Abstract

The A* algorithm is commonly used to solve NP-hard combinatorial optimization problems. When provided with an accurate heuristic function, A* can solve such problems in time complexity that is polynomial in the solution depth. This fact implies that accurate heuristic approximation for many such problems is also NP-hard. In this context, we examine a line of recent publications that propose the use of deep neural networks for heuristic approximation. We assert that these works suffer from inherent scalability limitations since — under the assumption that \( P \neq NP \) — such approaches result in either (a) network sizes that scale exponentially in the instance sizes or (b) heuristic approximation accuracy that scales inversely with the instance sizes. Our claim is supported by experimental results for three representative NP-hard search problems that show that fitting deep neural networks accurately to heuristic functions necessitates network sizes that scale exponentially with the instance size.

1 Introduction

Principal computational problems such as planning and scheduling [Wilkins, 2014], routing [Toth and Vigo, 2002], and combinatorial optimization [Papadimitriou and Steiglitz, 1998] are known to be NP-Hard (NP-H) in their general form. Consequently, there are no known polynomial-time algorithms for solving them. Moreover, for many of these problems (belonging to the NP-Complete class), it is unknown if a polynomial-time solver is attainable. This complexity gap, known as the P vs. NP problem [Cook, 2003], remains one of the biggest open CS questions to date (2022).

Recognizing the challenges/unattainability of polynomial complexity solvers, many researchers are focusing on reducing the exponential complexity of known solvers using heuristic functions [Pearl, 1984]. One prominent example of an optimization algorithm that utilizes such heuristics is the A* algorithm [Hart et al., 1968]. A line of publications [Goldenberg et al., 2014; Felner et al., 2018] exhibited this algorithm’s ability to achieve exponential reductions in computational time when paired with an informative heuristic function. Moreover, it is easy to show that given a sufficiently accurate heuristic function, A* can solve NP-H problems in complexity that is polynomial in the solution length and the branching factor. This fact has two major implications that are discussed in this paper.

1. Attaining a sufficiently accurate heuristic function could enable scalable solutions to a large class of fundamental CS problems.
2. Querying such a sufficiently accurate heuristic function is NP-H.

Our work expands upon the initial discussion on scalability of heuristic approximations by Pendurkar et al. [2022]. We show that training a heuristic function to arbitrary precision is indeed attainable using common machine learning approaches for problems that can be reduced to a discrete search space. This fact along with implication 1 motivates a line of publications [McAleer et al., 2018; Agostinelli et al., 2019; 2021b] to propose methods for accurately approximating heuristic values using universal function approximators. However, our paper asserts that, given implication 2, these methods are not ‘scalable’ in nature. That is, the complexity of such heuristic approximators does not scale polynomially with increasing problem sizes. Such a formal discussion can help place previous publications in an appropriate context. Namely by showing that the applicability of heuristic approximations is inherently limited due to scalability barriers. Our claims are supported by rigorous experiments examining the minimal fully connected neural network that is required to fit true heuristic values of three NP-H problems to various levels of precision.

2 Preliminaries

The class of problems that are solvable in polynomial time by a deterministic Turing machine is denoted by $P$. NP is the class of problems for which a solution can be verified in polynomial time. As a result, these problems are solvable in polynomial time by a non-deterministic Turing machine. NP-Hard (NP-H) is a class of problems to which every problem in NP can be reduced in polynomial time. Finally, NP-Complete (NP-C) is a class of problems that belong to both NP and NP-H. As a result, proving that any problem in NP-C is also in P implies that $P=NP$.

A large set of NP-H problems can be reduced to a least cost path problem over an appropriate graph, $G = \{V, E\}$. In such cases, each edge $(e \in E)$ is affiliated with a non-negative cost ($c : E \rightarrow \mathbb{R}^+$). Consequently, the least cost path is defined as $p = \arg \min_{E' \subseteq E} \sum_{e \in E'} c(e)$ subject to $E'$ is an ordered set of edges leading from a given start vertex ($v_s \in V$) to a defined goal vertex ($v_g \in V$). The resulting path ($p$) is a solution to the original NP-H problem. Despite this, the least cost path problem is not NP-H as, for all known reductions from an NP-H problem, the size of the resulting graph scales exponentially with the problem instance size, thus the reduction is not of polynomial complexity.

The resulting least cost path problems are commonly solved using the A* algorithm. A* searches for the least cost path over the graph without constructing the full graph explicitly. The cost function $g : V \rightarrow \mathbb{R}^+$ estimates the cumulative cost from the start vertex to any vertex $v \in V$. A* is also guided by a heuristic function, $h : V \rightarrow \mathbb{R}$, that estimates the least cost path from any vertex in the search graph, $v \in V$, to a goal vertex. The vertices of the search graph are denoted states hereafter, and the set of all states is denoted by $S$. A* was shown to expand the minimal number of vertices that is required for finding and proving an optimal (least cost) solution [Pearl, 1984].

Given an accurate heuristic function, that is, $\forall s \in S, h(s) = h^*(s)$, where $h^*(s)$ is the true minimal cost between $s$ and a goal state, A* would expand only the states along the optimal (least cost) path.

That is, the complexity of solving the problem is polynomial in the solution depth and branching factor (the maximum degree of the vertices in the search graph). While current, domain-specific heuristic functions were shown to result in orders of magnitude speedups [Helmert and Mattmüller, 2008] for NP-H problems, they lack sufficient accuracy to allow polynomial time complexity. Attempting to close this gap, a recent body of work suggested fitting a universal function approximator [Higgins, 2021], e.g., deep neural networks, to $h^*$.

2.1 Related Work

One of the initial works on approximating heuristic learning was done by Arfaee et al. [2010] which we refer as Bootstrap Learning Heuristic (BLH). They proposed an iterative method ("bootstrap

\footnote{1}Code is available at https://github.com/Pi-Star-Lab/unscaleable-heuristic-approximator
\footnote{2}The reduction function is specific to each domain [Bulteau et al., 2015; Cormen et al., 2009; Gupta and Nau, 1992].
\footnote{This claim assumes an admissible and consistent heuristic function [Pearl, 1984].}
\footnote{This claim assumes that f-values ties are broken in favor of higher g values}
Tree Search (MCTS) to solve the Rubik’s Cube domain. They learn a value function (negative of the heuristic function) using temporal difference learning (bootstrapping), with a neural network as the value function approximator, and MCTS as the search algorithm. Following [Agostinelli et al., 2019] presented the DeepCubeA algorithm which learns a heuristic function similar to [McAleer et al., 2018], but replaces MCTS with weighted A* [Ebendt and Drechsler, 2009] and uses a distinct state distribution to perform bellman updates. DeepCubeA presents state-of-the-art results on various NP-H domains in terms of solution quality and run time complexity (number of generated states).

Another line of work focused on learning a policy [Orseau et al., 2018, Orseau and Lelis, 2021], where a policy is a function that maps states (vertices in the search graph) to operators (edges in the search graph). The optimal policy is the one returning the edge that follows the least cost path to the goal state. [Orseau et al., 2018] proposed using a policy-guided search algorithm and provided theoretical guarantees on the number of states expanded. Policy-guided Heuristic Search (PHS) [Orseau and Lelis, 2021] uses both a heuristic function along with a policy for a better performance.

On the other hand, there are various universal approximators that can approximate any well behaved function to arbitrary precision. A 2-layer feedforward neural network, with non-polynomial activation function and sufficient number of neurons in the intermediate hidden layer is one such universal function approximator [Cybenko, 1989, Hornik et al., 1989, Pinkus, 1999]. Another universal function approximator, is a neural network with non-affine continuous activation function, and a fixed number of neurons per layer, but with sufficient number of such layers [Kidger and Lyons, 2020].

3 The feasibility of approximating $h^*$

At first glance, fitting a universal function approximator to $h^*$ seems promising given Corollary [1].

**Corollary 1.** A universal function approximator can fit $h^* : S \mapsto \mathbb{R}$ to arbitrary precision assuming a discrete state space $S$ with a bounded heuristic range $h^* (S)$.

Corollary [1] is an extension of universal function approximation theorem for discrete input (state) space with a bounded function ($h^*$) range. Thus, Corollary [1] shows that it is possible to fit a universal function approximator (like a two layer neural network) to $h^*$ sufficiently accurate such that the A* algorithm would run in time complexity that is polynomial in the optimal solution depth (number of edges) and branching factor. This is because, A* would expand only the states along an optimal path while generating all their neighbors. We defer the proof to Appendix [B.1]. Despite this positive result, Lemma [1] shows that querying such an approximator for a large set of NP-C problems is, in fact, NP-H. Note that previous work [Bruck and Goodman, 1988] already proved that querying a neural network that outputs an exact solution to a NP-H problem is itself NP-H. We extend this result to heuristic approximation. We first define $\epsilon$ Bounded NP-Complete problems.

**Definition 1.** A problem belongs to the set of $\epsilon$ Bounded NP-C problems (eBNP-C) iff:

1. It is an NP-C decision problem defined by
   - a discrete and bounded state space $S$.
   - a start state, $s \in S$.
   - a set of solutions per start state (potentially empty), $solutions(s)$.
   - a cost function $cost : solutions(s) \mapsto \mathbb{R}$.
   - a target solution cost bound, $k$.

2. It answers the following question: is there solution $l \in solutions(s)$ such that $cost(l) \leq k$.

3. For any $s \in S$ and any two solutions $l_1, l_2 \in solutions(s)$, it must hold that $\lvert cost(l_1) - cost(l_2) \rvert = 0 \text{ OR } \lvert cost(l_1) - cost(l_2) \rvert \geq \epsilon$.

**Lemma 1.** Calling (querying) a function approximator $\hat{h} : S \mapsto \mathbb{R}$, satisfying $\max_{s \in S} \lvert h^*(s) - \hat{h}(s) \rvert < \epsilon/2$ is NP-H when considering $S$ spanned by any eBNP-C problem.

The proof is in Appendix [B.2]. Lemma [1] implies that learning accurate heuristic values for state spaces spanned by eBNP-C problems is not scalable in nature. For example, consider fitting $h^*$ to $\epsilon/2$
precision using an artificial neural network. Querying such a network using forward propagation has a complexity of $O(n^2 l)$ where $n > 1$ is the maximal layer width and $l \geq 1$ is the number of hidden layers. That is, the querying complexity grows polynomially with $l$ and $n$. As a result, $P \neq NP$ and Lemma 1 necessitate that either $l$, $n$, or both, grow non polynomially with an $\epsilon$ BNP-C instance size.

Further, [Helmert and Mattmüller, 2008] show that a precision of $\epsilon/2$ is required to avoid expansion of exponential number of nodes by the A* algorithm.

4 Experiments

The experimental study is designed to address the following questions.

1. How does the complexity of a universal function approximator, $\hat{h} \approx h^*$, scale with a state space spanned by an $\epsilon$ BNP-C problem?
   (a) Does such an approximator suffer from ineffective learning due to memorization of the training data (overfitting)?
2. Is the approximator scalability trend sensitive to variation in the target approximation precision?

4.1 Universal Function Approximators

We use the two artificial neural network structures as universal function approximators. First, a 2-layer neural network (1 hidden layer), with Rectified Linear Unit (ReLU) [Nair and Hinton, 2010] as the activation function. We refer to this approximator as ‘fixed depth’ where the number of neurons in the hidden layer may vary, but the number of layers is fixed. Second, we use a neural network with a fixed width per layer, but allow any number of layers. As in the previous setting, we use ReLU as the activation function for each layer. Similarly, we refer to this structure as ‘fixed width’. We used residual connections [He et al., 2016] and batch normalization [Ioffe and Szegedy, 2015] to mitigate issues such as vanishing gradients that were observed during optimization. Our fixed width architecture follows that of [Agostinelli et al., 2019], with two differences. First, we reduce the number of neurons per layer to allow a more gradual increase of the number of parameters following addition of layers. Following theoretical constraints on the minimal layer size presented by [Kidger and Lyons, 2020], the size of each layer was set as the number of input dimensions + 3. Second, we just have 1 layer before the residual blocks (as opposed to 2) for an odd number of hidden layers. This enables us to have an odd number of layers. Doing so also contributes to a more gradual increase in the number of parameters.

Following previous work [Arfaee et al., 2010] [Agostinelli et al., 2019, 2021b] we only consider the ‘fixed depth’ and ‘fixed width’ structures. Other variants of neural networks, like convolutional neural networks used by [Orseau and Lelis, 2021], are not considered as they make specific assumptions regarding the state feature space (e.g., spatial locality) which do not hold in the domains used in our experiments.

4.2 Setup

Fitting Criterion: For a model to be considered as ‘fitting’, we train the model on a fixed size dataset until a certain number of epochs and check whether it follows a ‘fitting criterion’. For the experiments, we use MSE as the fitting criterion and set a maximum of 300 epochs to achieve it. We use different threshold values ($T$) to relax the fitting criterion.

Loss Function: We use MSE as the loss function throughout the experiments. Note, this is a relaxation of the strict requirements presented in Section 3 so that the experiments are more in line with [Agostinelli et al., 2019, 2021a,b] [Orseau and Lelis, 2021].

Datasets: For all domains, the training set ($D$) included $10^6$ (a million) samples. For all cases, the test set was set to $2 \times 10^5$ random samples.

---

4

1ReLU activation function satisfies the properties required by the universal function approximation theorem [Sonoda and Murata, 2017].

2Residual networks were shown to be universal function approximators [Lin and Jegelka, 2018].
Optimization: We optimize the neural networks using the Adam optimizer [Kingma and Ba, 2015]. Note that Adam is not guaranteed to converge on the global optimum. However, it is widely used in the literature [Agostinelli et al., 2019, Orseau and Lelis, 2021] as it usually leads to near optimal solutions in practice. To mitigate the impact of local minimas, we train each neural network 5 times and report the lowest loss. It is important to note that, in our setting, a model failing to fit $D$ does not necessarily mean that it is unable to fit $D$ as no guarantees are provided regarding convergence to the global optimum.

Domains: We choose domains that (a) have a state space spanned by an cBNP-C problem and (b) in which the number of states grows relatively slowly to have comparison over a larger range of instance sizes. Following these reasons, we choose Pancake sorting, Travelling salesman problem (TSP), and Blocks world (BW). Details about the domains and cBNP-C equivalency are in Appendix C.2.

Note that several effective methods were presented for pruning the state space for the domains considered [Valenzano and Yang, 2017, Fitzpatrick et al., 2021, Slaney and Thiébaux, 2001]. However, we avoid using such methods as our experiments are designed to study the scalability of heuristic approximation and not present/analyze a state-of-the-art method.

4.3 Results

4.3.1 Scaling neural networks for increasing problem size:

To address Experiment Question 1, we report the smallest number of parameters (weights and biases) that are able to fit the dataset for increasing problem sizes. Figure 1 shows plots for the number of parameters on a log scale for a fixed depth approximator. For BW we see a linear trend (on a log scale) that suggests that the minimum number of parameters grows exponentially as we increase problem size. For pancake and TSP, we can see a linear trend until a certain problem size and then decrease in minimum number of required parameters. This suggests that, after a certain point, the approximator (neural network) starts overfitting, denoting that the neural network is expressive enough to memorize the entire training set. For instance, if we consider the pancake puzzle, the rise is linear until problem size 12, and after that it starts decreasing, suggesting memorization of the training samples. We see a similar trend at problem size 7 for TSP.

Figure 1: Increase in minimum number of parameters (log scale) for a ‘fixed depth’ network required to fit problems with increasing sizes. On the x-axis we have the problem sizes for each of the domains. On y-axis to the left, we have the number of parameters on a log scale. On y-axis to the right, we have MSE values. Loss thresholds are 0.2, 0.35 and 0.2 for pancake, TSP and BW respectively.

Figure 2: Increase in minimum number of parameters (log scale) for a ‘fixed width’ network required to fit problems with increasing sizes. On the x-axis we have the problem sizes for each of the domains. On y-axis to the left, we have the number of parameters on a log scale. On y-axis to the right, we have MSE values. Loss thresholds are 0.1 0.35 and 0.02 for pancake, TSP and BW respectively.

We see similar patterns for the fixed width case as shown in Figure 2. The plots are noisier than Figure 1 as adding another layer of size $n$ increase the number of parameters by $n^2$ extra weight and
extra bias parameters, making it challenging to observe a continuum of the number of parameters. We can see that the number of parameters required to fit is similar across fixed depth and fixed width cases ($\sim 5 \times 10^5$) at the point when memorization begins.

To validate our overfitting hypothesis (Question 1a), we include a test error for each approximator size in Figure 1 and Figure 2. It is easy to see that while the minimal reported network size stagnates past some problem size (for pancake and TSP), the test error continues to increase which is a common indicator of overfitting. For BW, by contrast, we do not see a rise in test error but also no stagnation.

These results also show that neural networks fail to find good latent structures for NP-H problems unlike in text and image based data. This suggests that a sufficiently accurate heuristic function for NP-hard problems might not have a meaningful latent structure and thus require memorization. As a result, the scalability of a heuristic approximator follows that of a naive lookup table, which grows exponentially with the instance size.

4.3.2 Invariance to $\epsilon$

It seems enticing to think that $h^*$ can easily be fitted if the acceptable loss threshold is set high enough. Although, it is true that fitting to a larger loss threshold is easier than fitting to a low threshold, we see that, even for larger thresholds, the number of parameters still scale exponentially with the problem size. Figure 3 shows the growth in number of parameters across problem sizes and for various representative loss thresholds. Although the number of parameters required for lower precision is lower, the required number of parameters still grows exponentially. We can also see that, as we increase the threshold, the point where overfitting begins changes. For instance, for pancake problem, we see that for threshold 0.6, the curve is mostly linear until problem size 17. Similar to our results in previous subsection, we begin to see overfitting when we have a minimum of $\sim 5 \times 10^5$ parameters, across different thresholds and domains. These results suggest that the answer to Question 2: “Is the approximator scalability trend sensitive to variation in the target approximation precision?” is ‘No’.

We also study the (in)variance of the trend to the choice of optimization objective and the details are presented in Appendix D. We observe the trend is agnostic to the choice of the optimization objective.

5 Summary

In this paper, we investigate the unscalability of heuristic approximators for NP-hard search problems. We provide theoretical justifications for our claim of unscalability while empirically verifying our claims through experiments on 3 representative domains. We also show (empirically) that, irrespective of the architecture of the neural network, choice of optimization objective, and required precision, the number of parameters needed to fit a heuristic function scale exponentially with the problem instance size. The main conclusion drawn from this paper is that heuristic search algorithms that rely on function approximators to fit heuristic values for NP-Hard problems are inherently not scalable. We expect that our paper will impact the research community by shifting its research efforts to other/additional ways of integrating heuristic search with machine learning.
References


A Summary of previous results

<table>
<thead>
<tr>
<th>Problem</th>
<th>State Space</th>
<th>Approach</th>
<th>Num. Parameters</th>
<th>Solution Quality</th>
<th>Expanded Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 Tile puzzle</td>
<td>3.00 × 10^{24}</td>
<td>DeepCubeA</td>
<td>3.00 × 10^4'</td>
<td>253.4</td>
<td>*5.73 × 10^9'</td>
</tr>
<tr>
<td>24 Tile puzzle</td>
<td>7.70 × 10^{24}</td>
<td>DeepCubeA</td>
<td>↑ 2.10 × 10^4'</td>
<td>↓ 89.5</td>
<td>*2.01 × 10^9'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PHS*</td>
<td>1.05 × 10^6</td>
<td>↑ 224.0</td>
<td>↓ 2.87 × 10^3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PHS_h</td>
<td>1.05 × 10^6</td>
<td>119.5</td>
<td>5.86 × 10^4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BLH</td>
<td>↓ +3.00 × 10^4</td>
<td>-</td>
<td>5.22 × 10^6</td>
</tr>
<tr>
<td>15 Tile puzzle</td>
<td>1.00 × 10^{13}</td>
<td>DeepCubeA</td>
<td>↑ 1.82 × 10^4'</td>
<td>52.0</td>
<td>*1.28 × 10^9'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BLH</td>
<td>↓ +3.00 × 10^4</td>
<td>-</td>
<td>↓ 1.01 × 10^4</td>
</tr>
<tr>
<td>Sokaban</td>
<td>*1.53 × 10^{15}</td>
<td>DeepCubeA</td>
<td>↑ 1.50 × 10^4'</td>
<td>32.9</td>
<td>*1.05 × 10^9'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PHS*</td>
<td>↓ 3.71 × 10^6</td>
<td>37.6</td>
<td>1.52 × 10^3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PHS_h</td>
<td>↓ 3.71 × 10^6</td>
<td>39.1</td>
<td>↑ 2.13 × 10^3</td>
</tr>
</tbody>
</table>

Table 1: Comparison of previous approaches: DeepCubeA [Agostinelli et al., 2019], Policy Guided Heuristic (PHS_h, PHS*) [Orseau and Lelis, 2021], Bootstrap Learning Heuristic (BLH) [Arfaee et al., 2010]. '*' in front of a value denotes some approximation based on additional assumptions. '¬' denotes unreported values in original paper. '↑' in front of value denotes the worst value, and '↓' denotes the best value for the domain. 'State Space' denotes the size of state space.

Table 1 shows a rough comparison between the results of previous approaches. For each domain ('problem') we report the size of its state space and for each approach, the number of parameters used to fit the underlying heuristic approximator, the resulting solution quality (cost), and its running complexity (number of expanded states).

For the number of parameters of BLH, we calculate a rough estimate assuming a 3 layer neural network with 1000 neurons in each layer. DeepCubeA reports the number of generated nodes instead of number of expanded nodes, which we approximate as generated_nodes / branching_factor. PHS* and PHS_h are two variants as used in [Orseau and Lelis, 2021]. For the 10×10 Sokoban grid with 4 boxes, the size of state space is approximately calculated as 100 × (100) × (100) where 100 are possible player locations (100) are box locations as well as box targets. We disregard parameters added due to any batch normalization layer. We note that the change in the number of parameters for a given approach results from varying the number of input features for the problem size, and not from changes to the hidden layers’ layout. An exception to this is DeepCubeA, which uses 6 residual blocks [He et al., 2016] for Tile puzzle and 4 for Rubik’s Cube and Sokoban. The results present a trend where scaling the problem size necessitates a larger approximator (w.r.t the number of parameters). If the approximator’s size is not increased, then we observe reduced accuracy ('Solution Quality') and/or increased computational complexity ('Expanded Nodes').

B Proofs in the feasibility of approximating $h^*$

B.1 Proof of Corollary I

*Proof.* Any $n + 1$ set of discrete values ($h^*(s)$) can be fitted with an $n^{th}$-order polynomial (by e.g., the Lagrange method of interpolation). The resulting polynomial maps from some finite and bounded domain space (defined by $S$) to another finite dimension space (defined by $h^*$) and can thus be fitted to arbitrary precision following the universal function approximation theorem [Hornik et al., 1989].

B.2 Proof of Lemma I

*Proof.*

1. **Known NP-C problem:** any eBNP-C problem.

2. **Poly time reduction:** any instance of the eBNP-C problem can be reduced to a single call of $h(s)$. Specifically, $eBNP-C(s, k) \equiv h(s) < (k + \epsilon/2)$.

3. **Equivalence:** if $eBNP-C(s, k) = True$ then $h(s) < h^*(s) + \epsilon/2 \leq k + \epsilon/2$. If $eBNP-C(s, k) = False$ then $h(s) \geq h^*(s) - \epsilon/2 \geq k + \epsilon - \epsilon/2 = k + \epsilon/2$.
C Additional Experimental Setup Details

C.1 Finding the minimum number of parameters to fit $D$

Note, if a neural network with 1-hidden layer of size $n$ can fit a given function, it must be that a neural network with $n + 1$ neurons in the hidden layer can also fit (by simply setting the additional weights as 0). Similarly, for fixed width case, if a neural network with $n$ layers can fit a function, it can also fit it with $n + 1$ layers, by setting the weight matrix of the added layer as the identity matrix. Following these understandings, we perform a binary search on the number of neurons (fixed depth) or layers (fixed width) to effectively approximate the minimum number of neurons or layers that is required to fit $D$.

C.2 Domains

**Pancake:** (1) **Description:** The pancake sorting problem is a NP-H problem [Bulteau et al., 2015], where the task is to sort pancakes stacked one on top of another in minimum number of steps. A step allows inserting a spatula at any position in the stack and flipping (inverting) all the pancakes above it. (2) **Dataset Generation:** Following [Agostinelli et al., 2019], we generate the training data by taking a random walk from the goal state. To calculate the $h^*$ values (or labels), we solve the problem with A* search and an admissible heuristic function known as the gap heuristic [Helmert, 2010]. (3) **Encoding:** The state encoding, $\phi(s)$ was defined through a one-hot encoding of each pancake location in the stack. An example of pancake problem along with state representation can be seen in Figure 4 (a). (4) $\epsilon$BNP-C: The decision problem for pancake is NP-C [Bulteau et al., 2015] and if the cost of each move is assumed to be constant (set to 1 in our case), it is also $\epsilon$BNP-C. As a result, we can bound the heuristic approximation error defined by Lemma 1 to $|h^*(s) - \hat{h}(s)| < 0.5$. (5) **Problem Size:** Problem size of $n$ is a pancake problem with $n$ pancakes.

**Travelling Salesman Problem (TSP):** (1) **Description:** TSP is a NP-H problem [Cormen et al., 2009] where, given a graph of cities, the task is to find the shortest route (sum of cost of edges) that visits every city (vertex in graph) exactly once while returning to the start city. (2) **Dataset Generation:** We generate complete weighted directed graphs with edges uniformly sampled from numbers in range $[0.1, 5]$ with 0.1 granularity. The start node is randomly picked. Note that, for the experiments, we consider only initial states, that is, no city is travelled, but use different graphs with different start cities. To calculate $h^*$, we use the Held–Karp algorithm [Held and Karp, 1962] as the solver. (3) **Encoding:** The state encoding, $\phi(s)$ was defined by a 1-dimensional representation of adjacency matrix of the graph concatenated with array of length $n$ denoting which nodes are visited and the start node. An example of TSP problem along with state representation can be seen in Figure 4 (b). (4) $\epsilon$BNP-C: The decision version of TSP is known to be NP-C [Cormen et al., 2009]. As the minimum possible edge cost is 0.1, difference of any two feasible solutions will be either 0 or $\geq 0.1$, thus the decision version of TSP is $\epsilon$BNP-C. (5) **Problem Size:** Problem size of $n$ is a TSP problem with $n$ distinct cities.

**Blocks World (BW):** (1) **Description:** The BW problem is a NP-H problem [Gupta and Nau, 1992] that consists of a number of blocks stacked into towers [Slaney and Thiébaux, 2001], where the task is to turn a given start state to a given goal state with a minimum number of steps. A step allows moving one block from the top of a tower onto the top of another one or to the table. (2) **Dataset Generation:** We fix a goal state and generate instances with uniform random start states. To calculate the $h^*$ values, we solve the instances with A* using the number of blocks that are in the wrong positions as heuristic.

Figure 4: Examples of problem state for each of three domains along with state encoding $\phi(s)$. 
We use a goal state where all blocks are stacked on each other in order with the lowest numbered block at the bottom. (3) **Encoding:** We use the same state encoding, $\phi(s)$, as used in [Slaney and Thiébaux, 2001]. An example of BW problem along with the state encoding can be seen in Figure 4 (c). (4) **ɛBNP-C:** It was previously shown [Gupta and Nau, 1992] that the decision problem of BW is NP-C. Given a constant operation cost of 1, the decision version of BW is in ɛBNP-C. (5) **Problem Size:** Problem size of $n$ is a BW problem with $n$ blocks.

**Rubik’s Cube and Tile Puzzle:** For both of the domains the minimum difference between any two solutions is 0 or $\geq 1$, and their decision variant is in NP-C [Demaine et al., 2018, Ratner and Warmuth, 1990] respectively. Thus, their decision version is in ɛBNP-C.

### D Invariance of unscalability trends to loss objectives

![Figure 5: Increase in minimum number of parameters (log scale) required to fit problems with increasing sizes on various loss functions. On the x-axis we have the problem sizes for each of the domains. On y-axis we have the number of parameters on a log scale. The thresholds used for scaled loss, accuracy threshold for cross entropy loss, MSE are 0.001, 90%, 0.2. Beyond the initial variants of $l_1$ loss, we now report results for two additional loss functions.

1. **categorical cross entropy loss** by viewing the heuristic learning as a classification problem.
2. **Scaled loss** defined as:
   $$\frac{1}{N} \sum_{i=1}^{N} (1 - \frac{h_\theta(s_i)}{h^*(s_i)})^2$$

The motivation behind the scaled loss variant is as follows. In many cases, the suboptimality factor of the A* algorithm is governed by the relative error in $h_\theta$ and not the absolute error. The weighted-A* algorithm [Ebenst and Drechsler, 2009] is a prominent example.

Figure 5 shows a plot for three loss functions for the pancake problem using the ‘fixed depth’ network structure (similar trends were observed for the other domains and network structure). These results suggest that the exponential size increase of a fitting neural network followed by stagnation (suggesting overfitting) is agnostic to the choice of the loss functions.