, define $w_{ab} = w_a + w_b$. These three rerooters along with the nodes they visit are depicted in Figure 7.4.

The $\sqrt{\text{LTS}}$ subtask bound given in Equation 3.28 tells us that for $\sqrt{\text{LTS}}$ with rerooter w_a is competitive with the best subtask decomposition. Therefore, it is also competitive with any subtask decomposition that is meaningful for analysis. Let us compare it with the subtask decomposition $n_1 \prec n^a = n_{T_{a,a}} \prec n^*$, which happens to be the best one, in this case. The node n^* is visited at

T_a^* with

$$\begin{aligned} T_a^* - 1 &\leq w_{a,<T_a^*} \max\left\{ \frac{c_1^r(n^a)}{w_{a,1}},\ \frac{c^r_{T_{a,a}}(n^*)}{w_{a,a}} \right\} \\ &= 2 \max\left\{ \frac{2^{11}-2}{1},\ \frac{2^{21}-2}{1} \right\} \\ &= 2^{22} - 4\,. \end{aligned}$$

This is already an improvement over the 2^{31} node visits of LTS. Similarly, for $\sqrt{\text{LTS}}$ with rerooter w_b, considering the subtask decomposition $n_1 \prec n^b = n_{T_{b,b}} \prec n^*$, the node n^* is visited at step T_b^* with

$$\begin{aligned} T_b^* - 1 &\leq w_{b,<T_b^*} \max\left\{ \frac{c_1^r(n^b)}{w_{b,1}},\ \frac{c^r_{T_{b,b}}(n^*)}{w_{b,b}} \right\} \\ &= 2 \max\left\{ \frac{2^{21}-2}{1},\ \frac{2^{11}-2}{1} \right\} \\ &= 2^{22} - 4\,. \end{aligned}$$

But for $\sqrt{\text{LTS}}$ with rerooter $w_{ab} = w_a + w_b$, we can consider the subtask decomposition $n_1 \prec n^a \prec n^b \prec n^*$ since both n^a and n^b now have non-zero rerooting weights:

$$\begin{aligned} T_{ab}^* - 1 &\leq w_{ab,<T_{ab}} \max\left\{ \frac{c_1^r(n^a)}{w_{ab,1}},\ \frac{c^r_{T_{ab,a}}(n^b)}{w_{ab,a}}, \frac{c^r_{T_{ab,b}}(n^*)}{w_{ab,b}} \right\} \\ &= 4 \max\left\{ \frac{2^{11}-2}{1}, \frac{2^{11}-2}{1}, \frac{2^{11}-2}{1} \right\} \\ &= 2^{13} - 8 \\ &\approx 4\sqrt{T_a^*}\,. \end{aligned}$$

Hence, the hybrid rerooter w_{ab} benefits from the synergy of w_a and w_b together, by decomposing into 3 subtasks rather than two, yielding a factor-512 improvement in this illustrative upper-bound comparison.

We propose a more general result, stated for the slenderness-cost variant of $\sqrt{\text{LTS}}$, as a variant of the subtask decomposition bound of Theorem 3.4.18 that takes advantage of the synergy of *balanced* additive rerooters.

Theorem 7.4.3. *Let w_a and w_b be two rerooters, with relative weights $u_a, u_b > 0$. Let $w = u_a w_a + u_b w_b$ be the rerooter used by $\sqrt{\text{LTS}}$. When visiting the node*

n_T at step T, for any $C \geq 1$ satisfying $\frac{1}{C} \leq \frac{u_a w_{a,<T}}{u_b w_{b,<T}} \leq C$, and provided the component weights in the displayed ratios are positive, then the number of node visits is bounded by

$$T \leq 1 + (C+1) \times \min_{D \in \mathcal{D}(n_T)} \max_{i<|D|} \min \left\{ \frac{w_{a,<T}}{w_{a,T_i}}, \frac{w_{b,<T}}{w_{b,T_i}} \right\} c^r_{T_i}(n_{T_{i+1}}). \quad (7.7)$$

The degenerate case in which one coefficient is zero reduces to the corresponding one-component rerooter and is not covered by this balanced-additive statement.

The proof is in Appendix F.2. The coefficients u_a and u_b influence the constant $C \geq 1$, but do not determine it alone because the cumulative component weights also depend on the resulting search trajectory.

7.5 Experiments

The goal of our experimental evaluation is twofold. First, we aim to establish rerooting as a flexible and scalable abstraction for exploiting structural information in policy tree search, avoiding explicit subgoal generation and reasoning while retaining much of their benefit through simple structural signals. Second, we evaluate the practical impact of rerooting on learning efficiency by measuring improvements in online training sample efficiency relative to non-rerooted baselines across a range of domains.

7.5.1 Baselines

We compare our methods against LTS [78], as at the time of this writing, there are no comprehensive empirical evaluations of $\sqrt{\text{LTS}}$ variants against LTS. We also compare against LevinTS(π^{SG})and PHS*(π^{SG})(Chapter 6, [103]), two SGPS instantiations that can be trained online and use the same clustering method as $\sqrt{\text{LTS}}$-L. SGPS is a key baseline because it (i) shares the same clustering component, (ii) generates discrete subgoals via a VQ-VAE [105] whereas our approach uses soft subtasks without explicit subgoal generation, and (iii) achieved state-of-the-art results on the domains we consider. Finally, we include Weighted A* (WA*) [83], a non-policy method that can be trained

online via the bootstrap process. We use a weight of 1.5 for WA*, which performs well in similar settings [79, 103]. We omit HIPS-ε because it requires a solution dataset which is unavailable for the complex environment studied, and prior work found LevinTS(π^{SG}) and PHS*(π^{SG}) outperformed it [103].

7.5.2 Environment Domains

Below are the environments tested throughout these experiments. For additional details about each environment, see Appendix B.

BoulderDash The agent collects a number of diamonds to unlock the exit. Some diamonds are locked in rooms that require a key. The environment contains dirt cells, which disappear when the agent walks over them, resulting in complex state observations and a large search space. Tuero et al. [103] provided a *standard* problem set and a *hard* problem set, and we use the more challenging hard problems in our analysis.

CraftWorld The agent collects raw materials and interacts with workbenches and furnaces to craft intermediary items, which can further be crafted into final products [5]. The environment can be in a deadlocked state by crafting an incorrect item, since items are consumed once used in a recipe. Similar to BoulderDash, we use the *hard* problems from Tuero et al. [103].

Sokoban The agent must push boxes into designated goal locations, avoiding deadlocks due to boxes getting stuck along walls. Sokoban is PSPACE-hard [21], and we use the first 50,000 training problems and 1,000 test problems from Boxoban [41].

Traveling Salesman Problem (TSP) A gridworld version of the TSP where the agent must visit specified city locations, then return to the starting location. To increase the difficulty of this domain, we use a modified version in which the agent deadlocks if it revisits a city other than the start city. This

prevents trivial solutions such as blindly visiting each city without planning the steps in between cities.

7.5.3 Training and Testing Procedure

All methods use Bootstrap training [7]. We randomly initialize neural networks for the policy and/or heuristic, then run each search algorithm on a subset of training problems with an initial expansion budget using the current policy/heuristic. Algorithms update their policy/heuristic from solution trajectories on solved problems. If an algorithm solves no new problems in a sweep over the training set, we increase the expansion budget and repeat the sweep (see Section 2.3.3 for the update schedule used). After each sweep, we evaluate on a separate validation set under the current budget; training ends once at least 95% of validation problems are solved. To handle methods which struggle to make progress, we impose a maximum aggregate computation-time budget of 1,000,000 seconds (approximately 11.5 CPU-days), stopping once this limit is reached. Reported training time is aggregate computation time: the sum of time each algorithm uses on each problem, over all threads used during training. Additional details are in Appendix F.3.

Each domain uses 10,000 training problems and 1,000 validation problems, except Sokoban (49,000 training and 1,000 validation). We repeat training with five seeds, which determine network initialization and the train/validation splits. For training-efficiency plots, we report the min/mean/max number of outstanding problems versus expansions and aggregate computation time. For final evaluation, we test on a held-out test set with a budget of 512,000 node expansions and report (averaged over seeds) problems solved, expansions-to-solution, solution length, and test time. Solved count is the primary test outcome. Expansions-to-solution and solution length are averages over solved instances; when methods solve different numbers of instances, these values describe conditional performance rather than an overall efficiency ranking.

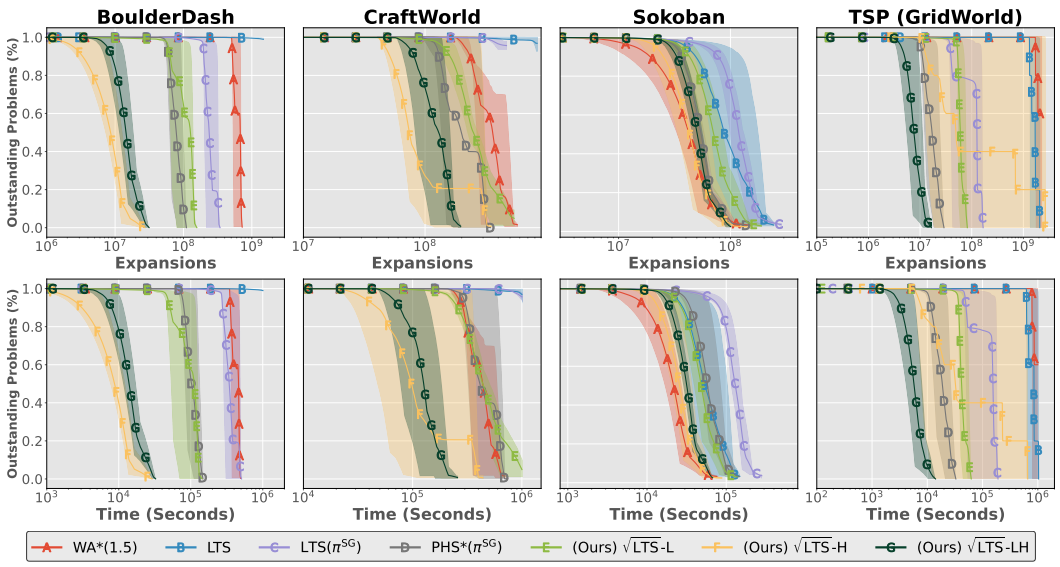


Figure 7.5: Average online training loss with respect to expansions (top) and aggregate computation time (bottom), on a log scale. Shaded regions show the minimum and maximum. Aggregate computation time measures the sum of the time an algorithm spends on each problem across all threads used during training.

7.5.4 Training Efficiency Results

Across the reported domains (Figure 7.5), the hybrid rerooter improves aggregate training efficiency relative to LTS and the evaluated subgoal-generation approaches; comparisons among the $\sqrt{\text{LTS}}$ rerooter variants are discussed separately below. The results highlight the value of combining the rerooters: In BoulderDash, $\sqrt{\text{LTS}}$-H is effective and $\sqrt{\text{LTS}}$-LH matches its performance. The remaining domains, Sokoban, CraftWorld, and TSP, all contain deadlock structure, where purely heuristic guidance can be less stable. In these domains, $\sqrt{\text{LTS}}$-H remains competitive but exhibits greater variability, suggesting that the learned heuristic can sometimes overcommit search effort to misleading regions of the tree during bootstrap training. This effect is most pronounced in CraftWorld and TSP; in these environments, WA* also requires substantially more time to complete training. By contrast, $\sqrt{\text{LTS}}$-LH is both the strongest and most stable method in these domains, indicating that the hybrid rerooter benefits from the heuristic signal when it is useful while retaining the structural robustness of the clustering rerooter $\sqrt{\text{LTS}}$-L when the

heuristic is unreliable. $\sqrt{\text{LTS}}$-L substantially outperforms LevinTS(π^{SG}) despite both using the same clustering technique and neither using heuristics. The cost-to-go rerooter $\sqrt{\text{LTS}}$-H outperforms WA* in every domain. Runtime measurements further support Theorem 7.4.1: the rerooters have low practical overhead relative to methods requiring subgoal generation, although test-time performance varies by domain. Together, these results are consistent with gains from how structural information is exploited rather than clustering alone, within the evaluated systems and domains.

7.5.5 Test Results

To assess whether the training speedup degrades solution quality, we evaluate the trained models on held-out test problems (100 per domain; 1,000 for Sokoban; Table 7.1). Methods that failed to complete training perform substantially worse, since their policies and heuristics remain largely uninformed. Despite faster training, our rerooting methods solve all test problems in the evaluated domains, indicating that the training-efficiency gains do not come at the expense of test coverage. Moreover, $\sqrt{\text{LTS}}$-LH has a lower conditional mean expansion count than the clustering-only rerooter $\sqrt{\text{LTS}}$-L on every test domain, where both methods solve all test problems; this suggests that combining global structure with local heuristic information improves generalization. In TSP and Sokoban, where the compared methods solve all test problems, our rerooting methods have higher conditional mean expansion counts than PHS*(π^{SG}) in both and WA* in Sokoban. We suspect this reflects weaker state-space clustering structure in these environments, and in Section 7.5.9 we perform an analysis to verify this conjecture.

7.5.6 Environment Domain Complexity Scaling

To test whether subgoal-based policy tree search methods like PHS*(π^{SG}) scale poorly as problem complexity increases, we run a training-efficiency experiment on BoulderDash while varying the fraction of dirt tiles which are task-irrelevant elements that substantially enlarge the state-space.

Table 7.1: Test results averaged over all seeds. Solved count is the primary outcome; expansions and solution lengths are conditional means over solved test instances and are not overall efficiency comparisons when solved counts differ. Time is measured during testing, in seconds.

Algorithm	Solved	Expansions	Length	Time
		BoulderDash		
WA*	100	480.52	**68.56**	0.73
LTS	10	195,451.08	68.92	119.86
LevinTS(π^{SG})	100	202.38	83.09	1.53
PHS*(π^{SG})	100	359.86	86.61	2.70
$\sqrt{\text{LTS}}$-L	100	131.18	71.63	0.61
$\sqrt{\text{LTS}}$-H	100	**92.37**	69.47	0.60
$\sqrt{\text{LTS}}$-LH	100	92.68	69.55	0.58
		CraftWorld		
WA*	100	1,888.35	**122.16**	3.62
LTS	100	306,224.22	142.37	373.44
LevinTS(π^{SG})	100	345,096.42	170.99	869.89
PHS*(π^{SG})	100	1,413.00	126.01	8.67
$\sqrt{\text{LTS}}$-L	100	8,802.62	183.74	44.15
$\sqrt{\text{LTS}}$-H	100	2,514.62	125.11	6.84
$\sqrt{\text{LTS}}$-LH	100	**1,347.52**	134.44	4.59
		Sokoban		
WA*	1,000	**1,201.03**	**32.91**	0.51
LTS	1,000	1,914.06	36.36	0.76
LevinTS(π^{SG})	1,000	2,010.88	35.82	1.98
PHS*(π^{SG})	1,000	1,630.60	34.66	1.56
$\sqrt{\text{LTS}}$-L	1,000	2,579.13	39.49	1.53
$\sqrt{\text{LTS}}$-H	1,000	2,626.82	34.76	1.60
$\sqrt{\text{LTS}}$-LH	1,000	1,736.03	36.37	1.10
		TSP (GridWorld)		
WA*	100	180,381.42	**35.40**	62.75
LTS	100	216.61	39.43	0.33
LevinTS(π^{SG})	100	76.99	36.40	0.69
PHS*(π^{SG})	100	**46.31**	36.39	0.49
$\sqrt{\text{LTS}}$-L	100	84.35	36.69	0.42
$\sqrt{\text{LTS}}$-H	100	55.44	35.92	0.37
$\sqrt{\text{LTS}}$-LH	100	56.12	36.16	0.38

Table 7.2: Training results averaged over increasing complexity BoulderDash environments. NPS stands for nodes per second, and time is measured in hours.

Algorithm	Expansions	Time	Solved	NPS
BoulderDash (10%)				
PHS*(π^{SG})	30,686,786	10.63	10,000	802.1
$\sqrt{\text{LTS}}$-L	28,455,464	8.36	9,998	946.0
$\sqrt{\text{LTS}}$-H	17,692,073	5.00	9,999	981.9
$\sqrt{\text{LTS}}$-LH	18,654,789	5.36	10,000	967.4
BoulderDash (20%)				
PHS*(π^{SG})	299,672,516	137.12	10,000	607.1
$\sqrt{\text{LTS}}$-L	65,307,017	20.38	9,997	889.9
$\sqrt{\text{LTS}}$-H	19,866,165	5.99	9,997	921.4
$\sqrt{\text{LTS}}$-LH	25,971,412	7.70	9,992	936.6
BoulderDash (30%)				
PHS*(π^{SG})	429,407,720	278.23	11	428.7
$\sqrt{\text{LTS}}$-L	225,438,691	72.51	9,964	863.7
$\sqrt{\text{LTS}}$-H	27,965,657	8.37	9,996	928.4
$\sqrt{\text{LTS}}$-LH	57,780,467	16.50	10,000	972.5
BoulderDash (40%)				
PHS*(π^{SG})	—	—	—	—
$\sqrt{\text{LTS}}$-L	458,508,999	153.23	9,903	831.2
$\sqrt{\text{LTS}}$-H	38,524,592	11.94	9,994	896.6
$\sqrt{\text{LTS}}$-LH	75,672,897	22.49	9,995	934.8
BoulderDash (50%)				
PHS*(π^{SG})	—	—	—	—
$\sqrt{\text{LTS}}$-L	895,423,066	317.41	605	783.6
$\sqrt{\text{LTS}}$-H	72,358,349	22.86	9,995	879.1
$\sqrt{\text{LTS}}$-LH	97,117,682	30.13	9,990	895.2

As complexity increases, PHS*(π^{SG}) times out and makes no progress once levels contain 30% or more dirt elements (Table 7.2). One possible contributing factor is its reliance on reconstructing grounded future states as subgoals: additional dirt may shift the reconstruction objective toward visually dense but task-irrelevant structure, diverting capacity from critical elements such as keys or diamonds. Although LevinTS(π^{SG}) and PHS*(π^{SG}) are more robust than earlier subgoal-based methods like HIPS-ε to invalid subgoal predictions [103], their low-level policies are still conditioned on generated subgoals, making learning increasingly brittle as state complexity grows.

In contrast, $\sqrt{\text{LTS}}$-L scales better by avoiding explicit subgoal reconstruction, though it eventually reaches a complexity threshold where training does not complete. $\sqrt{\text{LTS}}$-H and $\sqrt{\text{LTS}}$-LH remain robust across all tested levels, completing training even in the hardest settings with comparable performance. For an analysis of why $\sqrt{\text{LTS}}$-L does not scale as well as $\sqrt{\text{LTS}}$-H and $\sqrt{\text{LTS}}$-LH, see Section 7.5.9 for ablations on how the percentage of dirt elements affects the quality of the clusters found. Under this complexity manipulation and training protocol, implicitly representing subtasks via rerooting scales substantially better than conditioning policies on explicitly generated subgoals.

7.5.7 Balanced Rerooters Analysis

The constant $C \geq 1$ in Theorem 7.4.3 signals that the bound is related to how *balanced* the two sub-rerooter components are. Some natural questions to ask are how balanced the two sub-rerooters are in $\sqrt{\text{LTS}}$-LH, and how sensitive are the observed node expansions to the factor C in Theorem 7.4.3.

To answer these questions, we ran an additional sensitivity study in Sokoban, fixing $u_b = 1$ and varying u_a, which are the *weighting* applied to each sub-rerooter component. These results are summarized in Table 7.3. Performance varies only modestly across a broad range of ratios, suggesting that the hybrid rerooter is not highly sensitive to exact tuning in this study. The best observed results occur when the two signals are roughly balanced, *i.e.*, when $w_{a,<T}/w_{b,<T} \approx 1$. In practice, $u_a = u_b = 1$ is a good default. One can then inspect the observed ratio $w_{a,<T}/w_{b,<T}$ and rescale if needed, but fine-tuning is not critical.

Table 7.3: Sensitivity analysis of unbalanced rerooters. $\sqrt{\text{LTS}}$-LH is trained and tested on the Sokoban environment, with varying ratios of u_a to u_b. Time is measured in seconds.

(u_a, u_b)	Expansions	Solution Length	Time (s)	$w_{a,<T}/w_{b,<T}$
(1.00, 1.00)	1,736.03	36.37	1.10	2.03
(0.75, 1.00)	1,477.30	36.14	0.90	1.31
(0.60, 1.00)	**1,400.88**	36.05	0.84	**0.99**
(0.50, 1.00)	1,523.21	36.00	0.92	0.81
(0.25, 1.00)	1,509.74	35.89	0.90	0.44

Table 7.4: Sensitivity analysis of heuristic inverse-temperature parameter. $\sqrt{\text{LTS}}$-LH is tested on the Sokoban environment, with varying values of α. Time is measured in seconds.

α	Expansions	Length	Time (s)
1	2,949.79	37.45	2.36
5	1,959.70	36.20	1.25
10	**1,400.88**	36.05	0.84
20	1,699.64	36.47	1.13

7.5.8 Analysis of Heuristic Rerooter Temperature

The parameter α in $\sqrt{\text{LTS}}$-H and $\sqrt{\text{LTS}}$-LH acts as an inverse-temperature parameter. When α is small, the weights are more similar across nodes, so the rerooter behaves conservatively. As α increases, the weight mass becomes more concentrated on nodes whose heuristic values are small relative to the root, causing rerooting to focus more aggressively on regions estimated to be closer to a goal.

Table 7.4 shows the result of varying the α inverse-temperature parameter on the Sokoban environment for the $\sqrt{\text{LTS}}$-LH algorithm. The same trained policy and heuristic networks were used for all test runs, with $(u_a, u_b) = (1, 1)$, as used throughout the headline experiments. The value $\alpha = 10$ was fixed a priori for the headline experiments; this table is a post hoc sensitivity analysis and does not select the value. As α decreases towards 1, the number of expansions increases due to the rerooting weight contribution from the heuristic rerooter being similar across all nodes. Larger values of α allow the search to focus on areas of the tree where progress is being made towards the goal. However, if α becomes too large, then the search can overcommit to what seems like progress being made but what could be a deadlocked subtree.

7.5.9 Analysis of Clustering-based Rerooters

From the results in Table 7.1 and Table 7.2, $\sqrt{\text{LTS}}$-L can perform quite a bit worse than $\sqrt{\text{LTS}}$-H and $\sqrt{\text{LTS}}$-LH. The usefulness of the information provided by the clusters found which $\sqrt{\text{LTS}}$-L relies on depends on the underlying

Table 7.5: Average number of edges crossing different clusters. This is a descriptive within-setting count; because it is not normalized by graph or cluster size, it is not a cross-domain structure metric.

Environment	Avg Edge Crossing per Cluster
BoulderDash (10%)	7.6219
BoulderDash (20%)	8.8399
BoulderDash (30%)	11.0756
BoulderDash (40%)	13.6613
BoulderDash (50%)	16.6101
CraftWorld	9.7805
Sokoban	16.6344
TSP (GridWorld)	35.2884

structure of the state-space. The clustering rerooter is most effective when the induced state graph has well-separated regions connected by narrow bottlenecks, because entering a new cluster then signals genuine progress. When comparing against $\text{PHS}^*(\pi^{\text{SG}})$ in Table 7.1, $\sqrt{\text{LTS}}$-L performs relatively worse in the Sokoban and TSP GridWorld environments. Since $\sqrt{\text{LTS}}$-L reached the time limit during training on the CraftWorld environment and did not fully complete the bootstrap process, we omit it from this analysis. In Table 7.2, as the complexity of the BoulderDash environments increases, $\sqrt{\text{LTS}}$-L continues to degrade relative to the other rerooters.

To provide a descriptive view of the induced graphs, we measured the average number of edges crossing each cluster boundary, averaged over problems. Because this count is not normalized by graph or cluster size, comparisons are qualitative and must be interpreted alongside the corresponding problem settings. These results are given in Table 7.5. Under this metric, BoulderDash exhibits a clear progression as the domain becomes less structured, CraftWorld remains relatively low, whereas Sokoban and TSP are substantially higher. In Sokoban, this helps explain why the clustering rerooter is less effective than the strongest baselines, and in TSP, the very large value is consistent with the strongest degradation, where the clustering rerooter requires nearly twice as many expansions as the heuristic/hybrid rerooters (despite both test-time results being overall low). Overall, these results are consistent with clustering-based rerooting benefiting from sharp, bottleneck-like state-space structure,

while this unnormalized count provides only a descriptive diagnostic rather than a definitive cross-domain characterization.

7.5.10 Ablation: Impact of Clustering Level

The Leiden clustering algorithm takes a base graph G_0 and produces a hierarchy of cluster graphs $\{G_1, \ldots, G_N\}$. The selected graph at level $k \in [1, N]$ influences what structures are used when colouring nodes, which the rerooters $\sqrt{\text{LTS}}$-L and $\sqrt{\text{LTS}}$-LH use to compute their rerooting weights.

Table 7.6 shows the bootstrap training loss for $\sqrt{\text{LTS}}$-L over the BoulderDash 20% dirt environment of varying cluster levels. At each clustering invocation, N denotes the hierarchy depth returned by Leiden and $k = \lfloor N/2 \rfloor$ selects the lower integer midpoint level. A cluster level of $k = \lfloor N/2 \rfloor$ completes training in the fewest number of expansions and time, but using $k = N$ is also competitive. We note that this will be dependent on the implementation of the Leiden algorithm being used as different approximations and optimizations can be used. See Appendix F.3 for details and the specific implementation used.

Table 7.6: Training results of $\sqrt{\text{LTS}}$-L over the BoulderDash 20% environments of varying cluster levels.

Cluster Level (k)	Expansions	Time (Hours)
1	166,690,697	42.86
$N/2$	**65,307,017**	**20.38**
N	77,504,769	21.44

7.5.11 Ablation: Impact of Graph Update Frequency

The cluster graphs from the Leiden algorithm are built using an update schedule which follows a geometric growth strategy with factor γ. How frequently the graphs are updated will have two major consequences. One is the overall speed of the algorithm, where using a smaller γ will result in more graphs being built under a fixed budget. Creating cluster graphs more frequently means that more time is spent in the `leiden` procedure than actually running the

Table 7.7: Test results of $\sqrt{\text{LTS}}$-L on the Sokoban environment of varying cluster graph update factors. NPS stands for nodes per second, and time is measured in seconds.

Cluster Update Frequency (γ)	Expansions	Time	NPS
1.05	2,932.19	2.01	7.57
1.10	2,664.08	1.36	8.00
1.20	2,715.87	1.42	8.47
1.50	2,835.79	1.49	8.71
2.00	3,187.88	1.71	8.67
4.00	3,357.53	2.12	9.24

search, resulting in a lower number of nodes-per-second which can be achieved. The second is that using a larger γ means that there will be more expansions between graph updates, with a larger percentage of those nodes being generated after the most recent cluster graph. If a node is expanded which does not have a colour assigned to it (as is the case when an expanded node was generated after the most recent graph clustering), then an approximation is used where we assume that node gets the same colour as its parent. A larger γ means that a larger percentage of the nodes will have an approximated colour.

Table 7.7 shows the test results of using the same trained policy on the Sokoban environment, but with varying values of γ. In general, as γ increases, the achieved number of nodes per second increases. We also see that as γ decreases, the number of expansions decreases, as expected, because the re-rooter can make greater use of the underlying state-space structure to assign informative weights rather than relying on approximations.

7.6 Conclusions

In this chapter, we examined rerooting as a general mechanism for exploiting structure in policy tree search and demonstrated its effectiveness across a range of instantiations. We introduced two complementary rerooters, and showed how they can be combined to take advantage of the benefits each provides while offsetting each of their downsides. Our rerooters achieve strong online-training efficiency relative to the bootstrap-based methods evaluated, while foregoing the need to rely on costly subgoal-generation models and state reconstruction.

Our results show that rerooting enables scalable and efficient search control by implicitly representing subtasks through weighted root selection rather than explicit subgoal reconstruction.

Perhaps surprisingly, a simple heuristic-based rerooter often outperforms a clustering-based alternative despite relying on substantially less structural information, highlighting that lightweight local signals can be sufficient to guide effective rerooting. It also indicates that the chosen clustering statistic can be weak, insufficiently informative, or too costly on some domains. At the same time, the heuristic rerooter is not without limitations: under certain conditions, normalization effects can cause rerooting weights to collapse, leading to degraded performance. The hybrid rerooter addresses this tradeoff by combining global structural guidance with local heuristic information, achieving performance close to the heuristic-based approach while being robust across domains.

Together, these findings position rerooting as a flexible and practical abstraction, and suggest that combining complementary structural signals offers a reliable path toward scalable policy tree search. We have demonstrated that $\sqrt{\text{LTS}}$ with a simple transformation of the classical heuristic *cost-to-go* can be very efficient. We have also successfully combined two rerooters. This opens the door to incorporating different types of side information to define more capable rerooters.

Algorithm 12 $\sqrt{\text{LTS}}$-L

Input: Root node n_1, budget $b > 0$, policy π_θ, graph update factor $\gamma > 1$, cluster level k
Output: Solution node, or status of failed search
$Q_1 \leftarrow \{n_1\}$
$V_0 \leftarrow \{n_1\}$, $E_0 \leftarrow \{\}$
$t \leftarrow 1$, $\tau \leftarrow 0$, $r_1 \leftarrow 1$
while $Q_t \neq \{\}$ **and** $t \leq b$ **do**
    $n_t = \arg\min_{n \in Q_t} \min_{n_i \prec n} \frac{1}{w_i} \left(\sum_{n_i \prec n' \preceq n} \frac{1}{\pi_\theta(n'|n_i)} \right)$ ▷ *root priority is* 0
    if `is_solution`(n_t) **then**
        return n_t
    ▷ *Find node's colour* ◁
    if `is_root`(n_t) **then**
        $w_t \leftarrow 1$
    else
        if `has_colour`(n_t) **then**
            $c_t \leftarrow$ `colour`(n_t)
        else
            $c_t \leftarrow$ `colour`(`par`(n_t))
        $\delta_{\tau,c_t} = |\{\ell : r_\tau < \ell \leq t,\ c_\ell = c_t\}|$
        $w_t \leftarrow 1/\left(M_{\tau,c_t} + \delta_{\tau,c_t}\right)$
    ▷ *Incrementally update tree and induced graph* ◁
    $Q_{t+1} \leftarrow Q_t \backslash \{n_t\} \cup \mathcal{C}(n_t)$
    $V_0 \leftarrow V_0 \cup \mathcal{C}(n_t)$
    $E_0 \leftarrow E_0 \cup \{(n_t, n') : n' \in \mathcal{C}(n_t)\}$
    ▷ *Clustering is expensive, we amortize the cost over the search steps using an exponential schedule* ◁
    if $t = r_{\tau+1}$ **then**
        $\{G_1, \ldots, G_N\} \leftarrow$ `leiden`$(G_0 = (V_0, E_0))$ ▷ *Cluster Step*
        ▷ *Update node colours and colour count map* ◁
        $(V_k, E_k) \leftarrow G_k$
        for i, U **in** `enumerate`(V_k) **do**
            for v **in** U **do**
                `colour`$(v) \leftarrow i$
        $\tau \leftarrow \tau + 1$
        $M_{\tau,c} \leftarrow |\{v \in V_0 : \texttt{colour}(v) = c\}|$ for all $c \in \{1, \ldots, |V_k|\}$
        $r_{\tau+1} \leftarrow \lceil \gamma r_\tau \rceil$
    $t \leftarrow t + 1$
return `False`

Algorithm 13 $\sqrt{\text{LTS}}$-H

Input: Root node n_1, budget $b > 0$, policy π_θ, nonnegative heuristic h_ω, inverse-temperature $\alpha > 0$
Output: Solution node, or status of failed search
$t \leftarrow 1$
$H_1 \leftarrow \max\{1, h_\omega(n_1)\}$
$Q_t \leftarrow \{n_1\}$
while $Q_t \neq \{\}$ **and** $t \leq b$ **do**
    $n_t = \arg\min_{n \in Q_t} \min_{n_i \prec n} \frac{1}{w_i} \left(\sum_{n_i \prec n' \preceq n} \frac{1}{\pi_\theta(n'|n_i)} \right)$ ▷ *root priority is* 0
    if `is_solution`(n_t) **then**
        return n_t
    if `is_root`(n_t) **then**
        $w_t \leftarrow 1$
    else
        $w_t \leftarrow \exp\left(-\alpha \frac{h_\omega(n_t)}{H_1}\right)$
    $Q_{t+1} \leftarrow Q_t \backslash \{n_t\} \cup \mathcal{C}(n_t)$
    $t \leftarrow t + 1$
return `False`

Algorithm 14 $\sqrt{\text{LTS}}$-LH

Input: Root node n_1, budget $b > 0$, policy π_θ, nonnegative heuristic h_ω, graph update factor $\gamma > 1$, cluster level k, inverse-temperature $\alpha > 0$, mixing coefficients $u_a, u_b \geq 0$
Output: Solution node, or status of failed search
$Q_1 \leftarrow \{n_1\}$
$V_0 \leftarrow \{n_1\}$, $E_0 \leftarrow \{\}$
$t \leftarrow 1$, $\tau \leftarrow 0$, $r_1 \leftarrow 1$, $H_1 \leftarrow \max\{1, h_\omega(n_1)\}$
while $Q_t \neq \{\}$ **and** $t \leq b$ **do**
    $n_t = \arg\min_{n \in Q_t} \min_{n_i \prec n} \frac{1}{w_i} \left(\sum_{n_i \prec n' \preceq n} \frac{1}{\pi_\theta(n'|n_i)} \right)$ ▷ *root priority is* 0
    if `is_solution`(n_t) **then**
        return n_t
    ▷ *Find node's colour* ◁
    if `is_root`(n_t) **then**
        $w_t \leftarrow 1$
    else
        if `has_colour`(n_t) **then**
            $c_t \leftarrow$ `colour`(n_t)
        else
            $c_t \leftarrow$ `colour`(`par`(n_t))
        $\delta_{\tau,c_t} = |\{\ell : r_\tau < \ell \leq t,\ c_\ell = c_t\}|$
        $w_t \leftarrow u_a \left(M_{\tau,c_t} + \delta_{\tau,c_t}\right)^{-1} + u_b \exp\left(-\alpha \frac{h_\omega(n_t)}{H_1}\right)$
    ▷ *Incrementally update tree and induced graph* ◁
    $Q_{t+1} \leftarrow Q_t \backslash \{n_t\} \cup \mathcal{C}(n_t)$
    $V_0 \leftarrow V_0 \cup \mathcal{C}(n_t)$
    $E_0 \leftarrow E_0 \cup \{(n_t, n') : n' \in \mathcal{C}(n_t)\}$
    ▷ *Clustering is expensive, we amortize the cost over the search steps using an exponential schedule* ◁
    if $t = r_{\tau+1}$ **then**
        $\{G_1, \ldots, G_N\} \leftarrow$ `leiden`$(G_0 = (V_0, E_0))$ ▷ *Cluster Step*
        ▷ *Update node colours and colour count map* ◁
        $(V_k, E_k) \leftarrow G_k$
        for i, U **in** `enumerate`(V_k) **do**
            for v **in** U **do**
                `colour`$(v) \leftarrow i$
        $\tau \leftarrow \tau + 1$
        $M_{\tau,c} \leftarrow |\{v \in V_0 :$ `colour`$(v) = c\}|$ for all $c \in \{1, \ldots, |V_k|\}$
        $r_{\tau+1} \leftarrow \lceil \gamma r_\tau \rceil$
    $t \leftarrow t + 1$
return `False`

Chapter 8

Conclusions and Future Work

The previous chapters developed and evaluated several learning-based approaches to search control in policy tree search. This final chapter brings those contributions together, reflects on their broader significance, and outlines promising directions for future work.

8.1 Contributions

This thesis studied how search effort should be allocated and redirected in policy tree search for challenging single-agent deterministic first-solution domains with sparse intermediate feedback. Its central claim has been that a key challenge in this setting is not only learning useful guidance, but making effective decisions about computation: deciding where additional search effort is most likely to improve time-to-solution. Throughout this thesis, search control was treated not as a fixed set of hand-designed rules, but as an adaptive capability that can be learned and improved by exploiting structure in the problems being solved.

The first contribution of this thesis was to show that instance-sensitive distributional prediction can improve restart-based policy tree search. Chapter 4 introduced Bayes DistNet, a Bayesian neural model for runtime-distribution prediction evaluated in low-data and censored-observation comparisons. Chapter 5 then showed how these predictions can be used to guide Luby Tree Search by replacing a fixed global shift with an instance-specific one. This produced a more adaptive restart strategy that reduced wasted effort on shallow rollouts

and showed more favourable behaviour than the naïve uniform-shift baseline under the artificial censoring protocol. In this way, the thesis demonstrated that predictive models of search behaviour can be used directly to improve decisions about how long search should continue before restarting.

The second contribution was to show that policy tree search can benefit from learned intermediate structure in the form of subgoals. Chapter 6 introduced subgoal-guided extensions of LevinTS and PHS* in which a learned subgoal generator and subgoal-conditioned policies induce a hierarchical decomposition of the task. A key feature of this approach is that it can learn not only from successful searches, but also from failed budget-bounded searches that would otherwise be discarded. This improved reported training expansion efficiency and allowed useful policies to be learned in domains where standard Bootstrap training is substantially less effective. More broadly, this contribution showed that explicit intermediate structure can decompose long trajectories into shorter conditioned segments and improve reported training efficiency in the evaluated sparse-feedback domains.

The third contribution was to show that some of the same benefits can be obtained without explicit subgoal generation. Chapter 7 introduced automated rerooters for $\sqrt{\text{LTS}}$ using structural information derived from the state space. These rerooters use global clustering signals, local heuristic information, and their combination to redirect computation toward promising parts of the search. This provides an alternative to explicit subgoal reasoning while still exploiting structure to improve search control; its overhead is lower than subgoal generation in the reported systems, but clustering and ancestor-based rerooting still incur computational cost. In addition to the empirical improvements in training efficiency and scalability, this chapter also strengthened the theoretical foundations of rerooting by providing a bound for additive rerooters.

Taken together, these contributions support the thesis that effective policy tree search in the setting studied here depends on learning how to control computation. Across the methods developed here, the recurring theme is that better search performance arises from adapting effort to the instance,

using informative intermediate structure, and redirecting computation when the search enters more promising regions of the tree. On the evaluated deterministic benchmark domains, the results show that instance-sensitive restart control, explicit subgoal conditioning, and structure-induced rerooting can improve particular measures of training or search efficiency. The extent to which these gains transfer beyond this setting remains open. More broadly, the work of this thesis suggests that progress in this class of policy tree search problems can come not only from learning better guidance, but from learning how to spend computation wisely.

8.2 Synthesis and Limitations

The methods in this thesis address different failure modes of policy tree search and make different information and computation tradeoffs. They should therefore not be treated as interchangeable.

Distribution-guided restart. This mechanism uses the initial state observation and successful-rollout data to estimate an instance-specific shift. It is most useful when successful rollout depths vary across instances and the conditional trajectory-length model is informative. Its online overhead is low after prediction, but the conditional model or selected shift can be misspecified.

Explicit subgoals. This mechanism uses a VQVAE, graph paths, and subgoal-conditioned policies to learn from failed search trees and decompose trajectories into conditioned segments. It can improve training expansion efficiency, but requires additional models, reconstruction, and path data; its main risks are reconstruction, path, and model-capacity failures.

Heuristic rerooting. This mechanism uses a learned cost-to-go estimate as a lightweight local goal-proximity signal. It has low online overhead and can redirect effort toward promising regions, but is vulnerable to heuristic miscalibration and excessive weight concentration.

Clustering rerooting. This mechanism uses the induced state graph and online clustering to exploit global structural separation. It can be effective when the graph contains meaningful bottleneck-like regions, but has medium-to-high overhead from graph updates and ancestor-priority evaluation, and is less useful when clusters are weak or uninformative.

These are empirical conditions from the evaluated domains, not universal selection rules. This thesis is limited to deterministic, single-agent first-solution search and does not provide solution-optimality guarantees. Its formal results require positive policy support for a solution path and the algorithm-specific search assumptions. The learned models may be misspecified or sensitive to distribution shift, and the empirical studies concentrate on structured benchmark domains. The experiments also do not fully separate model capacity from search-control effects, nor do they provide guarantees for dynamic, history-dependent rerooters with stronger duplicate pruning. Training, clustering, and model-inference overhead remain important considerations in applying these methods. Within these bounds, the results provide a basis for treating search control as an explicit, empirically testable design dimension of policy tree search.

8.3 Future Work

One natural direction for future work is to unify the search-control mechanisms developed in this thesis within a single adaptive framework. In this thesis, restart control, learned subgoal guidance, and rerooting were studied separately. However, these mechanisms address related aspects of the same underlying problem: deciding how computation should be distributed during search. A promising next step is therefore to investigate methods that can combine them, or decide dynamically which form of search control should dominate for a given instance or at a particular stage of the search. Such a framework could make policy tree search both more flexible and more robust across a wider range of problem types.

A second direction is to incorporate uncertainty more directly into search

control. Chapter 4 showed that uncertainty estimates are useful when data are scarce or censored, but the later methods in this thesis mostly use learned predictions as point estimates. This could include developing models that separate aleatoric outcome variation from epistemic weight uncertainty and that represent defective hitting-time distributions with both a success probability and a conditional successful-trajectory-length model. Future work could study how uncertainty can be integrated into the decision-making process itself, for example by moderating restart decisions, influencing which generated subgoals should be trusted, or controlling how strongly rerooting should respond to structural signals. This may be especially important in out-of-distribution settings, where search guidance can otherwise become overconfident.

A third direction is to study richer forms of learned structure. This thesis considered two important forms: explicit subgoals and implicit rerooting structure. Future work could investigate whether these ideas can be extended to more abstract representations of problem structure, such as bottlenecks, latent regions, or predictive models of future rerooting opportunities. More generally, an important open question is when explicit hierarchical decomposition is worth its additional modelling cost, and when lower-overhead structural redirection is the better choice. Answering this would require cost-aware selection between mechanisms and more principled joint optimization of policy learning and search-control decisions.

A fourth direction is to extend the framework developed here beyond the first-solution deterministic setting emphasized in this thesis. The problems considered here are particularly well suited to policy tree search because the dominant issue is how to find any solution efficiently in a large search space. It would be valuable to investigate how the same ideas transfer to settings in which stochasticity, partial observability, or stronger solution-quality requirements play a larger role. Common evaluation protocols that account jointly for solved and unsolved instances, together with guarantees for dynamic rerooters, would make such comparisons more reliable. This would help determine how broadly the principle of computation-aware search control applies beyond the class of problems studied in this thesis.

In closing, this thesis has argued that search control should be treated as a learning problem in its own right. The methods developed here are one step in that direction and motivate further study of better policies, heuristics, models, and decisions about how computation itself is used.

References

[1] A. Agostinelli, K. Arulkumaran, M. Sarrico, P. Richemond, and A. A. Bharath, "Memory-efficient episodic control reinforcement learning with dynamic online K-means," *arXiv preprint arXiv:1911.09560*, 2019.

[2] F. Agostinelli, S. McAleer, A. Shmakov, and P. Baldi, "Solving the Rubik's cube with deep reinforcement learning and search," *Nature Machine Intelligence*, vol. 1, no. 8, pp. 356–363, 2019.

[3] F. Agostinelli, S. S. Shperberg, A. Shmakov, S. McAleer, R. Fox, and P. Baldi, "Q* search: Heuristic search with deep q-networks," in *ICAPS Workshop on Bridging the Gap between AI Planning and Reinforcement Learning*, 2024.

[4] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete problems in AI safety," arXiv preprint arXiv:1606.06565, 2016.

[5] J. Andreas, D. Klein, and S. Levine, "Modular multitask reinforcement learning with policy sketches," in *Proceedings of the 34th International Conference on Machine Learning*, 2017.

[6] S. J. Arfaee, S. Zilles, and R. Holte, "Bootstrap learning of heuristic functions," in *Proceedings of the International Symposium on Combinatorial Search*, vol. 1, 2010, pp. 52–60.

[7] S. J. Arfaee, S. Zilles, and R. C. Holte, "Learning heuristic functions for large state spaces," *Artificial Intelligence*, vol. 175, no. 16-17, pp. 2075–2098, 2011.

[8] A. Aubret, L. Matignon, and S. Hassas, "Distop: Discovering a topological representation to learn diverse and rewarding skills," *IEEE Transactions on Cognitive and Developmental Systems*, vol. 15, no. 4, pp. 1905–1915, 2023.

[9] M. Bain and C. Sammut, "A framework for behavioural cloning," in *Machine intelligence 15*, 1995, pp. 103–129.

[10] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe, "Variational inference: A review for statisticians," *Journal of the American statistical Association*, vol. 112, no. 518, pp. 859–877, 2017.

[11] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, P10008, 2008.

[12] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra, "Weight uncertainty in neural networks," in *Proceedings of the 32nd International Conference on Machine Learning*, 2015.

[13] M. M. Botvinick, Y. Niv, and A. G. Barto, "Hierarchically organized behavior and its neural foundations: A reinforcement learning perspective," *cognition*, vol. 113, no. 3, pp. 262–280, 2009.

[14] U. Brandes et al., "Maximizing modularity is hard," *arXiv preprint physics/0608255*, 2006.

[15] C. B. Browne et al., "A survey of Monte Carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[16] A. Canesse, M. Petitbois, L. Denoyer, S. Lamprier, and R. Portelas, "Navigation with QPHIL: Quantizing planner for hierarchical implicit q-learning," in *2025 International Joint Conference on Neural Networks (IJCNN)*, 2025.

[17] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *International Conference on Computer Aided Verification*, Springer, 2000.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[19] C. G. Correa, M. K. Ho, F. Callaway, N. D. Daw, and T. L. Griffiths, "Humans decompose tasks by trading off utility and computational cost," *PLoS computational biology*, vol. 19, no. 6, 2023.

[20] J. C. Culberson and J. Schaeffer, "Pattern databases," *Computational Intelligence*, vol. 14, no. 3, pp. 318–334, 1998.

[21] J. C. Culberson, "Sokoban is PSPACE-complete," in *International Conference on Fun with Algorithms (FUN)*, 1999, pp. 65–76.

[22] K. Czechowski et al., "Subgoal search for complex reasoning tasks," *Advances in Neural Information Processing Systems*, 2021.

[23] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, pp. 269–271, 1959.

[24] F. Donnarumma, D. Maisto, and G. Pezzulo, "Problem solving as probabilistic inference with subgoaling: Explaining human successes and pitfalls in the tower of hanoi," *PLoS computational biology*, vol. 12, no. 4, 2016.

[25] J. E. Doran and D. Michie, "Experiments with the graph traverser program," *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, vol. 294, no. 1437, pp. 235–259, 1966.

[26] S. Edelkamp and S. Schrödl, *Heuristic search: theory and applications*. Elsevier, 2011.

[27] K. Eggensperger, M. Lindauer, and F. Hutter, "Neural networks for predicting algorithm runtime distributions," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, 2018.

[28] J. B. Evans and Ö. Şimşek, "Creating multi-level skill hierarchies in reinforcement learning," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[29] A. Felner, R. E. Korf, and S. Hanan, "Additive pattern database heuristics," *Journal of Artificial Intelligence Research*, vol. 22, pp. 279–318, 2004.

[30] P. Ferber, M. Helmert, and J. Hoffmann, "Neural network heuristics for classical planning: A study of hyperparameter space," in *ECAI*, 2020.

[31] M. Fortunato, C. Blundell, and O. Vinyals, "Bayesian recurrent neural networks," arXiv preprint arXiv:1704.02798, 2017.

[32] M. Gagliolo and J. Schmidhuber, "A neural network model for inter-problem adaptive online time allocation," in *International Conference on Artificial Neural Networks*, Springer, 2005.

[33] M. Gagliolo and J. Schmidhuber, "Impact of censored sampling on the performance of restart strategies," in *International Conference on Principles and Practice of Constraint Programming*, 2006.

[34] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[35] M. Gebser, B. Kaufmann, and T. Schaub, "Conflict-driven answer set solving: From theory to practice," *Artificial Intelligence*, vol. 187, 2012.

[36] A. E. Gelfand and A. F. Smith, "Sampling-based approaches to calculating marginal densities," *Journal of the American statistical association*, vol. 85, no. 410, pp. 398–409, 1990.

[37] A. Gerevini and I. Serina, "LPG: A planner based on local search for planning graphs with action costs," in *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems*, 2002.

[38] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011.

[39] C. P. Gomes, B. Selman, and H. Kautz, "Boosting combinatorial search through randomization," in *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, 1998.

[40] A. Graves, "Practical variational inference for neural networks," in *Advances in neural information processing systems*, 2011.

[41] A. Guez et al., *An investigation of model-free planning: Boxoban levels*, https://github.com/deepmind/boxoban-levels, 2018.

[42] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[43] W. D. Harvey, "Nonsystematic backtracking search," Ph.D. dissertation, Stanford, CA, USA, 1995.

[44] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[45] G. E. Hinton and D. Van Camp, "Keeping the neural networks simple by minimizing the description length of the weights," in *Proceedings of the sixth annual conference on Computational learning theory*, 1993.

[46] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*, PMLR, 2015.

[47] R. Islam et al., "Discrete factorial representations as an abstraction for goal conditioned reinforcement learning," *arXiv preprint arXiv:2211.00247*, 2022.

[48] E. Karpas, O. Betzalel, S. E. Shimony, D. Tolpin, and A. Felner, "Rational deployment of multiple heuristics in optimal state-space search," *Artificial Intelligence*, vol. 256, pp. 181–210, 2018.

[49] A. Kendall and Y. Gal, "What uncertainties do we need in Bayesian deep learning for computer vision?" In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017.

[50] J. Kim, Y. Seo, and J. Shin, "Landmark-guided subgoal generation in hierarchical reinforcement learning," *Advances in Neural Information Processing Systems*, 2021.

[51] D. P. Kingma, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[52] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *CoRR*, vol. abs/1312.6114, 2014.

[53] J. Klein and M. Moeschberger, *Survival analysis: Techniques for censored and truncated data*. New York: Springer-Verlag, 2003.

[54] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *European Conference on Machine Learning*, Springer, 2006.

[55] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[56] M. Krajňanskỳ, J. Hoffmann, O. Buffet, and A. Fern, "Learning pruning rules for heuristic search planning," in *21st European Conference on Artificial Intelligence*, 2014.

[57] K. Kujanpää, J. Pajarinen, and A. Ilin, "Hierarchical imitation learning with vector quantized models," in *International Conference on Machine Learning*, PMLR, 2023.

[58] K. Kujanpää, J. Pajarinen, and A. Ilin, "Hybrid search for efficient planning with completeness guarantees," *Advances in Neural Information Processing Systems*, 2024.

[59] C. Y. Lee, "An algorithm for path connections and its applications," *IRE Transactions on Electronic Computers*, 1961.

[60] S. Lee, D. Cho, J. Park, and H. J. Kim, "CQM: Curriculum reinforcement learning with a quantized world model," in *Advances in Neural Information Processing Systems*, 2023.

[61] E. A. Leicht and M. E. Newman, "Community structure in directed networks," *Physical review letters*, vol. 100, no. 11, p. 118 703, 2008.

[62] S. Li, J. Zhang, J. Wang, Y. Yu, and C. Zhang, "Active hierarchical exploration with stable subgoal representation learning," *arXiv preprint arXiv:2105.14750*, 2021.

[63] G. Liu and K. Ramakrishnan, "A* Prune: An algorithm for finding K shortest paths subject to multiple constraints," in *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society*, 2001.

[64] M. Luby, A. Sinclair, and D. Zuckerman, "Optimal speedup of Las Vegas algorithms," *Information Processing Letters*, vol. 47, no. 4, pp. 173–180, 1993.

[65] D. J. MacKay, "A practical Bayesian framework for backpropagation networks," *Neural computation*, vol. 4, no. 3, pp. 448–472, 1992.

[66] P. Mazzaglia, T. Verbelen, B. Dhoedt, A. Lacoste, and S. Rajeswar, "Choreographer: Learning and adapting skills in imagination," *arXiv preprint arXiv:2211.13350*, 2022.

[67] E. F. Moore, "The shortest path through a maze," in *Proceedings of the International Symposium on the Theory of Switching*, Harvard University Press, 1959, pp. 285–292.

[68] H. Nakhost, *Random walk planning: Theory, practice, and application*. University of Alberta (Canada), 2013.

[69] V. Narayanan and M. Likhachev, "Heuristic search on graphs with existence priors for expensive-to-evaluate edges," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 27, 2017.

[70] R. M. Neal, *Bayesian learning for neural networks*. Springer Science & Business Media, 2012, vol. 118.

[71] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical review E*, vol. 69, no. 2, 2004.

[72] A. Y. Ng and S. J. Russell, "Algorithms for inverse reinforcement learning," in *Proceedings of the Seventeenth International Conference on Machine Learning*, 2000.

[73] M. Nikolić, F. Marić, and P. Janičić, "Instance-based selection of policies for SAT solvers," in *Theory and Applications of Satisfiability Testing*, O. Kullmann, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 326–340.

[74] N. J. Nilsson, *Principles of artificial intelligence*. Morgan Kaufmann, 2014.

[75] M. Opper and C. Archambeau, "The variational Gaussian approximation revisited," *Neural computation*, vol. 21, no. 3, pp. 786–792, 2009.

[76] L. Orseau, M. Hutter, and L. H. Lelis, "Levin tree search with context models," *arXiv preprint arXiv:2305.16945*, 2023.

[77] L. Orseau, M. Hutter, and L. H. Lelis, "Exponential speedups by rerooting Levin tree search," *arXiv preprint arXiv:2412.05196*, 2024.

[78] L. Orseau, L. Lelis, T. Lattimore, and T. Weber, "Single-agent policy tree search with guarantees," *Advances in Neural Information Processing Systems*, 2018.

[79] L. Orseau and L. H. Lelis, "Policy-guided heuristic search with guarantees," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021.

[80] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., 2019.

[81] J. Pearl, *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., 1984.

[82] J. S. Penberthy and D. S. Weld, "Temporal planning with continuous change," in *Proceedings of the Twelfth AAAI National Conference on Artificial Intelligence*, ser. AAAI'94, Seattle, Washington: AAAI Press, 1994, pp. 1010–1015.

[83] I. Pohl, "Heuristic search viewed as path finding in a graph," *Artificial intelligence*, vol. 1, no. 3-4, pp. 193–204, 1970.

[84] D. A. Pomerleau, "ALVINN: An autonomous land vehicle in a neural network," *Advances in Neural Information Processing Systems*, 1988.

[85] L. Prechelt, "Automatic early stopping using cross validation: Quantifying the criteria," *Neural Networks*, vol. 11, no. 4, pp. 761–767, 1998.

[86] R. Ramesh, M. Tomar, and B. Ravindran, "Successor options: An option discovery framework for reinforcement learning," *arXiv preprint arXiv:1905.05731*, 2019.

[87] C. D. Rosin, "Multi-armed bandits with episode context," *Annals of Mathematics and Artificial Intelligence*, vol. 61, no. 3, pp. 203–230, 2011.

[88] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd. USA: Prentice Hall Press, 2009, ISBN: 0136042597.

[89] M. Salerno, R. Fuentetaja, D. Speck, and J. Seipp, "Merging Cartesian abstractions for classical planning," in *Proceedings of the 28th European Conference on Artificial Intelligence (ECAI 2025)*, IOS Press, 2025.

[90] J. Schrittwieser et al., "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[91] J. Seipp and M. Helmert, "Counterexample-guided cartesian abstraction refinement," in *Proceedings of the International Conference on Automated Planning and Scheduling*, 2013.

[92] K. Shridhar, F. Laumann, and M. Liwicki, *A comprehensive guide to Bayesian convolutional neural network with variational inference*, arXiv preprint arXiv:1901.02731, 2019.

[93] D. Silver et al., "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[94] D. Silver et al., "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[95] D. Speck and J. Seipp, "New refinement strategies for cartesian abstractions," in *Proceedings of the International Conference on Automated Planning and Scheduling*, 2022.

[96] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[97] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in Neural Information Processing Systems*, 2014.

[98] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial intelligence*, vol. 112, no. 1-2, pp. 181–211, 1999.

[99] C. S. Tan, R. Mohd-Mokhtar, and M. R. Arshad, "A comprehensive review of coverage path planning in robotics using classical and heuristic algorithms," *IEEE Access*, vol. 9, pp. 119 310–119 342, 2021.

[100] J. T. Thayer and W. Ruml, "Faster than weighted A*: An optimistic approach to bounded suboptimal search.," in *ICAPS*, 2008.

[101] V. A. Traag, L. Waltman, and N. J. Van Eck, "From Louvain to Leiden: Guaranteeing well-connected communities," *Scientific reports*, vol. 9, no. 1, pp. 1–12, 2019.

[102] J. Tuero and M. Buro, "Bayes DistNet - a robust neural network for algorithm runtime distribution predictions," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, 2021.

[103] J. Tuero, M. Buro, and L. Lelis, "Subgoal-guided policy heuristic search with learned subgoals," in *Forty-second International Conference on Machine Learning*, 2025.

[104] J. Tuero, M. Buro, L. Orseau, and L. Lelis, "Structure-induced information for rerooting Levin tree search," in *Forty-third International Conference on Machine Learning*, 2026.

[105] A. Van Den Oord, O. Vinyals, et al., "Neural discrete representation learning," *Advances in Neural Information Processing Systems*, 2017.

[106] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3, pp. 279–292, 1992.

[107] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3, pp. 229–256, 1992.

[108] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "SATzilla: Portfolio-based algorithm selection for SAT," *Journal of artificial intelligence research*, vol. 32, pp. 565–606, 2008.

[109] V. Zambaldi et al., "Relational deep reinforcement learning," *arXiv preprint arXiv:1806.01830*, 2018.

[110] M. Zawalski et al., "Fast and precise: Adjusting planning horizon with adaptive subgoal search," *arXiv preprint arXiv:2206.00702*, 2022.

[111] X. Zeng, H. Peng, A. Li, C. Liu, L. He, and P. S. Yu, "Hierarchical state abstraction based on structural information principles," *arXiv preprint arXiv:2304.12000*, 2023.

[112] X. Zeng, H. Peng, D. Su, and A. Li, "Hierarchical decision making based on structural information principles," *Journal of Machine Learning Research*, vol. 26, no. 182, pp. 1–55, 2025.

[113] L. Zhang, G. Yang, and B. C. Stadie, "World model as a graph: Learning latent landmarks for planning," in *International Conference on Machine Learning*, PMLR, 2021.

[114] Q. Zhang, Y. Yang, J. Ruan, X. Xiong, D. Xing, and B. Xu, "Balancing exploration and exploitation in hierarchical reinforcement learning via latent landmark graphs," in *2023 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2023.

[115] W. Zhang, *State-space search: Algorithms, complexity, extensions, and applications*. Springer Science & Business Media, 1999.

Appendix A

Supplementary Material for Chapter 3

A.1 multiTS Bounds

Theorem 3.3.1 (Orseau et al. [78], Theorem 4, adapted)**.** *If $\pi^+_{d_{\max}} > 0$, the expected number of sampled transitions multiTS requires before returning a (random) first-hitting goal node $n^* \in G_{d_{\max}} \subseteq \mathcal{N}^g$ is bounded by*

$$\mathbb{E}\left[L(\text{multiTS}, n^*)\right] \leq \frac{d_{\max}}{\pi^+_{d_{\max}}}. \tag{3.5}$$

If $\pi^+_{d_{\max}} = 0$, no finite bound of this form is possible.

Proof. Consider a single call to `sample_trajectory`$(d_{\max})$ given in Algorithm 3. By the sampling procedure in Algorithm 3 and its repeated use in Algorithm 4, one trial follows the policy π from the root and stops as soon as a target node in $G_{d_{\max}}$ is encountered, or else when the depth bound is exhausted.

For each $n \in G_{d_{\max}}$, let E_n denote the event that the trial terminates at n. These events are disjoint, because a single sampled trajectory can terminate at only one first target node. Moreover, by definition of the policy over action sequences,

$$\Pr(E_n) = \pi(n).$$

Hence, the probability that one trial succeeds is

$$p = \sum_{n \in G_{d_{\max}}} \Pr(E_n) = \sum_{n \in G_{d_{\max}}} \pi(n) = \pi^+_{d_{\max}}.$$

Let K be the number of independent trials until the first successful one. Since each trial is an independent sample from the same policy and each succeeds with probability $p = \pi^+_{d_{\max}}$, the random variable K follows a geometric distribution with parameter $\pi^+_{d_{\max}}$. Therefore,

$$\mathbb{E}[K] = \frac{1}{\pi^+_{d_{\max}}}.$$

Now, consider the cost of each trial under the sampled-transition convention used in Section 3.3. A failed trial runs until the cutoff and therefore samples at most $d_{\max}$ transitions, while a successful trial halts as soon as a target node is reached and therefore samples at most $d_{\max}$ transitions. Thus, if N denotes the total number of sampled transitions until success, then

$$N \leq K d_{\max}.$$

Taking expectation gives

$$\mathbb{E}[N] \leq d_{\max}\mathbb{E}[K] = \frac{d_{\max}}{\pi^+_{d_{\max}}}.$$

■

A.2 LubyTS Bounds

Orseau et al. [78] provide a bound on the expected running time when using a restart strategy based on A6519.

Lemma A.2.1 (Orseau et al. [78], Theorem 5)**.** *For all distributions p over halting times, the expected running time of the restarting strategy based on A6519 is bounded by*

$$\min_t t + \frac{t}{q(t)}\left(\log_2 \frac{t}{q(t)} + 6.1\right),$$

where q is the cumulative distribution of p.

The proof is given in [78]. This result can then be used to provide a bound on the expected number of sampled transitions LubyTS requires before finding a solution node.

Theorem 3.3.2 (Orseau et al. [78], Theorem 6, adapted)**.** *The expected number of sampled transitions LubyTS requires before reaching a (random) first-hitting goal node $n^* \in G_d \subseteq \mathcal{N}^g$ is bounded by*

$$\mathbb{E}[L(\text{LubyTS}, n^*)] \leq \min_{\substack{d \in \mathbb{N}_1 \\ \pi_d^+ > 0}} d + \frac{d}{\pi_d^+} \left(\log_2 \frac{d}{\pi_d^+} + 6.1 \right). \tag{3.6}$$

If there is no d such that $\pi_d^+ > 0$, then the policy assigns zero probability to every finite-depth goal-reaching trajectory, and the theorem provides no finite guarantee.

Proof. We follow a similar argument as in Theorem 3.3.1, except that instead of fixing a single depth bound in advance, we wrap the sampling process into the universal restart strategy A6519 from Lemma A.2.1.

Define a randomized program ρ as follows: starting from the root, repeatedly sample actions according to π until a target node is reached. Let $p(t)$ denote the probability that ρ halts at time t. By the same disjoint-events argument in Theorem 3.3.1, halting at time t means terminating at some target node $n \in \mathcal{N}^g$ with $d(n) = t$, thus

$$p(t) = \sum_{\substack{n \in \mathcal{N}^g \\ d(n)=t,\ \text{anc}(n) \cap \mathcal{N}^g = \emptyset}} \pi(n).$$

Let $q(d)$ be the cumulative halting-time distribution:

$$q(d) := \sum_{t=1}^{d} p(t).$$

Substituting the expression for $p(t)$, we obtain

$$q(d) = \sum_{t=1}^{d} \sum_{\substack{n \in \mathcal{N}^g \\ d(n)=t,\ \text{anc}(n) \cap \mathcal{N}^g = \emptyset}} \pi(n) = \sum_{n \in G_d} \pi(n) = \pi_d^+.$$

Lemma A.2.1 now applies directly, with π_d^+ substituted for $q(d)$. ■

A.3 LevinTS

A.3.1 LevinTS Bounds

Theorem 3.4.4 (Orseau et al. [78], Theorem 3, adapted)**.** *The number of node expansions LTS requires before reaching the goal node with minimal Levin cost $n^* \in \mathcal{N}^g$ is bounded by*

$$L(\text{LTS}, n^*) \leq 1 + \frac{d(n^*)}{\pi(n^*)}. \tag{3.8}$$

Proof. Let C be the cost of the goal node $n^* \in \mathcal{N}^g$ expanded at time T, and let $\mathcal{N}_C$ be the set of descendants of the root node n_1 with cost at most C. Then,

$$\begin{aligned} |\mathcal{N}_C| &= |\{n : \tfrac{d}{\pi}(n_1 \prec n) \leq C\}| \\ &\leq \sum_{n \in \mathcal{L}(\mathcal{N}_C)} d(n) \\ &\leq \sum_{n \in \mathcal{L}(\mathcal{N}_C)} \pi(n) C \\ &\leq C \end{aligned}$$

Therefore, the number of nodes expanded by LTS is bounded by $|\{n_1\} \cup \mathcal{N}_C| \leq 1 + C$. ■

A.3.2 LevinTS State Pruning

Theorem 3.4.5 (Orseau et al. [78], Theorem 2, adapted)**.** *Assume the search problem is deterministic and the policy is state-based. Let $n, m \in \mathcal{N}$ be two nodes that represent the same state s. Suppose that*

$$d(m) \leq d(n) \qquad \textit{and} \qquad \pi(m) \geq \pi(n).$$

Then n can be safely pruned for LTS: for every descendant $n' \in \text{desc}(n)$, there exists a corresponding descendant $m' \in \text{desc}(m)$, obtained by following the same action suffix from m that leads from n to n', such that m' and n' represent the same state and $\frac{d}{\pi}(m') \leq \frac{d}{\pi}(n')$. In particular, if $n', m' \in \mathcal{N}^g$, pruning n cannot remove a solution whose $\frac{d}{\pi}$ cost is smaller than all solutions still reachable through m.

Proof. Let $n' \in \mathrm{desc}(n)$ be arbitrary, and let $a_{1:k} = (a_1, \ldots, a_k)$ be the action sequence from n to n', where $k = d(n') - d(n) \geq 0$. Since n and m represent the same state s and the search problem is deterministic, applying the same action sequence $a_{1:k}$ from m yields a unique descendant $m' \in \mathrm{desc}(m)$ with n' and m' representing the same state s'. Moreover, because the policy is conditioned only on the current state, the suffix path probability from these equal states is the same. Hence

$$\pi(n' \mid n) = \pi(m' \mid m).$$

Let this common value be denoted by q. Then

$$\pi(n') = \pi(n)\, q \qquad \text{and} \qquad \pi(m') = \pi(m)\, q.$$

By assumption, $\pi(m) \geq \pi(n)$, so

$$\pi(m') \geq \pi(n').$$

Likewise,

$$d(m') = d(m) + k \leq d(n) + k = d(n').$$

Combining these inequalities gives

$$\tfrac{d}{\pi}(m') = \frac{d(m')}{\pi(m')} \leq \frac{d(n')}{\pi(n')} = \tfrac{d}{\pi}(n').$$

Thus every descendant of n, and in particular every goal descendant of n, has a corresponding descendant of m with the same represented state and no larger Levin cost. Pruning n therefore cannot remove a solution that is strictly better, in Levin cost, than all solutions still reachable through m. ■

Algorithm 15 depicts the *lazy* state pruning check, where nodes are unconditionally generated and are checked for pruning when the node is removed from OPEN. If the node can be pruned, then the search continues to the next iteration. If the node cannot be pruned, then it is added to CLOSED and the rest of the `BestFirstSearch` procedure continues. The overhead for state pruning is a map where the keys are the previously expanded states and the values are the (d, π)-pairs.

Algorithm 15 LTS State Prune Check

```
1: function CAN_PRUNE(n, T)
2:     ▷ T maps states to (d, π)-pairs ◁
3:     (d_s, π_s) ← (∞, 0)
4:     if s(n) in T then
5:         (d_s, π_s) ← T[s(n)]
6:     if d_s ≤ d(n) and π_s ≥ π(n) then
7:         return true                          ▷ Safely prune
8:     T[s(n)] ← (d(n), π(n))                   ▷ Save better result
9:     return false
```

A.4 PHS

A.4.1 PHS Bounds

Theorem 3.4.7 (Orseau and Lelis [79], Theorem 1, adapted)**.** *Let ℓ be a non-negative expansion-loss function and let $\eta(\cdot) \geq 1$ be any heuristic factor. Let $n^* \in \arg\min_{n \in \mathcal{N}^g} \varphi^+(n)$ be a finite-cost goal node. Then the cumulative expansion loss incurred by PHS before returning a solution is bounded by*

$$L(\mathrm{PHS}, n^*) \leq \frac{g_\ell(n^*)}{\pi(n^*)}\, \eta^+(n^*) \sum_{n' \in \mathcal{L}_\varphi(n^*)} \frac{\pi(n')}{\eta^+(n')}. \tag{3.13}$$

Proof. Since $\varphi^+(n)$ is non-decreasing from parent to child, the set $\mathcal{N}_\varphi(n)$ is a rooted-prefix closed tree.

We first show that PHS expands no node outside $\mathcal{N}_\varphi(n)$ before expanding n. Until n is expanded, there is always some frontier node u on the root-to-n path. Since $u \preceq n$, we have

$$\varphi_{\mathrm{PHS}}(u) \leq \varphi^+(u) \leq \varphi^+(n).$$

Therefore, while n remains unexpanded, PHS has a frontier node of φ_{PHS}-value at most $\varphi^+(n)$ available on the path to n.

Now suppose, for contradiction, that PHS expands a node $v \notin \mathcal{N}_\varphi(n)$ before expanding n. Since $v \notin \mathcal{N}_\varphi(n)$, there exists an ancestor $z \preceq v$ such that

$$\varphi_{\mathrm{PHS}}(z) > \varphi^+(n).$$

The node z must be expanded before v can be generated or expanded. But at the time z is selected, the path to n still contains a frontier node u with

$\varphi_{\mathrm{PHS}}(u) \leq \varphi^+(n) < \varphi_{\mathrm{PHS}}(z)$, contradicting the best-first ordering of PHS. Hence every node expanded before n belongs to $\mathcal{N}_\varphi(n)$, and so

$$L(\mathrm{PHS}, n) \leq L(\mathcal{N}_\varphi(n)).$$

Moreover, every node in $\mathcal{N}_\varphi(n)$ lies on a root-to-leaf path to some leaf in $\mathcal{L}_\varphi(n)$, so

$$\mathcal{N}_\varphi(n) = \bigcup_{n' \in \mathcal{L}_\varphi(n)} \mathrm{anc}_+(n').$$

Therefore,

$$L\left(\mathcal{N}_\varphi(n)\right) = L\left(\bigcup_{n' \in \mathcal{L}_\varphi(n)} \mathrm{anc}_+(n')\right) \leq \sum_{n' \in \mathcal{L}_\varphi(n)} L(\mathrm{anc}_+(n')) = \sum_{n' \in \mathcal{L}_\varphi(n)} g_\ell(n').$$

Using

$$\varphi^+(n') = \eta^+(n')\frac{g_\ell(n')}{\pi(n')} \leq \varphi^+(n),$$

we obtain

$$g_\ell(n') \leq \varphi^+(n)\frac{\pi(n')}{\eta^+(n')}.$$

Summing over $n' \in \mathcal{L}_\varphi(n)$ gives

$$L(\mathrm{PHS}, n) \leq \varphi^+(n) \sum_{n' \in \mathcal{L}_\varphi(n)} \frac{\pi(n')}{\eta^+(n')}.$$

Since this holds for any n, it holds in particular for the returned solution n^*. Substituting

$$\varphi^+(n^*) = \eta^+(n^*)\frac{g_\ell(n^*)}{\pi(n^*)}$$

yields the stated bound. ■

Corollary 3.4.8 (Orseau and Lelis [79], Corollary 2, adapted)**.** *From Theorem 3.4.7, if $\eta(n) \equiv 1$, then*

$$L(\mathrm{PHS}, n^*) \leq \frac{g_\ell(n^*)}{\pi(n^*)}. \tag{3.14}$$

Proof. If $\eta(n) \equiv 1$ for all n, then $\eta^+(n) \equiv 1$ as well, and Theorem 3.4.7 simplifies to

$$L(\mathrm{PHS}, n^*) \leq \frac{g_\ell(n^*)}{\pi(n^*)} \sum_{n' \in \mathcal{L}_\varphi(n^*)} \pi(n').$$

Since the leaf set $\mathcal{L}_\varphi(n^*)$ is prefix-free, $\sum_{n' \in \mathcal{L}_\varphi(n^*)} \pi(n') \leq 1$, and therefore

$$L(\mathrm{PHS}, n^*) \leq \frac{g_\ell(n^*)}{\pi(n^*)}.$$

■

A.4.2 PHS State Pruning

Theorem 3.4.9 (Orseau and Lelis [79], Appendix G, adapted)**.** *Assume the search problem is deterministic, the policy is state-based, and both the expansion-loss function $\ell(\cdot)$ and heuristic factor $\eta(\cdot)$ are state-based: for all nodes $u, v \in \mathcal{N}$, if u and v represent the same state s, then $\ell(u) = \ell(v)$ and $\eta(u) = \eta(v)$. Let $m, n \in \mathcal{N}$ be two nodes that represent the same state s. Suppose that*

$$\varphi_{\mathrm{PHS}}(m) \leq \varphi_{\mathrm{PHS}}(n) \qquad \text{and} \qquad \pi(m) \geq \pi(n).$$

Then n is safely pruned for PHS: for every descendant $n' \in \mathrm{desc}(n)$, there exists a corresponding descendant $m' \in \mathrm{desc}(m)$ such that m' and n' represent the same state s' and $\varphi_{\mathrm{PHS}}(m') \leq \varphi_{\mathrm{PHS}}(n')$.

Proof. Let $n' \in \mathrm{desc}(n)$ be arbitrary. Since n and m represent the same state s and the search problem is deterministic, applying from m the same suffix action sequence that leads from n to n' yields a unique descendant $m' \in \mathrm{desc}(m)$ such that n' and m' represent the same state s'. Because the policy is state-based and the two suffixes pass through the same sequence of represented states, the suffix path probabilities are equal:

$$\pi(m' \mid m) = \pi(n' \mid n).$$

Let this common suffix probability be q. Similarly, because the expansion loss is state-based, the cumulative expansion loss incurred along the two suffixes is the same. Let this common suffix loss be $\Delta \geq 0$. Then

$$\pi(m') = \pi(m)q, \qquad \pi(n') = \pi(n)q,$$

and

$$g_\ell(m') = g_\ell(m) + \Delta, \qquad g_\ell(n') = g_\ell(n) + \Delta.$$

The zero-probability cases are handled by the infinite-cost convention, so consider the nontrivial case in which the relevant costs are finite. Since m and n represent the same state and η is state-based, we have $\eta(m) = \eta(n)$. Therefore,

$$\phi_{\text{PHS}}(m) \leq \phi_{\text{PHS}}(n) \quad \Longrightarrow \quad \frac{g_\ell(m)}{\pi(m)} \leq \frac{g_\ell(n)}{\pi(n)}.$$

Moreover, $\pi(m) \geq \pi(n)$ implies

$$\frac{\Delta}{\pi(m)} \leq \frac{\Delta}{\pi(n)}.$$

Combining these two inequalities gives

$$\frac{g_\ell(m')}{\pi(m')} = \frac{g_\ell(m) + \Delta}{\pi(m)q} = \frac{1}{q}\left(\frac{g_\ell(m)}{\pi(m)} + \frac{\Delta}{\pi(m)}\right) \leq \frac{1}{q}\left(\frac{g_\ell(n)}{\pi(n)} + \frac{\Delta}{\pi(n)}\right) = \frac{g_\ell(n')}{\pi(n')}.$$

Finally, since m' and n' represent the same state and η is state-based, $\eta(m') = \eta(n')$. Hence

$$\varphi_{\text{PHS}}(m') = \eta(m')\frac{g_\ell(m')}{\pi(m')} \leq \eta(n')\frac{g_\ell(n')}{\pi(n')} = \varphi_{\text{PHS}}(n').$$

Thus every descendant of n has a corresponding descendant of m with the same represented state and no larger PHS value. Therefore n may be safely pruned. ■

Algorithm 16 depicts the *lazy* state pruning check, where nodes are unconditionally generated and are checked for pruning when the node is removed from OPEN. If the node can be pruned, then the search continues to the next iteration. If the node cannot be pruned, then it is added to CLOSED and the rest of the `BestFirstSearch` procedure continues. The overhead for state pruning is a map where the keys are the previously expanded states and the values are retained $(\varphi_{\text{PHS}}, \pi)$-pairs. The pruning condition above applies to PHS instantiations whose heuristic factor is state-based. It should not be applied automatically to PHS*, since η_{PHS^*} depends on path-dependent quantities such as $g_\ell(n)$ and $\pi(n)$.

Algorithm 16 PHS State Prune Check

```
function CAN_PRUNE(n, T)
    ▷ T maps states to (φ_PHS, π)-pairs ◁
    (φ_s, π_s) ← (∞, 0)
    if s(n) in T then
        (φ_s, π_s) ← T[s(n)]
    if φ_s ≤ φ_PHS(n) and π_s ≥ π(n) then
        return true                                  ▷ Safely prune
    T[s(n)] ← (φ_PHS(n), π(n))                       ▷ Update retained pair
    return false
```

A.5 $\sqrt{\text{LTS}}$

To prove the subtask decomposition bound given in Theorem 3.4.18, we first prove some intermediate results. Define the following:

$$c_{\sqrt{\text{LTS}}}(n_k \prec n) = \frac{1}{w_k} \frac{d}{\pi}(n_k \prec n), \tag{A.1}$$

$$c_{\sqrt{\text{LTS}}}(n) = \min_{n_1 \preceq n_k \prec n} c_{\sqrt{\text{LTS}}}(n_k \prec n). \tag{A.2}$$

A.5.1 $\sqrt{\text{LTS}}$ Simple Bound

Lemma A.5.1 (Orseau et al. [77], adapted)**.** *For every finite cost bound $C \geq 0$, at step T of the $\sqrt{\text{LTS}}$ search, the number of visited nodes of cost at most C excluding the root n_1 is bounded by*

$$|\{n_t : t \leq T, c_{\sqrt{\text{LTS}}}(n_t) \leq C\}| \leq w_{<T} C. \tag{A.3}$$

Proof. Let $A = |\{n_t : t \leq T, c_{\sqrt{\text{LTS}}}(n_t) \leq C\}|$. Then,

$$
\begin{aligned}
A &= \left| \{n_t : 1 < t \leq T, \min_{n_1 \preceq n_k \prec n_t} c_{\sqrt{\text{LTS}}}(n_k \prec n_t) \leq C\} \right| && (1)\\
&= \left| \bigcup_{n_k : k < T} \{n_t : t \leq T, n_1 \preceq n_k \prec n_t, c_{\sqrt{\text{LTS}}}(n_k \prec n_t) \leq C\} \right| && (2)\\
&= \left| \bigcup_{k<T} \{n_t : t \leq T, c_{\sqrt{\text{LTS}}}(n_k \prec n_t) \leq C\} \right| && (3)\\
&\leq \sum_{k<T} \left| \{n_t : t \leq T, c_{\sqrt{\text{LTS}}}(n_k \prec n_t) \leq C\} \right| && (4)\\
&= \sum_{k<T} \left| \{n_t : t \leq T, \frac{1}{w_k} \tfrac{d}{\pi}(n_k \prec n_t) \leq C\} \right| && (5)\\
&= \sum_{k<T} \left| \{n_t : t \leq T, \tfrac{d}{\pi}(n_k \prec n_t) \leq w_k C\} \right| \\
&\leq \sum_{k<T} \left| \{n : \tfrac{d}{\pi}(n_k \prec n_t) \leq w_k C\} \right| && (6)\\
&\leq \sum_{n_k : k<T} w_k C = w_{<T} C, && (7)
\end{aligned}
$$

with

(1) Using Equation A.2,

(2) Using the logical fact that $\min_k a_k \leq C \Leftrightarrow \exists k$ such that $a_k \leq C$, and expressing this existential condition set-theoretically as a union over the ancestor-indexed sets,

(3) Dropping the explicit ancestor condition; if n_t is not a descendant of n_k then $c_{\sqrt{\text{LTS}}}(n_k \prec n_t) = \infty \not\leq C$,

(4) Union bound,

(5) Using Equation A.1,

(6) Relax from visited nodes n_t such that $t \leq T$, to all nodes n,

(7) Using the self-counting property (Equation 3.20).

■

Theorem A.5.2 (Orseau et al. [77], adapted)**.** *The number T of nodes visited by $\sqrt{\text{LTS}}$ upon visiting the node n_T is such that*

$$T \leq 1 + w_{<T} \max_{n_1 \prec n \preceq n_T} c_{\sqrt{\text{LTS}}}(n).$$

Equivalently,

$$T \leq 1 + \max_{n_1 \prec n \preceq n_T} \min_{n_k \prec n} \frac{w_{<T}}{w_k} \frac{d}{\pi}(n_k \prec n).$$

Proof. To bound the number of node T, we need to first find a cost bound C such that every node visited before n_T has cost at most C. We will bound the nodes visited by $\sqrt{\text{LTS}}$ excluding the root (since $c_{\sqrt{\text{LTS}}}(n_1) = \infty$), then will include it after. Since $c_{\sqrt{\text{LTS}}}$ is not monotone along a path, $C \neq c_{\sqrt{\text{LTS}}}(n_T)$ is general. A similar technique used in Equation 3.11 for the proof of Theorem 3.4.7 allows for a monotone version to be used for analysis Let

$$C = \max_{n_1 \prec n \preceq n_T} c_{\sqrt{\text{LTS}}}(n)$$

be the largest $\sqrt{\text{LTS}}$ cost that appears on the root-to-n_T path, excluding the root, and let $n_{\max}$ be the deepest ancestor on that path to n_T that has cost at most C. Best-first search always visits nodes in increasing cost order. Since every node on the root-to-$n_{\max}$ path has cost at most C, there is always a frontier node on that path with cost at most C until $n_{\max}$ is visited. Hence no node with cost greater than C can be visited before $n_{\max}$.

If $n_{\max} = n_T$, then we are done as the target node has already been reached with cost C. Otherwise, if $n_{\max} \prec n_T$, then there are nodes which were visited between $n_{\max}$ and n_T. Once `BestFirstSearch` visits $n_{\max}$, the search queue contains at least one node on the desired path whose cost is $\leq C$. Therefore, `BestFirstSearch` cannot choose a node of cost greater than C, because it always chooses a minimum cost node from the queue. By induction, n_T is visited before any node of cost larger than C is visited.

Therefore, the root n_1 is visited first regardless of cost, and every other

visited node up to time T has cost at most C. Therefore,

$$\begin{aligned} T &= |\{n_1\}| + |\{n_t : 1 < t \le T, c_{\sqrt{\text{LTS}}}(n_t) \le C\}| \\ &\le 1 + w_{<T} C && \text{(Lemma A.5.1)} \\ &\le 1 + w_{<T} \max_{n_1 \prec n \preceq n_T} c_{\sqrt{\text{LTS}}}(n). && \text{(Definition of } C\text{)} \end{aligned}$$

■

A.5.2 $\sqrt{\text{LTS}}$ Subtask Decomposition Bound

Theorem 3.4.18 ($\sqrt{\text{LTS}}$ subtask bound, Orseau et al. [77], Corollary 12, adapted)**.** *Let $\mathcal{D}(n^*)$ be the set of all such subtask decompositions of n^*. For $D \in \mathcal{D}(n^*)$, the selected ancestors of n^* are expanded at steps $T_1, T_2, \ldots, T_{|D|}$, where necessarily $T_1 = 1$ and $T_{|D|} = T$. Then the number of search steps T that $\sqrt{\text{LTS}}$ takes before visiting n^* is bounded by*

$$T \le 1 + \min_{D \in \mathcal{D}(n^*)} \max_{i < |D|} \frac{w_{<T}}{w_{T_i}} \frac{d}{\pi}(n_{T_i} \prec n_{T_{i+1}}), \tag{3.28}$$

where $w_{<T} = \sum_{j<T} w_j$ is the cumulative rerooting weight of the nodes expanded up to step T (excluded). The bound is finite for decompositions in which each selected segment $n_{T_i} \prec n_{T_{i+1}}$ has positive suffix probability $\pi(n_{T_{i+1}} \mid n_{T_i}) > 0$ and positive rerooting weight $w_{T_i} > 0$.

Equivalently,

$$T \le 1 + w_{<T} \max_{i<m} c_{\sqrt{\text{LTS}}}(n_{T_i} \prec n_{T_{i+1}}).$$

Proof. Let $n_{\max} \in \arg\max_{n_1 \prec n \preceq n_T} c_{\sqrt{\text{LTS}}}(n)$ by the node on the root-to-n_T path with the largest $\sqrt{\text{LTS}}$ cost. Consider an arbitrary subtask decomposition $n_1 = n_{T_1} \prec n_{T_2} \prec \cdots \prec n_{T_m} = n_T$ of n_T. For the subtask decomposition from n_1 to n_T, there exists some index j such that $n_{\max}$ lies in segment j:

$$n_{T_j} \prec n_{\max} \preceq n_{T_{j+1}}.$$

Then,

$$\begin{aligned}
\max_{n_1 \prec n \preceq n_T} c_{\sqrt{\text{LTS}}}(n) &= c_{\sqrt{\text{LTS}}}(n_{\max}) && (1)\\
&= \min_{n_1 \preceq n_k \prec n_{\max}} c_{\sqrt{\text{LTS}}}(n_k \prec n_{\max}) && (2)\\
&\leq c_{\sqrt{\text{LTS}}}(n_{T_j} \prec n_{\max}) && (3)\\
&\leq c_{\sqrt{\text{LTS}}}(n_{T_j} \prec n_{T_{j+1}}) && (4)\\
&\leq \max_{i<m} c_{\sqrt{\text{LTS}}}(n_{T_i} \prec n_{T_{i+1}}). && (5)
\end{aligned}$$

Then using the above inequality into Theorem A.5.2,

$$\begin{aligned}
T &\leq 1 + w_{<T} \max_{n_1 \prec n \preceq n_T} c_{\sqrt{\text{LTS}}}(n)\\
&\leq 1 + w_{<T} \max_{i<m} c_{\sqrt{\text{LTS}}}(n_{T_i} \prec n_{T_{i+1}}),
\end{aligned}$$

with

(1) By definition of $n_{\max}$,

(2) By Equation A.2,

(3) Upper-bound the minimum by evaluating it at one specific ancestor, namely n_{T_j},

(4) $n_{\max} \preceq n_{T_{j+1}}$, and $\frac{d}{\pi}(n_{T_j} \prec \cdot)$ is monotonically increasing from parent to child, and therefore $c_{\sqrt{\text{LTS}}}(n_{T_j} \prec \cdot)$ is also monotone from parent to child,

(5) Upper-bound by the maximum over all segments.

■

Corollary 3.4.23 (Robust $\sqrt{\text{LTS}}$ guarantee, Orseau et al. [77], Corollary 17, adapted)**.** *For every subtask decomposition, where the selected ancestors of n^* are expanded at steps $T_1, T_2, \ldots, T_{|D|}$, where necessarily $T_1 = 1$ and $T_{|D|} = T$, the number of search steps T that $\sqrt{\text{LTS}}$ takes to visit n^* when using robust rerooting weights is bounded by*

$$T \leq 1 + \left[1 + \ln\left(\frac{\tilde{w}_{<T}}{\tilde{w}_1}\right)\right] \max_{i<m} \frac{\tilde{w}_{\leq T_i}}{\tilde{w}_{T_i}} \frac{d}{\pi}(n_{T_i} \prec n_{T_{i+1}}). \tag{3.30}$$

The statement assumes the relevant reparameterized weights in the denominators are positive. If $\widetilde{w}_{T_i} = 0$ for a selected subtask root, that decomposition gives an infinite contribution to the bound.

Proof. To use $\tilde{w}$ in the subtask decomposition bound, we need to find how w_t, $w_{<T}$, and w_{T_I} relate to $\tilde{w}$. First, observe that

$$\begin{aligned} w_t &= \frac{\tilde{w}_t}{\tilde{w}_{\leq t}} = \frac{\tilde{w}_{\leq t} - \tilde{w}_{<t}}{\tilde{w}_{\leq t}} = 1 - \frac{\tilde{w}_{<t}}{\tilde{w}_{\leq t}} = 1 - \frac{1}{\tilde{w}_{\leq t}/\tilde{w}_{<t}} \\ &\leq \ln\left(\frac{\tilde{w}_{\leq t}}{\tilde{w}_{<t}}\right), \end{aligned}$$

using the fact that for $x \geq 1$,

$$1 - \frac{1}{x} \leq \ln(x).$$

Next, we use the telescoping property:

$$\begin{aligned} w_{<T} &= w_1 + \sum_{t=2}^{T-1} w_t \\ &\leq w_1 + \sum_{t=2}^{T-1} \ln\left(\frac{\tilde{w}_{\leq t}}{\tilde{w}_{\leq (t-1)}}\right) \qquad (1) \\ &= w_1 + \ln\left(\frac{\tilde{w}_{\leq 2}}{\tilde{w}_{\leq 1}}\right) + \ln\left(\frac{\tilde{w}_{\leq 3}}{\tilde{w}_{\leq 2}}\right) + \cdots + \ln\left(\frac{\tilde{w}_{\leq (T-2)}}{\tilde{w}_{\leq (T-3)}}\right) + \ln\left(\frac{\tilde{w}_{\leq (T-1)}}{\tilde{w}_{\leq (T-2)}}\right) \\ &= 1 + \ln\left(\frac{\tilde{w}_{<T}}{\tilde{w}_1}\right), \qquad (2) \end{aligned}$$

where (1) is due to $\tilde{w}_{<t} = \tilde{w}_{\leq (t-1)}$, and (2) is due to $w_1 = 1$ and the telescoping sum. Finally,

$$w_{T_i} = \frac{\tilde{w}_{T_i}}{\tilde{w}_{\leq T_i}} \qquad \Rightarrow \qquad \frac{1}{w_{T_i}} = \frac{\tilde{w}_{\leq T_i}}{\tilde{w}_{T_i}}$$

The factor in the bound now becomes

$$\frac{w_{<T}}{w_{T_i}} = w_{<T}\,\frac{\tilde{w}_{\leq T_i}}{\tilde{w}_{T_i}} \leq \left[1 + \ln\left(\frac{\tilde{w}_{<T}}{\tilde{w}_1}\right)\right]\,\frac{\tilde{w}_{\leq T_i}}{\tilde{w}_{T_i}}$$

■

A.5.3 $\sqrt{\text{LTS}}$ State Pruning

Definition 3.4.20 (Safely prunable node)**.** A node n_t visited at step t is safely prunable if there exists a node n_j with $j < t$ such that $s(n_t) = s(n_j)$ and for each non-empty action sequence y such that $n_{\underset{\sim}{t}}$ (resp. $n_{\underset{\sim}{j}}$) is the descendant of n_t (resp. n_j) following y, with $j < \underset{\sim}{j}$ and $t < \underset{\sim}{t}$, then $\varphi_{\sqrt{\text{LTS}}}(n_{\underset{\sim}{j}}) \leq \varphi_{\sqrt{\text{LTS}}}(n_{\underset{\sim}{t}})$.

Theorem 3.4.21. *Assume the search problem is deterministic, the policy is state-based, and the rerooting weights are state-based: for all nodes $n_i, n_j \in \mathcal{N}$, if $s(n_i) = s(n_j)$, then $w_i = w_j$. Let n_t be a node visited at step t. Suppose that there exists a node n_j with $j < t$ such that $s(n_j) = s(n_t)$ and that the following holds: for every ancestor $n_{\widetilde{t}} \prec n_t$ with $\widetilde{t} < t$, there exists an ancestor $n_{\widetilde{j}} \prec n_j$ with $\widetilde{j} < j$ satisfying*

$$d(n_j) - d(n_{\widetilde{j}}) \leq d(n_t) - d(n_{\widetilde{t}})\,,$$
$$\text{and} \qquad w_{\widetilde{j}}\,\pi(n_j \mid n_{\widetilde{j}}) \geq w_{\widetilde{t}}\,\pi(n_t \mid n_{\widetilde{t}}).$$

Then, n_t is safely prunable for $\sqrt{\text{LTS}}$. That is, for every descendant $n_t \prec n_{\underset{\sim}{t}}$, if $n_{\underset{\sim}{j}}$ is the descendant of n_j obtained by following the same action suffix y from n_j as $n_{\underset{\sim}{t}}$ follows from n_t, then $n_{\underset{\sim}{j}}$ and $n_{\underset{\sim}{t}}$ represent the same state, and

$$\varphi_{\sqrt{\text{LTS}}}(n_{\underset{\sim}{j}}) \leq \varphi_{\sqrt{\text{LTS}}}(n_{\underset{\sim}{t}}).$$

Proof. Let y be any non-empty action suffix, and let $n_{\underset{\sim}{t}}$ and $n_{\underset{\sim}{j}}$ be the descendants of n_t and n_j respectively by following y, where $t < \underset{\sim}{t}$ and $j < \underset{\sim}{j}$. Since $s(n_t) = s(n_j)$ and the problem is deterministic, $n_{\underset{\sim}{t}}$ and $n_{\underset{\sim}{j}}$ represent the same state. Let $n_{t'} \prec n_{\underset{\sim}{t}}$ be an arbitrary ancestor of $n_{\underset{\sim}{t}}$, where $t' < \underset{\sim}{t}$. We need to show that there exists an ancestor $n_{j'} \prec n_{\underset{\sim}{j}}$, where $j' < \underset{\sim}{j}$, such that

$$\frac{d(n_{\underset{\sim}{j}}) - d(n_{j'})}{w_{j'}\pi(n_{\underset{\sim}{j}} \mid n_{j'})} \leq \frac{d(n_{\underset{\sim}{t}}) - d(n_{t'})}{w_{t'}\pi(n_{\underset{\sim}{t}} \mid n_{t'})}.$$

There are two cases:

Case 1: $n_{t'} \prec n_t$**.** Since $n_{t'} \prec n_t$, the assumption of the theorem yields an ancestor $n_{j'} \prec n_j$ such that

$$d(n_j) - d(n_{j'}) \leq d(n_t) - d(n_{t'})\,,$$
$$\text{and} \qquad w_{j'}\,\pi(n_j \mid n_{j'}) \geq w_{t'}\,\pi(n_t \mid n_{t'}).$$

Because $n_{\underset{\sim}{j}}$ and $n_{\underset{\sim}{t}}$ are obtained by following the same action suffix y from equal states, the suffix contributes the same additional depth and the same

conditional path probability on both sides. Hence

$$\begin{aligned} d(n_{\underset{\sim}{j}}) - d(n_{j'}) &= (d(n_j) - d(n_{j'})) + |y| \\ &\leq (d(n_t) - d(n_{t'})) + |y| \\ &= d(n_{\underset{\sim}{t}}) - d(n_{t'}), \end{aligned}$$

and

$$\begin{aligned} w_{j'}\,\pi(n_{\underset{\sim}{j}} \mid n_{j'}) &= w_{j'}\,\pi(n_j \mid n_{j'})\pi(n_{\underset{\sim}{j}} \mid n_j) \\ &\geq w_{t'}\,\pi(n_t \mid n_{t'})\pi(n_{\underset{\sim}{t}} \mid n_t) \\ &= w_{t'}\,\pi(n_{\underset{\sim}{t}} \mid n_{t'}). \end{aligned}$$

Combining both of the above yields

$$\frac{d(n_{\underset{\sim}{j}}) - d(n_{j'})}{w_{j'}\pi(n_{\underset{\sim}{j}} \mid n_{j'})} \leq \frac{d(n_{\underset{\sim}{t}}) - d(n_{t'})}{w_{t'}\pi(n_{\underset{\sim}{t}} \mid n_{t'})}.$$

Case 2: $n_t \preceq n_{t'} \prec n_{\underset{\sim}{t}}$**.** In this case, $n_{t'}$ falls on the suffix from n_t to $n_{\underset{\sim}{t}}$. Let $n_{j'}$ be the corresponding node on the suffix from n_j to $n_{\underset{\sim}{j}}$, *i.e.*, the node reached after the same number of suffix actions from n_j as $n_{t'}$ is from n_t. Because the problem is deterministic, $s(n_{j'}) = s(n_{t'})$. Since rerooting weights are state-based, $w_{j'} = w_{t'}$. Also, the remaining suffix from $n_{j'}$ to $n_{\underset{\sim}{j}}$ is the same as the remaining suffix from $n_{t'}$ to $n_{\underset{\sim}{t}}$. Thus,

$$\begin{aligned} d(n_{\underset{\sim}{j}}) - d(n_{j'}) &= d(n_{\underset{\sim}{t}}) - d(n_{t'})\,, \\ \text{and} \qquad \pi(n_{\underset{\sim}{j}} \mid n_{j'}) &= \pi(n_{\underset{\sim}{t}} \mid n_{t'}). \end{aligned}$$

Therefore,

$$\frac{d(n_{\underset{\sim}{j}}) - d(n_{j'})}{w_{j'}\pi(n_{\underset{\sim}{j}} \mid n_{j'})} = \frac{d(n_{\underset{\sim}{t}}) - d(n_{t'})}{w_{t'}\pi(n_{\underset{\sim}{t}} \mid n_{t'})}.$$

Since $n_{t'} \prec n_{\underset{\sim}{t}}$ was arbitrary, we have shown that for every candidate rerooting ancestor of $n_{\underset{\sim}{t}}$, there is a corresponding candidate rerooting ancestor of $n_{\underset{\sim}{j}}$ with rooted cost no greater. Therefore,

$$\min_{n_{j'} \prec n_{\underset{\sim}{j}}} \frac{d(n_{\underset{\sim}{j}}) - d(n_{j'})}{w_{j'}\pi(n_{\underset{\sim}{j}} \mid n_{j'})} \leq \min_{n_{t'} \prec n_{\underset{\sim}{t}}} \frac{d(n_{\underset{\sim}{t}}) - d(n_{t'})}{w_{t'}\pi(n_{\underset{\sim}{t}} \mid n_{t'})},$$

that is,

$$\varphi_{\sqrt{\text{LTS}}}(n_{\underset{\sim}{j}}) \leq \varphi_{\sqrt{\text{LTS}}}(n_{\underset{\sim}{t}}).$$

Since this holds for every suffix y, Definition 3.4.20 implies that n_t is safely prunable. ■

Algorithm 17 depicts the *lazy* state pruning check, where nodes are unconditionally generated and are checked for pruning when the node is removed from OPEN. If the node can be pruned, then the search continues to the next iteration. If the node cannot be pruned, then it is added to CLOSED and the rest of the `BestFirstSearch` procedure continues. Unlike LTS and PHS, safe duplicate pruning for $\sqrt{\text{LTS}}$ is not generally characterized by a single-node dominance test. The reason is that $\varphi_{\sqrt{\text{LTS}}}(n)$ depends on the set of rerooting points available along the ancestor chain of n. Accordingly, a node can be pruned only when its entire ancestor chain is covered, in the sense of Definition 3.4.20, by the ancestor chains of the retained equal-state nodes. As a result, the overhead for state pruning can be quite costly.

Algorithm 17 $\sqrt{\text{LTS}}$ State Prune Check

```
1: function CoveredByNode(n_j, n_j', n_t, n_t')
2:     ▷ n_j' ≺ n_j and n_t' ≺ n_t ◁
3:     Δd_j = d(n_j) − d(n_j'), Δd_t = d(n_t) − d(n_t')
4:     Δπ_j = π(n_j | n_j'), Δπ_t = π(n_t | n_t')
5:     if Δd_j ≤ Δd_t and w_j' Δπ_j ≥ w_t' Δπ_t then
6:         return true
7:     return false
8: function WitnessDominates(n_j, n_t)
9:     for all n_t' ≺ n_t do
10:        found ← false
11:        for all n_j' ≺ n_j do
12:            if CoveredByNode(n_j, n_j', n_t, n_t') then
13:                found ← true
14:                break
15:        if found = false then
16:            return false
17:    return true
18: function can_prune(n_t)
19:    for all n_j ∈ PreviousNodes(s(n_t)) do
20:        if WitnessDominates(n_j, n_t) then
21:            return true
22:    PreviousNodes(s(n_t)) ← PreviousNodes(s(n_t)) ∪ {n_t}
23:    return false
```

Appendix B

Environments

All grid environments expose a channel-first state observation $\mathbf{o}(s) \in \{0,1\}^{C \times H \times W}$. Each discrete cell type has a separate binary channel, such as channels for the agent, movable boxes, goal locations, walls, and other environment-specific objects. The dimensions C, H, and W are determined by the observation schema of the corresponding environment. These state observations, rather than handcrafted instance features, are used by the convolutional models in Chapters 5, 6, and 7.

B.1 Box-World

Box-World is a grid-based environment with coloured keys and locked boxes [109]. Keys are consumed when used on a lock of the matching colour, with the agent receiving a new key matching the colour of the box. The objective

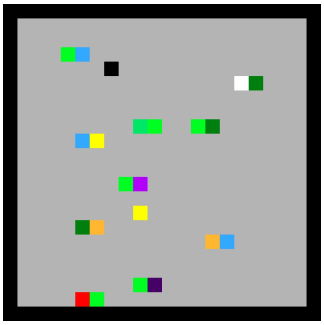


Figure B.1: A Box-World game state. The agent in black needs to open coloured locks (right pixel) to receive a coloured key (left pixel) until it gets to the goal (white).

is to open the designated goal box by opening the correct sequence of locks. If the agent opens a corresponding lock on the wrong box, the resulting scenario can become deadlocked. Box-World uses an open source problem generator [1] with a goal length of 5, and several distractor paths of length 3. A distractor path is a sequence of key and locks which can be used by the agent, but does not progress to the goal.

B.2 BoulderDash

BoulderDash is a grid-based environment that is modelled after the popular games *Boulder Dash* and *Emerald Mine*. The agent must gather a number of diamonds to unlock the exit door, and proceed to enter it. Diamonds can be inside locked sub-rooms in which the corresponding key must be gathered first. Various dirt elements exist, which disappear once the agent walks over them, and can significantly increase the state-space size.

We use a custom environment and problem generator that generate problems by randomly placing the keys, diamonds, and an exit door [2]. Varying levels of *difficulty* can be created by placing different amount of dirt elements, which increases the search space.

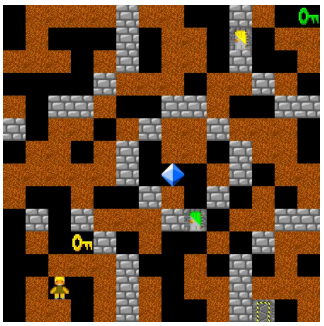


Figure B.2: A BoulderDash game state. The agent must get the green key to unlock the room containing the diamond. Once the diamond is collected, the exit in the bottom-right room will open.

[1] https://github.com/nathangrinsztajn/Box-World
[2] https://github.com/tuero/boulderdash_cpp

B.3 CraftWorld

CraftWorld is a grid-based environment where the agent is in a room with various raw materials and workbenches [5]. The agent can collect raw materials to create tools. The goal of the level is to create a particular item, which often requires multiple primary and secondary inputs.

We use a custom implementation and problem generator [3]. The problem generator is derived from the open-source level generator[4], which follows the procedure detailed by Andreas et al. (2017).

Figure B.3: A CraftWorld game state. The agent must create an iron pickaxe to get the gem so that they can craft a ring.

B.4 Sokoban

Sokoban is a grid-based environment where the agent must push boxes onto specified goal locations. The box can only be pushed; not pulled, so boxes can get stuck on walls of the grid. Sokoban is PSPACE-hard [21]. We use a custom implementation [5], and we use the Boxoban training and test problems [41].

B.5 TSP GridWorld

TSP GridWorld is a grid-based environment where the agent must visit all of the cities before returning back to the first city it visited. There are two

[3]https://github.com/tuero/craftworld_cpp_v2
[4]https://github.com/jacobandreas/psketch/tree/master
[5]https://github.com/tuero/sokoban_cpp

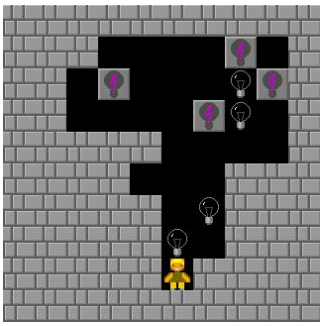


Figure B.4: A Sokoban game state. The agent must push boxes to the goal locations without getting boxes stuck in corners.

versions of this environment: the first allows the agent to revisit previous cities, and the second will deadlock if the agent revisits a previously visited city. The deadlocking version is more difficult, as it prevents trivial solutions that correspond to *sweeping* across the state-space. The TSP is NP-hard [34].

TSP uses a custom implementation and problem generator [6], which randomly places the agent starting location and the cities.

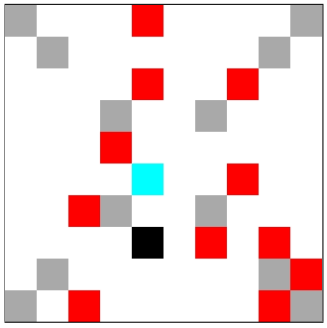


Figure B.5: A TSP GridWorld game state. The agent (black) must visit each city (red) then return back to the first city (blue). The grey boxes are non-traversable obstacles.

[6] https://github.com/tuero/tsp_cpp

Appendix C

Supplementary Material for Chapter 4

Table C.1: Comparison of DistNet and Bayes DistNet, for various numbers of observed runtimes per instance. Metrics are averaged across 10-folds, repeated for multiple seeds. From left to right: NLLH, KL-Divergence, KS-Distance, and percentage of density area outside the expected range of 0 to 1.5 times the maximum observed runtime for each instance.

Scenario	Num Samples	Model	Distribution	NLLH	KLD	D-KS	Mass
clasp_factoring	1	Bayes DistNet	InverseGaussian	0.587 ± 0.089	0.590 ± 0.060	0.495 ± 0.035	0.284 ± 0.030
			Lognormal	0.459 ± 0.123	0.508 ± 0.081	0.450 ± 0.043	0.226 ± 0.047
		DistNet	InverseGaussian	0.737 ± 0.205	0.770 ± 0.139	0.521 ± 0.042	0.295 ± 0.044
			Lognormal	1.236 ± 0.557	1.061 ± 0.383	0.590 ± 0.051	0.403 ± 0.058
clasp_factoring	2	Bayes DistNet	InverseGaussian	0.316 ± 0.180	0.419 ± 0.115	0.391 ± 0.057	0.171 ± 0.073
			Lognormal	0.113 ± 0.165	0.295 ± 0.103	0.326 ± 0.050	0.092 ± 0.062
		DistNet	InverseGaussian	0.436 ± 0.152	0.565 ± 0.106	0.454 ± 0.038	0.231 ± 0.029
			Lognormal	0.705 ± 0.137	0.691 ± 0.090	0.523 ± 0.041	0.348 ± 0.044
clasp_factoring	4	Bayes DistNet	InverseGaussian	-0.040 ± 0.060	0.203 ± 0.037	0.277 ± 0.021	0.037 ± 0.018
			Lognormal	-0.054 ± 0.051	0.196 ± 0.032	0.274 ± 0.015	0.032 ± 0.015
		DistNet	InverseGaussian	0.224 ± 0.065	0.412 ± 0.044	0.392 ± 0.026	0.180 ± 0.022
			Lognormal	0.289 ± 0.213	0.431 ± 0.133	0.414 ± 0.069	0.198 ± 0.086
clasp_factoring	8	Bayes DistNet	InverseGaussian	-0.110 ± 0.021	0.160 ± 0.016	0.251 ± 0.012	0.021 ± 0.004
			Lognormal	-0.112 ± 0.025	0.160 ± 0.017	0.253 ± 0.012	0.019 ± 0.006
		DistNet	InverseGaussian	-0.001 ± 0.202	0.275 ± 0.130	0.293 ± 0.069	0.077 ± 0.084
			Lognormal	-0.135 ± 0.022	0.192 ± 0.018	0.254 ± 0.012	0.014 ± 0.003
clasp_factoring	16	Bayes DistNet	InverseGaussian	-0.127 ± 0.023	0.149 ± 0.016	0.245 ± 0.014	0.018 ± 0.004
			Lognormal	-0.136 ± 0.022	0.145 ± 0.015	0.245 ± 0.012	0.014 ± 0.004
		DistNet	InverseGaussian	-0.140 ± 0.096	0.207 ± 0.054	0.255 ± 0.035	0.019 ± 0.044
			Lognormal	-0.156 ± 0.022	0.179 ± 0.018	0.248 ± 0.013	0.010 ± 0.002
lpg-zeno	1	Bayes DistNet	InverseGaussian	0.562 ± 0.167	0.923 ± 0.126	0.659 ± 0.060	0.174 ± 0.065
			Lognormal	0.520 ± 0.292	0.901 ± 0.208	0.661 ± 0.066	0.126 ± 0.023
		DistNet	InverseGaussian	1.779 ± 1.106	1.835 ± 0.762	0.790 ± 0.073	0.438 ± 0.087
			Lognormal	1.106 ± 0.368	1.290 ± 0.253	0.738 ± 0.065	0.488 ± 0.078
lpg-zeno	2	Bayes DistNet	InverseGaussian	0.407 ± 0.316	0.827 ± 0.220	0.636 ± 0.079	0.107 ± 0.010
			Lognormal	0.391 ± 0.369	0.823 ± 0.253	0.635 ± 0.082	0.085 ± 0.014
		DistNet	InverseGaussian	0.970 ± 0.700	1.281 ± 0.498	0.723 ± 0.088	0.333 ± 0.058
			Lognormal	0.868 ± 0.403	1.154 ± 0.276	0.714 ± 0.070	0.378 ± 0.060
lpg-zeno	4	Bayes DistNet	InverseGaussian	-0.009 ± 0.234	0.605 ± 0.150	0.572 ± 0.049	0.050 ± 0.026
			Lognormal	-0.021 ± 0.422	0.612 ± 0.257	0.565 ± 0.060	0.043 ± 0.135
		DistNet	InverseGaussian	0.126 ± 0.136	0.702 ± 0.101	0.596 ± 0.035	0.219 ± 0.026
			Lognormal	0.126 ± 0.303	0.716 ± 0.193	0.591 ± 0.057	0.143 ± 0.106
lpg-zeno	8	Bayes DistNet	InverseGaussian	-0.627 ± 0.183	0.217 ± 0.118	0.319 ± 0.139	0.003 ± 0.002
			Lognormal	-0.611 ± 0.192	0.232 ± 0.124	0.321 ± 0.145	0.002 ± 0.002
		DistNet	InverseGaussian	-0.265 ± 0.093	0.438 ± 0.079	0.429 ± 0.074	0.139 ± 0.019
			Lognormal	-0.613 ± 0.239	0.266 ± 0.159	0.327 ± 0.137	0.013 ± 0.027
lpg-zeno	16	Bayes DistNet	InverseGaussian	-0.672 ± 0.119	0.186 ± 0.075	0.298 ± 0.117	0.001 ± 0.001
			Lognormal	-0.664 ± 0.127	0.197 ± 0.080	0.297 ± 0.124	0.001 ± 0.001
		DistNet	InverseGaussian	-0.446 ± 0.120	0.350 ± 0.084	0.374 ± 0.079	0.073 ± 0.043
			Lognormal	-0.682 ± 0.160	0.234 ± 0.120	0.304 ± 0.119	0.000 ± 0.000

Table C.2: Comparison of DistNet and Bayes DistNet, for various levels of censoring, each with 8 observations per instance. Metrics are averaged across 10-folds, repeated for multiple seeds. From left to right: NLLH, KL-Divergence, KS-Distance, and percentage of density area outside the expected range of 0 to 1.5 times the maximum observed runtime for each instance.

Scenario	Censoring (%)	Model	Distribution	NLLH	KLD	D-KS	Mass
clasp_factoring	0	Bayes DistNet	InverseGaussian	-0.110 ± 0.021	0.160 ± 0.016	0.251 ± 0.012	0.021 ± 0.004
			Lognormal	-0.112 ± 0.025	0.160 ± 0.017	0.253 ± 0.012	0.019 ± 0.006
		DistNet	InverseGaussian	-0.001 ± 0.202	0.275 ± 0.130	0.293 ± 0.069	0.077 ± 0.084
			Lognormal	-0.135 ± 0.022	0.192 ± 0.018	0.254 ± 0.012	0.014 ± 0.003
clasp_factoring	20	Bayes DistNet	InverseGaussian	-0.078 ± 0.025	0.175 ± 0.017	0.257 ± 0.012	0.033 ± 0.006
			Lognormal	-0.095 ± 0.022	0.166 ± 0.016	0.254 ± 0.012	0.026 ± 0.005
		DistNet	InverseGaussian	0.087 ± 0.150	0.315 ± 0.099	0.333 ± 0.058	0.125 ± 0.064
			Lognormal	-0.103 ± 0.036	0.196 ± 0.020	0.264 ± 0.017	0.031 ± 0.015
clasp_factoring	40	Bayes DistNet	InverseGaussian	-0.055 ± 0.033	0.188 ± 0.021	0.264 ± 0.016	0.041 ± 0.010
			Lognormal	-0.087 ± 0.025	0.172 ± 0.017	0.257 ± 0.012	0.029 ± 0.007
		DistNet	InverseGaussian	0.167 ± 0.065	0.361 ± 0.040	0.372 ± 0.031	0.170 ± 0.031
			Lognormal	0.058 ± 0.179	0.285 ± 0.102	0.329 ± 0.072	0.105 ± 0.086
clasp_factoring	60	Bayes DistNet	InverseGaussian	0.059 ± 0.037	0.254 ± 0.022	0.302 ± 0.014	0.084 ± 0.014
			Lognormal	-0.037 ± 0.037	0.200 ± 0.022	0.275 ± 0.016	0.047 ± 0.013
		DistNet	InverseGaussian	0.227 ± 0.045	0.399 ± 0.030	0.409 ± 0.019	0.207 ± 0.019
			Lognormal	0.395 ± 0.221	0.487 ± 0.134	0.455 ± 0.072	0.259 ± 0.091
clasp_factoring	80	Bayes DistNet	InverseGaussian	0.372 ± 0.050	0.451 ± 0.033	0.418 ± 0.014	0.205 ± 0.020
			Lognormal	0.285 ± 0.057	0.397 ± 0.036	0.397 ± 0.018	0.164 ± 0.021
		DistNet	InverseGaussian	0.345 ± 0.072	0.488 ± 0.051	0.459 ± 0.029	0.253 ± 0.028
			Lognormal	0.663 ± 0.138	0.661 ± 0.090	0.545 ± 0.038	0.358 ± 0.045
lpg-zeno	0	Bayes DistNet	InverseGaussian	-0.627 ± 0.183	0.217 ± 0.118	0.319 ± 0.139	0.003 ± 0.002
			Lognormal	-0.611 ± 0.192	0.232 ± 0.124	0.321 ± 0.145	0.002 ± 0.002
		DistNet	InverseGaussian	-0.265 ± 0.093	0.438 ± 0.079	0.429 ± 0.074	0.139 ± 0.019
			Lognormal	-0.613 ± 0.239	0.266 ± 0.159	0.327 ± 0.137	0.013 ± 0.027
lpg-zeno	20	Bayes DistNet	InverseGaussian	-0.509 ± 0.217	0.300 ± 0.141	0.347 ± 0.135	0.019 ± 0.020
			Lognormal	-0.561 ± 0.226	0.284 ± 0.138	0.328 ± 0.153	0.007 ± 0.009
		DistNet	InverseGaussian	-0.158 ± 0.175	0.493 ± 0.122	0.465 ± 0.093	0.170 ± 0.041
			Lognormal	-0.505 ± 0.338	0.352 ± 0.186	0.343 ± 0.162	0.038 ± 0.074
lpg-zeno	40	Bayes DistNet	InverseGaussian	-0.155 ± 0.235	0.501 ± 0.158	0.446 ± 0.121	0.109 ± 0.033
			Lognormal	-0.214 ± 0.281	0.483 ± 0.184	0.432 ± 0.134	0.086 ± 0.032
		DistNet	InverseGaussian	-0.000 ± 0.226	0.579 ± 0.155	0.525 ± 0.089	0.226 ± 0.053
			Lognormal	0.139 ± 0.393	0.677 ± 0.244	0.540 ± 0.118	0.250 ± 0.117
lpg-zeno	60	Bayes DistNet	InverseGaussian	0.234 ± 0.311	0.740 ± 0.207	0.541 ± 0.119	0.192 ± 0.042
			Lognormal	0.201 ± 0.341	0.729 ± 0.225	0.529 ± 0.126	0.181 ± 0.045
		DistNet	InverseGaussian	0.194 ± 0.204	0.714 ± 0.141	0.596 ± 0.073	0.286 ± 0.046
			Lognormal	0.487 ± 0.287	0.897 ± 0.186	0.630 ± 0.084	0.359 ± 0.068
lpg-zeno	80	Bayes DistNet	InverseGaussian	0.993 ± 0.421	1.214 ± 0.277	0.684 ± 0.088	0.315 ± 0.054
			Lognormal	0.926 ± 0.391	1.179 ± 0.259	0.677 ± 0.092	0.290 ± 0.049
		DistNet	InverseGaussian	2.663 ± 1.033	2.419 ± 0.698	0.774 ± 0.060	0.507 ± 0.081
			Lognormal	1.330 ± 0.394	1.449 ± 0.260	0.757 ± 0.066	0.497 ± 0.074

Appendix D

Supplementary Material for Chapter 5

D.1 Shifted LubyTS Bound

Theorem 5.2.1. *Let $X \in \mathbb{N}_0$ be a fixed shift, and let* LubyTS_X *denote the variant of LubyTS which samples rollout i to depth $d_i = X + A6519(i)$. For $u \in \mathbb{N}_1$, define*

$$q_X(u) := \pi^+_{X+u},$$

where π^+_{X+u} is the probability that a policy-generated trajectory reaches a goal within at most $X + u$ steps. Then, under the sampled-transition convention of Section 3.3, the expected number of sampled transitions LubyTS_X *requires before returning some (random) first-hitting goal node $n^* \in G_{X+u} \subseteq \mathcal{N}^g$ is bounded by*

$$\mathbb{E}[L(\mathrm{LubyTS}_X, n^*)] \leq \min_{\substack{u \in \mathbb{N}_1 \\ q_X(u) > 0}} \left[u + \frac{u}{q_X(u)} \left(\log_2 \frac{u}{q_X(u)} + 6.1 \right) + \frac{X\, 2^{\lceil \log_2 u \rceil}}{q_X(u)} \right]. \tag{5.4}$$

Equivalently, since $2^{\lceil \log_2 u \rceil} \leq 2u$,

$$\mathbb{E}[L(\mathrm{LubyTS}_X, n^*)] \leq \min_{\substack{u \in \mathbb{N}_1 \\ q_X(u) > 0}} \left[u + \frac{u}{q_X(u)} \left(\log_2 \frac{u}{q_X(u)} + 6.1 + 2X \right) \right]. \tag{5.5}$$

If there is no u such that $q_X(u) > 0$, then the shifted schedule has no finite success-probability depth to exploit, and the theorem provides no finite guarantee.

Proof. Fix $u \in \mathbb{N}_1$, and write $q = q_X(u) = \pi^+_{X+u}$. A rollout whose unshifted A6519 component satisfies A6519$(i) \geq u$ has total depth at least $X + u$. Therefore, such a rollout reaches a goal node in G_{X+u} with probability at least q. Rollouts with $A6519(i) < u$ may also reach a goal because of the shift X, but we ignore those possible successes in the analysis. This can only increase the upper bound.

We decompose the cost of rollout $X + A6519(i)$. First consider the contribution from the A6519 part. Under the pessimistic process described above, a rollout can succeed only when $A6519(i) \geq u$, and each such rollout succeeds with probability at least q. From Lemma A.2.1, applying the A6519 universal-restart bound to this excess-depth process gives

$$\mathbb{E}[\text{excess-depth cost}] \leq u + \frac{u}{q}\left(\log_2 \frac{u}{q} + 6.1\right). \tag{D.1}$$

It remains to account for the additional shift cost X, which is incurred on every rollout. Let $\hat{u} := 2^{\lceil \log_2 u \rceil}$. By the structure of the A6519 sequence, every consecutive block of $\hat{u}$ rollouts contains at least one rollout whose A6519 component is at least $\hat{u}$, and hence at least u. Thus, conditioned on reaching such a block, the probability that the block contains a successful rollout is at least q. The expected number of blocks before success is therefore at most $1/q$, and each block contains at most $\hat{u}$ rollouts. Hence, the expected number of rollouts before success is at most

$$\frac{\hat{u}}{q} = \frac{2^{\lceil \log_2 u \rceil}}{q}.$$

Since each rollout pays the additional shift cost X, the expected shift overhead is at most

$$\frac{X\, 2^{\lceil \log_2 u \rceil}}{q}.$$

Combining the excess-depth cost and the shift overhead gives

$$\mathbb{E}[L(\text{LubyTS}_X, n^*)] \leq u + \frac{u}{q}\left(\log_2 \frac{u}{q} + 6.1\right) + \frac{X\, 2^{\lceil \log_2 u \rceil}}{q}.$$

Substituting $q = q_X(u) = \pi^+_{X+u}$ and minimizing over $u \in \mathbb{N}_1$ such that $q_X(u) > 0$ proves the first claim. The second claim follows from $2^{\lceil \log_2 u \rceil} \leq 2u$ for $u \geq 1$. ■

Appendix E

Supplementary Material for Chapter 6

E.1 Louvain Algorithm

As specified in Section 6.4.3, Louvain treats the directed generated-state graph as undirected by ignoring transition directions; all resulting clustering edges are unweighted.

A *partition* of a graph is a grouping of its nodes into mutually exclusive groups called *clusters*. The *modularity* [61, 71] of a graph measures the relative density of edges inside the clusters to outgoing edges. The modularity of a cluster Q_c is given by

$$Q_c = \frac{\Sigma_{\text{in}}^C}{2m} - \rho \left(\frac{\Sigma_{\text{tot}}^C}{2m} \right)^2, \tag{E.1}$$

where Σ_{in}^C is the total edge weight between nodes in cluster c, Σ_{tot}^C is the sum of weighted degrees of nodes in c (and therefore includes edges from c to other clusters), m is the sum of all edge weights in the graph, and $\rho > 0$ is a balancing parameter. The modularity of a graph G is then given by the sum of all cluster modularities: $Q = \sum_c Q_c$. A clustering that maximizes the modularity of the graph is one that has dense connections between nodes in the same clusters, with sparse connections between each of the clusters.

It has been shown that finding a partition of a graph that maximizes the modularity is NP-hard [14], so approximation algorithms are generally used. The Louvain algorithm [11] is an iterative hierarchical graph clustering algorithm. It works by first placing every node of the given graph G_0 into its

own cluster. Nodes are then moved into a neighbouring cluster iteratively if it increases the overall modularity of the graph. This process stops once no additional progress can be made. The result is a clustering over the graph, which can then be used to create a new aggregate graph G_1 as follows: clusters of G_0 become nodes in graph G_1, and two nodes in G_1 have an edge joining them if there exists an edge that connects neighbouring clusters in G_0. The resulting graph G_1 can then be used in this process again, which will produce another aggregate graph G_2, and so on until either the final graph has a single node or the returned graph G_{i+1} is the same graph as G_i (*i.e.*, no nodes were able to be moved to a different cluster). The result from this process is a hierarchy of graph clusterings, in which earlier graphs contain many smaller clusters which are then merged into fewer larger clusters.

The Louvain algorithm is given as pseudocode in Algorithm 18, which is adapted from Evans and Şimşek [28].

E.2 Implementation and Machine Details

The algorithms and environment are implemented in C++, adhering to the C++20 standard. The codebase [1] is compiled using the *GNU Compiler Collection* version 13.3.0, and uses the PyTorch 2.4 C++ frontend [80]. Where available, the official implementations of comparison methods are used. All experiments were conducted on an Intel i9-7960X and Nvidia 3090, with 128GB of system memory running Ubuntu 24.04.

E.3 Additional Experimental Details

In our experiments, PHS*(π^{SG}), LevinTS(π^{SG}), PHS*(π), LevinTS(π), and WA* all use ResNet-based [44] networks. PHS*(π) uses a single network with two heads, one for the policy and one for the heuristic. For PHS*(π^{SG}), we use separate networks for the conditional low-level policy, and a two-headed network for the subgoal policy and global heuristic. LevinTS(π^{SG}) follows the same network structure, without the heuristic head. We use the

[1] https://github.com/tuero/subgoal-guided-policy-search

Adam optimizer [51], with learning rate of 3E-4 and L2-regularization of 1E-4. The policy and heuristic networks for PHS*(π), LevinTS(π), PHS*(π^{SG}), and LevinTS(π^{SG}) both use 128 ResNet channels, with PHS*(π^{SG}) and LevinTS(π^{SG}) using half the number of blocks (4 versus 8) due to the fact that they have both a low-level and high-level policy. The VQVAE subgoal generator uses a codebook size of 4, a codebook dimension of size 128, and $\beta = 0.25$. Codebook entries are updated using exponential moving averages of encoder assignments with decay 0.99 and numerical stabilizer $\epsilon = 10^{-5}$, following the EMA variant of Van Den Oord, Vinyals, et al. [105]. We use the open source implementation for HIPS-ε in our experiments[2]. We use ϵ = 1E-3 for all environments, and use the version of HIPS-ε which assumes access to an environment dynamics model. All other hyperparameters follows the authors' choices in their Box-World experiments [58].

[2]https://github.com/kallekku/HIPS

Algorithm 18 Louvain

Input: Base graph $G_0 = (V_0, E_0)$
Output: Hierarchy of graph clusterings
for $i = 0, 1, \ldots$ **do**
    $C_i \leftarrow \{\{u\} \mid u \in V_i\}$ ▷ *Initialize singleton cluster over each node.*
    $Q_{\text{old}} \leftarrow \sum_{c \in C_i} (\Sigma^c_{\text{in}}/2m - \rho(\Sigma^c_{\text{tot}})^2/(2m)^2)$ ▷ *Old modularity of C_i*
    repeat
        for each $u \in V_i$ **do**
            ▷ *Set of clusters with u removed from its current cluster* ◁
            $C_i^{-u} \leftarrow \{\, c \setminus \{u\} \mid c \in C_i \,\} \setminus \{\emptyset\}$
            ▷ *Set of candidate neighbouring clusters u can move into* ◁
            $N_i(u) \leftarrow \{\, c \in C_i^{-u} \mid \exists v \in c \text{ with } (u, v) \in E_i \,\}$
            ▷ *Choose a best neighbouring cluster for u to move into* ◁
            $c^\star \in \arg\max_{c \in N_i(u)} \Delta Q_i(u \to c)$
            if $N_i(u) \neq \emptyset$ **and** $\Delta Q_i(u \to c^\star) > 0$ **then**
                ▷ *Move u into neighbour cluster if modularity increases* ◁
                $C_i \leftarrow \left(C_i^{-u} \setminus \{c^\star\}\right) \cup \{c^\star \cup \{u\}\}$
    until no change in C_i
    $Q_{\text{new}} \leftarrow \sum_{c \in C_i} (\Sigma^c_{\text{in}}/2m - \rho(\Sigma^c_{\text{tot}})^2/(2m)^2)$ ▷ *New modularity of C_i*
    if $Q_{\text{new}} > Q_{\text{old}}$ **then**
        ▷ *Parent graph G_{i+1}, where nodes are clusters from G_i.* ◁
        $V_{i+1} \leftarrow \{c \mid c \in C_i\}$
        $E_{i+1} \leftarrow \{(c_i, c_j) \mid c_i, c_j \in C_i, c_i \text{ and } c_j \text{ are neighbours in } C_i\}$
        $G_{i+1} \leftarrow (V_{i+1}, E_{i+1})$
    else
        break ▷ *No further gain.*
return $(G_0, \ldots, G_N)$

Appendix F

Supplementary Material for Chapter 7

F.1 Proof of $\sqrt{\text{LTS}}$-L Runtime

Theorem 7.4.1. *Let N be the number of node expansions performed by $\sqrt{\text{LTS}}$-L, and let D be the maximum depth of any generated node after these expansions. Assume that each expanded node generates at most $b < \infty$ children, that successor generation, policy lookup, colour lookup, and colour count updates take $O(1)$ time per generated node, and that the priority key of a generated node can be computed from stored ancestor information in $O(d(n))$ time. Assume, as an implementation-level runtime model consistent with the Leiden analysis of Traag et al. [101], that a Leiden call on a graph $G = (V, E)$ truncated to a fixed hierarchy level k takes $O(k|E|)$ time. If Leiden clustering is invoked according to a geometric schedule with fixed factor $\gamma > 1$, then the runtime of $\sqrt{\text{LTS}}$-L over N expansions is*

$$O\left(bN\log(bN) + bDN + kbN\frac{\gamma}{\gamma - 1}\right).$$

In particular, for fixed b, k, and γ, this is

$$O(N\log N + DN).$$

If k grows with the graph hierarchy depth, the clustering term retains its explicit factor k.

Proof. At expansion i, let M_i be the number of nodes in the priority queue. Since each expansion removes one node and inserts at most b children, $M_i \leq$

$1 + (b-1)i \le bN + 1$. Removing the minimum-priority node therefore takes $O(\log M_i)$ time, and inserting the generated children takes $O(b \log M_i)$ time.

For each generated child n, the $\sqrt{\text{LTS}}$-L priority key is computed once, when the node is inserted into the priority queue. This is valid because rerooting weights are fixed once assigned, so later cluster updates do not change the priorities of already generated frontier nodes. By assumption, this key computation takes $O(d(n))$ time from stored ancestor information. Since $d(n) \le D$, the total non-clustering cost over N expansions is

$$O\left(\sum_{i=1}^{N}\big((1+b)\log M_i + bD\big)\right) = O(bN\log(bN) + bDN)\,.$$

It remains to bound the clustering cost. Let the clustering calls occur at steps $r_0, r_1, \ldots, r_J$, where $r_j = \lfloor \gamma^j \rfloor$ up to constant factors and $J = \lfloor \log_\gamma N \rfloor$. At step r_j, at most r_j nodes have been expanded, and each expansion generates at most b children. Hence the graph passed to Leiden contains at most $O(br_j)$ generated edges. By the Leiden runtime assumption stated in the theorem, the j-th Leiden call therefore costs

$$O(k|E_j|) = O(kbr_j) = O(kb\gamma^j),$$

where E_j is the edge set of the graph clustered at the j-th call. Summing over all calls up to step N gives

$$\begin{aligned} O\left(kb\sum_{j=0}^{\lfloor \log_\gamma N \rfloor}\gamma^j\right) &= O\left(kb\frac{\gamma^{\lfloor \log_\gamma N \rfloor + 1} - 1}{\gamma - 1}\right) \\ &= O\left(kbN\frac{\gamma}{\gamma - 1}\right). \end{aligned}$$

For fixed k and fixed $\gamma > 1$, this clustering overhead is linear in bN, and is dominated by the priority-queue term for $N \ge 2$. Combining the non-clustering and clustering costs gives the stated bound. ■

F.2 Proof of Additive Rerooter Bound

Theorem 7.4.3. *Let w_a and w_b be two rerooters, with relative weights $u_a, u_b > 0$. Let $w = u_a w_a + u_b w_b$ be the rerooter used by $\sqrt{\text{LTS}}$. When visiting the node n_T at step T, for any $C \geq 1$ satisfying $\frac{1}{C} \leq \frac{u_a w_{a,<T}}{u_b w_{b,<T}} \leq C$, and provided the component weights in the displayed ratios are positive, then the number of node visits is bounded by*

$$T \leq 1 + (C+1) \times \min_{D \in \mathcal{D}(n_T)} \max_{i<|D|} \min \left\{ \frac{w_{a,<T}}{w_{a,T_i}}, \frac{w_{b,<T}}{w_{b,T_i}} \right\} c^r_{T_i}(n_{T_{i+1}}). \tag{7.7}$$

The degenerate case in which one coefficient is zero reduces to the corresponding one-component rerooter and is not covered by this balanced-additive statement.

Proof. Since the balance ratio is well-defined and bounded by C, we have $u_a w_{a,<T} \leq C u_b w_{b,<T}$ and $u_b w_{b,<T} \leq C u_a w_{a,<T}$, so $w_{<T} = u_a w_{a,<T} + u_b w_{b,<T} \leq (C+1) u_a w_{a,<T}$ and also $w_{<T} \leq (C+1) u_b w_{b,<T}$. Therefore,

$$\begin{aligned}
\frac{w_{<T}}{w_t} &= \frac{w_{<T}}{u_a w_{a,t} + u_b w_{b,t}} \\
&\leq \min \left\{ \frac{w_{<T}}{u_a w_{a,t}}, \frac{w_{<T}}{u_b w_{b,t}} \right\} \\
&\leq (C+1) \min \left\{ \frac{u_a w_{a,<T}}{u_a w_{a,t}}, \frac{u_b w_{b,<T}}{u_b w_{b,t}} \right\} \\
&= (C+1) \min \left\{ \frac{w_{a,<T}}{w_{a,t}}, \frac{w_{b,<T}}{w_{b,t}} \right\}.
\end{aligned}$$

Plugging this into the slenderness-cost analogue of Equation 3.28, obtained by replacing each rooted Levin term $\frac{d}{\pi}(n_{T_i} \prec n_{T_{i+1}})$ with the rooted slenderness term $c^r_{T_i}(n_{T_{i+1}})$, gives the result. ∎

F.3 Implementation and Machine Details

The algorithms and environment are implemented in C++, adhering to the C++23 standard. The code[1] is compiled using the *GNU Compiler Collection* version 15.2.0, and uses the PyTorch 2.7 C++ frontend [80]. The Leiden clustering subroutine uses `leidenalg`[2], an efficient open source implementation.

[1] https://github.com/tuero/siirlts2026

[2] https://github.com/vtraag/libleidenalg

Where available, the official implementations of comparison methods are used. All experiments used 8 threads for the bootstrap training and testing, and were conducted on an Intel i9-7960X and Nvidia 3090, with 128GB of system memory running Ubuntu 24.04.

F.4 Additional Experimental Details

During the online bootstrap training process, all algorithms start with an initial budget of 4,000 node expansions. Batched inference is used during training, where generated nodes are placed into a queue before sending to the policy/heuristic networks for inference. A batch size of 32 is used, with smaller batches being used if the open priority queue becomes empty before 32 additional nodes are generated. In our experiments, all algorithms use ResNet-based [44] networks. Algorithms which use both a policy and heuristic use a single network with two heads. The networks for WA*, $\sqrt{\text{LTS}}$-L, $\sqrt{\text{LTS}}$-H and $\sqrt{\text{LTS}}$-LH use 8 blocks of 128 ResNet channels, are trained using the Adam optimizer [51], with learning rate of 3E-4 and L2-regularization of 1E-4. The baselines LevinTS(π^{SG}) and PHS*(π^{SG}) are trained following Tuero et al. [103] and use the open source implementation.[3]

For $\sqrt{\text{LTS}}$-L and $\sqrt{\text{LTS}}$-LH, we use a graph update factor of $\gamma = 1.2$ for the geometric schedule for when the Leiden clustering procedure is run. We use a cluster level $k = N$, where N is the hierarchy depth returned at that clustering invocation; the final available level is selected, with no rounding required. This experimental choice is distinct from the fixed-k assumption used to simplify the asymptotic runtime statement; under the fixed search budget, each clustered graph has finite size and depth. For $\sqrt{\text{LTS}}$-H and $\sqrt{\text{LTS}}$-LH, we use $\alpha = 10$ for the inverse-temperature heuristic parameter, selected a priori from expected solution-length scales rather than from validation or test performance. The later ablation is descriptive and does not determine this value. All reported results use $u_a = u_b = 1$ for $\sqrt{\text{LTS}}$-LH. For a sensitivity analysis on having balanced rerooters, see Section 7.5.7.

[3] https://github.com/tuero/subgoal-guided-policy-search

F.5 Complete Training Results

Table F.1: Average online training loss with respect to expansions and aggregate computation time. Aggregate computation time measures the sum of the time an algorithm spends on each problem across all threads used during training, in hours.

Algorithm	Expansions	Time	Solved
BoulderDash			
WA*	652,027,388.80	122.87	9,966.60
LTS	1,462,860,716.80	278.12	106.00
LevinTS(π^{SG})	259,846,338.00	104.19	9,999.60
PHS*(π^{SG})	94,266,331.60	35.12	10,000.00
$\sqrt{\text{LTS}}$-L	128,970,198.60	32.57	9,996.00
$\sqrt{\text{LTS}}$-H	**24,154,363.00**	**7.46**	9,999.40
$\sqrt{\text{LTS}}$-LH	28,257,820.80	8.47	9,995.80
CraftWorld			
WA*	456,670,096.60	155.71	9,847.40
LTS	804,101,431.20	277.91	315.00
LevinTS(π^{SG})	422,689,637.00	278.07	434.20
PHS*(π^{SG})	252,454,918.60	153.53	9,998.20
$\sqrt{\text{LTS}}$-L	479,154,851.00	257.24	9,507.40
$\sqrt{\text{LTS}}$-H	164,845,073.20	60.20	9,939.40
$\sqrt{\text{LTS}}$-LH	**155,766,283.20**	**48.51**	9,925.00
Sokoban			
WA*	131,496,170.40	18.77	47,411.80
LTS	172,870,658.20	29.86	47,378.00
LevinTS(π^{SG})	233,050,490.80	69.79	47,226.20
PHS*(π^{SG})	127,096,718.60	38.15	47,452.00
$\sqrt{\text{LTS}}$-L	170,776,344.60	34.91	47,564.20
$\sqrt{\text{LTS}}$-H	100,488,660.60	17.89	47,798.60
$\sqrt{\text{LTS}}$-LH	**95,196,226.80**	**17.88**	47,756.60
TSP (GridWorld)			
WA*	2,059,512,675.20	258.51	4,000.60
LTS	1,654,329,587.60	226.23	9,999.60
LevinTS(π^{SG})	122,659,044.00	41.26	10,000.00
PHS*(π^{SG})	22,764,606.80	7.21	10,000.00
$\sqrt{\text{LTS}}$-L	67,925,838.20	13.91	10,000.00
$\sqrt{\text{LTS}}$-H	693,847,701.00	53.30	10,000.00
$\sqrt{\text{LTS}}$-LH	**13,481,743.40**	**3.33**	10,000.00