Utilising Uncertainty for Efficient Learning of Likely-Admissible Heuristics

Ofir Marom, Benjamin Rosman
University of the Witwatersrand, Johannesburg, South Africa

Abstract

Likely-admissible heuristics have previously been introduced as heuristics that are admissible with some probability. While such heuristics only produce likely-optimal plans, they have the advantage that it is more feasible to learn such heuristics from training data using machine learning algorithms. Naturally, it is ideal if this training data consists of optimal plans, but such data is often prohibitive to produce. To overcome this, previous work introduced a bootstrap procedure that generates training data using random task generation that incrementally learns on more complex tasks. However, 1) using random task generation is inefficient and; 2) the procedure generates non-optimal plans for training and this causes errors to compound as learning progresses, resulting in high suboptimality. In this paper we introduce a framework that utilises uncertainty to overcome the shortcomings of previous approaches. In particular, we show that we can use uncertainty to efficiently explore task-space when generating training tasks, and then learn likely-admissible heuristics that produce low suboptimality. We illustrate the advantages of our approach on the 15-puzzle, 24-puzzle, 24-pancake and 15-blocksworld domains using Bayesian neural networks to model uncertainty.

1 Introduction

It is well-known that admissible heuristics lead to optimal plans when using A* and its variants (Pearl 1984). Unfortunately, crafting strong admissible heuristics is difficult, often requiring expert domain knowledge or high memory resources (Junghanns and Schaeffer 2001; Felner and Adler 2005; Zahavi et al. 2008; Slaney and Thiébaux 2001; Haslum and Scholz 2003; Helmert 2010). This is particularly evident in domains with large state-spaces that are known to be extremely challenging to solve with admissible heuristics.

An alternative approach is to leverage machine learning techniques and learn a heuristic from data. Such heuristics have been shown to require far less domain knowledge and memory resources, but unfortunately this approach also sacrifices optimality because the heuristic is no longer guaranteed to be admissible (Arfaee, Zilles, and Holte 2010; 2011; Ernandes and Gori 2004; Groshev et al. 2017; Thayer, Dionne, and Ruml 2011; Samadi, Felner, and Schaeffer 2008; McAleer et al. 2018).

One way to introduce the notion of admissibility with statistical machine learning algorithms is through *likely-admissible heuristics* (Ernandes and Gori 2004). As we review in section 2, likely-admissible heuristics are admissible with some probability, and this leads to likely-optimal plans. While the authors who introduce likely-admissible heuristics develop a novel cost function to encourage admissibility (Ernandes and Gori 2004), they do not provide a way of specifying likely-admissibility in a formal probabilistic way. Furthermore, this cost function relies on having access to training data of optimal plans, which is often prohibitive to produce.

Subsequently, a bootstrap procedure was developed that does not require access to optimal plans for training (Arfaee, Zilles, and Holte 2010; 2011). This procedure works by generating a set of random training tasks and proceeds to switch between planning and learning. The core idea is to try solve each training task within some specified time limit. If not enough tasks are solved in an iteration, the time limit is increased so that more tasks can be solved in the next iteration. Otherwise the heuristic is learned on the solved tasks, thus becoming a stronger heuristic, and the time limit remains fixed for the next iteration. As discussed by the authors of this procedure there are two main shortcomings with this approach: 1) much time is wasted because the procedure tries to solve *all tasks* and many tasks are not solvable in the given time limit when the heuristic is still weak; and 2) the heuristic learns from non-optimal plans and so errors compound with each iteration of the procedure, resulting in a final heuristic that produces high suboptimality.

In this paper we propose to overcome both these limitations by utilising *epistemic* and *aleatoric* uncertainty. Epistemic uncertainty accounts for uncertainty in the model of a process and can be explained away given enough data while aleatoric uncertainty accounts for uncertainty that the model cannot explain away due to probabilistic variation in the data. As we describe in section 3, epistemic uncertainty can be used to efficiently explore task-space so as to generate training tasks that are of the right level - that is, the tasks are easy enough to solve while being difficult enough for learning to progress. Meanwhile, epistemic and

aleatoric uncertainty are combined so that the heuristic remains likely-admissible during planning and this mitigates the compounding errors that result in high suboptimality.

In section 5 we use a concrete implementation with Bayesian neural networks (Blundell et al. 2015; Kendall and Gal 2017) to model these uncertainties and run experiments on the 15-puzzle, 24-puzzle, 24-pancake and 15-blocksworld domains where we demonstrate our approach empirically and show that it is more efficient to train than random task generation, while producing plans that have low suboptimality. In fact, for the 15-puzzle we show that our approach can produce optimality statistics that are competitive with previous approaches that *trained on optimal plans*.

2 Background

Planning

A planning domain is defined as a tuple $\langle \mathcal{S}, \mathcal{O}, \mathcal{E}, \mathcal{C}, \mathcal{S}_0, \mathcal{S}_g \rangle$ where $\mathcal{S}$ is the state-space, $\mathcal{O}(s)$ is an operator function that returns the legal operators than can be executed in state $s \in \mathcal{S}$, $\mathcal{E}(s, o)$ is an effect function that returns the next state $s' \in \mathcal{S}$ when operator $o \in \mathcal{O}(s)$ is applied in s, $\mathcal{C}(s, s')$ is a strictly positive and bounded function for the cost of moving from state s to s', $\mathcal{S}_0 \subseteq \mathcal{S}$ is the set of possible start states and $\mathcal{S}_g \subseteq \mathcal{S}$ is the set of possible goal states. A planning task under a given domain is then defined by $\mathcal{T} = \langle \mathcal{S}, \mathcal{O}, \mathcal{E}, \mathcal{C}, s_0, s_g \rangle$ where $s_0 \in \mathcal{S}_0$ and $s_g \in \mathcal{S}_g$ and $s_0 \neq s_g$. For ease of notation we assume in this paper that each domain has a unique goal state and therefore tasks in a given domain differ in their start state only. We discuss how to extend our framework to the case with multiple goal states in the supplementary material.

Many planning algorithms additionally require a heuristic function $h(s)$ that returns an estimate of the optimal cost-to-goal from s. We denote the optimal, and unknown, cost-to-goal from s with $h^*(s)$. Then given a planning task $\mathcal{T}$, a planning algorithm aims to finds a plan $\pi = (s_0, s_1, s_2, ..., s_n = s_g)$. If h is an admissible heuristic so that $h(s) \leq h^*(s)$ for all $s \in \mathcal{S}$ then certain planning algorithms, for example A* and IDA*, guarantee that π is optimal in the sense that it is a plan with minimal cost. We denote optimal plans with π^*.

While admissibility is a desirable property, guaranteeing admissibility of a heuristic is often difficult. An alternative, albeit weaker, property is to require the heuristic to be likely-admissible (Ernandes and Gori 2004). A likely-admissible heuristic, denoted h^α, is admissible with probability $\alpha \in (0, 1)$ i.e. $P(h^\alpha(s) \leq h^*(s)) = \alpha$ for all $s \in \mathcal{S}$. Since only overestimations of states within the optimal plan π^* can affect optimality with A* search, the probability that A* guided by h^α returns an optimal plan is at least $p_{\pi^*} = \alpha^{|\pi^*|}$ where $|\pi^*|$ is the length of π^* (Ernandes and Gori 2004). Since admissibility is a sufficient but not a necessary condition for optimally, p_{π^*} is a statistical lower bound for optimality.

Learning Heuristics from Data

Given a plan π we can compute for each $s_j \in \pi$ where $j < n$ the cost-to-goal from s_j to $s_n = s_g$, $y_j = \sum_{i=j}^{n-1} \mathcal{C}(s_i, s_{i+1})$. Therefore, given a training dataset of M plans from a domain $\Pi = \{\pi_i\}_{i=1}^M$ we can construct a training dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$ of size $N = \sum_{i=1}^M |\pi_i| - M$ by computing the cost-to-goal for the relevant states in each plan where $x_i = F(s_i)$ and F is a function that converts a state to a feature representation that can be used as input to a supervised machine learning algorithm.

Then any number of regression based supervised learning algorithms can be used to train a heuristic from $\mathcal{D}$. Once trained, this heuristic can be used to solve new, and previously unseen, tasks from the given domain. In general, such a heuristic is not admissible and therefore will lead to non-optimal plans. The quality of the learned heuristic depends largely on the plans used for training - intuitively, training on optimal or near-optimal plans will produce a heuristic that is a closer approximation of h^* and therefore have lower suboptimality. Neural networks are a common choice of supervised learning algorithm and have shown to produce good results on a variety of domains (Arfaee, Zilles, and Holte 2010; 2011; Ernandes and Gori 2004). We call heuristics based on neural networks *neural heuristics*.

Bayesian Neural Networks

One can view a neural network as a probabilistic model $P(y|x, \mathbf{w})$ (Murphy 2012) where x is the input to the network and $\mathbf{w}$ are the weights. For regression it is most common to assume that $y|x, \mathbf{w} \in \mathbb{R}$ follows a Gaussian distribution so that $y|x, \mathbf{w} \sim \mathcal{N}(\hat{y}(x, \mathbf{w}), \sigma_a^2(x, \mathbf{w}))$ where $\hat{y}$ is the mean and σ_a^2 is the variance of the distribution. Typically when doing regression with neural networks only a single output neuron is used to learn $\hat{y}$ while σ_a^2 is assumed to be a known constant, and this corresponds to the well-known squared loss objective $(\hat{y} - y)^2$ (Murphy 2012). However, it is straightforward to adjust the network to have two output neurons and learn σ_a^2 as well. This leads to the following minimisation objective function (Kendall and Gal 2017):

$$\mathcal{L}(\mathcal{D}, \mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \frac{(\hat{y}(x_i, \mathbf{w}) - y_i)^2}{2\sigma_a^2(x_i, \mathbf{w})} + \frac{1}{2} \log \sigma_a^2(x_i, \mathbf{w}), \tag{1}$$

In practice, we learn an second output neuron $\rho \in \mathbb{R}$ and then apply a transformation to get $\sigma_a \in \mathbb{R}^+$ such as $\sigma_a = \log(1 + \exp(\rho))$. Then σ_a^2 captures the noise in the data and is thus a measure of aleatoric uncertainty.

Epistemic uncertainty is more difficult to model as it requires the posterior distribution $P(\mathbf{w}|\mathcal{D})$ which is generally intractable to compute because of the large number of parameters and functional form of neural networks (Shridhar, Laumann, and Liwicki 2019). One effective and computationally efficient method to approximate the posterior distribution is to use a *weight uncertainty neural network* (WUNN) (Blundell et al. 2015) and choose to model the weights not as fixed point values but as independent Gaussians, $w_i|\mathcal{D} \sim \mathcal{N}(\mu_{w_i|\mathcal{D}}, \sigma^2_{w_i|\mathcal{D}})$. Using a Gaussian prior for the weights with mean μ_0 and variance σ_0^2 so that $w_i \sim \mathcal{N}(\mu_0, \sigma_0^2)$, the variational approximation to the posterior can be shown to be the following minimisation objective:

$$\mathcal{L}(\mathcal{D}, \theta) = \beta KL[P(\mathbf{w}|\theta)||P(\mathbf{w})] - \mathbb{E}_{P(\mathbf{w}|\theta)}[\log P(\mathcal{D}|\mathbf{w})], \tag{2}$$

where KL is the Kullback-Leibler divergence, $\theta = \{(\mu_{w_i|\mathcal{D}}, \sigma^2_{w_i|\mathcal{D}})\}_i$ for each weight connection in the network and $\beta > 0$ controls how much weight to give the prior relative to the likelihood (Higgins et al. 2017). A common strategy is to decay β over time so as to initially give more emphasis to the prior and gradually reduce this as the network is trained on more data. In practice, the objective function is approximated with Monte Carlo during training by sampling $z^s_i \sim \mathcal{N}(0, 1)$ for each weight connection in the network and then applying the transformation $w^s_i = z^s_i \sigma_{w_i|\mathcal{D}} + \mu_{w_i|\mathcal{D}}$ where $s \in [1, S]$ and S is the number of Monte Carlo samples averaged over.

Given a WUNN trained to minimise this objective, we can get a measure of epistemic uncertainty for an input x by sampling weights from the network and computing the variance of the network output $\sigma^2_e(x) = \frac{1}{K}\sum_{k=1}^{K} \hat{y}^2(x, \mathbf{w}_k) - \hat{y}^2(x)$, where K is the number of samples from the network and $\hat{y}(x) = \frac{1}{K}\sum_{k=1}^{K} \hat{y}(x, \mathbf{w}_k)$. The intuition behind WUNNs is as follows: suppose a new input x is passed through the WUNN that is very different from what has previously been observed in training. When the network was trained it was required to fit the parameters of the network to both the training data *and the prior*. Therefore, this unseen data will tend towards the prior. So by setting a high prior variance for the weights we can expose observations that are dissimilar to what was seen in training as the model will return a large value for σ^2_e. Meanwhile if x is similar to what was observed in training then the model will return a low value for σ^2_e because the likelihood term was maximised to fit such data.

Finally, given an input x we can approximate the posterior predictive distribution as

$$y|x \sim \mathcal{N}(\hat{y}(x), \sigma^2_t(x)), \tag{3}$$

where $\sigma^2_t(x) = \sigma^2_a(x) + \sigma^2_e(x)$ and $\sigma^2_a(x) = \frac{1}{K}\sum_{k=1}^{K} \sigma^2_a(x, \mathbf{w}_k)$.[1]

3 Conceptual Framework

Our framework assumes that we have a model $\mathcal{M}$ of the form

$$y|x \sim \mathbb{D}(\mu(x), \nu_a(x) + \nu_e(x)), \tag{4}$$

where $\mathbb{D}$ is a distribution with mean parameter μ, aleatoric uncertainty parameter ν_a and epistemic uncertainty parameter ν_e. For the purposes of this paper we use the formalisation as described in equation 3 but it is possible to use a different distribution for $\mathbb{D}$, such as a Laplace distribution, or use different techniques to model uncertainty (Gal and Ghahramani 2015; Hernandez-Lobato and Adams 2015; Sun, Chen, and Carin 2017; Blundell et al. 2015).

We leverage epistemic and aleatoric uncertainty to resolve the shortcomings of previous approaches that only learn the mean parameter μ. In this context epistemic uncertainty arises because a state s may induce features $x = F(s)$ that are dissimilar to what was observed during training, while aleatoric uncertainty arises because either 1) the features extracted from F are insufficient to deterministically explain the cost-to-goal observations; or 2) we train on non-optimal plans so identical states can map to different cost-to-goal observations.

Given such a model, likely-admissibility can be formally described by using a value y^α such that $P(y^\alpha \leq y|x) = \alpha$ where $\alpha \in (0, 1)$ for the heuristic. The value y^α can be computed from the inverse cumulative distribution function of $\mathbb{D}$ which is done either analytically or using approximation methods depending on the form of $\mathbb{D}$. We note that for this formalisation of likely-admissibility to hold we require that $\mathcal{M}$ learns from training data of optimal plans so that $y|F(s)$ is a model for $h^*(s)$.

Combining these ideas we introduce two algorithms: *GenerateTask* (Algorithm 1) and *LearnHeuristic* (Algorithm 2). The algorithm *GenerateTask* is used to generate a training task that is then solved with *LearnHeuristic*. The key idea for *GenerateTask* is to use epistemic uncertainty to generate a task that is easy enough to solve while being difficult enough so that learning from the solved task makes the heuristic stronger. To achieve this, we start at the goal state s_g. With some user specified value ϵ, we take steps back from the goal state until we observe a state s such that $\nu_e(F(s)) \geq \epsilon$. We then stop execution and build a task $\mathcal{T}$ with start state $s_0 = s$. The algorithm is designed to efficiently explore the task-space so as to seek states with high epistemic uncertainty by sampling states from the softmax distribution derived from the epistemic uncertainties of all possible next states (see *GenerateTask* line 16) and so our algorithm covers the state-space more efficiently than using random task generation. Since the algorithm builds tasks by taking steps back from the goal we further need access to an undo effect function $\mathcal{E}^{\text{undo}}(s')$ that returns the set of all states s that move to s' under some operator in $\mathcal{O}(s)$, i.e. $\mathcal{E}^{\text{undo}}(s') = \{s \mid \exists o \in \mathcal{O}(s) \text{ such that } \mathcal{E}(s, o) = s' \text{ where } s \in \mathcal{S}\}$. However, if the domain has reversable actions, as is the case of the domains used in this paper, we may simply use $\mathcal{E}^{\text{undo}}(s') = \mathcal{E}^{rev}(s') = \{\mathcal{E}(s', o) \mid o \in \mathcal{O}(s')\}$.

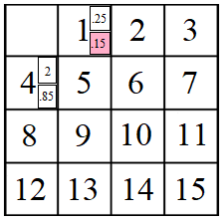
The next step is to learn the heuristic. This is achieved with the algorithm *LearnHeuristic*. The algorithm proceeds to generate *NumTasksPerIter* tasks with *GenerateTask* then solve each task using a user specified admissibility probability α. Solving the tasks with y^α ensures that plans that the heuristic is trained on are encouraged to be likely-optimal and this mitigates the compounding error problem that exists with previous approaches. We note that in our algorithm the training data consists of non-optimal plans and so y^α is not formally a likely-admissible heuristic. Nevertheless, we make the assumption that the plans are close enough to optimal that any errors incurred are negligible - in practice the larger α is set, the closer this assumption holds.

A subtle but important requirement for the algorithm is that when we train the heuristic, the epistemic uncertainty on all entries in the training dataset must fall below ϵ (see *LearnHeuristic* line 18). This is important to ensure that the

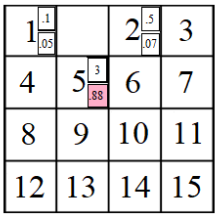
[1] For clarity, the subscripts t, a and e in these equations stand for 'total', 'aleatoric' and 'epistemic' respectively.

next time *GenerateTask* is called, we generate a task that is different to what has previously been trained on.

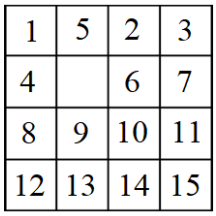
To provide a better intuition for how the algorithms operate, we provide a visual illustration for the 15-puzzle. Consider figure 1. The first step in figure 1a shows the starting point for the algorithm which is the goal state. We sample from the softmax distribution derived from the epistemic uncertainties of the possible previous states. We were more likely to take a step with the DOWN operator but we happened to sample the RIGHT operator. Since the RIGHT operator leads to a state with $\nu_e = 0.25$ which is less than $\epsilon = 1$ we continue to run *GenerateTask*. In the second step shown in figure 1b we again sample from the softmax distribution and this time we sample the DOWN operator which leads to a state with $\nu_e = 3$. Since this value is greater than $\epsilon = 1$ we stop *GenerateTask*. Figure 1c then shows the start state that is used to build the training task to be solved by the heuristic. In this process we are more likely to sample states with high epistemic uncertainly and this makes exploration in task-space more efficient than random task generation.



(a) First step.



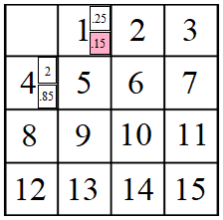
(b) Second step.



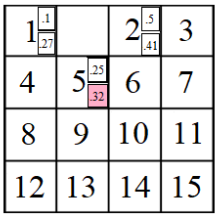
(c) Start state that is used to generate the task.

Figure 1: Hypothetical simulation of *GenerateTask* with $\epsilon = 1$ for the 15-puzzle. The blocks on the top right of a tile represent the epistemic uncertainty. The blocks on the bottom right of a tile represent the derived softmax distribution, with the pink block being the sampled operator.

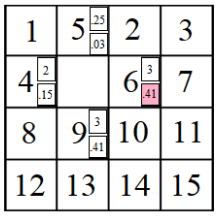
When running *LearnHeuristic* we solve this task and add the plan to the training dataset. Thereafter, we train the heuristic until $\nu_e < \epsilon$ for all states in the training dataset. Suppose in the next iteration of the algorithm we generate a task that takes the same trajectory as in figure 1. As shown in figure 2 we will generate a more complex task because in the previous run of training the heuristic would have reduced the epistemic uncertainty of the state in figure 1c below ϵ. This incremental increase in complexity of tasks makes the heuristic stronger with each iteration of the procedure.



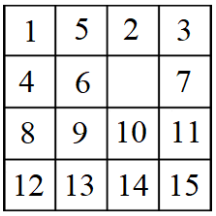
(a) First step.



(b) Second step.



(c) Third step.



(d) Start state that is used to generate the task.

Figure 2: Hypothetical simulation of *GenerateTask* after training on the task that was generated with start state as in figure 1c.

4 Practical Implementation

In section 3 we describe a conceptual algorithm to efficiently learn a likely-admissible heuristic. However, to get an algorithm that works in practice with the setup as described in section 2 with WUNNs where $\mu = \hat{y}$, $\nu_a = \sigma_a^2$ and $\nu_e = \sigma_e^2$ requires additional considerations. In this section we describe these and in the supplementary material we include two additional algorithms, *GenerateTaskPrac* and *LearnHeuristicPrac* that implement them.

Algorithm *GenerateTask* requires us to store every visited state, however, this becomes restrictive if the state-space is large. An alternative approach is to include an additional parameter *MaxSteps* and let the algorithm terminate if this number of steps has been taken. A second modification, that was introduced previously (Arfaee, Zilles, and Holte 2010; 2011), is to exclude at each step the state that would take you back to the previously observed state. This encourages the algorithm to move further from the goal state with every step. Lastly, we use $\mathcal{E}^{rev}$ in our implementation as all our domains have reversable actions.

Algorithm *LearnHeuristic* requires various considerations. For planning it is advantageous to expand as many nodes per second as possible, but working with WUNNs requires multiple passes through the network to get good estimates and this is slow. A simple solution is to learn two separate neural networks: one WUNN that learns σ_e^2 and is only used for generating tasks and one standard feed-forward neural network (FFNN) that learns $\hat{y}$ and σ_a^2 and is only used for planning.

Unfortunately, for planning under our framework we still need σ_e^2 to compute σ_t^2. Furthermore, σ_a^2 is only reliable on data that is similar to what the heuristic has already been

Algorithm 1: GenerateTask generate task using epistemic uncertainty.

Input: { $\mathcal{M}$, // as per equation 4
ϵ // epistemic uncertainty threshold
}
$s' = s_g$ // start at the goal state
initialise a set *visited*$\langle\mathcal{S}\rangle$
repeat
  visited $\cup \{s'\}$
  if *visited* = $\mathcal{S}$ **then**
    exit loop // epistemic uncertainty is below the threshold for every state
  end
  initialise a set *states*$\langle(\mathcal{S}, \mathbb{R}^+)\rangle$
  foreach $s \in \mathcal{E}^{undo}(s')$ **do**
    $x = F(s)$
    compute $\nu_e(x)$ from $\mathcal{M}$
    states $\cup \{(s, \nu_e(x))\}$
  end
  sample from softmax distribution derived from the epistemic uncertainties in *states* to obtain some pair $(s, \nu_e(x))$ // seek uncertainty
  if $\nu_e(x) \geq \epsilon$ **then**
    // found state with epistemic uncertainty above threshold
    $\mathcal{T} = \langle \mathcal{S}, \mathcal{O}, \mathcal{E}, \mathcal{C}, s, s_g \rangle$
    return($\mathcal{T}$)
  end
  $s' = s$
until *forever*

Algorithm 2: LearnHeuristic learn a likely-admissible heuristic.

Input: { *NumTasksPerIter*, // number of tasks per iteration
α, // admissibility probability
ϵ // epistemic uncertainty threshold for *GenerateTask*
}
initialise a model $\mathcal{M}$ as per equation 4
initialise a set $\mathbb{D}\langle(F(\mathcal{S}), \mathbb{R}^+)\rangle$
repeat
  for $i \in 1$: *NumTasksPerIter* **do**
    $\mathcal{T} = GenerateTask(\mathcal{M}, \epsilon)$
    solve $\mathcal{T}$ using $\max(y^\alpha, 0)$ as the heuristic to obtain a plan π // floor heuristic at 0 since $\mathbb{D}$ may have negative support while $h^* \geq 0$
    foreach $s_j \in \pi$ **do**
      if $(s_j \neq s_g)$ **then**
        compute y_j, the cost-to-goal from s_j
        $x_j = F(s_j)$
        $\mathbb{D} \cup \{(x_j, y_j)\}$
      end
    end
  end
  train $\mathcal{M}$ from $\mathbb{D}$ until for all entries (x_i, y_i) we have $\nu_e(x_i) < \epsilon$ and the training error $\sum_i (\mu(x_i) - y_i)^2$ is sufficiently small // ensure *GenerateTask* does not create tasks similar to tasks already trained on
until *forever*

trained on, and can otherwise be highly erroneous. Therefore, we found the following approximation to work well: in our algorithm we maintain a training dataset with observed features / cost-to-goal pairs. Let y^q be quantile q of the cost-to-goal values in the training data. When planning, if $\hat{y} < y^q$ use $\sigma_t^2 = \sigma_a^2$ otherwise use $\sigma_t^2 = \epsilon$. The intuition behind this approximation is that if we are in a region of the feature-space where the mean value is less than a cost-to-goal value that was previously trained on, then we can assume σ_e^2 is negligible and just use σ_a^2. Otherwise, we use ϵ as a lower bound for σ_e^2 and disregard σ_a^2 because we may be in a region of the feature-space where this measure is inaccurate. Since we now have $\sigma_t^2 \leq \sigma_a^2 + \sigma_e^2$, this approximation has the effect of producing a heuristic that has lower admissibility probability for a given value of α.

A second consideration is the handling of β in equation 2. If β is too large then it impossible to achieve the requirement in *LearnHeuristic* line 18 because we cannot decrease the epistemic uncertainty sufficiently. However, if we make β too small then we generate tasks that are too complex for the heuristic to solve quickly. Coupled with this, just ensuring $\sigma_e^2 < \epsilon$ is not sufficient because WUNNs obtain an empirical estimate of the epistemic uncertainty which contains sampling errors. Therefore, it is prudent to ensure that $\sigma_e^2 < \kappa\epsilon$ where $\kappa \in (0, 1)$. In summary our approach is as follows: start with some initial value for β, say $\beta := \beta_0$. Train for some maximum number of iterations *MaxTrainIter* and at each iteration sample a mini-batch of size *MiniBatchSize* from the training data. After each training iteration, if $\sigma_e^2 < \kappa\epsilon$ for all entries in the entire training dataset then stop training early, otherwise complete training for *MaxTrainIter* iterations and then set $\beta := \gamma\beta$ where $\gamma \in (0, 1)$. This criterion ensures that we either reduce the epistemic uncertainty on the training data sufficiently or reduce the importance of the prior in the next iteration.

Next, we consider the choice of α. We want α to be as large as possible so as to encourage admissibility. However, if α is too large then the heuristic can grossly underestimate the optimal cost-to-goal and this leads to more nodes being generated during planning - in fact, this can make planning infeasible. To mitigate this, we first introduce a time limit t_{max} so that if the planner cannot solve a task in t_{max} seconds we stop planning. We start with an initial value for α, say $\alpha := \alpha_0$, and we try to solve all *NumTasksPerIter* tasks with y^α within t_{max} seconds. We keep track of how many tasks

are solved and if the number of solved tasks in an iteration is less than some threshold *NumTasksPerIterThresh* we update $\alpha := \max(\alpha - \Delta, 0.5)$ where $\Delta > 0$. The idea behind this update is that if we cannot solve enough tasks in an iteration we sacrifice admissibility so that we can make learning progress. We floor the value of α at 0.5 which then reverts to the mean value $\hat{y}$.

Additional considerations include 1) we must cut-off *LearnHeuristicPrac* after some *NumIter* number of iterations; 2) Instead of a set $\mathbb{D}$ we use a list *MemoryBuffer* for the training data and we store only the most recent *MemoryBufferMaxRecords* added to the list; 3) when computing σ_e^2 with the WUNN we must decide on the number of samples K to use from the network; and 4) we must decide on appropriate priors for the weights of the WUNN μ_0 and σ_0^2.

5 Experiments

In this section we demonstrate our algorithm on the 15-puzzle, 24-puzzle, 24-pancake and 15-blocksworld domains. In all domains, we used the following parameter values: $\epsilon = 1$, *NumTasksPerIter* $= 10$, *NumTasksPerIterThresh* $= 6$, $\alpha_0 = 0.99$, $\Delta = 0.05$, $\beta_0 = 0.05$, $\kappa = 0.64$, *MemoryBufferMaxRecords* $= 25,000$, $\mu_0 = 0$, $\sigma_0^2 = 10$, $q = 0.95$, $K = 100$. The number of iterations *NumIter* depends on the experiment as we describe further below. We set γ as a function of *NumIter* by specifying $\beta_{NumIter} = 0.00001$ and solving $\gamma^{NumIter}\beta_0 = \beta_{NumIter}$. To reduce variance when training the WUNN we use the local reparameterisation trick (Kingma, Salimans, and Welling 2015). All the networks use $relu$ activations, have a single hidden layer and are trained with the Adam optimiser. For the WUNN we use an initial learning rate of 0.01 and $S = 5$ Monte Carlo samples to approximate the objective function in equation 2. For the FFNN we use an initial learning rate of 0.001. The network weights were initialised with He Normal initialization (He et al. 2015) while the biases were initialised to 0. We use IDA* for planning. When running the final learned heuristic on test tasks we use $\sigma_t^2 = \sigma_a^2$ - this makes the assumption that epistemic uncertainty is negligible for all states by the end of training.

For training the FFNN we train for *TrainIter* $= 1000$ iterations using the entire memory buffer at each iteration. For training the WUNN we train for *MaxTrainIter* $= 5000$ iterations using a minibatch size of *MiniBatchSize* $= 100$ and we use a sampling scheme that gives more weight to states with epistemic uncertainty greater than or equal to $\kappa\epsilon$. This can be achieved by sampling from the following (unnormalised) distribution, without replacement:

$$f(x) = \begin{cases} \exp(\sigma_e(x)) & \text{if } \sigma_e^2(x) \geq \kappa\epsilon, \\ \exp(-C) & \text{otherwise,} \end{cases}$$

where $C > 0$. Since $\sigma_e \geq 0$ this distribution gives more weight to states with epistemic uncertainty that still need to be reduced below the required threshold. We use $C = 1$ in our implementation. In all our experiments we report average statistics over multiple runs and we include more detailed tables with standard deviations for each statistic in the supplementary material.

Suboptimality Experiments for the 15-puzzle

For the 15-puzzle we use a low-level feature representation of the raw game-state and 20 hidden neurons for both the FFNN and WUNN. Previously, a representation was described that requires 16^2 bits where for each 16 puzzle number, the location of that number is encoded with a 16 bit one-hot vector (Ernandes and Gori 2004). We use a more efficient encoding that requires only $16 \times 2 \times 4$ bits where for each 16 puzzle number the horizontal and vertical locations of that number are encoded with a 4 bit one-hot vector.

For the first experiment we aim to show that our approach results in low suboptimality. We run the algorithm *LearnHeuristicPrac* to learn the neural heuristic. We run the procedure for *NumIter* $= 50$ iterations, $t_{max} = 60$ and *MaxSteps* $= 1,000$. Once training is completed, we test the performance of our algorithm on the standard 100 15-puzzle benchmark tasks (Korf 1985) that have an average optimal cost-to-goal of 53.05.

One problem that arises when using a representation of the raw game-state is that such a representation does not exhibit aleatoric uncertainty since the network can, in principle, learn a deterministic output for each input state. To alleviate this, we use dropout which is a popular technique to inject noise into the training process by dropping out neurons randomly during training. We use a dropout rate of 2.5% in the hidden layer when training the FFNN. Dropout is commonly seen as a way to reduce overfitting (Srivastava et al. 2014) but can also be viewed as a technique to estimate aleatoric uncertainty (Osband 2016). In this context, after dropout is applied, the latent feature-space of the hidden layer is insufficient to deterministically explain the cost-to-goal observations which induces aleatoric uncertainty.

In addition, we train a second neural heuristic that uses *LearnHeuristicPrac* but we make a modification that the FFNN has one output neuron and therefore planning and learning is done with $\hat{y}$ only. This is consistent with previous approaches that learn neural heuristics (Arfaee, Zilles, and Holte 2010; 2011; Ernandes and Gori 2004). Results are shown in table 1 and are averages over 10 independent runs. The experiment that uses a single output FFNN has N/A under the α column while MD corresponds to using the admissible Manhattan Distance heuristic. Columns "Generated", "Time", "Subopt" and "Optimal" are the average nodes generated, planning time (in seconds), suboptimality [2] and percentage tasks solved optimally respectively.

We see that the single output FFNN produces a neural heuristic that produces a high suboptimality of 10.75%. This result conforms to what was previously described in the literature in that errors compound with each iteration (Arfaee, Zilles, and Holte 2010; 2011). Using our approach, we can achieve far lower suboptimality - for example, only 2.46% for $\alpha = 0.90$. However, we note that using $\alpha = 0.9$ expands 9.76 times more nodes. This suggests that the improved optimality achieved in our algorithm comes at the price of increased planning time. However, we can adjust the trade-off

[2] Average suboptimality is computed by taking $u_i = \frac{y_i}{y_i^*} - 1$ for each test task where y_i is the cost-to-goal obtained from the planner and y_i^* is the optimal cost-to-goal, then averaging all u_i values.

between planning time and suboptimality by varying α. We see that as α increases the trend is for suboptimality to decrease while nodes generated increases.

α	Time	Generated	Subopt	Optimal
MD	80.7	363,028,080	0.0%	100%
0.95	74.6	78,787,262	2.2%	67.8%
0.9	26.7	29,342,747	2.5%	65.2%
0.75	8.7	9,357,055	3.0%	59.0%
0.5	5.1	5,284,645	3.4%	52.3%
0.25	4.9	5,107,840	4.5%	38.3%
0.1	3.9	4,285,483	5.3%	30.7%
0.05	3.8	4,189,753	5.6%	25.3%
N/A	2.3	3,071,956	10.8%	10.9%

Table 1: Suboptimality results for 15-puzzle.

We note that previous work that trained on optimal plans and that used 4 separate neural networks to get better heuristic estimates was able to solve 63.86% tasks optimally (Ernandes and Gori 2004). Meanwhile our approach with $\alpha = 0.9$ is able to achieve a slightly higher percentage of 65.2%. [3] We use only a single network for planning while we do not have access to *any training data of optimal plans*.

Efficiency Experiments for the 15-puzzle

For the second experiment on the 15-puzzle we show that our approach to generating tasks using *GenerateTaskPrac* is more efficient than random task generation. In particular, previous work has suggested an alternative way to generate tasks (Arfaee, Zilles, and Holte 2010; 2011). The idea is to simply increment the number of steps at each iteration using a parameter *LengthInc*. For example, if *LengthInc* $= 10$ we would first generate *NumTasksPerIter* tasks by taking 10 steps at random from the goal state, then 20 steps, then 30 steps and so on.

Two key considerations are required to illustrate efficiency. Firstly, an efficient algorithm would learn about the domain from a small number of tasks because it would generate tasks in accordance with the amount of new knowledge it can gain from solving them. Secondly, an efficient algorithm would solve new tasks quickly because it would generate tasks that are incrementally more difficult than previously solved tasks.

Given these considerations our experimental setup is as follows: we run *LearnHeuristicPrac* but we train a FFNN with only a single output neuron that plans and learns with $\hat{y}$. For this experiment we keep the number of tasks small, so *NumIter* $= 20$ and we keep the time limit to solve each task small as well, so $t_{max} = 1$. We then train additional FFNN networks using *LearnHeuristicPrac* but this time instead of using *GenerateTaskPrac* to generate tasks, we generate tasks using fixed steps back from the goal with *LengthInc* $\in \{1, 2, 4, 6, 8, 10\}$.

[3] The statistic 63.86% was obtained from 700 test tasks with an average optimal cost-to-goal of 52.62 that were not made public. Meanwhile, we test on the standard 100 15-puzzle benchmark tasks that have an average optimal cost-to-goal of 53.05.

Once the neural heuristics are trained we test performance as follows: we generate a set of 100 test tasks $\{\mathcal{T}_i^{test}\}_{i=1}^{100}$. Task $\mathcal{T}_k^{test}$ is generated by taking k steps back at random from the goal, so that each task in the test set should be more difficult to solve than the previous one. We then give each of the neural heuristics a time limit of 60 seconds to solve as many test tasks as possible. We repeat this process independently 10 times across each neural heuristic and we furthermore run the experiment over 10 independent runs.

Table 2 shows our results. The experiment that uses *GenerateTaskPrac* to generate tasks has GTP under the *LengthInc* column. Columns "Solved Train" and "Solved Test" are the average percentage of tasks solved during training and at test time respectively. Consider the case *LengthInc* $= 1$. In this case 100% of training tasks are solved. However, because each successive iteration only takes 1 step back from the goal and there are only a small number of tasks to learn from, this neural heuristic only solves 38.59% of the test tasks. Now consider the case *LengthInc* $= 10$. In this case large steps are taken back from the goal but because there is a short time limit to solve each task, this neural heuristic can only solve 11.80% of training tasks. As a result, the network doesn't learn from enough tasks during training and can therefore only solve 31.83% of the test tasks.

Our approach finds a balance between these two restrictions. The training tasks generated are incrementally more difficult than previously observed training tasks because we stop execution of the algorithm as soon as we observe a state with high epistemic uncertainty. We can therefore solve 93.3% of tasks during training. Furthermore, because our approach seeks uncertainty when generating tasks, each new task that is trained on provides useful domain knowledge for the neural heuristic to learn from. Therefore, we can solve 60.59% of the test tasks. In fact, our approach outperforms all the values of *LengthInc* $\in \{1, 2, 4, 6, 8, 10\}$ in terms of performance on the test tasks. A further advantage of our approach is that the parameter *LengthInc*, which is domain dependant, does not need to be tuned for each new domain - we use the same β_0 and the same approach to set γ for all experiments across the domains.

LengthInc	Solved Train	Solved Test
1	100%	38.6%
2	95.1%	48.2%
4	61.8%	51.4%
6	36.2%	39.6%
8	19.2%	35.2%
10	11.8%	31.8%
GTP	93.3%	60.6%

Table 2: Efficiency results for 15-puzzle.

Experiments for Other Domains

We run the same experiments for suboptimality on the 24-puzzle, 24-pancake and 15-blocksworld domains. These domains have large state-spaces and so using a low-level representation is more difficult than in the case of the 15-puzzle. Instead, we use high-level feature representations as input

to the network that are similar to previous approaches (Arfaee, Zilles, and Holte 2010; 2011). The features used make extensive use of pattern databases (PDBs) (Korf and Felner 2002). PDBs are a general method of generating admissible heuristics, making them a good choice of features for a variety of domains. We use *NumIter* $= 75$, $t_{max} = 5 \times 60$ and *MaxSteps* $= 5000$ for these domains due to their increased complexity over the 15-puzzle. We use 8 neurons for the hidden layer for both the FFNN and WUNN and we do not use dropout because the feature representations now naturally induce aleatoric uncertainty. When planning we now use $\max(y^{\alpha}, h^{ad})$ where h^{ad} is the maximum of all the admissible heuristics passed as features to the networks.

24-puzzle: we use 2 sets of disjoint 5-5-5-5-4 PDBs and their sums as features (f1-f12). Additionally, we have a feature with the number of out-of-place tiles (f13), a feature for the Manhattan distance (f14) and a feature for the position of the blank tile (f15). The total number of features for this domain is 15 comprising of 14 admissible heuristics (f1-14). For testing we use the standard 50 24-puzzle benchmark tasks that have an average optimal cost-to-goal of 100.78 (Korf and Taylor 1996).

24-pancake: we use 2 sets of location-based disjoint 5-5-5-5-4 PDBs (Yang et al. 2008) and their sums as features (f1-f12). Additionally, we have a binary feature indicating whether the middle pancake is out of place, and the number of the largest out-of-place pancake. The total number of features for this domain is 14 comprising of 12 admissible heuristics (f1-f12). For testing, we generate 50 random tasks that have an average optimal cost-to-goal of 22.56.

15-blocksworld: we use 12 4-block PDBs (f1-f12), the number of out of place blocks (f13), and the number of stacks of blocks (f14) as features. The total number of features for this domain is 14 comprising of 13 admissible heuristics (f1-f13). The goal state is chosen as the state with all blocks in one stack. For testing, we generate 50 random tasks that have an average optimal cost-to-goal of 21.72.

Table 3 show our results which average over 5 independent runs. The rows with "Boot" under the α column are results from a competing method that learns a single output FFNN (Arfaee, Zilles, and Holte 2010), while "Gap" is the domain specific gap heuristic for the 24-pancake (Helmert 2010) and "PDB" uses a memory intensive heuristic that comprises of disjoint 6-6-6-6 PDBs with partially created disjoined 8-8-8 PDBs for the 24-puzzle (Felner and Adler 2005). We see that, as in the case of the 15-puzzle, the single output FFNNs have high suboptimality. With our approach we can achieve *far lower suboptimality*, albeit at the price of longer planning times. For example, with $\alpha = 0.9$ we obtain 2.6%, 1.3% and 0.07% suboptimality while previous approaches obtain 6.1%, 10.3% and 6.9% suboptimality for the 24-puzzle, 24-pancake and 15-blocksworld domains respectively.[4] As in the case of the 15-puzzle, the trend is that as α increases, suboptimality decreases while nodes generated increases. These experiments also illustrate that our implementation is *robust* as we use the same parameters across three domains and obtain similar outcomes. We also emphasise that our implementation works with both low-level and high-level feature representations.

α	Time	Generated	Subopt	Optimal
		24-puzzle		
PDB	?	$65,135,068,005$	0.0%	100%
0.95	$2,665$	$1,233,965,823$	2.2%	28.8%
0.9	$1,371$	$628,101,474$	2.6%	20.0%
0.75	549	$274,003,465$	3.7%	5.2%
0.5	189	$99,244,234$	4.6%	1.2%
0.25	121	$66,147,586$	5.2%	0.0%
0.1	87	$46,988,530$	5.8%	0.0%
0.05	84	$40,046,361$	6.3%	0.0%
N/A	25	$11,719,659$	11.3%	0.0%
Boot	274	$164,589,698$	6.1%	?
		24-pancake		
Gap	0.03	$48,599.5$	0.0%	100%
0.95	365	$104,132,601$	1.1%	76.0%
0.9	199	$54,089,822$	1.3%	72.4%
0.75	54	$13,001,211$	1.9%	59.2%
0.5	20	$4,530,281$	2.2%	53.2%
0.25	12	$2,511,066$	3.5%	37.2%
0.1	8	$1,162,755$	3.8%	30.8%
0.05	5	$871,908$	4.0%	30.8%
N/A	1	$210,622$	10.6%	8.4%
Boot	2	$1,856,645$	10.3%	?
		15-blocksworld		
0.95	56	$115,691,681$	0.02%	99.6%
0.9	54	$112,390,208$	0.07%	98.4%
0.75	51	$101,109,757$	0.23%	95.6%
0.5	38	$69,663,441$	1.0%	84.8%
0.25	44	$63,963,572$	4.3%	50.8%
0.1	36	$50,951,658$	9.7%	34.0%
0.05	29	$42,499,655$	13.4%	24.0%
N/A	21	$31,178,090$	7.1%	38.4%
Boot	16	$3,651,438$	6.9%	?

Table 3: Suboptimality results for 24-puzzle, 24-pancake and 15-blocksworld.

A final point of discussion is that our approach to learning heuristics takes longer to train because solving tasks with a likely-admissible heuristic requires more nodes to be generated during planning. We include a table with runtimes in the supplementary material.

6 Concluding Remarks

In this paper we have utilised uncertainty to efficiently learn likely-admissible heuristics. Our approach uses epistemic uncertainty to explore task-space and generate training tasks of the right level to learn from. Meanwhile, we combine epistemic and aleatoric uncertainty to learn a likely-admissible heuristic that leads to low suboptimality. We illustrate our approach empirically with a practical implementation based on Bayesian neural networks and demonstrate that it results in low suboptimality that outperforms competing methods across a variety of domains.

[4] For testing, the bootstrap method uses the standard 50 benchmark tasks for the 24-puzzle, 1000 tasks with an average optimal cost-to-goal of 22.75 for the 24-pancake and 200 tasks with an average optimal cost-to-goal of 22.73 for the 15-blocksworld.

Acknowledgements

The authors wish to thank the anonymous reviewers for their thorough feedback and helpful comments.

References

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2010. Bootstrap learning of heuristic functions. *Proceedings of the 3rd Annual Symposium on Combinatorial Search* 52–60.

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2011. Learning heuristic functions for large state spaces. *Artificial Intelligence* 175(16-17):2075–2089.

Blundell, C.; Cornebise, J.; Kavukcuoglu, K.; and Wierstra, D. 2015. Weight uncertainty in neural networks. *Proceedings of the 25th International Conference on Machine Learning*.

Ernandes, M., and Gori, M. 2004. Likely-admissible and sub-symbolic heuristics. *Proceedings of the 16th European Conference on Artificial Intelligence* 613–617.

Felner, A., and Adler, A. 2005. Solving the 24 puzzle with instance dependent pattern databases. *Proceedings of the 6th International Symposium on Abstraction, Reformulation and Approximation* 248–260.

Gal, Y., and Ghahramani, Z. 2015. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. *arXiv:1506.02142*.

Groshev, E.; Tamar, A.; Srivastava, S.; and Abbeel, P. 2017. Learning generalized reactive policies using deep neural networks. *arXiv:1708.07280*.

Haslum, P., and Scholz, U. 2003. Domain knowledge in planning: Representation and use. *Workshop on PDDL at International Conference on Automated Planning and Scheduling*.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *arXiv:1502.01852*.

Helmert, M. 2010. Landmark heuristics for the pancake problem. *Proceedings of the 3rd International Symposium on Combinatorial Search* 109–110.

Hernandez-Lobato, J. M., and Adams, R. 2015. Probabilistic backpropagation for scalable learning of bayesian neural networks. *Proceedings of the 25th International Conference on Machine Learning*.

Higgins, I.; Matthey, L.; Pal, A.; Burgess, C.; Glorot, X.; Botvinick, M.; Shakir, M.; and Lerchner, A. 2017. beta-vae: Learning basic visual concepts with a constrained variational framework. *Proceed of the 5th International Conference on Learning Representations*.

Junghanns, A., and Schaeffer, J. 2001. Sokoban: Enhancing general single agent search methods using domain knowledge. *Artificial Intelligence* 129 (1-2):219–251.

Kendall, A., and Gal, Y. 2017. What uncertainties do we need in bayesian deep learning for computer vision? *ArXiv:1703.04977*.

Kingma, D.; Salimans, T.; and Welling, M. 2015. Variational dropout and the local reparameterization trick. *Proceedings of the 25th Advances in Neural Information Processing Systems*.

Korf, R. E., and Felner, A. 2002. Disjoint pattern database heuristics. *Artificial Intelligence* 22:9–22.

Korf, R. E., and Taylor, L. A. 1996. Finding optimal solutions to the twenty-four puzzle. *Proceedings of the 13th National Conference on Artificial Intelligence* 1202–1207.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

McAleer, S.; Agostinelli, F.; Shmakov, A.; and Baldi, P. 2018. Solving the rubik's cube without human knowledge. *arXiv:1805.07470*.

Murphy, K. P. 2012. *Machine Learning A Probabilistic Perspective*. MIT Press.

Osband, I. 2016. Risk versus uncertainty in deep learning: Bayes, bootstrap and the dangers of dropout. *Workshop on Bayesian Deep Learning at Advances in Neural Information Processing System*.

Pearl, J. 1984. *Heuristics*. Addison-Wesley.

Samadi, M.; Felner, A.; and Schaeffer, J. 2008. Learning from multiple heuristics. *Proceedings of the 23rd National Conference on Artificial Intelligence* 357–362.

Shridhar, K.; Laumann, F.; and Liwicki, M. 2019. A comprehensive guide to bayesian convolutional neural network with variational inference. *arXiv:1901.02731*.

Slaney, J., and Thiébaux, S. 2001. Blocks world revisited. *Artificial Intelligence* 125:119–153.

Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I.; and Salakhutdinov, R. 2014. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research* 15(1):1929–1958.

Sun, S.; Chen, C.; and Carin, L. 2017. Learning structured weight uncertainty in bayesian neural networks. *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics* 1283–1292.

Thayer, J. T.; Dionne, A. J.; and Ruml, W. 2011. Learning inadmissible heuristics during search. *Proceedings of the 21st International Conference on Automated Planning and Scheduling*.

Yang, F.; Culberson, J.; Holte, R.; Zahavi, U.; and Felner, A. 2008. A general theory of additive state space abstractions. *Artificial Intelligence* (32):631–662.

Zahavi, U.; Felner, A.; Holte, R.; and Schaeffer, J. 2008. Duality in permutation state spaces and the dual search algorithm. *Artificial Intelligence* 172 (4-5):514–540.