

Chair of Machine Learning and Reasoning (Computer Science 6)

RWTH AACHEN UNIVERSITY

This thesis was submitted to the Chair of Machine Learning and Reasoning.

Beyond Greedy Execution: Neural-Guided Search in Classical Planning

Bachelor Thesis

Presented by

Ibrahim Eisawy
449715

Supervised by Dr. Simon Ståhlberg, Ph.D.

1st Examiner Univ.-Prof. Ph. D. Geffner, Hector

2nd Examiner Univ.-Prof. Dr. rer. nat. Morris, Christopher

Aachen, September 22, 2026

Abstract

Greedy execution of learned value functions and policies is the most direct approach for using learned models in classical planning. However, the learned models' guidance is inherently imperfect: it may lack the expressiveness to distinguish states with different true distances to the goal and may fail to generalize to instances larger than those seen during training. Although greedy execution is efficient, it offers no mechanism to detect or recover from such errors.

This thesis investigates how search can compensate for these failures. We benchmark A*, Weighted A*, Greedy Best-First Search, Beam Search, Q*, Weighted Q*, and AlphaZero-style Monte Carlo Tree Search across six planning domains, comparing each with greedy execution of its guiding model. We find that search algorithms do not behave uniformly: A* and Q* often solve fewer instances than their greedy counterparts because combining learned estimates with path cost makes the search sensitive to their relative numerical scale, potentially causing either the heuristic or the path cost to dominate. Beam Search and AlphaZero-style search avoid this dependence and achieve the strongest overall results.

We then choose the best-performing algorithm overall, AlphaZero, and investigate how to further improve it by modifying its policy prior. We first flatten the prior uniformly to balance preserving useful policy information with keeping low-probability actions accessible.

Finally, we use Bellman residuals from a distributional value function to redistribute the prior in a targeted way toward actions where the model is least self-consistent. The residual ordering is informative in three of five domains and can reduce average node expansions by up to about 60% while often producing plans up to 50% shorter. Overall, search can compensate for imperfect learned guidance, but the benefit depends on both the domain and how that guidance is integrated into the search.

Contents

1 Introduction **1**

1.1 Motivation . . . 1
1.2 Contribution . . . 1
1.3 Research Questions . . . 3
1.4 Thesis Structure . . . 3

2 Background & Related Works **4**

2.1 Classical Planning . . . 4
2.2 Learned Value Functions and Policies for Planning . . . 5
2.3 Learned Heuristics for Search . . . 6
2.4 Monte Carlo Tree Search in Classical Planning . . . 7
2.5 Relational Graph Neural Networks . . . 8
2.6 Distributional Action-Value Functions . . . 9
2.7 Limitation of existing approaches . . . 10

3 Learning-Guided Search Algorithms **11**

3.1 Greedy Baselines . . . 11
3.2 Value-based Search Algorithms . . . 12
3.2.1 A* and Weighted A* . . . 12
3.2.2 Greedy Best-First Search . . . 13
3.2.3 Beam Search . . . 13
3.3 Q-Based Search . . . 14
3.3.1 Q* and Weighted Q* . . . 14
3.4 Tree-based Approach . . . 14
3.4.1 AlphaZero-style MCTS . . . 15

4 Experimental Setup **16**

4.1 Domain Selection . . . 16
4.2 Instance Generation . . . 17
4.3 Data Splits and Test Distributions . . . 17
4.4 Evaluation Metrics . . . 18
4.5 Model Training and Computational Budgets . . . 19

5 Neural-Guided Search Benchmark Results **20**

5.1 Overview . . . 20
5.2 Greedy Execution . . . 20
5.3 Value-Based Search . . . 21
5.4 Q-Based Search . . . 24
5.5 Tree-Based Search . . . 25
5.6 Comparison and Selection . . . 26

6 Improving AlphaZero-style MCTS 29
6.1 Motivation . . . 29
6.2 Prior Flattening . . . 30
6.3 Bellman Residual and Error Scaling . . . 31
6.3.1 The Per-Action Residual . . . 31
6.3.2 Scaling and Aggregation . . . 32
6.4 Injection into the Tree Policy . . . 32
6.4.1 Flattening the Prior . . . 33
6.4.2 Additive and Multiplicative Priors . . . 33
6.4.3 Parameters . . . 34
6.5 Controls: shuffle, random . . . 34

7 Evaluation of AlphaZero-style MCTS Modifications 36
7.1 Overview . . . 36
7.2 Flattening the Prior . . . 36
7.2.1 Effect of Prior Flattening . . . 36
7.3 The Bellman Inconsistency Signal . . . 38
7.3.1 Overall Signal Performance . . . 38
7.4 Does the Bellman Residual e_a Identify Useful Actions? . . . 39
7.4.1 When the Residual Ordering Is Unhelpful . . . 39
7.4.2 What the Residual Fails to Measure . . . 41
7.4.3 Effect of β on Search Performance . . . 41
7.4.4 Multiplicative . . . 42
7.4.5 Summary of Out-of-Distribution Bounded Bellman Results . . . 43
7.5 Out-of-Distribution Unbounded Test Set . . . 43
7.5.1 Bellman Signal . . . 44
7.5.2 The Cost of the Residual Computation . . . 45

8 Limitations & Future Work 47
8.1 Computational Limitations of the Bellman Signal . . . 47
8.2 Limitations and Extensions of the Bellman Signal . . . 47
8.3 Experimental Design . . . 48
8.4 Extending to Other Search Algorithms . . . 49

9 Conclusion **50**

A Appendix **51**

List of Acronyms **57**

List of Symbols **58**

List of Figures **60**

List of Tables **61**

List of References **63**

1 Introduction

1.1 Motivation

The planning problem is a fundamental area of artificial intelligence concerned with computing a sequence of actions, a plan, that transforms an initial state into a desired goal state. Basic components of a planning task are states of the current world, applicable actions for each state, and goal states that can be reached from the initial state through a plan. There exist several classes of planning problems. The simplest among them are the discrete, finite, fully observable, and deterministic planning problems, commonly known as classical planning [9].

Despite the simple formulation of classical planning, deciding whether a plan exists even for tasks using a basic action representation such as propositional STRIPS is PSPACE-complete. The main difficulty comes from the rapid growth of the state space. This happens because a planning state is defined by a set of variables whose possible value combinations can produce exponentially many states as the problem grows. Searching this space can therefore become computationally expensive, with the time required to find a solution increasing exponentially with problem size [2].

A common approach to classical planning is to model it as a path-finding problem in a graph. Graph nodes can correspond to planning states, while edges may correspond to actions that transform one state into another. Solving the planning problem then amounts to finding a path, or plan, from the initial state to a goal state.

One such approach is the greedy policy, where a learned value function estimates the cost from each state to the goal. At each step, the policy selects the successor with the lowest estimated value and follows that path without backtracking. When the value function is accurate enough, this can produce near-optimal solutions. However, learned models are inherently imperfect and can suffer from limited expressiveness or poor generalization to larger test instances with more objects than seen during training, leading to inaccurate state estimates. These failures are not visible to the greedy policy, which receives no indication that its guidance has become unreliable and has no mechanism to recover once an incorrect action is chosen.

1.2 Contribution

To address this problem, we consider the heuristic search approach. In this approach, the search algorithm explores states based on a given heuristic function that can be manually

crafted or learned through learning models [8]. The search algorithm explores multiple states and expand ones even with non-optimal values. This allows the search to recover from poor decisions by backtracking and exploring alternative paths.

Heuristic search has been studied extensively in classical planning, but typically with heuristics that are cheap to evaluate, so search cost is dominated by the number of states examined rather than the computation performed at each one. Several works on learned heuristics use GBFS as the search strategy, with limited comparison against alternative search algorithms [3, 6, 7, 10, 13]. In this thesis, we instead use learned heuristics that can be more informative but are also more expensive to evaluate. It is therefore unclear which search algorithm is best suited to this type of guidance. We therefore compare A*, Weighted A*, Greedy Best-First Search, Beam Search, Q*, Weighted Q*, and MCTS in an AlphaZero-style configuration, representing different search paradigms. They are evaluated across planning domains where greedy execution of the learned model tends to fail, including Barman, Grid, Logistics, Satellite, Rovers, and Goldminer. Each algorithm is compared against the greedy execution of the model that guides it, so that the contribution of search can be separated from the quality of the guidance.

The comparison identifies a property that distinguishes the two strongest algorithms from the rest: neither Beam Search nor AlphaZero-style MCTS maintains a global ordering over an unbounded open list, so neither depends on the scale of the learned estimates but only on the order they set over a state's successors. This matters because even a poorly trained model can estimate distances at the wrong scale while still ranking a state's successors correctly.

The second part of the thesis takes the strongest of these algorithms, AlphaZero-style Monte Carlo tree search (MCTS), and examines how its search can be improved. Its selection rule scores an action by combining the value estimates accumulated during search with an exploration term that is proportional to the learned policy prior. Actions assigned very low prior probability therefore receive almost no exploration bonus and are unlikely to be selected within any realistic search budget, resulting in the tree search repeatedly following actions favored by the policy. To avoid this, we reduce the influence of the learned policy by redistributing the prior probability over actions. We evaluate this modification across the full range from the original learned prior to a completely uniform one, where all applicable actions receive equal probability. The original AlphaZero approach applies the same operation as Dirichlet noise at the root of its search tree during self-play [14], for the same reason. Here it is applied at every node and at evaluation time.

Finally, we ask whether this adjustment can be targeted rather than applied uniformly. We use a distributional value function that predicts a return distribution for each action instead of a single value, allowing the model's prediction to be compared with its own one-step Bellman target. A disagreement between the two indicates local inconsistency in the model's predictions. We investigate whether these inconsistencies identify states where the model's action ranking

should be trusted less, and use the resulting residual to determine both how much prior probability to redistribute and which actions should receive it. The evaluation includes two controls that apply comparable changes to the prior while breaking the connection to the measured inconsistencies. These controls make it possible to separate improvements caused by the Bellman signal itself from those caused simply by modifying the prior.

1.3 Research Questions

Based on these limitations, this thesis investigates how search can improve the robustness of learned guidance while controlling the additional computational cost. This leads to the first research question: across a range of search algorithms guided by the different learned models, which of them recover the most from the failures of greedy execution, at what cost in search effort, and what property of an algorithm accounts for the difference?

The second research question asks how the strongest search approach can be further improved by adapting how it uses imperfect learned guidance. For the best-performing algorithm, AlphaZero-style MCTS, we investigate this in two steps: first by changing how strongly the learned policy prior is trusted, and then by using Bellman inconsistency to determine where that prior should be adjusted.

1.4 Thesis Structure

Chapter 2 introduces the necessary background and reviews the most relevant related work. Chapter 3 describes the search algorithms compared in the first part of the thesis, and Chapter 4 the experimental setup common to all of the experiments reported. Chapter 5 presents the benchmark and answers the first research question. Chapter 6 describes the two modifications to the AlphaZero-style MCTS tree policy, prior flattening and the Bellman inconsistency signal, and Chapter 7 evaluates them, answering the second and third questions. Chapter 8 discusses the limitations and future work and Chapter 9 concludes.

2 Background & Related Works

2.1 Classical Planning

The planning domain definition language (PDDL) is a formal language used in artificial intelligence to describe planning problems. It captures the core components of a planning task, such as states, actions, and goals, allowing a planning system to compute a sequence of actions, a plan, that transforms an initial state into a goal state.

PDDL supports several classes of planning problems, the simplest of which are discrete, finite, fully observable, and deterministic planning problems, commonly known as *classical planning* [9]. A classical planning problem is defined as a pair $P = \langle D, I \rangle$, where D is the domain that defines the state variables (facts that may be true or false) and actions, and I (the instance) defines the initial state and goal condition. In other words, the domain is the set of rules describing the planning world itself, while I is a specific problem instance in the domain that specifies the initial state of the world and the goal we want to achieve [16].

More precisely, D contains a set of predicate symbols and a set of action schemas. Each action schema has preconditions and effects described by atoms of the form $p(x_1, \dots, x_k)$, where each x_i represents an argument of the action schema. An instance $I = \langle O, Init, Goal \rangle$ consists of a set of objects O together with two sets of ground atoms: $Init$ and $Goal$, describing the current state of the environment and the goal state that must be satisfied.

Defining planning problems as two components, D and I, where D describes the general domain and I is specific to a particular instance, is important for defining and computing general policies. A general policy is a strategy that generalizes across instances of the same planning domain rather than solving each instance from scratch. A separate domain description and its instances help planners understand the shared structure across different problem instances, such as common actions and predicates [16].

A classical planning problem can equivalently be described as a state model $P = \langle D, I \rangle$ with

$$S(P) = \langle S, s_0, S_G, Act, A, f \rangle,$$

where:

- S is the set of states, each represented as a set of ground atoms,
- $s_0 \in S$ is the initial state,

- $S_G \subseteq S$ is the (non-empty) set of goal states,
- $A(s)$ is the set of ground actions applicable in state s (i.e., whose preconditions hold in s),
- f is the transition function, where

$$f(a, s_i) = s_{i+1}$$

  gives the successor state resulting from applying action $a \in A(s)$ in state s_i.

A solution to the problem, a plan, is a sequence of actions $a_1, \dots, a_n$ such that applying them successively to s_0 yields a state in S_G.

Despite its relatively simple formulation, classical planning is generally NP-hard [9]. This is mainly due to the size of the state space S. Since a state is a subset of the ground atoms that can be formed from the predicates and objects of the domain, the number of possible states grows exponentially with the number of predicates and objects. Consequently, the state space can become extremely large even for moderately sized planning problems, and no known algorithm can solve all classical planning instances in polynomial time.

2.2 Learned Value Functions and Policies for Planning

The classical planning state model is commonly viewed as a graph, where states correspond to nodes and actions correspond to directed edges connecting a state to its successor states. Finding a plan then amounts to finding a path from s_0 to some goal state $s_G \in S_G$. This graph-based perspective forms the basis for both the greedy policy and the search-based approaches discussed in this thesis.

A natural way to guide planning toward the goal is to associate each state with an estimate of its remaining cost. A value function $V(s)$ assigns such an estimate to every state $s \in S$, with the aim of approximating the true cost-to-go $V^*(s)$, which denotes the minimum number of actions required to reach a goal state from s [8]. An alternative representation is the action-value function $Q(s, a)$, which estimates the cost-to-go after applying action a in state s [19].

A learned value function can directly define a greedy policy π_V. From the current state s, the policy selects the successor $s' = f(a, s)$ with the lowest predicted value $V(s')$ among all $a \in A(s)$, applies the corresponding action, and repeats until a goal state is reached. It does not backtrack or reconsider discarded states. Once V has been trained, greedy execution is computationally cheap as it requires no explicit search over previously encountered states.

The neural-guided approaches investigated in this thesis build directly on the work presented in [15, 16], which use Relational Graph Neural Network (RGNN) to learn general value functions and policies for classical planning domains. In [15], a R-GNN is trained in a supervised manner to approximate the optimal value function $V^*(s)$. The approach builds on results showing

that general policies can be expressed using features constructed from domain predicates through description logics, and that relevant fragments of these logics correspond closely to the expressive power of graph neural networks. The later work [16] instead learns the value function without optimal supervision by minimizing a loss derived from the Bellman equation, trading optimality for stronger generalization to larger instances.

In both approaches, the greedy execution proposed is efficient but offers little robustness to errors in the learned model. Because the policy commits to the successor with the lowest estimated value, an inaccurate prediction can lead it down a path from which it cannot recover, producing a suboptimal plan or failing to reach the goal entirely. This dependence on the learned model becomes particularly important when its architecture cannot represent the required distinctions between states or when it is evaluated on instances substantially larger than those seen during training. Since the greedy policy has no mechanism to detect or compensate for such errors, these limitations motivate the search-based methods investigated in the following chapters.

2.3 Learned Heuristics for Search

The limitations of the greedy execution motivate a more principled approach to navigating the planning state space. Rather than committing greedily to a single path, search algorithms maintain a set of candidate states to explore, while retaining the ability to backtrack and consider alternative paths. Heuristic search guides this process using a heuristic function $h(s)$ that estimates the true cost-to-go $V^*(s)$ from state s to the nearest goal. Since computing $V^*(s)$ exactly is as hard as solving the planning problem itself, $h(s)$ trades accuracy for computational efficiency and may be either manually designed or learned from data.

A separate line of work uses learned value functions directly as heuristics in classical search. In [7], a feed-forward neural network predicts state cost-to-go values and guides greedy best-first search (GBFS) in Fast Downward. Unlike greedy execution, the search can explore multiple states and recover from inaccurate estimates. Later work follows the same general approach while changing the learned heuristic. [13] learns domain-independent heuristics with hypergraph networks, while [3] and [4] learn heuristics over graph representations of grounded and lifted planning tasks; all three evaluate them using GBFS. [1] instead studies how training-sample generation affects heuristic quality while keeping the search algorithm fixed. One of the few papers that discusses the choice of search algorithm, [11], differentiates only between GBFS and A*.

This leaves two important questions largely unexplored. First, it is unclear how the same learned heuristic behaves across different search algorithms and which search strategies are more robust to imperfect predictions. Second, it is unclear whether the additional robustness gained from broader search justifies the extra computational cost of evaluating the learned model more

often. These questions motivate the comparison of neural-guided search algorithms carried out in this thesis.

2.4 Monte Carlo Tree Search in Classical Planning

MCTS is a tree-based search algorithm for navigating combinatorial spaces [18]. Unlike the heuristic-search methods of the previous section, which maintain a flat open list of states, MCTS builds a partial search tree incrementally over many iterations, focusing computation on promising regions of the search space while still exploring alternatives.

Each iteration consists of selection, expansion, evaluation, and backpropagation. During selection, the algorithm traverses the current tree according to a tree policy until it reaches a node with an unexplored child. A commonly used tree policy is upper confidence bound for trees (UCT),

$$a^* = \arg\max_a \left[Q(s,a) + c \cdot \sqrt{\frac{\ln N(s)}{N(s,a)}} \right],$$

where $Q(s,a)$ is the current value estimate for taking action a in state s,, $N(s,a)$ is the number of times that action has been selected, $N(s)$ is the number of visits to the state, and c is the exploration constant, which controls the balance between exploitation and exploration. After selecting an unexplored child, the new node is evaluated, and its value is propagated back through the visited path to update the stored statistics [17].

MCTS was originally developed for adversarial games, where each rollout terminates after a bounded number of moves with a clear win or loss outcome, and the reward signal is typically bounded in $[0, 1]$. Classical planning differs from this setting in two important ways. First, goals are often sparse and state spaces large, so a random rollout from a partially expanded node is unlikely to reach a goal state at all. Second, the reward signal in planning is a cost-to-go estimate, which is unbounded and not confined to a fixed known range, violating the assumptions behind the tree policy formula UCT that is used to select actions during the tree search.

Schulte and Keller [12] reformulate MCTS for classical planning by using a trial-based heuristic tree search (THTS) framework rather than learned methods, and instead adapt the structure of the search itself to better fit classical planning. They show that standard best-first search algorithms such as A*, Weighted A*, and GBFS can each be expressed as THTS instances by choosing an appropriate action selection rule, initialization procedure, and backup function. Using this same framework, they adapt UCT to classical planning, but find that its default Monte-Carlo backup, which averages reward estimates over a node's children, is poorly suited to deterministic planning, where a single best path matters more than an expected outcome.

They therefore introduce backup variants, such as GreedyUCT*, that instead propagate the minimum value among a node's children, similar to how GBFS or A* select their next state. They show that this combination outperforms plain UCT in both coverage and plan quality.

They also introduce a transposition table that detects when newly generated nodes correspond to states already present elsewhere in the tree. They detect when a state is reached via a cheaper path and relocate the existing subtree rather than re-expand it. As a result, parts of the search tree are converted into a directed acyclic graph to avoid redundant exploration of the same state through different action sequences.

2.5 Relational Graph Neural Networks

The value functions and policies used to guide search in this thesis are learned with Relational Graph Neural Networks (R-GNNs) [16]. Unlike a conventional graph neural network, where message passing is defined over an explicit edge structure, an R-GNN operates directly on the ground atoms of a planning task, allowing it to represent states with varying numbers of objects and relational structures.

Input: The network receives the current state, the applicable actions, and the goal, all encoded as ground atoms. A node is created for each object and applicable ground action in the planning instance. All node embeddings are initialized to zero, so information about objects and actions enters only through the relations in which they participate.

The network uses three types of ground atoms. A state atom $p(o_1, \dots, o_k)$ uses the predicate relation p. Goal atoms use separate relations depending on whether they are already satisfied, written as $p^{\text{goal-true}}$ and $p^{\text{goal-false}}$. An applicable action $a = op(o_1, \dots, o_k)$ uses a relation corresponding to its action schema, where the action node is included together with its object arguments.

Message passing: Each predicate and action-schema relation has its own multilayer perceptron (MLP). For a ground atom with ordered argument nodes $(i_1, \dots, i_k)$, their embeddings are concatenated and transformed as

$$x = [\, h_{i_1}^{(t)} \,\|\, \cdots \,\|\, h_{i_k}^{(t)} \,], \qquad [\, m_1 \,\|\, \cdots \,\|\, m_k \,] = x + \mathrm{MLP}_r(x). \tag{2.1}$$

The vector m is split into k blocks, one message for each argument position, and these messages are sent back to their corresponding nodes. The arguments are ordered, so exchanging them changes the messages they receive. An atom with one argument sends its message back to its single object node, which is how unary facts such as $truck(t_0)$ enter an embedding that starts at zero.

Aggregation and update: Each node aggregates the messages it receives and combines them with its current embedding through a residual update,

$$h_v^{(t+1)} = h_v^{(t)} + \mathrm{LayerNorm}\big(\mathrm{MLP}_{\mathrm{upd}}[\, a_v \,\|\, h_v^{(t)} \,]\big). \tag{2.2}$$

The same relation and update parameters are shared across all message-passing iterations.

Readouts: After the final iteration, the state-value model pools the object embeddings,

$$V(s) = \mathrm{MLP}_V\Big(\sum_{o \in \mathcal{O}} h_o\Big), \tag{2.3}$$

with $\mathcal{O}$ denoting the set of object nodes.

The action-value model instead scores each action node together with a pooled object representation,

$$Q(s,a) = \mathrm{MLP}_A\Big[h_a \,\Big\|\, \mathrm{MLP}_O\Big(\sum_{o \in \mathcal{O}} h_o\Big)\Big]. \tag{2.4}$$

The policy uses the same action readout and applies a softmax over the applicable actions, while the distributional model of Section 2.6 passes the action embeddings to a separate quantile head.

Because the architecture is defined over predicates and ground atoms rather than a fixed-size input, the same model can be applied to planning instances with different numbers of objects.

2.6 Distributional Action-Value Functions

A standard action-value function $Q(s,a)$ predicts a single expected return for each action. Distributional reinforcement learning instead models the full return distribution $Z(s,a)$. In this thesis, we use Quantile Regression Deep Q-Network (QR-DQN) [5], which represents the distribution using M quantile values of equal weight at fixed midpoint fractions

$$\tau_i = \frac{i + 0.5}{M}, \qquad i = 0, \dots, M-1, \tag{2.5}$$

with $z_i(s,a)$ denoting the value predicted at τ_i. The corresponding scalar action-value estimate is the mean of these quantiles,

$$\mathbb{E}[Z(s,a)] = \frac{1}{M} \sum_{i=0}^{M-1} z_i(s,a). \tag{2.6}$$

The QR-DQN model uses a relational graph neural network architecture that produces a separate distributional prediction for each applicable action. It is trained by distributional Q-learning using the quantile Huber loss, with a reward of $r = -1$ per action and a discount factor of $\gamma = 0.999$. Returns are therefore negative, and larger values of $\mathbb{E}[Z(s, a)]$ correspond to actions expected to reach the goal in fewer steps. Unlike implicit quantile network (IQN), which samples quantile fractions during evaluation, quantile regression deep Q-network (QR-DQN) uses the same fixed set of quantiles at every state.

2.7 Limitation of existing approaches

Learned value functions and policies remain limited by two main problems: errors in greedy decision-making and inaccuracies caused by limited model expressiveness or poor generalization. Greedy execution approaches based on learned value functions commit to the predicted best successor without backtracking, so errors in the learned estimates cannot be corrected later. These limitations are especially problematic in NP-hard domains, where even small prediction errors can cause the greedy policy to fail.

Heuristic search can mitigate these problems, but at a higher computational cost because it expands and evaluates multiple states rather than committing to a single path. This increases the number of node expansions and the overall runtime. Although heuristic search still depends on imperfect heuristic estimates, it can backtrack and explore alternative paths when local errors occur, making it more robust than greedy execution.

Current research largely focuses on making learned heuristics faster and cheaper to evaluate without sacrificing too much accuracy. However, less attention has been given to systematically studying how different search algorithms behave when guided by the imperfect learned models. In practice, these value functions can be noisy or inconsistent, while classical search algorithms were originally designed assuming different heuristic properties. A* is the clearest example: it relies on assumptions that the heuristics is admissible and consistent, which learned neural value functions may violate. As a result, A* may need to reopen states or spend substantial search effort exploring misleading parts of the state space.

This gap motivates the central question of this thesis: in domains where greedy policies fail, which search algorithms best balance exploration vs exploitation under imperfect learned guidance? Beyond this, this thesis also investigates whether search itself can be made more efficient under imperfect guidance, asking whether model's self-inconsistency can be used as a signal to guide exploration and improve search decisions.

3 Learning-Guided Search Algorithms

Learned value functions and policies can provide useful guidance for classical planning, but their predictions are inherently imperfect, especially when applied to instances outside their training distribution or to domains that exceed their expressive capabilities. This is a major problem for greedy execution of such models because, once an incorrect action is selected, the policy continues to commit to the same path without backtracking or considering alternative states.

Search mitigates this problem by recovering from local mistakes and exploring alternative states, thereby increasing robustness to errors in the learned model. However, this robustness comes at the cost of additional state expansions and model evaluations. Different search algorithms address this trade-off by balancing exploration and exploitation differently, aiming for an optimal compromise between robustness, plan quality, and search effort.

This chapter introduces the search algorithms evaluated directly in Chapter 5 and the learned models used to guide them. Their comparison addresses the first research question: in domains where greedy policies fail, which search algorithms best balance robustness, plan quality, and search effort under imperfect learned guidance?

3.1 Greedy Baselines

To evaluate whether search improves over direct execution of the learned guidance, we use three greedy baselines corresponding to the learned models considered in this thesis: Greedy-V, guided by a state-value function $V(s)$; Greedy-Q, guided by an action-value function $Q(s, a)$; and Greedy-SAC, guided by a learned soft actor-critic (SAC) policy $\pi(a \mid s)$. Each baseline greedily selects one action at every state and follows the resulting trajectory without backtracking.

For Greedy-V, all applicable actions $a \in A(s)$ are applied to generate their successor states $s' = f(a, s)$. The action leading to the successor with the lowest predicted cost-to-go is selected:

$$a^* = \arg\min_{a \in A(s)} V(f(a, s)). \tag{3.1}$$

The successors are evaluated in a single batch at each state. An execution of length ℓ therefore requires ℓ batched forward passes, containing a total of $\sum_{i=1}^{\ell} |A(s_i)|$ successor evaluations.

The second baseline, **Greedy-Q**, uses the learned action-value function directly:

$$a^* = \arg\max_{a \in A(s)} Q(s, a). \tag{3.2}$$

Since one forward pass scores all applicable actions, an execution of length ℓ requires ℓ forward passes. The Q-model uses a per-step reward of $r = -1$, so larger values correspond to shorter expected distances to the goal, whereas V predicts cost-to-go directly.

Finally, **Greedy-SAC** follows the learned SAC policy by selecting the action with the highest probability,

$$a^* = \arg\max_{a \in A(s)} \pi(a \mid s). \tag{3.3}$$

Together, these baselines measure the performance of each learned model under direct greedy execution, allowing the corresponding search-based methods to show whether exploring alternative states compensates for errors in the learned guidance.

3.2 Value-based Search Algorithms

The first family of search methods uses the learned state-value function V(s) as a heuristic estimate of the remaining cost to the goal. In contrast to greedy execution, these algorithms retain multiple candidate states and can therefore backtrack from local errors and reconsider alternative paths. The same learned state-value model is used throughout this family, so their performance differences mainly reflect how strongly each algorithm relies on the learned estimate and how much alternative search it retains.

3.2.1 A* and Weighted A*

A* evaluates each candidate state using:

$$f(s) = g(s) + h(s), \tag{3.4}$$

where $g(s)$ is the path cost accumulated from the initial state to s, and $h(s) = V(s)$ is the learned estimate of the remaining cost to the goal. Since the learned heuristic is neither guaranteed to be admissible nor consistent, the usual optimality guarantees of A* do not necessarily apply.

Weighted A* increases the influence of the learned heuristic by modifying the evaluation function to,

$$f(s) = g(s) + w\,h(s), \qquad w > 1, \tag{3.5}$$

so larger values of w place greater reliance on the learned estimate and reduce the relative influence of the accumulated path cost.

Both A* and Weighted A* follow the same expansion pattern. They expand one state at a time. Although the successors of that state can be evaluated together, this limits the size of the batches sent to the GPU compared with Beam Search.

3.2.2 Greedy Best-First Search

Greedy Best-First Search (GBFS) removes the accumulated path cost entirely and prioritizes states solely according to their heuristic:

$$f(s) = h(s) = V(s). \tag{3.6}$$

States predicted to be closest to the goal are expanded first regardless of the cost already incurred in reaching them. This can make the search efficient when the learned value function ranks states accurately, but it also makes greedy best-first search (GBFS) heavily dependent on the quality of these estimates.

3.2.3 Beam Search

Beam Search limits the number of candidate states to k (the beam size) retained during the search. At each search step, the successors generated from the current beam are evaluated using the learned value function. Their value estimates are then sorted, and only the k most promising states are retained for the next step.

The parameter k directly controls the exploration-exploitation trade-off in the algorithm. A small beam strongly exploits the learned value estimates by discarding most alternative states, while a larger beam retains more alternatives and therefore increases the chance of recovering from inaccurate rankings, at the cost of evaluating and retaining more states.

Beam Search is also particularly well suited to GPU batching, since it can collect the successors generated from all states in the current beam and evaluate them together in a single large batch.

To study the effect of beam width, Chapter 5 evaluates a wide range of beam sizes, allowing us to examine how retaining more alternatives affects coverage, solution quality, and search effort across domains.

Beam Search also differs from the other value-based methods in how it uses the learned values. A*, Weighted A*, and GBFS use the numerical magnitude of $V(s)$ directly in their evaluation functions, whereas Beam Search uses the values only to rank candidates within each step. Its behaviour therefore depends primarily on whether the model orders states correctly rather

than on whether its predicted values are well calibrated, an advantage that becomes clear in the Chapter 5 results.

3.3 Q-Based Search

Q-based search replaces the learned state-value heuristic $V(s)$ with an action-value function $Q(s, a)$, which directly scores the applicable actions in the current state. This allows all action values to be obtained in a single model evaluation while still maintaining alternative states.

3.3.1 Q* and Weighted Q*

Q* uses the learned action-value function in place of the state heuristic used by A*. Since the Q-network is trained with a reward of -1 per action, larger Q-values correspond to shorter expected paths to the goal, so $-Q(s, a)$ can be interpreted as an estimate of the remaining cost after selecting action a in state s.

The search maintains a priority queue over state–action pairs using the evaluation function

$$f(s, a) = g(s) + w(-Q(s, a)), \tag{3.7}$$

where $g(s)$ is the path cost from the initial state to s, and w controls the influence of the learned action-value estimate. For $w = 1$, this gives Q*, while larger values produce Weighted Q* by increasing the influence of the learned estimate.

The weight is applied only to the heuristic term. This is important because the learned Q-values can be compressed relative to the true cost-to-go. If w is too small, $g(s)$ can dominate the score and make the search behave more like uniform-cost search. Increasing w compensates for this scale mismatch.

3.4 Tree-based Approach

Unlike the previous methods, which prioritize candidate states through a global evaluation function, the tree-based approach organizes search around repeated traversals of a partial search tree. Each state corresponds to a node in the tree, while actions correspond to edges leading to successor states. Search effort is distributed among these edges according to both their estimated quality and how often they have already been explored.

The tree-based method considered in this thesis is an AlphaZero-style variant of MCTS. It combines a learned policy, which indicates which actions are initially promising, with learned value estimates that assess states encountered during search.

3.4.1 AlphaZero-style MCTS

AlphaZero-style MCTS improves classical MCTS by integrating a learned policy and a learned value network [14]. During selection, the learned policy provides a prior over actions, while visit counts and value estimates determine how search effort is distributed. When a new state is expanded, its value is estimated directly by the learned value network and propagated back through the visited path.

Concretely, each simulation descends from the root by selecting, at each node, the action maximizing:

$$\text{score}(a) = Q_{\text{norm}}(s,a) + c \cdot P(a \mid s) \cdot \frac{\sqrt{N(s)}}{1 + N(s,a)}. \tag{3.8}$$

where $N(s)$ is the number of simulations that have passed through s, $N(s,a)$ the number of times action a has been selected in s, c an exploration constant, and $Q_{\text{norm}}(s,a)$ the normalized action-value estimate. The first term favors actions with high estimated values, while the second favors actions that the policy predicts to be promising but that have been selected infrequently.

Classical MCTS assumes rewards are bounded to a certain range, typically [0,1]. The exploration term of the tree policy is then derived under this assumption. However, this assumption doesn't hold with cost-to-go estimates in planning, which are unbounded and vary in scale across domains and instances, so no fixed c balances the two terms. To keep the two terms comparable in scale, action-value estimates are min–max normalized against the range of values observed so far in the search.

Two additional adaptations follow the planning formulation of Schulte and Keller [12]. First, we use a transposition table that shares states reached through different action sequences rather than expanding them repeatedly. States already visited on the current descent are excluded to avoid cycles, while dead ends receive a fixed low value. While Schulte and Keller back up the minimum value because their search is formulated in terms of cost, our formulation uses rewards, where higher values are better, so we back up the maximum instead. In both cases, the backup selects the best child rather than averaging over alternatives.

In this thesis, due to computational constraints, we consider using AlphaZero-style MCTS post-training, using pre-trained policy and value networks to guide search rather than performing full self-play training. For each instance, a single search space is built from the initial state, and no action is committed during search, so the tree remains rooted at the initial state throughout. When a goal state is reached, the corresponding path defines a plan, and the shortest plan found is retained. The search terminates when a solution is found or when the computational budget is exhausted.

4 Experimental Setup

4.1 Domain Selection

In this work, we focus on planning domains where purely greedy approaches often fail due to limitations in the model's expressiveness or its ability to generalize to larger instances. These are domains where local decisions based on approximate value function estimates can easily lead to suboptimal plans or long execution paths. This highlights the need for search mechanisms that enable backtracking and consideration of alternative paths.

The domains where purely greedy approaches fail include:

- **Barman**: a planning domain involving the preparation of cocktails by combining ingredients using various tools, requiring precise sequencing of actions and resource management.
- **Grid**: navigating a structured environment to move objects or visit locations, emphasizing spatial reasoning and global planning beyond greedy decision-making.
- **Logistics**: transporting packages between locations using different modes of transport (e.g., trucks and airplanes), requiring coordination and multi-step planning.
- **Satellite**: scheduling and executing imaging tasks with satellites, involving resource constraints such as instrument availability and orientation.
- **Rovers**: controlling planetary rovers to achieve scientific goals, requiring navigation, resource management, and task sequencing under constraints.
- **Goldminer**: collecting gold in a grid-like environment while handling obstacles and dependencies, requiring strategic exploration and planning.

For each of these domains, we train the four models used in this thesis: the supervised value function $V(s)$, the action-value function $Q(s, a)$, the soft actor-critic (SAC) policy, and the distributional model of Section 2.6. The one exception is Barman, where the training of the SAC policy did not converge, so no policy model is available for that domain. Barman is therefore evaluated only with the algorithms guided by $V(s)$ and $Q(s, a)$, and is excluded from the greedy SAC policy baseline, from the tree-based search of Section 3.4, and consequently from all experiments in Chapter 7.

4.2 Instance Generation

All instances are generated using the official generator for each domain from the PDDL generators collection.[1] Since the generator parameters control structural properties rather than the final number of objects directly, instance size cannot be determined in advance. We therefore generate candidate instances first and filter them afterward. For each domain, the training, validation, and test splits are generated using the same procedure, differing only in the accepted object-count range. Every retained instance is then passed to Fast Downward to verify solvability and to compute their respective optimal plans we use A* with the LM-cut heuristic.

4.3 Data Splits and Test Distributions

Each domain is divided into a training, a validation, and a test split, produced as described in Section 4.2 and separated by the object-count range for each split. Table 4.1 reports the resulting ranges for train and validation splits, which are shared by all of the experiments in this thesis. The models are trained on these sets once and evaluated unchanged on each of the test sets described below. The training split has the smallest object sizes, the validation split uses larger instances, and the test split contains the largest instances in every domain.

Table 4.1: Training and validation splits, by object count.

Domain	Split	Min	Max	Mean	n
Grid	Train	4	15	9.8	400
	Val	16	30	23.0	120
Barman	Train	12	23	17.5	400
	Val	22	30	25.9	120
Logistics	Train	9	18	13.7	400
	Val	15	25	20.0	120
Rovers	Train	11	40	24.6	400
	Val	41	50	45.5	120
Satellite	Train	8	13	10.5	400
	Val	14	25	19.5	120
Goldminer	Train	6	44	26.5	400
	Val	44	51	50.4	120

Both object count and distance to the goal affect how difficult it is for a learned model to guide search. Object size determines how far the model must generalize beyond the instances seen during training, while distance to the goal determines how many decision steps the model must guide reliably. Since the standard test split varies both at once, we use separate test sets to

[1] `https://github.com/AI-Planning/pddl-generators`

distinguish their effects. The unbounded set leaves both unconstrained, while the bounded set preserves the same object counts but limits the remaining distance to the goal. This isolates the effect of size and also keeps searches short enough so that the additional cost of the Chapter 5 modifications does not dominate.

We therefore evaluate on two test sets, summarized in Table 4.2. The unbounded set is used for the main benchmark of Chapter 6, while the bounded-distance set separates the effect of larger instance size from the additional difficulty caused by longer solution paths and is used for the later Bellman-signal experiments.

Table 4.2: The two test sets used in this thesis, defined by object count and distance to the goal. The unbounded out-of-distribution set is the standard test split for each domain. The bounded-distance set keeps the same object sizes but restricts the remaining distance to the goal to $d \in \{5, \dots, 22\}$, matching the range seen during training.

Domain	Test set	Min	Max	Mean	n	d
Grid	Out-of-distribution (unbounded)	22	179	72.2	113	—
	Out-of-distribution (bounded)	22	179	72.2	113	5–22
Logistics	Out-of-distribution (unbounded)	22	35	28.3	113	—
	Out-of-distribution (bounded)	22	35	28.3	113	5–22
Rovers	Out-of-distribution (unbounded)	50	95	74.0	120	—
	Out-of-distribution (bounded)	50	95	74.0	120	5–22
Satellite	Out-of-distribution (unbounded)	48	95	74.0	120	—
	Out-of-distribution (bounded)	46	95	75.0	120	5–22
Goldminer	Out-of-distribution (unbounded)	49	100	77.7	120	—
	Out-of-distribution (bounded)	49	100	77.9	118	5–22

The *bounded-distance* set isolates the effect of instance size from the effect of distance to the goal. Rather than generating new instances, it is constructed from the out-of-distribution (unbounded) test set by tracing the optimal plan of each instance backward from the goal and selecting the state that lies exactly $d \in \{5, \dots, 22\}$ actions before the goal. This state is then used as the initial state of a new planning task. The resulting tasks therefore retain the object counts of the out-of-distribution (unbounded) set while having a known and bounded distance to the goal, with optimal cost exactly d by construction.

4.4 Evaluation Metrics

The performance of each search algorithm can be determined based on five key parameters, which include both efficiency and effectiveness:

- **Coverage** - The number of problem instances for which a solution is found within the specified time limit, reported over all instances of a test set. Coverage is an indicator of the robustness of an algorithm for solving problems in various domains and under various conditions of heuristic guidance.
- **Number of Expanded States** - The total number of states expanded by the algorithm to solve the problem. The fewer the number of states expanded, the lower the computational cost and the more effective the guidance from the heuristic.
- **Plan Length** - The number of actions in the returned plan. All domains considered here have unit action costs, so plan length is the plan cost.
- **Quality** - The ratio of the returned plan length to the optimal plan length, C/C^*, averaged over the instances solved. A value of 1.00 means the returned plans are optimal, and larger values indicate longer plans.

The performance metrics offer an effective way to assess the efficiency and effectiveness of each algorithm in solving planning problems.

4.5 Model Training and Computational Budgets

To ensure comparability, all search algorithms operate on the same planning instances and use the same training and validation splits. The learned models are trained once per domain and kept unchanged during evaluation, and the algorithms are compared under the same computational budgets, 1800s, so that performance differences are attributed to the search strategy rather than to differences in the guidance it receives.

All learned models use the Relational Graph Neural Network (R-GNN) architecture. Four separate models are trained per domain: supervised state-value function $V(s)$ using optimal distance-to-goal labels from Fast Downward, an action-value function $Q(s, a)$ trained with deep Q-network (DQN), a policy $\pi(a \mid s)$ with two critics trained using discrete Soft Actor-Critic (SAC), and a distributional action-value function $Z(s, a)$ trained by quantile regression as described in Section 2.6. The three reinforcement-learning models use a reward of -1 per action and a discount factor of $\gamma = 0.999$ and do not require optimal labels.

5 Neural-Guided Search Benchmark Results

5.1 Overview

This chapter benchmarks the algorithms of Chapter 3 on the out-of-distribution (unbounded) test set of Section 4.3 under the wall-clock limit of Section 4.5. Results are grouped by their respective learned model used for guidance: value-based methods use $V(s)$, Q-based methods use $Q(s, a)$, and tree-search methods use the learned SAC policy and critics. Comparisons between algorithms that use the same underlying model mainly reflect differences in search strategy, whereas comparisons across different model families also reflect differences in the quality of the learned models themselves.

5.2 Greedy Execution

Table 5.1: Greedy Execution Algorithms. **Avg. Expanded** and **Avg. Solution Length** are computed over the *common set* **n** of instances solved by every algorithm in a certain domain

Domain	Algorithm	Coverage	Avg. Expanded	Avg. Solution Length
Grid Common set n=22	Greedy-V	22/113 (19.5%)	66.1	66.1
	Greedy-Q	88/113 (77.9%)	71.5	71.5
	Greedy-SAC	**93/113 (82.3%)**	**63.4**	**63.4**
Logistics Common set n=96	Greedy-V	96/120 (80.0%)	29.1	29.1
	Greedy-Q	103/120 (85.8%)	44.7	44.7
	Greedy-SAC	**108/120 (90.0%)**	**23.4**	**23.4**
Rovers Common set n=13	Greedy-V	17/120 (14.2%)	206.5	206.5
	Greedy-Q	**65/120 (54.2%)**	104.5	104.5
	Greedy-SAC	51/120 (42.5%)	**36.8**	**36.8**
Satellite Common set n=19	Greedy-V	37/120 (30.8%)	**54.6**	**54.6**
	Greedy-Q	45/120 (37.5%)	92.6	92.6
	Greedy-SAC	**55/120 (45.8%)**	69.8	69.8
Barman Common set n=2	Greedy-V	8/109 (7.3%)	157.5	157.5
	Greedy-Q	**52/109 (47.7%)**	**57.0**	**57.0**
	Greedy-SAC	0/109 (0.0%)	—	—
Goldminer Common set n=56	Greedy-V	0/120 (0.0%)	—	—
	Greedy-Q	74/120 (61.7%)	**40.3**	**40.3**
	Greedy-SAC	**84/120 (70.0%)**	43.2	43.2

Table 5.1 reports the three greedy baselines of Section 3.1. Greedy-SAC is the strongest baseline in most domains where it is available, but not in all of them: in Rovers, Greedy-Q performs

markedly better. Overall, no model is best across every domain, and the choice of model cannot be determined independently of the domain.

The two zero entries in Table 5.1 have different causes. Barman has no Greedy-SAC row at all, because the training runs of the policy completed without diverging and plateaued at a single solved validation instance. In Goldminer, the supervised value function trains successfully but fails to generalize: its mean error is 1.19 steps on training states, while on the test set it predicts an average distance of 698 where the true distance is around 29. Greedy-V therefore solves no Goldminer instances, and the same model accounts for the value-based results reported in Section 5.3.

The remaining domains show that accurate local predictions do not necessarily translate into successful greedy execution. In Grid, $V(s)$ selects an optimal successor on 85.5% of sampled optimal-plan states, compared with 72.5% for the SAC policy, yet Greedy-V solves only 19.5% of the instances while Greedy-SAC solves 82.3%. A similar pattern holds for $Q(s, a)$, which ranks actions more accurately than the SAC policy at individual decisions in all five domains, while achieving lower instance-level coverage in four. The problem is that these measurements are on states lying along optimal plans, which resemble the states the models were trained on, whereas the test instances contain far more objects. When the greedy policy makes a wrong decision, it moves away from the optimal path and continues evaluating states on which the model may be less reliable. Since in greedy execution alternative choices are never reconsidered, these errors compound over the remainder of the execution.

The failure traces show how these coverage failures develop during greedy execution. Between 96% and 100% of unsolved instances exhaust the step limit, with 76%–99% of their steps revisiting previously encountered states. Greedy execution initially deprioritizes successors that have already been seen, but once all successors of the current state have been visited, this preference is removed and the policy can repeatedly return to earlier states and oscillate until the limit is reached. Since it has no mechanism to detect and recover from local errors, these baselines provide the reference for the following sections, which examine whether search can overcome these generalization and expressivity failures.

5.3 Value-Based Search

Table 5.2 reports the algorithms guided by the supervised value function. A* solves fewer instances than Greedy-V in five of the six domains, most clearly in Logistics, where coverage falls from 80.0% to 30.8%. At the same time, it returns the shortest plans in every domain. This reflects how A* reacts to heuristic error: when the learned heuristic underestimates the remaining distance, many states receive similar f-values and must be expanded before the search progresses. As a result, A* often exhausts the time limit, but when it does solve an instance, it tends to return very short plans.

Table 5.2: Value-Based Search. **Avg. Expanded** and **Avg. Solution Length** are computed over the common set **n** of instances solved by every algorithm in a certain domain

Domain	Algorithm	Coverage	Avg. Expanded	Avg. Solution Length
Grid Common set n=19	Greedy-V	22/113 (19.5%)	**23.3**	23.3
	A*	41/113 (36.3%)	206.6	19.1
	W-A^*_2	64/113 (56.6%)	27.3	19.7
	W-A^*_5	60/113 (53.1%)	73.7	20.0
	GBFS	45/113 (39.8%)	2 812.4	26.1
	Beam (k=200)	**73/113 (64.6%)**	2 080.6	**18.9**
Logistics Common set n=37	Greedy-V	96/120 (80.0%)	**18.1**	**18.1**
	A*	37/120 (30.8%)	976.4	**18.1**
	W-A^*_2	100/120 (83.3%)	18.4	**18.1**
	W-A^*_5	100/120 (83.3%)	18.4	**18.1**
	GBFS	**108/120 (90.0%)**	18.2	18.2
	Beam (k=200)	106/120 (88.3%)	1 678.1	**18.1**
Rovers Common set n=7	Greedy-V	17/120 (14.2%)	54.7	54.7
	A*	16/120 (13.3%)	609.7	11.6
	W-A^*_2	31/120 (25.8%)	18.3	**11.4**
	W-A^*_5	39/120 (32.5%)	**16.9**	11.6
	GBFS	17/120 (14.2%)	2 099.4	14.7
	Beam (k=40)	**55/120 (45.8%)**	400.7	18.7
Satellite Common set n=17	Greedy-V	37/120 (30.8%)	69.0	69.0
	A*	24/120 (20.0%)	853.4	**15.9**
	W-A^*_2	42/120 (35.0%)	74.4	18.1
	W-A^*_5	43/120 (35.8%)	**56.7**	18.5
	GBFS	29/120 (24.2%)	228.4	22.4
	Beam (k=80)	**54/120 (45.0%)**	840.1	17.1
Barman Common set n=6	Greedy-V	8/109 (7.3%)	—	—
	A*	0/109 (0.0%)	—	—
	W-A^*_2	0/109 (0.0%)	—	—
	W-A^*_5	9/109 (8.3%)	1 951.8	**72.3**
	GBFS	11/109 (10.1%)	**503.5**	82.2
	Beam (k=40)	**50/109 (45.9%)**	9 118.5	288.7
Goldminer Common set n=1	Greedy-V	0/120 (0.0%)	—	—
	A*	0/120 (0.0%)	—	—
	W-A^*_2	1/120 (0.8%)	22 539.0	**48.0**
	W-A^*_5	1/120 (0.8%)	7 281.0	54.0
	GBFS	1/120 (0.8%)	**3 351.0**	65.0
	Beam (k=200)	**113/120 (94.2%)**	11 661.0	71.0

Weighted A* improves coverage substantially by increasing the influence of the learned heuristic. With $w = 2$, it solves every instance solved by A* and additionally solves 63 instances in Logistics, 23 in Grid, 18 in Satellite, and 15 in Rovers. On the common solved set, Weighted A* with $w = 2$ requires only about one-sixth to one-half as many expansions as standard A*. In Logistics, for example, A* completes eleven f-layers before reaching the solution layer, whereas Weighted A* completes none. This suggests that the learned heuristic preserves useful ordering even when its numerical scale is compressed, causing many states to receive very similar f-values. Increasing its weight spreads these values apart and allows the search to follow the heuristic ranking more strongly.

How strongly Weighted A* follows the heuristic depends on the weight w. At smaller weight values, the path cost $g(s)$ still has substantial influence, so the search may continue to expand groups of states with similar f-values. As w increases, Weighted A* relies more strongly on the learned heuristic and less on the accumulated path cost, eventually approaching greedy best-first search. Table 5.2 shows why this can be undesirable: GBFS has lower coverage than Weighted A* in most domains and returns much longer plans. Giving the heuristic too much weight therefore makes the search more vulnerable to the model errors discussed in Section 5.2.

Beam Search is the strongest algorithm in this family across all six domains. Its advantage is particularly large in Goldminer, where it reaches 94.2% coverage despite its poorly trained supervised value model, compared with at most 1.7% coverage for the other value-based methods. Instead of expanding one state at a time, it expands every state in the current layer and retains only the best k. This allows it to preserve multiple alternatives at each depth and rely on the ranking of the learned values rather than their exact numerical scale.

The higher coverage of Beam Search comes at the cost of many more expansions. On Grid instances solved by both Beam Search and A*, both reach solutions at depth 26, but A* expands a median of only 64 states compared with about 3,251 for Beam Search. This additional search effort provides robustness by retaining multiple alternatives at each depth. This difference becomes important when the heuristic is misleading: A* then would spend the remaining budget walking back through a growing open list, whereas Beam Search already preserves alternative states within each layer and can recover from an incorrect ranking without committing to a single path.

Beam width controls how this budget is spent, and no single value is best across domains. Narrow beams are cheaper per level and can reach greater depth: in Rovers, width 5 reaches 191 levels, compared with only 31 for width 200. Wider beams instead examine more alternatives at each level but progress more slowly. This is beneficial in domains such as Logistics and Goldminer, where broader exploration helps compensate for unreliable values, but in Rovers and Satellite, wide beams can exhaust the time limit before reaching the required depth.

Barman is the exception, where the widest beams fail because the beam becomes empty rather than because time runs out.

5.4 Q-Based Search

Table 5.3: Q-Based Search. **Avg. Expanded** and **Avg. Solution Length** are computed over the common set **n** of instances solved by every algorithm in a certain domain

Domain	Algorithm	Coverage	Avg. Expanded	Avg. Solution Length
Grid Common set n=36	Greedy-Q	88/113 (77.9%)	**94.8**	94.8
	Q*	36/113 (31.9%)	6791.4	**29.5**
	W-Q^*_4	94/113 (83.2%)	112.4	32.9
	W-Q^*_8	**102/113 (90.3%)**	102.3	34.7
	W-Q^*_{12}	100/113 (88.5%)	108.2	35.0
Logistics Common set n=80	Greedy-Q	**103/120 (85.8%)**	40.7	40.7
	Q*	80/120 (66.7%)	4314.2	**21.9**
	W-Q^*_4	99/120 (82.5%)	**25.5**	**21.9**
	W-Q^*_8	100/120 (83.3%)	**25.5**	**21.9**
	W-Q^*_{12}	100/120 (83.3%)	**25.5**	**21.9**
Rovers Common set n=12	Greedy-Q	**65/120 (54.2%)**	**45.9**	45.9
	Q*	17/120 (14.2%)	10970.5	**16.0**
	W-Q^*_4	31/120 (25.8%)	260.7	18.6
	W-Q^*_8	29/120 (24.2%)	248.7	18.8
	W-Q^*_{12}	30/120 (25.0%)	284.0	19.4
Satellite Common set n=22	Greedy-Q	45/120 (37.5%)	**88.8**	88.8
	Q*	44/120 (36.7%)	1727.6	**14.0**
	W-Q^*_4	**46/120 (38.3%)**	350.4	16.5
	W-Q^*_8	40/120 (33.3%)	302.4	16.9
	W-Q^*_{12}	43/120 (35.8%)	1811.2	17.3
Barman Common set n=37	Greedy-Q	52/109 (47.7%)	**99.9**	99.9
	Q*	4/109 (3.7%)	—	—
	W-Q^*_4	47/109 (43.1%)	310.6	**57.4**
	W-Q^*_8	**55/109 (50.5%)**	156.5	**57.4**
	W-Q^*_{12}	54/109 (49.5%)	128.1	**57.4**
Goldminer Common set n=1	Greedy-Q	**74/120 (61.7%)**	**26.0**	**26.0**
	Q*	1/120 (0.8%)	31666.0	**26.0**
	W-Q^*_4	1/120 (0.8%)	2308.0	**26.0**
	W-Q^*_8	1/120 (0.8%)	4300.0	27.0
	W-Q^*_{12}	1/120 (0.8%)	5475.0	27.0

* Barman: Q^* solves 4 instances, none of them solved by all other arms, so it is excluded from the common set; its two averages are shown as —.

Table 5.3 reports the algorithms guided by the learned action value function. Q* solves fewer instances with much more expansions than the greedy execution of the same model in all six domains, and by very large margins in four of them, but the plans it returns are far shorter.

Similar to A*, Q* is sensitive to the numerical scale of the learned estimates. In Grid, the predicted Q-values increasingly underestimate the true cost-to-go on deeper test states, compressing the heuristic contribution relative to the accumulated path cost $g(s)$ and causing many states to receive very similar f-values. In a typical Grid search, about 6,000 distinct f-values lie within a range of only about 10.7. Many states therefore receive almost identical scores, so their ordering depends more on the tie-breaking rule than on the learned model. The search consequently behaves more like breadth-first search and expands many states before making progress, resulting in most runs failing due to the time limit. Similar to A*, Q* also returns the shortest plans among the Q-based methods on the instances it solves.

Weighted Q* improves substantially over Q*, achieving higher coverage in most domains while expanding much less states. In Grid, the span of □ values encountered grows from 10.1 at $w = 1$ to 728 at $w = 12$, and coverage rises from 28.3% to 90.3% while the average number of expanded states falls from 1.143 to around 72.

5.5 Tree-Based Search

Table 5.4: Tree-Based Search. **Avg. Expanded** and **Avg. Solution Length** are computed over the common set **n** of instances solved by every algorithm in a certain domain

Domain	Algorithm	Coverage	Avg. Expanded	Avg. Solution Length
Grid	Greedy-SAC	93/113 (82.3%)	**234.9**	234.9
Common set n=89	AlphaZero	**101/113 (89.4%)**	1 140.5	**168.8**
Logistics	Greedy-SAC	108/120 (90.0%)	**70.9**	70.9
Common set n=108	AlphaZero	**111/120 (92.5%)**	614.3	**46.3**
Rovers	Greedy-SAC	51/120 (42.5%)	**102.3**	102.3
Common set n=43	AlphaZero	**65/120 (54.2%)**	752.8	**55.6**
Satellite	Greedy-SAC	55/120 (45.8%)	**97.3**	97.3
Common set n=53	AlphaZero	**89/120 (74.2%)**	1 755.3	**48.4**
Goldminer	Greedy-SAC	84/120 (70.0%)	**47.5**	47.5
Common set n=83	AlphaZero	**112/120 (93.3%)**	626.3	**45.5**

Table 5.4 compares AlphaZero-style MCTS with greedy execution using the same learned policy. It is the only algorithm in the benchmark that improves on its greedy baseline in both coverage and plan length. This differs from the trade-offs observed in the previous sections, where Beam Search improved coverage at the cost of longer plans, while Q* produced shorter plans at the cost of lower coverage.

Because the tree uses the same policy as the greedy baseline, its gains must come from overriding that policy. This happens frequently: along returned plans, the selected action differs from the policy's preferred action on 68.1% of steps in Satellite, 43.9% in Rovers, 38.7% in Grid, 45.9% in Logistics, and 26.5% in Goldminer. In Satellite, the policy is much less confident at overridden steps, assigning its preferred action a median probability of 0.512 compared with 0.986 when that action is followed. In the other domains, however, the tree often overrides even highly confident policy choices. As the preferred action accumulates visits, its exploration bonus decreases and the backed-up value estimates become more influential, allowing alternatives with very small priors to become competitive.

5.6 Comparison and Selection

Table 5.5 collects the coverage, expansions, and plan length of every algorithm considered in this chapter. Three methods stand out. Beam Search is the strongest value-guided method across all six domains, while AlphaZero-style MCTS improves on its greedy baseline in every domain in which it is available and achieves the highest overall coverage in three of the five. Weighted Q* takes the remaining domains, reaching the highest coverage of any algorithm in Grid at 90.3

The expansion and plan-length results in Table 5.5 are computed independently over the instances solved by each algorithm, so they cannot be compared directly across methods. Table 5.6 therefore compares the three strongest methods, and their greedy executions, on the instances they solve in common. AlphaZero expands fewer states than either competitor in four of the five domains in which it runs, showing that its high coverage does not come from simply expanding more states. Beam Search is the most expensive method overall, and it returns the longest plans in three domains. Weighted Q* produces the shortest plans in every domain, but achieves substantially lower coverage in several of them. Its plan-length advantage must therefore be interpreted together with the smaller and generally easier set of instances it is able to solve.

Beam Search and AlphaZero-style MCTS share a robustness to value scaling that the other methods lack. This is important because the learned estimates are often miscalibrated, with mean signed errors ranging from -2.73 in Rovers to $+520.77$ in Goldminer. Beam Search depends only on the ordering of heuristic values, so any strictly increasing transformation leaves the retained states unchanged. AlphaZero normalises values using the running minimum and maximum observed during search, so linear rescaling cancels out and leaves the search identical. A* has neither property: since $f(s) = g(s) + h(s)$ combines the heuristic with the accumulated path cost, both terms must remain on compatible scales for their relative influence to be preserved.

AlphaZero-style MCTS is carried forward into Chapter 6 because it provides the strongest overall basis for further improvement, combining competitive coverage and plan quality with a flexible tree policy whose components can be modified directly. Chapter 6 therefore focuses on modifying the learned prior and exploration constant to make the AlphaZero search more reliable and efficient.

Table 5.5: Performance on the unbounded out-of-distribution test set. Each algorithm is evaluated on the instances it solves; no common solved set is used across algorithms.

	Satellite			Rovers			Logistics		
Algorithm	Cov.	Exp.	Plan	Cov.	Exp.	Plan	Cov.	Exp.	Plan
Value-based search									
Greedy-V	30.8%	100	100.3	14.2%	231	230.9	80.0%	**29**	29.1
A*	20.0%	843	16.5	13.3%	484	**10.6**	30.8%	976	**18.1**
W-A^*_{w*}	35.8% (5)	363	19.7	32.5% (5)	656	26.3	83.3% (5)	104	23.9
GBFS	24.2%	348	24.5	14.2%	2 292	23.8	90.0%	2 205	27.6
Beam$_{k*}$	45.0% (80)	1 235	23.4	45.8% (40)	3 493	204.0	88.3% (200)	5 779	56.0
Q-based search									
Greedy-Q	37.5%	139	139.3	**54.2%**	218	217.9	85.8%	77	77.3
Q*	36.7%	5 549	**16.3**	14.2%	10 462	15.7	66.7%	4 314	21.9
W-Q^*_{w*}	38.3% (4)	1 506	19.6	25.8% (4)	926	36.4	83.3% (8)	49	23.9
Tree-based search									
Greedy-SAC	45.8%	**96**	96.3	42.5%	**137**	136.7	90.0%	71	70.9
AlphaZero	**74.2%**	2 839	77.9	**54.2%**	1 247	62.3	**92.5%**	881	53.5

	Grid			Barman			Goldminer		
Algorithm	Cov.	Exp.	Plan	Cov.	Exp.	Plan	Cov.	Exp.	Plan
Value-based search									
Greedy-V	19.5%	66	66.1	7.3%	161	161.3	0.0%	—	—
A*	36.3%	1 746	**30.7**	0.0%	—	—	0.0%	—	—
W-A^*_{w*}	56.6% (2)	2 646	55.0	8.3% (5)	2 705	79.3	0.8% (2)	22 539	48.0
GBFS	39.8%	4 952	90.0	10.1%	1 293	96.5	0.8%	3 351	65.0
Beam$_{k*}$	64.6% (200)	18 947	116.0	45.9% (40)	11 166	336.4	**94.2%** (200)	69 494	459.7
Q-based search									
Greedy-Q	77.9%	317	316.9	47.7%	**109**	109.4	61.7%	**43**	42.6
Q*	31.9%	6 791	29.5	3.7%	21 311	**53.3**	0.8%	31 666	**26.0**
W-Q^*_{w*}	**90.3%** (8)	1 106	91.5	**50.5%** (8)	206	59.7	0.8% (4)	2 308	**26.0**
Tree-based search									
Greedy-SAC	82.3%	**250**	249.6	0.0%	—	—	70.0%	48	47.9
AlphaZero	89.4%	1 345	191.5	—	—	—	93.3%	1 456	48.2

Table 5.6: Best Search algorithms from the benchmark compared on matched instances. For each domain, **Avg. Expanded** and **Avg. Solution Length** are computed over that domain's common set **n**. Weighted Q^* and Beam Search are each shown at their best-performing setting for that domain, given in the row label.

Domain	Algorithm	Coverage	Avg. Expanded	Avg. Solution Length
Grid Common set n=21	Greedy-Q	88/113 (77.9%)	63.0	63.0
	Greedy-V	22/113 (19.5%)	62.1	62.1
	Greedy-SAC	93/113 (82.3%)	**59.5**	59.5
	W-Q^*_8	**102/113 (90.3%)**	61.0	27.2
	Beam (k=200)	73/113 (64.6%)	2 924.9	**24.0**
	AlphaZero	101/113 (89.4%)	80.8	39.0
Logistics Common set n=96	Greedy-Q	103/120 (85.8%)	44.7	44.7
	Greedy-V	96/120 (80.0%)	29.1	29.1
	Greedy-SAC	108/120 (90.0%)	**23.4**	**23.4**
	W-Q^*_8	100/120 (83.3%)	27.9	**23.4**
	Beam (k=200)	106/120 (88.3%)	2 315.7	23.5
	AlphaZero	**111/120 (92.5%)**	76.7	**23.4**
Rovers Common set n=9	Greedy-Q	**65/120 (54.2%)**	30.7	30.7
	Greedy-V	17/120 (14.2%)	168.2	168.2
	Greedy-SAC	51/120 (42.5%)	**25.0**	25.0
	W-Q^*_4	31/120 (25.8%)	212.2	**18.8**
	Beam (k=40)	55/120 (45.8%)	1 661.2	90.1
	AlphaZero	**65/120 (54.2%)**	70.6	18.9
Satellite Common set n=18	Greedy-Q	45/120 (37.5%)	91.3	91.3
	Greedy-V	37/120 (30.8%)	**56.6**	56.6
	Greedy-SAC	55/120 (45.8%)	63.4	63.4
	W-Q^*_4	46/120 (38.3%)	429.8	**16.9**
	Beam (k=80)	54/120 (45.0%)	891.2	17.2
	AlphaZero	**89/120 (74.2%)**	360.1	23.7
Barman Common set n=1	Greedy-Q	52/109 (47.7%)	**61.0**	**61.0**
	Greedy-V	8/109 (7.3%)	193.0	193.0
	Greedy-SAC	0/109 (0.0%)	—	—
	W-Q^*_8	**55/109 (50.5%)**	67.0	63.0
	Beam (k=40)	50/109 (45.9%)	16 508.0	497.0
	AlphaZero	0/109 (0.0%)	–	–
Goldminer Common set n=56	Greedy-Q	74/120 (61.7%)	**40.3**	**40.3**
	Greedy-V	0/120 (0.0%)	—	—
	Greedy-SAC	84/120 (70.0%)	43.2	43.2
	W-Q^*_4	1/120 (0.8%)	—	—
	Beam (k=200)	**113/120 (94.2%)**	63 882.8	424.3
	AlphaZero	112/120 (93.3%)	291.6	41.1

6 Improving AlphaZero-style MCTS

6.1 Motivation

As shown in Table 5.6, AlphaZero-style MCTS is the strongest in the benchmark, and it is also the one with the most room to improve. Weighted-Q* and Beam Search each expose a single global parameter, so the only thing that can be adjusted is how much exploration takes place everywhere; the guidance itself enters only as a scalar heuristic. The tree policy of Section 3.4.1 draws on three separate sources of information, with only two of them coming from outside the search: the policy prior and the exploration constant that scales it. Both can be modified without touching what the search has measured. The remainder of this chapter is concerned with doing so.

The learned policy enters the selection rule through the exploration term, so actions with near-zero prior probability may never be explored within a realistic budget. This is common in the measured priors, which are often close to deterministic. Flattening the prior addresses this by giving every applicable action some probability, reducing the risk that the tree search simply follows an incorrect policy choice. This is the first modification studied in this chapter. The same modification can also have the opposite effect when the model is already correct, wasting search effort on alternatives that do not need additional exploration. What is therefore missing is not simply more exploration, but a signal indicating where additional exploration is useful.

A value $V(s)$ or action value $Q(s, a)$ predicts how promising a state or action is, but does not indicate whether that prediction is reliable. As discussed in Section 3.1, limited model expressivity or poor size generalization can produce misleading scalar estimates without giving the search any indication that they are unreliable. A distributional value function provides such a signal by predicting a full return distribution Z(s,a) for each action rather than a single number. This prediction is compared with its one-step Bellman target, formed from the immediate reward and the successor-state distribution. Their difference gives a residual that measures local inconsistency. We hypothesize that larger residuals identify states where the model's action ranking is less reliable, and use the signal to decide how much of the prior's weight to move at each state. The aim is to redistribute the same prior weight as uniform flattening, but direct it toward the states and actions with the largest model disagreements.

Three questions guide the remainder of this chapter and the analysis in Chapter 7. First, does flattening the learned prior improve the search, and if so, by how much? Second, can the Bellman inconsistency signal improve on uniform flattening by using the measured Bellman

residuals to direct more of the redistributed prior toward actions where the model is less consistent? Third, if an improvement is observed, does it come from assigning the redistributed prior weight to the right actions, or simply from redistributing more weight overall? Answering these questions is important for showing whether the observed improvements come from the specific modifications being tested rather than from unrelated changes in the search.

6.2 Prior Flattening

The selection rule of Section 3.4.1 uses the prior policy as a multiplier of the exploration term. This means that actions assigned very low prior probability receive little opportunity to be explored. As the preferred actions accumulate visits, such alternatives can eventually become competitive, but if their prior is extremely small, especially when their value estimate is also worse, the required number of simulations may exceed the available search budget. Additionally, actions with zero prior would receive no exploration bonus at all. As a result, search can remain confined to actions already favored by the policy, and when the policy is confidently wrong, the action needed to correct the mistake may never be considered.

We therefore mix the prior with the uniform distribution over the applicable actions before it is used,

$$P_w(a) = (1 - w) \cdot P(a \mid s) + \frac{w}{K_s}, \qquad K_s = |A(s)|, \tag{6.1}$$

where $w \in [0, 1]$ controls how much weight is moved from the learned policy to the uniform distribution. At $w = 0$, the original prior is unchanged, while at $w = 1$ the learned policy is ignored completely and all applicable actions receive equal prior probability. For values between these extremes, every action receives some minimum prior probability while the policy still influences their relative preference.

The operation is not new to AlphaZero. The original algorithm mixes Dirichlet noise into the prior at the root of the search tree, $P = (1 - \varepsilon)P + \varepsilon\eta$, for the same reason: without it, self-play would never try the actions the current policy considers weak, and the policy could not improve beyond them [14]. There, it is applied at a single node and only during training, since at evaluation time the policy is assumed good enough to be trusted. The modification above applies the same mixture at every node but during evaluation. This difference follows from the evaluation setting: in AlphaZero, a policy trained through self-play is evaluated on the same distribution it was trained on, whereas the policies used here are tested on instances substantially larger than those seen during training, making their confidence less reliable.

6.3 Bellman Residual and Error Scaling

The flattening approach applies the same adjustment at every state, regardless of how reliable the model is there. This makes the method unable to distinguish between states where the policy can be trusted and states where its predictions are less reliable. We therefore use the Bellman inconsistency signal to assess the model's own predictions and decide where the prior needs more adjustment and how large that adjustment should be.

6.3.1 The Per-Action Residual

Quantile Regression Deep Q-Network (QR-DQN) model of Section 2.6 predicts, for every action applicable in a state, a return distribution. Because transitions are deterministic, each such prediction can be checked against the prediction the model makes one step later. Let $s' = T(s, a)$ denote the successor reached by applying a in s. The distributional Bellman target for the pair (s, a) is

$$Y_a = \begin{cases} \delta_r, & \text{if } s' \text{ is a goal state,} \\ r + \gamma\, Z(s', b^*), & \text{otherwise,} \end{cases} \tag{6.2}$$

where δ_r denotes the degenerate distribution placing all of its mass at r, and

$$b^* \in \arg\max_{b \in A(s')} \mathbb{E}[Z(s', b)] \tag{6.3}$$

is the action the model considers best at the successor. When several actions attain the same expected value, the one whose target Y_a lies closest to the prediction is used, so that ties are resolved in the direction least favorable to detecting an inconsistency. When the successor is a goal state the model does not need to evaluate the state s', as the remaining return after reaching the goal is exactly the reward of the final action.

The prediction $Z(s, a)$ and the target Y_a describe the same quantity, computed one step apart, so a model that satisfies the Bellman equation assigns them the same distribution. We measure how far the two distributions differ using the 1-Wasserstein distance, computed as the average difference between their quantile functions. Both distributions are represented by M quantile values of equal weight on a shared grid, so this distance reduces to the mean absolute difference between their sorted values. e_a is therefore the Bellman residual for action a, with larger values indicating greater disagreement between the model's current prediction and its one-step Bellman target.

$$e_a \;=\; W_1(Z(s, a),\, Y_a) \;=\; \frac{1}{M} \sum_{i=1}^{M} |z_{(i)}(s, a) - y_{(i),a}|, \tag{6.4}$$

where $z_{(i)}$ and $y_{(i)}$ denote the i-th smallest value of each. No resampling or interpolation between the grids is required, since both sides are evaluated at the same quantile fractions.

Comparing the full curves rather than their means allows us to detect disagreements in the shape of the two distributions even when their mean is similar.

The residual is on the same scale as the predicted returns. Since each action receives a reward of $r = -1$, a residual of one corresponds approximately to a disagreement of one action step between the predicted distribution and its Bellman target. Discounting compresses returns as the distance to the goal increases, but because both $Z(s, a)$ and $r + \gamma Z(s', b^*)$ are affected by the same discounting structure, much of this effect is shared by both sides of the comparison. The residual can therefore still be interpreted approximately in terms of action steps across states at different distances from the goal.

6.3.2 Scaling and Aggregation

The residual has no fixed upper limit, and using it directly would allow a single badly mispredicted action to dominate. We therefore map it onto $[0, 1)$ by

$$x_a = \frac{e_a}{e_a + \tau_{\text{step}}}, \qquad \tau_{\text{step}} = 1, \tag{6.5}$$

which increases with e_a but gradually approaches an upper limit of 1. The parameter τ_{step} controls how quickly this scaling grows. With $\tau_{\text{step}} = 1$, a residual of 1 maps to $x_a = 0.5$, a residual of 2 maps to $x_a = 2/3$, and larger residuals receive progressively smaller additional increases.

The scaled residuals are per-action, and so can indicate which of a state's actions the model is least consistent about. A second quantity is needed to express how inconsistent the state is as a whole, which the exploration term of Section 6.4.1 uses to decide how much to explore there at all. We obtain it by averaging the scaled residuals over the applicable actions,

$$g_s = \frac{1}{K_s} \sum_{a \in A(s)} x_a, \qquad K_s = |A(s)|. \tag{6.6}$$

Since g_s is normalised by the number of applicable actions rather than by a global branching factor, it is comparable across states without further calibration.

Together, x_a and g_s capture two complementary aspects of the Bellman inconsistency signal. The per-action signal x_a measures which of a state's actions the model is least consistent about, while the state-level signal g_s measures the overall inconsistency of the current state.

6.4 Injection into the Tree Policy

The selection rule of Section 3.4.1 combines value estimates, visit counts, the learned prior, and an exploration constant. Only the prior and exploration constant enter from outside the search, making them the two components the signal can modify without changing what the

search has already measured. We therefore adjust both while leaving the accumulated value estimates unchanged.

6.4.1 Flattening the Prior

Before the signal is applied, the prior is mixed with the uniform distribution over applicable actions,

$$P_0(a) = \big(1 - \varepsilon_p\big)\, P(a) + \frac{\varepsilon_p}{K_s}, \tag{6.7}$$

where $P(a)$ is the softmax of the learned policy.

The exploration constant is scaled by the state-level signal,

$$c(s) = c_0(1 + \kappa\, g_s), \qquad c_0 = 1.5, \tag{6.8}$$

so that a state whose actions the model disagrees about receives a larger exploration term relative to its value estimates, while a self-consistent state is left at the baseline constant. The parameter κ controls the strength of this effect.

6.4.2 Additive and Multiplicative Priors

The Bellman signal can be incorporated into the prior in two different ways. The two variants differ in how the per-action residual changes the probability assigned to each action. In the multiplicative variant, an action keeps its original prior probability but is multiplied by a larger factor when its residual is larger. The resulting values are then renormalised so that they sum to one.

$$P_\times(a) = \frac{P_0(a)\,(1 + \beta x_a)}{\sum_{b \in A(s)} P_0(b)\,(1 + \beta x_b)} \tag{6.9}$$

Because the adjustment is proportional to the action's existing prior, actions that start with very low probability can only receive a limited increase, even when their residual is large.

The additive variant instead adds signal mass independently of the existing prior,

$$P_+(a) = \frac{P_0(a) + \beta x_a / K_s}{1 + \beta g_s}, \tag{6.10}$$

The numerator increases each action's prior according to its scaled residual. Since the added terms would otherwise make the probabilities sum to more than one, the denominator renormalises them. Because g_s is the average of the scaled residuals, the total added mass is βg_s, giving the normalising factor $1 + \beta g_s$. Dividing each action's contribution by K_s also prevents the total amount added from increasing simply because a state has more applicable actions.

Unlike the multiplicative variant, the additive form can give large probability to an action even when its original prior was very small. This makes it better able to recover actions that the

learned policy had effectively ruled out, but it also makes the adjustment more aggressive. If the residual ordering is misleading, the additive form can move too much probability away from an otherwise useful policy prior and is therefore more likely to reduce coverage.

In both cases the modified prior replaces $P_\cdot(a)$ in the selection rule, which is otherwise unchanged:

$$\text{score}(a) = Q_{\text{norm}}(s,a) + c(s)\, P_\cdot(a)\, \frac{\sqrt{N(s)}}{1+N(s,a)}, \tag{6.11}$$

with $P_\cdot$ denoting either P_+ or $P_\times$. The modification is applied once, at the moment a node is expanded: the prior computed there is stored on the node and reused at every subsequent selection, and $c(s)$ is stored alongside it. Selection itself is not altered, and the value estimates and visit counts are unaffected.

6.4.3 Parameters

Unless stated otherwise, the experiments use $\tau_{\text{step}} = 1$, $\kappa = 1$, $\beta = 1$, $\varepsilon_p = 0.0$, and $c_0 = 1.5$. The parameter β controls the strength of the signal injection, ε_p sets the size of the prior floor, and κ controls how strongly the state-level inconsistency g_s increases the exploration constant $c(s)$. These defaults are not assumed to be optimal across domains, so their effects are examined separately in Chapter 7.

6.5 Controls: shuffle, random

The additive signal changes the prior in two ways: it determines how much probability mass is moved away from the original prior, and how that mass is distributed across the applicable actions. The controls in this section are designed to separate these two effects. To make them explicit, the additive prior from Section 6.4 can be rewritten as a mixture of the original prior P_0 and a residual-based distribution q:

$$\begin{aligned} P_+(a) &= \frac{P_0(a) + \beta x_a / K_s}{1+\beta g_s} \\ &= \frac{1}{1+\beta g_s}\, P_0(a) + \frac{\beta g_s}{1+\beta g_s} \cdot \frac{x_a}{K_s g_s} \\ &= (1-W)\, P_0(a) + W\, q(a). \end{aligned} \tag{6.12}$$

Here,

$$W = \frac{\beta g_s}{1+\beta g_s}, \tag{6.13}$$

is the total weight moved away from the original prior and depends only on the overall inconsistency g_s.

The distribution

$$q(a) = \frac{x_a}{K_s\, g_s}, \tag{6.14}$$

determines where that weight is placed across the applicable actions and depends on how the residuals differ between its actions. Since $g_s = \frac{1}{K_s} \sum_a x_a$, the values $q(a)$ sum to one.

We therefore compare the full signal with two controls that replace q. The *shuffle* control preserves the residual values measured at each state but randomly reassigns them among the applicable actions. Because the multiset x_a is unchanged, the state-level inconsistency g_s and therefore the redistributed weight W are preserved exactly. The control thus moves the same amount of prior weight at the same states as the full signal, while breaking only the link between a particular action and its residual. Any improvement of the full signal over this control therefore comes from assigning the redistributed weight to the right actions.

The *random* control removes one additional part of the signal. Instead of using the measured residuals, it samples values from a lognormal distribution fitted separately for each domain so that the average g_s, and therefore the average amount of redistributed prior weight, matches the measured signal. However, it does not preserve g_s at each individual state, so the amount of prior weight moved at a state is no longer related to how inconsistent the model is there. This control therefore tests whether the state-level magnitude of the Bellman signal is useful, beyond simply applying a similar average amount of prior modification across the domain.

Together, the three comparisons separate the effect of the Bellman signal from simpler changes to the prior. Prior flattening shows what can be gained by redistributing probability without using inconsistency information; the shuffle control preserves the state-level amount of redistribution while breaking the action-specific assignment; the random control matches the average amount of redistribution produced by the full signal while removing both state-level and action-level information about where the model is inconsistent. Any improvement beyond all three is attributed to the information contained in the Bellman signal itself.

7 Evaluation of AlphaZero-style MCTS Modifications

7.1 Overview

The experiments in this chapter apply the modifications introduced in Chapter 6 to the AlphaZero-style MCTS. The most straightforward evaluation set is the out-of-distribution set used for the benchmark in Chapter 6, as it places huge generalization demand on the learned models. However, this set is impractical for the distributional model for reasons discussed in Section 7.6. The main experiments are therefore conducted on the out-of-distribution set with distance-to-goal bounded between five and twenty-two actions.

The baseline in these experiments is the AlphaZero-style MCTS search using the learned policy prior, with the distributional model of Section 2.6 supplying the leaf values. The modifications are evaluated in two stages. First, the prior-flattening configuration from Section 6.2 mixes a fixed proportion w of uniform probability into the learned prior at every state. Second, the Bellman signal is evaluated in its two forms, additive and multiplicative. Both are applied on top of a prior floor ε_p chosen to match the best flattening result for each domain, and are evaluated across different injection strengths β. To determine whether any improvement comes from the Bellman signal itself rather than from the amount of redistribution alone, the shuffled and random controls of Section 6.5 are evaluated alongside these signal variants.

7.2 Flattening the Prior

Table 7.1 summarizes the strongest prior-flattening configuration for each domain and its effect relative to the unflattened search. These selected weights are then carried into the Bellman-signal experiments through the corresponding ε values, so that the signal is evaluated on top of the best prior-flattening setting for each domain rather than from the original learned prior.

7.2.1 Effect of Prior Flattening

Figure 7.1 shows the effect of varying how strongly the learned policy contributes to the prior, from $w = 0$, where the prior is determined entirely by the policy, to $w = 1$, where the policy is ignored and all applicable actions receive equal prior probability. The main result is that no single value of w performs best across all five domains. The same modification that increases coverage from 74.6% to 100% in Goldminer reduces it to 69.9% in Satellite. The per-domain

Table 7.1: Best prior-flattening configuration for each domain. The baseline corresponds to the original unflattened prior ($w = 0$). Coverage is reported for both the baseline and the selected weight, while expansions and plan length are shown relative to the baseline on the common solved set. Full results across all tested w values are reported in Appendix A.1.

		Coverage		At chosen w	
Domain	Chosen w	w=0	chosen	Expansions	Plan length
Goldminer	0.90	74.6%	100.0%	56.1%	53.9%
Rovers	0.80	90.8%	96.7%	52.0%	91.4%
Logistics	0.80	99.1%	100.0%	143.1%	21.7%
Grid	0.40	100.0%	100.0%	132.1%	79.5%
Satellite	0.00	100.0%	100.0%	100.0%	100.0%

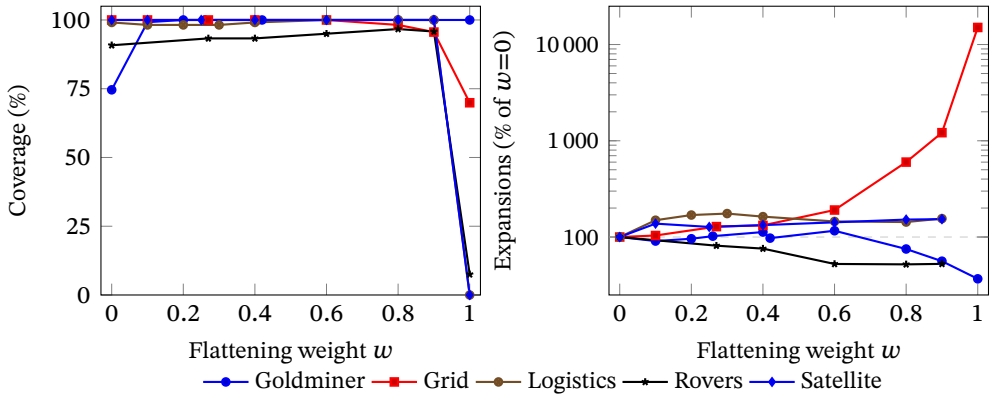


Figure 7.1: Effect of flattening the policy prior. Coverage is measured over each domain's full test set, while expansions are reported relative to the unflattened baseline on the common solved set, using a logarithmic scale. For Logistics, Rovers, and Satellite, too few instances are solved at $w = 1$ to define the expansion ratio.

results in Appendix A.1 reveal additional trade-offs beyond coverage: in Logistics and Grid, flattening substantially improves plan length, while in Rovers and Goldminer the weights that give the best coverage, shortest plans, and fewest expansions differ. No single weight optimizes all metrics. Overall, the benefit of flattening the prior depends strongly on the domain and on how reliable the learned policy is within it.

For four of the five domains, some reliance on the learned policy remains beneficial, although moderate flattening can still improve performance. Goldminer is the exception, where removing the policy entirely improves performance, while Satellite depends most strongly on the learned prior and degrades under any amount of flattening. This shows that flattening is useful only when the policy suppresses helpful alternatives; when the policy is already reliable, reducing its influence can instead make the search less efficient.

7.3 The Bellman Inconsistency Signal

7.3.1 Overall Signal Performance

Table 7.2: Additive Bellman signal and matched controls on the bounded out-of-distribution set. Full sweeps over β and κ are reported in Appendix A.2.

Domain	Configuration	Coverage	Expansions	Plan length	Quality
Goldminer ε=0.90 (Common set n=88) opt. 11.78	Baseline	88/118 (74.6%)	439 (100.0%)	22.57	1.915
	β=1, κ=1	**118/118 (100.0%)**	284 (64.6%)	**12.24**	**1.039**
	Shuffled control	**118/118 (100.0%)**	**181 (41.2%)**	12.34	1.047
	Random control	**118/118 (100.0%)**	220 (50.1%)	12.66	1.074
Grid ε=0.40 (Common set n=105) opt. 12.47	Baseline	**113/113 (100.0%)**	**37 (100.0%)**	22.95	1.841
	β=2, κ=1	109/113 (96.5%)	595 (1595.1%)	**15.42**	**1.237**
	Shuffled control	109/113 (96.5%)	521 (1396.1%)	15.66	1.256
	Random control	109/113 (96.5%)	509 (1365.0%)	15.49	1.242
Logistics ε=0.80 (Common set n=104) opt. 12.75	Baseline	**112/113 (99.1%)**	202 (100.0%)	91.14	7.149
	β=1, κ=1	105/113 (92.9%)	386 (191.2%)	**16.38**	**1.284**
	Shuffled control	110/113 (97.3%)	174 (85.9%)	21.55	1.690
	Random control	110/113 (97.3%)	**113 (55.7%)**	19.81	1.554
Rovers ε=0.80 (Common set n=89) opt. 11.61	Baseline	109/120 (90.8%)	423 (100.0%)	44.96	3.903
	β=4, κ=0	**111/120 (92.5%)**	**143 (33.7%)**	32.35	2.761
	Shuffled control	105/120 (87.5%)	260 (61.5%)	**29.52**	**2.531**
	Random control	103/120 (85.8%)	318 (75.2%)	30.90	2.638
Satellite ε=0 (Common set n=108) opt. 10.82	Baseline	**120/120 (100.0%)**	74 (100.0%)	**11.34**	**1.003**
	β=4, κ=1	118/120 (98.3%)	**26 (34.4%)**	11.36	1.004
	Shuffled control	114/120 (95.0%)	46 (62.4%)	11.58	1.024
	Random control	112/120 (93.3%)	48 (64.7%)	11.60	1.027

Table 7.2 reports the additive signal relative to the AlphaZero-style MCTS baseline on the bounded distance-to-goal test set. The signal uses the ε_p value corresponding to the best flattening result for each domain, while the shuffled and random controls use matched ε_p, β, and κ values. Together, these comparisons test whether any additional improvement comes from the Bellman signal itself and, if so, whether that improvement comes from its action-specific placement or from the state-level inconsistency g_s that controls how much prior weight is redistributed.

In the Satellite and Rovers domains, the Bellman signal clearly outperforms both controls in both coverage and avg. expansion metrics. In Satellite, the signal additionally requires far fewer expansions than both the baseline and the two controls, while in Rovers it returns shorter plans than the baseline. In Logistics, the signal does not reduce expansions. However, it still returns

the shortest plans of the three signal-based configurations. This suggests that the residual can still improve plan quality even when it does not make the search more efficient. Among the controls themselves, however, the shuffled and random configurations show no noticeable difference, even in the domains where the residual's action ordering is informative.

7.4 Does the Bellman Residual e_a Identify Useful Actions?

Whether these differences in performance over the controls follow the pattern predicted by the Bellman residual can be tested directly. The residual e_a measures a disagreement between the model's prediction for an action and the prediction implied one step later, and the hypothesis of Chapter 6 is that such disagreements mark states in which the model's ranking over actions is less reliable. If this is what drives the gains in Satellite and Rovers, the action carrying the largest residual should more often correspond to an action worth taking than one chosen at random.

We measure this by sampling states along the returned plans and comparing how often the action with the largest residual is the action selected by the plan against the rate expected from choosing an applicable action at random. This comparison is possible on the bounded distance-to-goal test set because the optimal remaining action sequence is available for each sampled state. The resulting ratio is 4.4 in Satellite, 2.9 in Rovers, and 2.5 in Logistics, but only 1.01 in Goldminer and 0.99 in Grid. The two domains, Goldminer and Grid, in which the residual ordering is essentially indistinguishable from chance are also those in which the controls match or outperform the Bellman signal.

The comparison between the shuffled and random controls also gives no clear evidence that the state-level magnitude of the residual provides useful information. Although the random control corrupts the measured g_s at each state, its results remain very similar to the shuffled control across all domains. This suggests that defining the state-level inconsistency g_s simply as the average Bellman residual across applicable actions may not capture useful state-level information.

7.4.1 When the Residual Ordering Is Unhelpful

Goldminer: Revisiting Previously Explored Paths

Goldminer illustrates this distinction clearly. Relative to the original baseline, all three configurations substantially reduce expansions: the Bellman signal uses approximately 64.6%, the shuffled control 41.2%, and the random control 50.1%. However, the Bellman signal is less efficient than either control and remains so across different run permutations. This shows that the improvement comes primarily from prior flattening rather than from preserving the Bellman residual ordering. The difference appears in the search behavior: the signal revisits

an already traversed path on 39.7% of descents, compared with 29.6% and 28.3% for the two controls, and this additional revisiting accounts for roughly half of its excess expansions. One possible explanation is that neighbouring states share much of their structure, so similar actions can continue to receive large residuals along a descent. In Goldminer, where the residual ordering is uninformative, repeatedly favoring those actions can steer the search back toward previously explored paths. Shuffling weakens this effect by breaking the consistent association between an action and its residual.

The controls' advantage over the baseline in Goldminer, however, should not be read as evidence that redistributing the prior probability helps. The baseline is the cheapest configuration on 82 of the 118 instances and has a median of only 28 expansions, compared with 66 for the shuffled control. Its higher mean arises because a small number of difficult instances within the common solved set require disproportionately many expansions. Goldminer therefore shows that using an uninformative residual ordering is worse than shuffling it, not that either modification is generally better than leaving the prior unchanged.

Logistics: Deep Search Paths

Useful per-action information is therefore necessary but not sufficient. Logistics shows this most clearly: its residuals are informative, and the signal returns the shortest plans. However, this does not translate into lower avg. expansions. This mismatch is driven by a small subset of difficult instances rather than by a general difference between the configurations. On most instances, all three configurations behave almost identically: their median expansion counts lie between 51 and 53, and their median plan lengths are all 15 actions. The difference in performance is driven by roughly one seventh of the test set, where the signal begins to search much more deeply. On these instances, the signal's search descents reach a mean depth of 37.8 actions, compared with only 2.9 on the remaining instances. No state is revisited during these descents, so the search effectively follows a single increasingly deep path rather than spreading its effort across a broader tree. The controls show the same tendency on these difficult instances, but to a smaller extent, reaching mean depths of 27.4 and 26.0. This is due to shuffling and randomizing the residual assignments, making the search less likely to keep favoring the same deep path. This difference in depth translates to a large difference in avg. expansions, with the signal requiring 1,573 expansions compared with 391 for the controls.

The effect described in Goldminer also appears in Logistics, where actions with large residuals at one state often continue to receive large residuals at successive states along the same path, but here it keeps the search moving along a deep trajectory rather than sending it back toward previously explored paths, which results in short plans. The shuffled control weakens this effect by reassigning the residuals at each state. On the difficult instances where these deep descents occur, the signal returns plans averaging 18 actions, compared with 52 for the shuffled control and 38 for the random control.

7.4.2 What the Residual Fails to Measure

It is important to note that the failure in Goldminer and Grid is not caused by the model being more self-consistent in those domains. The mean scaled residual x_a is 0.354 in Goldminer and 0.366 in Grid, compared with 0.320 in Satellite, so the model shows at least as much local inconsistency in the two domains where the signal is ineffective as in the domain where it performs best. The difference, however, is in whether that inconsistency is informative for action selection. As discussed in Section 6.3, the residual compares the model with its own one-step Bellman target, so it measures disagreement between neighbouring predictions rather than error relative to the true cost-to-go. A model can therefore remain self-consistent while being systematically wrong, and it can also produce large residuals from local fluctuations that do not indicate which successor is better. In three of the five domains, however, the residual ordering is informative enough to help distinguish useful actions; in Goldminer and Grid, it is not.

7.4.3 Effect of β on Search Performance

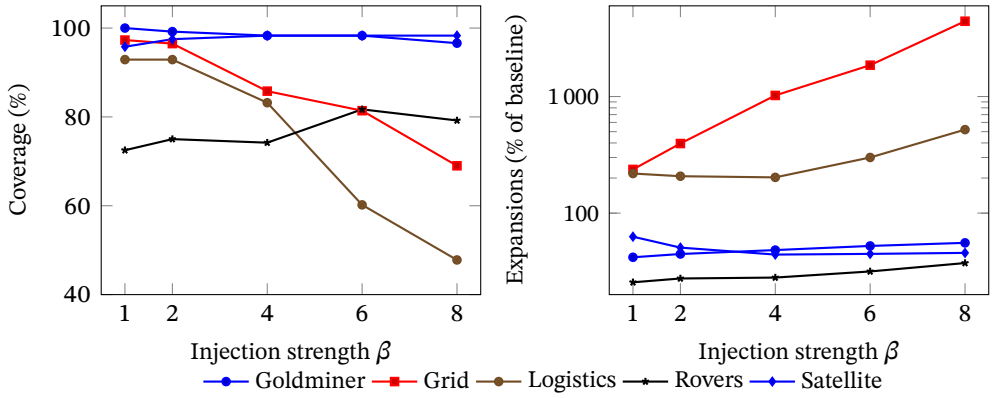


Figure 7.2: Effect of the injection strength β on the additive signal, at $\kappa = 1$. Coverage is reported over each domain's full test set, while expansions are reported relative to the corresponding QR-DQN-based AlphaZero-style MCTS baseline averaged over each domain's common solved set, on a logarithmic scale. Each domain uses the ε_p value reported in Table 7.2.

Figure 7.2 shows that increasing β helps only when the residual ordering is informative, and the signal still has enough influence over the prior. Satellite, where $\varepsilon_p = 0$, benefits most: coverage remains high while avg. expansions fall until the improvement levels off around $\beta = 4$. In Goldminer and Logistics, the large prior floor leaves less probability mass for the residual-directed adjustment to affect, while in Grid, stronger residual injection quickly becomes harmful. Rovers shows an intermediate case, gaining coverage up to $\beta = 6$ with a moderate increase in avg. expansions.

7.4.4 Multiplicative

Table 7.3: Multiplicative signal on the bounded out-of-distribution set. The ε_p value for each domain is chosen to match the best flattening weight w from Section 7.2. The full parameter sweep is reported in Table A.3. Controls use matched ε_p, β, and κ values.

Domain	Configuration	Coverage	Expansions	Plan length	Quality
Goldminer ε=0.90 (Common set n=88) opt. 11.78	Baseline	88/118 (74.6%)	439 (100.0%)	22.57	1.915
	Signal (κ=1)	**118/118 (100.0%)**	254 (57.9%)	12.24	1.039
	Shuffled control	**118/118 (100.0%)**	188 (42.8%)	**12.17**	**1.033**
	Random control	**118/118 (100.0%)**	**182 (41.5%)**	12.22	1.037
Grid ε=0.40 (Common set n=110) opt. 12.85	Baseline	**113/113 (100.0%)**	**40 (100.0%)**	23.48	1.827
	Signal (κ=0)	111/113 (98.2%)	108 (272.7%)	17.45	1.358
	Shuffled control	111/113 (98.2%)	126 (317.7%)	17.88	1.391
	Random control	**113/113 (100.0%)**	110 (276.3%)	**17.07**	**1.328**
Logistics ε=0.80 (Common set n=104) opt. 12.75	Baseline	**112/113 (99.1%)**	208 (100.0%)	97.99	7.686
	Signal (κ=0)	104/113 (92.0%)	534 (257.0%)	**20.10**	**1.576**
	Shuffled control	111/113 (98.2%)	**134 (64.7%)**	22.45	1.761
	Random control	109/113 (96.5%)	153 (73.6%)	21.55	1.690
Rovers ε=0.60 (Common set n=90) opt. 11.74	Baseline	**109/120 (90.8%)**	356 (100.0%)	43.40	3.724
	Signal (κ=0)	104/120 (86.7%)	**143 (40.0%)**	**28.66**	**2.430**
	Shuffled control	103/120 (85.8%)	248 (69.7%)	33.14	2.824
	Random control	104/120 (86.7%)	200 (56.2%)	30.94	2.634
Satellite ε=0 (Common set n=106) opt. 10.66	Baseline	**120/120 (100.0%)**	56 (100.0%)	10.92	1.003
	Signal (κ=0)	110/120 (91.7%)	**53 (94.1%)**	10.92	1.003
	Shuffled control	106/120 (88.3%)	56 (99.5%)	10.92	1.002
	Random control	108/120 (90.0%)	55 (98.3%)	**10.89**	**1.000**

The multiplicative form shows the same domain-level pattern as the additive form, which further supports the conclusion that the main effect comes from the Bellman signal rather than from whether the prior is flattened. In Goldminer, the signal and both controls again perform similarly, and none improves on the baseline. Grid again shows no improvement and little separation between the signal and the controls. In Logistics, the controls remain better in both coverage and avg. expansions, while the signal returns the shortest plans. Rovers again gives the signal the lowest avg. expansions and shortest plans among the modified configurations. Satellite also shows the similar improvement to the additive approach, where the signal gets the lowest avg. expansions among the three modified configurations.

The main difference is the magnitude. With the exploration bonus held at the same setting, the multiplicative form is weaker than the additive form in most domains. This follows from how the two variants modify the prior. The multiplicative form scales each action according to

both its existing prior probability and its residual, whereas the additive form can increase an action's probability independently of how much prior mass it originally received.

When the learned policy is peaked, the multiplicative form therefore changes the action preferences more conservatively. This limits how much it can improve the search, but also makes it less likely to disrupt a prior that already supports good coverage. The additive form can move substantially more probability toward actions the policy initially considered unlikely, giving it greater potential to reduce expansions, but also a greater risk of reducing coverage when that adjustment is harmful. Even in Satellite, where the learned prior is already useful, the additive variant still achieves the better result.

7.4.5 Summary of Out-of-Distribution Bounded Bellman Results

Overall, these results answer the question posed at the start of the chapter. In three of the five domains, the Bellman residual provides useful information about action selection: the action with the largest residual is chosen between roughly two and a half and four and a half times as often as chance. Using the per-action ordering given by the residual e_a, the signal concentrates the search on fewer paths and improves efficiency in Satellite and Rovers. It also produces shorter plans in Rovers and Logistics, while Satellite shows little room for further improvement because the baseline plans are already close to optimal. In the remaining two domains, the residual ordering is not informative, not because the model is more self-consistent there, but because its local inconsistencies do not detect the more broader inconsistency of the domains.

7.5 Out-of-Distribution Unbounded Test Set

The evaluations so far were conducted on the bounded distance-to-goal test set, where searches are short enough that the cost of computing the Bellman signal is not a major limitation. The unbounded set removes this restriction. Its instances are the original test instances of Section 4.3 from which the bounded set was derived, so the goals may lie at any distance from the initial state rather than being restricted to five to twenty-two actions away. Because the residual e_a is computed for every applicable child at each expansion, the signal evaluates $K_s + 1$ states per expanded node, where K_s is the number of applicable actions. This overhead is therefore paid repeatedly as the search expands more nodes. In Rovers, for example, the baseline requires a median of 1,297 expansions on the unbounded set, compared with only 69 on the bounded set. This section examines how the computational overhead of the Bellman signal affects performance on longer searches and whether its benefits remain despite that added cost.

Table 7.4: Performance on the unbounded out-of-distribution test set. Each algorithm is evaluated on the instances it solves; no common solved set is used across algorithms.

	Satellite			Rovers			Logistics		
Algorithm	Cov.	Exp.	Plan	Cov.	Exp.	Plan	Cov.	Exp.	Plan
Value-based search									
Greedy-V	30.8%	100	100.3	14.2%	231	230.9	80.0%	**29**	29.1
A*	20.0%	843	16.5	13.3%	484	**10.6**	30.8%	976	**18.1**
$W\text{-}A^*_5$	35.8%	363	19.7	32.5%	656	26.3	83.3%	104	23.9
GBFS	24.2%	348	24.5	14.2%	2 292	23.8	90.0%	2 205	27.6
Beam_{k^*}	45.0% (80)	1 235	23.4	45.8% (40)	3 493	204.0	88.3% (200)	5 779	56.0
Q-based search									
Greedy-Q	37.5%	139	139.3	54.2%	218	217.9	85.8%	77	77.3
Q*	36.7%	5 549	16.3	14.2%	10 462	15.7	66.7%	4 314	21.9
$W\text{-}Q^*_{w^*}$	38.3% (4)	1 506	19.6	25.8% (4)	926	36.4	83.3% (8)	49	23.9
Tree-based search									
Greedy-SAC	45.8%	96	96.3	42.5%	**137**	136.7	90.0%	71	70.9
AlphaZero (benchmark)	74.2%	2 839	77.9	54.2%	1 247	62.3	92.5%	881	53.5
AlphaZero (flattening)	**100.0%**	50	30.2	**84.2%**	4 376	95.5	**95.0%**	2 315	48.1
AlphaZero (bellman signal)	**100.0%**	**42**	**24.2**	46.7%	1 832	58.4	85.0%	401	27.4

7.5.1 Bellman Signal

Table 7.4 compares the full set of search algorithms on the unbounded out-of-distribution test set. The three AlphaZero-style rows represent successive modifications of the same search: the benchmark version uses the original learned policy prior Section 5.5., the flattening version redistributes a fixed amount of prior probability toward alternative actions, and the Bellman version uses the inconsistency signal to determine how this redistribution is applied, configuration selected from the full inconsistency-signal sweep reported in Appendix A.4. Satellite shows that the Bellman signal can improve further on prior flattening. Both variants maintain complete coverage, while the Bellman version reduces average expansions and average plan length. In Rovers and Logistics, however, the additional model evaluations required by the signal reduce coverage relative to prior flattening, from 84.2% to 46.7% and from 95.0% to 85.0%, respectively. The computational reason for this drop is examined in Section 7.5.2.

These results can also be compared directly with previous work, where learned heuristics are commonly evaluated using GBFS [3, 4, 6, 13]. On the same unbounded test set, GBFS reaches only 24.2% coverage in Satellite and 14.2% in Rovers, while both modified AlphaZero variants remain substantially higher even with the computational cost of the Bellman signal. In Logistics, prior flattening also improves coverage over GBFS, reaching 95.0% compared with 90.0%, while the Bellman variant reaches 85.0%. These results show that the choice of search algorithm can strongly affect the performance of imperfect learned guidance.

Table 7.5: Computational cost of the Bellman signal on the bounded and unbounded test sets. Baseline expansions show how much search each domain requires, while QR-DQN calls, simulation speed, and time-outs show how the additional residual evaluations affect the fixed time budget. Bounded runs were executed on CPU and unbounded runs on GPU, so speeds are comparable within a set but not between sets.

Domain	Set	Baseline exp.	QR-DQN calls	Baseline sim./s	Signal sim./s	Signal time-outs
Satellite	bounded	32	32	10.9	0.40	2/120
	unbounded	52	73	18.8	3.3	**0/120**
Logistics	bounded	59	106	41.6	9.5	8/113
	unbounded	106	128	27.6	9.7	18/120
Rovers	bounded	69	116	6.0	1.1	9/120
	unbounded	3 267	2 719	11.8	3.3	**70/120**

7.5.2 The Cost of the Residual Computation

Table 7.5 reports the computational cost of the Bellman signal on the bounded and unbounded test sets. The baseline expansion column shows how much search each domain requires, the QR-DQN column reports the number of model calls made by the signal per instance, and the rate columns show how quickly the baseline and signal complete simulations. Simulation rates were measured by running both configurations consecutively on the same node with a fixed simulation budget. Because the bounded runs were executed on CPU and the unbounded runs on GPU, rates are comparable within each set but not across sets. Expansions and model calls are medians over solved instances.

The number of per-expansion calls is fixed and does not depend on the branching factor. At each expansion, the signal requires the parent's quantile estimates and estimates for every applicable successor. One model call evaluates the parent, and a second evaluates all uncached successors in a single batch. Across all domains and both test sets, the measured ratio is exactly 2.00 model calls per expansion. A larger branching factor therefore increases the size of the second batch rather than the number of calls.

However, fixing the number of calls does not fix the cost of an expansion. The network processes every state in the batch, so a larger branching factor makes the second call proportionally more expensive: the work per expansion grows with the number and size of the successors even though the call count does not. This is what the simulation rates show. In bounded Satellite, the rate falls from 10.9 simulations per second for the baseline to 0.40 for the Bellman signal, a factor of 27, compared with factors of 4.4 in Logistics and 5.3 in Rovers. On the unbounded set, the corresponding factors are 5.7, 2.9, and 3.6. These differences follow the median number of applicable actions per state, which is about 239 in bounded Satellite and 93 on its unbounded instances, compared with 16–29 in Logistics and Rovers. Satellite therefore has the highest cost per expansion despite being the domain with the highest coverage.

The deciding factor is therefore how many times this additional cost is paid before a solution is found. On the unbounded set, the baseline needs a median of only 52 expansions in Satellite and 106 in Logistics, compared with 3,267 in Rovers. Relative to plan length, this corresponds to 2.5, 3.7, and 47.0 expansions per plan step, respectively. At the measured signal rates, the 1800-second budget permits roughly 5,900 expansions in Satellite, 17,500 in Logistics, and 5,900 in Rovers. This is about 110, 165, and only 1.8 times the baseline's median number of expansions needed to solve an instance in each domain.

These differences explain the time-out pattern. Satellite records no time-outs and retains complete coverage because a typical 52-expansion search remains far below the budget even at the lower simulation rate. Logistics loses 18 of 120 instances: most searches remain inexpensive, but a smaller set of harder instances reaches the time limit because the signal completes fewer simulations per second. Rovers loses 70 of 120 because its searches already require far more expansions, leaving much less headroom for the additional cost.

On the instances that are solved, the same effects observed on the bounded set remain visible. In Rovers, the Bellman signal again reduces search effort, requiring 834 expansions compared with 2,988 for the baseline on the common solved set. In Logistics, it again improves plan quality, reducing average plan length from 40.90 to 26.96, also at the cost of more expansions. This suggests that the signal retains the benefits seen on the bounded set when a solution is found; the main problem on the unbounded set is that the added computation prevents some harder instances from being solved within the time limit.

8 Limitations & Future Work

8.1 Computational Limitations of the Bellman Signal

A central limitation of the proposed Bellman signal is the additional computation required at each expansion. Computing the residual requires evaluating the distributional model at the current state and at all applicable successors. Because the successors are evaluated together in a single forward pass, each expansion needs exactly two model calls instead of one, regardless of the branching factor. Batching therefore bounds the number of calls, and it does so most usefully where many successors are applicable, but it does not make those calls cheap. Measured with both configurations run consecutively on the same node, in satellite the signal is 5.7 times slower on the unbounded instances, against 2.9 to 5.3 in Logistics and Rovers.

The effect on coverage depends on how this per-expansion slowdown interacts with the amount of search required. On the unbounded set, the baseline requires a median of 52 expansions in Satellite and 106 in Logistics, but 3,267 in Rovers, or 2.5, 3.7, and 47.0 expansions per plan step, respectively. The comparison between unbounded Satellite and Rovers shows this most clearly: the signal runs at the same 3.3 simulations per second in both, yet Satellite records no timeouts and retains complete coverage, while Rovers times out on 70 of 120 instances. The slowdown therefore becomes limiting only when it is repeated over searches that already require many expansions.

8.2 Limitations and Extensions of the Bellman Signal

The residual compares the model with its own one-step Bellman target, so it measures disagreement between neighbouring predictions rather than error relative to the true cost-to-go. A model whose errors change smoothly across a state's successors can therefore appear self-consistent even when its predictions are wrong. Chapter 7 shows this limitation directly: in Goldminer and Grid, the model is at least as inconsistent as in the domain where the signal performs best, yet there the residual identifies useful actions no better than chance. Self-consistency should therefore be interpreted only as a proxy for reliability, and the results show that this proxy is informative in some domains but not in others.

One way to address this is to compare independently trained models rather than relying only on a model's consistency with itself. Multiple independently trained models could provide an uncertainty estimate from the spread of their predictions at a state, allowing errors that are reproduced consistently within a single model to become visible when different models

disagree. This would measure a different aspect of reliability than the Bellman residual and could be especially useful in Goldminer and Grid, where the residual provides no useful action ordering even though the model may still be uncertain. The main drawback is computational cost, since several models would need to be trained and evaluated instead of one.

A cheaper alternative is to use another signal already available from the same distributional model. Because the model predicts a full return distribution for each action, the width of that distribution can be used as an indication of how inconsistent the prediction is, regardless of whether it agrees with the one-step Bellman target or not. A narrow distribution corresponds to a more concentrated prediction, while a wider distribution indicates greater inconsistency. The distribution width could therefore be used instead of the residual or combined with it. Unlike using multiple independently trained models, this would require no additional model evaluations because the quantiles are already computed by the distributional model.

A further extension is to move beyond the current one-step comparison. The Bellman residual currently is being computed over a single transition, comparing a state's prediction with the prediction at its immediate successor. A multi-step version could instead measure whether the model's predictions remain consistent over an entire short trajectory. Chapter 7 suggests that this may matter because the residuals are correlated across successive states. A signal based on several steps of lookahead could measure this longer-range consistency directly, rather than relying on a sequence of separate one-step residuals.

8.3 Experimental Design

The averages for expansions, plan length, and quality are computed only over the instances solved by every arm in a comparison block. This can favor methods with lower coverage, because their averages are then based only on the easier instances they managed to solve. A similar limitation is with the quality metric. Plan quality can be reported only where optimal costs are known. On the bounded set, these are exact, since the instances are cut from optimal plans at a known remaining distance, but no optimal plans exist for the unbounded set, so the results of Section 7.5 compare plan lengths between arms without a reference point and cannot say how far from optimal any of them are.

The shuffle and random control results in Section 7.3.1 suggest that the state-level part of the Bellman signal was ineffective. The shuffled control preserves the measured state inconsistency g_s, while the random control breaks this state-wise information, yet the two give very similar results across domains. One likely reason is that taking the average residual across all successor actions may not reflect how reliable the state itself is. The averaging may hide important differences between states, even when the action-level residuals themselves are informative. Future work should therefore investigate alternative state-level signals that more directly

capture when the model's guidance at a state is unreliable and whether additional exploration would help resolve that inconsistency.

The exploration constant was fixed at $c_{\text{puct}} = 1.5$ throughout the reported experiments. A separate Rovers run with $c_{\text{puct}} = 0.75$ solved between forty-two and fifty-seven more instances across the tested configurations than the corresponding runs at 1.5. Future work should therefore evaluate a wider range of exploration bonus values and investigate whether it can instead be adapted per action, similar to how the prior is adjusted using the Bellman signal.

Barman is absent from every experiment in Chapter 7. Its policy never learned to compose the skills the test goals require, for the reasons given in Section 4.1, and its distributional model did not train either. Both failures could not be identified: the run completed without diverging and produced a model, but its predictions were unusable and did not reveal any clear cause for the failure. Barman is therefore the one domain of the six benchmarked in Chapter 5 for which nothing in the second half of this thesis could be evaluated on, and whether the modifications studied here would help or harm in a domain of its structure remains unknown.

8.4 Extending to Other Search Algorithms

In this thesis, we applied the Bellman signal only to the tree policy of the AlphaZero-style search because this algorithm has an explicit action prior that we can modify without changing the search's accumulated statistics. The other algorithms from Chapter 3 lack such a prior, but they still include components whose dependence on the learned model can be adjusted locally. In Beam Search, for example, the beam width could vary by state, retaining more successors where the model is inconsistent instead of using the same fixed width everywhere. This is particularly relevant because the benchmark in Section 5 shows that Beam Search has by far the highest average number of expansions among the evaluated algorithms. Likewise, Weighted A* and Weighted Q* could use heuristic weights that vary by state and action, respectively. In both cases, the residual could reduce the influence of the learned estimate where the model appears less reliable. Whether the Bellman signal is informative enough to support these adaptations remains an open question.

9 Conclusion

In this thesis, we examined whether neural-guided search can compensate for failures in greedy execution of learned guidance in classical planning, and whether modifying how that guidance is incorporated can further improve performance.

The first research question asked which search algorithms recover best from greedy failures and at what search cost. Beam Search, Weighted Q*, and AlphaZero-style MCTS are the strongest methods overall. AlphaZero improves coverage over its greedy execution in all five available domains and improves both coverage and plan length in four; in Satellite, coverage rises from 45.8% to 74.2%, while plan length falls from 63.4 to 23.7. Prior work commonly evaluates learned heuristics using GBFS [3, 4, 6, 13], yet across six domains GBFS achieves only 29.9% average coverage, compared with 76.0% for the best of Beam Search, Weighted Q*, and AlphaZero. In most domains, Weighted Q* and AlphaZero achieve these results with much fewer expansions and shorter plan lengths than GBFS.

The second research question asked how the strongest approach could be improved by adapting its use of imperfect guidance. For AlphaZero-style MCTS, we investigated two approaches. The first was prior flattening, which reduces reliance on the learned policy by mixing its prior with a uniform distribution. Its effect was strongly domain-dependent, ranging from large coverage gains in Goldminer to severe degradation when the learned prior was removed in Satellite.

Second, we used the Bellman inconsistency signal to target where the prior should be adjusted. Its main limitation is computational cost due to repeated QR-DQN calls. On the unbounded set, throughput falls from 18.2 to 6.0 simulations per second in Rovers and from 19.2 to 8.9 in Logistics, producing 70 and 18 timeouts. Satellite is less affected because its searches require fewer expansions. There, the Bellman variant maintains complete coverage and reduces expansions further from 50 to 42 and plan length from 30.2 to 24.2 relative to prior flattening results.

Despite this overhead, the Bellman signal still improves over simpler uses of the learned models. In Satellite, it reaches complete coverage compared with 45.8% and 24.2% for Greedy-SAC and GBFS, respectively, while using 56% and 88% fewer expansions. In Rovers, it reaches 46.7% coverage despite the timeouts, compared with 42.5% for Greedy-SAC and 14.2% for GBFS, while also improving on them in plan length and search effort. In Logistics, coverage falls to 85.0%, compared with 90.0% for both Greedy-SAC and GBFS due to the timeouts, but it produces much shorter plans than Greedy-SAC (27.4 vs. 70.9) and far fewer expansions than GBFS (401 vs. 2,205).

A Appendix

Table A.1: Prior-flattening results on the bounded out-of-distribution set for all tested weights w. Coverage is reported over the full test set; expansions, plan length, and quality use each domain's common solved set.

Domain	Weight	Coverage	Expansions	Plan length	Quality
common set $n = 88$, mean optimal length 11.78					
Goldminer	$w = 0$	88/118 (74.6%)	439 (100.0%)	22.57	1.915
	$w = 0.10$	117/118 (99.2%)	398 (90.8%)	20.86	1.770
	$w = 0.20$	118/118 (100.0%)	421 (96.0%)	18.60	1.579
	$w = 0.26$	118/118 (100.0%)	448 (102.1%)	16.93	1.437
	$w = 0.40$	118/118 (100.0%)	494 (112.6%)	15.59	1.323
	$w = 0.42$	118/118 (100.0%)	428 (97.4%)	15.59	1.323
	$w = 0.60$	118/118 (100.0%)	510 (116.2%)	13.19	1.120
	$w = 0.80$	118/118 (100.0%)	330 (75.1%)	**12.12**	**1.029**
	$w = 0.90$	118/118 (100.0%)	246 (56.1%)	12.17	1.033
	$w = 1.00$	118/118 (100.0%)	**161** **(36.8%)**	12.51	1.062
common set $n = 78$, mean optimal length 10.71					
Grid	$w = 0$	113/113 (100.0%)	**28** **(100.0%)**	17.24	1.611
	$w = 0.10$	113/113 (100.0%)	29 (103.7%)	16.44	1.535
	$w = 0.27$	113/113 (100.0%)	36 (128.3%)	14.51	1.356
	$w = 0.40$	113/113 (100.0%)	37 (132.1%)	13.71	1.280
	$w = 0.60$	113/113 (100.0%)	54 (190.8%)	**12.76**	**1.192**
	$w = 0.80$	111/113 (98.2%)	170 (600.5%)	12.81	1.196
	$w = 0.90$	108/113 (95.6%)	344 (1214.0%)	12.96	1.211
	$w = 1.00$	79/113 (69.9%)	4262 (15040.8%)	14.62	1.365

Continued on next page

Table A.1 – continued

Domain	Weight	Coverage	Expansions		Plan length	Quality
common set $n = 110$, mean optimal length 13.10						
Logistics	$w = 0$	112/113 (99.1%)	**330**	**(100.0%)**	118.07	9.013
	$w = 0.10$	111/113 (98.2%)	494	(149.5%)	104.73	7.994
	$w = 0.20$	111/113 (98.2%)	559	(169.2%)	119.54	9.125
	$w = 0.30$	111/113 (98.2%)	579	(175.3%)	89.70	6.847
	$w = 0.40$	112/113 (99.1%)	538	(162.9%)	52.68	4.022
	$w = 0.60$	113/113 (100.0%)	480	(145.2%)	29.34	2.239
	$w = 0.80$	113/113 (100.0%)	473	(143.1%)	**25.65**	**1.958**
	$w = 0.90$	113/113 (100.0%)	514	(155.4%)	26.16	1.997
	$w = 1.00$	0/113 (0.0%)	—		—	—
common set $n = 105$, mean optimal length 13.01						
Rovers	$w = 0$	109/120 (90.8%)	503	(100.0%)	50.10	3.851
	$w = 0.27$	112/120 (93.3%)	409	(81.3%)	40.88	3.142
	$w = 0.40$	112/120 (93.3%)	382	(75.8%)	**40.68**	**3.127**
	$w = 0.60$	114/120 (95.0%)	264	(52.5%)	44.11	3.391
	$w = 0.80$	**116/120 (96.7%)**	**262**	**(52.0%)**	45.78	3.519
	$w = 0.90$	115/120 (95.8%)	265	(52.6%)	52.99	4.073
	$w = 1.00$	9/120 (7.5%)	—		—	—
common set $n = 120$, mean optimal length 13.20						
Satellite	$w = 0$	120/120 (100.0%)	**94**	**(100.0%)**	**11.62**	**0.880**
	$w = 0.10$	120/120 (100.0%)	130	(137.8%)	11.68	0.885
	$w = 0.25$	120/120 (100.0%)	120	(127.3%)	11.73	0.889
	$w = 0.40$	120/120 (100.0%)	126	(133.0%)	11.76	0.891
	$w = 0.60$	120/120 (100.0%)	134	(142.1%)	12.02	0.910
	$w = 0.80$	120/120 (100.0%)	143	(151.8%)	12.07	0.914
	$w = 0.90$	120/120 (100.0%)	145	(153.4%)	12.10	0.917
	$w = 1.00$	0/120 (0.0%)	—		—	—

Table A.2: Additive Bellman-signal results on the bounded out-of-distribution set across all tested β values, for $\kappa = 1$ and $\kappa = 0$. Coverage is reported over the full test set; expansions, plan length, and quality use each domain's common solved set.

Domain	Configuration	Coverage	Expansions	Plan length	Quality
$\varepsilon_p = 0.90$, $c_{\text{puct}} = 1.5$, common set $n = 82$					
	Baseline	88/118 (74.6%)	403 (100.0%)	20.94	1.641
	κ=1, β=1	**118/118 (100.0%)**	**168 (41.8%)**	11.62	1.027
	κ=1, β=2	117/118 (99.2%)	180 (44.6%)	11.60	1.023
	κ=1, β=4	116/118 (98.3%)	194 (48.2%)	11.62	1.024
	κ=1, β=6	116/118 (98.3%)	211 (52.3%)	11.45	1.015
Goldminer	κ=1, β=8	114/118 (96.6%)	224 (55.6%)	11.57	1.021
	κ=0, β=1	117/118 (99.2%)	213 (52.9%)	11.60	1.024
	κ=0, β=2	116/118 (98.3%)	272 (67.5%)	11.48	1.017
	κ=0, β=4	114/118 (96.6%)	200 (49.6%)	11.48	1.016
	κ=0, β=6	114/118 (96.6%)	226 (56.0%)	**11.43**	**1.014**
	κ=0, β=8	113/118 (95.8%)	258 (64.2%)	11.50	1.019
$\varepsilon_p = 0.40$, $c_{\text{puct}} = 1.5$, common set $n = 78$					
	Baseline	**113/113 (100.0%)**	**30 (100.0%)**	17.72	1.623
	κ=1, β=1	110/113 (97.3%)	70 (237.1%)	12.73	1.175
	κ=1, β=2	109/113 (96.5%)	117 (395.6%)	**12.33**	**1.132**
	κ=1, β=4	97/113 (85.8%)	303 (1022.1%)	12.94	1.169
	κ=1, β=6	92/113 (81.4%)	552 (1858.6%)	13.46	1.217
Grid	κ=1, β=8	78/113 (69.0%)	1 314 (4427.5%)	13.82	1.252
	κ=0, β=1	111/113 (98.2%)	51 (171.9%)	13.06	1.199
	κ=0, β=2	111/113 (98.2%)	68 (229.3%)	12.60	1.150
	κ=0, β=4	108/113 (95.6%)	122 (410.4%)	13.15	1.199
	κ=0, β=6	97/113 (85.8%)	387 (1304.5%)	13.28	1.216
	κ=0, β=8	87/113 (77.0%)	892 (3006.3%)	14.26	1.303
$\varepsilon_p = 0.80$, $c_{\text{puct}} = 1.5$, common set $n = 49$					
	Baseline	**112/113 (99.1%)**	**83 (100.0%)**	33.76	3.927
	κ=1, β=1	105/113 (92.9%)	181 (218.6%)	**14.24**	**1.516**
	κ=1, β=2	105/113 (92.9%)	172 (207.3%)	17.49	1.813
	κ=1, β=4	94/113 (83.2%)	168 (202.6%)	15.35	1.648
	κ=1, β=6	68/113 (60.2%)	249 (300.2%)	20.80	2.102
Logistics	κ=1, β=8	54/113 (47.8%)	432 (520.3%)	29.24	2.959

Continued on next page

Table A.2 – continued

Domain	Configuration	Coverage	Expansions	Plan length	Quality
	κ=0, β=1	106/113 (93.8%)	240 (288.5%)	18.63	1.909
	κ=0, β=2	103/113 (91.2%)	209 (251.8%)	18.98	1.989
	κ=0, β=4	96/113 (85.0%)	221 (266.0%)	20.57	2.254
	κ=0, β=6	68/113 (60.2%)	348 (419.6%)	25.82	2.675
	κ=0, β=8	51/113 (45.1%)	750 (904.0%)	30.73	3.111
$\varepsilon_p = 0.80$, $c_{\text{puct}} = 1.5$, common set $n = 82$					
	Baseline	109/120 (90.8%)	362 (100.0%)	44.57	3.759
	κ=1, β=1	107/120 (89.2%)	140 (38.5%)	24.00	2.218
	κ=1, β=2	103/120 (85.8%)	128 (35.4%)	23.96	2.181
	κ=1, β=4	109/120 (90.8%)	141 (39.0%)	**23.16**	**2.081**
	κ=1, β=6	105/120 (87.5%)	196 (54.1%)	23.91	2.108
Rovers	κ=1, β=8	102/120 (85.0%)	243 (67.2%)	27.49	2.481
	κ=0, β=1	110/120 (91.7%)	114 (31.4%)	28.65	2.591
	κ=0, β=2	**111/120 (92.5%)**	101 (27.9%)	27.61	2.472
	κ=0, β=4	**111/120 (92.5%)**	**101 (27.8%)**	28.10	2.489
	κ=0, β=6	**111/120 (92.5%)**	160 (44.0%)	29.02	2.537
	κ=0, β=8	108/120 (90.0%)	179 (49.3%)	30.96	2.749
$\varepsilon_p = 0$, $c_{\text{puct}} = 1.5$, common set $n = 100$					
	Baseline	**120/120 (100.0%)**	57 (100.0%)	10.85	**1.003**
	κ=1, β=1	115/120 (95.8%)	36 (62.8%)	10.85	**1.003**
	κ=1, β=2	117/120 (97.5%)	29 (50.6%)	**10.84**	**1.003**
	κ=1, β=4	118/120 (98.3%)	**25 (44.1%)**	10.86	1.004
	κ=1, β=6	118/120 (98.3%)	25 (44.7%)	10.86	1.004
Satellite	κ=1, β=8	118/120 (98.3%)	26 (45.6%)	10.89	1.007
	κ=0, β=1	118/120 (98.3%)	26 (46.1%)	**10.84**	**1.003**
	κ=0, β=2	119/120 (99.2%)	25 (44.8%)	10.86	1.004
	κ=0, β=4	118/120 (98.3%)	25 (44.7%)	10.86	1.004
	κ=0, β=6	118/120 (98.3%)	26 (45.4%)	10.89	1.007
	κ=0, β=8	118/120 (98.3%)	26 (45.1%)	10.89	1.008

Table A.3: Multiplicative Bellman-signal results on the bounded out-of-distribution set for both κ settings, with $\beta = 1$ in all configurations. Coverage is reported over the full test set; expansions, plan length, and quality use each domain's common solved set.

Domain	Configuration	Coverage	Expansions	Plan length	Quality
$\varepsilon_p = 0.90$, common set $n = 86$, mean optimal length 11.57					
	Baseline	88/118 (74.6%)	443 (100.0%)	21.94	1.896
	Signal, κ=0	117/118 (99.2%)	260 (58.6%)	11.90	1.028
	Signal, κ=1	**118/118 (100.0%)**	210 (47.3%)	11.92	1.030
Goldminer	Shuffled control, κ=0	117/118 (99.2%)	176 (39.7%)	12.29	1.062
	Random control, κ=0	117/118 (99.2%)	164 (37.1%)	12.10	1.046
	Shuffled control, κ=1	**118/118 (100.0%)**	**141 (31.8%)**	**11.87**	**1.026**
	Random control, κ=1	**118/118 (100.0%)**	146 (33.0%)	11.90	1.028
$\varepsilon_p = 0.40$, common set $n = 110$, mean optimal length 12.85					
	Baseline	**113/113 (100.0%)**	**40 (100.0%)**	23.48	1.827
	Signal, κ=0	111/113 (98.2%)	108 (272.7%)	17.45	1.358
	Signal, κ=1	111/113 (98.2%)	172 (432.1%)	16.57	1.289
Grid	Shuffled control, κ=0	111/113 (98.2%)	126 (317.7%)	17.88	1.391
	Random control, κ=0	**113/113 (100.0%)**	110 (276.3%)	17.07	1.328
	Shuffled control, κ=1	110/113 (97.3%)	172 (432.1%)	**16.49**	**1.283**
	Random control, κ=1	111/113 (98.2%)	196 (492.5%)	16.55	1.287
$\varepsilon_p = 0.80$, common set $n = 103$, mean optimal length 12.72					
	Baseline	**112/113 (99.1%)**	202 (100.0%)	91.69	7.209
	Signal, κ=0	104/113 (92.0%)	438 (217.0%)	18.99	1.493
	Signal, κ=1	105/113 (92.9%)	390 (193.0%)	17.37	1.366
Logistics	Shuffled control, κ=0	111/113 (98.2%)	**119 (58.9%)**	22.02	1.731
	Random control, κ=0	109/113 (96.5%)	124 (61.3%)	20.99	1.650
	Shuffled control, κ=1	110/113 (97.3%)	240 (118.7%)	17.96	1.412
	Random control, κ=1	109/113 (96.5%)	153 (75.7%)	**16.49**	**1.296**
$\varepsilon_p = 0.60$, common set $n = 84$, mean optimal length 11.37					
	Baseline	**109/120 (90.8%)**	293 (100.0%)	42.55	3.771
	Signal, κ=0	104/120 (86.7%)	**127 (43.4%)**	27.27	2.384
	Signal, κ=1	104/120 (86.7%)	159 (54.2%)	24.27	2.121
Rovers	Shuffled control, κ=0	103/120 (85.8%)	202 (68.9%)	30.29	2.659
	Random control, κ=0	104/120 (86.7%)	141 (48.2%)	29.56	2.595
	Shuffled control, κ=1	98/120 (81.7%)	186 (63.4%)	24.87	2.168
	Random control, κ=1	98/120 (81.7%)	192 (65.4%)	**23.49**	**2.048**

Continued on next page

Table A.3 – continued

Domain	Configuration	Coverage	Expansions	Plan length	Quality
$\varepsilon_p = 0$, common set $n = 94$, mean optimal length 10.14					
	Baseline	**120/120 (100.0%)**	55 (100.0%)	10.34	1.003
	Signal, κ=0	110/120 (91.7%)	**51 (93.0%)**	10.34	1.003
	Signal, κ=1	102/120 (85.0%)	79 (144.0%)	10.34	1.004
Satellite	Shuffled control, κ=0	106/120 (88.3%)	54 (97.9%)	10.33	1.002
	Random control, κ=0	108/120 (90.0%)	52 (93.7%)	**10.30**	**1.000**
	Shuffled control, κ=1	95/120 (79.2%)	97 (177.0%)	10.35	1.004
	Random control, κ=1	95/120 (79.2%)	106 (192.4%)	10.33	1.003

Table A.4: Additive Bellman-signal results on the unbounded out-of-distribution set across all tested β values. Coverage is reported over the full test set; expansions and plan length use each domain's common solved set.

Domain	Configuration	Coverage	Expansions	Plan length
$\varepsilon_p = 0.60$, $\kappa = 1$, $c_{\text{puct}} = 1.5$, common set $n = 52$				
	Baseline	**116/120 (96.7%)**	**117 (100.0%)**	41.79
	κ=1, β=1	102/120 (85.0%)	534 (456.1%)	23.87
Logistics	κ=1, β=2	102/120 (85.0%)	424 (362.2%)	**23.60**
	κ=1, β=4	91/120 (75.8%)	723 (618.2%)	**23.60**
	κ=1, β=8	52/120 (43.3%)	2 335 (1996.1%)	37.63
$\varepsilon_p = 0.90$, $\kappa = 1$, $c_{\text{puct}} = 1.5$, common set $n = 20$				
	Baseline	**59/120 (49.2%)**	**247 (100.0%)**	23.90
	κ=1, β=1	50/120 (41.7%)	384 (155.3%)	14.80
Rovers	κ=1, β=2	43/120 (35.8%)	584 (236.2%)	**14.70**
	κ=1, β=4	28/120 (23.3%)	892 (360.8%)	16.15
	κ=1, β=6	23/120 (19.2%)	1 412 (571.1%)	19.20
	κ=1, β=8	20/120 (16.7%)	1 955 (791.1%)	19.85
$\varepsilon_p = 0.90$, $\kappa = 1$, $c_{\text{puct}} = 1.5$, common set $n = 115$				
	Baseline	**120/120 (100.0%)**	89 (100.0%)	26.77
	κ=1, β=1	**120/120 (100.0%)**	41 (46.4%)	24.29
	κ=1, β=2	**120/120 (100.0%)**	**41 (46.0%)**	**24.02**
Satellite	κ=1, β=4	**120/120 (100.0%)**	43 (48.3%)	24.65
	κ=1, β=6	116/120 (96.7%)	53 (59.9%)	26.17
	κ=1, β=8	115/120 (95.8%)	122 (136.7%)	30.44

List of Acronyms

DQN	Deep Q-Network
GBFS	Greedy Best-First Search
IQN	Implicit Quantile Network
MCTS	Monte Carlo Tree Search
PDDL	Planning Domain Definition Language
QR-DQN	Quantile Regression Deep Q-Network
SAC	Soft Actor-Critic
UCT	Upper Confidence Bound for Trees

List of Symbols

Classical Planning

s	a state
s'	the successor of s under an action
$A(s)$	set of actions applicable in s
K_s	number of actions applicable in s, $K_s = \|A(s)\|$
$f(a, s)$	state resulting from applying action a in state s
$d^*(s)$	optimal distance from s to a goal state
C	cost of a returned plan
C^*	cost of an optimal plan

Heuristic Search

$g(s)$	accumulated cost of the path from the initial state to s
$h(s)$	heuristic estimate of the remaining cost from s
$f(s)$	evaluation function, $f(s) = g(s) + h(s)$
w	heuristic weight in Weighted A* and Weighted Q*; flattening weight in Chapter 7
k	beam width in Beam Search

Learned Models

$V(s)$	learned state value function
$Q(s, a)$	learned action value function
$\pi(a \mid s)$	learned policy
$Z(s, a)$	learned return distribution for the pair (s, a)
$z_i(s, a)$	value predicted at the i-th quantile fraction
τ_i	i-th quantile fraction
M	number of quantiles representing Z
r	reward received for applying an action
γ	discount factor

Tree Search

$N(s)$	number of simulations that have passed through s
$N(s, a)$	number of times action a has been selected in s
$P(a \mid s)$	policy prior over the actions applicable in s
$P_0(a)$	prior after flattening
$P_w(a)$	prior flattened by weight w, $P_w = (1 - w)P + w/K_s$

$P_+(a)$	prior after the additive injection
$P_\times(a)$	prior after the multiplicative injection
$Q_{\text{norm}}(s,a)$	value estimate normalised against the range observed in the search
c_0	exploration constant
$c(s)$	state-dependent exploration constant, $c(s) = c_0(1 + \varkappa g_s)$

Bellman Inconsistency Signal

Y_a	distributional Bellman target for the pair (s,a)
e_a	Bellman residual, $W_1(Z(s,a), Y_a)$
W_1	1-Wasserstein distance
x_a	scaled residual, $e_a/\big(e_a + \tau_{\text{step}}\big)$
g_s	mean of x_a over the actions applicable in s
τ_{step}	residual at which the scaled signal reaches one half
β	strength of the injection
$\varkappa$	strength of the exploration channel
ε_p	prior floor
W	share of the prior displaced by the additive injection
$q(a)$	distribution receiving the displaced prior mass

Evaluation

n	size of the common set of instances a table is averaged over

List of Figures

7.1 Effect of flattening the policy prior. Coverage is measured over each domain's full test set, while expansions are reported relative to the unflattened baseline on the common solved set, using a logarithmic scale. For Logistics, Rovers, and Satellite, too few instances are solved at $w = 1$ to define the expansion ratio. . . 37

7.2 Effect of the injection strength β on the additive signal, at $\kappa = 1$. Coverage is reported over each domain's full test set, while expansions are reported relative to the corresponding QR-DQN-based AlphaZero-style MCTS baseline averaged over each domain's common solved set, on a logarithmic scale. Each domain uses the ε_p value reported in Table 7.2. 41

List of Tables

4.1 Training and validation splits, by object count. 17
4.2 The two test sets used in this thesis, defined by object count and distance to the goal. The unbounded out-of-distribution set is the standard test split for each domain. The bounded-distance set keeps the same object sizes but restricts the remaining distance to the goal to $d \in \{5, \dots, 22\}$, matching the range seen during training. 18

5.1 Greedy Execution Algorithms. **Avg. Expanded** and **Avg. Solution Length** are computed over the *common set* **n** of instances solved by every algorithm in a certain domain 20
5.2 Value-Based Search. **Avg. Expanded** and **Avg. Solution Length** are computed over the common set **n** of instances solved by every algorithm in a certain domain 22
5.3 Q-Based Search. **Avg. Expanded** and **Avg. Solution Length** are computed over the common set **n** of instances solved by every algorithm in a certain domain 24
5.4 Tree-Based Search. **Avg. Expanded** and **Avg. Solution Length** are computed over the common set **n** of instances solved by every algorithm in a certain domain 25
5.5 Performance on the unbounded out-of-distribution test set. Each algorithm is evaluated on the instances it solves; no common solved set is used across algorithms. 27
5.6 Best Search algorithms from the benchmark compared on matched instances. For each domain, **Avg. Expanded** and **Avg. Solution Length** are computed over that domain's common set **n**. Weighted Q^* and Beam Search are each shown at their best-performing setting for that domain, given in the row label. 28

7.1 Best prior-flattening configuration for each domain. The baseline corresponds to the original unflattened prior ($w = 0$). Coverage is reported for both the baseline and the selected weight, while expansions and plan length are shown relative to the baseline on the common solved set. Full results across all tested w values are reported in Appendix A.1. 37
7.2 Additive Bellman signal and matched controls on the bounded out-of-distribution set. Full sweeps over β and κ are reported in Appendix A.2. 38

7.3 Multiplicative signal on the bounded out-of-distribution set. The ε_p value for each domain is chosen to match the best flattening weight w from Section 7.2. The full parameter sweep is reported in Table A.3. Controls use matched ε_p, β, and κ values. 42
7.4 Performance on the unbounded out-of-distribution test set. Each algorithm is evaluated on the instances it solves; no common solved set is used across algorithms. 44
7.5 Computational cost of the Bellman signal on the bounded and unbounded test sets. Baseline expansions show how much search each domain requires, while QR-DQN calls, simulation speed, and time-outs show how the additional residual evaluations affect the fixed time budget. Bounded runs were executed on CPU and unbounded runs on GPU, so speeds are comparable within a set but not between sets. 45

A.1 Prior-flattening results on the bounded out-of-distribution set for all tested weights w. Coverage is reported over the full test set; expansions, plan length, and quality use each domain's common solved set. 51
A.2 Additive Bellman-signal results on the bounded out-of-distribution set across all tested β values, for $\kappa = 1$ and $\kappa = 0$. Coverage is reported over the full test set; expansions, plan length, and quality use each domain's common solved set. 53
A.3 Multiplicative Bellman-signal results on the bounded out-of-distribution set for both κ settings, with $\beta = 1$ in all configurations. Coverage is reported over the full test set; expansions, plan length, and quality use each domain's common solved set. 55
A.4 Additive Bellman-signal results on the unbounded out-of-distribution set across all tested β values. Coverage is reported over the full test set; expansions and plan length use each domain's common solved set. 56

List of References

[1] R. V. Bettker, P. P. Minini, A. G. Pereira, and M. Ritt, "Understanding sample generation strategies for learning heuristic functions in classical planning," *Journal of Artificial Intelligence Research*, vol. 80, pp. 243–271, 2024.

[2] T. Bylander, "The computational complexity of propositional strips planning," *Artificial Intelligence*, vol. 69, no. 1, pp. 165–204, 1994, ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(94)90081-7 [Online]. Available: https://www.sciencedirect.com/science/article/pii/0004370294900817

[3] D. Z. Chen, S. Thiébaux, and F. Trevizan, "Learning domain-independent heuristics for grounded and lifted planning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, 2024, pp. 20 078–20 086.

[4] D. Z. Chen, F. Trevizan, and S. Thiébaux, "Return to tradition: Learning reliable heuristics with classical machine learning," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 34, 2024, pp. 68–76.

[5] W. Dabney, M. Rowland, M. G. Bellemare, and R. Munos, "Distributional reinforcement learning with quantile regression," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 2018.

[6] P. Ferber, F. Geißer, F. Trevizan, M. Helmert, and J. Hoffmann, "Neural network heuristic functions for classical planning: Bootstrapping and comparison to other methods," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 32, 2022, pp. 583–587.

[7] P. Ferber, M. Helmert, and J. Hoffmann, "Neural network heuristics for classical planning: A study of hyperparameter space.," in *ECAI*, 2020, pp. 2346–2353.

[8] H. Geffner and B. Bonet, *A concise introduction to models and methods for automated planning*. Morgan & Claypool Publishers, 2013.

[9] P. Haslum, N. Lipovetzky, D. Magazzeni, C. Muise, R. Brachman, F. Rossi, and P. Stone, *An introduction to the planning domain definition language*. Springer, 2019, vol. 13.

[10] R. Karia and S. Srivastava, "Learning generalized relational heuristic networks for model-agnostic planning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, 2021, pp. 8064–8073.

[11] C. Núñez-Molina, M. Asai, P. Mesejo, and J. Fernández-Olivares, "On using admissible bounds for learning forward search heuristics," *arXiv preprint arXiv:2308.11905*, 2023.

[12] T. Schulte and T. Keller, "Balancing exploration and exploitation in classical planning," in *Proceedings of the International Symposium on Combinatorial Search*, vol. 5, 2014, pp. 139–147.

[13] W. Shen, F. Trevizan, and S. Thiébaux, "Learning domain-independent planning heuristics with hypergraph networks," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 30, 2020, pp. 574–584.

[14] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al., "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[15] S. Ståhlberg, B. Bonet, and H. Geffner, "Learning general optimal policies with graph neural networks: Expressive power, transparency, and limits," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 32, 2022, pp. 629–637.

[16] S. Ståhlberg, B. Bonet, and H. Geffner, "Learning generalized policies without supervision using gnns," *arXiv preprint arXiv:2205.06002*, 2022.

[17] M. Straka, *Npfl139, lecture 11 – ucb, monte carlo tree search, alphazero*, Lecture slides, Apr. 2024. [Online]. Available: `https://ufal.mff.cuni.cz/~straka/courses/npfl139/2324/slides.pdf/npfl139-2324-11.pdf`

[18] M. Świechowski, K. Godlewski, B. Sawicki, and J. Mańdziuk, "Monte carlo tree search: A review of recent modifications and applications," *Artificial Intelligence Review*, vol. 56, no. 3, pp. 2497–2562, 2023.

[19] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3, pp. 279–292, 1992.