Learning Graph Search Heuristics

Michal Pándy *
University of Cambridge

Rex Ying
Stanford University

Gabriele Corso
MIT

Petar Veličković
DeepMind

Jure Leskovec
Stanford University

Pietro Liò
University of Cambridge

Abstract

Searching for a path between two nodes in a graph is one of the most well-studied and fundamental problems in computer science. In numerous domains such as robotics, AI, or biology, practitioners develop search heuristics to accelerate their pathfinding algorithms. However, it is a laborious and complex process to hand-design heuristics based on the problem and the structure of a given use case. Here we present PHIL (Path Heuristic with Imitation Learning), a novel neural architecture and a training algorithm for discovering graph search and navigation heuristics from data by leveraging recent advances in imitation learning and graph representation learning. At training time, we aggregate datasets of search trajectories and ground-truth shortest path distances, which we use to train a specialized graph neural network-based heuristic function using backpropagation through steps of the pathfinding process. Our heuristic function learns graph embeddings useful for inferring node distances, runs in constant time independent of graph sizes, and can be easily incorporated in an algorithm such as A* at test time. Experiments show that PHIL reduces the number of explored nodes compared to state-of-the-art methods on benchmark datasets by 40.8% on average and allows for fast planning in time-critical robotics domains.

1 Introduction

Search heuristics are essential in several domains, including robotics, AI, biology, and chemistry [1, 2, 3, 4, 5, 6]. For example, in robotics, complex robot geometries often yield slow collision checks, and search algorithms are constrained by the robot's onboard computation resources, requiring well-performing search heuristics that visit as few nodes as possible [1, 4]. In AI, domain-specific search heuristics are useful for improving the performance of inference engines operating on knowledge bases [3, 5]. Search heuristics have been previously also developed to reduce search efforts in protein-protein interaction networks [6] and in planning chemical reactions that can synthesize target chemical products [2]. This broad set of applications underlines the importance of good search heuristics that are applicable to a wide range of problems.

While there has been significant progress in designing search heuristics, it remains a challenging problem. Classical approaches [7, 8] tend to hand-design search heuristics, which require domain knowledge and a lot of trial and error. Domain-independent classical approaches [9, 10] develop useful meta-heuristics; however, learning-based methods demonstrate that this process can be learned from data. Learning-based methods face a different set of challenges. Firstly, the data distribution is not i.i.d., as newly encountered graph nodes depend on past heuristic values, which means that supervised learning-based methods [11, 12, 13, 14, 15] under-perform methods that take into account the sequential decision making aspect of the problem [1]. Secondly, heuristics should run fast, with ideally constant time complexity. Otherwise, the overall asymptotic time complexity of the search

*Correspondence to mpmisko@gmail.com.

Physical Reasoning and Inductive Biases for the Real World at NeurIPS 2021.

procedure could be increased. Finally, as the environment (search graph) sizes increase, reinforcement learning-based heuristic learning approaches tend to perform poorly [16]. State-of-the-art imitation learning-based methods can learn useful search heuristics [1]; however, these methods still rely on feature-engineering for a specific domain and do not generally guarantee a constant time complexity with respect to graph sizes.

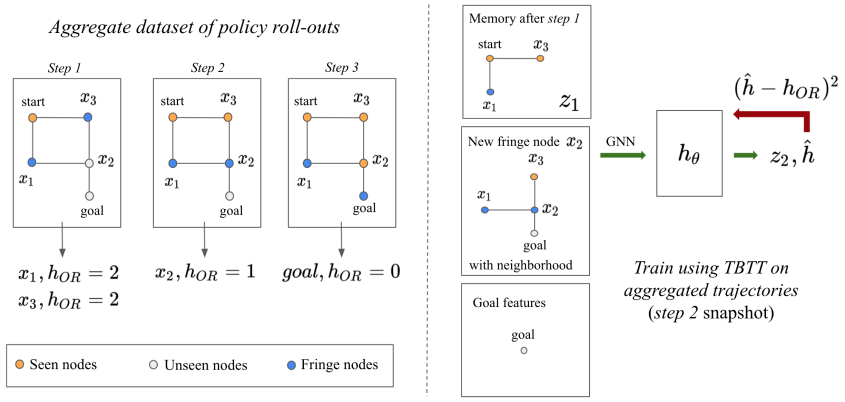


Figure 1: Main components of PHIL: On the left, we roll-out and aggregate search trajectories from the start node to the goal node. Each trajectory step contains a set of newly added fringe nodes with bounded random subsets of their 1-hop neighborhoods and their oracle (h^*) distances to the goal node. On the right, we use truncated backpropagation through time on each collected trajectory to train h_θ, where $\hat{h}$ is the predicted distance between x_2 and x_g, and z_2 is the updated state of the memory.

In this paper, we propose *Path Heuristic with Imitation Learning* (PHIL, Figure 1), a framework that extends the recent imitation learning-based heuristic search paradigm with a learnable *explored graph memory*. This means that PHIL learns a representation that allows it to capture the structure of the so far explored graph, so that it can then better select what node to explore next. We train our approach to predict the oracle node-to-goal distances of graph nodes during search. Key to our approach is a *specialized graph neural network architecture*, which allows us to apply PHIL to diverse graphs from different domains and encodes search-specific inductive biases in a constant time complexity.

2 Preliminaries

Graph search. Suppose that we are given an unweighted connected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of nodes, and $\mathcal{E}$ a corresponding set of edges. Further suppose that each node $i \in \mathcal{V}$ has corresponding features $x_i \in \mathbb{R}^{D_v}$, and each edge $(i, j) \in \mathcal{E}$ has features $e_{ij} \in \mathbb{R}^{D_e}$. Assume that we are also given a start node $v_s \in \mathcal{V}$ and a goal node $v_g \in \mathcal{V}$. At any stage of our search algorithm, we can partition the nodes of our graph into three sets as $\mathcal{V} = \mathcal{V}_{seen} \cup \mathcal{V}_{fringe} \cup \mathcal{V}_{unseen}$, where $\mathcal{V}_{seen}$ are the nodes already explored, $\mathcal{V}_{fringe}$ are candidate nodes for exploration (i.e., all nodes connected to any node in $\mathcal{V}_{seen}$, but not yet in $\mathcal{V}_{seen}$), and $\mathcal{V}_{unseen}$ is the rest of the graph. Each *expansion* moves a node from $\mathcal{V}_{fringe}$ to $\mathcal{V}_{seen}$, and adds the neighbors of the given node from $\mathcal{V}_{unseen}$ to $\mathcal{V}_{fringe}$. We call the set of newly added fringe nodes $\mathcal{V}_{new}$ at each search step. At the start of the search procedure, $\mathcal{V}_{seen} = \{v_s\}$ and we expand the nodes until v_g is encountered (i.e., until $v_g \in \mathcal{V}_{seen}$).

Greedy best-first search. We can perform *greedy best-first search* using a greedy fringe expansion policy, such that we always expand the node $v \in \mathcal{V}_{fringe}$ that minimizes $h(v, v_g)$. Here, $h : \mathcal{V} \times \mathcal{V} \longrightarrow \mathbb{R}$ is a tailored heuristic function for a given use case. In our work, we are interested in learning a function h that predicts shortest path lengths, this way minimizing $|\mathcal{V}_{seen}|$ in a *greedy best-first search* regime.

Imitation of perfect heuristics. Partially observable Markov decision processes (POMDPs) are a suitable framework to describe the problem of learning search heuristics [1]. We can have $s = (\mathcal{V}_{seen}, \mathcal{V}_{fringe}, \mathcal{V}_{unseen})$ as our state, an action $a \in \mathcal{A}$ corresponds to moving a node from $\mathcal{V}_{fringe}$ to $\mathcal{V}_{seen}$, and the observations $o \in \mathcal{O}$ are the features of newly included nodes in $\mathcal{V}_{fringe}$. We also

define a history $\psi \in \Psi$ as a sequence of observations $\psi = o_1, o_2, o_3, ...$ Our work leverages the observation that using a heuristic function during greedy best-first search that correctly determines the length of the shortest path between fringe nodes and the goal node will also yield minimal $|\mathcal{V}_{seen}|$. For training, we adopt a perfect heuristic h^*, similar to [1], which has full information about s during search. Such oracle can provide ground-truth distances $h^*(s, v, v_g)$, where $v \in \mathcal{V}_{fringe}$. To conclude, we define a *greedy best-first search policy* π_θ that uses a parameterized heuristic h_θ to expand nodes from $\mathcal{V}_{fringe}$ with minimal heuristic values.

3 Approach

Training objective. With the aim of minimizing $|\mathcal{V}_{seen}|$ after search, our goal is to train a parameterized heuristic function $h_\theta : \Psi \times \mathcal{V} \times \mathcal{V} \longrightarrow \mathbb{R}$ to predict ground-truth node distances h^* and use h_θ within a greedy best-first policy π_θ at test time. More specifically, we assume access to a distribution over graphs $P_{\mathcal{G}}$, a start-goal node distribution $P_{v_{sg}}(\cdot \,|\, \mathcal{G})$, and a time horizon T. Moreover, we assume a joint state-history distribution $s, \psi \sim P_s(\cdot \mid \mathcal{G}, t, \pi_\theta, v_s, v_g)$, where P_s represents the probability our search being in state s, at time $0 \leq t \leq T$ on graph $\mathcal{G}$ with pathfinding problem (v_s, v_g), with a greedy best-first search policy π_θ using heuristic h_θ. Hence, our goal is minimizing:

$$\mathcal{L}(\theta) = \mathop{\mathbb{E}}_{\substack{\mathcal{G}\sim P_{\mathcal{G}},\\ (v_s,v_g)\sim P_{v_{sg}}\\ t\sim\mathcal{U}(0,\ldots,T),\\ s,\psi\sim P_s}} \Big[\frac{1}{|\mathcal{V}_{new}|} \sum_{v\in\mathcal{V}_{new}} (h^*(s, v, v_g) - h_\theta(\psi, v, v_g))^2\Big] \tag{1}$$

Imitation learning algorithm. The high-level idea of our algorithm (Appendix C) is that we aggregate trajectories of search traces (i.e., sequences of new fringe nodes) and use truncated backpropagation through time to optimize h_θ after each data-collection step. In particular, after sampling a graph $\mathcal{G}$ and a search problem v_s, v_g, we execute our greedy learned policy π_θ induced by h_θ for $t \sim \mathcal{U}(0, \ldots, T - t_\tau)$ expansions, where T is the episode time horizon, and t_τ is the roll-out length. We obtain a new state $s = (\mathcal{V}^0_{seen}, \mathcal{V}^0_{fringe}, \mathcal{V}^0_{unseen})$, and an initial memory state z_t. Afterward, we execute/roll-out for t_τ steps our mixture policy π_{mix}, which is obtained by probabilistically blending π_θ and the greedy best-first policy induced by the oracle heuristic π^*. In a roll-out, we collect sequences of new fringe nodes, together with their ground-truth distances to the goal v_g, given by h^*. Once the roll-out is complete, we aggregate the obtained trajectory and the initial state for the following optimization using backpropagation through time.

Recurrent GNN architecture. In each forward pass, h_θ obtains a set of new fringe nodes $\mathcal{V}_{new}$, the goal node v_g, and the memory z_t at time step t. We represent each node in $\mathcal{V}_{new}$ using its features $x_i \in \mathbb{R}^{D_v}$, and likewise the goal node v_g using its features $x_g \in \mathbb{R}^{D_v}$. Further, for each $i \in \mathcal{V}_{new}$, we uniformly sample an $n \in \mathbb{N}_{\geq 0}$ bounded set of nodes present in the 1-hop neighborhood of i, calling this set $\mathcal{N}_i$, with $|\mathcal{N}_i| \leq n$. This sampling step produces a set of neighboring node features, where each $j \in \mathcal{N}_i$ has features $x_j \in \mathbb{R}^{D_v}$, and corresponding edge features $e_{ij} \in \mathbb{R}^{D_e}$.

Algorithm 1: Heuristic func. (h_θ) forward pass

Obtain $x_i, x_j, e_{ij}, x_g\ z_t$;
$x_i \leftarrow f(x_i, x_g, D_{EUC}(x_i, x_g), D_{COS}(x_i, x_g))$;
$x_j \leftarrow f(x_j, x_g, D_{EUC}(x_j, x_g), D_{COS}(x_j, x_g))$;
$g_i \leftarrow \phi(x_i, \oplus_{j\in\mathcal{N}_i} \gamma(x_i, x_j, e_{ij}))$;
$g'_i, z_{i,t+1} \leftarrow \text{GRU}(g_i, z_t)$;
$z_{t+1} \leftarrow \overline{z_{i,t+1}}$;
$\hat{h}_i \leftarrow \text{MLP}(g'_i, x_g)$;
return $\hat{h}_i, z_{t+1}$;

h_θ **forward pass.** In Algorithm 1, $f, \phi, \gamma, \text{GRU}[17], \text{MLP}$ are each parameterised differentiable functions, with ϕ, γ representing the *update* and *message* functions [18] of a graph neural network, respectively. In our forward pass, using the function f, we first project x_i, x_j into a node embedding space, together with the goal features x_g, and their Euclidean (D_{EUC}) and cosine distances (D_{COS}). After that, using a 1-layer GNN, we perform a single convolution over each x_i and the corresponding neighborhood $\mathcal{N}_i$, to obtain g_i. Our graph convolution processing step allows us to easily incorporate edge features and work with variable sizes of $\mathcal{N}_i$. After the graph convolution, we apply the GRU module over each embedding g_i to obtain hidden states $z_{i,t+1}$, and new embeddings g'_i. We compute the sample mean of $z_{i,t+1}$ for each node $i \in \mathcal{V}_{new}$ to obtain a new hidden state z_{t+1}, and process g'_i with x_g using an MLP to compute the distances between the graph nodes.

Permutation invariant $\mathcal{V}_{new}$ embedding. There is a trade-off between processing new fringe nodes in batch, as in Algorithm 1, and processing them sequentially. Namely, when we process the nodes in

batch, we do not use the in-batch observations to predict batch node values, which means that z_t is slightly outdated. On the other hand, in PHIL, batch processing allows us to compute the heuristic values of all $v \in \mathcal{V}_{new}$ in parallel on a GPU and preserves the memory's permutation invariance with respect to nodes in $\mathcal{V}_{new}$. This approach provides additional scalability as we can process values in parallel and PHIL does not have to infer permutation invariance in $\mathcal{V}_{new}$ from data.

4 Experiments

Heuristic search in grids. In this section, we evaluate PHIL on 8, 200×200 8-connected grid graph-based benchmark datasets by Bhardwaj *et al.* [1]. Each dataset contains 200 training graphs, 70 validation graphs, and 100 test graphs. Example graphs from each dataset can be found in Table 1. For a detailed description of datasets and baselines, please refer to Appendix F.

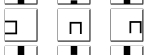
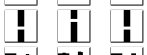
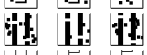
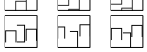
Dataset	Graph Examples	SaIL	SL	CEM	QL	h_{euc}	h_{man}	A*	MHA*	PHIL
Alternating gaps		1.000	11.077	1.077	25.641	25.641	25.641	25.641	25.641	**0.615**
Single Bugtrap		1.000	1.354	**0.361**	6.329	1.165	1.215	6.329	1.772	0.544
Shifting gaps		1.000	4.462	9.615	9.615	4.865	5.663	9.615	7.731	**0.260**
Forest		1.000	1.194	1.333	3.361	1.139	1.194	27.778	2.083	**0.778**
Bugtrap+Forest		1.000	2.612	1.238	6.803	2.789	2.293	6.803	3.177	**0.810**
Gaps+Forest		1.000	4.525	4.525	4.525	4.525	4.525	4.525	4.525	**0.213**
Mazes		1.000	2.311	4.650	3.874	1.796	1.660	9.709	2.709	**0.495**
Multiple Bugtraps		1.000	1.002	2.088	1.743	1.353	1.288	2.088	1.829	**0.382**

Table 1: The number of expanded graph nodes of PHIL with respect to SaIL. We can observe that out of all baselines, SaIL performs best. PHIL outperforms SaIL by 48.8% on average over all datasets, with a maximal search effort reduction of 78.7% in the *Gaps+Forest* dataset.

As we can see in Table 1, PHIL outperforms the best baseline (*SaIL*) on all datasets, with an average reduction of explored nodes before v_g is found of 48.8%. Even with *CEM* performing better than PHIL on *Single Bugtrap*, PHIL reduces the necessary search effort compared with the best baseline on each dataset by 40.8% on average. Qualitatively, observing Figure 2, we can attribute these results to PHIL's ability to **reduce the redundancy in explored nodes** during a search, as can be seen in Appendix A. Further, PHIL converges in up to $N = 36$ iterations, with $t_\tau = 32$ (i.e., after observing less than $N * t_\tau * max(|\mathcal{V}_{new}|) \approx 9,216$ shortest path distances, where we take $max(|\mathcal{V}_{new}|) = 8$ as the maximal size of $\mathcal{V}_{new}$). According to figures reported in [1], this is approximately $5\times$ less data than it takes for SaIL to converge. Although neither the SaIL or PHIL code-bases were optimized for runtime speed, we found that our implementation of PHIL runs about $7\times$ faster than the publicly available implementation of SaIL on the *Gaps+Forest* dataset.

Dataset	SL	A*	h_{euc}	BFS	PHIL	Shortest path
Room simple	1.124	76.052	1.000	291.888	**0.978**	0.938
Room adversarial	2.022	67.215	1.000	238.768	**0.895**	0.853

Table 2: Results of PHIL in the context of planning for indoor UAV flight. PHIL outperforms all baselines in both the *room simple* and *room adversarial* environments while remaining close performance-wise to the optimal number of expansions.

Planning for drone flight. In our final experiment, we use PHIL to plan collision-free paths in a practical drone flight use case within an indoor environment. For more detail about each environment, please refer to the supplementary material. We discretize the environments into 3D grid graphs of size $50 \times 50 \times 25$, and randomly remove 5 sub-graphs of size $5 \times 5 \times 5$ both during training and testing, this way simulating real-life planning scenarios with random obstacles. In Table 2 we report the ratio of expanded nodes with respect to h_{euc}. As we can observe in Table 2, PHIL outperforms all baselines in both environments. Interestingly, PHIL expands only approximately 4.2% more nodes in the simple room than least possible and 4.9% more in the adversarial room case. The same figures for the greedy method (h_{euc}) are 6.6% and 17.2%, respectively. These results indicate that PHIL is capable of learning planning strategies that are close to optimal in both *simple* and *adversarial* graphs[2], while the performance of naive heuristics degrades.

[2]We provide a video demonstration of PHIL running in room adversarial: `https://cutt.ly/eniu5ax`.

References

[1] Mohak Bhardwaj, Sanjiban Choudhury, and Sebastian Scherer. "Learning heuristic search via imitation". In: *Conference on Robot Learning*. 2017.

[2] Binghong Chen et al. "Retro*: Learning retrosynthetic planning with neural guided A* search". In: *ICML*. 2020.

[3] Martin Gebser et al. "Domain-specific heuristics in answer set programming". In: *AAAI*. 2013.

[4] Thi Thoa Mac et al. "Heuristic approaches in robot path planning: A survey". In: *Robotics and Autonomous Systems*. 2016.

[5] Abhishek Sharma and Keith M. Goolsbey. "Identifying useful inference paths in large commonsense knowledge bases by retrograde analysis". In: *AAAI*. 2017.

[6] Cheng-Yu Yeh et al. "Pathway detection from protein interaction networks and gene expression data using color-coding methods and A* search algorithms". In: *The Scientific World booktitle*. 2012.

[7] Danish Khalidi, Dhaval Gujarathi, and Indranil Saha. "T*: A heuristic search based path planning algorithm for temporal logic specifications". In: *ICRA*. 2020.

[8] Bhargav Adabala and Zlatan Ajanovic. "A multi-heuristic search-based motion planning for autonomous parking". In: *30th International Conference on Automated Planning and Scheduling: Planning and Robotics Workshop*. 2020.

[9] Nir Lipovetzky and Hector Geffner. "Best-first width search: Exploration and exploitation in classical planning". In: *Thirty-First AAAI Conference on Artificial Intelligence*. 2017.

[10] Florent Teichteil-Königsbuch, Miquel Ramırez, and Nir Lipovetzky. "Boundary Extension Features for Width-Based Planning with Simulators on Continuous-State Domains." In: *IJCAI*. 2020, pp. 4183–4189.

[11] Jes ús Virseda, Daniel Borrajo, and Vidal Alcázar. "Learning heuristic functions for cost-based planning". In: *Planning and Learning*. 2013.

[12] Christopher Makoto Wilt and Wheeler Ruml. "Building a heuristic for greedy search." In: *SOCS*. 2015.

[13] Caelan Reed Garrett, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. "Learning to rank for synthesizing planning heuristics". In: *IJCAI*. 2016.

[14] Jordan Thayer, Austin Dionne, and Wheeler Ruml. "Learning inadmissible heuristics during search". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. 2011.

[15] Soonkyum Kim and Byungchul An. "Learning heuristic A*: efficient graph search using neural network". In: *ICRA*. 2020.

[16] Sanjiban Choudhury et al. "Data-driven planning via imitation learning". In: *The International Journal of Robotics Research* 37.13-14 (2018), pp. 1632–1672.

[17] Kyunghyun Cho et al. "Learning phrase representations using rnn encoder–decoder for statistical machine translation". In: *EMNLP*. 2014.

[18] Justin Gilmer et al. "Neural message passing for quantum chemistry". In: *ICML*. 2017.

[19] Weihua Hu et al. "Open graph benchmark: Datasets for machine learning on graphs". In: *NeurIPS*. 2020.

[20] Prithviraj Sen et al. "Collective classification in network data". In: *AI magazine*. 2008.

[21] Oleksandr Shchur et al. "Pitfalls of graph neural network evaluation". In: *arXiv preprint arXiv:1811.05868*. 2018.

[22] Marinka Zitnik and Jure Leskovec. "Predicting multicellular function through multi-layer tissue networks". In: *Bioinformatics*. 2017.

[23] Christopher Morris et al. "Tudataset: A collection of benchmark datasets for learning with graphs". In: *arXiv preprint arXiv:2007.08663*. 2020.

[24] Geoff Boeing. "OSMnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks". In: *Computers, Environment and Urban Systems*. 2017.

[25] Miltiadis Allamanis. "The adverse effects of code duplication in machine learning models of code". In: *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 2019.

[26] Zhenqin Wu et al. "Moleculenet: a benchmark for molecular machine learning". In: *Chemical science*. 2018.

[27] Jiaxuan You, Rex Ying, and Jure Leskovec. "Position-aware graph neural networks". In: *ICML*. 2019.

[28] Aditya Grover and Jure Leskovec. "node2vec: Scalable feature learning for networks". In: *ACM SIGKDD*. 2016.

[29] Stuart Russell and Peter Norvig. *Artificial intelligence: a modern approach*. 2002.

[30] Sandip Aine et al. "Multi-heuristic A*". In: *The International booktitle of Robotics Research*. 2016.

[31] Edo Cohen-Karlik, Avichai Ben David, and Amir Globerson. "Regularizing towards permutation invariance in recurrent models". In: *NeurIPS*. 2020.

[32] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *NeurIPS Deep Learning Workshop*. 2013.

[33] Pieter-Tjerk De Boer et al. "A tutorial on the cross-entropy method". In: *Annals of operations research*. 2005.

[34] E. Rohmer, S. P. N. Singh, and M. Freese. "Coppeliasim (formerly v-rep): a versatile and scalable robot simulation framework". In: *IROS*. 2013.

[35] Daniel Lenton et al. "Ivy: Templated deep learning for inter-framework portability". In: *arXiv preprint arXiv:2102.02886*. 2021.

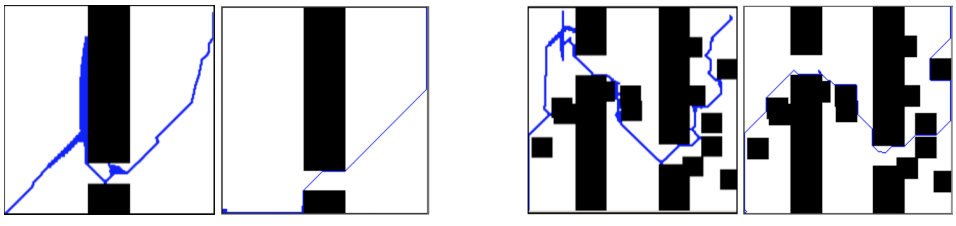
[36] Petar Velickovic et al. "Graph attention networks". In: *ICLR*. 2018.

[37] Guohao Li et al. "Deepergcn: All you need to train deeper gcns". In: *arXiv preprint arXiv:2006.07739*. 2020.

[38] Petar Velickovic et al. "Neural execution of graph algorithms". In: *ICLR*. 2020.

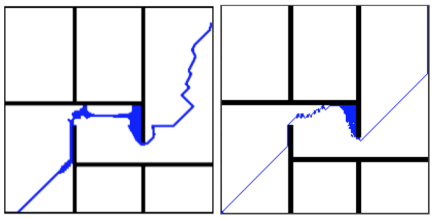
[39] Petar Velickovic. *TikZ*. `https://github.com/PetarV-/TikZ`, last accessed on 01/6/21.

A SaIL and PHIL qualitative comparisons

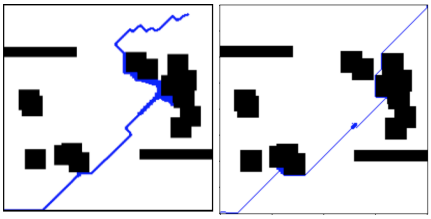


(a) Shifting gaps (SaIL left, PHIL right)

(b) Gaps+Forest (SaIL left, PHIL right)



(c) Mazes (SaIL left, PHIL right)



(d) Forest (SaIL left, PHIL right)

Figure 2: In each image pair of this figure, we provide a qualitative comparison with the SaIL method. In particular, we show comparisons on the *Shifting gaps*, *Gaps+Forest*, *Mazes*, and *Forest* datasets. We can observe that PHIL (right) learns the appropriate heuristics for the given dataset and makes fewer redundant expansions than SaIL (left).

B Diverse graph experiments

In this experiment, our goal is to demonstrate the general applicability of PHIL to various graphs. We train PHIL on 4 different groups of graph datasets: citation networks, biological networks, abstract syntax trees (ASTs), and road networks. We use the same graph for citation networks and road networks for training and evaluation, and we use 100 random v_s, v_g pairs for testing. In the case of biological networks and ASTs, we usually have train/validation/test splits of $80/10/10$, and in the case of the OGB [19] datasets, we use the provided splits.

	Dataset	$\lvert\mathcal{D}\rvert$	$\lvert\bar{\mathcal{V}}\rvert$	$\lvert\bar{\mathcal{E}}\rvert$	SL	A*	h_{euc}	BFS	PHIL
Citation Networks	Cora (Sen *et al.* [20])	1	2,708	5,429	2.201	2.067	1.000	4.001	**0.530**
	PubMed (Sen *et al.* [20]))	1	19,717	44,338	2.157	2.983	1.000	3.853	**0.745**
	CiteSeer (Sen *et al.* [20]))	1	3,327	4,732	1.636	1.487	1.000	2.190	**0.599**
	Coauthor (cs) (Schur *et al.* [21])	1	18,333	81,894	1.571	1.069	1.000	2.820	**0.835**
	Coauthor (physics) (Schur *et al.* [21])	1	34,493	247,962	4.076	1.081	1.000	4.523	**0.964**
Biological Networks	OGBG-Molhiv (Hu *et al.* [19])	41,127	25.5	27.5	1.086	1.065	**1.000**	1.267	1.016
	PPI (Zitnik *et al.* [22])	24	2,372.67	34,113.16	0.772	0.831	1.000	5.618	**0.658**
	Proteins (Full) (Morris *et al.* [23])	1,113	39.06	72.82	0.995	0.997	1.000	2.645	**0.978**
	Enzymes (Morris *et al.* [23])	600	32.63	62.14	1.073	1.007	1.000	1.358	**0.896**
ASTs	OGBG-Code2 (Hu *et al.* [19])	452,741	125.2	124.2	1.196	1.013	**1.000**	1.267	1.219
Road Networks	OSMnx - Modena (Boeing [24])	1	29,324	38,309	2.904	3.085	1.000	3.493	**0.489**
	OSMnx - New York (Boeing [24])	1	54,128	89,618	39.424	36.529	1.000	63.352	**0.962**

Table 3: Comparison of PHIL with baseline approaches on 4 groups of datasets: citation networks, biological networks, abstract syntax trees, and road networks. We can observe that, on average across all datasets, PHIL outperforms the best baseline per dataset by 13.9%. Discounting the OGBG datasets, this number becomes 16.24%.

Discussion. The results presented in Table B suggest that PHIL can learn superior search heuristics compared with baseline methods, outperforming top baselines per dataset in terms of visited nodes during a search by 13.9% on average. Two datasets where PHIL fell short compared to other baselines are the *OGBG-Molhiv* and *OGBG-Code2* datasets. The *OGBG-Code2* dataset adopts a *project split* [25] and OGBG-Mohliv adopts a *scaffold split* [26], both of which ensure that graphs of different structure are present in the training & test sets. Although PHIL improved upon uninformed search (BFS) in the OGB datasets, structural graph consistency is explicitly discouraged in the above-mentioned OGBG splits. Without the OGBG datasets, PHIL improves on the top baselines per dataset by 16.24% on average, and upon the Euclidean node feature heuristic (h_{euc}) by 18.04%. Note

Algorithm 2: PHIL— Sequential Heuristic Training

Obtain hyperparameters T, β_0, N, m, t_τ;
Initialize $\mathcal{D} \leftarrow \emptyset$, h_{θ_1};
for $i = 1, \ldots, N$ **do**
  Sample $\mathcal{G} \sim P_{\mathcal{G}}$;
  Sample $v_s, v_g \sim P_{v_{sg}}$;
  Set $\beta \leftarrow \beta_0^i$;
  Set mixture policy $\pi_{mix} \leftarrow (1-\beta) * \pi_{\theta_i} + \beta * \pi^*$;
  Collect m trajectories τ_{ij} as follows;
  for $j = 1, \ldots, m$ **do**
    Sample $t \sim \mathcal{U}(0, ..., T - t_\tau)$;
    Roll-in t time steps of $\pi_{\theta i}$ to obtain z_t and new state $s_t = (\mathcal{V}^0_{seen}, \mathcal{V}^0_{fringe}, \mathcal{V}^0_{unseen})$;
    Roll-out trajectory τ_{ij} as follows;
    for $k = 1, \ldots, t_\tau$ **do**
      Update s_{t+k-1} using π_{mix} to get new state s_{t+k} and new fringe state $\mathcal{V}^k_{fringe}$;
      Obtain new fringe nodes $\mathcal{V}_{new} = \mathcal{V}^k_{fringe} \setminus \mathcal{V}^{k-1}_{fringe}$;
      Update trajectory $\tau_{ij} \leftarrow \tau_{ij} \cup \{(\mathcal{V}_{new}, h^*(s_{t+k}, \mathcal{V}_{new}, v_g))\}$;
    Update dataset $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\tau_{ij}, z_t)\}$ or $\mathcal{D} \cup \{(\tau_{ij}, 0)\}$;
  Train h_{θ_i} using TBTT on each $\tau \in \mathcal{D}$ to get $h_{\theta_{i+1}}$;
return *best performing* h_{θ_i} *on validation*;

that we trained PHIL up to $N = 60$ iterations, which means that it only encountered a small subset of the pathfinding problems in the single graph setting, which means that PHIL had to generalize to learn useful heuristics. Even in Cora, the $|\mathcal{D}| = 1$ dataset with least number of nodes, PHIL observed roughly $6,000$ node distances during training, which is less than 0.2% of total distances in the Cora graph.

C PHIL algorithm pseudocode

In Algorithm 2, we provide a pseudocode of PHIL's imitation learning algorithm.

D Discussion, Limitations, and Future Work

Injection from nodes and edges to features If multiple graph nodes or edges have the same features, our heuristic learning method is challenging to apply. To ensure that PHIL has a constant time complexity, we bound the number of neighbouring nodes used for graph convolutions around new fringe nodes, and do not perform any graph convolutions on goal nodes. However, if multiple graph nodes have the same features, or they perhaps do not have any features at all, we may need to perform operations that are not constant in the size of the graph, such as sampling anchor nodes as in position-aware GNN [27], or collecting more expressive Node2Vec [28] style features. For use cases where injections from nodes & edges to features are hard to guarantee, we encourage practitioners to increase n or potentially perform multiple convolutions on fringe and goal nodes.

Restricted fringe evaluation PHIL only evaluates new fringe nodes which are obtained after expanding a node. In practice, this means that once PHIL assigns a heuristic value to a node, the value is never updated. While this approach is favorable in terms of the time-complexity of heuristic computations, it does not allow PHIL to re-evaluate potentially promising nodes for expansion, based on its updated belief about the POMDP state. We believe that methods that predict the features of promising nodes to expand combined with locality-sensitive hashing or approaches that incorporate node value uncertainty present promising avenues for future work.

Solutions not necessarily optimal For the A* search algorithm to find optimal solutions, the used heuristics needs to be *admissible* [29]. In our approach, we do not guarantee the trained heuristics to be admissible, which means that when combined with A*, PHIL would not guarantee optimal

final solutions. On the other hand, our approach is concerned with finding feasible solutions as quickly as possible, which is motivated by possible applications in Section 1. As in [1], our learned heuristics can be easily incorporated into a framework such as multi-heuristic A* [30], where any number of inadmissible heuristics can be used with a single admissible heuristic, and the final solution cost sub-optimality is bounded. An interesting avenue for future work would be to design heuristic learning loss functions that discourage models from over-estimating heuristic values.

Full memory permutation invariance As noted in Section 3, our memory module is invariant to the permutation of nodes in $\mathcal{V}_{new}$. However, due to how the GRU module is applied, we do not guarantee that that the memory is permutation invariant with respect to the sequence in which nodes are expanded, or equivalently the sequence of $\mathcal{V}_{new}$ sets. It could be desirable to guarantee such permutation invariance, as the observations are still nodes and edges of a graph, which may not contain sequential inductive biases. Recent work by Cohen *et al.* [31] shows that a simple regularization trick can help efficiently train permutation invariant RNNs. It would be interesting to explore in which cases does full permutation invariance improve PHIL's performance.

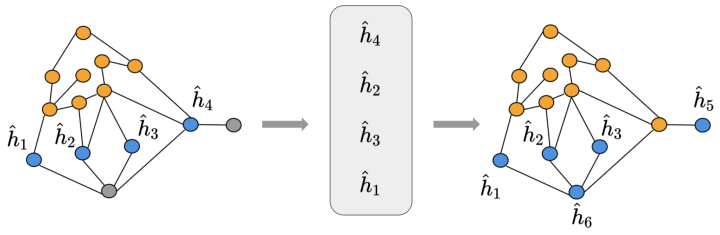
Directed graphs As one may notice, most of the examples in this work include graphs that are undirected. The main reason for this is that once we have directed edges in a graph, it may happen that a particular node does not have a path toward the goal, which means that the oracle cost would be effectively undefined. One option for avoiding this issue is adding parallel backward edges during training, ensuring that paths to goals always exist (assuming that the start and goal nodes are parts of the same connected component). This way, PHIL is correspondingly penalised for expanding a node that does not immediately lie on a path to the goal.

Ethical considerations Search algorithms with heuristics can be used within unethical systems. However, our work is not tailored for any particular use cases, and hence we do not believe that it has clear direct negative consequences.

E Practical implementation details

As noted in our abstract, at test time, the heuristic function obtained from PHIL can be directly used as a heuristic in an algorithm such as A* or greedy best-first search. In practice, this means that we maintain a priority queue of nodes and distances predicted by the PHIL heuristic and greedily expand the nodes which are predicted to be closest to the goal, as seen in Figure 3.

To ensure fast training in Algorithm 2, we maintain two priority queues, one for PHIL and one for the oracle heuristic. On every expansion, we update both the PHIL queue and oracle queue. This way, probabilistically blending the two greedy policies comes down to either popping from the PHIL queue or the oracle queue.



Candidate Priority Queue

Figure 3: PHIL is used as a heuristic to predict the distances of each node, similarly as a heuristic function in best-first search. This figure illustrates a single queue 'pop' operation.

F Training, Baselines, Hyperparameters, and Datasets

Baselines In our experiments we use a range of both *learning-based* and *classical baselines*. Our *baselines* include:

- **SaIL** [1]: State-of-the-art imitation learning-based approach for learning search heuristics on grid graphs.
- **Supervised Learning** (SL): An MLP trained to predict distances between nodes conditioned on node features.
- **Deep Q Learning** (QL) [32]: A DQN agent trained to explore the graphs such that $|\mathcal{V}_{seen}|$ is minimized. The agent receives a negative reward of -1 until the goal node is reached in each episode.
- **Cross Entropy Method - Evolutionary Strategies** (CEM) [33]: Derivative free optimization of h_θ using evolutionary strategies. As explained in [1], the initial population of policies is sampled using a batch size of 40. Then, each policy is evaluated on 5 graphs and assigned a score based on the number of expanded nodes for the fitness function. After computing the fitness function, 20% of best-performing policies are selected to be a part of the next population.
- **Multi-heuristic A*** (MHA*)[30]: Multi-heuristic A* using the Euclidean, Manhattan, and d_{obs} heuristics in a round-robin fashion, where d_{obs} is the distance of the closest uncovered obstacle for a given node.
- **A* search** (A*): A* search algorithm using a Euclidean heuristic function on the node features.
- **Greedy best-first search** (h_{man}, h_{euc}): Greedy best-first search using Manhattan and Euclidean heuristics, respectively.
- **Breadth-first-search** (BFS): Vanilla breadth-first-search without any heuristic.

For more information about the *learning-based baselines*, we refer the reader to Bhardwaj *et al.* [1], where these baselines are described in detail.

Datasets. Here, we provide more details about the datasets used in our experiments. For more information about datasets used in our *Heuristic search in grids* experiments, we refer the reader to Bhardwaj and Choudhury *et al.* [1].

In our experiments using diverse graphs (Appendix B), we use the node & edge features provided in each dataset for training and testing. The only two exceptions to this are the PPI and the OSMnx datasets. As discussed in Appendix D, we added node labels (i.e., 120 dimensional label vectors) as features in the PPI dataset because the default node features were not unique. In the OSMnx networks, we did not use all the provided node and edge features, rather only the geographical node coordinates.

In static graphs (such as the OSMnx networks), it may be helpful to augment the graph with expressive features that could be useful for heuristic computations, such as eigenvectors of the graph Laplacian matrix. In non-static graphs, these operations would typically be too expensive to re-compute on each pathfinding attempt. Note that computing more expressive features such as betweenness or percolation centrality have higher time complexities than computing shortest path distances between all pairs of nodes. Due to their time complexity, these features are impractical pre-compute, though they would likely lead to superior search heuristics.

In Section 4, we built our environment using the CoppeliaSim simulator [34], and the Ivy framework by Lenton *et al.* [35]. Figure 4 presents the environments which we refer to as *room adversarial* and *room simple* in Table 2. The *room simple* environment is equivalent to the Ivy drone demo environment — a single room with a table surrounded by chairs in the middle of the room and various furniture close to the walls. The main difference between *room simple* and *room adversarial* is that *room simple* only contains a single table in the middle of the room. In contrast, in *room adversarial*, three tables span to a wall, this was creating a bottleneck for naive heuristics. Note that the drones are not allowed to fly under furniture to make the environment more challenging.

Hyperparameters in diverse graphs With regards to the diverse graph experiments (Appendix B), there are in-practice three parameters that we can tune for each dataset: n, t_τ T. Recall that the hyperparameter n is the maximal size of the 1-hop neighborhood around the new fringe nodes sampled during training. We found that a rule of thumb of setting n to the minimum of 8 and the average node degree in the given graph dataset worked reasonably well. In practice, n should be tuned to optimize the search effort & wall-clock performance trade-off. In terms of T, it is advisable

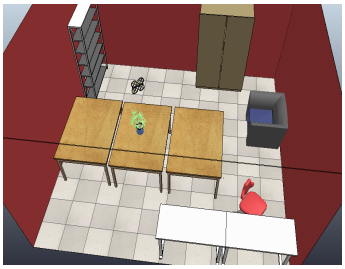
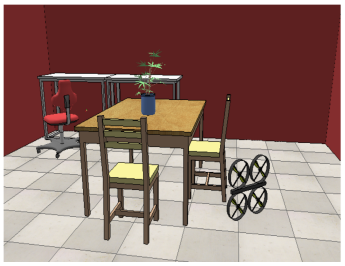



Figure 4: We present two CoppeliaSim environments used for our UAV flight experiments. On the left, we have the *adversarial room environment*, on the right, we present the *room simple* environment.

to set it such that the algorithm can reach target nodes during training. Hence, we found that setting T roughly to the largest graph diameter is a suitable choice. In terms of t_τ, we train each sequence using the stored old rolled-in states, and for larger graphs ($> 10,000$ nodes), we use $t_\tau = 128$, while for smaller graphs (between $10,000$ and $5,000$ nodes), we use $t_\tau = 64$, and otherwise we use a rule of thumb of taking $t_\tau \approx T * 0.2$.

Grid graphs & drone flight hyperparameters In our *Heuristic search in grids*, we use $T = 256$, $t_\tau = 32$, $\beta_0 = 0.7$, $n = 8$. Finally, in our drone flight experiments, we use the same configuration as in the grid graph experiments. The only differences are that we set $n = 4$, $t_\tau = 64$, and we randomize start and goal nodes both during training and testing.

Optimization PHIL is generally trained using the Adam optimizer with a learning rate of 0.001 and a batch size of 32. Once we sample $t \sim \mathcal{U}(0, \ldots, T - t_\tau)$ for a roll-in, it can happen that the target is reached in less than $t + t_\tau$ steps. In such cases, we continue the sequence until we reach t_τ steps or all the graph nodes are explored, in which case we end the episode. Another approach would be to end episodes once the goal node is reached. In practice, we did not find significant performance differences between the two methods.

G Ablation studies

For the ablation studies, we use three, 4-connected versions of down-scaled datasets provided in Bhardwaj *et al.* [1]: *Gaps+Forest*, *Forest*, and *Alternating gaps*. We downscale each dataset $5\times$, to dimensions 40×40. While the *Gaps+Forest* dataset presents a more challenging environment with multiple planning bottlenecks and obstacles, *Forest* and *Alternating gaps* are simpler environments with more straightforward heuristics. Figure 5 present example graphs from the down-scaled datasets *Gaps+Forest*, *Forest*, and *Alternating gaps*.

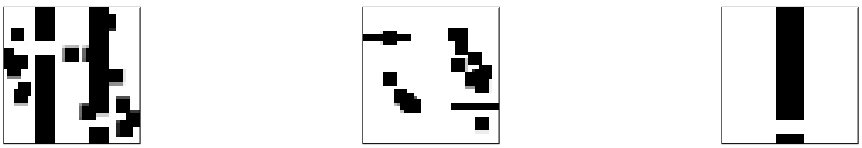


Figure 5: This figure presents the down-scaled datasets used for the ablation experiments. We have *Gaps+Forest* (left), *Forest* (centre), and *Alternating gaps* (right).

In each ablation experiment, unless stated otherwise, we use a batch size of 8, the Adam optimizer with a learning rate of 0.01, and up to 20 roll-in steps. The MLP in PHIL has 3 layers of width 128, *LeakyReLU* activations, and the memory state has $d = 64$ dimensions. 2D grid coordinates are used as node features, similarly as in *Heuristic search in grids*. We report the number of explored nodes with respect to A* or with respect to the optimal number of expansions (i.e., the shortest path distances between nodes). All experiments are performed across 3 random seeds, and standard deviations are used for error bars.

G.1 Zeroed-out states vs. Rolled-in states

There is a trade-off between using rolled-in states for downstream training using backpropagation through time (i.e., storing z_t after each roll-in) and using zeroed-out states (i.e., storing zeroed-out initial states). Namely, past rolled-in states z_t are out-of-distribution for optimized versions of h_θ, but PHIL may use these embeddings for inferring initial node distances because z_t contains information about the rolled-in graph. On the other hand, zeroed-out states are always in-distribution for h_θ, but the algorithm is constrained to start from an initial state of $\vec{\mathbf{0}}$ in regions where this may never be the case at test time. Our goal is to gain insight into when one method is preferable over to the other in this ablation.

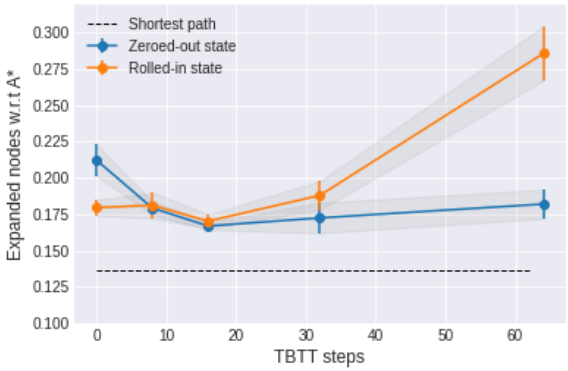


Figure 6: This plot illustrates the performance of PHIL with zeroed-out initial states vs. stored rolled-in initial states during training across multiple TBTT steps.

We train two versions of PHIL, one with zeroed-out initial states for TBTT on each trajectory (PHIL-zero), and one with initial states stored from roll-ins of the past versions of PHIL (PHIL-roll). Both versions are trained for $t_\tau = 0, 8, 16, 32, 64$ TBTT steps using the *Gaps+Forest* graph dataset. Figure 6 illustrates the performance progress across TBTT steps of both of these approaches. Firstly, we observe that up to around $t_\tau = 16$ steps; both approaches positively benefit from performing more backpropagation steps through time, reinforcing that the memory module brings overall performance benefits. Further, we may see the performance difference of PHIL-roll and PHIL-zero at 0 steps. In this region, PHIL-roll outperforms PHIL-zero by about 3%, which is following the intuition of the rolled-in states containing useful information about the embeddings of rolled-in graphs. Secondly, we can notice that the performance of PHIL-zero plateaus after 16 steps. This plateau suggests that the graph may not contain useful information for inferring node distances beyond 16 backpropagated steps. Finally, the performance of PHIL-roll decreases much steeper than that of PHIL-zero once it starts deteriorating after 16 steps, which could mean that there is some form of error compounding once PHIL makes predictions from a wrongly initialised state during training.

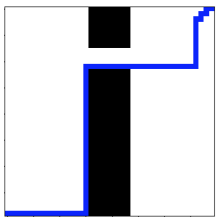


Figure 7: This figure illustrates PHIL using the DeeperGCN (*softmax*) GNN learning the optimal heuristic strategy in the *Alternating gaps* dataset.

Conclusion The main takeaway of this ablation experiment is that if one would like to perform only a few TBTT steps (i.e., low t_τ), using rolled-in states will likely provide some performance benefits. On the other hand, as t_τ increases, it is preferable to use zeroed-out initial states. Practitioners can determine this choice on a per-problem basis.

G.2 Effect of GNN choice

In this ablation we assess what effects do different graph neural networks have on the performance of PHIL. Namely, we train the PHIL-zero from the previous ablation, using $t_\tau = 16$, $n = 4$ and five different GNNs: GAT (Veličković *et al.* [36]), MPNN (*max*), MPNN (*sum*) (Gilmer *et al.* [18]), DeeperGCN (*softmax*), and DeeperGCN (*power*) (Li *et al.* [37]). We report the ratio of explored nodes with respect to A* in Table 4.

GNN	Alternating gaps	Forest	Gaps+Forest
GAT	0.077 ± 0.0110	0.065 ± 0.0013	**0.154 ± 0.0112**
MPNN (*max*)	0.071 ± 0.0834	**0.064 ± 0.0004**	0.158 ± 0.0143
MPNN (*sum*)	0.095 ± 0.0487	0.07 ± 0.0085	0.187 ± 0.0135
DeeperGCN (*softmax*)	**0.069 ± 0.0008**	**0.064 ± 0.0030**	0.164 ± 0.0113
DeeperGCN (*power*)	0.076 ± 0.0027	0.07 ± 0.0101	0.19 ± 0.0037

Table 4: This table presents the results obtained using different graph convolutions in the three graph datasets from the ablation study. We can observe that maximisation-based aggregation performs better on average, while attention can provide performance benefits in the challenging *Gaps+Forest* graphs.

In general, we find that maximisation-based aggregation approaches tend to perform better than other approaches by 1.57% on average. Note that this comparison only includes the MPNN and DeeperGCN GNNs. DeeperGCN (*softmax*) achieves the best results on both the *Alternating gaps* and *Forest* datasets. Further, using DeeperGCN (*softmax*), PHIL learned the optimal strategy for finding the goal in the *Alternating gaps* dataset, which is to follow the path along the bottleneck wall, as seen in Figure 7.

GAT outperforms other approaches on the *Gaps+Forest* dataset, which is also the most complex of the three datasets. This finding suggests that forms of attention could be useful for learning heuristics in more complex graphs. In Figure 8, we can see a side-by-side comparison of GAT, DeeperGCN (*softmax*), and MPNN (*sum*) in the *Gaps+Forest* graph dataset. We may observe that the GAT-based and DeeperGCN-based heuristics find strategies that are close to optimal, while MPNN (*sum*) performs a similar strategy, but with slightly more expansions.

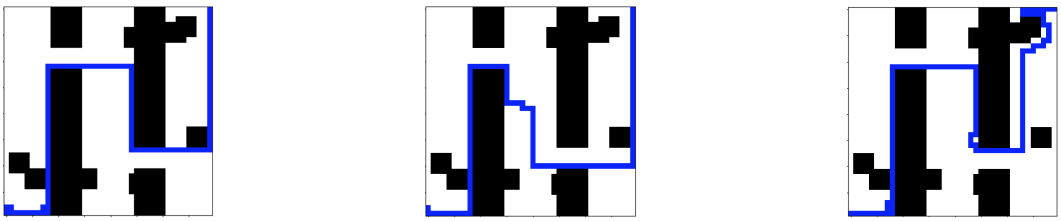


Figure 8: We compare the solutions of PHIL trained with a GAT (left), DeeperGCN (*softmax*) (middle), and MPNN (*sum*) (right). Results demonstrate that MPNN (*sum*) makes several redundant expansions while the other two methods expand paths close to optimally.

Conclusion The GNN ablation demonstrates that maximisation-based aggregation performs better, with an average reduction of explored nodes by 1.57%. This finding is consistent with Veličković *et al.* [38], where GNNs are applied to execute graph algorithms. While the problem solved by Veličković *et al.* [38] is different, its nature is similar: train a GNN to *select* which node to explore next in order to imitate a reference graph algorithm. In PHIL, scores are assigned to nodes rather than nodes being selected, but the downstream operation is node selection. Further, these experiments also suggest that attention can be helpful in more complex graphs, with GAT outperforming other approaches in the *Gaps+Forest* graphs.

G.3 Increasing neighborhood size

As explained in the approach (Section 3), for each new fringe node we uniformly sample an $n \in \mathbb{N}_{\geq 0}$ bounded neighborhood of nodes, which we then use for graph convolutions. We hypothesized that with increasing n, the performance of PHIL will improve. In Figure 9, we validate this hypothesis

on *Alternating gaps*, *Forest*, and *Gaps+Forest* datasets, by gradually setting $n = 0, 1, 2, 4$. We train PHIL using a DeeperGCN (*softmax*) graph convolution, $t_\tau = 16$, and with otherwise the same set of hyperparameters as in the *Zeroed-out states vs. Rolled-in states* (Appendix G.1) experiment. In Figure 9, we report the explored node ratio of each method with respect to the optimal number of explored nodes, that is, the shortest paths between start & goal pairs.

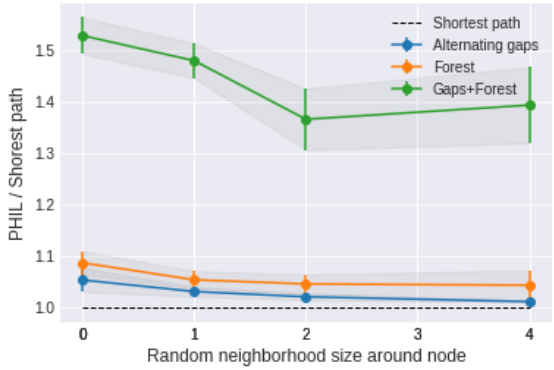


Figure 9: This plot illustrates the performance of PHIL with increasing $n = |\mathcal{N}_i|$ through $0, 1, 2, 4$ for each evaluated fringe node i.

Conclusion In Figure 9, we may observe that on all datasets the performance of PHIL improves with increasing $n = |\mathcal{N}_i|$ until $n = 2$, after which it plateaus. These results suggest that PHIL can benefit from additional nodes sampled from the neighbourhoods of evaluated fringe nodes, even if this sampling is performed uniformly at random.

G.4 Increasing memory capacity

In the final ablation experiment, we consider what effects changing the capacity of memory (d) has on the overall performance of PHIL. We alter $d = 1, 16, 32, 64, 128, 256$ across all datasets. In Figure 10, the ratio of explored nodes is presented with respect to the shortest path length, which is the optimal baseline.

Focusing on the *Gaps+Forest* dataset, in Figure 10 we can observe that the performance of PHIL generally improves until about $d = 32$ by approximately 40% with respect to $d = 1$, after which it starts getting worse. Hence, we can conclude that additional memory capacity can be helpful for PHIL to learn representations better suited for inferring distances of newly added fringe nodes. Note that we trained PHIL for a fixed number of iterations ($N = 36$) in all cases, which means that the decrease in performance after $d = 32$ could also be due to the GRU module having more parameters to optimise, which may take longer to converge. However, it could also easily be that the memory module starts overfitting to samples in the aggregated dataset during training. In the case of the simpler *Alternating gaps* and *Forest* datasets, the differences between different amounts of memory capacity are marginal. These findings are supported by approaches such as [1] achieving good performances in simpler environments. By implication, performance decrease in the *Gaps+Forest* dataset for larger values of d is more likely due to overfitting than optimisation difficulties.

Conclusion Additional memory capacity is crucial for PHIL to learn useful representations in more challenging graphs, while the importance of additional memory decreases as the graphs are simpler. However, a certain amount of overfitting is observed for larger values of d, which means that d should be tuned on a per problem basis.

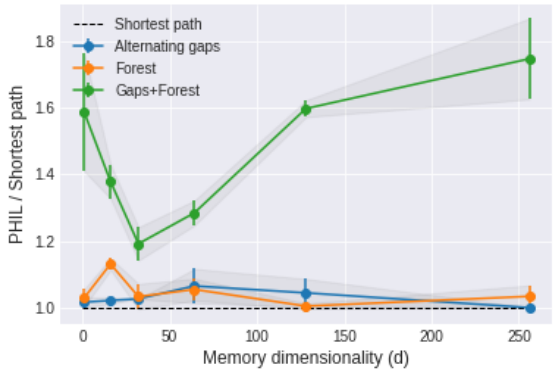


Figure 10: This plot illustrates the performance of PHIL with increasing d through $1, 16, 32, 64, 128, 256$ on the three ablation datasets.

H Architecture

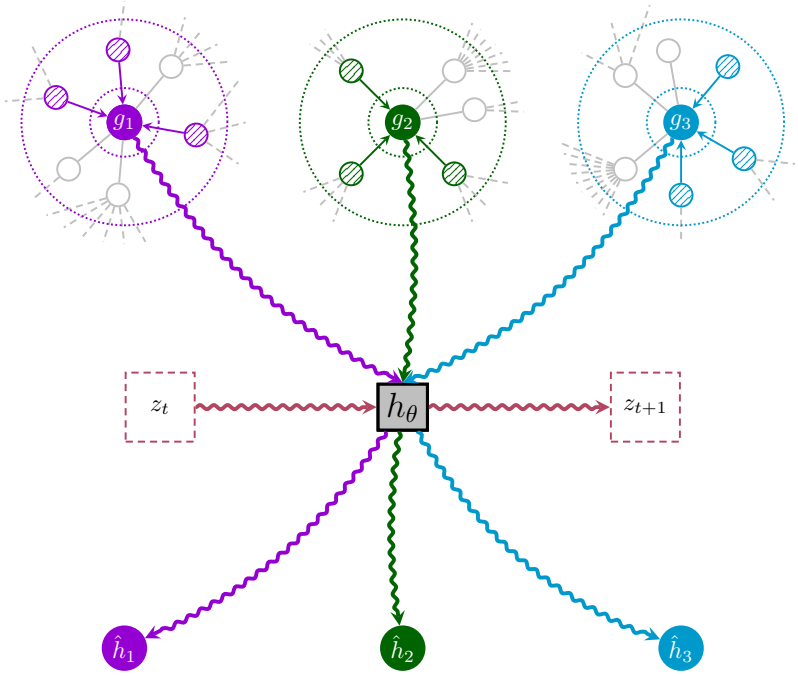


Figure 11: This figure illustrates the computation performed in a single forward pass of h_θ. In the case of this figure, we would have $|\mathcal{V}_{new}| = n = 3$, with g_i representing the convolved embeddings, and $\hat{h}_i$ the output heuristic values. Horizontally, we illustrate the update of memory z_t to z_{t+1}. This figure is adapted from [39].

I Modena use-case

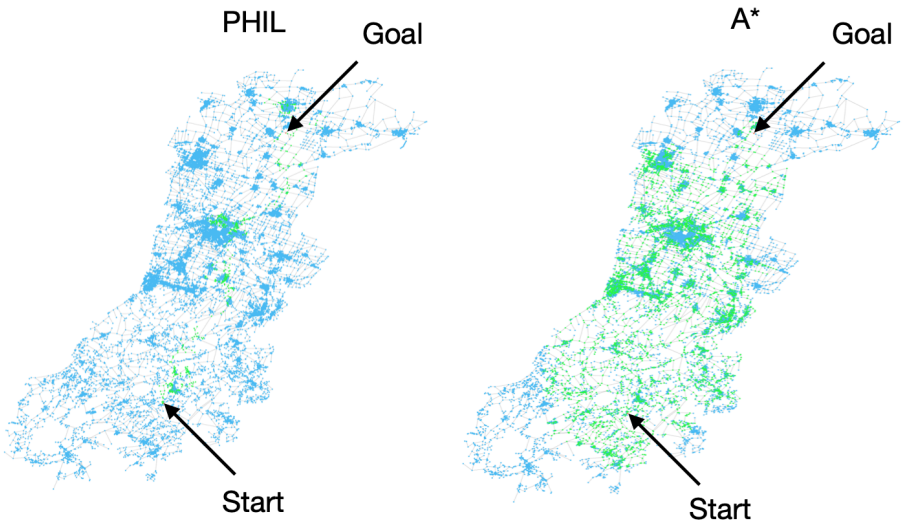


Figure 12: This figure illustrates the road network of Modena, Italy. We contrast the search effort of PHIL (left) and A* (right). We can observe that PHIL expands (shown in green) considerably fewer nodes searching for the goal v_g.