Learning Policies and Heuristics for Bidirectional Search

by

Kenneth Gilles Tjhia

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

Master of Science

in

Statistical Machine Learning

Department of Computing Science

University of Alberta

Abstract

This thesis empirically investigates the comparative ease of learning policies and heuristics for bidirectional versus unidirectional search in satisficing classical planning. Our research explores the potential advantages of bidirectional search in terms of learnability and efficiency of the resulting guidance mechanisms. We employ a learning framework where neural networks parameterize policies and heuristics, trained on solutions generated by bidirectional and unidirectional search methods across three classical planning domains: Sliding Tile Puzzle, Witness Puzzles (Triangles and Colours), and Pancake Puzzle. We compare base algorithms that span a spectrum from pure heuristic-based to pure policy-based approaches. As models improve, they solve more problems and find different solutions, potentially enhancing the quality of training data. Experimental results suggest that learning satisficing policies and heuristics for bidirectional search is often easier and leads to more efficient search guidance. Consistently, during training, bidirectional searches achieved greater or equal solve rates after seeing fewer problems and using fewer cumulative expansions. The resulting learned bidirectional guidance mechanisms were efficient in terms of expansions and generalized well to test sets. These findings were particularly pronounced in the Witness and Pancake domains. We propose several possible explanations for these observations, including the potential for bidirectional search to yield different solution distributions, allow for complementary interaction between the two directions predictions, and reduce the effective search depth. This research contributes to the ongoing exploration of learning in planning, raising new questions about the relationship between search strategies and the ease of learning effective guidance.

Acknowledgements

First off, I want to express my deepest thanks to my advisor, Levi Lelis. Levi, your guidance and support have been the backbone of my Masters journey. From day one, I was impressed by your dedication to nurturing young researchers like myself. Your insights and patient mentorship have not only shaped my research but have had a profound impact on my growth as both a professional and a person. I feel incredibly fortunate to have learned from you. This thesis simply would not have been possible without your expertise and the countless hours you invested in guiding me through this process.

I owe a huge debt of gratitude to Mahta Shafieesabet, my partner and rock-solid support system. Mahta, your unwavering encouragement in every aspect of my life has been my anchor throughout this challenging yet rewarding journey. Even when I doubted myself, your belief in me never wavered. Thank you for your patience, love, and for being my constant source of strength and motivation. To my family, despite the physical distance, your emotional presence has been crucial to my success and perseverance. Your sacrifices and unconditional love have fueled my determination to push through the toughest times. The values you have instilled in me have guided me through both life and academia. I can't thank you enough for always believing in me.

I want to give a heartfelt shout-out to my current and past lab mates, and my academic comrades-in-arms. Our discussions, collaborations, and shared experiences have not only contributed to my research but have made this journey enjoyable and memorable. A special thanks to Shivham, Zahra, Samuel, Diego, Valentin, Tales, Justin, Sumedh, Rubens, Saqib, Habib, Elham, Zaheen, Mahdieh, Thirupathi, Amirhossein, Reza, Parnian, Mahdi, Quazi, and many others. Your friendship and intellectual contributions have been invaluable.

Lastly, I am incredibly grateful to the University of Alberta Computing Science department. The resources, opportunities, and supportive atmosphere have been key to my academic growth. It is clear to me that the faculty and staff are committed to fostering a culture of innovation and collaboration, which has been a constant source of motivation throughout my Masters program.

To everyone who has been a part of this journey, whether I have named you or not, know that your impact on my life and work means the world to me. I will always cherish the support and experiences you have provided.

Contents

List of Tables

List of Figures

List of Algorithms

Abbreviations

A* A* Search. ix, 2, 17–19, 27, 28, 32–34, 44, 45, 47, 48, 50, 51, 58, 60–62, 75, 81

BFS Best-first Search. 7, 12, 17–22, 29

BiA* Bidirectional A*. ix, 19, 33, 34, 44–48, 50, 51, 59–61, 63, 68, 69, 76, 82

BiBFS Bidirectional Best-first Search. 8, 10, 11, 13, 14, 17, 19, 20, 24, 58

BiLTS Bidirectional Levin Tree Search. ix, 20, 33, 44, 45, 47, 48, 59, 60, 65, 69, 78, 84

BiPHS* Bidirectional Policy-guided Search. ix, x, 21, 33, 34, 44–48, 51, 59, 60, 67–69, 80, 86

COL Witness Colours Puzzle Domain. viii, 25–27, 30, 32, 33, 38, 39, 44, 45, 47, 48, 60, 62–72, 75–86

CP Classical Planning. viii, 3–7, 9, 12, 13, 18–20, 22, 59

LTS Levin Tree Search. ix, 17, 19–21, 27, 28, 33, 34, 44, 45, 47, 48, 59, 60, 64, 77, 83

NN Neural Network. 12, 29, 30, 59

PAN Pancake Puzzle Domain. viii, ix, 25–27, 30, 32–34, 42–48, 59–72, 75–86

PF Pathfinding. 5, 7, 10, 17, 18, 20

PHS* Policy-guided Heuristic Search. ix, x, 17, 21, 27, 28, 33, 34, 44, 45, 47–49, 51, 58–60, 66, 79, 85

STP Sliding Tile Puzzle Domain. viii, 22, 23, 25–27, 30, 32–34, 40, 41, 44–48, 51, 59–70, 72, 75–86

TRI Witness Triangle Puzzle Domain. viii, 24–27, 30, 32, 33, 36, 37, 44, 47, 48, 60, 62–67, 69–72, 75–86

UniBFS Unidirectional Best-first Search. 7, 9–14, 17, 19, 21, 58, 59

WA* Weighted * Search. 19

Chapter 1

Introduction

Classical planning is a fundamental area of artificial intelligence that focuses on developing algorithms to find sequences of actions that transform an initial state into a goal state (Fikes & Nilsson 1971; LaValle 2006; Russell & Norvig 2016). While significant progress has been made in this field, the challenge of efficiently solving complex planning problems remains. One approach that has gained traction in recent years is the use of machine learning techniques to improve search algorithms by learning policies and heuristics (Arfaee et al. 2011; Orseau & Lelis 2021; Sakaue & Oki 2022; Pendurkar et al. 2023; Pendurkar et al. 2024).

Within the broader field of classical planning, satisficing planning has emerged as a practical approach for tackling real-world problems where finding any valid solution quickly is crucial. This is in contrast to optimal and bounded suboptimal planning, which often require more time but give guarantees about the cost of a returned solution.

Traditionally, unidirectional search methods, such as forward search, have been the primary focus of research in classical planning (Bonet & Geffner 1999; Bonet & Geffner 2001; Hoffmann 2001; Helmert 2006; Richter & Westphal 2010; Segovia-Aguas et al. 2021). However, bidirectional search, which can simultaneously explore the state space from both the initial and goal states, has shown promise in certain domains (Torralba et al. 2014). Bidirectional search has the potential to reduce the effective search depth and quickly connect partial solutions from both ends (Sturtevant & Felner 2018), which could be particularly advantageous in problems where the connection between start and goal states is challenging from a single direction.

This thesis aims to address the gap in knowledge regarding the comparative ease of learning policies and heuristics for bidirectional versus unidirectional search in satisficing classical planning. We hypothesize that bidirectional search may offer advantages in two key areas: the ease of learning effective policies and heuristics, and the efficiency of the resulting search processes. To test this hypothesis, we employ a learning framework using neural networks to parameterize policies and

heuristics—collectively referred to as *guidance mechanisms*—trained on solutions generated by both search methods across three classical planning domains: Sliding Tile Puzzle, Witness Puzzles (Triangles and Colours), and Pancake Puzzle. We compare unidirectional and bidirectional variants of three base algorithms: A* Search (Hart et al. 1968), Policy-Guided Heuristic Search (Orseau & Lelis 2021), and Levin Tree Search (Orseau et al. 2018). These algorithms represent a spectrum from pure heuristic-based to pure policy-based approaches, allowing for a comprehensive comparison of learning and performance in bidirectional and unidirectional contexts.

Our research methodology involves an iterative process where neural network models are trained on solutions generated by bidirectional and unidirectional search methods (Arfaee et al. 2011). As these models improve, the searches are capable of solving more problems and finding different solutions to previously encountered ones, potentially enhancing the quality of the training data. This approach allows us to compare the learning trajectories and final performance of guidance mechanisms for both search strategies.

In all three domains considered, bidirectional variants achieve similar or greater solve rates, sometimes significantly so, after seeing fewer problems and using fewer cumulative expansions. These findings demonstrate the comparative ease of learning bidirectional versus unidirectional guidance mechanisms.

On held-out test sets, the learned bidirectional guidance mechanisms often have greater or equal solve rates, and on average require fewer expansions per problem. This demonstrates the efficiency of the learned bidirectional guidance mechanisms.

Together, these findings suggest that bidirectional search often provides a more favorable learning environment, allowing for quicker acquisition of effective policies and heuristics. The implications of this research are significant for the field of satisficing classical planning, as it indicates potential for developing more efficient and scalable planning algorithms in a learning setting, which can be particularly beneficial when hand-crafted policies or heuristics are difficult to devise.

By leveraging the advantages of bidirectional search in learning policies and heuristics, we open up new avenues for improving the efficiency and effectiveness of AI planning systems in diverse real-world scenarios where finding any solution quickly is crucial.

Chapter 2

Background

2.1 Notation

We use $\mathbb{R}$ for the set of real numbers, and $\mathbb{R}_{\geq 0}$ for the set of nonnegative real numbers. For arbitrary sets X and Y, we use $X \subset Y$ to mean X is a proper subset of Y, and $X \subseteq Y$ to mean X is a subset—not necessarily proper—of Y. By $|X|$ we denote the cardinality of X. By $f : X \multimap Y$ we mean f is a *partial* function from X to Y. For any positive integer N, we define $[N] \coloneqq \{1, 2, \ldots, N\}$, and X^N to be the n-ary Cartesian power of X, $X^n \coloneqq \{(x_1, \ldots, x_n) \mid \forall_{i \in [n]} x_i \in X\}$. When X is countable, we use $\Delta(X)$ to denote the set of all probability distributions over X, and 2^X to denote the power set of X. Except where noted otherwise, arg min always returns the set of all minimizing elements.

2.2 Classical Planning

This thesis examines algorithms for solving instances of the classical planning problem (CP) (Fikes & Nilsson 1971). Our focus is on finding *satisficing* solutions (Simon 1956), which in this context means that solutions of any cost are acceptable. While some authors refer to this as satisficing classical planning, for brevity, we will use the term "classical planning" throughout this work.

We formalize CP instances using a state-space model (LaValle 2006; Russell & Norvig 2016). Each CP instance is defined by a start state, a goal function that defines a set of goal states, a function that returns the set of valid actions for a given state, and a deterministic transition function that returns the resulting state after applying an action.

A solution to a CP instance is a sequence of valid actions that, when applied in order from the start state, leads to a goal state. Because we are in the satisficing setting, where given a CP instance all solutions are equally preferred, we do not introduce a transition cost function, which is typically

used to define the criteria of an optimal solution. However, when contrasting our approach with others, we sometimes refer to notions of optimality. In such cases, we use the length of a solution as a measure of optimality—which is equivalent to assigning cost 1 to all transitions.

Here we provide the formal definition of a CP instance that we use throughout this thesis:

Definition 2.1 (Classical Planning) An instance of CP consists of the following components:

- A finite, non-empty set of *states* $\mathcal{S}$.
- A *start* state $s^1 \in \mathcal{S}$.
- A *goal* function $G : \mathcal{S} \to \{true, false\}$ that returns *true* for at-least one $s \in \mathcal{S} \setminus \{s^1\}$, and $false$ otherwise. This function defines the goal criteria for the problem instance. For convenience of notation we define $\mathcal{S}^* \coloneqq \{s \in \mathcal{S} \mid G(s) = true\}$ to be the set of goal states.
- A finite, non-empty set of *actions* $\mathcal{A}$.
- An *action function* $A : \mathcal{S} \to 2^{\mathcal{A}}$, mapping each state to a set of *valid* actions.
- A partial function $T : \mathcal{S} \times \mathcal{A} \multimap \mathcal{S}$, called the *transition function*, mapping each state and its valid actions to a resulting state, i.e. $T(s, a) = s'$ where $s, s' \in \mathcal{S}$ and $a \in A(s)$.

A solution to a CP instance is any sequence of actions $(a_i)_{i=1}^{l} \in \mathcal{A}^l$, $l \geq 1$, satisfying:

- $a_1 \in A(s^1)$.
- for $i = 2, \ldots, l$, $a_i \in A(s_i)$ where $s_i = T(s_{i-1}, a_{i-1})$ and $s_1 = s^1$.
- $T(s_l, a_l) = s_{l+1} \in \mathcal{S}^*$.

We refer to sequences satisfying the first two criteria, where s^1 may be replaced by any arbitrary state, as *paths*. These sequences do not necessarily need to satisfy the third criterion.

The length of a path is defined to be the number of actions it comprises. As previously mentioned, we take the length of a path as its cost, implicitly assuming all transitions have cost 1. A solution is then considered *optimal* if it is minimal in length, i.e. any solution of length l is optimal if there does not exist a solution of length $l' < l$. In this work, we assume that at least one solution exists for each instance.

Due to the fixed start state and deterministic transitions, each path corresponds to a unique sequence of states. Thus, with a slight abuse of notation we will sometimes refer to paths as

sequences of states or state-action tuples.

With this formulation, each instance of CP can be identified as a pathfinding (PF) (Definition 2.2) problem instance (Pohl 1970; Pearl 1984; Geffner & Bonet 2022) on a finite, directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where each state $s \in \mathcal{S}$ corresponds to a unique node $n \in \mathcal{N}$. A directed edge labelled a exists from node n to n' if and only if $s' = T(s, a)$ for some $a \in A(s)$, where s and s' are the states corresponding to nodes n and n', respectively. Each path in the CP instance corresponds to a sequence of edges, or nodes, in the PF instance, which we also call a path. If our CP formulation included transition costs, these would correspond to a weight function on the edges. As mentioned, however, we use the length of a path as its cost. We provide a formal definition of PF in Definition 2.2.

Definition 2.2 An instance of pathfinding (PF) consists of:

- A directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is the set of nodes and $\mathcal{E}$ is the set of directed edges.
- A start node $n^1 \in \mathcal{N}$.
- A set of goal nodes $\mathcal{N}^* \subset \mathcal{N} \setminus \{n^1\}$.

A solution to such an instance is a path from n^1 to any node in $\mathcal{N}^*$, i.e. a sequence $(e_i)_{i=1}^{l}$, where e_i in $\mathcal{E}$ for all $i \in [l]$, and $e_1 = (n^1, n)$ and $e_l = (n', n^*)$ for some $n, n' \in \mathcal{N}$ and $n^* \in \mathcal{N}^*$. Paths and path costs are defined analogously to in CP, and similarly we assume at least one solution exists.

In the context of CP, we additionally label each edge $e \in \mathcal{E}$ with the action $a \in \mathcal{A}$ associated with its corresponding transition. The set of paths (solutions) in a CP instance corresponds to the set of paths (solutions) in its associated PF instance. Figure 2.1 illustrates an example of a CP instance and its associated PF instance.

It is crucial to understand that we wish to avoid explicitly constructing the entire set of states $\mathcal{S}$, often even $\mathcal{S}^*$, or their corresponding nodes in the associated graph representation. The set $\mathcal{S}$ is presumed to be sufficiently large to preclude complete enumeration. Moreover, in some instances, computing even a single $s \in \mathcal{S}^*$ is equivalent to solving the entire problem—a situation exemplified by the Witness domains (Section 3.1.2) examined in this study. The graph representation of a CP instance is implicitly defined using the initial state, the action function, the transition function, and the goal function. In this thesis, we consider algorithms that construct only the necessary parts of the graph as required during the search process. Our main criteria for comparing the efficiency of learned guidance mechanisms is minimizing this portion of the graph.

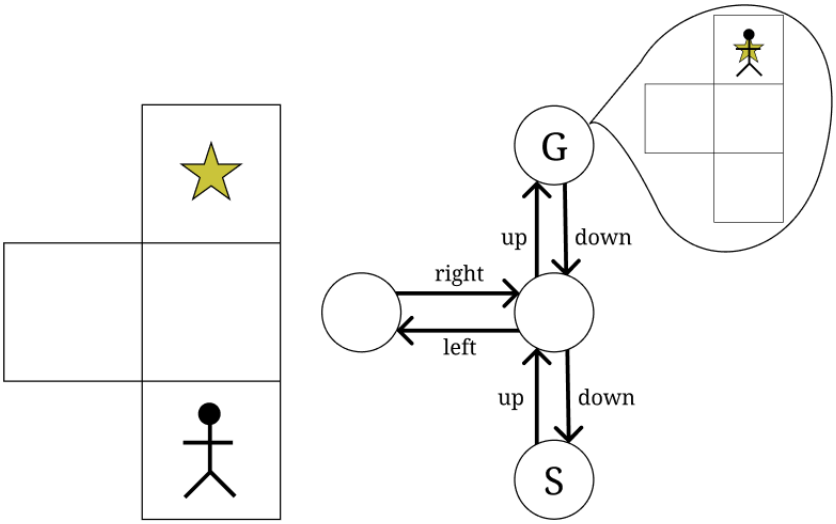

Figure 2.1: The left part of the image depicts the initial state of a classical planning problem where a stick figure is positioned in the bottom square. The right part of the image illustrates the corresponding graph for this problem, with the node labelled S corresponding to the start state. Each state in this problem, and therefore also node in the graph, corresponds to a configuration where the stick figure occupies one of the squares. The possible actions are {up, down, left, right}. The valid actions at any state are those directions in which there is a square adjacent to the stick figure, which correspond to outgoing edges in the graph. Executing an action moves the stick figure to the adjacent square in the specified direction.
The single goal state is achieved when the stick figure reaches the square containing the star, which corresponds to the node labeled G in the graph. There are bijections between both the graph nodes and the CP instance states, and the edges and CP instance transitions.

The state-space model formulation of CP and the associated PF problem is particularly well-suited to best-first search (BFS) algorithms (Dijkstra 1959; Hart et al. 1968; Nilsson 1982; Pearl 1984; Dechter & Pearl 1985; Korf 1993; LaValle 2006; Russell & Norvig 2016), which we discuss next.

2.3 Best-First Search

In this section we introduce two classes of graph search algorithms for PF that generalize all algorithms under our consideration.

2.3.1 Unidirectional Best-first Search

Unidirectional best-first search (UniBFS) (Dijkstra 1959; Hart et al. 1968; Pearl 1984; Dechter & Pearl 1985; Russell & Norvig 2016) is a class of graph search algorithms that explore a graph systematically using a real-valued node priority function (Section 2.4), f, and two sets of nodes: *frontier* and *visited*. The version presented here differs in several aspects from the standard description, which we will discuss subsequently.

Both sets, *frontier* and *visited*, initially contain only the start node. The search proceeds by iteratively removing a node with minimal f value from *frontier*, then *expanding* it. Expanding a node n involves:

For each successor n' of n:

- If $n' \in$ *visited*, continue to the next successor.
- If $n' \notin$ *visited*:
  - Add n' to *visited*.
  - Record n as n''s parent.
  - Check if $n' \in \mathcal{N}^*$, and return with success if true. Otherwise, evaluate and record $f(n')$ then add n' to *frontier*.

We say a node has been visited once it is added to the *visited* set. It can be seen that, absent some other termination criteria, the last node to be visited is always a goal node. We call a node expanded once all of its successors have been visited.

This description covers a basic version of UniBFS. Other variations may employ more complex termination criteria, revisit nodes, or incorporate additional features. Notably, in our approach, the goal check is performed while visiting a node, rather than expanding it, and a node's f value, which may depend on the path taken to reach it, is never re-evaluated. We describe UniBFS this way

because we focus on the satisficing setting, where all solutions are equally acceptable. Moreover, we consider the number of expansions an algorithm uses as our primary measure of efficiency. Given these criteria, it is reasonable to return solutions at the earliest opportunity.

If one is interested in optimal solutions, priority functions satisfying certain properties can guarantee optimality. However, this may require checking the goal during expansion rather than visitation, or re-evaluating a node's f value (Pearl 1984; Felner et al. 2011). These aspects will be discussed in more detail in Section 2.4.1.

Upon reaching a goal node, we reconstruct the solution by tracing backward from this node to the start node using the parent records, collecting the actions along the way. The search may also terminate without finding a solution when a predetermined computational budget is exhausted; in this work, we use the number of node expansions as our budget metric.

2.3.2 Bidirectional Best-first Search

Bidirectional best-first search (BiBFS) was first discussed in the literature as early as 1966 (Nicholson 1966; Doran 1966). Pohl (1969), however, was the first to introduce goal-directed BiBFS, which will be a major focus of this thesis.

In BiBFS, the algorithm alternates between two searches: the forward search and the backward search. Each search utilizes its own priority function f and maintains separate *frontier* and *visited* sets. Throughout this thesis, we will use the subscripts F and B to denote objects belonging to the forward and backward searches, respectively.

The forward search operates on the original graph, referred to as the forward graph. In contrast, the backward search operates on a derived graph that we call a backward graph.

A backward graph can always be constructed by keeping the nodes of the forward graph but reversing all edges while keeping their labels, as in the example shown in Figure 2.2, and two of the domains considered in this thesis (Section 3.1.1, Section 3.1.3). We will refer to this construction as a *simple* backward graph. While it can sometimes be more efficient to use a different construction of the backward graph (Green 1969; Nilsson 1982; Bonet & Geffner 1999; Alcázar et al. 2013), as in the Witness domains of Section 3.1.2, for the remainder of this chapter we describe BiBFS using a simple backward graph.

In the remainder of this chapter we assume the forward and backward graphs satisfy the following:

Assumption 2.1 There is exactly one forward goal node, which corresponds to exactly one node in the backward graph, n_B^1.

Assumption 2.2 For each node in the forward graph, no two incoming edges have the same label.

Assumption 2.1 allows us to initiate the backward search from a single node in the simple backward graph. Assumption 2.2 ensures that no node in the simple backward graph has two outgoing edges with the same label. In Section 2.3.4 we will state requirements for the CP instance so that these may be satisfied.

The main differences in the search procedure are as follows:

- Initialization: For the forward direction, initialize $frontier_F$ and $visited_F$ as in UniBFS, as singleton sets containing the forward start node. For the backward direction, initialize $frontier_B$ and $visited_B$ as singleton sets containing n_B^1.
- Search process: In each iteration, select a search direction and remove the node with the minimal f value from the corresponding *frontier* set for expansion.
- Expansion: Instead of checking if a successor n' is the goal node, we now check if n' is contained in the *visited* set of the opposite search direction, in which case a solution can readily be constructed.

Reconstructing a solution is done similarly to UniBFS, by recording actions as we trace a path from the goal node up the common visited node and back to the start.

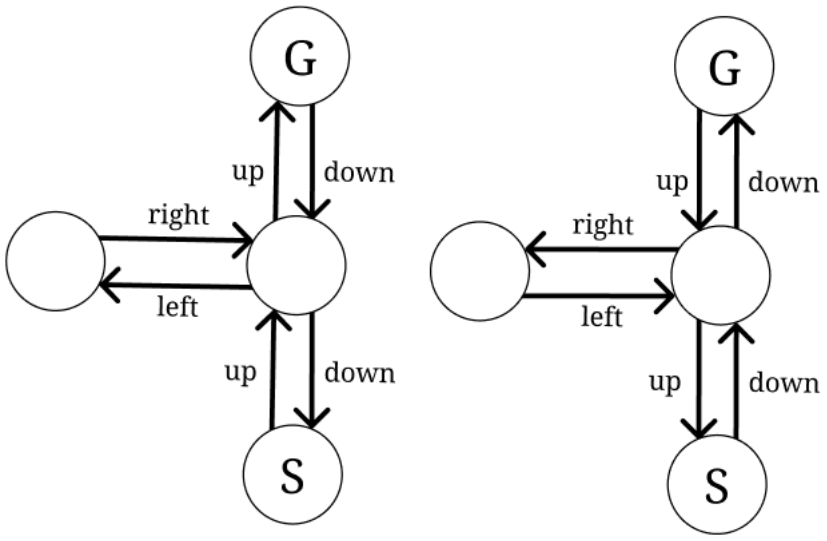

Figure 2.2: The left part of the image depicts the forward graph associated with the CP instance of Figure 2.1. The right part of the image depicts a simple backward graph. The nodes remain unchanged, but each edge has been reversed. In our notation the edge labels remain the same.

As with the description of UniBFS, here we described a simple version of BiBFS. Again, since we are only interested in satisficing solutions, we do a simple goal-check that returns the first solution found and do not re-expand nodes.

Throughout most of this thesis, we focus on a specific bidirectional search strategy: after choosing an initial direction arbitrarily, we strictly alternate directions for each subsequent expansion. Henceforth, we will use BiBFS to refer specifically to this strictly alternating strategy, and will explicitly mention when we mean another approach.

In Section 2.4 we discuss priority functions, but for now we state an assumption that all priority functions considered in this thesis satisfy, and that will allow us to get a straightforward upper bound on the number of node expansions in bidirectional search compared to its unidirectional counterpart, provided that the strict alternation criteria is also used. This relationship is formalized in Theorem 2.1. This theorem, along with the experimental results of Appendix A.1, where the strictly alternating approach outperforms an alternative selection criteria that always selects the node with minimal f value from either direction's *frontier*, provide justifications for our decision to focus on the strictly alternating approach.

Assumption 2.3 Whenever two priority functions f_F and f_B are used in bidirectional search on the forward graph and its associated backward graph, $\mathcal{G}_F$ and $\mathcal{G}_B$, respectively, $f_F(n)$ may depend only on the portion of $\mathcal{G}_F$ that is constructed at the time n is visited and potentially also the forward goal node. In particular, $f_F(n)$ is independent of the progress of the backward search and does not depend on $frontier_B$. $f_B(n)$ is similarly independent of the progress of the forward search.

Bidirectional search algorithms that adhere to this assumption are called *front-to-end* searches (Barker & Korf 2015). All algorithms examined in this work fall into this category.

Theorem 2.1 Let P be an instance of PF, and f_F and f_B be two priority functions defined on the associated forward and backward graphs $\mathcal{G}_F$ and $\mathcal{G}_B$, respectively, that satisfy Assumption 2.3. Then, on P, BiBFS using f_F and f_B will expand no more than $2 \cdot \min\{E_F, E_B\}$, where E_F is the number of expansions UniBFS uses on $\mathcal{G}_F$ with priority function f_F (denoted UniBFS$_F$), and E_B is the number of expansions UniBFS uses on $\mathcal{G}_B$ with priority function f_B (denoted UniBFS$_B$).

Proof. Suppose, without loss of generality, that $E_F \leq E_B$, i.e. UniBFS$_F$ expands at most as many nodes as UniBFS$_B$, and let $u_1, u_2, \ldots, u_{E_F}$ be the sequence of nodes expanded by UniBFS$_F$. Since BiBFS strictly alternates directions and f_F and f_B are computed independently of the progress of the opposite search direction, we have two cases:

1. The first expansion of BiBFS is in the *forward* direction, in which case BiBFS expands node u_i in iteration $2i - 1$, so either the searches meet somewhere before node u_{E_F}, or BiBFS expands $2E_F - 1$ nodes.

2. The first expansion of BiBFS is in the *backward* direction, in which case BiBFS expands node u_i in iteration $2i$, so either the searches meet somewhere before node u_{E_F}, or BiBFS expands $2E_F$ nodes.

In either case, BiBFS expands at most $2E_F$ nodes, proving the theorem. □

The bound of Theorem 2.1 is pessimistic, however, and in practice the two searches often meet after fewer expansions compared to the unidirectional case (Nilsson 1982; Kaindl & Kainz 1997; Alcázar et al. 2013; Barker & Korf 2015). These potential savings depend on both the structures of the graphs and the priority functions used to guide the searches (Nilsson 1982). The following two examples illustrate this.

Example 1 Consider a perfect b-ary tree of height l, where the start node is the root and the single goal node is a leaf. Assume a simple backward graph. In the worst case, with adversarially chosen priority functions that maximize the number of expansions:

- A UniBFS algorithm would need to expand all $2^l - 1 = \mathcal{O}(2^l)$ internal nodes to find the goal.
- For BiBFS, the backward graph is a single chain from the goal to the root. Thus, it expands at most $2l - 2 = \mathcal{O}(l)$ total nodes.

Example 2 Consider a square grid graph of width $n > 2$ nodes with bidirectional edges, where start and goal nodes are at opposite corners. The optimal solution path cost is $l = 2(n-1)$. Again, consider a simple backward graph and adversarially chosen priority functions:

- In the worst case, a UniBFS algorithm would expand all $n^2 - 3 = \mathcal{O}(l^2)$ nodes (excluding the goal and its immediate neighbors).
- A BiBFS search, in the worst case, can expand at most $n^2 - 2n = \mathcal{O}(l^2)$ nodes, since the sets of expanded nodes in each direction must always be separated by two unexpanded nodes.

This example shows less substantial savings in the worst case for BiBFS compared to the previous tree example.

In both examples, under the best-case scenario with ideal priority functions that expand nodes only along a single optimal solution path of cost l, unidirectional and bidirectional searches both expand $l - 1$ nodes. This represents the optimal number of expansions any UniBFS or BiBFS algorithm could achieve, regardless of the direction selection strategy employed. Consequently, the strictly alternating strategy of BiBFS can, in the best case, match this minimum number of expansions required to find a solution.

In contrast, the other direction selection strategy we consider—always selecting a node with minimal f value from either direction's *frontier*—can lead to significantly more expansions in the worst case. Consider a scenario where one direction's priority function assigns lower f values to every node not on a solution path compared to the f values of nodes on solution paths from either direction. In this case, at least every non-solution path from that direction would need to be explored before a solution could be discovered, regardless of the number of expansions a UniBFS from the other direction would use. This scenario illustrates why the upper bound established in Theorem 2.1 does not apply to this alternative strategy.

2.3.3 Batch Best-first Search

To improve practical efficiency, this work considers a generalization of (Uni/Bi)BFS similar to the works of Agostinelli et al. (2019) and Orseau & Lelis (2021). Instead of immediately evaluating the f values of successor nodes and adding them to the *frontier* set during expansion, we collect these successors into an evaluation buffer. After each expansion, we check if the buffer contains at least K nodes, where K is a predefined positive integer. If the buffer has at least K nodes, we then evaluate all nodes in the buffer, add them to the *frontier* set, and clear the buffer.

This approach is particularly beneficial when there are computational savings in evaluating the f values of multiple nodes simultaneously. For example, in our experiments, a neural network (NN) is used to compute parts of the f values, making batch processing more efficient since several nodes are evaluated in parallel.

It is important to note that different values of K can affect the order of node expansion, and hence which solution is found. For instance, with $K = 1$, the successors of the currently expanded node are immediately added to the *frontier* set and can be expanded in the next iteration. However, with $K > 1$, a node might not be available for expansion until several iterations have passed, as it waits for the buffer to reach the required size.

As a consequence, larger values of K result in a more thorough exploration of the current frontier before growing it further. This approach deviates from strictly adhering to the priority function order when expanding nodes. Such behavior can be advantageous, as previous research has demonstrated that incorporating random exploration mechanisms can enhance the likelihood of finding solutions (Valenzano et al. 2014).

2.3.4 Best-first Search for Classical Planning

Here we formalize requirements on the CP instance so that Assumptions 2.1 and 2.2 are satisfied. Then present the pseudocode for batch versions of unidirectional and bidirectional BFS for CP, in Algorithms 1 and 2 respectively.

First note that in our notation, we use the identifier of an action in the forward direction to also represent its 'reverse' action in the backward direction.

Next, observe that the pseudocode for BiBFS (Algorithm 2) has the following parameters: A_F, A_B, T_F, T_B, f_F, f_B, which are not present in the UniBFS pseudocode (Algorithm 1). A_F and T_F are the action and transition functions A, T from the UniBFS pseudocode and Definition 2.1. A_B and T_B are similarly the action and transition functions corresponding to the backward direction. We will describe their characteristics in more detail below. f_F and f_B are the priority functions for the forward and backward search, respectively. To apply BiBFS to an instance of CP we must be able to construct these backward objects.

In order for our graphs to satisfy Assumption 2.1, we require that only one $s^* \in \mathcal{S}$ is a goal state. This ensures that we can initiate the backward search from a single backward node.

To satisfy Assumption 2.2, we require that for all $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}$, $T_F(s, a) = T_F(s', a)$ implies $s' = s$. This is to prevent a situation where T_B would need to map a state and valid action to a set of next states, rather than a single state.

If the CP instance satisfies these requirements, in order to construct nodes in the backward graph for BiBFS we must further be able to construct $A_B : \mathcal{S} \to 2^{\mathcal{A}}$ and $T_B : \mathcal{S} \times \mathcal{A} \dashrightarrow \mathcal{S}$ so that for all $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}$:

1. $a \in \mathcal{A}_F(s)$ and $T_F(s, a) = s'$ if and only if $a \in A_B(s')$ and $T_B(s', a) = s$.
2. $a \in \mathcal{A}_B(s)$ and $T_B(s, a) = s'$ if and only if $a \in A_F(s')$ and $T_F(s', a) = s$.

Together, these conditions guarantee that we can uniquely reverse each transition of the CP instance, ensuring that a valid solution can be constructed once the goal condition is met.

Next, we will provide more detail about the behaviour of the CreateNode, CheckGoal, and EvalBatch functions, which are undefined in the pseudocode. The behaviour of these functions depends on whether the search is unidirectional or bidirectional.

CreateNode(s', n, a) creates and returns a search node n' corresponding to a state s' and a parent node n, which has its own corresponding state s. There are at least three things CreateNode must do:

- Store the corresponding state, $n'.state \leftarrow s'$.
- Store the parent node, $n'.parent \leftarrow n$.
- Store the action a which was applied from s—the state corresponding to the parent node n—to result in s', $n'.action \leftarrow a$.

- Set the g attribute of a node, $n'.g \leftarrow n.g + 1$ (or 0 if n is $null$), which is the cost of the path the search took to visit n'. The priority functions considered in this work will make use of this attribute.

The *action* and *parent* attributes are used to reconstruct a solution inside the CheckGoal function. The *state* attribute is what is used for inclusion checks in the *frontier* and *visited* sets. For BiBFS, we additionally take an argument *dir* specifying the direction of the search that this node was visited in, $n'.dir \leftarrow dir$, which is used during EvalBatch to determine how to set the f attribute of a node and which *frontier* set to add it to.

CheckGoal(n) checks whether a node n can be immediately used to construct a solution.

In UniBFS, CheckGoal(n) simply checks if $G(n.state) = true$. If this is the case, CheckGoal constructs a solution by recording the *action* attributes while traversing the *parent* pointers up to the start node, then returns it. If $G(n.state) = false$, CheckGoal returns $null$.

In BiBFS, CheckGoal($n, visited_{otherdir}$) checks if $n \in visited_{otherdir}$. In this case, by our requirements on the backward search objects, we can construct a solution by recording actions and traversing up the path in each direction from n to the corresponding start node. If $n \notin visited_{otherdir}$, then we can not yet construct a solution and CheckGoal returns $null$.

EvalBatch simply computes the f attribute of each node passed in the buffer using its corresponding directions priority function, adds it to the appropriate *frontier* set, then clears the buffer.

The UniBFS and BiBFS frameworks described here are sufficient to capture a large and diverse set of searching strategies, each parameterized by the priority functions they use. In the next section we discuss these in more detail.

Algorithm 1 Unidirectional Batch Best-First Search

Input:

Start state s^1, action function A, transition function T, goal function G.
Priority function f.
Batch evaluation parameter K, expansion budget *budget*.

Output:

Solution *sol*, number of expansions *exp*.

$n^1 \leftarrow$ CreateNode$(s^1, null, null)$
$exp \leftarrow 0$
$buff \leftarrow \emptyset$
$frontier \leftarrow \{n^1\}$
$visited \leftarrow \{n^1\}$
while $exp < budget$ **do**
    $n \leftarrow$ random node from $\arg\min\{n'.f \mid n' \in frontier\}$
    $frontier \leftarrow frontier \setminus \{n\}$
    $exp \leftarrow exp + 1$
    for $a \in A(n.state)$ **do**
        $n' \leftarrow$ CreateNode$(T(n.state, a), n, a)$
        if $n' \notin visited$ **then**
            $visited \leftarrow visited \cup \{n'\}$
            $sol \leftarrow$ CheckGoal(n')
            if $sol \neq null$ **then**
                return sol, exp
            $buff \leftarrow buff \cup \{n'\}$
    if $|buff| \geq K$ **or** $|frontier| = 0$ **then**
        EvalBatch$(buff, frontier)$
return $null, exp$

Algorithm 2 Bidirectional Batch Best-First Search

Input:

Start state s^1, goal state s^*, action functions A_F, A_B, transition functions T_F, T_B.
Priority functions f_F, f_B.
Batch evaluation parameter K, expansion budget *budget*.

Output: Solution *sol*, number of expansions e.

$exp \leftarrow 0$
$n_F^1 \leftarrow \textsc{CreateNode}(s^1, null, null, F)$
$frontier_F \leftarrow \{n_F^1\}$
$visited_F \leftarrow \{n_F^1\}$
$n_B^1 \leftarrow \textsc{CreateNode}(s^*, null, null, B)$
$frontier_B \leftarrow \{n_B^1\}$
$visited_B \leftarrow \{n_B^1\}$
$dir, otherdir \leftarrow F, B$
while $exp < budget$ **do**
    $n \leftarrow$ random node from $\arg\min\{n'.f \mid n' \in frontier_{dir}\}$
    $frontier_{dir} \leftarrow frontier_{dir} \setminus \{n\}$
    $exp \leftarrow exp + 1$
    $successors \leftarrow \emptyset$
    for $a \in A_{dir}(n.state)$ **do**
        $n' \leftarrow \textsc{CreateNode}(T_{dir}(n.state, a), n, a, dir)$
        if $n' \notin visited_{dir}$ **then**
            $visited_{dir} \leftarrow visited_{dir} \cup \{n'\}$
            $sol \leftarrow \textsc{CheckGoal}(n', visited_{otherdir})$
            if $sol \neq null$ **then**
                return sol, e
            $buff \leftarrow buff \cup \{n'\}$
    if $|buff| \geq K$ **or** $|frontier_{otherdir}| = 0$ **then**
        $\textsc{EvalBatch}(buff, frontier_F, frontier_B)$
    $dir, otherdir \leftarrow otherdir, dir$
return $null, exp$

2.4 Priority Functions

In the following subsections, unless otherwise stated, our discussion will focus on the UniBFS setting, where the priority functions we consider were originally introduced. We will explicitly mention when discussing their application to the BiBFS setting.

For our purposes, given a PF instance with graph $\mathcal{G}$, we consider a priority function to be a real-valued function defined on the nodes of $\mathcal{G}$ which may depend on: (i) the sub-tree of $\mathcal{G}$ constructed so far, which is defined by the *visited* set, and (ii) the goal node.

BFS algorithms may be further categorized into two classes: *uninformed* and *informed* (Pearl 1984).

Uninformed searches include algorithms such as breadth-first search (Russell & Norvig 2016) and Dijkstra's algorithm (Dijkstra 1959), the latter being applicable for graphs with arbitrary positive real-valued edge weights. The characteristic feature of uninformed BFS algorithms is that their priority functions depend solely on information about previously visited portions of the graph and do not incorporate any domain-specific knowledge or information about the goal.

In contrast, informed BFS algorithms utilize domain-specific information in their priority functions and often depend on assumptions about the unexplored portions of the graph, particularly regarding the goal. This thesis primarily focuses on comparing the learnability of parameterized priority functions for both unidirectional and bidirectional informed BFS algorithms. A secondary focus is on the efficiency of the learned priority functions.

The three algorithms we compare are based on A* Search (Hart et al. 1968), Levin Tree Search (LTS) (Orseau et al. 2018), and a version of Policy-Guided Heuristic Search (PHS*) (Orseau & Lelis 2021). It is important to note, however, that these algorithms were originally described using a unidirectional best-first search framework that differs slightly from the UniBFS algorithm we presented in Section 2.3.1. The original framework, outlined in Hart et al. (1968), has two key differences:

Assumption 2.4 The goal check for a node is performed during its expansion rather than when it is generated.

Assumption 2.5 Nodes are re-evaluated by updating their g values, $g(n)$ whenever a lower-cost path to them is discovered, and are then re-added to the frontier.

Under Assumption 2.5, $g(n)$ is not necessarily equal to the g attribute described in Section 2.3.4, and Algorithms 1 and 2. $g(n)$ is equal to the cheapest cost of a path currently discovered from the start node to n, while our g attribute is the cost of the first path discovered from the start node to n.

To facilitate the following discussion, we define three key quantities:

Definition 2.3 • $d(n, n')$: The cost of the cheapest path between nodes n and n'.

- $g^*(n) \coloneqq d(n^1, n)$: The cost of the cheapest path from the start node to n.
- $h^*(n) \coloneqq \min_{n^* \in \mathcal{N}^*}\{d(n, n^*)\}$: The cost of the cheapest path from n to any goal node.

These quantities are intrinsic to the problem instance (PF or CP) and are independent of the search algorithm. In contrast, $g(n)$ depends on the order in which nodes are visited, which is additionally determined by the priority function. The algorithms we consider in this thesis all make use of $g(n)$ along with a learned quantity; they are all instances of (Uni/Bi)BFS and differ only in the quantity they learn and how it is combined with $g(n)$. In the following subsections we discuss this in more detail.

2.4.1 A* Search

A* search (Hart et al. 1968) and its variants (Pohl 1970; Korf 1985; Botea et al. 2004; Koenig et al. 2004; Hansen & Zhou 2007; Harabor & Grastien 2011) are perhaps the most widely used informed BFS algorithms. A* search makes use of a *heuristic* function $h : \mathcal{N} \to \mathbb{R}_{\geq 0}$ that estimates the lowest solution cost that can be achieved from n, that is, an estimate of h^*. There are two very important properties a heuristic function might satisfy:

1. If $h(n) \leq h^*(n)$ for all $n \in \mathcal{N}$, we say h is *admissible*. Note that this implies $h(n^*) = 0$ for any goal node $n^* \in \mathcal{N}^*$.
2. If $h(n) \leq d(n, n') + h(n')$ for all $n, n' \in \mathcal{N}$, we say h is *consistent*. It can shown easily be shown that consistency implies admissibility. Consider any goal node $n^* \in \mathcal{N}^*$, then we have $h(n) \leq d(n, n^*) + h(n^*) = h^*(n)$ (Pearl 1984). The converse is not true.

The priority function used by A* is $f(n) = g(n) + h(n)$ for a given heuristic h. Thus, $f(n)$ can be interpreted as an estimate of the smallest solution cost achievable from n, considering the path already taken to n. For excellent discussions of A* in its original setting, we refer the reader to (Hart et al. 1968; Pearl 1984; Dechter & Pearl 1985; Edelkamp & Schrödl 2011; Russell & Norvig 2016). Briefly, in its original formulation, even relaxing the assumption of unit edge weights, A* with an admissible heuristic is guaranteed to find an optimal solution. Furthermore, A* with a consistent heuristic is additionally guaranteed to expand the minimum number of nodes when finding an optimal solution, with a caveat: A* may expand an arbitrary number of nodes that have an f value equal to the optimal cost (Pearl 1984; Holte 2010; Russell & Norvig 2016).

While it is interesting to discuss how these guarantees change with the UniBFS framework described in this thesis, the learning setting we are primarily interested in does not guarantee consistent or even admissible heuristics, and we omit that discussion here. Of more interest to us is finding any solution while expanding only a small number of nodes. Perhaps the most widely used heuristic search algorithm towards this end is Weighted A* Search (WA*) (Pohl 1970).

WA* uses the priority function

$$f_A(n) = g(n) + w \times h(n),$$

for some $w > 1$. When h is admissible, WA* guarantees that the solution it returns has cost no more than w times the optimal cost. Ebendt & Drechsler (2009) provides a comprehensive treatment on WA*. For our setting, however, where we have no bounds on the degree of inadmissibility and do not seek bounded suboptimal solutions, our primary interest is in WA*'s ability to reduce the number of expansions. Wilt & Ruml (2012) empirically show that WA* can increase or decrease the number of expansions in the satisficing setting; their main finding is that while increasing w typically results in fewer expansions up to a point, when $h(n)$ is poorly correlated with the true path length from n to the nearest goal (here we mean path length and not the general cost), increasing w results in requiring *more* expansions. Because of this, we will perform preliminary experiments in each domain we consider to determine empirically whether $w = 1$ or $w = 2.5$ performs better (Appendix A.3).

While bidirectional searches employing the A* priority function have been extensively studied, this research typically focuses on optimal and bounded suboptimal settings, requiring more complex termination criteria than those described in our BiBFS framework. Although not central to our current focus, we direct the reader interested in this area to Sturtevant & Felner (2018), which discusses this among other cost optimal bidirectional search algorithms.

Throughout the rest of this thesis, we will use the terms A* and BiA* to refer to the application of the priority function $f_A(n) = g(n)+w\times h(n)$ in the batch BFS frameworks outlined in Algorithm 1 and Algorithm 2, respectively, unless explicitly stated otherwise.

2.4.2 Levin Tree Search

LTS (Levin Tree Search) (Orseau & Lelis 2021) was initially proposed under Assumption 2.4. This algorithm employs a policy to direct the search process, which we define as follows:

Definition 2.4 Given a CP instance with states $\mathcal{S}$ and actions $\mathcal{A}$, a policy is a function $\pi : \mathcal{S} \to \Delta(\mathcal{A})$. That is, π defines for each state $s \in \mathcal{S}$ a probability distribution over actions, and we write $\pi(a|s)$ to denote the probability that $\pi(s)$ assigns to action a.

A policy π induces a probability distribution over all paths beginning from the start state,

$\mathbb{P}((a_i)_{i=1}^n) = \prod_{i=1}^n \pi(a_i|s_i)$ where $s_1 = s^1$ and $s_i = T(s_{i-1}, a_{i-1})$ for $i = 2, \ldots, n$.

Consider a CP instance, its associated PF instance, and a policy π. For each node n that is reachable from the start node n^1 there exists a non-empty set of paths $\mathcal{P}_n$ such that each $p \in \mathcal{P}_n$ begins at n^1 and terminates at n. We define the probability of each of these sequences as the probability of the same sequence of actions in the CP instance, defined above.

LTS expands nodes in order of the quantity $\min_{p \in \mathcal{P}_n} (|p| + 1)/\pi(p)$, which we call the Levin cost of node n. The minimum exists for the following reasons:

1. Every path $p \in \mathcal{P}_n$ containing a cycle will have $(|p|+1)/\pi(p) > (|p'|+1)/\pi(p')$, where $p' \in \mathcal{P}_n$ is p with the cycle removed (the numerator decreases while the denominator cannot increase). Thus the path of minimum Levin cost must be acyclic.

2. Since the graph associated with the PF instance has finitely many nodes and edges, for each node n there are finitely many acyclic paths beginning from n^1 and terminating at n.

Thus, this minimum exists, and the minimizing path $p \in \mathcal{P}_n$ must contain n exactly once. Orseau et al. (2018) show that using the priority function

$$f_L(n) = \frac{g(n) + 1}{\pi(n)},$$

where, with a slight abuse of notation, $\pi(n)$ is defined to be $\pi(p)$: the probability of the path p of length $g(n)$ that was first taken to visit n, results in n being expanded at its Levin cost. Therefore, upon first visiting each node n, we have $f_L(n) = (g(n)+1)/\pi(n) = (|p|+1)/\pi(p)$, where $p \in \arg\min_{p' \in \mathcal{P}_n} (|p'| + 1)/\pi(p')$.

They also prove that when expanding node n, no more than $f_L(n)$ nodes have been expanded in total. Applied to the set of goal nodes, this implies that no more than $\min_{n \in \mathcal{N}^*} f_L(n)$ nodes are expanded before a goal node is expanded. While this bound was derived in the setting where a node n is goal checked during its expansion (Assumption 2.4), it still holds in our setting where n is goal checked during visitation.

Consequently, this priority function provides a direct bound on the number of expansions needed to visit a goal node, dependent only on the probability assigned to paths terminating at that node. Once a solution is discovered, we can optimize this bound with respect to the policy.

In our BiBFS setting, where each search direction will have its own policy defining its priority function, using the previous bound together with Theorem 2.1, we have that BiBFS will expand no more nodes than 2 times the minimum Levin cost of any goal node in either search direction.

Throughout the rest of this thesis, we will use the terms LTS and BiLTS to refer to the application of the priority function $f_L(n) = (g(n) + 1)/\pi(n)$ in the batch BFS frameworks outlined in

Algorithm 1 and Algorithm 2, respectively, unless explicitly stated otherwise.

2.4.3 Policy-Guidied Heuristic Search

Orseau & Lelis (2021) subsequently introduced Policy-Guided Heuristic Search and its variant PHS*, which we focus on in this thesis. PHS* extends LTS to incorporate a heuristic h in addition to the policy. The priority function for PHS* is:

$$f_P(n) = \frac{g(n) + 1 + h(n)}{\pi(n)^{1+h(n)/(g(n)+1)}}.$$

This function simplifies to the LTS priority function when h equals zero everywhere. PHS* yields a bound comparable to that of LTS, but with an additional scaling factor that may be less than, equal to, or greater than 1. Unlike LTS, practically optimizing π or h to directly minimize this bound is computationally prohibitive. In their study, Orseau & Lelis (2021) employed the same loss functions we will use in this work (Section 3.4). They found that PHS*, utilizing learned heuristics and policies within a UniBFS framework, empirically outperformed LTS in two out of three tested domains.

Throughout the rest of this thesis, we will use the terms PHS* and BiPHS* to refer to the application of the priority function $f_P(n) = (g(n) + 1 + h(n))/\pi(n)^{1+h(n)/(g(n)+1)}$ in the batch BFS frameworks outlined in Algorithm 1 and Algorithm 2, respectively, unless explicitly stated otherwise.

Chapter 3

Experimental Setup

All of our experiments consist of using either a unidirectional or bidirectional batch BFS algorithm, described in Algorithms 1 and 2, within the learning framework to be described in Section 3.5 (Algorithm 3), using one of the domains described in Sections 3.1 and 3.2.

3.1 Domains

Using our definition of CP, a domain is defined as a set of CP instances that have the same set of states, action function, and transition function, but may differ in their start state and goal states. In the following subsections we describe the domains under our consideration. Each domain will be described in terms of its CP components.

3.1.1 Sliding Tile Puzzle

In the Sliding Tile Puzzle (STP) domain (Gates & Papadimitriou 1979; Korf & Taylor 1996; Heydari & Sudborough 1997; Chitturi et al. 2009; Bulteau et al. 2015; Valenzano & Yang 2017) we consider *square* grids of width n cells. The states are all configurations of the grid with the n^2 cells labelled with the integers $0, 1, \ldots, n^2 - 1$. Given a state, the valid actions are the directions {up, down, left, right} in which there is a cell adjacent to the where 0 is, the result of taking an action swaps the 0 label with the label in that cell. The single goal state is the configuration with 0 in the top left cell and the remaining cells labelled in ascending order from left to right and top to bottom with the integers $1, \ldots, n^2 - 1$. The start states are those states that can be reached from the goal state by taking valid actions; this ensures that all instances are solvable. Figure 3.1 depicts this domain.

In this domain we use a simple backward graph.

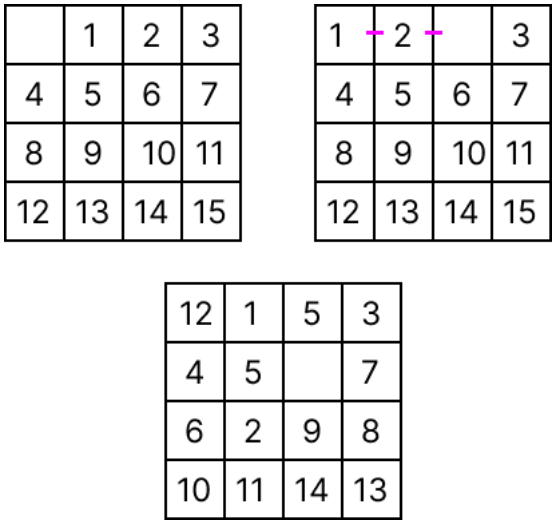

Figure 3.1: Example of a STP of width of 4. The top left figure shows the goal state. The top right figure shows a state two actions away from the goal. Swapping the blank tile with the tile to its left twice, as indicated by the purple lines, results in a goal state. The bottom figure shows a random start state.

3.1.2 The Witness

We consider two domains based on puzzles from the Witness video game. Abel et al. (2020) and Stevens et al. (2023) discuss such puzzles in more detail. Here, we only briefly describe the domains used in our experiments. First, we will describe the shared aspects between these domains, then discuss the particularities of each in the following subsections.

The Witness puzzle domains consist of a square grid of width n cells. In each Witness type domain, each cell can contain one object from a set of object types defined by the domain. Each start state is a configuration with a placement of objects in the cells, and empty cell borders. We can visualize the actions as drawing a line that traces the borders of the cells, beginning at the bottom left corner of the bottom left cell. The valid actions are the directions {up, down, left, right} where extending the line will not intersect with itself or leave the grid. The result of taking an action extends the line; the objects within the cells remain fixed. The goal states depend on the particular domain and the placement of the objects within the cells, however in each instance there is a predefined location on the outer border of the grid where the line must terminate. In our experiments, the required ending location of the line varies from instance to instance, but the starting location is fixed.

In these domains we do not use a simple backward graph. Rather, the nodes of the backward graph correspond to a different set of states $\mathcal{S}_B$. Specifically, states in $\mathcal{S}_B$ consist of the same grid and placement of objects, except now the line segment is visualized as beginning from the location where it must terminate in a forward goal state. The valid actions and result of applying them are

the same as in the forward direction. Figure 2.2 illustrates a forward and backward state of the same instance.

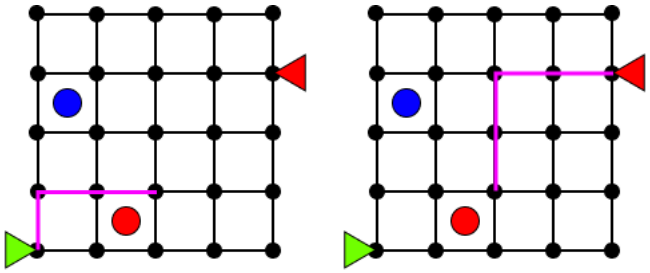

Figure 3.2: Example of a Witness type puzzle of width 4 forward and backward state. The green and red triangles indicate where the line must start and end in a forward goal state, respectively. In contrast, the line must begin at the red arrow in the backward states. The left part of the figure depicts a forward state, with valid actions {up, down, right}. The right part of the image depicts a backward state, with valid actions {down, left, right}. These two states together satisfy the goal of a Witness Colours Puzzle (Section 3.1.2.2) instance in a BiBFS setting.

3.1.2.1 Triangles Puzzle

In the Witness Triangles (TRI) domain, each cell can be blank or contain a number of triangles. The goal states are all configurations where the line begins at the bottom left corner of the bottom left cell, terminates at the red arrow, and for each cell with a triangle in it, traces exactly as many edges of the cell as there are triangles. Figure 3.3 illustrates an example.

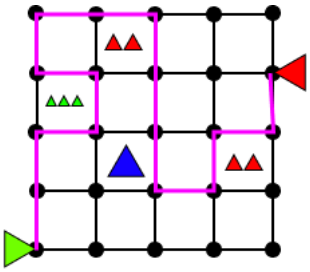

Figure 3.3: Example of a TRI puzzle of width 4 goal state. The purple begins at the green arrow and ends at the red arrow. For each cell with a triangle in it, the line traces as many edges of it as there are triangles. All configurations featuring connected segments of the purple line starting from the green triangle represent forward states that can be encountered while progressing towards this goal state. All configurations featuring connected segments of the purple line starting from the red triangle represent backward states that can be encountered while progressing towards this goal state.

3.1.2.2 Colours Puzzle

In the Witness Colours (COL) domain, each cell can contain one of three colours, or no colour. The line must divide the grid into regions so that all coloured cells within a region share the same colour. Figure 3.4 illustrates an example.

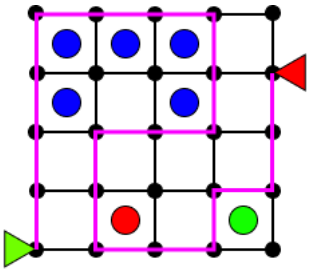

Figure 3.4: Example of a COL puzzle of width 4 goal state. The line divides the grid into three regions, one containing blue colours, one containing red colours, and one containing green colours. All configurations featuring connected segments of the purple line starting from the green triangle represent forward states that can be encountered while progressing towards this goal state. All configurations featuring connected segments of the purple line starting from the red triangle represent backward states that can be encountered while progressing towards this goal state.

3.1.3 Pancake Puzzle

The Pancake (PAN) domain has long been studied in the planning and theoretical computer science communities (Gates & Papadimitriou 1979; Korf & Taylor 1996; Heydari & Sudborough 1997; Chitturi et al. 2009; Helmert 2010; Bulteau et al. 2015; Valenzano & Yang 2017). In this domain, each each state is a permutation of n 'pancakes', each of a unique size, arranged on top of one another in a stack. The start state is a random such permutation. The valid actions in any state correspond to all but the top pancake. The result of an action flips the portion of the stack above the chosen pancake. The goal state is the configuration where the pancakes are arranged in decreasing order of size from bottom to top. Figure 3.5 illustrates an example.

In this domain we use a simple backward graph.

3.2 Datasets

For STP, TRI, and COL, we employ problems of width 4 and 5. For PAN, we utilize stacks of size 10 and 12. In each domain and size category, we distribute a set of 52,000 unique problem instances into three subsets: a training set of 50,000 problems, and validation and test sets of 1,000 problems each. The following subsections briefly outline the generation procedure used for each domain.

3.2.1 Sliding Tile Puzzle Datasets Generation

For STP width 4, we generate all problem sets ourselves using the same procedure outlined in Orseau & Lelis (2021), which is described below. For width 5, we use their exact training and test problems, while generating our own validation problems, again following their procedure.

Orseau & Lelis (2021) employ distinct generation procedures for training and test problems, omitting validation sets entirely. In our approach, we generate validation sets using the same method as test problems. For training problems, they create instances by executing random walks of 50-1000 steps (uniformly chosen) from the goal state. In contrast, validation and test problems utilize random, solvable grid permutations. This distinction serves two key purposes: it ensures a robust evaluation of learned policies across a spectrum of complexities, and it assesses how the performance generalizes to a supposedly more difficult problem distribution, consisting of problems not encountered during the training phase.

3.2.2 Witness Datasets Generation

For TRI and COL of widths 4 and 5, we employ a consistent method to generate training, validation, and test sets. The line always starts at the bottom left corner of the bottom left cell. We randomly select a distinct ending location and perform uniform random walks over valid actions until reaching it or restarting if no valid actions remain. To ensure sufficient complexity, the generated line's length must be at least twice the grid's width.

After generating a valid line, we place objects to create the start state. For COL, we first check that the line divides the grid into at least 3 regions (width 4) or 4 regions (width 5), otherwise restarting. We then choose a color for each region and apply it to each cell within that region with probability 0.75. For TRI, we consider each cell adjacent to the grid and place the number of triangles adjacent to the line with probability 0.75.

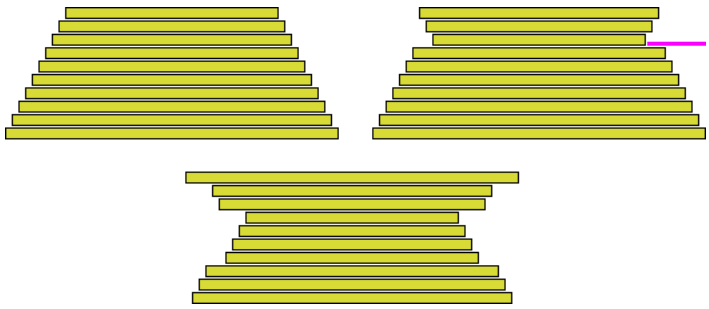

Figure 3.5: Example of a PAN puzzle of size 10. The top left figure shows the goal state. The top right figure shows a state one actions away from the goal. Flipping the stack at the purple line results in a goal state. The Bottom figure shows a random start state.

This approach generates diverse, solvable puzzles with consistent complexities across all problem sets for both domains.

3.2.3 Pancake Datasets Generation

For the PAN domain with stack sizes 10 and 12, we generate problems similarly to STP. Training problems are created by taking random walks of 50-1000 steps away from the goal state. For validation and test sets, we use random permutations of the stack. This approach allows for a diverse range of problem difficulties and enables assessment of the models' generalization capabilities.

3.3 Neural Network Architectures

In the unidirectional setting, we decompose a single network into three modules: $\psi_{\theta_f} : \mathbb{R}^n \to \mathbb{R}^d$ (feature module), $\pi_{\theta_p} : \mathbb{R}^d \to \Delta(\mathcal{A})$ (policy module), and $h_{\theta_h} : \mathbb{R}^d \to \mathbb{R}$ (heuristic module). The feature module is used by all algorithms, the policy module is used by LTS and PHS*, and the heuristic module is used by A* and PHS*. The subscripted θ's represent the parameters of these modules.

ψ_{θ_f} takes a state representation $s \in \mathbb{R}^n$ (detailed in Appendix D) and outputs an embedding $\psi_{\theta_f}(s) = x \in \mathbb{R}^d$. This embedding feeds into the policy and/or heuristic modules.

The policy module outputs an action distribution using a fully connected network with a final Softmax layer of size $|\mathcal{A}|$. Invalid action masking is optional, which Appendix A.2 empirically investigates.

The heuristic module outputs a real value, clipped to nonnegative during search/inference but unclipped for loss computation.

In the bidirectional setting, we use two independent networks: forward and backward. For TRI and COL, these have identical structures. For STP and PAN, however, the backward policy and heuristic networks are slightly enlarged. This difference arises because STP and PANhave a single, fixed goal state, allowing the forward network to implicitly encode this in its weights. The backward network, however, must handle varying start states. Thus, for these domains, we pass embeddings of both the current and start states to the backward heuristic and/or policy networks, expanding their domains to $\mathbb{R}^{2d}$. In TRI and COL, this distinction isn't necessary as the state representations encode both start and goal conditions.

We use the exact architecture described in Orseau & Lelis (2021) for the STP, TRI, and COL domains (except for enlarging the backward heuristic and policy modules for STP, as discussed). For the PAN domain, we replace the convolutional layers of the feature module with linear layers.

The exact architectures are detailed in Appendix C, and Figure 3.6 graphically depicts the

different modules in the forward and backward networks.

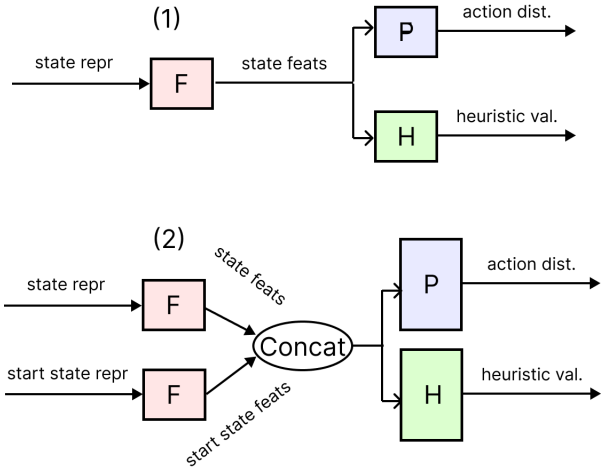

Figure 3.6: Depiction of the neural network architectures. The top figure depicts a network that takes a single state as input, and the bottom figure depicts a network that takes two states (current states and forward start state) as input. Each network has a single feature module, and either a policy module (LTS based searches), a heuristic module (A* based searches), or both a policy and a heuristic module (PHS* based searched). The network in the bottom figure uses the same feature module to process both states.

3.4 Loss Functions

We employ the same loss functions as described in Orseau et al. (2018).

- Heuristic module: given the sequence of states along a solution path, we compute the mean-squared error over it with respect to the true distances from the goal in the trajectory.
- Policy modules: given the sequence of states and actions along a solution path, we compute the average negative log-likelihood of actions along the trajectory and scale this quantity with the total number of expansions the search used when discovering this solution. In Appendix A.2, we explore the impact of masking out invalid actions.

As per Orseau & Lelis (2021), we apply L_2 regularization to both losses, and sum them together when both modules are present.

For bidirectional algorithms, we compute the above loss with full trajectories for both directions and then sum them together. In this bidirectional scenario, we scale the policy loss by the cumulative number of expansions across both search directions.

3.5 Learning Framework

We employ a learning framework similar to that of Orseau & Lelis (2021), which itself draws from Arfaee et al. (2011). Our study encompasses two problem sizes for each of the four domains described in Section 3.1.

Each search algorithm we examine uses a priority function, detailed in Section 2.4, which is parameterized by a neural network (Section 3.3). The learning algorithm accepts as one of its inputs a chosen search algorithm, with network weights randomly initialized according to a seed value. The learning process then unfolds as follows:

- The algorithm attempts to solve a batch of $B = 32$ training problems.
- The search algorithms parameters are updated using the trajectories of any solutions found during this batch.
- This process repeats for subsequent batches until all training problems have been processed.
- After each complete sweep over the training set (referred to as an epoch), validation is performed.
- The model that expands the fewest total nodes over the validation set is retained.
- Training continues for a total of $E = 10$ epochs.
- The final model, which used the smallest total number of expansions on the validation set, is used for testing at the conclusion of the learning process.

The learning algorithm's pseudocode is presented in Algorithm 3. The training and validation sets are as described in Section 3.2. *ALG* is a unidirectional or bidirectional BFS as described in Algorithm 1, Algorithm 2, and Section 2.3.4, with a particular priority function parameterized by a NN with weights θ.

The UpdateParams function shuffles the set of solution trajectories, if any are discovered, then treats each trajectory as a batch of input-target pairs and performs 10 steps of Adam (Kingma & Ba 2014) to update the NN weights.

Algorithm 3 Learning Algorithm

Input:
Training dataset $\mathcal{D}_T$, validation dataset $\mathcal{D}_V$
Search algorithm ALG
Randomly initialized priority function parameters θ.
Number of training epochs E, batch problems parameter B.
Batch evaluation parameter K, expansion budget $budget$.

Output: Final priority function parameters θ^*.

$\theta^* \leftarrow \theta$
$exp^* \leftarrow budget \times |\mathcal{D}_V|$
for $i = 1, \ldots, E$ **do**
  Shuffle $\mathcal{D}_T$
  for each batch b of size B in $\mathcal{D}_T$ **do**
    $solutions \leftarrow \emptyset$
    for $p \in b$ **do**
      $sol, \_ \leftarrow ALG(p, \theta, K, budget)$
      if $sol \neq null$ **then**
        $solutions \leftarrow solutions \cup \{sol\}$
    $\theta \leftarrow \textsc{UpdateParams}(ALG, \theta, solutions)$
  ▷ *Validation* ◁
  $exp \leftarrow 0$
  for $p \in \mathcal{D}_V$ **do**
    $\_, e \leftarrow ALG(p, \theta, K, budget)$
    $exp \leftarrow \exp + e$
  if $exp < exp^*$ **then**
    $exp^* \leftarrow exp$
    $\theta^* \leftarrow \theta$
return θ^*

3.6 Hyperparameters

We use the following hyperparameters in all experiments:

Search algorithms (same parameters used for training, validation, and test):

- $K = 32$: Batch expansion parameter.
- $budget$: Expansion budget. 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.

Learning algorithm:

- Random seeds $7, 17, 31, 53, 97$: these determine the NN's initialization, the shuffling of the training dataset in each epoch, and the shuffling of each batch of solutions before updating

the model.

- $B = 32$: Number of problems per batch.
- $E = 10$: Number of training epochs.

Optimization parameters:

- 0.001: L_2 regularization parameter.
- 0.0001: Learning rate for the Adam optimizer.
- 10: Number of gradient steps per solution.

Chapter 4

Comparing Unidirectional and Bidirectional Searches

In this chapter, we present our primary comparisons. For each domain and size configuration, we compare the best-performing unidirectional version of each algorithm with its best-performing bidirectional counterpart, as identified in our preliminary studies (detailed in appendix A). In particular: (i) all bidirectional algorithms strictly alternate search directions with each expansion, (ii) all policy-based algorithms do not use invalid action masking, (iii) (Bi)A* algorithms use $w = 1$ for COL and TRI, and $w = 2.5$ for PAN and STP.

4.1 Training and Validation

Our study's primary objective is to compare the learnability of effective policies and heuristics in unidirectional versus bidirectional searches. We investigate how the quality of learned heuristics and policies evolves with respect to both the number of problems encountered and the cumulative expansions used. Our metrics for quality are the average number of problems solved and expansions used per problem. We focus particularly on learning guidance mechanisms that quickly achieve high solve rates, successfully solving the majority of training and validation problems while minimizing cumulative expansions.

Although we couldn't use time as a fair comparison due to hardware variations across experiments, we found that the number of expansions correlates strongly with time. This holds even in Witness domains where goal checking is more complex than a simple hash lookup (STP and PAN use simple backward graphs). Moreover, the average expansion time across algorithms remains relatively consistent. While the backward models in PAN and STP domains require larger architectures, as detailed in section 3.3, this discrepancy could potentially be minimized with more

efficient hardware utilization.

Our results demonstrate that across all base algorithms, domains, and domain sizes, bidirectional searches learn effective guidance mechanisms more quickly than their unidirectional counterparts. This quickness is measured in terms of both problems seen and cumulative expansions. The learned bidirectional mechanisms either nearly match, exceed, or significantly outperform their unidirectional counterparts in terms of solve rate. Furthermore, in all cases except STP 5, the bidirectional variants use fewer expansions to solve problems.

These findings support our hypothesis that learning policies and heuristics is indeed easier in a bidirectional setting, as evidenced by the reduced total number of expansions and problems required during the learning process.

We propose several conjectures to explain these results: (i) bidirectional search may yield more diverse and informative training data through different solution distributions, (ii) the ability to discover solutions more quickly in bidirectional search could accelerate learning and improve performance, (iii) bidirectional search may effectively reduce the search depth, simplifying the learning task for policies and heuristics.

Figures 4.1 to 4.8 illustrate the performance of all algorithms on a particular domain.

The left columns display solve rates (top), and average expansions per problem (middle), versus batch over all training batches (all epochs). For each batch, we calculate the average number of problems solved and the average number of expansions used per problem. We smooth the data using a 500-point fixed-window simple moving average. Solid or dashed lines represent the median over 5 random seeds, while shaded areas show the min and max. Vertical dotted lines indicate the ends of each epoch.

The right columns show solve rates (top), and average expansions per problem (middle), at the end of each training epoch on the validation set. Solid or dashed lines represent the median over 5 random seeds, with shaded areas showing the min and max.

For solve rates versus batch (top left), the bidirectional variants outperform or nearly match their unidirectional counterparts in every domain and size, often significantly. In all domains, the solve rates grow quicker for the bidirectional variants. In the TRI and COL domains, BiA* achieves a significantly greater solve rate at the end of training compared to A*. In the PAN 12 domain, LTS and PHS* barely solve any problems until the last few epochs, where some seeds rapidly learn, and others still solve barely any problems. In contrast, BiLTS and BiPHS* achieve nearly perfect solve rates within the first epoch. In this domain, A* is not able to learn at all, while BiA* also achieves a perfect solve rate within the first epoch. We discuss a possible explanation for this later on in this section.

The results on the validation set (top right) validate these trends.

For expansions versus batch (middle left), the results are identical except for one notable exception: in STP 5, the heuristic-based unidirectional algorithms, A* and PHS*, consistently use slightly fewer average expansions per problem compared to their bidirectional counterparts. Furthermore, the different between the unidirectional and bidirectional A* based searches is greater than that of the PHS* based searches, indicating that it may be the interplay of the heuristics with bidirectional search that leads to the poorer expansion efficiency.

The bottom plots display the solve rate versus cumulative expansions over all 10 training epochs, which is a measure of how quickly the algorithms are able to learn. For each batch of 32 problems, we calculate the average number of problems solved and the cumulative number of expansions used in this and all prior batches. These results are averaged over 5 random seeds and smoothed using a 500-point fixed-window simple moving average. Here, we showcase the best-performing version of each algorithm for the particular domain, as determined in appendix A. (Bi)A* is displayed with green lines, (Bi)LTS with orange lines, and (Bi)PHS* with purple lines. The unidirectional variants are displayed with solid lines, and the bidirectional variants use dashed lines.

These plots also show that the bidirectional variant of each algorithm always learns quicker in terms of the cumulative number of expansions used. The difference is significant in all domains, and especially pronounced in PAN domains.

We note that the large differences in the PAN domains are likely due to the fixed expansion budget not being adequate for the unidirectional searches. However, in smaller scale experiments with an even smaller expansion budget, we observed the same trend. This result may provide support for conjectures (ii) and (iii).

Another interesting trend is that in the STP and PAN domains, which are similar in that we use simple backward graphs for both, and their state spaces are both permutation sets, the ordering of performance from best to worst is generally BiA*, BiPHS*, LTS, while the situation is reversed in the Witness domains. These results are similar to those of Orseau & Lelis (2021), where they considered only unidirectional variants, and it was observed that learning a heuristic versus a policy is easier in STP domains, and the situation is reversed in the Witness domains. This suggests the same trend may extend to the bidirectional setting.

In summary, our results consistently demonstrate that bidirectional variants outperform their unidirectional counterparts across various domains and sizes, both in terms of solve rates and learning speed during training. Bidirectional algorithms generally achieve higher solve rates more quickly, measured by both problems seen and cumulative expansions. This advantage is particularly pronounced in the PAN domain. However, in the STP domain, unidirectional heuristic-based algorithms solve problems using slightly fewer expansions, though bidirectional variants still achieve similar solve rates more rapidly. These observations support our initial hypothesis that learning policies and heuristics is easier in a bidirectional setting.

These findings suggest that bidirectional search approaches often offer significant advantages in learning efficiency and problem-solving performance for classical planning tasks, though the relative ease of learning unidirectional and bidirectional policies versus heuristics may vary depending on specific domain characteristics.

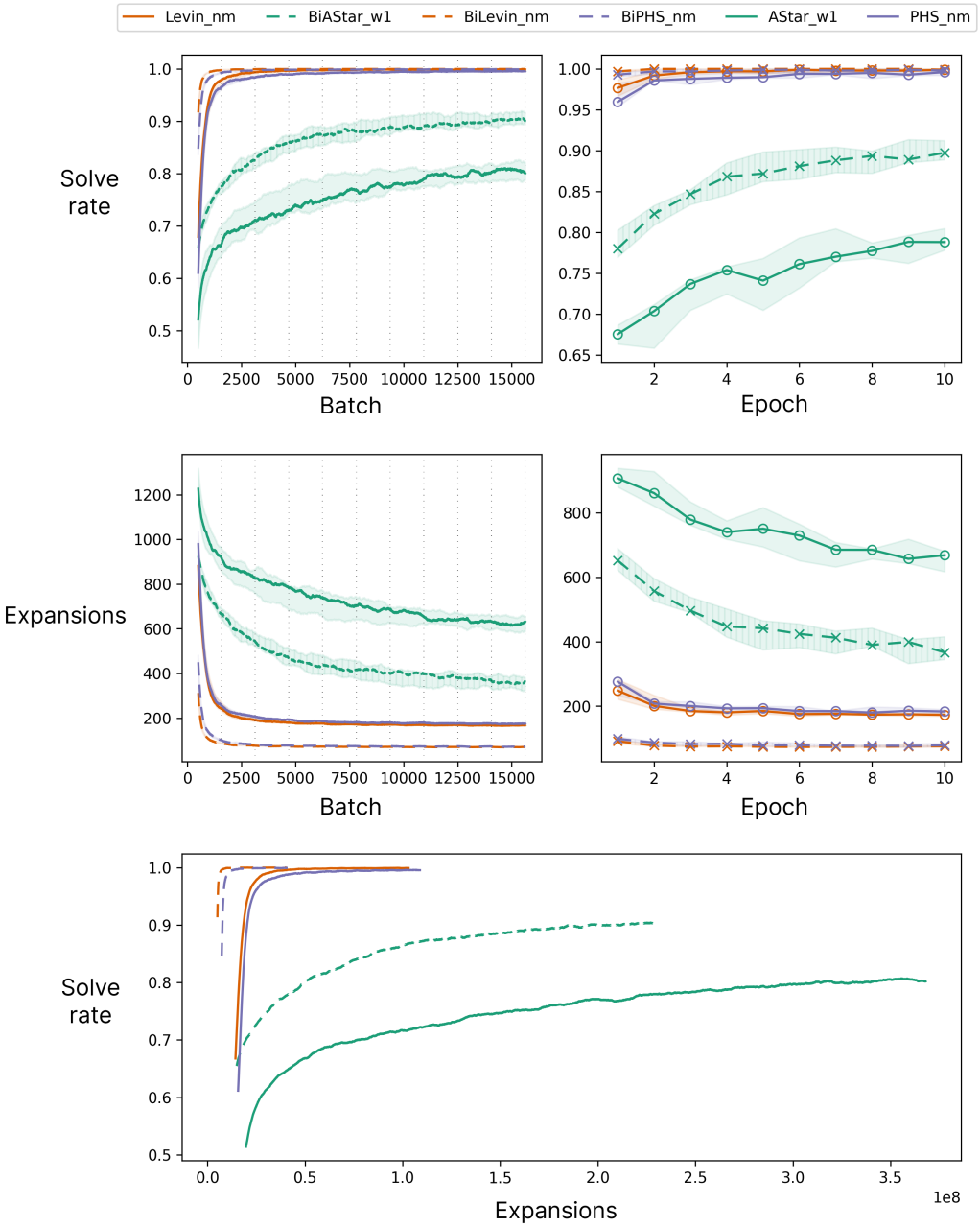

Figure 4.1: Comparison of search algorithms and their bidirectional variants on TRI width 4 during training, using an expansion budget of 2000. Top row shows solve rates vs. batches (training set, left) and epochs (validation set, right). Middle row displays average expansions vs. batches (training set, left) and epochs (validation set, right). Bottom row presents solve rates vs. total expansions. Top and middle rows show median (solid line) and range (shaded area) over 5 random seeds, while the bottom row shows the mean over 5 random seeds. All data smoothed using a 500-point fixed-window simple moving average.

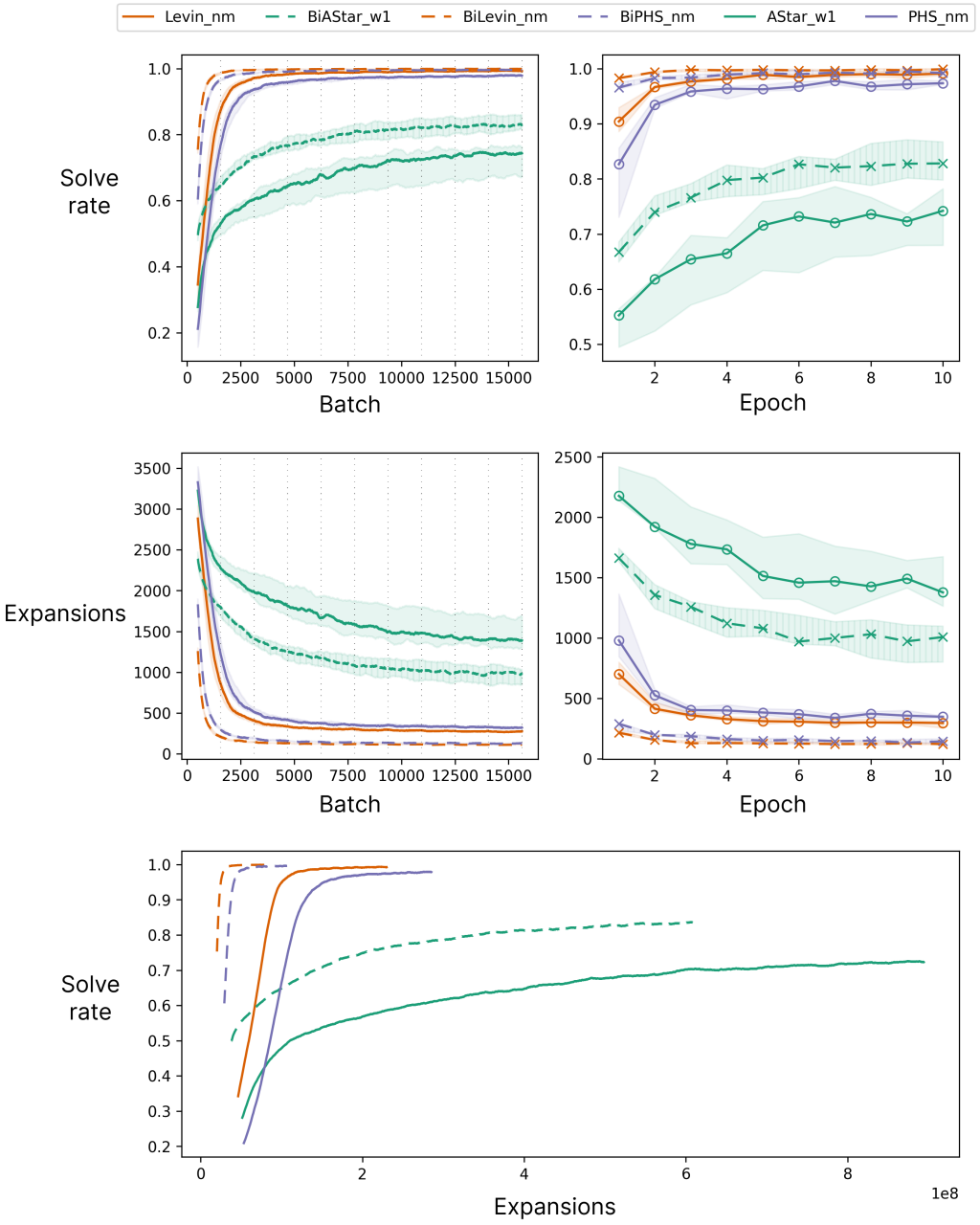

Figure 4.2: Comparison of search algorithms and their bidirectional variants on TRI width 5 during training, using an expansion budget of 4000. Top row shows solve rates vs. batches (training set, left) and epochs (validation set, right). Middle row displays average expansions vs. batches (training set, left) and epochs (validation set, right). Bottom row presents solve rates vs. total expansions. Top and middle rows show median (solid line) and range (shaded area) over 5 random seeds, while the bottom row shows the mean over 5 random seeds. All data smoothed using a 500-point fixed-window simple moving average.

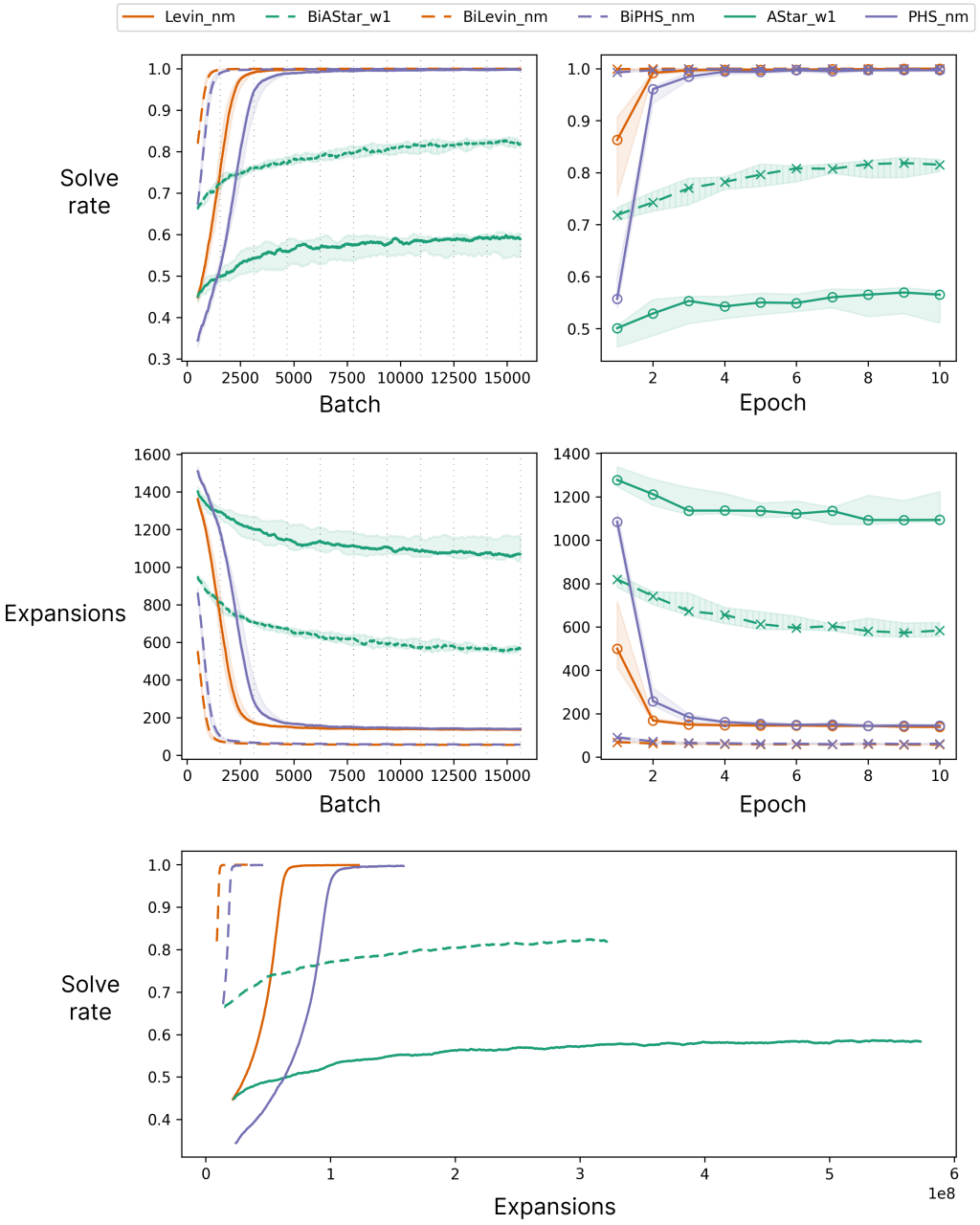

Figure 4.3: Comparison of search algorithms and their bidirectional variants on COL width 4 during training, using an expansion budget of 2000. Top row shows solve rates vs. batches (training set, left) and epochs (validation set, right). Middle row displays average expansions vs. batches (training set, left) and epochs (validation set, right). Bottom row presents solve rates vs. total expansions. Top and middle rows show median (solid line) and range (shaded area) over 5 random seeds, while the bottom row shows the mean over 5 random seeds. All data smoothed using a 500-point fixed-window simple moving average.

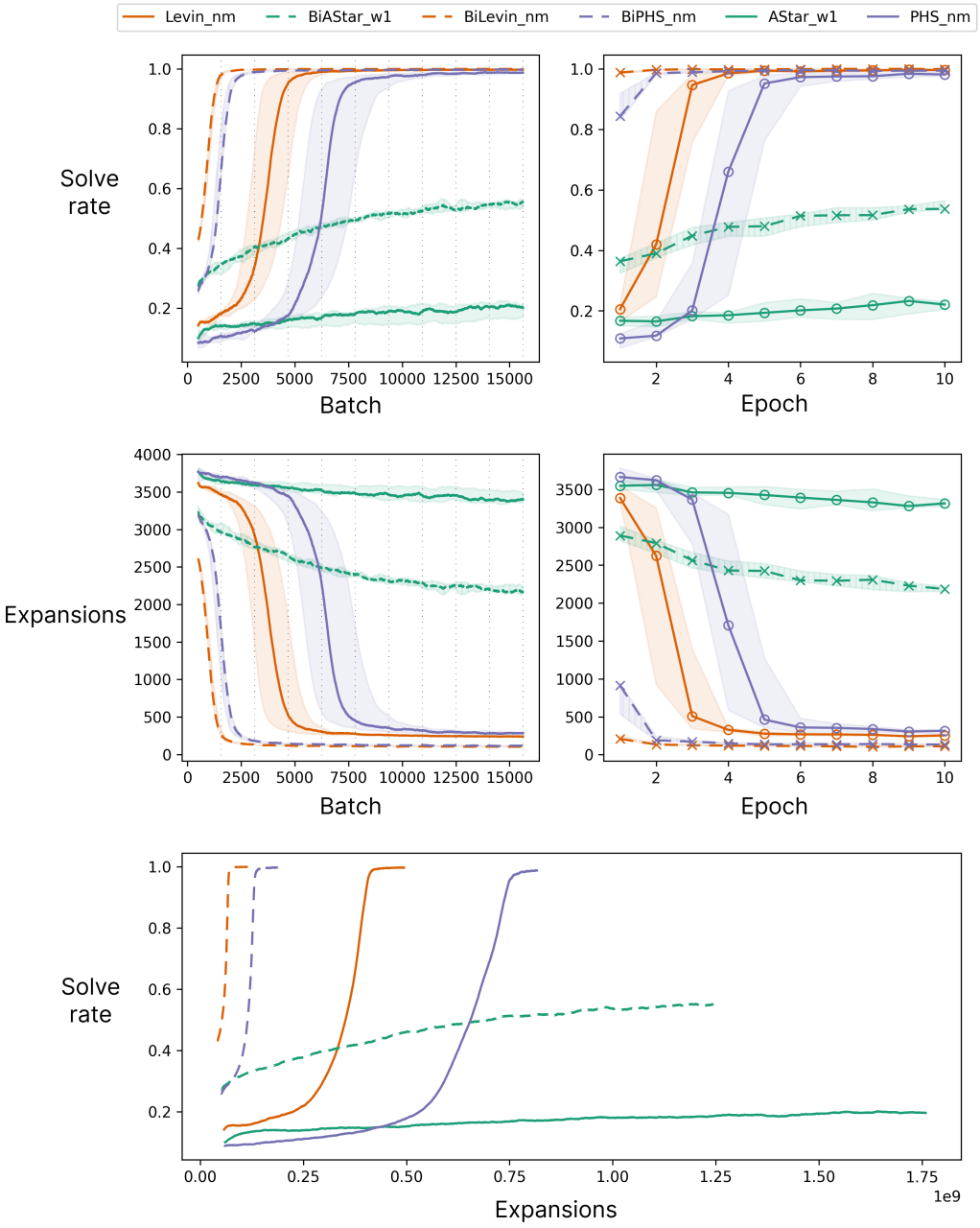

Figure 4.4: Comparison of search algorithms and their bidirectional variants on COL width 5 during training, using an expansion budget of 4000. Top row shows solve rates vs. batches (training set, left) and epochs (validation set, right). Middle row displays average expansions vs. batches (training set, left) and epochs (validation set, right). Bottom row presents solve rates vs. total expansions. Top and middle rows show median (solid line) and range (shaded area) over 5 random seeds, while the bottom row shows the mean over 5 random seeds. All data smoothed using a 500-point fixed-window simple moving average.

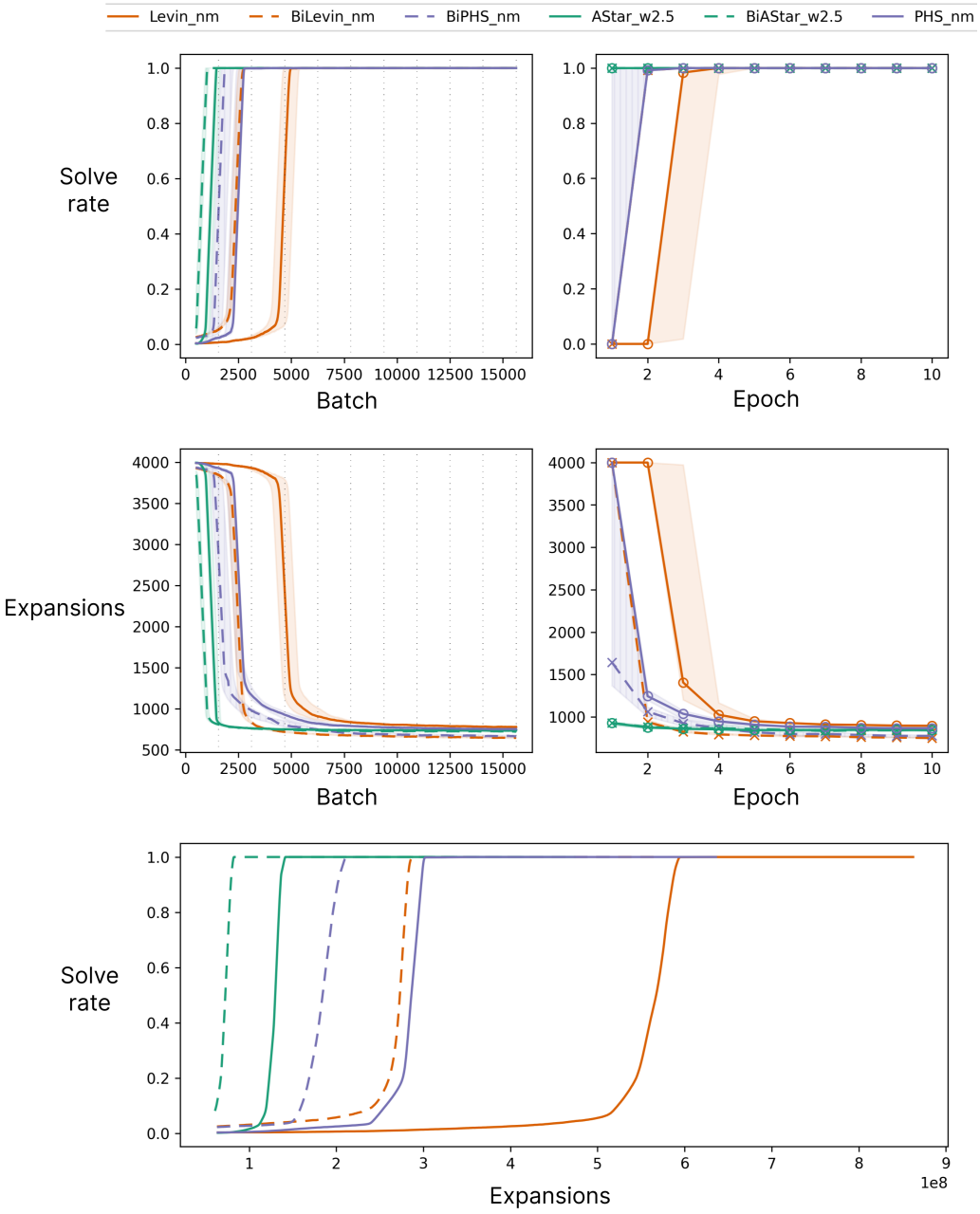

Figure 4.5: Comparison of search algorithms and their bidirectional variants on STP width 4 during training, using an expansion budget of 4000. Top row shows solve rates vs. batches (training set, left) and epochs (validation set, right). Middle row displays average expansions vs. batches (training set, left) and epochs (validation set, right). Bottom row presents solve rates vs. total expansions. Top and middle rows show median (solid line) and range (shaded area) over 5 random seeds, while the bottom row shows the mean over 5 random seeds. All data smoothed using a 500-point fixed-window simple moving average.

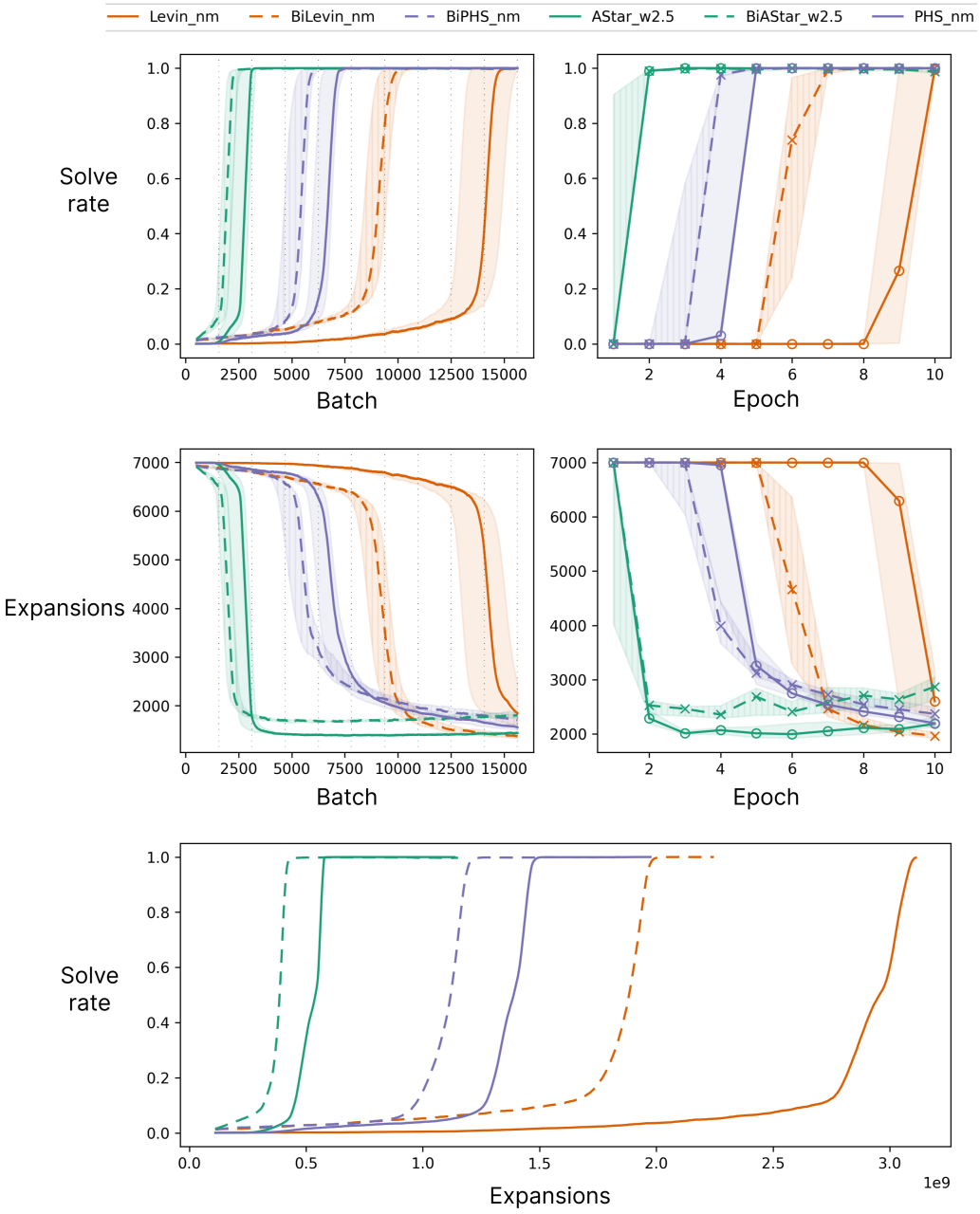

Figure 4.6: Comparison of search algorithms and their bidirectional variants on STP width 5 during training, using an expansion budget of 7000. Top row shows solve rates vs. batches (training set, left) and epochs (validation set, right). Middle row displays average expansions vs. batches (training set, left) and epochs (validation set, right). Bottom row presents solve rates vs. total expansions. Top and middle rows show median (solid line) and range (shaded area) over 5 random seeds, while the bottom row shows the mean over 5 random seeds. All data smoothed using a 500-point fixed-window simple moving average.

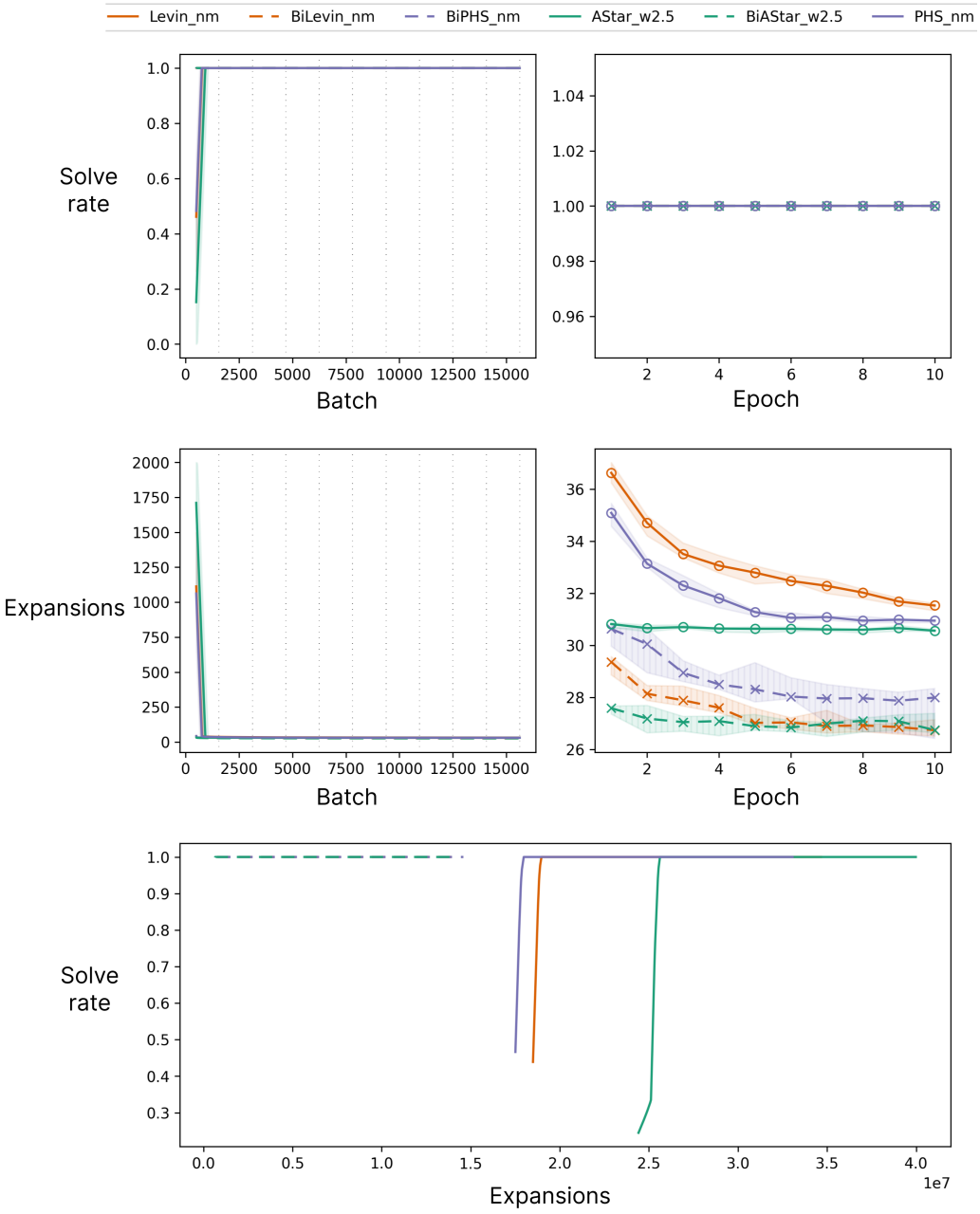

Figure 4.7: Comparison of search algorithms and their bidirectional variants on PAN size 10 during training, using an expansion budget of 2000. Top row shows solve rates vs. batches (training set, left) and epochs (validation set, right). Middle row displays average expansions vs. batches (training set, left) and epochs (validation set, right). Bottom row presents solve rates vs. total expansions. Top and middle rows show median (solid line) and range (shaded area) over 5 random seeds, while the bottom row shows the mean over 5 random seeds. All data smoothed using a 500-point fixed-window simple moving average.

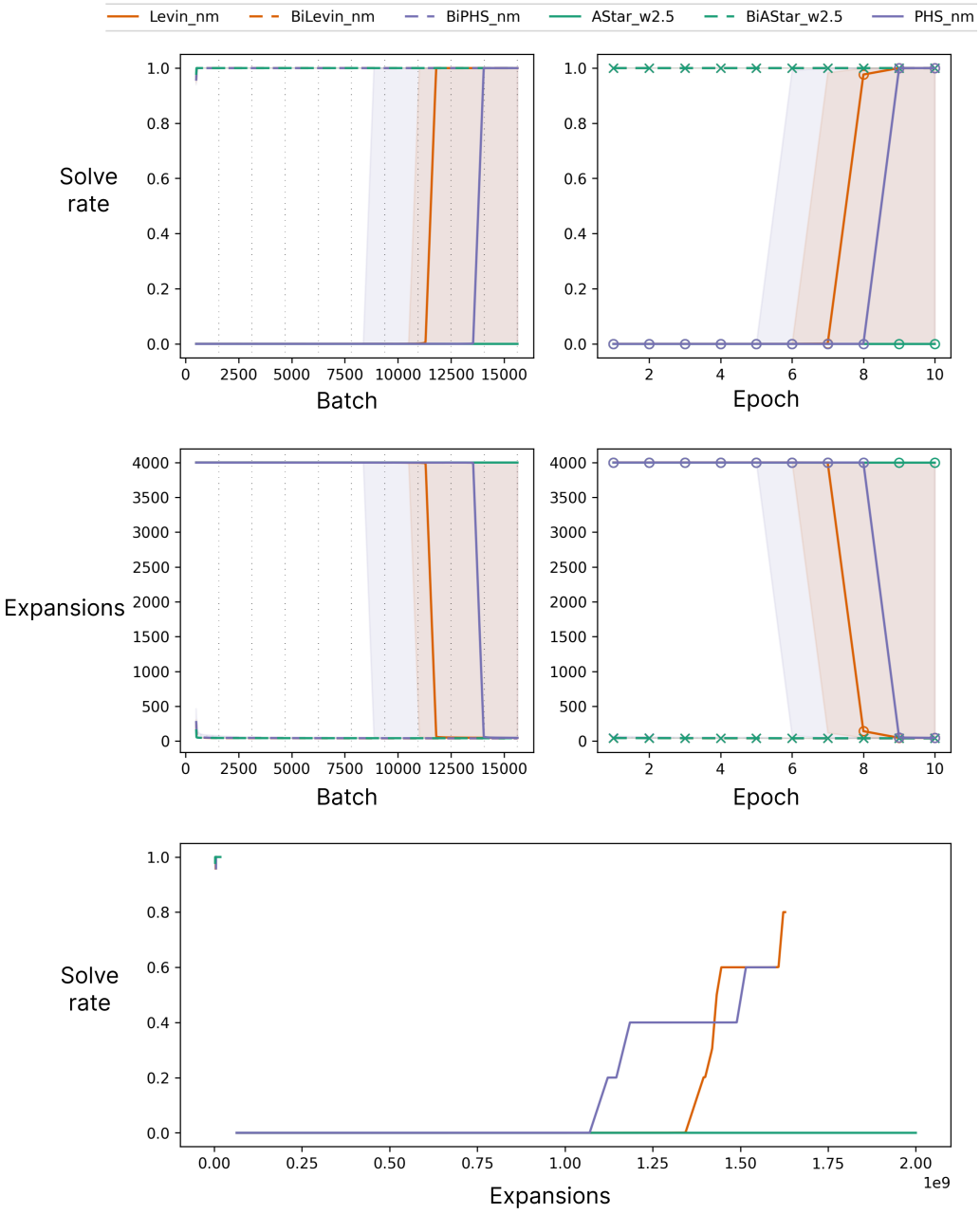

Figure 4.8: Comparison of search algorithms and their bidirectional variants on PAN size 12 during training, using an expansion budget of 4000. Top row shows solve rates vs. batches (training set, left) and epochs (validation set, right). Middle row displays average expansions vs. batches (training set, left) and epochs (validation set, right). Bottom row presents solve rates vs. total expansions. Top and middle rows show median (solid line) and range (shaded area) over 5 random seeds, while the bottom row shows the mean over 5 random seeds. All data smoothed using a 500-point fixed-window simple moving average.

4.2 Testing

In this section, we present our secondary comparisons: the quality of the learned policies and heuristics, as measured primarily by their solve rate performance and secondarily by the number of expansions they use to solve problems. These measurements are conducted on held-out test sets (section 3.2). We employ the same expansion budgets as in training (section 3.6): 2000 for TRI 4, COL 4, and PAN 10; 4000 for STP 4, TRI 5, COL 5, and PAN 12; and 7000 for STP 5.

In this section, we present comparisons of the optimal variants of each algorithm for each domain in Tables 4.1, 4.2 and B.1. These optimal variants were determined in appendix A. All data shown is averaged over 5 random seeds using during training. For each seed, we use the model that expanded the fewest total nodes on a validation set during any epoch of training.

In table 4.1 and table 4.2, the top numbers in a row represent the mean over the seeds, while the bottom numbers (in parentheses) show the standard deviation. The columns are organized in pairs, presenting unidirectional and bidirectional variants of each base algorithm. Each row corresponds to a specific domain and size configuration. Within column pairs, bold entries highlight the best mean performance. Underscored entries indicate the best overall mean performance across all algorithms for that particular domain and size. Each table uses a different subset of the test problems, which will be detailed when necessary

Table 4.1 displays the number of problems solved and average expansions per problem on all test problems, including those that were not solved.

In terms of problems solved, bidirectional variants generally outperformed or matched their unidirectional counterparts across all domains, with one exception: in STP 5, A* slightly edges out BiA*. However, BiA* surpasses A* in all other domains, with particularly notable differences in the Witness domains, especially COL. A striking example is PAN 12, where A* failed to solve any test problems, while BiA* solved all of them. As mentioned in section 4.1, this was likely due to an inadequate expansion budget for A* to solve enough problems to learn effectively. However, this observation supports our hypothesis that learning policies and heuristics is easier in a bidirectional setting. Significant performance gaps are also evident between LTS and BiLTS, as well as between PHS* and BiPHS*, in this domain.

The differences in average number of expansions used are even more pronounced, which is expected as this metric includes the average expansions on unsolved problems. In Witness and PAN domains, bidirectional algorithms used significantly fewer expansions, even when a similar number of problems was solved. Similarly to during training, which used a different problem distribution, we again we see that A* uses fewer expansions compared to BiA* in the STP domains. Also, in STP 5, the difference between A* and BiA* is again greater than that of PHS* and BiPHS*, giving more evidence that it may be the heuristic, and lack of a policy, that leads to poorer expansion

efficiency of bidirectional algorithms in this domain. On the other hand, in contrast to training, PHS* now only outperformed BiPHS* on STP 5, and not STP 4. This might suggest that as STP size grows, mixing a policy and a heuristic, compared to a pure policy based approach, leads to worse performance scaling of bidirectional versus unidirectional searches. More experiments would need to be carried out in order to test this hypothesis.

Table 4.2 presents the average number of expansions per problem and the average solution length, but only for problems commonly solved by all algorithms and all seeds. This comparison of expansions is more equitable as it only considers problems that were always solved. Although we are in a satisficing setting and solution lengths (cost) are not a primary concern, it is interesting to compare whether the solution qualities differ significantly.

In PAN 12, the unidirectional variants did not solve at least 100 problems in all seeds (note the large standard deviation in table 4.1), thus are excluded from the comparison.

In the Witness type domains, all bidirectional variants use fewer than half the expansions, with the difference between A* and BiA* being most significant, especially in the COL domains. This suggests that this domain is particularly challenging for learning unidirectional pure heuristic-based search algorithms.

In the STP domains, we again observe that unidirectional searches using a heuristic expand generally expand fewer nodes compared to their bidirectional counterparts. The exception is that PHS* only outperformed BiPHS* in STP 5, and not STP 4, as mentioned regarding table 4.1 this might suggest that as STP size grows, mixing a policy and a heuristic, compared to a pure policy based approach, leads to worse performance scaling of bidirectional versus unidirectional searches.

In all other cases (except BiPHS* in STP), whenever a bidirectional algorithm uses fewer expansions in the smaller domain size, the difference grows in the larger domain size. More experiments across more diverse domains and sizes should be done to investigate this further.

Solution lengths are very similar (identical after rounding) in the Witness type domains. However, in STP the unidirectional variants find shorter or equal length solutions in all cases except LTS in STP 5. Interestingly, BiLTS expands fewer nodes in both STP 4 and 5, but only in STP 5 does it find shorter solutions. It remains unclear whether this trend would continue as we scale the domain size. If so, it could indicate that also in terms of solution length, bidirectional pure policy-based search scaled better in this domain.

In PAN 10, the unidirectional variants found short solutions, but in PAN 12 completely failed to learn. Again, this is likely due to an inadequate expansion budget. This still shows, however, that in this domain, in our satisficing setting and learning framework, bidirectional search scaled better.

In summary, these results demonstrate that in our learning framework, the learned bidirectional guidance mechanisms generally outperformed their unidirectional counterparts in terms of problems

solved and expansions used, our primary efficiency metrics. This was particularly evident in the Witness and PAN domains. Exceptions were observed with BiA* in STP, and BiPHS* in STP 5. These results suggest that in certain domains, learned bidirectional search algorithms offer significant efficiency advantages. However, the primary objective of this thesis is to investigate whether an adequate guidance mechanism can be learned more quickly using bidirectional search, in terms of problems seen and cumulative expansions used, which is demonstrated in section 4.1. The results in this section further illustrate that the resulting learned bidirectional mechanism often outperforms its unidirectional counterpart, though performance varies depending on the domain.

		A*	BiA*	LTS	BiLTS	PHS*	BiPHS*
TRI 4	Solved	797.6 (18.4)	**901.6 (20.3)**	998.2 (1.9)	**999.2 (0.4)**	995.0 (2.8)	**998.6 (0.5)**
	Expansions	633.6 (25.3)	**366.9 (37.0)**	173.8 (7.3)	**74.6 (1.0)**	181.8 (4.5)	**75.9 (1.5)**
TRI 5	Solved	725.2 (40.5)	**831.2 (27.0)**	990.4 (3.2)	**998.0 (1.2)**	977.0 (2.9)	**992.8 (1.8)**
	Expansions	1462.9 (167.0)	**964.3 (109.2)**	278.6 (23.3)	**121.3 (4.0)**	323.6 (13.0)	**139.5 (8.1)**
COL 4	Solved	566.0 (28.9)	**819.0 (7.2)**	998.6 (0.9)	**999.2 (0.8)**	997.0 (1.2)	**998.6 (1.7)**
	Expansions	1108.5 (58.9)	**573.7 (18.0)**	138.9 (2.4)	**57.3 (2.0)**	142.5 (2.7)	**60.6 (2.1)**
COL 5	Solved	188.0 (12.6)	**535.8 (10.1)**	996.6 (0.5)	**999.4 (0.9)**	985.4 (5.3)	**996.8 (1.8)**
	Expansions	3432.7 (42.3)	**2243.4 (31.2)**	254.6 (9.2)	**109.1 (5.6)**	297.8 (22.8)	**123.7 (7.4)**
STP 4	Solved	**1000.0 (0.0)**	**1000.0 (0.0)**	**1000.0 (0.0)**	**1000.0 (0.0)**	**1000.0 (0.0)**	**1000.0 (0.0)**
	Expansions	**837.0 (4.7)**	843.5 (15.0)	887.8 (2.8)	**745.7 (5.4)**	862.5 (2.9)	**772.0 (13.8)**
STP 5	Solved	**998.8 (1.1)**	988.4 (9.0)	995.8 (8.8)	**1000.0 (0.0)**	**1000.0 (0.0)**	**1000.0 (0.0)**
	Expansions	**2181.6 (73.8)**	2793.9 (187.7)	2612.3 (376.3)	**1971.1 (40.9)**	**2210.3 (76.9)**	2414.4 (88.4)
PAN 10	Solved	**1000.0 (0.0)**	**1000.0 (0.0)**	**1000.0 (0.0)**	**1000.0 (0.0)**	**1000.0 (0.0)**	**1000.0 (0.0)**
	Expansions	30.7 (0.1)	**26.8 (0.3)**	31.6 (0.1)	**26.8 (0.3)**	31.0 (0.1)	**27.4 (0.7)**
PAN 12	Solved	0.0 (0.0)	**1000.0 (0.0)**	800.2 (446.8)	**1000.0 (0.0)**	600.0 (547.7)	**1000.0 (0.0)**
	Expansions	4000.0 (0.0)	**40.5 (0.5)**	837.1 (1766.5)	**38.6 (0.1)**	1626.2 (2166.9)	**39.3 (0.7)**

Table 4.1: Comparison of solve rate and average expansions per problem on test sets for optimal algorithm variants for each domain. Data shows means (top) and standard deviations (parenthesized, bottom) averaged over 5 random seeds. Each seed uses the model with lowest total node expansion on the validation set after any training epoch. Columns are paired to display unidirectional and bidirectional variants of each base algorithm. Bold entries indicate best mean performance within column pairs; underscored entries denote best overall mean performance. Expansion budgets used are: 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.

		A*	BiA*	LTS	BiLTS	PHS*	BiPHS*
TRI 4 (755.0)	Expansions	235.6 (244.6)	**92.5 (94.0)**	131.7 (50.3)	**52.3 (25.8)**	131.3 (50.0)	**52.1 (25.4)**
	Sol. Length	**12.0 (2.6)**	**12.0 (2.6)**	**12.0 (2.6)**	**12.0 (2.6)**	**12.0 (2.6)**	**12.0 (2.6)**
TRI 5 (642.0)	Expansions	339.9 (406.0)	**122.4 (132.4)**	154.2 (60.0)	**63.5 (24.4)**	154.3 (62.1)	**63.3 (24.6)**
	Sol. Length	**13.8 (2.9)**	**13.8 (2.9)**	13.9 (2.9)	**13.8 (2.9)**	**13.9 (2.9)**	**13.9 (2.9)**
COL 4 (494.0)	Expansions	311.9 (310.0)	**62.8 (70.2)**	81.4 (25.4)	**28.2 (11.6)**	80.6 (27.4)	**28.8 (22.2)**
	Sol. Length	**9.2 (1.6)**	**9.2 (1.6)**	**9.2 (1.6)**	**9.2 (1.6)**	**9.2 (1.6)**	**9.2 (1.6)**
COL 5 (148.0)	Expansions	668.5 (613.7)	**88.6 (78.4)**	103.6 (23.7)	**38.8 (9.5)**	103.4 (19.9)	**38.5 (9.2)**
	Sol. Length	**10.7 (1.1)**	**10.7 (1.1)**	**10.7 (1.1)**	**10.7 (1.1)**	**10.7 (1.1)**	**10.7 (1.1)**
STP 4 (1000.0)	Expansions	**837.0 (103.3)**	843.5 (123.3)	887.8 (107.1)	**745.7 (100.2)**	862.5 (106.7)	**772.0 (102.6)**
	Sol. Length	**57.1 (6.5)**	61.2 (7.4)	**64.7 (7.6)**	70.1 (8.0)	**62.7 (7.6)**	76.6 (9.8)
STP 5 (918.0)	Expansions	**2161.8 (300.2)**	2735.0 (445.4)	2581.0 (323.5)	**1960.7 (203.5)**	**2199.5 (208.6)**	2404.1 (233.7)
	Sol. Length	**146.1 (14.9)**	163.0 (16.5)	199.7 (19.8)	**172.8 (15.1)**	**173.3 (16.2)**	220.9 (20.3)
PAN 10 (1000.0)	Expansions	30.7 (5.1)	**26.8 (5.6)**	31.6 (5.5)	**26.8 (6.0)**	31.0 (5.3)	**27.4 (5.4)**
	Sol. Length	**8.8 (1.2)**	9.2 (1.3)	**9.0 (1.3)**	10.1 (1.6)	**8.8 (1.2)**	10.8 (1.4)
PAN 12 (1000.0)	Expansions		**40.5 (7.0)**		**38.6 (6.4)**		**39.3 (4.4)**
	Sol. Length		**12.3 (1.6)**		**13.2 (1.7)**		**16.1 (1.5)**

Table 4.2: Comparison of average expansions per problem and solution length for optimal algorithm variants on commonly solved test problems (across all algorithms and seeds) for each domain. Data shows means (top) and standard deviations (parenthesized, bottom) averaged over 5 random seeds. Each seed uses the model with lowest total node expansion on the validation set after any training epoch. Only algorithms solving at least 100 problems each seed are included. Columns are paired to display unidirectional and bidirectional variants of each base algorithm. Bold entries indicate best mean performance within column pairs; underscored entries denote best overall mean performance. Expansion budgets used are: 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.

Chapter 5

Related Work

Bidirectional search has been an active area of research in recent years (Sturtevant & Chen 2016; Sharon et al. 2016; Holte et al. 2017; Shaham et al. 2017; Eckerle et al. 2017; Chen et al. 2017; Sturtevant & Felner 2018; Barley et al. 2018; Hu & Speck 2022). However, these studies focus on optimal or bounded-suboptimal settings, in contrast to the satisficing setting we consider in this work. Additionally, this research does not employ a learning framework as we do.

Xie (2016), Kuroiwa & Fukunaga (2020), and Mohammad Lavasani (2024) present bidirectional algorithms for satisficing search or classical planning, but the latter two use front-to-front heuristics, whereas we consider front-to-end heuristics and policies. Furthermore, none of these papers consider learning the guiding mechanisms as we do.

There is prior work on learning policies or heuristics for search algorithms (Bramanti-Gregor & Davis 1991; Thayer et al. 2011; Arfaee et al. 2011; Pendurkar et al. 2023; Pendurkar et al. 2024; Orseau & Lelis 2021; Sakaue & Oki 2022). However, these studies focus on unidirectional search methods. Our work extends this line of research by comparing the learnability of policies and heuristics for bidirectional versus unidirectional search in satisficing classical planning.

Our research is most closely related to Orseau & Lelis (2021), which employs a similar learning framework (Arfaee et al. 2011) and introduces the PHS* algorithm that we utilize in this study. However, their work does not consider bidirectional variants. Our study builds upon this foundation by explicitly investigating the potential advantages of bidirectional search in terms of learnability and efficiency of the resulting guidance mechanisms across multiple classical planning domains.

To the best of our knowledge, our work is the first to systematically compare the learnability of policies and heuristics for bidirectional versus unidirectional search in satisficing classical planning.

Chapter 6

Conclusions

Our primary objective was to show that learning policies and heuristics for satisficing classical planning is easier in a bidirectional setting compared to a unidirectional one, in terms of requiring exposure to fewer problems and requiring fewer cumulative expansions. The results of Section 4.1, which show the learning trajectories during training, demonstrate this. In all three domains considered—Sliding Tile Puzzle, Witness Puzzles, and Pancake Puzzle—bidirectional variants achieve similar or greater solve rates, sometimes significantly so, after seeing fewer problems and using fewer cumulative expansions.

This was particularly the case in all Witness domains (Triangles and Colours), where the difference was especially pronounced between A* and BiA*, suggesting that these domains may be challenging for unidirectional learning of pure heuristic-based guidance mechanisms.

In the Pancake 12 domain, unidirectional algorithms exhibited unstable learning, with some seeds completely failing to learn (all A* seeds completely failed to learn). This was likely due to an inadequate expansion budget. However, it is notable that all bidirectional variants were able to quickly achieve perfect solve rates with this same budget. This supports our conjecture that the ability of bidirectional searches to connect solution paths quicker in terms of expansions can result in more efficient learning. In preliminary testing with half the expansion budget, bidirectional algorithms were still quickly able to learn.

These findings support our primary objective of showing the comparative ease of learning bidirectional versus unidirectional guidance mechanisms. The results suggest that bidirectional search often provides a more favorable learning environment, allowing for quicker acquisition of effective policies and heuristics.

Our secondary objective was to show that the final learned guidance mechanisms are still efficient in terms of the number of expansions required to solve a problem, and that they generalize to problems not seen during training.

During training and validation (Section 4.1), the bidirectional guidance mechanisms always used fewer expansions per problem, often significantly so, except in STP where A* and PHS* outperformed their unidirectional counterparts, though the different was only marginal.

On the test sets, we also observed that bidirectional variants require fewer average expansions per problem, but again A* outperformed BiA*in STP. Now, however, PHS* only outperformed BiPHS* on STP 5 and not STP 4. This suggests that mixing a policy and heuristic in this domain may become less effective as the problem size increases, potentially due to the increased complexity of learning both components simultaneously for larger state spaces. More experiments should be done to investigate this.

The bidirectional solve rates on the test sets were also better in all cases except that A* marginally outperformed BiA* on STP 5.

These results support our secondary objective, showing that the learned bidirectional guidance mechanisms are efficient in terms of expansions, and generalize to problems not seen during training.

We propose several explanations for the observed benefits of bidirectional search in our learning-based, satisficing classical planning setting. Bidirectional search may find different solutions than unidirectional search, yielding different distributions of trajectories used for training which potentially leads to more diverse and informative training data. The structure of the backward graph might enable finding solutions with fewer expansions, allowing for more frequent model updates. These advantages may be especially relevant early in the learning process, when untrained models result in search behavior that more closely resembles breadth-first search. Lastly, the forward and backward guidance mechanisms can learn to interact in a complementary fashion, for example by learning different features relevant to the search direction, or allowing each direction to specialize in different regions of the state space, compensating for plateaus in each other's prediction quality.

While our results are promising, they primarily serve as a foundation for further investigation into the potential benefits of bidirectional search in learning for classical planning. Future work should explore these conjectures in more detail, examining how bidirectional search characteristics influence the learning process across a wider range of priority functions, and investigate the scalability of these benefits to larger and more complex problem domains. A limitation of this study is the rigid learning framework, which uses fixed expansion budgets and datasets. It would be beneficial to explore the consistency of these findings in more flexible learning settings. Additionally, future research should investigate the significant performance differences observed in some domains when using invalid action masking and the best-first search direction selection criteria. These aspects could provide further insights into optimizing bidirectional search algorithms and their learning processes.

In conclusion, this research contributes insights to the ongoing exploration of learning in planning, raising important questions about the relationship between search strategies and the ease of

learning effective guidance. The demonstrated advantages of bidirectional search in learning policies and heuristics for satisficing classical planning open up new avenues for improving the efficiency and effectiveness of AI planning systems. Our findings show that bidirectional search often leads to faster learning of efficient guidance mechanisms, though the benefits vary depending on specific domain characteristics.

References

Abel, Zachary, Jeffrey Bosboom, Michael Coulombe, Erik Demaine, Linus Hamilton, Adam Hesterberg, Justin Kopinsky, Jayson Lynch, Mikhail Rudoy, et al. (2020). "Who witnesses The Witness? Finding witnesses in The Witness is hard and sometimes impossible." In: *Theoretical Computer Science* 839, pp. 41–102.

Agarap, Abien (2018). "Deep learning using rectified linear units (relu)." In: *arXiv preprint arXiv:1803.08375*.

Agostinelli, Forest, Stephen McAleer, Alexander Shmakov, and Pierre Baldi (2019). "Solving the Rubik's cube with deep reinforcement learning and search." In: *Nature Machine Intelligence* 1.8, pp. 356–363.

Alcázar, Vidal, Daniel Borrajo, Susana Fernández, and Raquel Fuentetaja (2013). "Revisiting Regression in Planning." In: *IJCAI*, pp. 2254–2260.

Arfaee, Shahab Jabbari, Sandra Zilles, and Robert Holte (2011). "Learning heuristic functions for large state spaces." In: *Artificial Intelligence* 175.16-17, pp. 2075–2098.

Barker, Joseph and Richard Korf (2015). "Limitations of front-to-end bidirectional heuristic search." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 29. 1.

Barley, Michael, Patricia Riddle, Carlos Linares López, Sean Dobson, and Ira Pohl (2018). "GBFHS: A generalized breadth-first heuristic search algorithm." In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 9. 1, pp. 28–36.

Bonet, Blai and Hector Geffner (1999). "Planning as heuristic search: New results." In: *European Conference on Planning*. Springer, pp. 360–372.

— (2001). "Planning as heuristic search." In: *Artificial Intelligence* 129.1-2, pp. 5–33.

Botea, Adi, Martin Müller, and Jonathan Schaeffer (2004). "Near optimal hierarchical path-finding." In: *J. Game Dev.* 1.1, pp. 1–30.

Bramanti-Gregor, Anna and Henry W Davis (1991). "Learning Admissible Heuristics while Solving Problems." In: *IJCAI*. Citeseer, pp. 184–191.

Bulteau, Laurent, Guillaume Fertin, and Irena Rusu (2015). "Pancake flipping is hard." In: *Journal of Computer and System Sciences* 81.8, pp. 1556–1574.

Chen, Jingwei, Robert Holte, Sandra Zilles, and Nathan Sturtevant (2017). "Front-to-end bidirectional heuristic search with near-optimal node expansions." In: *arXiv preprint arXiv:1703.03868*.

Chitturi, Bhadrachalam, William Fahle, Zhaobing Meng, Linda Morales, Charles Shields, Ivan Hal Sudborough, and Walter Voit (2009). "An (18/11) n upper bound for sorting by prefix reversals." In: *Theoretical Computer Science* 410.36, pp. 3372–3390.

Dechter, Rina and Judea Pearl (1985). "Generalized best-first search strategies and the optimality of A*." In: *Journal of the ACM (JACM)* 32.3, pp. 505–536.

Dijkstra, Edsger (1959). "A Note on Two Problems in Connexion with Graphs." In: *Numerische Mathematik* 1, pp. 269–271.

Doran, J. (1966). "Doubletree searching and the Graph Traverser." In: *Research Memorandum EPU*.

Ebendt, Rüdiger and Rolf Drechsler (2009). "Weighted A* search–unifying view and application." In: *Artificial Intelligence* 173.14, pp. 1310–1342.

Eckerle, Jürgen, Jingwei Chen, Nathan Sturtevant, Sandra Zilles, and Robert Holte (2017). "Sufficient conditions for node expansion in bidirectional heuristic search." In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 27, pp. 79–87.

Edelkamp, Stefan and Stefan Schrödl (2011). *Heuristic search: theory and applications*. Elsevier.

Felner, Ariel, Uzi Zahavi, Robert Holte, Jonathan Schaeffer, Nathan Sturtevant, and Zhifu Zhang (2011). "Inconsistent heuristics in theory and practice." In: *Artificial Intelligence* 175.9-10, pp. 1570–1603.

Fikes, Richard and Nils Nilsson (1971). "STRIPS: A new approach to the application of theorem proving to problem solving." In: *Artificial intelligence* 2.3-4, pp. 189–208.

Gates, William and Christos Papadimitriou (1979). "Bounds for sorting by prefix reversal." In: *Discrete mathematics* 27.1, pp. 47–57.

Geffner, Hector and Blai Bonet (2022). *A concise introduction to models and methods for automated planning*. Springer Nature.

Green, Cordell (1969). "Application of theorem proving to problem solving." In: *Proceedings of the 1st international joint conference on Artificial intelligence*, pp. 219–239.

Hansen, Eric and Rong Zhou (2007). "Anytime heuristic search." In: *Journal of Artificial Intelligence Research* 28, pp. 267–297.

Harabor, Daniel and Alban Grastien (2011). "Online graph pruning for pathfinding on grid maps." In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 25. 1, pp. 1114–1119.

Hart, Peter, Nils Nilsson, and Bertram Raphael (1968). "A formal basis for the heuristic determination of minimum cost paths." In: *IEEE transactions on Systems Science and Cybernetics* 4.2, pp. 100–107.

Helmert, Malte (2006). "The fast downward planning system." In: *Journal of Artificial Intelligence Research* 26, pp. 191–246.

— (2010). "Landmark heuristics for the pancake problem." In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 1. 1, pp. 109–110.

Heydari, Mohammad and Ivan Hal Sudborough (1997). "On the diameter of the pancake network." In: *Journal of Algorithms* 25.1, pp. 67–94.

Hoffmann, Jörg (2001). "FF: The fast-forward planning system." In: *AI magazine* 22.3, pp. 57–57.

Holte, Robert (2010). "Common misconceptions concerning heuristic search." In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 1. 1, pp. 46–51.

Holte, Robert, Ariel Felner, Guni Sharon, Nathan Sturtevant, and Jingwei Chen (2017). "MM: A bidirectional search algorithm that is guaranteed to meet in the middle." In: *Artificial Intelligence* 252, pp. 232–266.

Hou, Yueqi, Xiaolong Liang, Jiaqiang Zhang, Qisong Yang, Aiwu Yang, and Ning Wang (2023). "Exploring the use of invalid action masking in reinforcement learning: A comparative study of on-policy and off-policy algorithms in real-time strategy games." In: *Applied Sciences* 13.14, p. 8283.

Hu, Kilian and David Speck (2022). "On Bidirectional Heuristic Search in Classical Planning: An Analysis of BAE." In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 15. 1, pp. 91–99.

Huang, Shengyi and Santiago Ontañón (2020). "A closer look at invalid action masking in policy gradient algorithms." In: *arXiv preprint arXiv:2006.14171*.

Kaindl, Hermann and Gerhard Kainz (1997). "Bidirectional heuristic search reconsidered." In: *Journal of Artificial Intelligence Research* 7, pp. 283–317.

Kingma, Diederik and Jimmy Ba (2014). "Adam: A method for stochastic optimization." In: *arXiv preprint arXiv:1412.6980*.

Koenig, Sven, Maxim Likhachev, and David Furcy (2004). "Lifelong planning A*." In: *Artificial Intelligence* 155.1-2, pp. 93–146.

Korf, Richard (1985). "Depth-first iterative-deepening: An optimal admissible tree search." In: *Artificial intelligence* 27.1, pp. 97–109.

— (1993). "Linear-space best-first search." In: *Artificial intelligence* 62.1, pp. 41–78.

Korf, Richard and Larry Taylor (1996). "Finding optimal solutions to the twenty-four puzzle." In: *Proceedings of the national conference on artificial intelligence*. Citeseer, pp. 1202–1207.

Kuroiwa, Ryo and Alex Fukunaga (2020). "Front-to-Front Heuristic Search for Satisficing Classical Planning." In: *IJCAI*, pp. 4098–4105.

LaValle, Steven (2006). *Planning algorithms*. Cambridge university press.

Mohammad Lavasani, Sepehr (2024). "Anchor Search: A Unified Framework for Unbounded Bidirectional Search." MA thesis. University of Alberta.

Nicholson, T. Alastair J. (1966). "Finding the shortest route between two points in a network." In: *The computer journal* 9.3, pp. 275–280.

Nilsson, Nils (1982). *Principles of artificial intelligence*. Springer Science & Business Media.

Orseau, Laurent and Levi Lelis (2021). "Policy-guided heuristic search with guarantees." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 14, pp. 12382–12390.

Orseau, Laurent, Levi Lelis, Tor Lattimore, and Théophane Weber (2018). "Single-agent policy tree search with guarantees." In: *Advances in Neural Information Processing Systems* 31.

Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, et al. (2019). "Pytorch: An imperative style, high-performance deep learning library." In: *Advances in neural information processing systems* 32.

Pearl, Judea (1984). *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc.

Pendurkar, Sumedh, Taoan Huang, Brendan Juba, Jiapeng Zhang, Sven Koenig, and Guni Sharon (2023). "The (Un) Scalability of Informed Heuristic Function Estimation in NP-Hard Search Problems." In: *Transactions on Machine Learning Research*.

Pendurkar, Sumedh, Levi Lelis, Nathan Sturtevant, and Guni Sharon (2024). "Curriculum Generation for Learning Guiding Functions in State-Space Search Algorithms." In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 17, pp. 91–99.

Pohl, Ira (1969). *Bi-directional and heuristic search in path problems*. Tech. rep. SLAC National Accelerator Laboratory (SLAC), Menlo Park, CA (United States).

— (1970). "Heuristic search viewed as path finding in a graph." In: *Artificial intelligence* 1.3-4, pp. 193–204.

Richter, Silvia and Matthias Westphal (2010). "The LAMA planner: Guiding cost-based anytime planning with landmarks." In: *Journal of Artificial Intelligence Research* 39, pp. 127–177.

Russell, Stuart and Peter Norvig (2016). *Artificial intelligence: a modern approach*. Pearson.

Sakaue, Shinsaku and Taihei Oki (2022). "Sample complexity of learning heuristic functions for greedy-best-first and A* search." In: *Advances in Neural Information Processing Systems* 35, pp. 2889–2901.

Segovia-Aguas, Javier, Sergio Jiménez, and Anders Jonsson (2021). "Generalized planning as heuristic search." In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 31, pp. 569–577.

Shaham, Eshed, Ariel Felner, Jingwei Chen, and Nathan Sturtevant (2017). "The minimal set of states that must be expanded in a front-to-end bidirectional search." In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 8. 1, pp. 82–90.

Sharon, Guni, Robert Holte, Ariel Felner, and Nathan Sturtevant (2016). "An improved priority function for bidirectional heuristic search." In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 7. 1, pp. 139–140.

Simon, Herbert (1956). "Rational choice and the structure of the environment." In: *Psychological review* 63.2, p. 129.

Stevens, Justin, Vadim Bulitko, and David Thue (2023). "Solving Witness-type Triangle Puzzles Faster with an Automatically Learned Human-Explainable Predicate." In: *arXiv preprint arXiv:2308.02666*.

Sturtevant, Nathan and Jingwei Chen (2016). "External Memory Bidirectional Search." In: *IJCAI*, pp. 676–682.

Sturtevant, Nathan and Ariel Felner (2018). "A brief history and recent achievements in bidirectional search." In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1.

Sutton, Richard and Andrew Barto (2018). *Reinforcement learning: An introduction*. MIT press.

Thayer, Jordan, Austin Dionne, and Wheeler Ruml (2011). "Learning inadmissible heuristics during search." In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 21, pp. 250–257.

Torralba, Alvaro, Vidal Alcázar, Daniel Borrajo, Peter Kissmann, and Stefan Edelkamp (2014). "SymBA*: A symbolic bidirectional A* planner." In: *International Planning Competition*, pp. 105–108.

Valenzano, Richard, Nathan Sturtevant, Jonathan Schaeffer, and Fan Xie (2014). "A comparison of knowledge-based GBFS enhancements and knowledge-free exploration." In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 24, pp. 375–379.

Valenzano, Richard and Danniel Yang (2017). "An analysis and enhancement of the gap heuristic for the pancake puzzle." In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 8. 1, pp. 109–117.

Wilt, Christopher and Wheeler Ruml (2012). "When does weighted A* fail?" In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 3. 1, pp. 137–144.

Xie, Fan (2016). "Exploration in Greedy Best-First Search for Satisficing Planning." PhD thesis. University of Alberta.

Appendix A

Preliminary Experiments

This appendix presents the results of three preliminary experiments: comparing two criteria for selecting the direction in bidirectional search, determining the effects of invalid action masking, and selecting the appropriate (Bi)A* weight value.

The results of these experiments are shown in the plots of figures A.1 to A.6. These plots display a measure of the solve rate versus the cumulative number of expansions over all 10 training epochs, which is a measure of how quickly the algorithms are able to learn. For each batch of 32 problems, we calculate the average number of problems solved and the cumulative number of expansions used in this and all prior batches. These results are averaged over 5 random seeds and smoothed using a 500-point fixed-window simple moving average. The final results are depicted as solid or dashed lines in the plots. Each row represents one domain, with the left column showing the smaller domain size and the right column showing the larger domain size.

In the plots, bidirectional algorithm versions may include 'BFS' (solid lines) in their labels, indicating that the search direction was chosen by always expanding the node of minimal f value in either of the search directions' frontiers. When 'BFS' is omitted (dashed lines), it signifies that the search direction was strictly alternated with each expansion.

The (Bi)A* algorithms may have suffixes '_w1' (green lines) or '_w2.5' (orange lines), denoting the value of w used in the priority function $f_A(n) = g(n) + w \times h(n)$.

For the (Bi)PHS* algorithms, suffixes '_m' (green lines) or '_nm' (orange lines) indicate whether action masking was used or not, respectively (appendix A.2).

A.1 Search Direction Selection Criteria

The upper bound established in theorem 2.1 demonstrates that strictly alternating variants of BiBFS cannot perform significantly worse than their UniBFS counterparts when using the same

priority function, which in our context includes NN weights. The drawback is that if one of the priority functions is uninformative, in the sense that its expansions are not making progress towards the other frontier, then this upper bound might be attained. On the other hand, even in this scenario, utilizing a 'BFS' criterion, which consistently expands the node with the lowest f value in either frontier, has the potential to match UniBFS performance. It is not clear how will play out in our learning context, where randomly initialized priority functions initially result in more breadth-first behavior before attuning to previously encountered solutions over time. Our experiments empirically investigate the effects of each selection criterion across our domains.

Figures A.2, A.4 and A.6 reveal that strictly alternating versions of all bidirectional search algorithms, represented by dashed lines, consistently outperform their 'BFS' counterparts. Moreover, this performance disparity becomes more pronounced in domains that prove challenging for a particular algorithm (BiA*, BiLTS, or BiPHS*), characterized by some variants failing to learn. This is evident in the STP domains for all three algorithms and in the PAN domains specifically for BiPHS*.

A.2 Invalid Action Masking

In the reinforcement learning (Sutton & Barto 2018) problem, which can be seen as a generalization of the CP problem we describe in this thesis, it has been observed that masking out invalid actions when computing an action probability distribution can be beneficial (Hou et al. 2023; Huang & Ontañón 2020). Because two of the base algorithms we examine make sure of a policy (LTS and PHS*), we examine the effect that masking out invalid actions has during learning.

Consider a CP instance and a state $s \in \mathcal{S}$. The invalid actions at state s is the set $\mathcal{A} \setminus A_{dir}(s)$. By masking invalid actions, we mean that, before applying Softmax, we explicitly set the policy modules outputs corresponding to these actions to a large negative number (-1×10^9). This results in the probability of these actions effectively being set to 0. Without doing this, the policy module could assign non-negligible probabilities to these actions. Note, however that these actions could never be taken, since they are not part of set of valid actions for state s. Several differences arise when masking invalid actions or not:

- When *not* masking invalid actions, the valid actions will in general have lower probabilities compared to when masking invalid actions, particularly in the earlier stages of training when the model has not yet learned to assign 0 probability to invalid actions.
- Therefore, not masking invalid actions initially results in larger values of f_L and f_P (section 2.4, and also larger values of the negative log-likelihood of a solution during the model update.

- When *not* masking invalid actions, the models will have to learn themselves, through gradient updates, to assign 0 probability to invalid actions. Meanwhile, when masking invalid actions, the invalid action outputs will always have 0 gradient.

It is not clear how these differences will play out in our learning framework.

In figures A.3 to A.6, we observe that for the majority of domain and size configurations, an algorithm's no-masking variant (orange) outperforms its masking counterpart (green). The difference is most significant in the STP domains, especially STP 5. An interesting observation is that in the TRI domains, LTS with masking performs better, but the situation is reversed when using BiLTS. This supports one of our hypotheses that bidirectional search find different solutions compares to unidirectional search.

The other exceptions are the 'BFS' versions of BiPHS* in COL 5 and PAN 12. In COL 5, the masked variant shows a slightly higher solve rate in the early stages of learning, while in PAN 12, the difference in learning speed and final solve rate is substantial. However, these differences are less apparent in the strictly alternating versions of BiPHS* which outperform (greatly in the case of PAN 12) their 'BFS' counterparts anyway.

While we do not investigate this further, we believe it is an interesting problem for future research.

Given the overall trend favoring no-masking variants, we decide to utilize no masking for all policy-based algorithms in our subsequent comparisons of unidirectional and bidirectional counterparts. This decision ensures consistency across our experiments.

A.3 A* Weighting

Wilt & Ruml (2012) empirically demonstrate that the w value used in (weighted) A* can either reduce or increase the number of expansions, depending on the domain. Consequently, we evaluate two values of w, 1.0 and 2.5, for each domain and size configuration in both A* and BiA*. We do not explore weighting the heuristic in (Bi)PHS*, though this presents an interesting avenue for future research.

Figures A.1 and A.2 illustrate that in the STP domains, A* and BiA* variants with $w = 2.5$ (orange lines) consistently outperform their $w = 1$ (green lines) counterparts. Conversely, in the Witness type domains, $w = 1$ consistently outperforms $w = 2.5$.

The PAN domains present a more nuanced picture. In PAN 12, no A* variant learns effectively, while for BiA*, the 'BFS' variants fail to learn. However, the strictly alternating versions both rapidly achieve perfect solve rates, with $w = 2.5$ reaching this milestone slightly faster. In PAN 10, A* with $w = 1$ attains a perfect solve rate marginally quicker than $w = 2.5$. For BiA*, the

situation mirrors that of PAN 12, although the 'BFS' versions achieve approximately 20% solve rate (instead of close to 0%), with $w = 2.5$ slightly outperforming $w = 1$.

Given these findings, we will use $w = 1$ for the Witness type domains, and $w = 2.5$ for PAN and STP domains, in both A* and BiA* when comparing unidirectional and bidirectional approaches.

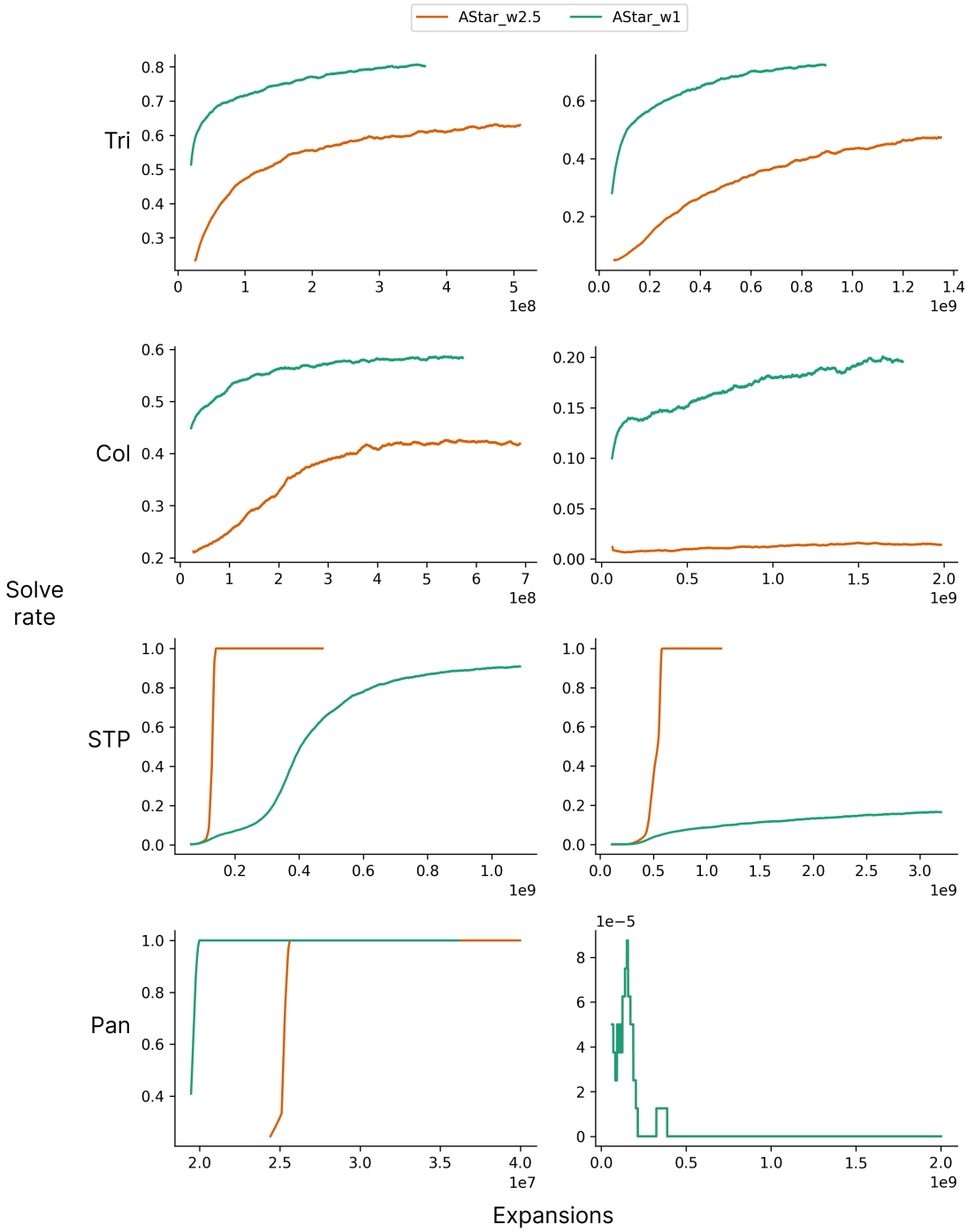

Figure A.1: Comparison of solve rates vs. total expansions for A* across domains during training. Left column shows smaller domain sizes, right column larger sizes. Lines represent mean solve rates over 5 random seeds. All data smoothed using a 500-point fixed-window simple moving average. Expansion budgets used are: 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.

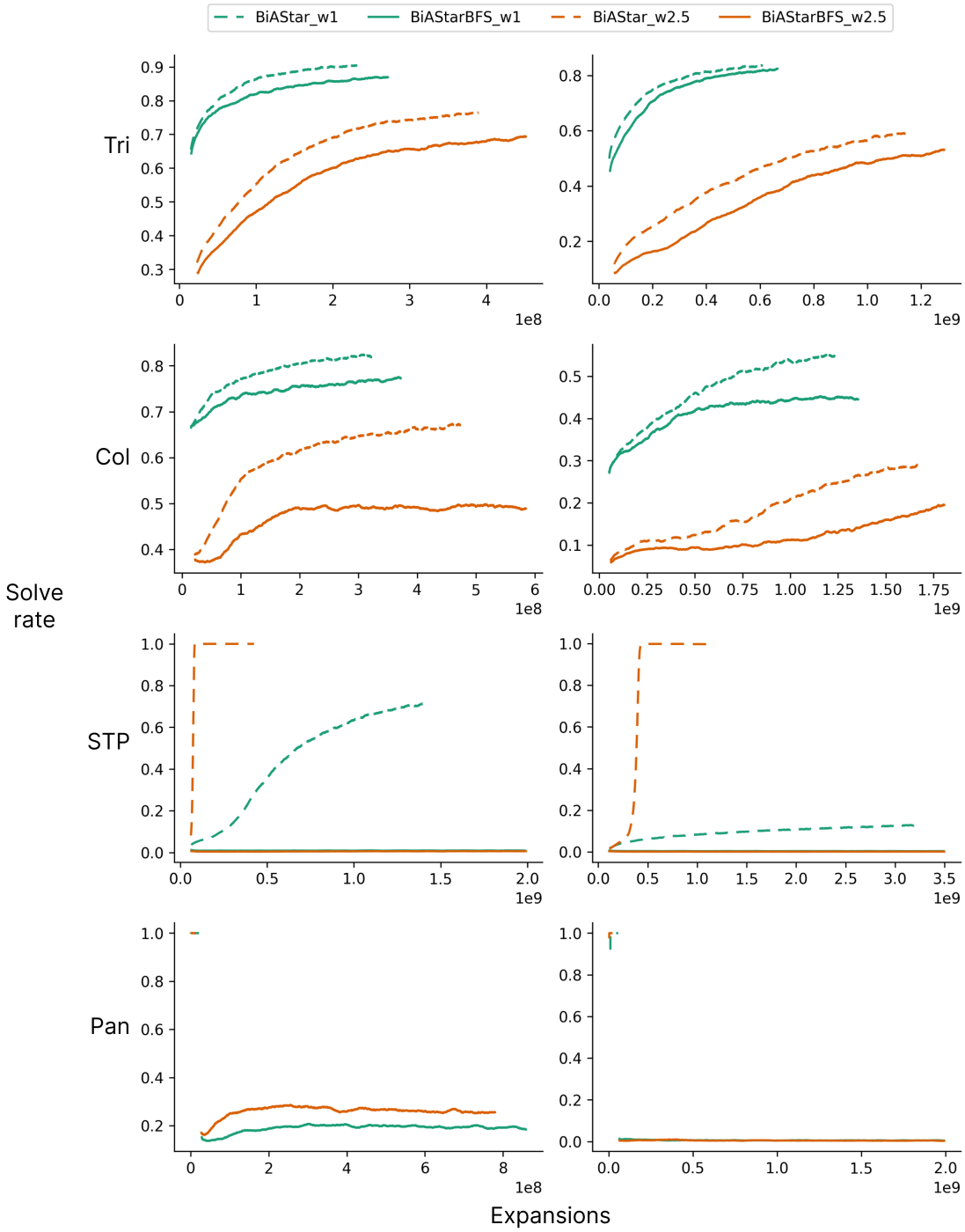

Figure A.2: Comparison of solve rates vs. total expansions for BiA* across domains during training. Left column shows smaller domain sizes, right column larger sizes. Lines represent mean solve rates over 5 random seeds. All data smoothed using a 500-point fixed-window simple moving average. Expansion budgets used are: 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.

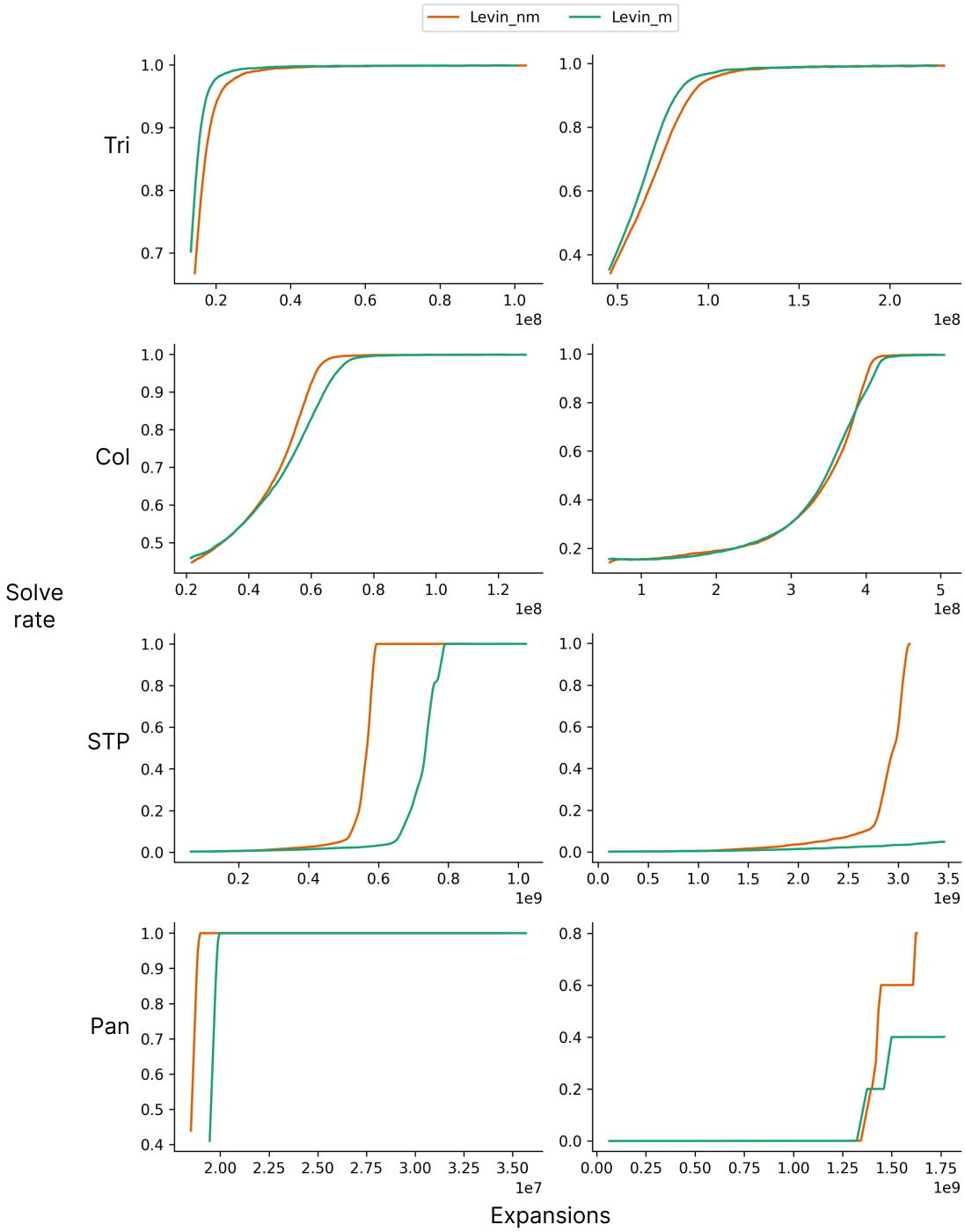

Figure A.3: Comparison of solve rates vs. total expansions for LTS across domains during training. Left column shows smaller domain sizes, right column larger sizes. Lines represent mean solve rates over 5 random seeds. All data smoothed using a 500-point fixed-window simple moving average. Expansion budgets used are: 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.

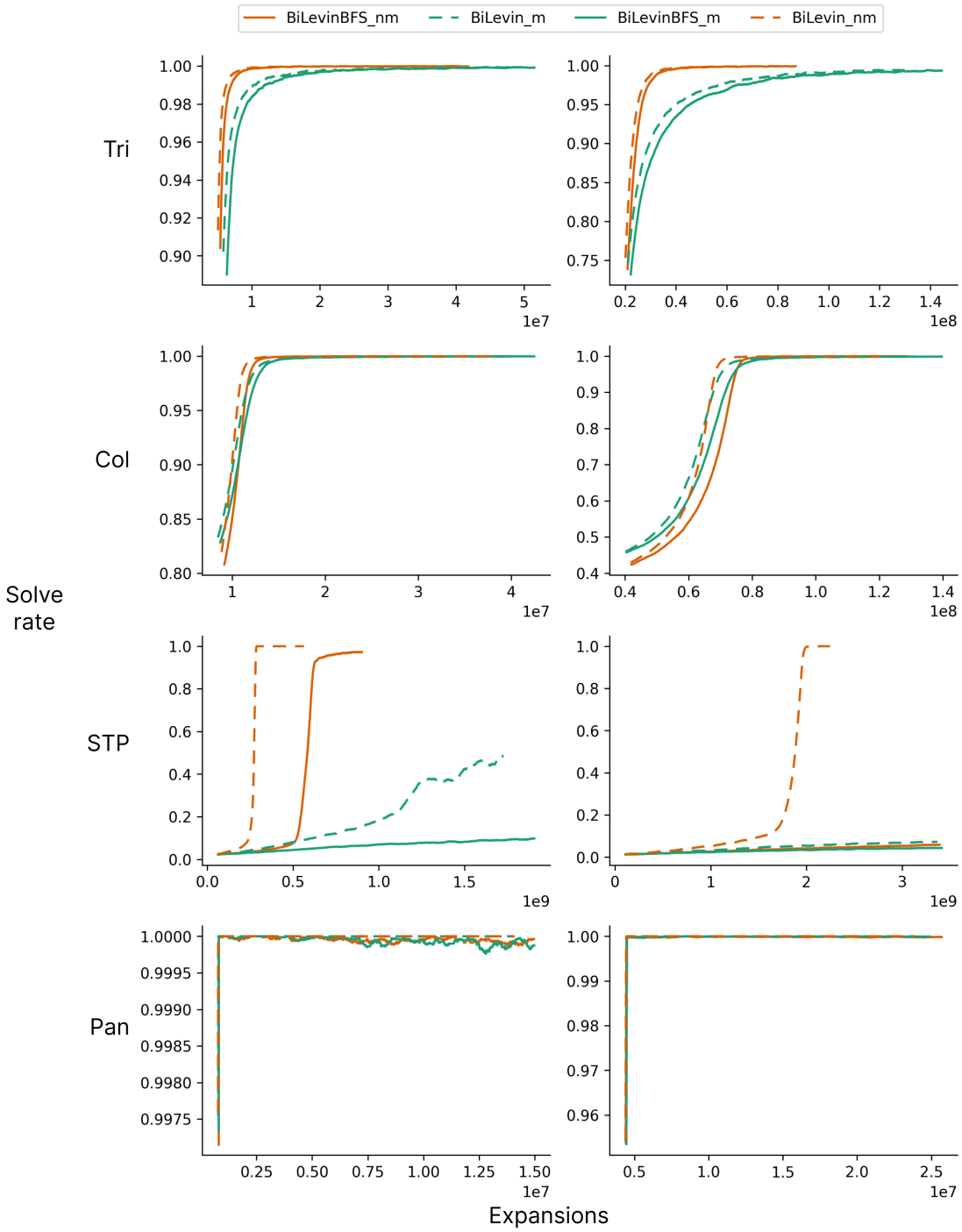

Figure A.4: Comparison of solve rates vs. total expansions for BiLTS across domains during training. Left column shows smaller domain sizes, right column larger sizes. Lines represent mean solve rates over 5 random seeds. All data smoothed using a 500-point fixed-window simple moving average. Expansion budgets used are: 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.

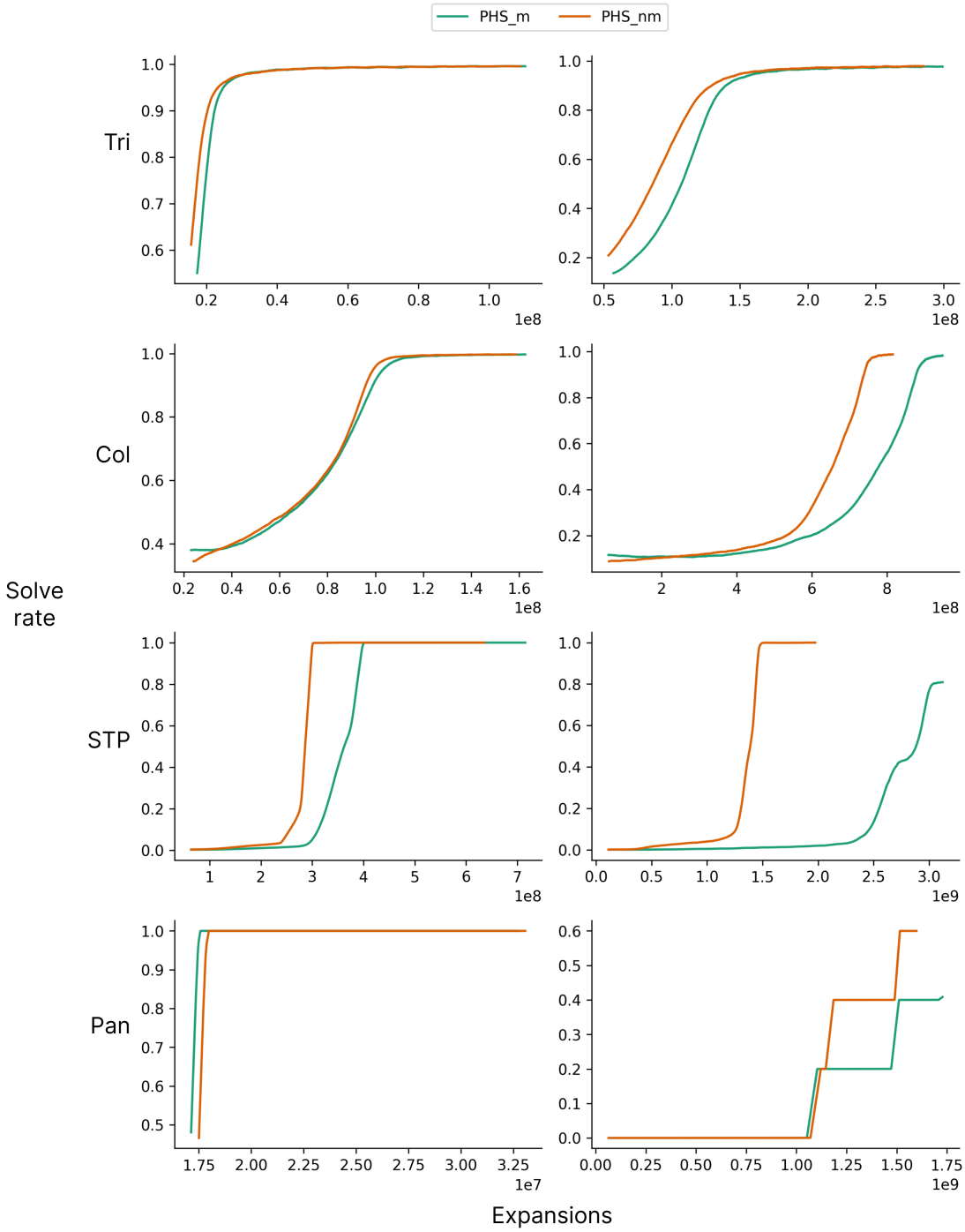

Figure A.5: Comparison of solve rates vs. total expansions for PHS* across domains during training. Left column shows smaller domain sizes, right column larger sizes. Lines represent mean solve rates over 5 random seeds. All data smoothed using a 500-point fixed-window simple moving average. Expansion budgets used are: 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.

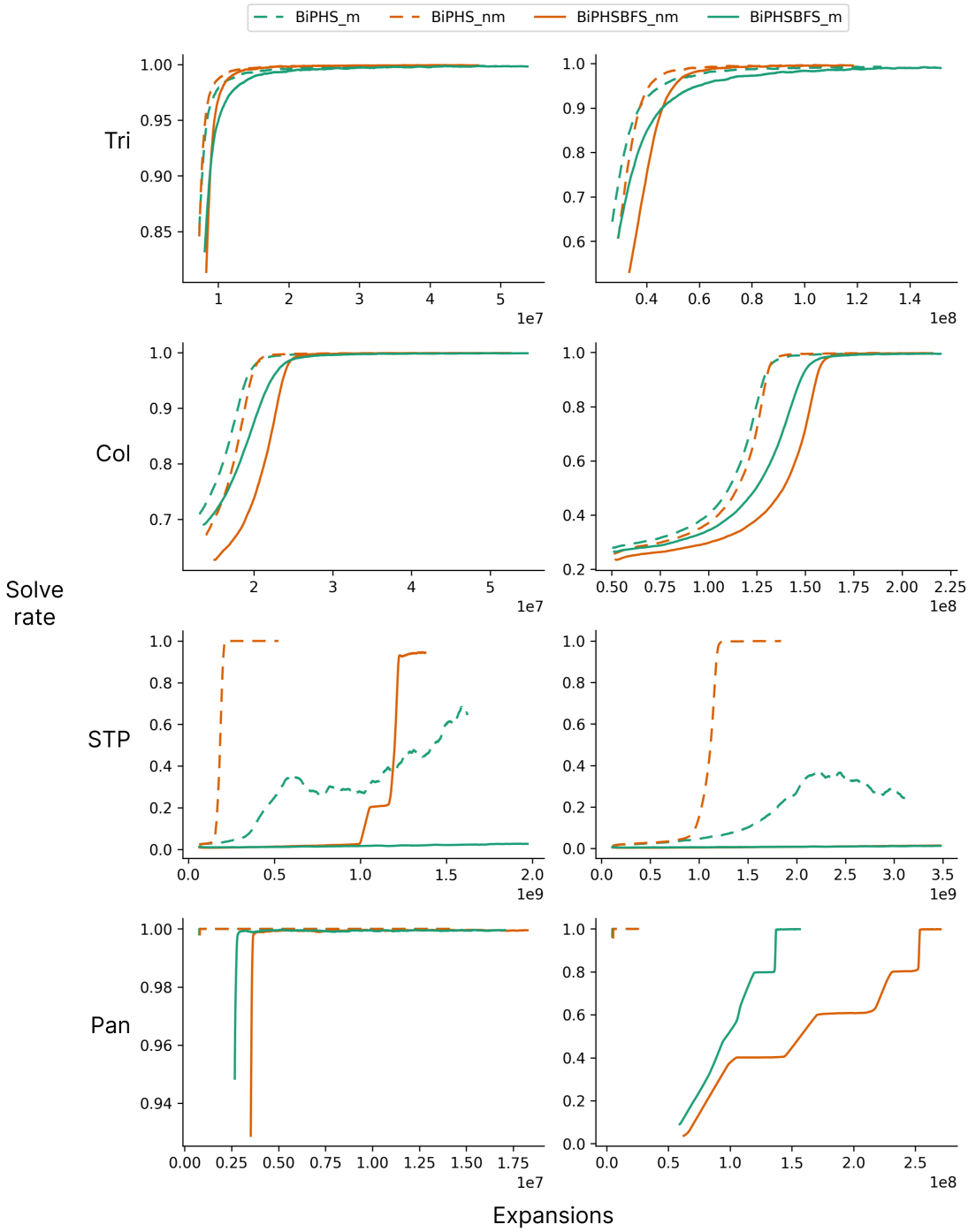

Figure A.6: Comparison of solve rates vs. total expansions for BiPHS* across domains during training. Left column shows smaller domain sizes, right column larger sizes. Lines represent mean solve rates over 5 random seeds. All data smoothed using a 500-point fixed-window simple moving average. Expansion budgets used are: 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.

Appendix B

Comparing Bidirectional Forward and Backward Metrics

Table B.1 shows bidirectional search metrics across domains on commonly solved test problems across all seeds and algorithms (number solved shown in parenthesis under the domain name): (i) fraction of solution path discovered in forward vs. backward direction, (ii) ratio of average forward action probability to total (forward + backward), and (iii) ratio of forward average absolute heuristic error to total (forward + backward). The latter two metrics are first averaged per solution path. Data shows means (top) and standard deviations (parenthesized, bottom) averaged over 5 random seeds. Each seed uses the model with lowest total node expansion on the validation set after any training epoch.

In the Witness domains, the fraction of the solution path found in both directions is generally equally distributed between them, except for BiA* which typically discovers more in the backward search, particularly in the COL domains. Notably, BiA* and BiPHS* solve significantly more problems in these domains using fewer expansions. This suggests the backward search problem may be more challenging for learning pure heuristic-based algorithms in these domains. Heuristic and policy accuracy's are generally comparable.

In the STP and PAN domains, a high proportion of the solution is found in the forward direction across all algorithms. This is somewhat expected, as the backward learning problem is more difficult, since (i) the policy/heuristic values of a state also depend on the changing start state, and (ii) backward models in these domains are larger to accommodate the two inputs (current state and start state), potentially requiring more data to learn equally effective policies or heuristics compared to the forward direction. While heuristic errors are equally accurate in these domains, the policy is noticeably more accurate in the forward direction. This indicates that learning a backward policy is particularly challenging in these domains.

		BiA*	BiLTS	BiPHS*
TRI 4 (998.0)	F-B Sol. Len.	0.456 (0.119)	0.494 (0.054)	0.493 (0.056)
	F-B Action Prob.		0.498 (0.010)	0.497 (0.010)
	F-B H Error	0.500 (0.006)		0.501 (0.008)
TRI 5 (975.0)	F-B Sol. Len.	0.489 (0.134)	0.498 (0.064)	0.495 (0.064)
	F-B Action Prob.		0.499 (0.009)	0.498 (0.009)
	F-B H Error	0.502 (0.005)		0.502 (0.007)
COL 4 (996.0)	F-B Sol. Len.	0.423 (0.121)	0.500 (0.041)	0.500 (0.044)
	F-B Action Prob.		0.496 (0.012)	0.496 (0.011)
	F-B H Error	0.504 (0.010)		0.501 (0.012)
COL 5 (988.0)	F-B Sol. Len.	0.373 (0.132)	0.498 (0.051)	0.498 (0.055)
	F-B Action Prob.		0.496 (0.010)	0.496 (0.009)
	F-B H Error	0.503 (0.007)		0.504 (0.011)
STP 4 (1000.0)	F-B Sol. Len.	0.920 (0.020)	0.805 (0.032)	0.775 (0.055)
	F-B Action Prob.		0.584 (0.006)	0.576 (0.007)
	F-B H Error	0.499 (0.006)		0.505 (0.011)
STP 5 (1000.0)	F-B Sol. Len.	0.974 (0.004)	0.889 (0.020)	0.857 (0.036)
	F-B Action Prob.		0.594 (0.004)	0.576 (0.004)
	F-B H Error	0.493 (0.006)		0.502 (0.007)
PAN 10 (1000.0)	F-B Sol. Len.	0.844 (0.041)	0.750 (0.045)	0.722 (0.054)
	F-B Action Prob.		0.677 (0.031)	0.667 (0.023)
	F-B H Error	0.499 (0.006)		0.498 (0.007)
PAN 12 (1000.0)	F-B Sol. Len.	0.895 (0.025)	0.808 (0.027)	0.688 (0.040)
	F-B Action Prob.		0.713 (0.026)	0.668 (0.017)
	F-B H Error	0.499 (0.007)		0.501 (0.007)

Table B.1: Comparison of bidirectional search metrics across domains on commonly solved test problems across all seeds and algorithms (number solved shown in parenthesis under domain name): (i) fraction of solution path discovered in forward vs. backward direction, (ii) ratio of average forward action probability to total (forward + backward) probability, and (iii) ratio of forward average absolute heuristic error to total (forward + backward). The latter two metrics are first averaged per solution path. Data shows means (top) and standard deviations (parenthesized, bottom) averaged over 5 random seeds. Each seed uses the model with lowest total node expansion on the validation set after any training epoch.

Appendix C

Neural Net Architecture Details

All fully-connected networks have appropriately sized input layers, that depend on the size of the inputs fed to them, and have outputs layers of either size 1 (heuristic modules), $|\mathcal{A}|$ (policy modules), or the size of the fully-connected layer they feed into (PAN feature module). A SoftMax function is applied to the outputs of the policy modules to obtain a probability distribution.

C.1 Forward Networks

The forward networks of STP, TRI, and COL domains use the exact architecture described in Orseau & Lelis (2021):

- Feature module: 2 convolutional layers, each with 32 kernels of size 2×2 and no padding.
- Policy module: 2 fully-connected hidden layers, each of size 128 with ReLU activations (Agarap 2018).
- Heuristic module: Same architecture as the policy module.

For the forward PAN network, we use a feature module with a single fully-connected hidden layer of size 256 with ReLU activations, and use policy and heuristic modules with a single fully-connected hidden layers of size 64.

C.2 Backward Networks

The backward feature modules of all domains are the same as the forward feature modules.

- For STP, the backward policy and heuristic modules each use three fully-connected layers of sizes 256, 198, and 128, all with ReLU activations.

- For PAN, the backward policy and heuristic modules each use three fully-connected layers of sizes 128, 96, and 64, all with ReLU activations.
- For TRI and COL, the forward and backward networks are completely identical.

We implement all networks using PyTorch (Paszke et al. 2019).

Appendix D

State Representations

In this section we describe the state representations which are passed to the feature modules.

For STP and PAN, each state is represented as an array of one-hot encodings:

- In STP of width n, each state is an $n \times n$ array. For grid cell (i, j) with value k, the array entry (i, j) is a length-n^2 array with a 1 at the kth position and 0s elsewhere.
- For PAN with stack size n, each state is an length-n array, where each entry is a one-hot encoding of one of the n pancakes.

In Witness-type domains of width n, each state is represented as a $(n + 1) \times (n + 1) \times (k + 5)$ array:

- For TRI, $k = 3$, with these channels representing positions of 1, 2, or 3 triangles. For example, the first channel represents the positions of the cells containing 1 triangle, so, if cell (i, j) contains 1 triangle, then array entry $(1, 1, 1)$ will have a 1, otherwise it will have a 0.
- For COL, $k = 4$, with these channels representing locations of the four possible colors.

The remaining 5 channels encode the locations on the grid of:

- Vertical line segments.
- Horizontal line segments.
- End point of the line.
- Required ending location.
- Starting location (which does not change from state to state, but is included for consistency with Orseau & Lelis (2021)).

These channels require $(n+1) \times (n+1)$ entries because they encode positions on the corners of the grid, while the previous channels encode cell locations and do not make use of the the $(n+1)$st row or column.

Appendix E

Additional Results for Preliminary Experiments

These plots provide further justification for our algorithm variant selections in appendix A, which we use for the unidirectional and bidirectional comparison (appendix B). They provide a more detailed perspective on the progression of solve rates and average expansions per problem throughout training batches. By displaying median, minimum, and maximum values, these plots illustrate the algorithms' resilience to various random factors (detailed in section 3.6). The observations drawn from these visualizations corroborate our findings presented in appendix A, reinforcing the validity of our choices.

Figures E.1 to E.6 show the solve rates versus batch for all variants of one of the algorithms we consider, on all training batches, across all domains. The data is smoothed using a 500-point fixed-window simple moving average. The lines show the median over 5 random seeds, while shaded areas show the min and max.

Figures E.7 to E.12 show the expansions per problem versus batch for all variants of one of the algorithms we consider, on all training batches, across all domains. The data is smoothed using a 500-point fixed-window simple moving average. The lines show the median over 5 random seeds, while shaded areas show the min and max.

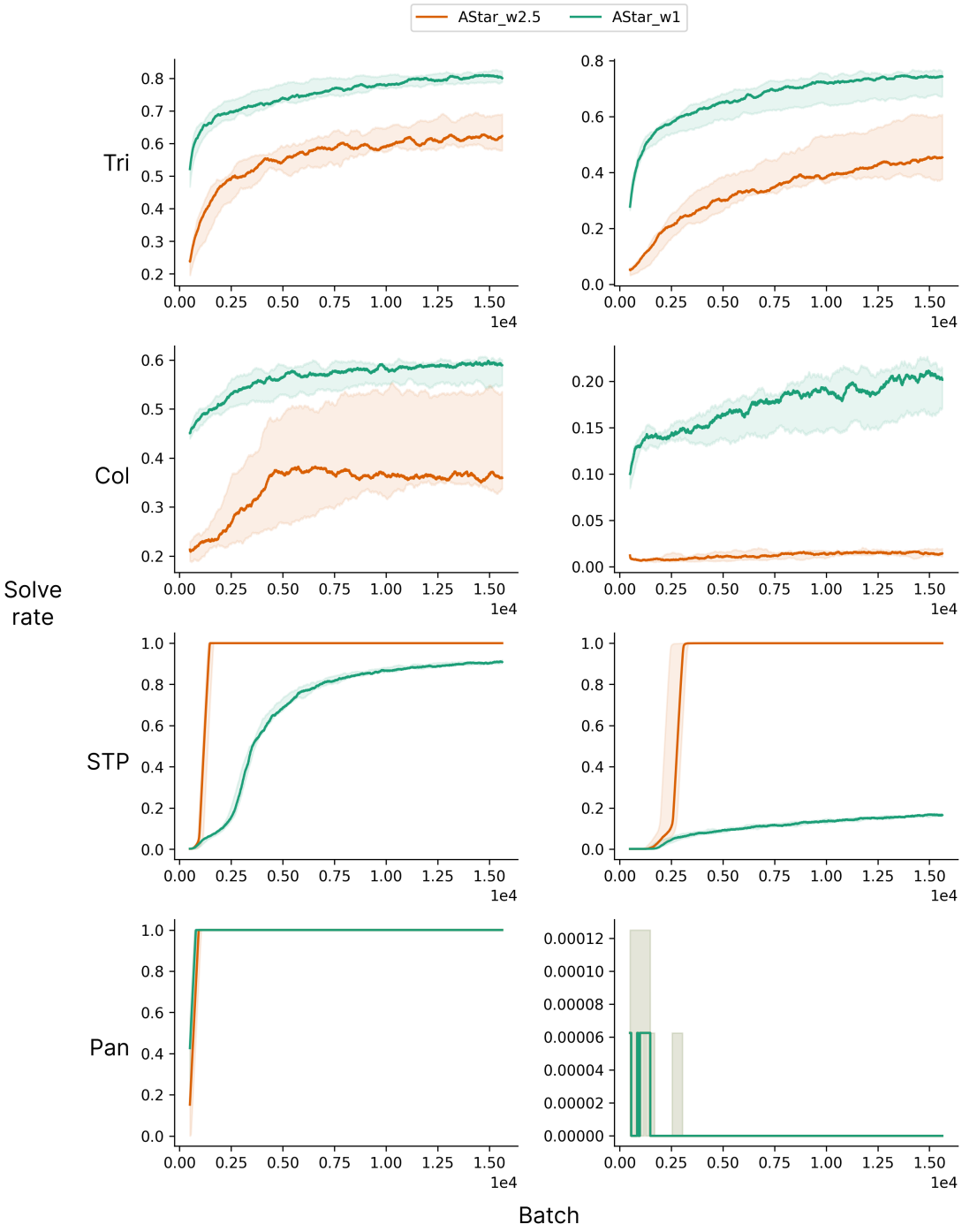

Figure E.1: Comparison of solve rate versus batch for A* algorithm variants on all training batches, across all domains. The data is smoothed using a 500-point fixed-window simple moving average. The lines show the median over 5 random seeds, while shaded areas show the min and max. Expansion budgets used are: 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.

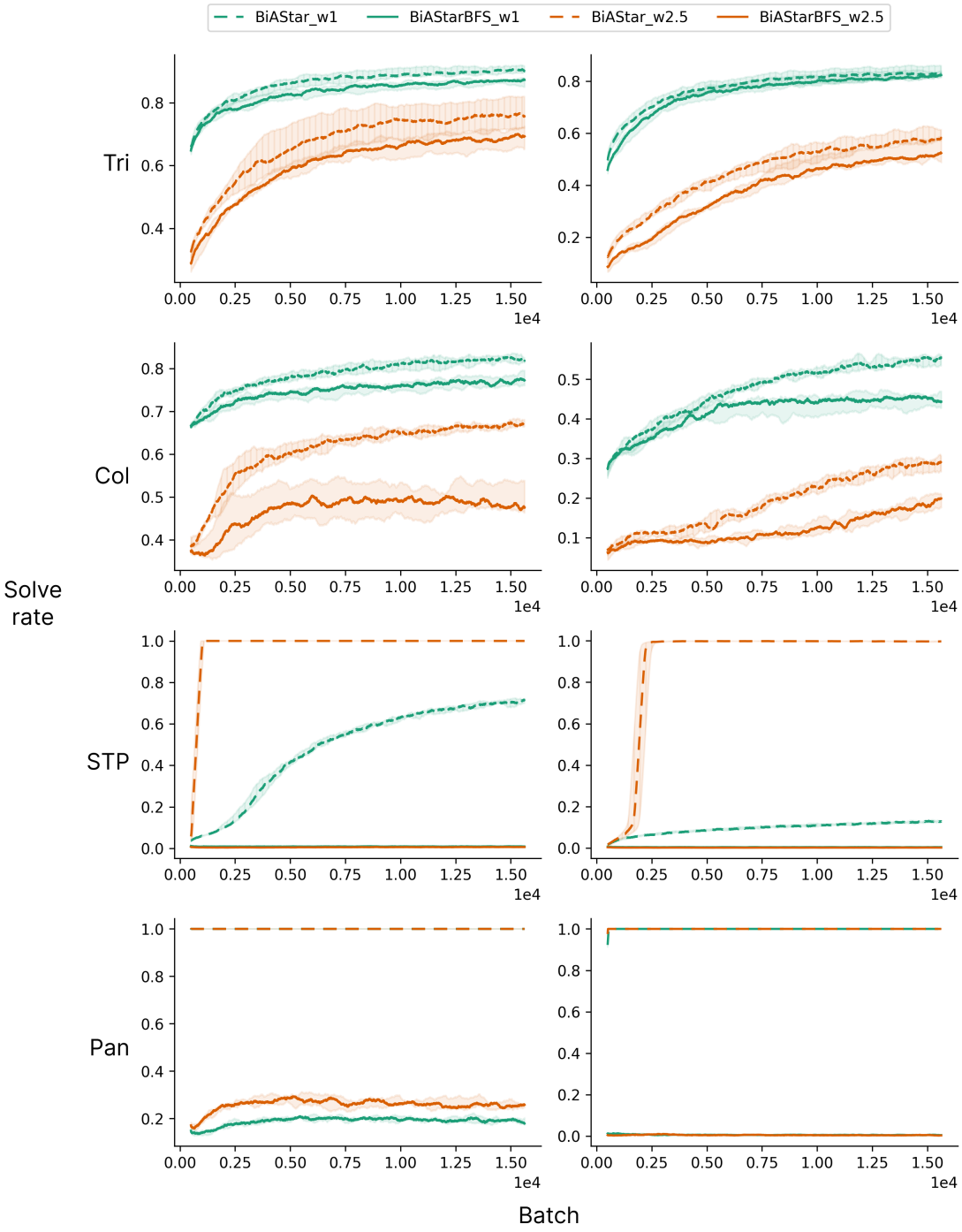

Figure E.2: Comparison of solve rate versus batch for BiA* algorithm variants on all training batches, across all domains. The data is smoothed using a 500-point fixed-window simple moving average. The lines show the median over 5 random seeds, while shaded areas show the min and max. Expansion budgets used are: 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.

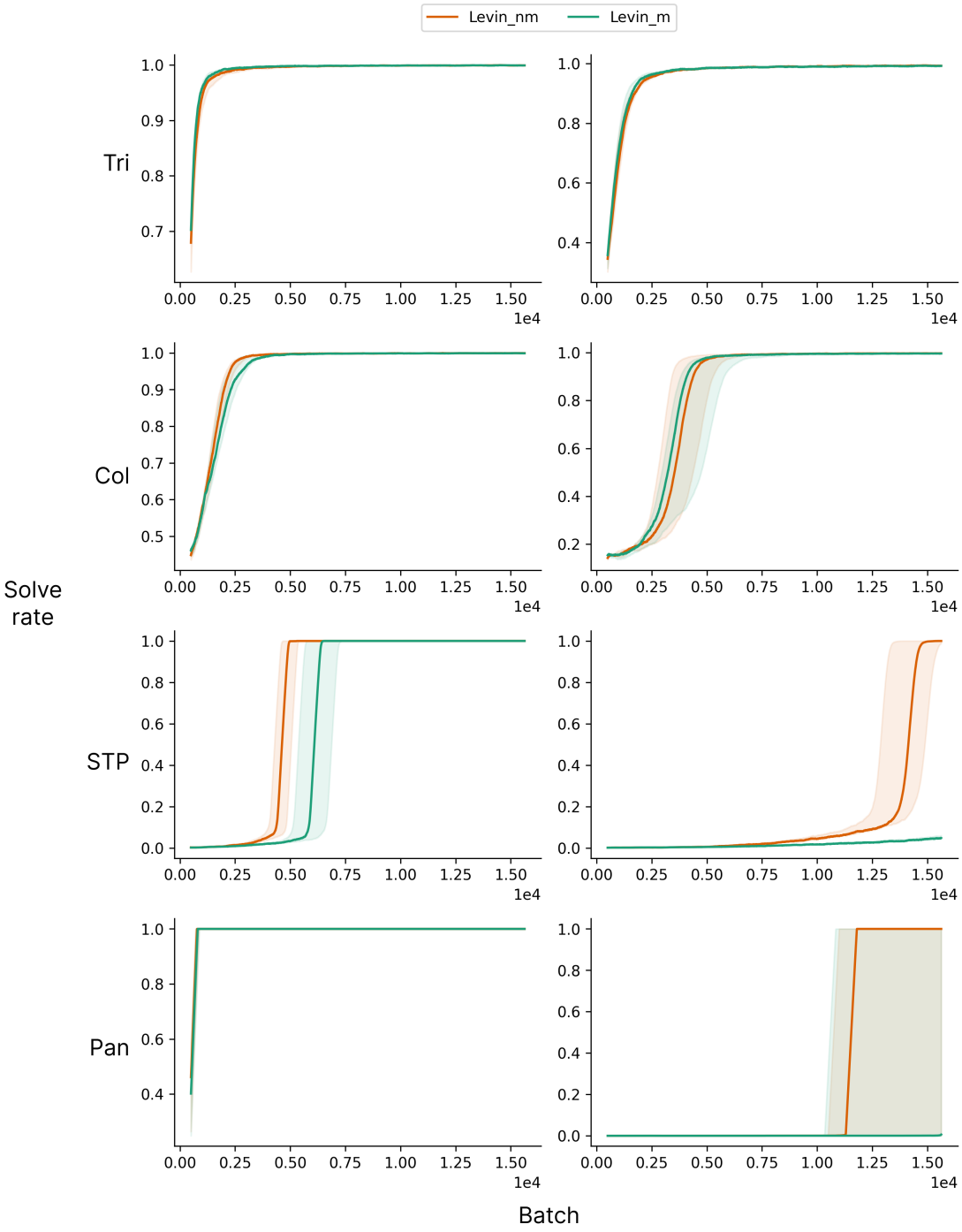

Figure E.3: Comparison of solve rate versus batch for LTS algorithm variants on all training batches, across all domains. The data is smoothed using a 500-point fixed-window simple moving average. The lines show the median over 5 random seeds, while shaded areas show the min and max. Expansion budgets used are: 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.

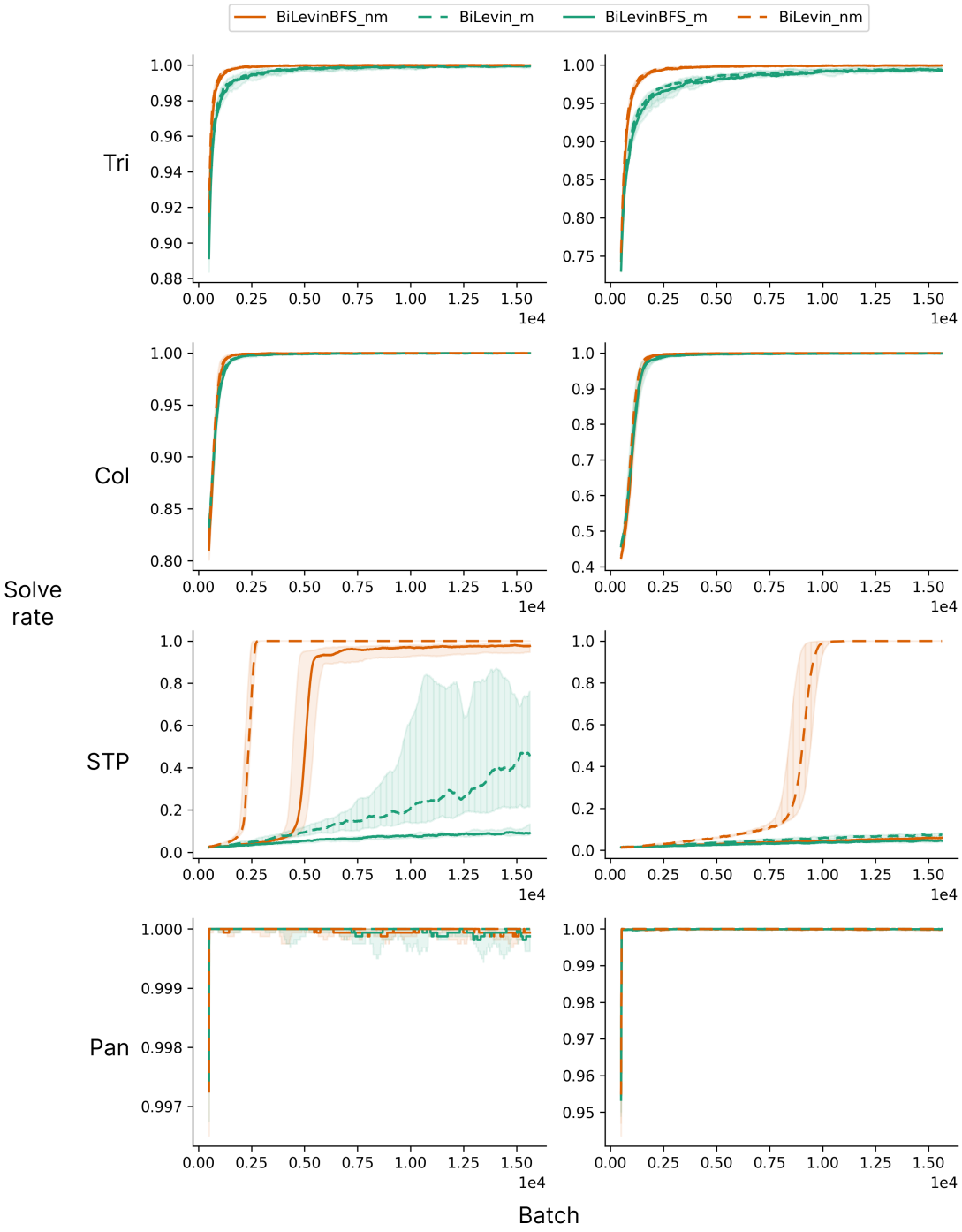

Figure E.4: Comparison of solve rate versus batch for BiLTS algorithm variants on all training batches, across all domains. The data is smoothed using a 500-point fixed-window simple moving average. The lines show the median over 5 random seeds, while shaded areas show the min and max. Expansion budgets used are: 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.

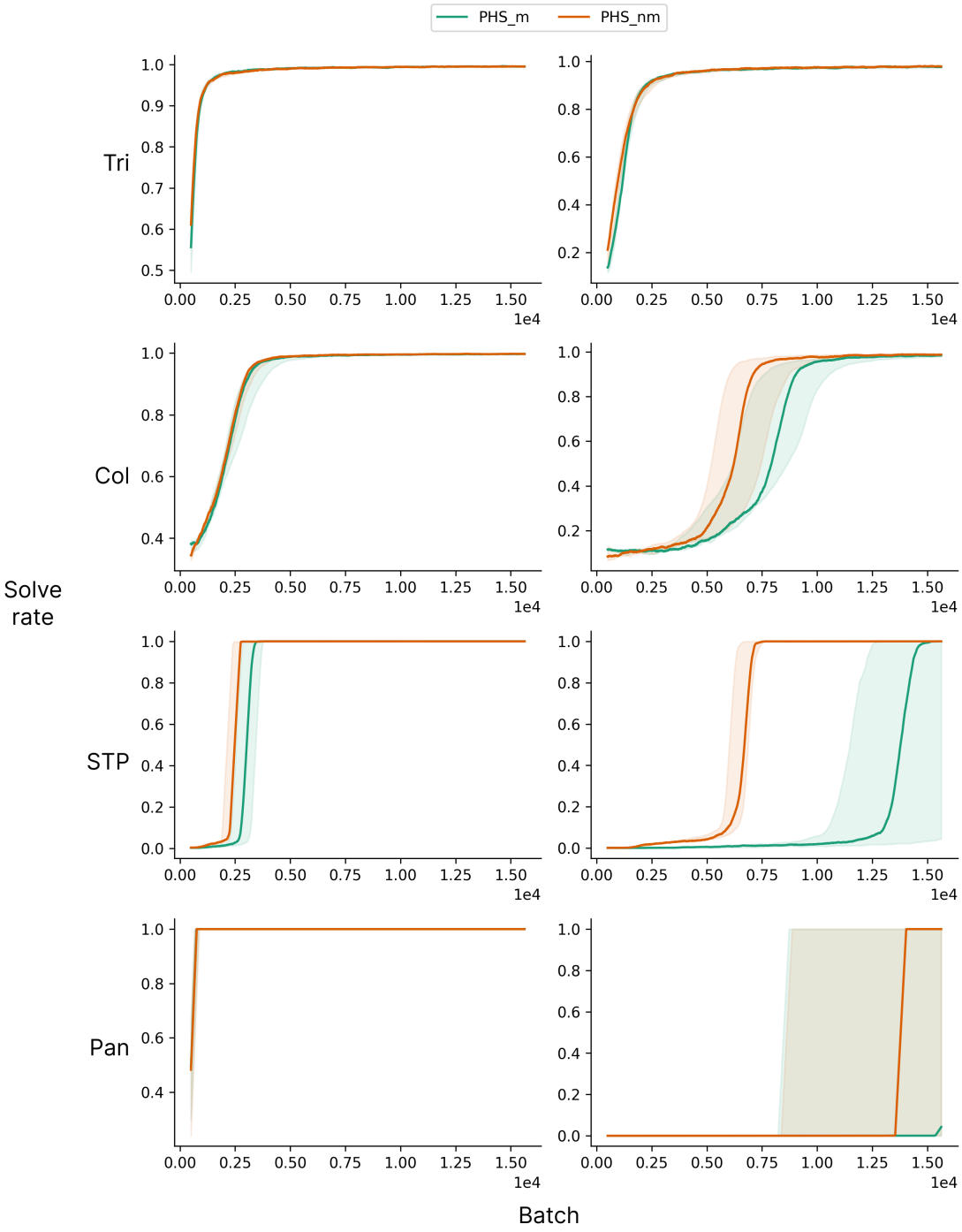

Figure E.5: Comparison of solve rate versus batch for PHS* algorithm variants on all training batches, across all domains. The data is smoothed using a 500-point fixed-window simple moving average. The lines show the median over 5 random seeds, while shaded areas show the min and max. Expansion budgets used are: 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.

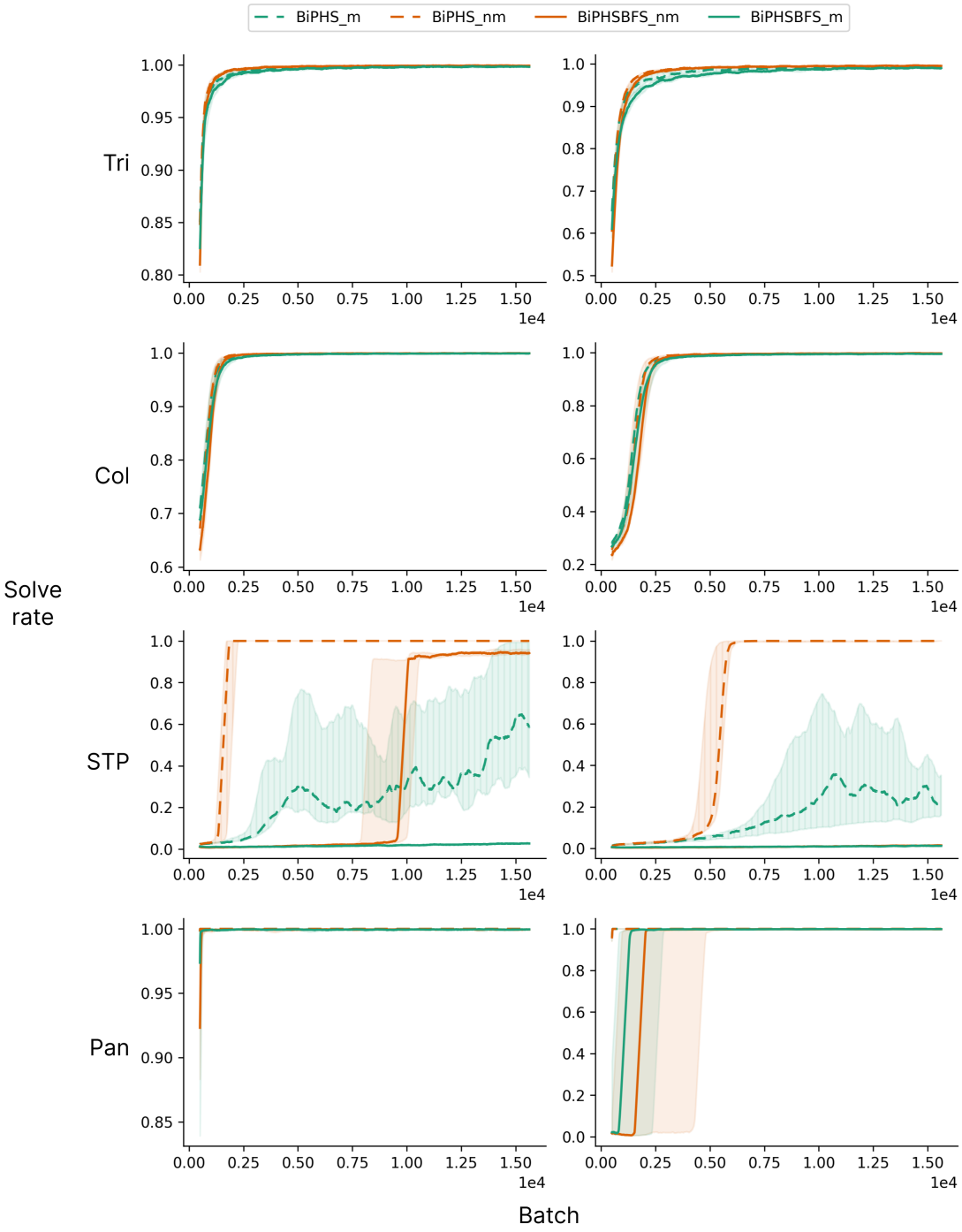

Figure E.6: Comparison of solve rate versus batch for BiPHS* algorithm variants on all training batches, across all domains. The data is smoothed using a 500-point fixed-window simple moving average. The lines show the median over 5 random seeds, while shaded areas show the min and max. Expansion budgets used are: 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.

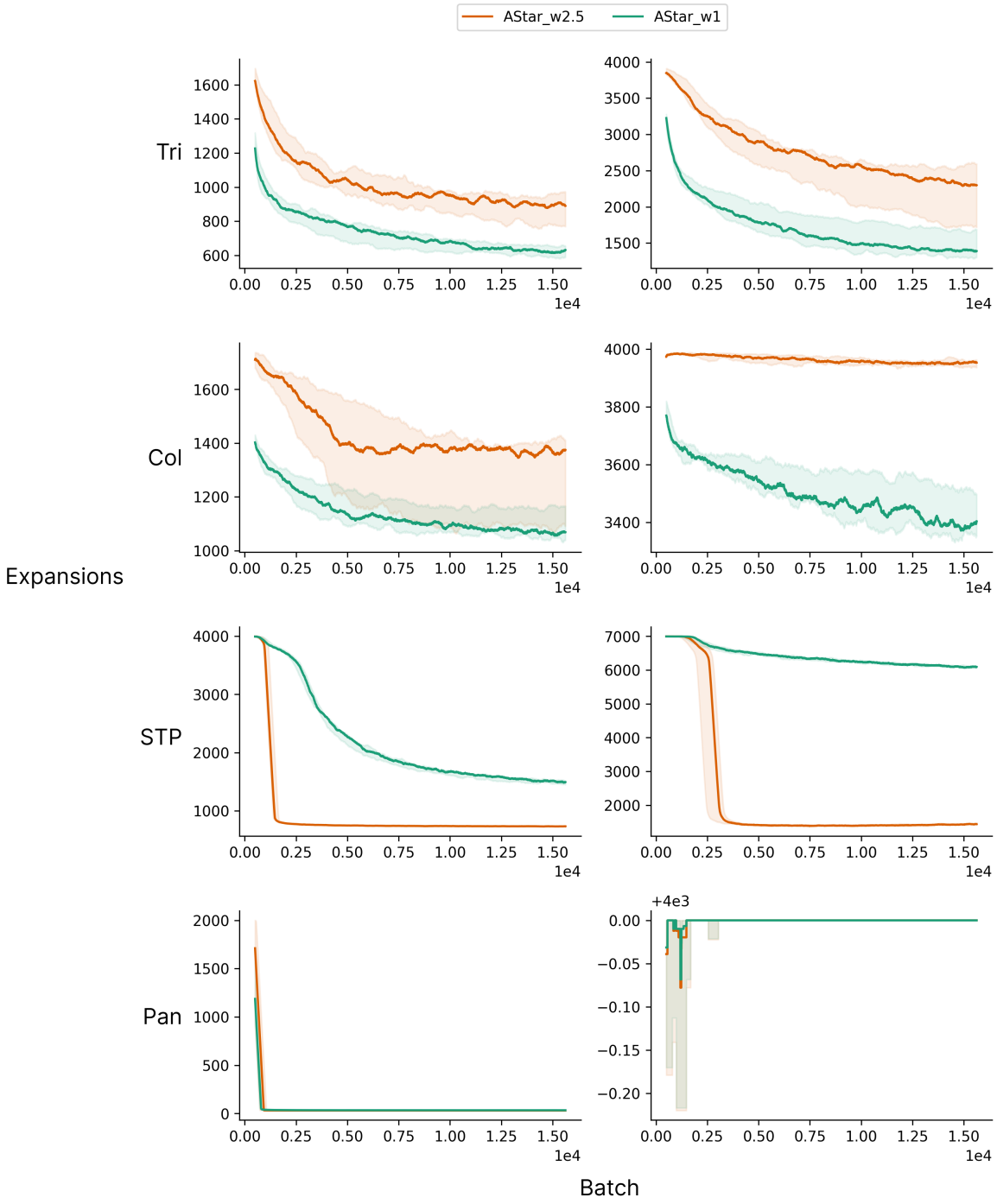

Figure E.7: Comparison of expansions per problem versus batch for A* algorithm variants on all training batches, across all domains. The data is smoothed using a 500-point fixed-window simple moving average. The lines show the median over 5 random seeds, while shaded areas show the min and max. Expansion budgets used are: 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.

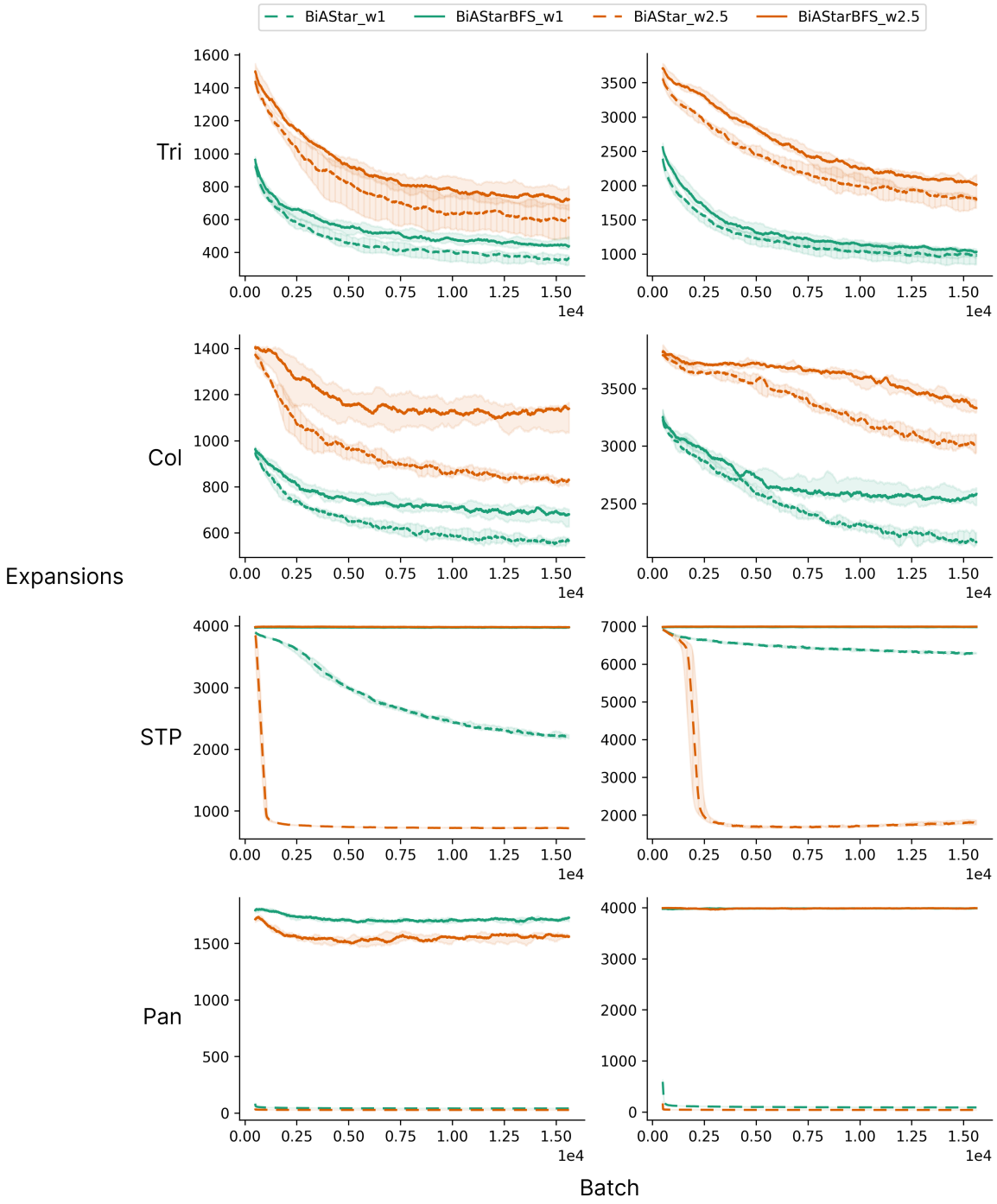

Figure E.8: Comparison of expansions per problem versus batch for BiA* algorithm variants on all training batches, across all domains. The data is smoothed using a 500-point fixed-window simple moving average. The lines show the median over 5 random seeds, while shaded areas show the min and max. Expansion budgets used are: 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.

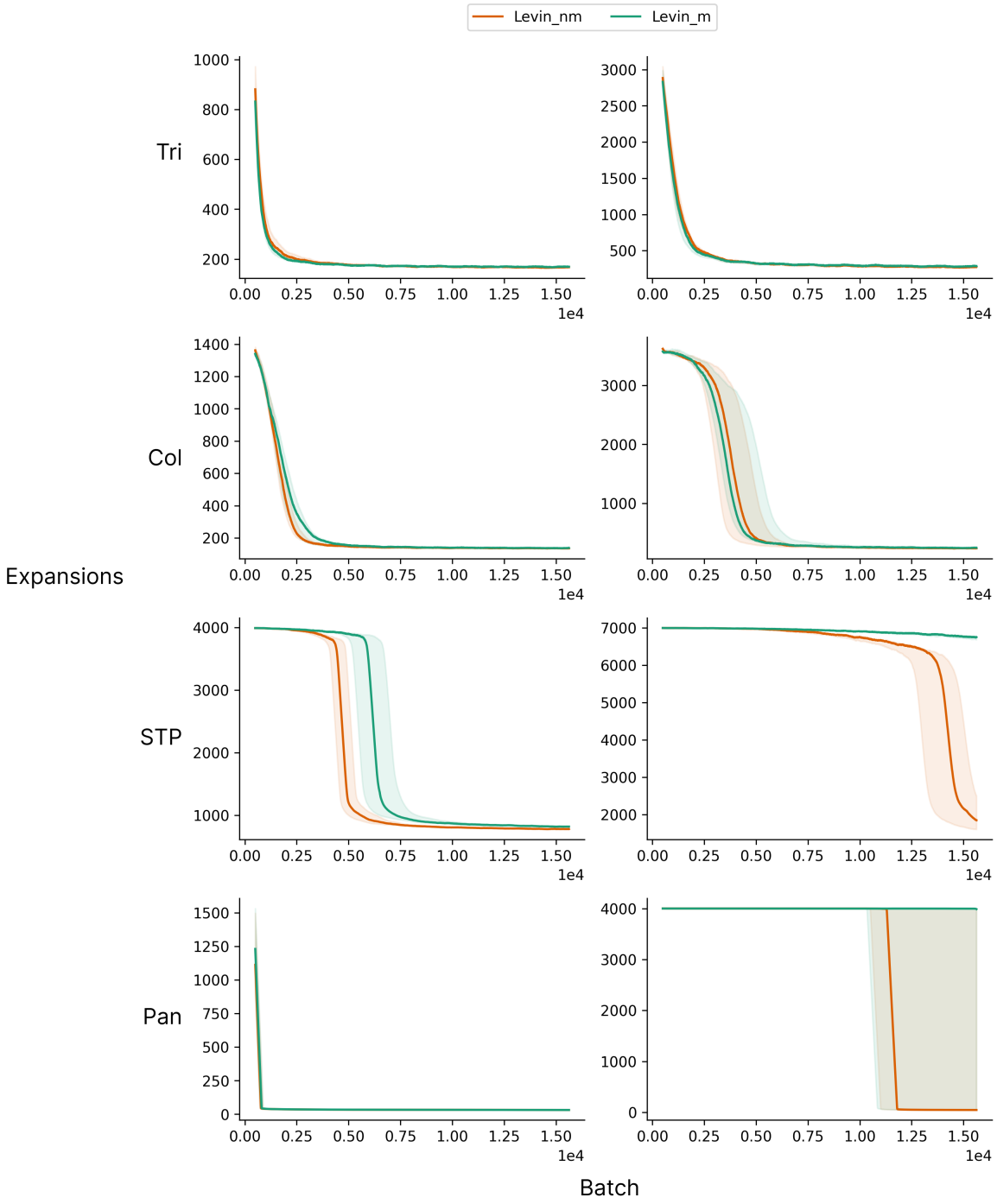

Figure E.9: Comparison of expansions per problem versus batch for LTS algorithm variants on all training batches, across all domains. The data is smoothed using a 500-point fixed-window simple moving average. The lines show the median over 5 random seeds, while shaded areas show the min and max. Expansion budgets used are: 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.

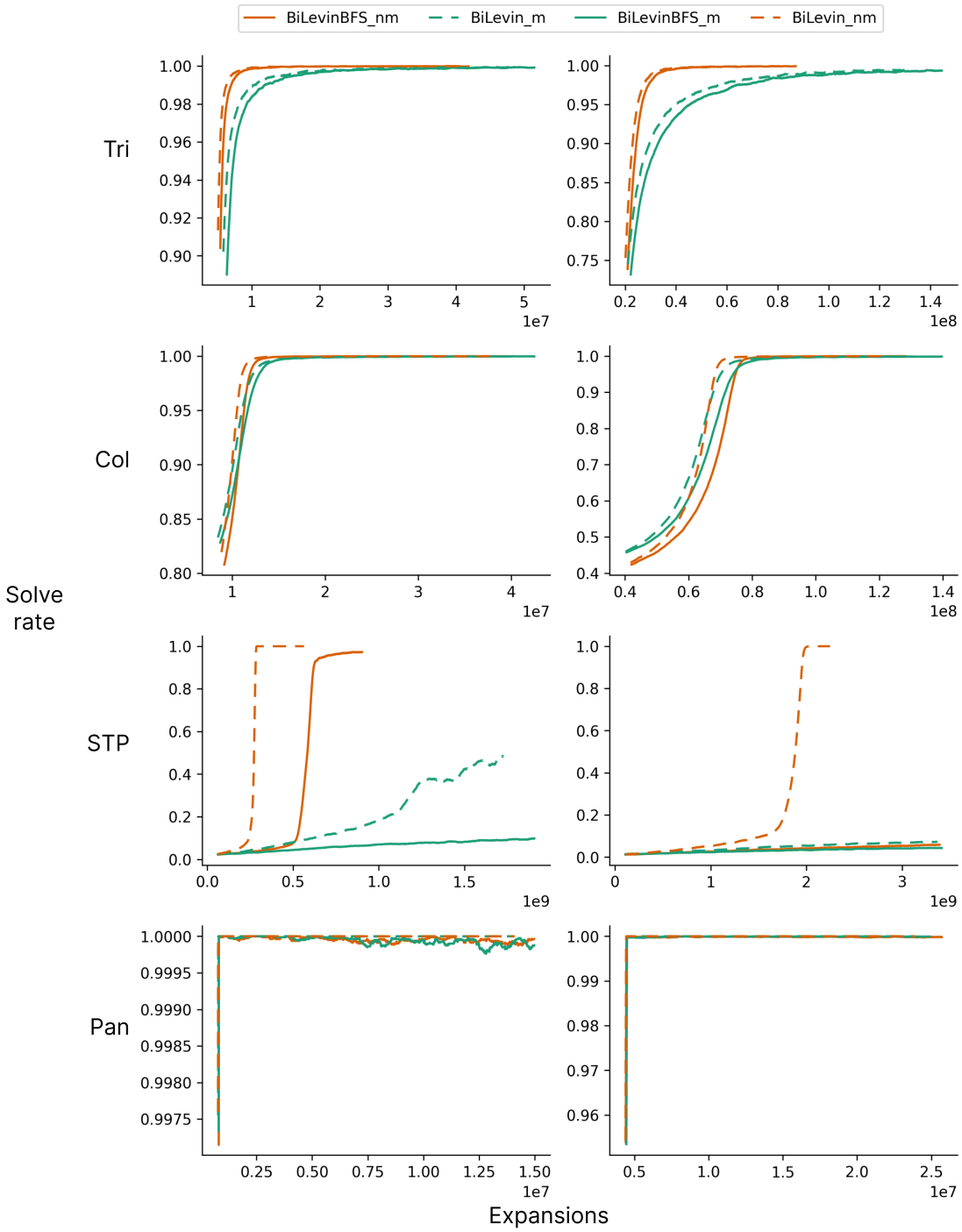

Figure E.10: Comparison of expansions per problem versus batch for BiLTS algorithm variants on all training batches, across all domains. The data is smoothed using a 500-point fixed-window simple moving average. The lines show the median over 5 random seeds, while shaded areas show the min and max. Expansion budgets used are: 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.

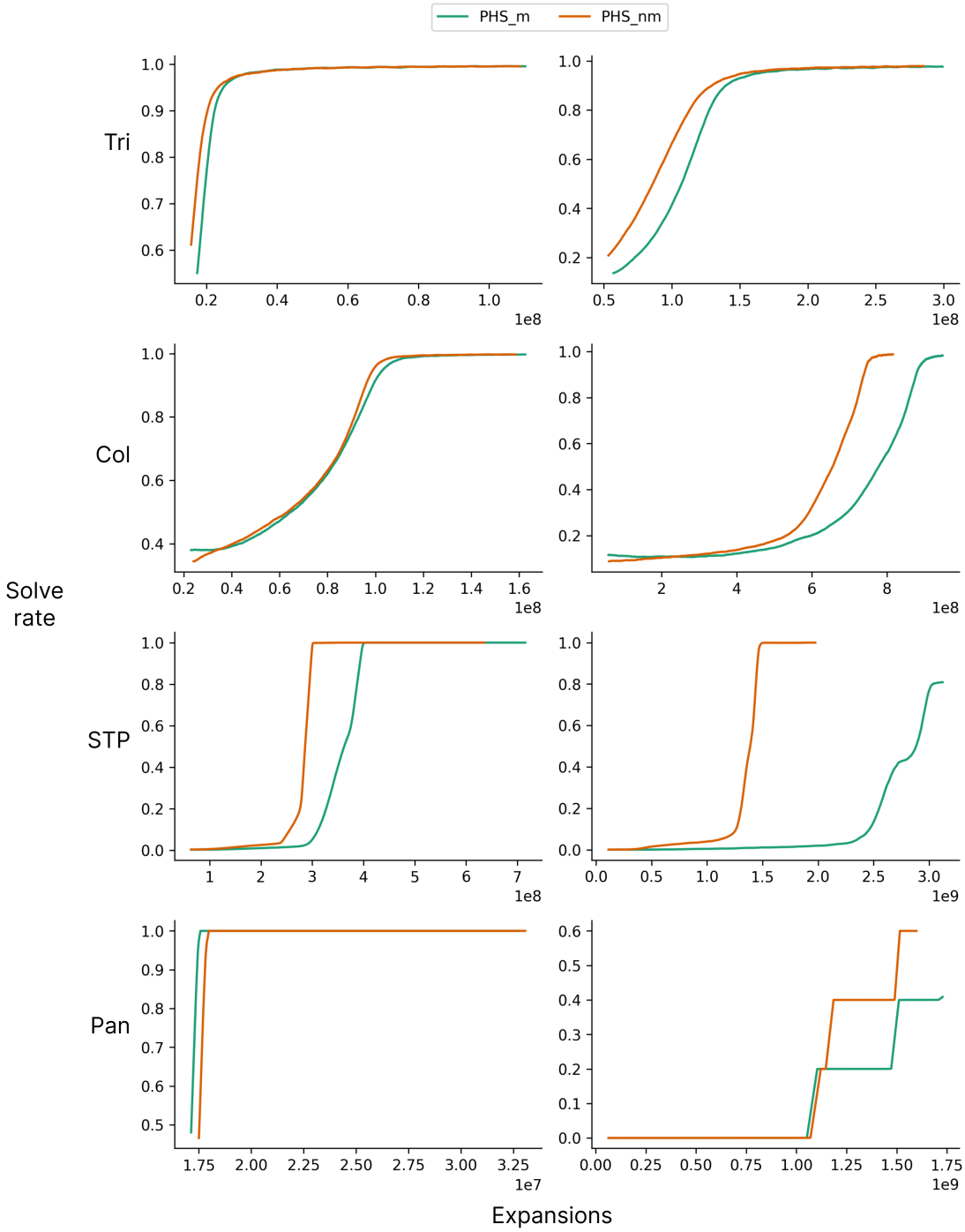

Figure E.11: Comparison of expansions per problem versus batch for PHS* algorithm variants on all training batches, across all domains. The data is smoothed using a 500-point fixed-window simple moving average. The lines show the median over 5 random seeds, while shaded areas show the min and max. Expansion budgets used are: 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.

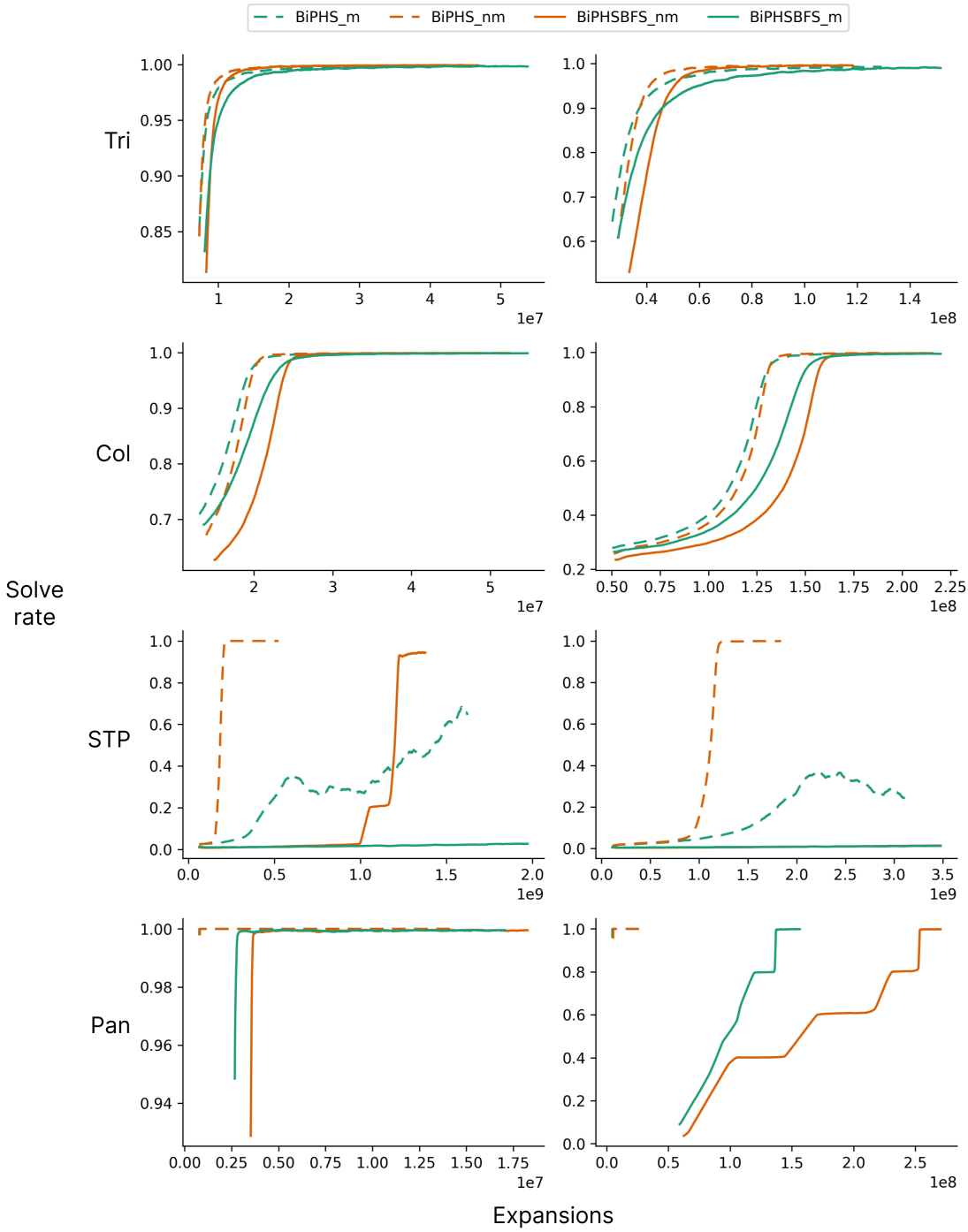

Figure E.12: Comparison of expansions per problem versus batch for BiPHS* algorithm variants on all training batches, across all domains. The data is smoothed using a 500-point fixed-window simple moving average. The lines show the median over 5 random seeds, while shaded areas show the min and max. Expansion budgets used are: 2000 for TRI 4, COL 4, and PAN 10. 4000 for STP 4, TRI 5, COL 5, and PAN 12. 7000 for STP 5.