Learning to Learn from Search

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

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

Abstract

The effectiveness of heuristic search when solving pathfinding problems often relies on the accuracy of the heuristic function. While it has been shown that heuristic functions can be learned using deep reinforcement learning, it is assumed that learning is only performed once and that the heuristic function remains fixed when attempting to solve a problem instance. However, heuristic search, itself, produces data from which a heuristic function can learn. For example, given the nodes expanded during search, Bellman updates of the states associated with those nodes can be quickly computed. In this paper, we introduce Learning to Learn from Search (L2LFS), an algorithm that trains a sequence of heuristic functions, where each heuristic function in the sequence learns from data obtained when performing heuristic search with the previous heuristic function. We will show that data obtained during search can be encoded into a fixed length encoding and that this data can be used to train a subsequent heuristic function that better estimates Bellman updates when compared to one that does not use this encoding. Future work will refine and repeat this process multiple times in hopes of yielding iteratively improving heuristic functions.

Introduction

Heuristic search, a widely used approach for solving pathfinding problems (Hart, Nilsson, and Raphael 1968; Bonet and Geffner 2001a), is guided by a heuristic function that approximates the cost-to-go, which is the cost of a shortest path from a given start state to a given goal. Given a heuristic function that accurately estimates the cost-to-go, heuristic search can quickly solve problems optimally (i.e., via a shortest path) or close to optimally. On the other hand, inaccurate heuristic functions can result in significantly sub-optimal solutions or solutions not being found due to impractically slow search or all available memory being consumed by search. However, data generated from heuristic search can be used to improve the heuristic function for a subsequent search. In particular, after performing A* search (Hart, Nilsson, and Raphael 1968), a Bellman update for the states associated with expanded nodes can quickly be obtained using their child nodes, which will have already been generated and for which heuristic values have already been computed. This data can be used to improve a subsequent heuristic function that would then be used in a subsequent search. This process can then be repeated with the objective of yielding iteratively improving heuristic functions.

In this paper, we introduce learning to learn from search (L2LFS), an algorithm that builds on DeepCubeA (Agostinelli et al. 2019) to train heuristic functions to improve their cost-to-go estimation (i.e., learn) based on data obtained from search. Data obtained during search that is used to improve a subsequent heuristic function is referred to as the **search summary**. A heuristic function that also takes a search summary as an input is referred to as a **summary-aware heuristic function (SAHF)**. In this paper, we make the following contributions:

- Showing that search summaries can be encoded into a fixed-length representation.
- Leveraging this encoding to train SAHFs that learn from search.
- Showing that SAHFs better estimate the Bellman updates of states seen during a subsequent search when compared to a heuristic function that is not summary-aware.

Related Work

Learning Heuristic Functions. Heuristic functions have traditionally been derived using domain knowledge or automated methods such as pattern databases (PDBs)(Culberson and Schaeffer 1998), transformations (Mostow and Prieditis 1989), or delete relaxations (Bonet and Geffner 2001b). While these approaches offer theoretical guarantees, their scalability is often limited by computational resources, especially in high-dimensional or complex domains (Muppasani et al. 2023). An alternative direction aims to learn heuristic functions, often at the cost of admissibility guarantees. Supervised approaches use expert-generated cost-to-go values (Samadi et al. 2008; Chrestien et al. 2021; Toyer et al. 2020), while other approaches improve heuristics by iteratively using the results of previous searches (Bramanti-Gregor and Davis 1993; Fink 2007; Arfaee, Zilles, and Holte 2011; Orseau and Lelis 2021). However, these methods typically ignore expanded nodes that do not contribute to the final solution and struggle when no solutions are found, limiting sample efficiency.

Recent work explores using large language models (LLMs) to generate heuristic functions without hand-crafted domain knowledge (Ling et al. 2025; Corrêa, Pereira, and Seipp 2025). While promising, these methods often rely on prompt engineering to inject domain-specific cues and remain largely untested across diverse and complex planning domains. A more efficient strategy learns from all expanded nodes by applying Bellman updates to estimate cost-to-go values (Thayer, Dionne, and Ruml 2011). These updates require only the child nodes of expanded states, making them fast to compute during A* search. Such updates can be used as training targets in approximate dynamic programming frameworks (Bertsekas and Tsitsiklis 1996), enabling the learning of more accurate heuristics without requiring expert demonstrations or solved instances. A notable example of this approach is DeepCubeA (Agostinelli et al. 2019), which we describe in more detail in the Background section.

Our work builds on these ideas by introducing summary-aware heuristic functions that leverage encoded representations of Bellman updates from prior search episodes. This allows the heuristic to generalize not just from states but from search summaries, effectively learning to learn from search.

Meta-Learning. Meta-learning (Hospedales et al. 2021), often described as "learning to learn," is a subfield of machine learning that focuses on building models capable of acquiring knowledge from a distribution of tasks to enable rapid adaptation to new ones. Unlike traditional models that are trained for a single, specific task, a meta-learning system is trained to be a generalist. Its goal is to leverage "meta knowledge" extracted from a variety of learning episodes to solve a new problem more effectively and with less data than a model trained from scratch. This capability is especially valuable in domains where data is scarce, such as medicine or robotics.

While both L2LFS and meta-learning aim to produce a generalizable learning system, their core objectives and operational paradigms are distinct. Meta-learning's central goal is to train a model that can rapidly adapt to a new and unseen task from a different distribution using minimal data. The learning process focuses on acquiring "meta knowledge" that enables this fast adaptation. L2LFS, on the other hand, is designed to learn an effective heuristic function for search by observing the search process, itself, within a continuous stream of attempts on problems from a single domain. Its learning is not about adapting to a new task but about a continuous refinement of the search process for a given class of problems. The "learning" that is being learned is the ability to generate a more accurate heuristic functions given the search history, rather than the ability to adapt to a novel problem domain.

Background

Pathfinding

A pathfinding domain is defined as a weighted directed graph (Pohl 1970), where nodes represent states, edges represent actions that transition between states, and weights on the edges represent transition costs. The transition function is represented by T, where $s' = T(s, a)$ if and only if there exists an edge connecting states s and s' for some action, a. The transition cost function is represented by c, where $c(s, a)$ is the transition cost when taking action a in state s. The set of all possible actions is denoted $\mathcal{A}$. A pathfinding problem instance is defined by a tuple (D, s_0, g), where D is the domain, s_0 is the start state, and g is the set of goal states. Given a pathfinding problem, the objective is to find a path, which is a sequence of actions, that transforms the start state into a goal state while attempting to minimize the path cost, where the path cost is the sum of transition costs. A shortest path (i.e., an optimal path) is a path from a given state to a given goal that has the lowest path cost possible.

Heuristic Search

Heuristic search is a widely used approach for solving pathfinding problems that is guided by a heuristic function, h, that maps a state, s, and a goal, g, to an estimate of its cost-to-go, which is the cost of a shortest path from s to a closest goal state in g. The most notable heuristic search algorithm, A* search (Hart, Nilsson, and Raphael 1968), maintains a search tree, where nodes represent states and edges represent actions. A* search iteratively selects nodes for expansion prioritized by their cost, which is the sum of their path cost from the start node (i.e., cost-to-come computed by the sum of transition costs) and their heuristic value (i.e., cost-to-go computed by h). A node is expanded by applying every possible action to the state associated with that node and creating a child node from the resulting states. A* search terminates when a node associated with a goal state is selected for expansion and returns the path to that node.

DeepCubeA

DeepCubeA (Agostinelli et al. 2019) learns a heuristic function represented as a DNN (Schmidhuber 2015; LeCun, Bengio, and Hinton 2015), h_θ, with parameters, θ. The learned heuristic function is then used with batch weighted A* search (BWAS) that selects a batch of B nodes for expansion at each iteration of search while performing weighted A* search (Pohl 1970). DeepCubeA generates states by starting from the goal and taking actions in reverse, which assumes the goal is known before training and that a reverse transition function exists [1]. Therefore, the heuristic function does not need to be given a goal since it is implicitly given during training. The heuristic function is trained using approximate value iteration (Bertsekas and Tsitsiklis 1996), a dynamic programming algorithm and foundational reinforcement learning algorithm (Bellman 1957; Sutton and Barto 2018). Approximate value iteration iteratively trains a neural network with gradient descent to approximate a Bellman update, $h'(s)$, for each state, s, in a given batch of states, using the loss function in Equation 1, where N is the batch size. The Bellman update in the context of pathfinding is shown in Equation 2, where g is the goal. The heuristic function used for the Bellman update in Equation 2 is h_{θ^-}, where θ^- is the parameters of the target network (Mnih

[1]Extensions of DeepCubeA have removed the need for both of these assumptions (Agostinelli, Panta, and Khandelwal 2024; Agostinelli and Soltani 2024).

et al. 2015) which are periodically updated to θ. Approximate value iteration with deep neural networks is referred to as deep approximate value iteration (DAVI).

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

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

Learning to Learn from Search

The L2LFS algorithm learns an initial heuristic function, h_{θ_0}, with parameters, θ_0, with DAVI. Next, a search summary encoder, μ_{ϕ_0}, with parameters, ϕ_0, is trained to encode search summaries. From this search summary encoder, a SAHF, h_{θ_1}, is trained with DAVI. This process can be performed for I_s iterations, yielding a sequence of heuristic functions, $\bar{h} = [h_{\theta_0}, ..., h_{\theta_{I_s}}]$, and summary encoders, $\bar{\mu} = [\mu_{\phi_0}, ..., \mu_{\phi_{I_s-1}}]$. $\bar{h}$ and $\bar{\mu}$ can then be used in sequence to solve problems, as shown in Algorithm 1. To train the summary encoders and SAHFs, we build on Algorithm 1 to generate training data for the summary encoders, which is then used to train the SAHFs, as shown in Algorithm 2.

The summary we obtain from search is a set of tuples, where each tuple contains a state associated with an expanded node and its Bellman update. We choose this because:

- Given a set of states and their Bellman updates, it is possible that a SAHF could learn to generalize beyond this and approximate a Bellman update for similar states.
- The Bellman update of states associated with expanded nodes can quickly be obtained since Equation 2 uses the child nodes, which A* search will have already generated, and for which heuristic values will have already been computed. Therefore, the Bellman update for these states amounts to performing a simple one-step backup in the search tree for each expanded node.

Algorithm 1: `L2LFS_solve`

```
Input: domain $\mathcal{D}$, start state $s_0$, heur fns $\bar{h}$, summary encs $\bar{\mu}$, max A* search itrs $I_A$
$\tilde{r}$ = None //initial summary encoding
path = None
for $k \in [0, \texttt{len}(\bar{h}))$ do
    path, $r$ = A*search($\mathcal{D}, s_0, \bar{h}[k], \tilde{r}, I_A$)
    if $k < (\texttt{len}(\bar{h}) - 1)$ then
        $\tilde{r} = \bar{\mu}[k](r)$
    end if
end for
return path, $r$
```

Training the Initial Heuristic Function

To train the initial heuristic function, we follow a similar approach to DeepCubeA by starting from the goal state and taking actions in reverse. However, we also generate additional data by performing A* search for a set maximum number of iterations, I_A, and add states associated with nodes selected for expansion to the training set. Furthermore, DeepCubeA updated the parameters of the target network when the loss went below a pre-determined threshold. We update the parameters of the target network after a set number of iterations. To account for the fact that this could lead to instability during training due to the fact that the target cost-to-gos may fluctuate more severely, we include the use of a replay buffer (Mnih et al. 2015) that is a first-in-first-out queue of a fixed size that stores the latest training examples. We do not update the cost-to-go targets of the states in the replay buffer, even though they may have been generated by a previous target network, as we found that this helped stabilize training. We also found that adding states seen during A* search to the training set helped improve the performance of A* search.

Since the SAHF has an additional input of a summary encoding, we train the initial heuristic function with this input and set the summary encoding to be zero all the time. The trained initial heuristic function will then be re-used to train the summary encoder.

Algorithm 2: `L2LFS_train`

```
Input: domain $\mathcal{D}$, summ itrs $I_s$, max A* search itrs $I_A$
$\bar{h}$ = []
$\bar{\mu}$ = []
$\bar{s}$ = gen_instances($\mathcal{D}$)
$\bar{\tilde{r}}^p$ = None //previous summaries encodings
$h_{\theta_0}$ = train_init_heur($\mathcal{D}$, $\bar{s}$)
$\bar{h}$.append($h_{\theta_0}$)
for $k \in [0, I_s)$ do
    $\bar{r}$ = []
    for $s \in \bar{s}$ do
        _, $r$ = L2LFS_solve($s, \bar{h}, \bar{\mu}, I_A$)
        $\bar{r}$.append($r$)
    end for
    $\mu_{\phi_k}$ = train_summ_enc($\bar{r}, \bar{h}[k]$)
    $\bar{\tilde{r}}$ = []
    for $r \in \bar{r}$ do
        $\tilde{r} = \mu_{\phi_k}(r)$
        $\bar{\tilde{r}}$.append($\tilde{r}$)
    end for
    $h_{\theta_{k+1}}$ = train_SAHF($\mathcal{D}, \bar{s}, \bar{r}, \bar{h}[k], \bar{\tilde{r}}^p$)
    $\bar{\mu}$.append($\mu_{\phi_k}$)
    $\bar{h}$.append($h_{\theta_{k+1}}$)
    $\bar{\tilde{r}}^p = \bar{\tilde{r}}$
end for
return $\bar{h}, \bar{\mu}$
```

Training the Summary Encoder

A search summary, r, is a a set of tuples, $\{..., (s_i, h'(s_i)), ...\}$, of states associated with the nodes expanded during search and their corresponding Bellman updates. We seek to train a summary encoder, μ_ϕ, with parameters, ϕ, to map r to a summary encoding, $\tilde{r}$, such that we can accurately recover any $h'(s_i) \in r$ given s_i and $\tilde{r}$.

To obtain a summary encoding, $\tilde{r}$, from a search summary, r, we use the deep set architecture (Zaheer et al. 2017) for μ_ϕ to map r to a vector, $\tilde{r}$, of size $N_{\tilde{r}}$. The deep set architecture independently processes each element in the set separately and combines them with a permutation invariant operator, such as a max or mean operator. In our work, each state and Bellman update pair are concatenated and given to a fully-connected residual network (He et al. 2016) with an output of size $2N_{\tilde{r}}$. Half of this $2N_{\tilde{r}}$ vector is used to compute softmax values over the $N_{\tilde{r}}$ entries for the final encoding and the other half are multiplied by the computed softmax values.

To ensure that we can recover Bellman updates for states in the summary given the summary encoding and state, we copy the parameters of the most recently trained heuristic function, h_θ, to obtain the decoder $h_{\tilde{\theta}}$. For each element in a summary, the decoder is trained to predict the Bellman update from the corresponding state and summary encoding. Both ϕ and $\tilde{\theta}$ are trained together with gradient descent using the loss function in Equation 3.

$$L([\phi, \tilde{\theta}]) = \frac{1}{N} \sum_{i}^{N} \frac{1}{|r_i|} \sum_{(s, h'(s)) \in r_i} (h'(s) - h_{\tilde{\theta}}(s, \mu_\phi(r_i)))^2 \tag{3}$$

Training the Summary-Aware Heuristic Function

Given the trained summary encoder, μ_{ϕ_k}, we can now train the SAHF, $h_{\theta_{k+1}}$. The SAHF is initialized with the parameters of h_{θ_k}. For each training state and its corresponding summary encoding, we perform A* search for a maximum of I_A iterations and add each state associated with a node selected for expansion to a replay buffer along with the given summary encoding. We then randomly sample from this replay buffer and train $h_{\theta_{k+1}}$ to approximate the Bellman update for each state using gradient descent using the loss function in Equation 4. We then repeat this process and perform A* search with the updated $h_{\theta_{k+1}}$ and add the resulting data to the replay buffer. Each summary encoding, $\tilde{r}$, given as input to $h_{\theta_{k+1}}$ has a corresponding previous summary encoding, $\tilde{r}^p$, given as input to h_{θ_k}. We use this, along with h_{θ_k}, to compute the Bellman update as shown in Equation 5.

$$L(\theta_{k+1}) = \frac{1}{N} \sum_{i}^{N} (h'(s_i, \tilde{r}_i^p) - h_{\theta_{k+1}}(s_i, \tilde{r}_i))^2 \tag{4}$$

$$h'(s, \tilde{r}) = \begin{cases} 0, & \text{if } s \in g, \\ \min\limits_{a \in \mathcal{A}} c(s,a) + h_{\theta_k}(T(s,a), \tilde{r}), & \text{otherwise.} \end{cases} \tag{5}$$

The states and summary encodings used for training will be correlated, since they are both produced from the same start state. As a result, the trained SAHF may perform poorly if it sees states outside of the distribution of states it saw during training, which come from nodes expanded during A* search. Therefore, states on which it may perform poorly include states associated with nodes that were generated, but not selected for expansion, as well as states associated with nodes seen when training a subsequent SAHF, $h_{\theta_{k+2}}$, for which $h_{\theta_{k+1}}$ will be used to compute the Bellman update. To address this, with a given random probability, states sampled for training will be given a summary encoding selected randomly from the replay buffer.

Experiments

We use the Rubik's cube to evaluate the performance of L2LFS. The Rubik's cube is represented to a DNN using a flat one-hot representation of the 54 stickers. The input is given to a linear layer of size 1,000, and then a residual neural network (He et al. 2016) with four residual blocks of size 1,000. This is finally given to an output linear layer of size 1. Layer normalization (Ba, Kiros, and Hinton 2016) is used in the residual layers along with the SPLASH activation function with a single hinge (Tavakoli, Agostinelli, and Baldi 2021). The summary encoder for the deep set also processes each element in the search summary independently using a network of the same architecture, with an additional input of the Bellman update. The size of the summary encoding is 1,000. The input to the SAHF is the concatenated state and summary encoding.

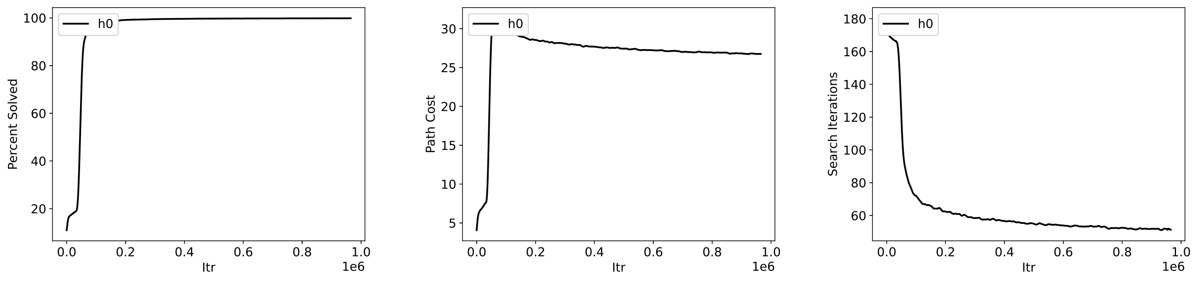
When training heuristic functions, starting states are generated for training by starting from the goal and, for each state, taking between 0 and 100 random actions. The maximum number of A* search iterations, I_A, is set to 200. For each training iteration, 50 searches are performed and the training batch size is 10,000. The size of the replay buffer is 1 million for the initial heuristic function and 10 million for the SAHFs. When training the initial heuristic function, the target network is updated every 100 iterations. The initial heuristic function is trained for 1 million iterations and the SAHFs are trained for 100,000 iterations. The summary encoders are trained for 100,000 iterations and use a batch size of 200 summaries[2]. The neural networks are trained with ADAM (Kingma and Ba 2014) optimizer with a starting learning rate of 0.001 and a decay rate of 0.9999993.

Initial Heuristic Function Training

The initial heuristic function was trained by using A* search to add states to the training set. The performance of the heuristic function when used with A* search is shown as a function of training iteration in Figure 1. The figure shows that the number of states solved increases as training iteration increases and the number of search iterations decreases. The number of solved states manages to almost reach 100%. The average path cost initially increases as the number of states solved increases and then decreases after the number of states solved approaches 100%.

In terms of BWAS, we are using a search batch size of 1 and a weight of 1, which is A* search. Note that previous approaches used BWAS to solve instances with a batch size of 10,000 and a weight of 0.6. The supplementary material of DeepCubeA showed that BWAS was not able to find paths

[2]Note that the batch size for the summary encoder is significantly smaller because each example can contain up to I_A states, which is set to 200.



(a) Percentage solved. (b) Path cost. (c) Search iterations.

Figure 1: Performance of A* search as a function of training iteration when training the initial heuristic function. The percentage of states solved increases as training increases. Since path cost is only reported for solved states, it initially increase, and then decreases as the heuristic function successfully solves close to 100% of instances.

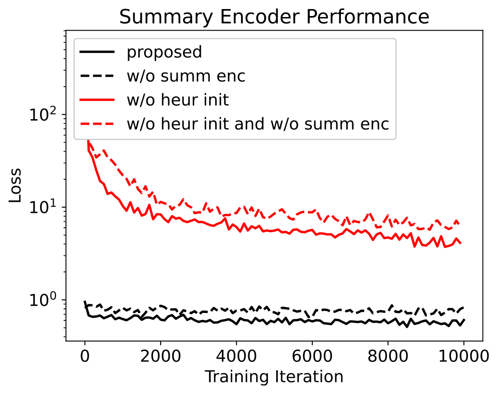


Figure 2: An ablation study when training the summary encoder. The proposed method uses the outputs of the summary encoder and initializes the decoder with the most recently trained heuristic function. Other methods are a combination of multiplying the output of the encoder by 0 (w/o summ enc) and using a random initialization for the decoder (w/o heur init). The plot shows that both using a summary encoder and heuristic function initialization for the decoder results in a more accurate prediction.

with a search batch size of 1 and a weight of 0.6 or greater (Agostinelli et al. 2019). This indicates that our approach to training the heuristic function improves performance on A* search compared to previous approaches. Future work will include more detailed comparisons.

Summary Encoder Performance

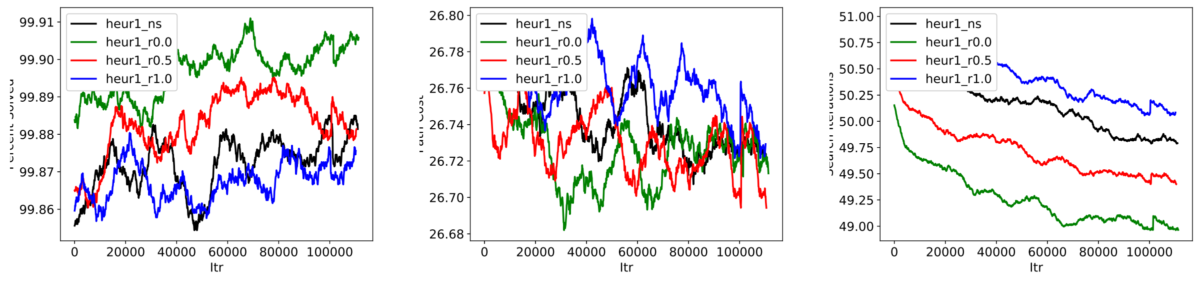
To ensure that our summary encoder encodes information relevant to the search summary, we perform an ablation study by comparing the performance of the summary encoder to the performance when the output of the summary encoder in Equation 3 is always multiplied by 0. Furthermore, we also examine the impact of initializing the decoder with the parameters of the most recently trained heuristic function, as opposed to a random initialization, both with the summary encoder and when multiplying its output by 0. The results of the ablation study are shown in Figure 2. Note that the y-axis is on a log-scale. The figure shows that loss is larger without the summary encoder, both with and without heuristic function initialization for the decoder. This indicates that the summary encoder learns to encode information relevant to the search summary. Furthermore, when not using the heuristic function initialization, the loss is significantly higher.

Summary-Aware Heuristic Function Performance

We now examine the performance of h_{θ_1}, which is the heuristic function that learns from the initial heuristic function, h_{θ_0}. To examine the effect of learning from a search summary encoding, we also train a heuristic function with effectively no summary encoding by always multiplying the summary encoding by 0 before giving it as input to the heuristic function. To examine the effect of training state correlation with the summary encoding, we train different SAHFs with probabilities 0.0, 0.5, and 1.0 for sampling a random summary encoding from the replay buffer.

We compare the performance of A* search during training for the different heuristic functions in Figure 3. The figure shows that the percentage solved increases and the number of search iterations decreases with lower summary encoding randomness. It also shows that the SAHFs outperform the heuristic function with no summary encoding in almost all cases. The path cost of states is similar for all cases, however, the path cost is only computed for states that are solved. So, the path cost may be higher for the SAHFs due to them solving a higher percentage of states.

While the results in Figure 3 indicate the SAHFs perform better when performing A* search, we also must consider how well they estimate the Bellman update, in general. This is especially important if we use h_{θ_1} to compute Bellman updates for h_{θ_2}, since the states used for training h_{θ_2} are determined by performing A* search with h_{θ_2}. The distribution of these states will probably differ from the states h_{θ_1} obtained for training. To investigate this, we compared all four heuristic functions to each other by generating a dataset with A* search with each heuristic function, using h_{θ_0} to compute a Bellman update, and comparing their accuracy at



(a) Percentage solved. (b) Path cost. (c) Search iterations.

Figure 3: Performance of A* search as a function of training iteration when training the heuristic function that learns from the initial heuristic function. Training a SAHF with encodings that are not sampled completely randomly (i.e. heur1_rand0.0 and heur1_rand0.5) increases percentage of states solved and decreases search iterations when compared to training a heuristic function with no summary encoding (i.e. heur1_ns). The lower the randomness the better the performance.

estimating the Bellman update for each dataset. For each A* search, we obtain a dataset of states associated with nodes expanded during search (popped) and with nodes not expanded during search (open). We also include a dataset of randomly generated states. The results in Figure 4 show that, for the popped dataset, SAHFs perform significantly better compared to the heuristic function with no summary encoding, with the one with no randomness performing the best. However, for the open dataset, the SAHFs only slightly outperform the heuristic function with no summary encoding in most cases. Notably, for the random dataset, the SAHF trained with a random state encoding probability of 0.0 performs significantly worse than all other heuristic functions.

Discussion and Future Work

The performance of the L2LFS depends on how effectively a search summary can be encoded so that it can be used to train a subsequent heuristic function. The lowest mean squared error obtained from our set encoder is around 0.34, which indicates there is a lot of room for improvement. Our current summary is a set of state and Bellman update tuples and our current approach for encoding them is using the deep set architecture. However, each element in the set is processed independently, which may make it difficulty to determine which parts of the encoding each element should affect. One way to addressing this could be to process each element relative to the start state. This way, the parts of the encoding that are affected by a given element could be more consistent across problem instances. Another approach is to use set encodings, such as the set transformer (Lee et al. 2019), that involve interaction amongst the elements in the set. Finally, additional information can be added to the search summary that may be relevant to approximating a Bellman update. For example, it has been shown that elements in the open list and closed set are relevant to improving search performance (Felner, Shperberg, and Buzhish 2021).

Figure 4 shows that performance decreases when data is obtained from the open set instead of the popped nodes. While this may indicate a limitation of the ability to learn from a search summary, this decrease in performance is also present for the heuristic function that is trained without a search summary. This indicates that the distribution of states seen during training should be modified to include other relevant states. While randomly sampling summary encodings during training partially addresses this, this is not likely to improve the performance on states closely related to a start state that could be seen when training a subsequent heuristic function. Therefore, future work could also obtain related states by performing stochastic versions of A* search, such as selecting nodes for expansion with a probability inversely proportional to their cost.

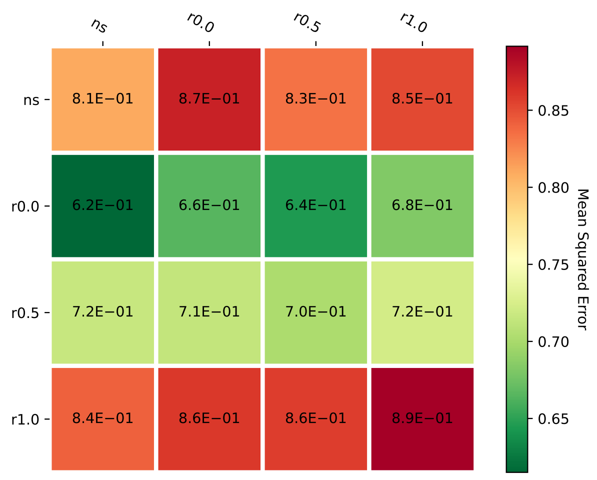
Once the issue of improving summary encoding and improving performance of SAHFs on states other than the ones from expanded nodes, we can repeat the training process shown in 2 to obtain subsequent summary encoders and subsequent heuristic functions. This will hopefully lead heuristic functions that iteratively improve in their estimate of the true cost-to-go. This work could also be extended to other kinds of heuristic search that can make use of learned heuristic functions, such as Q* search (Agostinelli et al. 2024).

Conclusion

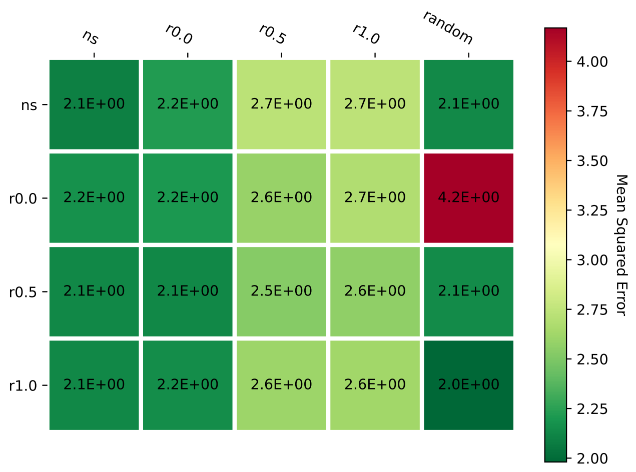
We introduce L2LFS, an algorithm for learning to learn from search. We showed that an encoding of a search summary could be obtained and our experiments indicate that the encoding preserves information relevant to the search summary. We then show that this search summary could be used to train a subsequent summary-aware heuristic function (SAHF). Our experiments showed that this led to better estimations of the Bellman update and better A* search performance when compared to not using a summary encoding. Our experiments also indicate that future work should focus on improving the summary encoding as well as varying the distribution of states seen when training the SAHF to better train subsequent SAHFs.

Acknowledgements

The work of Shahaf Shperberg was supported by the Israel Science Foundation (ISF) grant #909/23, by Israel's



(a) Test states from nodes expanded during search.



(b) Test states from unexpanded nodes and random states.

Figure 4: The performance when estimating the Bellman update for the trained heuristic functions for step $k = 1$. Each row is a trained SAHF and each column is data generated from either performing search with a SAHF or sampling random states. "ns" means the heuristic function was trained without a summary encoding and "r<prob>" is means the SAHF is trained while sampling a random encoding with probability <prob>. The results show that the heuristic functions perform better when given a summary when tested on data from nodes expanded, but do not perform as well for out-of-distribution training data, such as data from nodes not expanded or random states. This indicates that improvements can be made to the training state distribution to better estimate the Bellman update for relevant states.

Ministry of Innovation, Science and Technology (MOST) grant #1001706842, in collaboration with Israel National Road Safety Authority and Netivei Israel, awarded to Shahaf Shperberg, by BSF grant #2024614 awarded to Shahaf Shperberg, as part of a joint NSF-BSF grant with Forest Agostinelli. This material is based upon work supported by the National Science Foundation under Award No. 2426622.

References

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

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

Agostinelli, F.; Shperberg, S. S.; Shmakov, A.; McAleer, S.; Fox, R.; and Baldi, P. 2024. Q* search: Heuristic search with deep q-networks. In *ICAPS Workshop on Bridging the Gap between AI Planning and Reinforcement Learning*.

Agostinelli, F.; and Soltani, M. 2024. Learning discrete world models for heuristic search. In *Reinforcement Learning Conference*.

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

Ba, J. L.; Kiros, J. R.; and Hinton, G. E. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450*.

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

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

Bonet, B.; and Geffner, H. 2001a. Planning as heuristic search. *Artificial Intelligence*, 129(1-2): 5–33.

Bonet, B.; and Geffner, H. 2001b. Planning as heuristic search. *Artif. Intell.*, 129(1-2): 5–33.

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

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

Corrêa, A. B.; Pereira, A. G.; and Seipp, J. 2025. Classical Planning with LLM-Generated Heuristics: Challenging the State of the Art with Python Code. *arXiv preprint arXiv:2503.18809*.

Culberson, J. C.; and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence*, 14(3): 318–334.

Felner, A.; Shperberg, S. S.; and Buzhish, H. 2021. The Closed List is an Obstacle Too. In *Proceedings of the International Symposium on Combinatorial Search*, volume 12, 121–125.

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

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

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

Hospedales, T.; Antoniou, A.; Micaelli, P.; and Storkey, A. 2021. Meta-learning in neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 44(9): 5149–5169.

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

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

Lee, J.; Lee, Y.; Kim, J.; Kosiorek, A.; Choi, S.; and Teh, Y. W. 2019. Set transformer: A framework for attention-based permutation-invariant neural networks. In *ICML*, 3744–3753. PMLR.

Ling, H.; Parashar, S.; Khurana, S.; Olson, B.; Basu, A.; Sinha, G.; Tu, Z.; Caverlee, J.; and Ji, S. 2025. Complex LLM planning via automated heuristics discovery. *arXiv preprint arXiv:2502.19295*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533.

Mostow, J.; and Prieditis, A. 1989. Discovering Admissible Search Heuristics by Abstracting and Optimizing. In *ML*, 240–240. Morgan Kaufmann.

Muppasani, B.; Pallagani, V.; Srivastava, B.; and Agostinelli, F. 2023. On Solving the Rubik's Cube with Domain-Independent Planners Using Standard Representations. *arXiv preprint arXiv:2307.13552*.

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

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

Samadi, M.; Siabani, M.; Felner, A.; and Holte, R. 2008. Compressing pattern databases with learning. In *ECAI*.

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

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

Tavakoli, M.; Agostinelli, F.; and Baldi, P. 2021. Splash: Learnable activation functions for improving accuracy and adversarial robustness. *Neural Networks*, 140: 1–12.

Thayer, J. T.; Dionne, A. J.; and Ruml, W. 2011. Learning Inadmissible Heuristics During Search. In *ICAPS*.

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

Zaheer, M.; Kottur, S.; Ravanbakhsh, S.; Poczos, B.; Salakhutdinov, R. R.; and Smola, A. J. 2017. Deep sets. *NIPS*, 30.