Data Efficient Training of Heuristic Functions for Satisficing Search

Forest Agostinelli[1], **Shahaf S. Shperberg**[2],

[1]Department of Computer Science and Engineering, University of South Carolina
[2]Faculty of Computer and Information Science, Ben-Gurion University of the Negev
foresta@cse.sc.edu, shperbsh@bgu.ac.il

Abstract

While heuristic functions trained with deep reinforcement learning have proven capable of solving problems ranging from puzzles to quantum computing, the data-hungry training process reduces the practicality of training heuristic functions for those without access to high-end hardware or who cannot wait days for training to complete. To address this problem, we carry out an empirical study focusing on satisficing search to find general guidelines to reduce the data required to train heuristic functions without sacrificing coverage. This study shows that straightforward techniques, including frequent target network updates, a novel technique to balance training data difficulty, and reusing previously generated problem instances, significantly improve data efficiency. Using the aforementioned techniques, we reduce the amount of training data needed by up to two orders of magnitude compared to previous methods and can reach 100% coverage on domains such as the Rubik's cube in 4 hours of training using a single GPU or 20 hours of training without a GPU on a consumer laptop while previous methods required multiple days of training and multiple GPUs.

Introduction

Deep reinforcement learning (Sutton and Barto 2018) (DRL) has been shown to be able to train domain-specific heuristic functions represented as deep neural networks (DNNs) (Schmidhuber 2015; LeCun, Bengio, and Hinton 2015) that, when combined with heuristic search, are capable of solving pathfinding problems such as solving the Rubik's cube (Agostinelli et al. 2019b), synthesizing quantum circuits (Zhang et al. 2020; Chen et al. 2024; Bao and Hartnett 2024; Turner, Agostinelli, and Fu 2025), parking lot optimization (Siddique, Gue, and Usher 2021), and chemical reaction mechanism pathfinding (Panta et al. 2024). While these domain-specific heuristic functions are trained in a largely domain-independent fashion, training often requires generating billions of problem instances and takes several days to complete, even when using multiple high-end graphics processing units (GPUs). Furthermore, it is often the case that solving problem instances with these learned heuristic functions is also resource intensive (Agostinelli et al. 2019b; Chervov et al. 2025).

Copyright © 2026, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

We aim to maximize coverage (i.e. the number of problem instances solved) using a learned heuristic function within a given budget of search iterations, while minimizing the amount of data required to train the heuristic function. To accomplish this, we carry out an empirical study to find general guidelines for reducing the data required to train heuristic functions without sacrificing coverage. We focus our study on satisficing search (i.e., search that is concerned with finding a feasible path rather than minimizing path cost) to push the limits of data efficiency without being restrained by path cost. We find that we can improve coverage with orders of magnitude less data compared to previous methods and solve problems using greedy best-first search with a maximum of 1,000 iterations without batching.

Background

Pathfinding

A pathfinding domain is defined by a state space, a set of actions $\mathcal{A}$, a transition function T, and a transition cost function c. The transition function is represented by T, where $s' = T(s, a)$ is the state resulting from taking action a in state s. The transition cost function is represented by c, where $c(s, a)$ is the associated transition cost. A pathfinding problem instance can be represented as a weighted directed graph (Pohl 1970), where nodes represent states and edges represent actions. An instance is defined by a tuple (D, s_0, g), where D is the pathfinding domain, s_0 is the start state, and g is a set of states considered goal states. Given a pathfinding problem, the objective is to find a path, which is a sequence of actions, that transforms the start state into a state in the goal set g. Given a path, the path cost is the sum of transition costs. The cost-to-go of a given state, s, with respect to goal g, is the cost of a shortest path from s to g.

Reinforcement Learning of Heuristic Functions

Heuristic search algorithms used to solve pathfinding problems, such as A* search (Hart, Nilsson, and Raphael 1968), weighted A* search (Pohl 1970), and greedy best-first search (GBFS) (Doran and Michie 1966), are guided by a heuristic function that maps a state, s, and a goal, g, to an estimate of the cost-to-go. It has been shown that domain-specific heuristic functions can be represented with deep neural networks (DNNs) and trained with deep reinforcement learning

in a largely domain-independent manner (Agostinelli et al. 2019b). Specifically, heuristic functions are trained using approximate value iteration (Bellman 1957; Bertsekas and Tsitsiklis 1996) which trains a DNN to approximate a Bellman update, $h'(s, g)$, for each state and goal tuple, (s, g), in a given batch of states and goals. The loss function in Equation 1 is minimized with gradient descent, where N is the batch size and θ are the parameters of the DNN. The Bellman update in the context of pathfinding is shown in Equation 2, where θ^- are the parameters of the target network (Mnih et al. 2015) which are periodically updated to θ.

$$L(\theta) = \frac{1}{N} \sum_{i}^{N} (h'(s_i, g_i) - h_\theta(s_i, g_i))^2 \tag{1}$$

$$h'(s, g) = \begin{cases} 0, & \text{if } s \in g, \\ \min_{a \in \mathcal{A}} c(s, a) + h_{\theta^-}(T(s, a), g), & \text{otherwise.} \end{cases} \tag{2}$$

Related Work

The pursuit of learning heuristic functions has a rich history preceding the deep RL methods described in the Background section. One approach is imitation learning, where cost-to-go values derived from domain knowledge or solvers are used to train heuristics via supervised learning. For example, Samadi, Felner, and Schaeffer (2008) reduced memory usage by training neural networks to approximate Pattern Database (PDB) heuristics. Other studies have designed architectures for heuristic learning in planning problems (Chrestien et al. 2021; Takahashi et al. 2019; Shen, Trevizan, and Thiébaux 2020; Ferber, Helmert, and Hoffmann 2020; Toyer et al. 2020), often proposing alternative loss functions such as learning to rank states to improve search efficiency (Garrett, Kaelbling, and Lozano-Pérez 2016; Bhardwaj, Choudhury, and Scherer 2017; Groshev et al. 2018; Chrestien et al. 2023).

Recent research has also explored leveraging large language models (LLMs) for automated heuristic discovery. For example, Ling et al. (2025) introduced a method to derive heuristic functions for planning tasks, while Corrêa, Pereira, and Seipp (2025) showed that LLM-generated heuristics can rival classical ones. However, these approaches often rely on encoding domain-specific knowledge within prompts, and their scalability to high-dimensional state spaces remains limited compared to reinforcement learning methods.

Another approach iteratively refines heuristics based on the costs of paths found during search. Bramanti-Gregor and Davis (1993) iteratively refined heuristics with A* and trained new heuristics via linear regression. Fink (2007) learned a weighted sum of admissible heuristics, while Arfaee, Zilles, and Holte (2011) used neural networks and random walks to generate easier instances when no problems were solved. Orseau and Lelis (2021) extended this by learning both a policy and a heuristic. A major limitation of these methods is their inability to learn from expanded nodes that do not contribute to a solution, leading to poor sample efficiency.

While modern approaches leveraging reinforcement learning achieved breakthroughs in scalability and solution quality without domain-specific heuristic information (Agostinelli et al. 2019b; Chen et al. 2024; Panta et al. 2024), they introduced a massive data burden, often requiring billions of training instances. Recent supervised learning approaches have demonstrated remarkable data efficiency by learning expressive heuristics from very few examples. For instance, Ståhlberg, Bonet, and Geffner (2025) learn generalized policies and value functions, while Chen, Trevizan, and Thiébaux (2024) propose WL-GOOSE to reliably learn heuristics using classical machine learning on graph representations. Both lines of work achieve strong planning performance using few training instances. However, these supervised methods, as with the imitation learning methods described previously, fundamentally require a pre-existing optimal solver or domain knowledge to generate ground-truth labels for training. In contrast, our deep RL approach learns heuristics purely from interaction with the environment and does not require optimal labels, making it applicable to domains where optimal solvers are unavailable.

When training heuristic functions with deep RL, the manner in which the target network is updated is crucial to obtaining good performance. Updates that are too quick can cause training instability due to the non-stationarity of the training target, and updates that are too slow will result in data-inefficient training. DeepCubeA checked the loss every 5,000 iterations and only updated the target network if the loss had passed below a given threshold (Agostinelli et al. 2019b). DeepCubeA generated 10^{10} training instances during training and used batch weighted A* search with a batch size of up to 10,000. Agostinelli, Panta, and Khandelwal (2024) updated the target network when the number of states that the greedy policy solved on a validation set increased. However, this also exacerbated the data-intensive nature of training since a separate search using the greedy policy needed to be done before obtaining more training data.

Recent work has sought to improve the sample efficiency of the training pipeline itself. For example, Hadar, Agostinelli, and Shperberg (2026) introduced Limited-Horizon Bellman-based Learning (LHBL). LHBL replaces single-step updates with a limited-depth search during training to generate more accurate target values and sample states that better reflect the search frontier. This mitigates the distributional shift that can result in depression regions (Aine et al. 2016) often seen in random-walk data generation. Chervov et al. (2025) learn a heuristic function from 8×10^9 training instances using supervised learning on the length of random walks, which removes the need to perform Bellman updates and results in faster training. However, their heuristic search relies on performing beam search with a beam size of between 260,000 and 16 million. In contrast to previous methods, our approach requires on the order of 10^8 training instances and solves problems with GBFS run for a maximum of 1,000 iterations without batching.

Methods

Training and Data Generation

The parameters of the target network, θ^-, are updated to the current parameters, θ, every U training iterations. During the first U training iterations, the target network is a function that outputs zero for all inputs. To generate problem instances for training, depending on the domain, either a random walk is performed in reverse from the goal or a start state is sampled, a forward random walk is performed, and the terminal state on that walk is used to create a goal. The length of each random walk (reverse or forward) is uniformly distributed between 0 and K.

Given a training batch size, N, $U \cdot N$ training data instances are generated per update. In our experiments, we also test the effect of LHBL (Hadar, Agostinelli, and Shperberg 2026) on data efficiency. LHBL obtains training data by sampling problem instances using a random walk of length K, performing A* search on these problem instances with h_{θ^-} for a maximum of I iterations, and adding all states expanded during search to the training dataset. Therefore, we first generate $\frac{U \cdot N}{I}$ problem instances, where problem i is generated with a random walk of length k_i. If problem i is solved, then a new problem is generated, also with a random walk of length k_i.

Training Data Difficulty

The maximum length of a random walk, K, used to generate training instances can be used to estimate the difficulty of the training instances generated. In this context, the larger the true cost-to-go of an instance, the greater the difficulty. If K is too small, then h_θ will only be trained on instances with lower cost-to-go, and may not generalize well to instances with higher cost-to-go. However, if K is too large, then target values computed from Equation 2 may be biased too high, causing h_θ to be inaccurate on instances with lower cost-to-go. Therefore, one may either require domain-specific knowledge about the maximum number of steps required to solve problems or perform a hyperparameter search over K.

To address this problem, we introduce a novel, yet simple, strategy to automatically balance training data difficulty when training with LHBL. We use the percentage of states solved with the A* search performed by LHBL as an estimate of how difficult the generated instances are for h_θ. We initialize K to 1 at the beginning of training. We then obtain training data with LHBL. If the search performed by LHBL solves at least 50% of instances, then K is set to $\min(2K, K_{\text{max}})$, where K_{max} is a given hyperparameter. This adaptive scheme is analogous to automated curriculum generation in reinforcement learning (Graves et al. 2017; Portelas et al. 2020), which utilizes learning progress to dynamically adjust the difficulty of tasks presented to an agent. In our case, the curriculum is driven by the solved-percentage metric, automatically increasing K as the model improves. In our experiments, we find that we can set K_{max} to a number many times higher than the maximum number of steps needed to solve problems without a loss in performance. Therefore, one no longer needs to have an accurate estimate of the maximum number of steps required to solve problems.

Domain	Cov	Update Freq	Difficulty	Reuse
RC	50	7.67E+07	7.67E+07	**3.33E+07**
	75	2.63E+08	1.50E+08	**6.33E+07**
	100	-	2.93E+08	**1.70E+08**
35p	50	1.80E+08	1.77E+08	**7.33E+07**
	75	2.00E+08	2.00E+08	**1.27E+08**
	100	-	3.73E+08	**2.97E+08**
LO7	20	6.67E+07	4.33E+07	**4.00E+07**
	40	-	4.57E+08	**3.83E+08**
	55	-	-	**4.80E+08**
Q	20	**2.00E+07**	**2.00E+07**	**2.00E+07**
	40	1.40E+08	1.27E+08	**4.00E+07**
	60	-	4.60E+08	**2.93E+08**

Table 1: The average number of training instances required to meet the given coverage threshold for each of the data reduction methods investigated in this paper. The average is only taken if all runs eventually meet the threshold. For each column, the minimum is taken across all settings investigated for the corresponding method.

Data Reuse

One way to possibly improve data efficiency is to reduce the training batch size, N, by a factor of R (i.e. set the batch size to $\frac{N}{R}$), thus reducing the amount of newly generated training data per update. However, reducing N may lead to a higher variance estimate of the gradient when performing stochastic gradient descent. An alternative approach is to use data reuse: we keep N the same, but generate only $\frac{U \cdot N}{R}$ data points each update and store them in a buffer. To perform the U network parameter updates, we uniformly sample batches of size N from these points. With this approach, a single data point will be sampled and reused on average R times per update. This mechanism directly addresses the data-hungry generation cost while maintaining the original batch size, trading off potential overfitting for better gradient estimates.

Experiments

The domains we use to test our methods are the 3x3x3 Rubik's Cube, the 35-puzzle, the 7x7 LightsOut puzzle, and approximate quantum circuit synthesis with a single qubit. Since the Rubik's cube, 35-puzzle, and LightsOut have reversible action spaces and a single goal state, a random walk is performed from the goal state. For quantum circuit synthesis, the quantum algorithm that a circuit implements can be represented as a $2^n \times 2^n$ matrix, where n is the number of qubits. Therefore, we represent states and goals as 2×2 matrices. The action space adds a single gate from the Clifford + T gate set, which can approximate any unitary with arbitrary tolerance, ϵ (Nielsen and Chuang 2010). The transition costs for the T gate is 10, since it is more computationally burdensome, and for all other gates is 1. A start state is sampled by starting from the empty circuit and taking a

forward random walk of length between 0 and 100. Then, a goal is generated by taking a forward random walk and using the resulting matrix from the terminal state of that walk as the goal. The tolerance for testing whether two matrices are equivalent is set to $\epsilon = 0.001$.

All of these domains feature problem instances with fixed-size state representations. Therefore, for all domains, the DNN architecture that represents the heuristic function is a fully-connected residual network (He et al. 2016) similar to the one used in DeepCubeA (Agostinelli et al. 2019b), where the first hidden layer is of size 1,000 with a linear activation function, followed by 4 residual blocks, followed by an output layer of size 1 and a linear activation function. Each residual block has two layers of size 1,000, uses batch normalization (Ioffe and Szegedy 2015), and uses rectified linear activation functions (Glorot, Bordes, and Bengio 2011). All states from the puzzles are fed into the network as flat one-hot representations (e.g., the Rubik's cube has a 324-dimensional input) while the matrices representing quantum algorithms use the Euler angle representation of the unitaries (Diaconis and Forrester 2017). While more expressive architectures such as Graph Neural Networks (GNNs) have shown state-of-the-art performance for pathfinding and planning problems, they are typically employed when problem instance sizes vary, which is not the case here. Furthermore, ResNets are significantly faster at inference than GNNs, which is critical for maximizing node expansions under a time limit. Critically, our proposed data efficiency improvements are fundamentally architecture-agnostic and can complement GNNs or other novel neural architectures without modification. The DNN is optimized with ADAM (Kingma and Ba 2014) with an initial learning rate of 0.001 and a decay rate of 0.9999993. Unless otherwise stated, we use a batch size of 10,000, train for 50,000 iterations, and set K to 30 for the Rubik's cube, 1,000 for the 35-puzzle, 50 for 7x7 LightsOut, and 1,000 for quantum circuit synthesis. We repeat each experiment three times.

To measure how coverage changes as a function of the amount of generated training data, we create a validation set for each domain. The validation set consists of N_v instances and each instance is generated using a random walk whose length is sampled from a uniform distribution between 0 and W_v. For the Rubik's Cube, 35-puzzle, 7x7 LightsOut, and quantum circuit synthesis, N_v is set to 1,000, 1,000, 100, 1,000 and W_v is set to 1,000, 10,000, 100, and 1,000 respectively. To solve instances, we perform GBFS with h_θ for a maximum of 1,000 iterations.

To measure the effect of update frequency on data efficiency, we try update frequencies, U, of 10, 100, and 1,000, both without and with LHBL, where LHBL performs A* search for a maximum of 1,000 iterations. We also use the original DeepCubeA settings of checking every 5,000 iterations if the loss has gone below a given threshold of 0.05, 1.0, and 1.0 for the Rubik's cube, 35-puzzle, and 7x7 LightsOut, respectively (Agostinelli et al. 2019a). Although DeepCubeA did not test on quantum circuit synthesis, we use a similar setting of checking every 5,000 iterations with a loss threshold of 1.0. The results in Figure 1 show that the DeepCubeA setting has relatively low coverage, that LHBL almost always results in better performance and that an update frequency of 100 performs best in most cases. Therefore, we use an update frequency of 100 and LHBL for all subsequent experiments.

To measure the effect of training data difficulty on data efficiency, we obtain results when multiplying the original (short) K used for each domain by 10 (medium) and 100 (long) both without and with the automatic balancing. The results in Figure 2 show that, for the Rubik's cube and 35-puzzle, larger values of K tend to result in data-inefficient learning. However, with balancing, larger K achieve higher coverage much faster than shorter ones. Notably, the 35-puzzle succeeds even with short random walks because its local gradients (e.g., Manhattan distance) are highly informative, despite requiring much deeper solutions ($\sim$ 80 steps) than the Rubik's cube ($\sim$ 26 steps). In contrast, the Rubik's cube has a "cliff-like" landscape that requires medium lookaheads to find meaningful learning signals. When automatically balancing for the 35-puzzle and quantum circuit synthesis, K did not exceed 1024 and 2048, respectively. Therefore, for all subsequent experiments, we use the medium random walk length for the 35-puzzle and quantum circuit synthesis and long random walk length for the Rubik's cube and LightsOut. Although 7x7 LightsOut shows that large K without automatic balancing performs better, when we reduced the data generated per update, we found that balancing became necessary. Therefore, we use balancing for all domains for subsequent experiments.

To measure the effect of data reuse on data efficiency, we reduce the data generated per update by a factor of R. We do this by reducing the batch size by R and by keeping the batch size the same and directly reducing the number of training instances generated by R, thus reusing generated data R times. We increase the number of iterations by R so the total data generated is unchanged. The results in Figure 3 show that, for the Rubik's cube, LightsOut, and quantum circuit synthesis, reusing data usually performs better compared to reducing batch size. For the 35-puzzle, both approaches perform similarly except in the case where $R = 10$, where reducing the batch size performs better. Table 1 isolates the effect of our different methods across all experiments. It illustrates that learning becomes more data-efficient when adding automatic difficulty adaptation and data reuse.

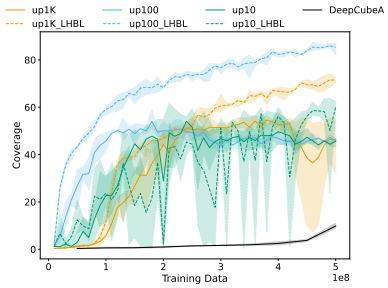
Discussion

Our experiments show that an update frequency of around 100 iterations, more difficult training data with automatic balancing of difficulty, and data reuse tend to result in higher coverage with less data. Although 7x7 LightsOut and quantum circuit synthesis did not reach 100% coverage using the allotted amount of data, future work can build on the methods presented in this paper to do so. The weak generalization in LightsOut likely stems from its reliance on linear algebra over GF(2), where local moves have complex global ripple effects. While Convolutional Neural Networks (CNNs) might possess a better inductive bias for this grid-based structure than our fully-connected ResNet, tailoring the network architecture for each domain was not the focus of this study. Instead, we maintained a fixed architecture to

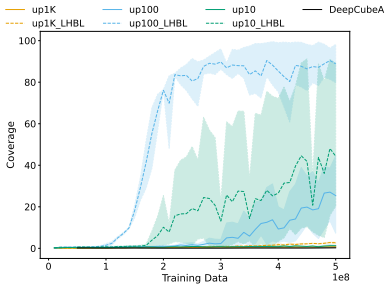
isolate the improvements gained specifically from our training pipeline. Achieving 100% coverage for the Rubik's cube required 1.7×10^8 training instances for data reuse with an R of 2. Data is processed at a rate of 2.4×10^3 instances/sec on a MacBook Air laptop with an M4 chip and 1.25×10^4 instances/sec using a single NVIDIA L40S GPU with 8 Intel Xeon Platinum CPUs, resulting in a total time to reach 100% coverage of 19.9 and 3.8 hours of training, respectively. Note that data processing time per instance refers to the average wall-clock time required to generate an instance via a random walk, perform the target value computation via search, and complete one forward and backward pass for the network update. For the 35-puzzle, these numbers are 3.82×10^3 instances/sec and 1.25×10^4 instances/sec, resulting in 22 and 6.6 hours of training, respectively. Even on a laptop, training time is significantly less than the multiple days of training required by previous methods that use one or more GPUs (Agostinelli et al. 2019b; Hadar, Agostinelli, and Shperberg 2026).

Conclusion

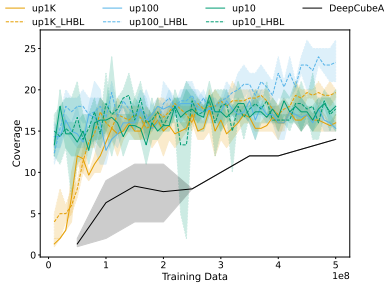
In this paper, we conducted an empirical study on data-efficient deep reinforcement learning for satisficing heuristic search. We demonstrated that simple strategies, specifically frequent target updates, automatic task difficulty balancing informed by curriculum learning, and data reuse, drastically reduce the amount of data required to train effective heuristic functions. For domains like the Rubik's cube, we achieved 100% coverage using up to two orders of magnitude less training data than previous methods. This reduces training from days on powerful servers to hours on a consumer laptop. By lessening the excessive data burden, these strategies make training heuristic functions represented as deep neural networks far more accessible to a wider span of practitioners and researchers.



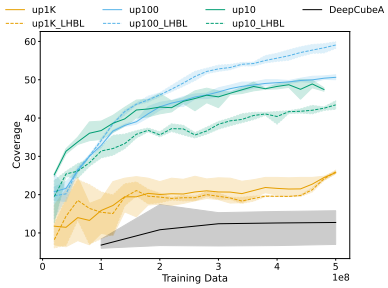
(a) Rubik's cube



(b) 35-puzzle

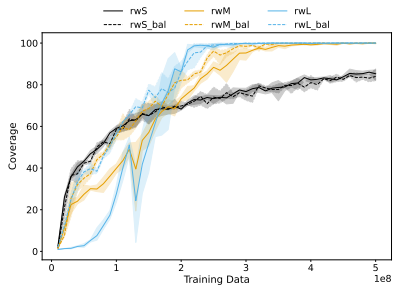


(c) 7x7 LightsOut

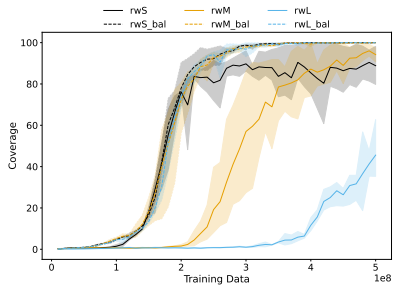


(d) Quantum Circuit Synthesis

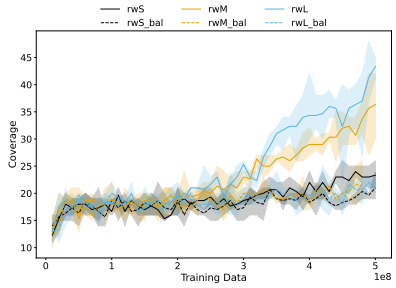
Figure 1: The effect of update frequency and LHBL on coverage. Coverage represents the percentage of solved validation tasks. An update frequency of 100 iterations while using LHBL performs best in most cases.



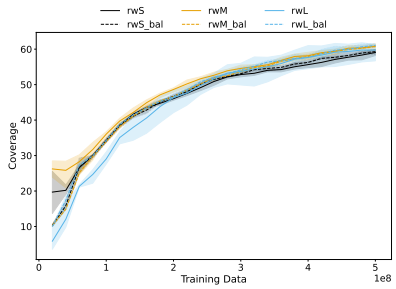
(a) Rubik's cube



(b) 35-puzzle

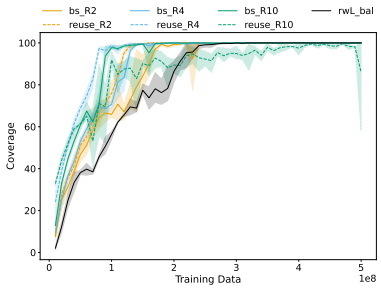


(c) 7x7 LightsOut

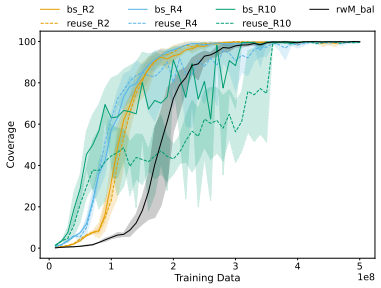


(d) Quantum Circuit Synthesis

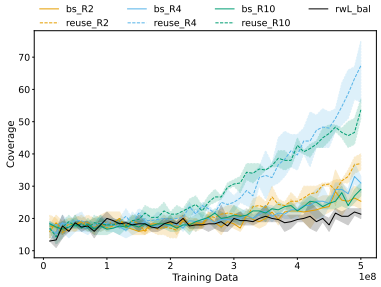
Figure 2: The effect of short (S), medium (M), and long (L) random walk lengths used to generate problem instances both without and with automatic difficulty balancing (_bal) on coverage. Coverage represents the percentage of solved validation tasks. Without balancing, longer random walks tend to result in data inefficiency. However, with balancing, longer random walks tend to be more data efficient than shorter ones. All settings use an update frequency of 100 and LHBL.



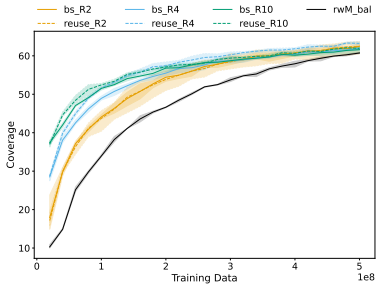
(a) Rubik's cube



(b) 35-puzzle



(c) 7x7 LightsOut



(d) Quantum Circuit Synthesis

Figure 3: The effect of batch size (bs) and data reuse (reuse) on coverage. Coverage represents the percentage of solved validation tasks. Data reuse tends to result in faster increases in coverage compared to reducing batch size. All settings use an update frequency of 100, LHBL, and automatic difficulty balancing.

References

Agostinelli, F.; McAleer, S.; Shmakov, A.; and Baldi, P. 2019a. DeepCubeA Code Ocean. https://codeocean.com/capsule/5723040/tree/v1.

Agostinelli, F.; McAleer, S.; Shmakov, A.; and Baldi, P. 2019b. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8): 356–363.

Agostinelli, F.; Panta, R.; and Khandelwal, V. 2024. Specifying goals to deep neural networks with answer set programming. In *34th International Conference on Automated Planning and Scheduling*.

Aine, S.; Swaminathan, S.; Narayanan, V.; Hwang, V.; and Likhachev, M. 2016. Multi-heuristic A*. *The International Journal of Robotics Research*, 35(1-3): 224–243.

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2011. Learning heuristic functions for large state spaces. *Artif. Intell.*, 175(16-17): 2075–2098.

Bao, N.; and Hartnett, G. S. 2024. Twisty-puzzle-inspired approach to Clifford synthesis. *Physical Review A*, 109(3): 032409.

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press.

Bertsekas, D. P.; and Tsitsiklis, J. N. 1996. *Neuro-dynamic programming*. Athena Scientific. ISBN 1-886529-10-8.

Bhardwaj, M.; Choudhury, S.; and Scherer, S. A. 2017. Learning Heuristic Search via Imitation. In *CoRL*, volume 78 of *PMLR*, 271–280. PMLR.

Bramanti-Gregor, A.; and Davis, H. W. 1993. The Statistical Learning of Accurate Heuristics. In *IJCAI*, 1079–1087.

Chen, D. Z.; Trevizan, F.; and Thiébaux, S. 2024. Return to Tradition: Learning Reliable Heuristics with Classical Machine Learning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 34, 68–77.

Chen, Q.; Du, Y.; Jiao, Y.; Lu, X.; Wu, X.; and Zhao, Q. 2024. Efficient and practical quantum compiler towards multi-qubit systems with deep reinforcement learning. *Quantum Science and Technology*, 9(4): 045002.

Chervov, A.; Khoruzhii, K.; Bukhal, N.; Naghiyev, J.; Zamkovoy, V.; Koltsov, I.; Cheldieva, L.; Sychev, A.; Lenin, A.; Obozov, M.; et al. 2025. A machine learning approach that beats Rubik's cubes. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Chrestien, L.; Pevný, T.; Edelkamp, S.; and Komenda, A. 2023. Optimize Planning Heuristics to Rank, not to Estimate Cost-to-Goal. *CoRR*, abs/2310.19463.

Chrestien, L.; Pevný, T.; Komenda, A.; and Edelkamp, S. 2021. Heuristic Search Planning with Deep Neural Networks using Imitation, Attention and Curriculum Learning. *CoRR*, abs/2112.01918.

Corrêa, A. B.; Pereira, A. G.; and Seipp, J. 2025. Classical Planning with LLM-Generated Heuristics: Challenging the State of the Art with Python Code. *arXiv preprint*.

Diaconis, P.; and Forrester, P. J. 2017. Hurwitz and the origins of random matrix theory in mathematics. *Random Matrices: Theory and Applications*, 6(01): 1730001.

Doran, J. E.; and Michie, D. 1966. Experiments with the Graph Traverser program. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 294(1437): 235–259.

Ferber, P.; Helmert, M.; and Hoffmann, J. 2020. Neural Network Heuristics for Classical Planning: A Study of Hyperparameter Space. In *ECAI*, volume 325, 2346–2353.

Fink, M. 2007. Online Learning of Search Heuristics. In *AISTATS*, volume 2 of *JMLR Proceedings*, 114–122.

Garrett, C. R.; Kaelbling, L. P.; and Lozano-Pérez, T. 2016. Learning to Rank for Synthesizing Planning Heuristics. In *IJCAI*, 3089–3095. IJCAI/AAAI Press.

Glorot, X.; Bordes, A.; and Bengio, Y. 2011. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 315–323.

Graves, A.; Bellemare, M. G.; Menick, J.; Munos, R.; and Kavukcuoglu, K. 2017. Automated Curriculum Learning for Neural Networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, 1311–1320. JMLR.org.

Groshev, E.; Goldstein, M.; Tamar, A.; Srivastava, S.; and Abbeel, P. 2018. Learning Generalized Reactive Policies Using Deep Neural Networks. In *ICAPS*, 408–416.

Hadar, G.; Agostinelli, F.; and Shperberg, S. S. 2026. Beyond Single-Step Updates: Reinforcement Learning of Heuristics with Limited-Horizon Search. In *Proceedings of the AAAI Conference on Artificial Intelligence*. Forthcoming.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2): 100–107.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

Ioffe, S.; and Szegedy, C. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, 448–456. pmlr.

Kingma, D. P.; and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

LeCun, Y.; Bengio, Y.; and Hinton, G. 2015. Deep learning. *nature*, 521(7553): 436.

Ling, H.; Parashar, S.; Khurana, S.; Olson, B.; Basu, A.; Sinha, G.; Tu, Z.; Caverlee, J.; and Ji, S. 2025. Complex LLM planning via automated heuristics discovery. *arXiv preprint*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control

through deep reinforcement learning. *Nature*, 518(7540): 529–533.

Nielsen, M. A.; and Chuang, I. L. 2010. *Quantum computation and quantum information*. Cambridge university press.

Orseau, L.; and Lelis, L. H. S. 2021. Policy-Guided Heuristic Search with Guarantees. In *AAAI*, 12382–12390.

Panta, R.; Tavakoli, M.; Geils, C.; Baldi, P.; and Agostinelli, F. 2024. Finding Reaction Mechanism Pathways with Deep Reinforcement Learning and Heuristic Search. In *ICAPS Workshop on Bridging the Gap between AI Planning and Reinforcement Learning*. ICAPS.

Pohl, I. 1970. Heuristic search viewed as path finding in a graph. *Artificial intelligence*, 1(3-4): 193–204.

Portelas, R.; Colas, C.; Weng, L.; Hoffman, K.; and Oudeyer, P.-Y. 2020. Automatic Curriculum Learning For Deep RL: A Short Survey. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, 4819–4825.

Samadi, M.; Felner, A.; and Schaeffer, J. 2008. Compressing pattern databases with learning. In *AAAI*, volume 8, 368–373.

Schmidhuber, J. 2015. Deep learning in neural networks: An overview. *Neural networks*, 61: 85–117.

Shen, W.; Trevizan, F. W.; and Thiébaux, S. 2020. Learning Domain-Independent Planning Heuristics with Hypergraph Networks. In *ICAPS*, 574–584. AAAI Press.

Siddique, P. J.; Gue, K. R.; and Usher, J. S. 2021. Puzzle-based parking. *Transportation Research Part C: Emerging Technologies*, 127: 103112.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2025. Learning More Expressive General Policies for Classical Planning Domains. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 26697–26706.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

Takahashi, T.; Sun, H.; Tian, D.; and Wang, Y. 2019. Learning Heuristic Functions for Mobile Robot Path Planning Using Deep Neural Networks. In *ICAPS*, 764–772.

Toyer, S.; Thiébaux, S.; Trevizan, F. W.; and Xie, L. 2020. ASNets: Deep Learning for Generalised Planning. *J. Artif. Intell. Res.*, 68: 1–68.

Turner, I.; Agostinelli, F.; and Fu, P. 2025. Quantum Circuit Synthesis with Deep Reinforcement Learning and Heuristic Search. In *ICAPS Workshop on Bridging the Gap between AI Planning and Reinforcement Learning*. ICAPS.

Zhang, Y.-H.; Zheng, P.-L.; Zhang, Y.; and Deng, D.-L. 2020. Topological Quantum Compiling with Reinforcement Learning. *Physical Review Letters*, 125(17): 170501.