

A Discussion, Limitations, and Future Work

Injection from nodes and edges to features. As mentioned in Section 5.2, 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 [48], or collecting more expressive Node2Vec [59] style features. Since the time complexity of these methods are relatively high (e.g. position-aware GNN’s time complexity is $O(|V|^2 \log^2 |V|)$ while the Dijkstra algorithm only runs in $O(|V| \log |V| + |E|)$), we do not use them unless necessary. 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. As explained in Section 4, 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 a best-first search algorithm to find optimal solutions, the used heuristics needs to be *admissible* [60]. In our approach, we do not guarantee the trained heuristics to be itadmissible, which means that when combined with best-first search, PHIL would not guarantee optimal final solutions. On the other hand, our approach is concerned with finding *satisficing* 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* [61], 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 4, 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.* [62] 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.

B 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 7.

To ensure fast training in Algorithm 1, 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.

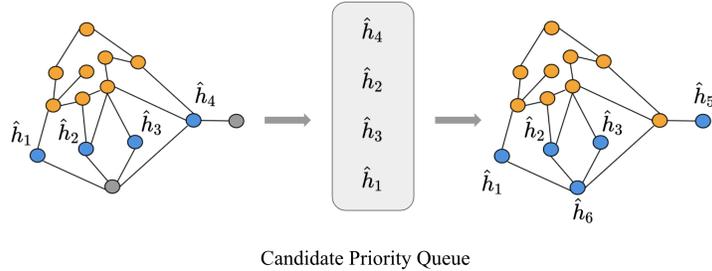


Figure 7: PHIL is used as a heuristic to predict the distances of each node, similarly as a heuristic function in best-first search. This figure illustrates a single queue ‘pop’ operation.

C Training, Baselines, Hyperparameters, and Datasets

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

- **Neural A*** [28]: Neural A* learns a differentiable version of A* search on two-dimensional grid graphs. It cannot be straightforwardly adapted to general graphs, which is why we use it as a baseline solely in Section 5.1.
- **SAIL** [1]: Imitation learning-based approach for learning search heuristics on grid graphs. SAIL is adapted in Section 5.2 to not include robotics domain specific features presented in [1], as these are not possible to compute for general graphs.
- **Supervised Learning (SL)**: An MLP trained to predict distances between nodes conditioned on node features. In Sections 5.2, Sections 5.3 the nodes are sampled from the graph uniformly at random. In Section 5.1, an oracle policy is rolled out, and data points are collected by sampling random actions during the roll-out as in SAIL. This is because in Section 5.1, the start-goal distribution is not uniform.
- **Deep Q Learning (QL)** [63]: 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)** [64]: 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.
- **Best-first width search (BFWS)** [12]: BFWS adapts greedy best-first search with a generic search history-based novelty metric, which allows it to escape search plateaus and explore relevant nodes. We implement BFWS in experiments from Section 5.1 and 5.3. In Section 5.2, we extend BFWS with boundary-extension features [13], which allows us to apply it to continuous feature spaces.
- **Multi-heuristic A* (MHA*)**[61]: 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. This baseline is only used in Section 5.1, where all of these heuristics are well-defined.
- **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.

Datasets. Here, we provide more details about the datasets used in Section 5.2 and Section 5.3. For more information about datasets used in Section 5.1, we refer the reader to Bhardwaj and Choudhury *et al.* [1].

In our experiments using diverse graphs, as noted in Section 5.2, we use the node & edge features provided in each dataset for training and testing. The three exceptions to this are the PPI, OSMnx datasets, and citation networks. As discussed in Appendix A, 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. We further augmented the OSMnx node features with their degree centrality and included Laplacian, modularity, and Bethe Hessian edge features. For citation networks, we only used the first 128 features to make the search problem more challenging.

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 5.3, we built our environment using the CoppeliaSim simulator [57], and the Ivy framework by Lenton *et al.* [58]. Figure 8 presents the environments which we refer to as *room adversarial* and *room simple* in Table 3. 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.

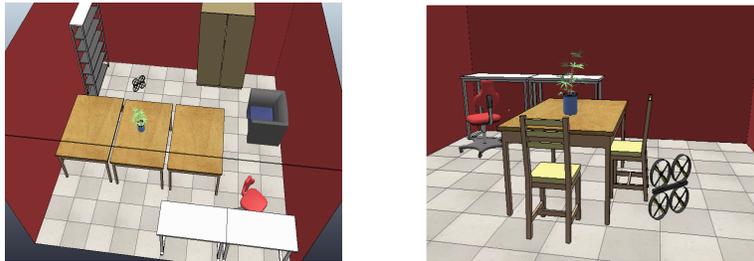


Figure 8: We present two CoppeliaSim environments used for our UAV flight experiments. On the left, we have the *adversarial room environment*, on the right, we present the *room simple* environment.

Hyperparameters in diverse graphs. With regards to the diverse graph experiments (Section 5.2), 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 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 grid graph experiments (Section 5.1), 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 $t \sim \mathcal{U}(0, \dots, T - t_\tau)$ 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.

D 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 9 present example graphs from the down-scaled datasets *Gaps+Forest*, *Forest*, and *Alternating gaps*.

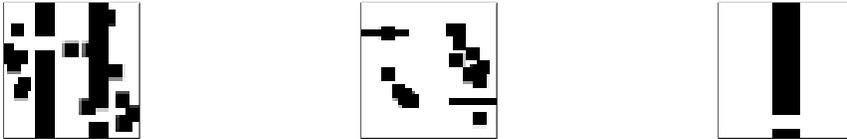


Figure 9: This figure presents the down-scaled datasets used for the ablation experiments. We have *Gaps+Forest* (left), *Forest* (centre), and *Alternating gaps* (right).

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 Section 5.1. 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.

D.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 $\vec{0}$ 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.

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 10 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

could be useful for learning heuristics in more complex graphs. In Figure 12, 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.

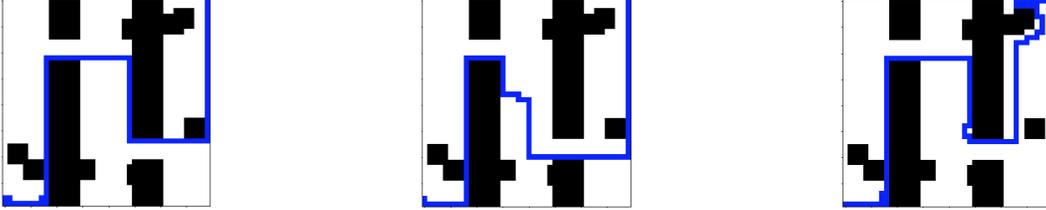


Figure 12: We compare the solutions of PHIL trained with a GAT (left), DeeperGCN (*softmax*) (middle), and MPNN (*sum*) (right). Results demonstrate that MPNN (*sum*) makes several redundant expansions while the other two methods expand paths close to optimally.

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.* [66], where GNNs are applied to execute graph algorithms. While the problem solved by Veličković *et al.* [66] 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.

D.3 Increasing neighborhood size

As explained in the approach (Section 4), 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 13, 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 D.1) experiment. In Figure 13, 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.

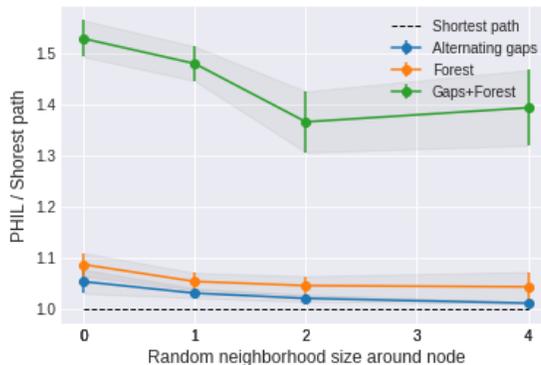


Figure 13: This plot illustrates the performance of PHIL with increasing $n = |\mathcal{N}_i|$ through 0, 1, 2, 4 for each evaluated fringe node i .

Conclusion. In Figure 13, 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.

D.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 14, 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 14 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 SAIL [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.

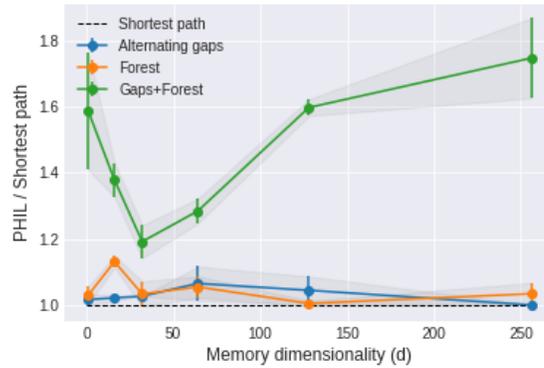


Figure 14: This plot illustrates the performance of PHIL with increasing d through 1, 16, 32, 64, 128, 256 on the three ablation datasets.

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.

E Architecture

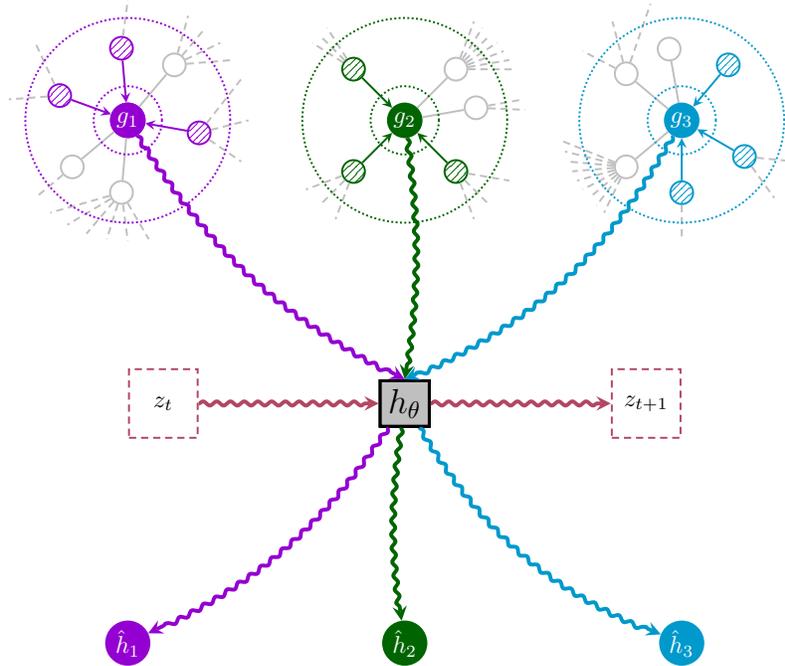


Figure 15: This figure illustrates the computation performed in a single forward pass of h_θ . In the case of this figure, we would have $|\mathcal{V}_{new}| = n = 3$, with g_i representing the convolved embeddings, and \hat{h}_i the output heuristic values. Horizontally, we illustrate the update of memory z_t to z_{t+1} . This figure is adapted from [67].

F Qualitative Examples

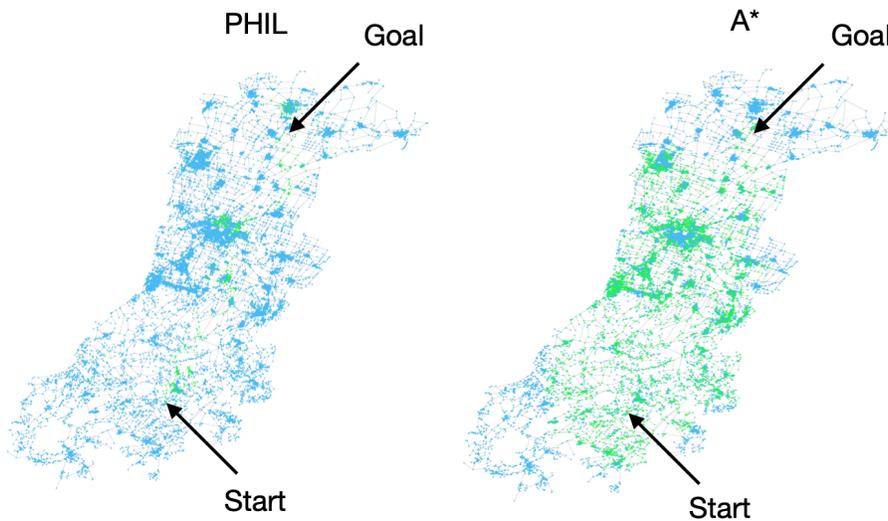


Figure 16: This figure illustrates the road network of Modena, Italy. We contrast the search effort of PHIL (left) and A* (right). We can observe that PHIL expands (shown in green) considerably fewer nodes searching for the goal v_g .

G Runtime Analysis

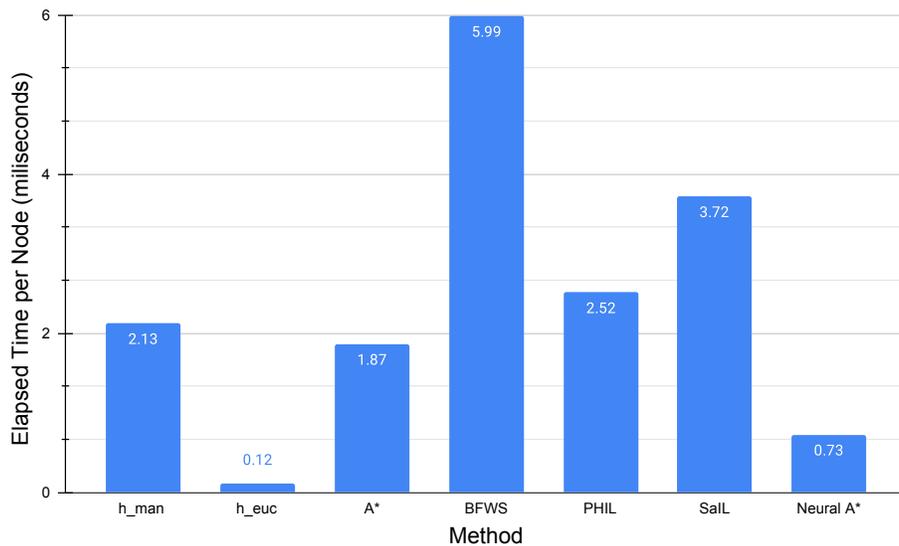


Figure 18: Timing experiments on grid search datasets [1].

We evaluate runtimes of 7 approaches during inference on all the datasets in Section 5.1. Each approach’s runtime are averaged by number of nodes it visited. The runtime experiments are conducted on a machine with a RTX5000 GPU and a Intel Xeon 5222 CPU.