Journal of Artificial Intelligence Research ? (?) 1-43 Submitted 5/12; published ?

Learning Inadmissible Heuristics During Search

Jordan T. Thayer jtd7 AT cs.unh.edu
Austin Dionne austin.dionne AT gmail.com
Wheeler Ruml ruml AT cs.unh.edu
Department of Computer Science
University of New Hampshire
33 Academic Way
Durham, NH 03824 USA

Abstract

When resources are plentiful, heuristic search problems can be solved optimally using algorithms such as A* and IDA*. However, in many practical settings, we must accept suboptimal solutions in order to reduce the resources required for heuristic search. While optimal search algorithms are limited to admissible heuristics, suboptimal search algorithms can exploit potentially more accurate inadmissible heuristics which may overestimate the cost-to-go. One way to construct such heuristics is through learning. Most previous work has focused on learning heuristics before the search, often using a large corpus of training instances. In this paper, we examine the complementary approach of learning improved heuristics during the solving of a single instance. Learning during solving does not require training instances or precomputation, or for the learned heuristic to generalize to another instance. We propose a new method for learning heuristics based on observing the behavior of an existing heuristic along paths in the search space. The method is simple and general, relying only on information readily available in any best-first search. We demonstrate its usefulness in greedy search and in a new bounded suboptimal search algorithm, called skeptical search, that is capable of using inadmissible heuristics while respecting a bound on solution suboptimality. In experiments across eight benchmark domains, we find that heuristics learned online result in both faster search and better solutions.

1. Introduction

Heuristic search is a widespread approach to automated planning and problem solving. If time and memory permit, we can use algorithms such as A* (Hart, Nilsson, & Raphael, 1968) to find solutions of minimal cost. These algorithms require an admissible heuristic evaluation function, that is, a heuristic that never over-estimates the true cost-to-go from a node to a goal. Under mild assumptions, it can be shown that no similarly informed algorithm can find provably optimal solutions while performing less work than A* (Dechter & Pearl, 1988). Unfortunately, problems are often too large and deadlines are often too short for finding provably optimal solutions (Helmert & Röger, 2007). When solving a problem optimally is impractical, suboptimal search can be a practical alternative. Suboptimal search algorithms sacrifice solution optimality in an attempt to reduce the resources needed for solving problems.

We will focus on two types of suboptimal search algorithms: greedy best-first search algorithms that attempt to find solutions of high quality as quickly as possible while providing no guarantees on solution quality, and bounded suboptimal search algorithms that return solutions whose cost is guaranteed to be within some user-provided factor of optimal. Suboptimal search algorithms tend to be faster than their optimal counterparts because they do not need to prove that the solutions they

return are optimal. By not proving solution optimality, they avoid having to expand all nodes that could potentially lead to a solution of lower cost. Because suboptimal search algorithms do not prove solution optimality, they can consider inadmissible sources of heuristic guidance.

This paper investigates learning as a way to construct these inadmissible estimates of cost-to-go. We are not the first to consider guiding search algorithms with inadmissible learned heuristics. As we later discuss in detail, several authors have proposed learning informed inadmissible heuristics by recording for many states the true cost-to-go, which we call h^*, and a set of features. They then learn a function from the features to a potentially inadmissible estimate of the cost-to-go, which we call $\widehat{h}$. Such an approach makes the limiting assumption that we either have access to a representative training set or the ability to generate one automatically, and sufficient resources to find h^* for many states. It further assumes that the training instances and test instances are similar enough to one another for the learning on the training instance to transfer effectively to the instances we truly care about solving. This can be problematic in settings, such as STRIPS planning, where instances can be very different from one another because of the expressivity of the problem description language.

In this paper, we demonstrate that learning heuristics during search itself is a practical and effective alternative to learning before search or learning interleaved with search. In Section 2, we present a new technique for improving heuristics during the execution of search, called single-step correction. It improves a given initial heuristic based on observing its behavior over paths in the search tree. We demonstrate that it works well in practice in an empirical study across eight benchmark domains. Heuristics learned during search find solutions up to three orders of magnitude faster than the base heuristic, and they also tend to improve solution quality substantially. We introduce a new algorithm, skeptical search, that is capable of using arbitrary inadmissible heuristics. In Section 3, we show that, although heuristics learned either offline or in between search episodes are often substantially more accurate than those learned online, they provide worse guidance, leading to slower solving of instances. In Section 4 we compare against work aimed a learning heuristics using a set of instances. Other related work is summarized in Sections 5 and 6.

⟨FIX: ADD TAXONOMY OF TIMES FOR LEARNING⟩

2. Learning During Search

Heuristic evaluation functions are the distinguishing component of heuristic search algorithms. Notated $h(n)$, these functions estimate the cost of the cheapest completion of a given node n, that is, the cost of the cheapest sequence of actions transforming the state represented by node n into a goal state. Our starting observation is that the optimal cost of a solution beneath some node p is the cost of completing its best child plus the cost of transition to that best child. More formally, let $h^*(n)$ represent the perfect heuristic function that exactly predicts the cost-to-go for all nodes. For any parent node p, if $bc(p)$ is the next node along an optimal path from p to a goal and $c(p, bc(p))$ is the cost of the transition between p and $bc(p)$, then:

$$h^*(p) = h^*(bc(p)) + c(p, bc(p)) \tag{1}$$

This is a slight generalization of move invariance (Christensen & Korf, 1986), which holds that the entire node evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of arriving at node n, should not vary between a parent and its best child. Here, rather than trying to hold $f(n)$ constant across nodes, we're trying to force the heuristic to differ by exactly $c(p, bc(p))$. A little algebra

shows us that these are equivalent:

$$\begin{aligned} f^*(p) &= f^*(bc(p)) \\ g(p) + h^*(p) &= g(bc(p)) + h^*(bc(p)) \\ h^*(p) &= h^*(bc(p)) + (g(bc(p)) - g(p)) \\ h^*(p) &= h^*(bc(p)) + c(p, bc(p)) \end{aligned}$$

Obviously, during the course of search, we do not have access to perfect heuristics. If we did, search would be unnecessary. We would simply perform hillclimbing from the root, expanding only those nodes along the solution. However, every time an imperfect heuristic deviates from the relationships described above, we have observed a mistake. Every observed mistake is an opportunity to learn an improvement to the underlying heuristic function. In this way, our perspective is that of temporal difference learning (Sutton, 1988). Using temporal difference learning to improve heuristics has been suggested before (Nilsson, 1998, pages 172-175), but to our knowledge never actually implemented and evaluated until this work. In the next section, we present the details of our approach.

2.1 Single-Step Error Corrections

We can measure the error in a heuristic for a single step by comparing heuristic values between the parent and the best child. With a measurement of the error across a single step, we can attempt to correct for the error by estimating the number of steps to go and adjusting the heuristic estimates accordingly. As shown in Equation 1, there is a relationship between the cost-to-go estimates of a parent and its best child. This allows us to define the single-step error in h at p as:

$$\epsilon_{h_p} = (h(bc(p)) + c(p, bc(p))) - h(p) \tag{2}$$

The sum of the cost-to-go heuristic and the single-step errors from a node p to the goal equals the true cost-to-go:

Theorem 1 *For any node p with a goal beneath it:*

$$h^*(p) = h(p) + \sum_{n \in p \rightsquigarrow goal} \epsilon_{h_n} \tag{3}$$

where $p \rightsquigarrow goal$ is the set of nodes along the path between the node p and the goal, including p and excluding the goal. ϵ_{h_n} is the single-step error in h between a node n and its best child.

Proof: The proof is by induction over the nodes in the path. For our base case, we show that when $bc(p)$ is the goal, Equation 3 holds:

$$\begin{aligned} h^*(p) &= c(p, bc(p)) && \text{because } bc(p) \text{ is the goal} \\ &= h(p) + c(p, bc(p)) - h(p) && \text{by algebra} \\ &= h(p) + c(p, bc(p)) + h(bc(p)) - h(p) && \text{because } h(bc(p)) = 0 \\ &= h(p) + \epsilon_{h_p} && \text{by Eq. 2} \\ &= h(p) + \textstyle\sum_{n \in p \rightsquigarrow goal} \epsilon_{h_n} && \text{because } p \rightsquigarrow goal = \{p\} \end{aligned}$$

As the best child of p was a goal, the optimal cost of completing p is exactly the arc cost from p to its best child.

For the inductive case, assuming that Equation 3 holds for $bc(p)$, we show that it holds for its parent p as well:

$$\begin{aligned} h^*(p) &= c(p, bc(p)) + h^*(bc(p)) && \text{by Eq. 1} \\ &= c(p, bc(p)) + h(bc(p)) + \textstyle\sum_{n \in bc(p) \rightsquigarrow goal} \epsilon_{h_n} && \text{by inductive assumption} \\ &= h(p) + \epsilon_{h_p} + \textstyle\sum_{n \in bc(p) \rightsquigarrow goal} \epsilon_{h_n} && \text{by Eq. 2} \\ &= h(p) + \textstyle\sum_{n \in p \rightsquigarrow goal} \epsilon_{h_n} && \text{by def. of } \rightsquigarrow \end{aligned}$$

which is exactly Equation 3, completing the proof. □

We define the mean one-step error $\bar{\epsilon}_h$ along the path from p to the goal as:

$$\bar{\epsilon}_{h_p} = \frac{\sum_{n \in p \rightsquigarrow goal} \epsilon_{h_n}}{d^*(p)} \tag{4}$$

where $d^*(p)$ is the length of the cost-optimal path between p and a goal. It is important to remember that the mean single-step error is defined in terms of the true length (number of arcs) of the remaining path, $d^*(p)$, and not the cost (sum of weights) of the remaining path, $h^*(n)$. We will reconsider this decision in Section 3.1, and while both approaches can be shown to be technically correct, using path length provided better performance empirically. Solving Equation 4 for $\sum_{n \in p \rightsquigarrow goal} \epsilon_{hn}$ yields:

$$\sum_{n \in p \rightsquigarrow goal} \epsilon_{hn} = d^*(p) \cdot \bar{\epsilon}_{h_p} \tag{5}$$

Substituting Equation 5 into Equation 3 we see that:

$$h^*(p) = h(p) + d^*(p) \cdot \bar{\epsilon}_{h_p} \tag{6}$$

This forms the basis for the *single step* heuristic correction method.

In a realistic setting, we are not going to have access to the true distance-to-go $d^*(n)$, and so we cannot use Equation 6 to produce an improved cost-to-go estimate directly. Given the important role that distance plays in Equation 6, we will assume that a heuristic estimate of search distance-to-go, call it $d(n)$, is available (In Section 3.1, we will consider heuristic correction without $d(n)$). If this assumption seems strong, note that in domains in which all actions have equal cost, $d(n) = h(n)$. In other domains, one can usually construct a distance-to-go heuristic using methods very similar to those for the cost-to-go heuristic. For example, one can track the number of actions required to solve a simplified version of the problem, in addition to the cost of those actions. Further examples are given by Pearl and Kim (1982), Ghallab and Allard (1983), and Thayer and Ruml (2009).

Just as we correct a given $h(n)$, we will want to correct $d(n)$. We take a similar strategy as before. In analogy to Equation 1, the perfect distance-to-go estimate $d^*(n)$ obeys:

$$d^*(p) = 1 + d^*(bc(p)) \tag{7}$$

Notice that $c(p, bc(p))$ has been replaced with 1 in the previous equation. That is because while a transition between two nodes may have a wide range of weights assigned to it, a distance estimate only cares about the number of transitions. When we are not working with perfectly informed heuristics, we must introduce a term that represents the error ϵ_{d_p} present in the heuristic when evaluated at a parent p and its best child:

$$d(p) = 1 + d(bc(p)) + \epsilon_{d_p} \tag{8}$$

Solving for the one-step distance error of the parent ϵ_{d_p}, we get:

$$\epsilon_{d_p} = (1 + d(bc(p))) - d(p) \tag{9}$$

Note that the single-step error is specific to the node p. Imagine a situation where several nodes, each with a different distance-to-go estimate, all generate the same goal node as their only child. All nodes share a best child, but each has a different single-step error. As a result, the error is specific to the generating node. We require that the best child selected for this calculation not represent the parent state of p. Thus, states with no children other than the inverse action back to their parent have no associated ϵ_d. Goals also have no best child. Using Equation 9, we prove the following analogue of Theorem 1:

Theorem 2 *For any node p with a goal beneath it:*

$$d^*(p) = d(p) + \sum_{n \in p \rightsquigarrow goal} \epsilon_{d_n} \tag{10}$$

where $p \rightsquigarrow goal$ is the set of nodes along an optimal path between the node p and a goal, including p and excluding the goal.

Proof: The proof is by induction over the nodes in the path. For our base case, we show that Equation 10 holds when $bc(p)$ is the goal:

$$\begin{array}{rcll} d^*(p) & = & 1 & \text{because } bc(p) \text{ is a goal} \\ & = & d(p) + 1 - d(p) & \text{by algebra} \\ & = & d(p) + 1 + d(bc(p)) - d(p) & \text{because } d(bc(p)) = 0 \\ & = & d(p) + \epsilon_{d_p} & \text{by Equation 9} \\ & = & d(p) + \sum_{n \in p \rightsquigarrow goal} \epsilon_{d_n} & \text{because } p \rightsquigarrow goal = \{p\} \end{array}$$

As the best child of p was a goal, p is obviously a single step away from the goal and the base case holds.

For the inductive case we show that by assuming that Equation 10 holds for $bc(p)$, we can show that it holds for its parent p as well:

$$\begin{array}{rcll} d^*(p) = & 1 + d^*(bc(p)) & & \text{by Eq. 7} \\ = & 1 + d(bc(p)) + \sum_{n \in bc(p) \rightsquigarrow goal} \epsilon_{d_n} & & \text{by inductive assumption} \\ = & d(p) + \epsilon_{d_p} + \sum_{n \in bc(p) \rightsquigarrow goal} \epsilon_{d_n} & & \text{by Eq. 9} \\ = & d(p) + \sum_{n \in p \rightsquigarrow goal} \epsilon_{d_n} & & \text{by def. of } \rightsquigarrow \text{ and } bc \end{array}$$

which is exactly Equation 10, completing the proof. □

We can define the mean one-step error $\bar{\epsilon}_{d_p}$ along the path from p to the goal as:

$$\bar{\epsilon}_{d_p} = \frac{\sum_{n \in p \rightsquigarrow goal} \epsilon_{d_n}}{d^*(p)} \tag{11}$$

Using Equations 10 and 11, we can define $d^*(p)$ in terms of $\bar{\epsilon}_d$:

$$d^*(p) = d(p) + d^*(p) \cdot \bar{\epsilon}_{d_p} \tag{12}$$

Solving Equation 12 for $d^*(p)$ yields:

$$d^*(p) = \frac{d(p)}{1 - \bar{\epsilon}_{d_p}} \tag{13}$$

Another way to think of Equation 13 is as the closed form of an infinite geometric series that recursively accounts for error in $d(p)$:

$$\begin{aligned} d^*(p) &= d(p) + d(p) \cdot \bar{\epsilon}_{d_p} + (d(p) \cdot \bar{\epsilon}_{d_p}) \cdot \bar{\epsilon}_{d_p} + \ldots & (14) \\ &= d(p) \cdot \textstyle\sum_{i=1}^{\infty} (\bar{\epsilon}_{d_p})^i & (15) \end{aligned}$$

This series takes the average single-step error, $\bar{\epsilon}_d$, and assumes that we will observe that error during each step that $d(n)$ is predicting. This results in some number of additional steps. Unfortunately, the mean single-step error will also be observed in the additional steps. Naturally, this results in more steps, during which the error will again be observed. This process recurs, resulting in the infinite series.

Substituting our compact equation for d^* (Equation 13) into our equation for h^* (Equation 6), we have:

$$h^*(p) = h(p) + \frac{d(p)}{1 - \bar{\epsilon}_{d_p}} \cdot \bar{\epsilon}_{h_p} \tag{16}$$

Given Equations 13 and 16, if we had both $\bar{\epsilon}_{d_n}$ and $\bar{\epsilon}_{h_n}$, we could construct perfect estimates of both the distance-to-go and cost-to-go beneath an arbitrary node n. The quantities $\bar{\epsilon_{d_n}}$ and $\bar{\epsilon_{h_n}}$ are the mean one-step errors along an optimal path through n to a goal in the distance-to-go and cost-to-go heuristics respectively. During a search, these values are unknown, although they are bounded. The average error can never be less than 0, and can never be larger the largest arc-cost in the case of $\bar{\epsilon_{h_n}}$ or 1 in the case of $\bar{\epsilon_{d_n}}$. The heart of our proposed method for learning during search is to estimate $\bar{\epsilon}_{h_n}$ and $\bar{\epsilon}_{d_n}$ using the observed errors described in Equations 9 and 2. We then use these estimated values to improve the performance of the cost-to-go and distance-to-go heuristics during the same search. To complete the approach, we now discuss two techniques for estimating $\bar{\epsilon_{d_n}}$ and $\bar{\epsilon_{h_n}}$ online.

2.1.1 GLOBAL ERROR MODEL

The *Global Error Model* assumes that the distribution of one-step errors across the entire search space is uniform and can be estimated by a global average of all observed single-step errors. We need only keep a running global sum of observed error in h and d as well as a running count of the number of observations taken. This is roughly equal to the number of expanded nodes, although some nodes may have no children and thus generate no observations. The one difficulty in employing the global error model is that we must estimate which child of node p is $bc(p)$. We assume it is the node with minimum $f(n) = g(n) + h(n)$ among all of p's children, breaking ties on $f(n)$ in favor of low $d(n)$. Pilot experiments showed this to be just as effective as using $\widehat{f}(n) = g(n) + \widehat{h}(n)$, where $\widehat{h}$ is the current corrected heuristic. We then calculate the corrected heuristics $\widehat{d}$ and $\widehat{h}$ using Equations 13 and 16 respectively:

$$\begin{aligned} \widehat{d}^{global}(n) &= \frac{d(n)}{1 - \bar{\epsilon}_d^{global}} & (17) \\ \widehat{h}^{global}(n) &= h(n) + \widehat{d}^{global}(n) \cdot \bar{\epsilon}_h^{global} & (18) \end{aligned}$$

This approach has the benefit of gaining information on average single-step error very quickly and the drawback of the values constantly fluctuating. Our estimates of single-step error change every time we receive an observation, which is at nearly every expansion. If we really want to expand nodes in the order dictated by the cost function, this would require resorting our open list after every expansion. In most benchmark search domains, heuristic computation and node expansion are cheap enough that the cost of the search would be dominated by the cost of constantly resorting the open list. In preliminary experiments, we investigated several approaches, including constantly resorting, a logarithmic resorting schedule, and no resorting. We found that no resorting performed the best empirically and those are the results presented below.

2.1.2 PATH-BASED ERROR MODEL

The *Path-based Error Model* calculates the mean one-step errors, $\bar{\epsilon}_d^{path}$ and $\bar{\epsilon}_h^{path}$, separately along each search path. This allows the model to capture variations in the heuristics' accuracy in different parts of the search space. This is done by passing the cumulative single-step error experienced by a parent node down to all of its children. We can then use the depth of the node to determine the average single-step error along this path. $\widehat{d}^{path}$ and $\widehat{h}^{path}$ and computed analogously to Equations 17 and 18:

$$\widehat{d}^{path}(n) = \frac{d(n)}{1 - \bar{\epsilon}_d^{path}} \tag{19}$$

$$\widehat{h}^{path}(n) = h(n) + \widehat{d}^{path}(n) \cdot \bar{\epsilon}_h^{path} \tag{20}$$

The path-based model has a distinct conceptual (and practical) advantage over the global error model: we need not estimate which node is the best child at the time that a parent node is expanded in order to compute average error. In the path-based model, we can simply say that every child of a node is the best child, as this is what the search has determined at the time of expansion. For when a node is expanded by best first search, the search (and evaluation function) have decided that this particular node, among all other nodes available for consideration, is best. If a node is best among all nodes, it must also be best among its siblings (or its siblings descendants, from which the siblings would derive their values). The practical effect of this is that we need not worry about resorting the open list, because the heuristic corrections of nodes in the path-based model never change.

In either model, if our estimate of $\bar{\epsilon}_d$ is ever as large as one, we assume we have infinite distance and cost-to-go. Because these are estimates, and not bounds, we don't discard nodes which we guess have infinite cost. This preserves the completeness of algorithms using the corrected heuristics. An alternate approach, that we do not pursue in this paper, would be to put these nodes in a reserve list that is only considered when nodes with finite estimated cost have been exhausted. The alternate list could then be sorted on another criteria, for example, the base heuristic or $g(n)$.

2.2 Worst and Best Case Scenarios

The single-step correction techniques presented above do have limitations. Figure 1 shows a grid pathfinding problem where single-step corrections perform poorly. The start state is marked with 's', and the goal is marked with 'g'. The grid is 4-connected. The numbers in the cells show the value of $d(n)$, the distance-to-go estimate. In this instance we use the Manhattan distance in a 4-connected grid under the free-space assumption for $d(n)$.

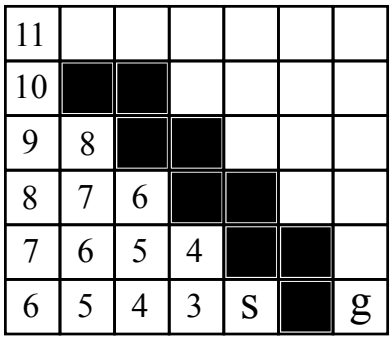


Figure 1: A worst-case domain for single-step corrections

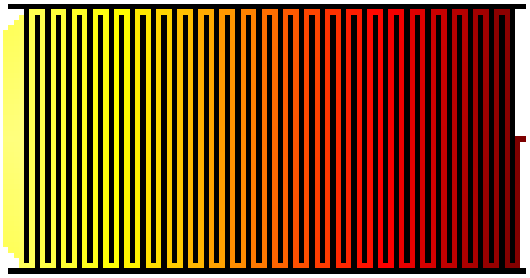
In this example, each move that could take us out of the beginning section into the half of the grid with the goal is a move that will increase the estimated distance-to-go. For any search to escape the beginning of the problem, it must experience a single-step error of two repeatedly. When we reach the state marked with a distance-to-go of eleven, the estimated single-step error will be two for both the global and path-based methods. Until the estimate is lowered below one by expanding many additional nodes with no single-step error, $\widehat{h}(n) = \widehat{d}(n) = \infty$, and our search will expand nodes in uniform cost order due to tie breaking (in the search algorithms presented here, we break ties in favor of low $g(n)$). Thus, if the cost-to-go estimates become infinitely large, we will perform a best-first-search on $g(n)$.

If we had just been doing a greedy search on the base heuristic in this example, we would go straight to the goal from the state marked eleven rather than performing uniform-cost-search. Therefore, greedy search on the corrected heuristic will perform much worse than the uncorrected heuristic. In fact, we can make the example above arbitrarily large, and so the performance gap could be made arbitrarily large as well. Any heuristic with large plateaus or local minima between the start and a goal can demonstrate this behavior. If the plateaus and minima are larger than the areas where the heuristic performs well, we would expect to see this pathology. It should be noted that this is arguably correct, albeit undesirable, behavior. If the heuristic is woefully uninformed, or worse yet misleading, it may be preferable to ignore it entirely and search according to cost incurred.

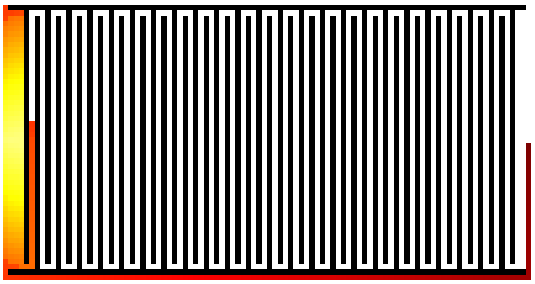
In contrast, the images in Figure 2 provide an example of structured error that works strongly in favor of the single-step correction method. In this ladder-like navigation problem, the error is, as before, highly structured and there are many nodes for which the heuristic is very poorly informed (those in between the 'rungs') and nodes for which the heuristic is perfectly informed (those on the outside of the ladder). Greedy search without correction is much slower than even A* for this problem. However, when learning is added to the solving process, as it is in the bottom panel of the figure, the performance is identical in this case.

This example demonstrates two things. The first is that the corrections can work incredibly well in some domains. The second is that, in order to produce the poor behavior noted in Figure 1, the heuristic must be incorrect early on *for all nodes leading to a reasonable goal*. It is not enough for the heuristic to merely be very incorrect early.

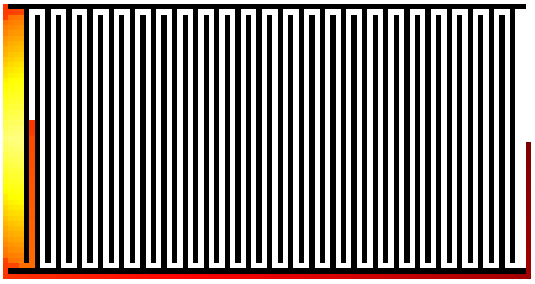
In the eight benchmark domains considered in the evaluation below, we observed neither of the behaviors present in these hand-crafted examples. This suggests that it is often the case that the



Greedy search



A* search



Greedy search with learning

Figure 2: A best-case domain for single-step corrections

heuristic is neither consistently misinformed, nor is it perfectly informed. This is to be expected, as heuristics are generally heavily engineered functions designed to work well in practice.

2.3 Performance of Single-Step Corrections

We will consider two ways of evaluating the quality of our learned heuristics. First, we look at how accurately they predict the true cost-to-go. We then consider their success in guiding heuristic search algorithms towards a goal.

2.3.1 ABSOLUTE ACCURACY

For the accuracy study, we consider three small benchmark domains:

Sliding Tiles Puzzles We examined 100 random 8-puzzle instances. In our implementation, the goal state has the blank in the upper-left, with the numeric tiles laid out in sequence left to right, top to bottom. All actions have unit cost. We do not consider moving back to the parent node's state, so very few duplicate states are encountered during search. Manhattan distance is used to estimate the cost and distance-to-go for all states.

Grid-world Navigation We tested on grid pathfinding problems using the "life" cost function. This cost function produces problems where actions have a large range of costs, short solutions are more costly than longer ones, and the search space includes several large g-value plateaus. These properties have recently seen significant interest (Benton, Talamadupula, Eyerich, Mattmueller, & Kambhampati, 2010; Wilt & Ruml, 2011). We examined 200 by 200 grids with 35% of cells blocked randomly. The cost function means that standard heuristics like Manhattan distance are no longer an accurate (or even admissible) estimate of cost-to-go for these grid problems. To compute a heuristic for these problems, we assume that there are no obstacles and analytically compute the cheapest solution from a node to the goal.

Vacuum World In this domain, which follows the first state space presented by Russell and Norvig (2003), a robot is charged with cleaning up a grid world. Movement is in the cardinal directions, and when the robot is on top of a pile of dirt, it may vacuum. The cost of movement is one plus the number of dirt piles that have already been vacuumed up. Cleaning has unit cost. We used 100 instances that are 200 by 200 with 5 piles of dirt and 35% of cells blocked randomly. An admissible cost-to-go heuristic is found by computing the spanning tree of all dirty cells and the robot. The edges in the spanning tree are then weighted, with the longest edge receiving the current robot weight, the next longest the robot weight plus one, and so on. The length of the solution is estimated inadmissibly by making a free space assumption and computing a greedy traversal of the dirty cells.

In each domain, we examined the following single-step correction techniques:

SS Path The path-based corrections based on single-step error computed as in Equation 20.

SS Global The global corrections based on single-step error computed as in Equation 18.

All algorithms were implemented in Objective Caml, compiled to 64-bit native code, and run on Linux systems with 3.16 GHz Intel Core2 duo processors and 8 GB of RAM. All of the algorithms share the same domain functions and data structures to help ensure fair comparisons.

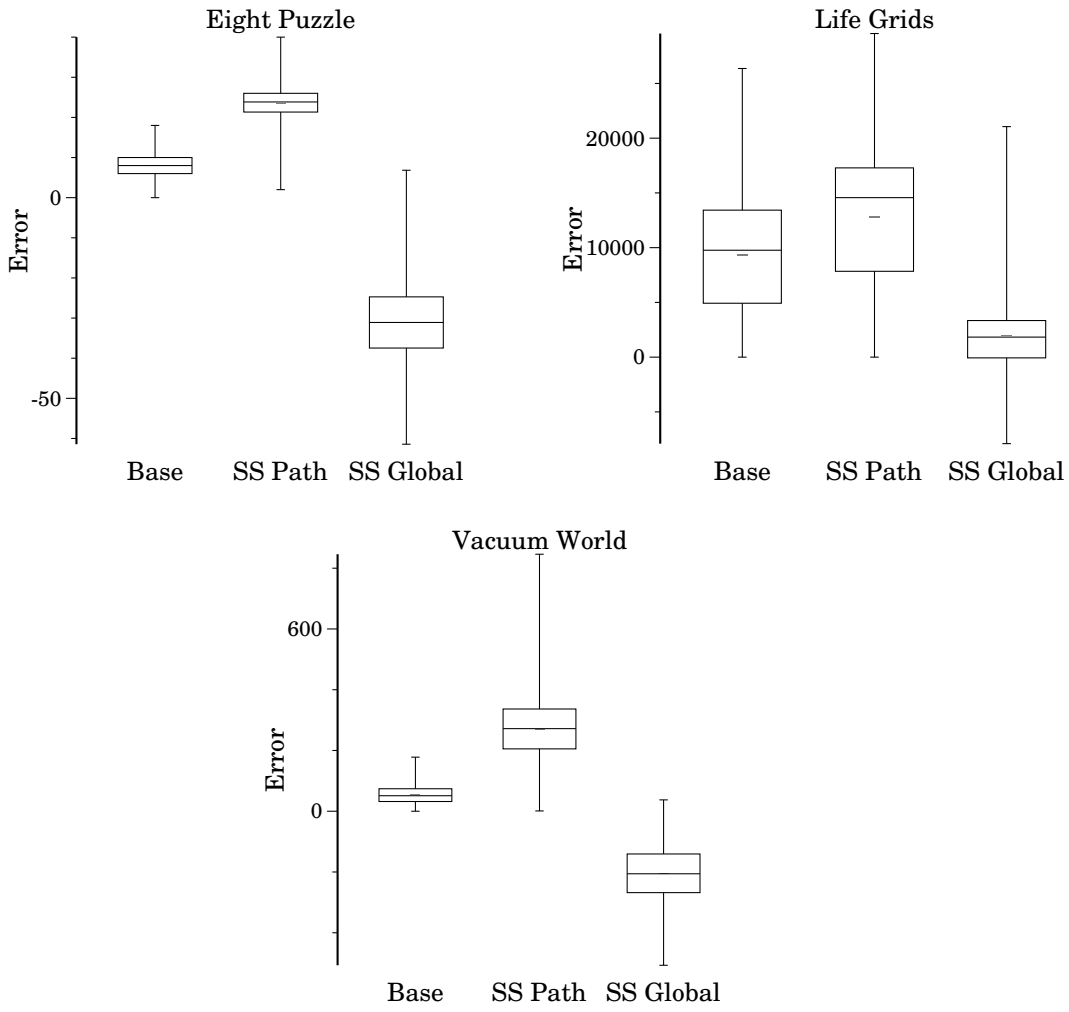


Figure 3: Accuracy of single-step corrections on the 8-puzzle, “life” grids, and vacuum world

Figure 3 shows the performance of the learned heuristics relative to truth on our small benchmark domains. The y-axis represents error, computed as $h^*(n) - \widehat{h}(n)$ where $\widehat{h}$ is the heuristic labeled on the x-axis. We present the data in the form of a box plot. The whiskers extend to the extreme values. The box shows data between the first and third quartile, and the line in the box shows the median value. The gray rectangle shows 95% confidence intervals about the mean. The intervals are so tight for most of the plots this rectangle will appear as a short line. Occasionally this line overlaps with the median, and can not be seen.

In all three plots, we see that the baseline, the admissible heuristic has all of its error above zero because it is required to underestimate the true cost-to-go. It is also relatively accurate when compared to the two learned heuristics. The extreme values for the admissible heuristic are always smaller than that of the learned heuristics. Further, the total range of values is also always smaller than that for the learned heuristics. In the 8-puzzle and vacuum world, the base heuristic is the most accurate, it has a mean error closer to zero than any of the other heuristics being considered.

In all three domains the path based correction has worse performance, in terms of error, than the base heuristic it is attempting to correct as it has median and mean values further away from 0 error and more extreme error values.

Given the performance of these heuristics relative to truth, we might expect a search algorithm guided by global corrections to perform best in life grids, while the base heuristic would perform best in the 8-puzzle and in vacuum worlds.

2.3.2 GUIDANCE

We now turn from the absolute accuracy and evaluate the performance of these heuristics inside of search algorithms. While absolute accuracy may give us some indications as to how a heuristic will perform inside of a search algorithm, it doesn't tell the whole story, and this is one of the most common misconceptions in heuristic search (Holte, 2010). We will see that, surprisingly, path-based corrections provide superior guidance despite being less accurate in absolute terms. We delay our evaluation of heuristics in bounded suboptimal search until Section 2.6 so that we can evaluate the guidance of the heuristics alone before examining their interaction with the admissible heuristics which are needed to provide guarantees on solution quality.

Greedy search (Doran & Michie, 1966) is a best-first heuristic search where best is determined solely by the heuristic. While this estimate may be admissible, greedy search can provide no guarantees on the quality of the solutions it returns, so there is no need to limit the heuristic by restricting it to be admissible. For the guidance study, we use four additional benchmark domains. They were omitted from the accuracy study because we cannot measure accuracy for all states as the search spaces are too large to be enumerated on our machines.

Fifteen Puzzle We examined the 100 instances of the 15-puzzle presented by Korf (1985). We use the Manhattan distance heuristic for both $h(n)$ and $d(n)$, just as we did in the 8-puzzle.

Dynamic Robot Following Likhachev, Gordon, and Thrun (2003), the goal is to find the fastest path from the initial state of the robot to some goal location and heading, taking momentum into account. We use worlds that are 200 by 200 cells in size. We scatter 25 lines, up to 70 cells in length, with random orientations across the domain and present results averaged over 100 instances. We precompute the shortest path from the goal to all states, ignoring dynamics. To compute h, we take the length of the shortest path from a node to a goal and divide it by the maximum velocity of the robot. For d, we use the number of actions along that path.

	8-Puzzle		Life Grids		Small Vacuums	
	nodes generated	cost	$\frac{sec}{1000}$	$\frac{cost}{1000}$	secs	cost
Baseline	582	*128*	*169*	2993	*0.990*	*2673*
SS Global	*763*	43	74	*3050*	0.405	2457
SS Path	**463**	**33**	**71**	**2795**	**0.260**	**2100**

Table 1: Performance of single-step corrections in greedy search on domains from accuracy study

Dock Robot We implemented a dock robot domain inspired by the running example of Ghallab, Nau, and Traverso (2004) and the depots domain from the International Planning Competition. Here, a robot must move containers to their desired locations. Containers are stacked at a location using a crane, and only the topmost container on a pile may be accessed at any time. The robot may drive between locations and load or unload itself using the crane at the location. We tested on 150 randomly configured problems having three locations laid out on a unit square and ten containers with random start and goal configurations. Driving between the depots has a cost of the distance between them, loading and unloading the robot costs 0.1, and the cost of using the crane was 0.05 times the height of the stack of containers at the depot. h was computed as the cost of driving between all depots with containers that did not belong to them in the goal configuration plus the cost of moving the deepest out of place container in the stack to the robot. d was computed similarly, but 1 is used rather than the actual costs.

Vacuums This differs from the accuracy study in that now there are 10 piles of dirt to remove instead of 5. The size of the state space is exponential in the number of dirt piles, so these problems are considerably more difficult than the previous ones. h and d are computed as before.

Table 1 presents the results of using the learned heuristics within a greedy best-first search for the domains used in the accuracy study. Algorithms are run until a solution is found, memory is exhausted, or 10 minutes have passed. We report the mean CPU time required to find a solution, except for the 8-puzzle where we report nodes generated because the times are extremely small, and the mean cost of that solution. The worst entry in a column is *italicized*, and the best value in each column is **bolded**. The table reveals that the more accurate predictors do not always lead to improved performance within a search algorithm. If they did, the global corrections, which were often more accurate than the path-based single-step approach, would have better performance. We see that, despite its relatively poor accuracy, path-based corrections provide the best performance in terms of both solving time and solution cost in a greedy search on these three small benchmarks. Further, global correction, which was more accurate than the base heuristic in two domains, often provides worse performance in terms of solving time.

We show results on the more difficult problems in Table 2. These problems are difficult enough that not all heuristics can guide greedy search to a solution using the machines we had at our disposal. When an algorithm fails to find a solution within system memory or within 10 minutes, we say that it failed. So, for more difficult instances, the cost column either reports the mean solution cost or the number of instances the algorithm failed to solve. Seconds is mean elapsed time for all instances, regardless of why the algorithm halted (i.e. timeouts score 600 seconds, memory exhaus-

	Fifteen Puzzle		Dynamic Robot		Dock Robot		Large Vacuums	
	$\frac{secs}{1000}$	cost	$\frac{secs}{1000}$	cost	secs	cost	secs	cost
Baseline	29	*302*	60	522	*169*	*Failed 55*	*9.07*	*9635*
SS Global	*177*	136	*563*	*1321*	77.2	Failed 24	3.56	6808
SS Path	**15**	**90**	**14**	**47**	**0.38**	**29**	**1.22**	**6063**

Table 2: Performance of single-step corrections when used in greedy search on larger problems

15-puzzle

Heuristic	$\frac{secs}{1000}$	cost
Manhattan Distance	29	302
7-8 PDB	*44*	85
Manhattan Distance SS Path	15	90
7-8 PDB SS Path	**13**	**65**

Table 3: Performance of learned heuristics compared to that of pattern databases

tion as long as it takes to exhaust memory, and so on). We see that path-based single-step corrections provide the best guidance, holds for these larger and more diverse benchmarks. The dockyard robot domain is particularly interesting. Here, the single-step path corrections solve more instances than the other approaches. By observing the performance of the heuristic on a single instance we can solve problems that we could not solve with the base heuristic alone.

2.4 Impact of Base Heuristic Accuracy

One might wonder if these observed improvements are limited to relatively weak heuristics like Manhattan Distance. In Table 3 we compare our best learning method with a modern pattern database for the 15-puzzle, the 7-8 PDB (Korf & Felner, 2002). The 7-8 PDB is the sum of PDB heuristics that have been computed such that they can be added together without becoming inadmissible. Rather than computing the distance of every tile from its goal location, a PDB heuristic works by enumerating the state space for a relaxed version of the problem, in this case one where all of the tiles other than 1 through 7 have no symbol on them. The space is enumerated using all of the actions available in the real problem, and the cost of reaching a state from the goal is recorded. During search, we then abstract the state we are examining into the pattern used in the pattern database, that is we imagine all of the tiles other than 1-7 have been wiped clean, and then ask the PDB how expensive our current configuration is. In the case of the 7-8 PDB, this abstraction and lookup is done twice, and then the values are summed up to provide an estimate of cost-to-go.

We see that using the pattern database heuristic has mixed results with respect to greedy search performance, solving times are longer (although fewer nodes are generated), and solution cost is reduced. However, our path-based heuristic finds solutions faster than the PDB heuristic and those solutions are not much worse on average, and on some instances our heuristic can find better solutions. This is accomplished without the benefit of the pre-computation needed to construct the pattern databases. If we add our path-based correction to the PDB heuristic (the last line of Table 3), it further improves performance, finding better solutions faster than ether the PDB alone

Learning	Vacuums $\frac{nodes}{1000}$	Vacuums $\frac{cost}{1000}$	Life Grids $\frac{nodes}{1000}$	Life Grids $\frac{cost}{1000}$
Base	206	10	115	2993
SS Global	62	**7**	42	3049
Same Instance	**48**	**7**	**36**	2992
Random Instance	64	**7**	**36**	**2983**

Table 4: The learning is instance specific

or path-based corrections on top of Manhattan distance. From this we conclude that single-step correction can improve the performance of even strong heuristics.

2.5 Instance Specific Heuristics

One advantage of online corrections is that they do not require the use of a set of training instances. This means we can avoid the problem of ensuring that our training instances are similar enough to our test instances for the learning to generalize. Since all of our learning is being performed online during the solving of a single instance, we needn't worry about generalization. However, we might wonder if the information being learned during the search is specific to one instance, or if it can be used to seed the estimated error values for searches on other instances in the same domain.

Table 4 shows the performance of our global single-step model used in greedy search in two new ways. The first row of the table shows the performance of the base heuristic and the second row of the table shows the performance of the global single-step model learned on line. The third line, "Same Instance" shows the performance of the global single-step model values for error learned by the global model on the same instance of the problem being solved but now being used statically (with learning turned off). "Random Instance" is similar, but the learned values come from a random instance. We use the global model because it is clear how to transfer the information learned from one instance to another: we simply take the final values we computed for $\bar{\epsilon}_h$ and $\bar{\epsilon}_d$ and use those as the average error in a new problem. In this table, we present results in terms of nodes generated in order to focus on search guidance and ignore the overhead of learning (Table 2 already demonstrated that using online learning can improve the speed of search algorithms). We present two domains, the vacuum domain, where learned heuristic errors are very different between instances, and life grids, where learned error is similar between instances.

As we saw before in Table 2, the online corrections produce better results than the base heuristic. Additionally, for both domains, using the errors learned previously for the same instance improves performance substantially. This shows us that the improved performance is not because of some fortuitous synergy between learning and search. If it were, the online model would out-perform the same errors fed into a static model. As it does not, we conclude that we are learning a meaningful ordering over the nodes.

We see that the heuristic learned from the same instance performs better than one from a random instance in the vacuum domain. This indicates that the technique is learning an instance-specific model online, and that instance-specific information is beneficial to our searches. Interestingly this is not the case for the vacuum problem. Recall that for our life grid instances, the start and goal state are always in the same location, and the obstacles are placed down uniformly at random. This

OptimisticSearch$(root, b, w)$
1. $incumbent \leftarrow$ null
2. $open \leftarrow \{root\}$
3. while($incumbent =$ null and $open \neq \{\}$)
4. remove n from $open$ with minimum $f'(n) = g(n) + w \cdot h(n)$
5. if n is a goal
6. $incumbent \leftarrow n$
7. otherwise, expand n and insert children into $open$
8. while($open \neq \{\}$)
9. $f_{min} \leftarrow n \in open$ with minimum $f(n) = g(n) + h(n)$
10. $f'_{min} \leftarrow n \in open$ with minimum $f'(n) = g(n) + w \cdot h(n)$
11. if $b \cdot f(f_{min}) \geq g(incumbent)$
12. return $incumbent$
13. otherwise, if $f'(f'_{min}) \leq g(incumbent)$
14. if f'_{min} is a goal
15. $incumbent \leftarrow min(f'_{min}, incumbent)$
16. otherwise, remove f'_{min} from $open$, expand it, and insert its children.
17. otherwise, remove f_{min} from $open$, expand it and insert children into $open$
18. return $incumbent$

Figure 4: Optimistic Search pseudo-code with escape hatch

suggests that the error in the heuristic is likely to be similar between any two random instances, and thus the learning should generalize well from one instance to another.

2.6 Bounded Suboptimality: Skeptical Search

So far we have seen how on-line learning can improve greedy best-first search, we now turn to the setting of bounded suboptimal search. Here, we require solutions whose quality is within a fixed factor of optimal.

Bounded suboptimal search algorithms like weighted A* (Pohl, 1973) rely on the admissibility of their base heuristic to obtain their suboptimality bound. However, some algorithms such as optimistic search (Thayer & Ruml, 2008) can use arbitrary heuristics for at least a portion of their search. Optimistic search works by running weighted A* with a weight higher than the desired suboptimality bound. This can be hand tuned per problem or per domain, although we found that a weight twice as large as the desired bound worked well in the domains they evaluated the algorithm in. However, looking closely at the algorithm will reveal that optimistic search can take advantage of any inadmissible heuristic. After finding an incumbent, additional nodes are expanded in A* order until we can prove the solution found was within the desired suboptimality bound.

Optimistic search proves that the incumbent is within the bound by comparing its cost to $f(f_{min})$, the estimated cost of the node with the smallest f value. The f value of a node acts as a lower bound on the cost of a solution through that node, so the f value of the node with the smallest f value acts as a lower bound on the cost of an optimal solution to a problem. Therefor,

SkepticalSearch$(root, w)$
1. $incumbent \leftarrow$ null
2. $open \leftarrow \{root\}$
3. while($incumbent =$ null and $open \neq \{\}$)
4. remove n from $open$ with minimum $\widehat{f}'(n) = g(n) + w \cdot \widehat{h}(n)$
5. if n is a goal
6. $incumbent \leftarrow n$
7. otherwise, expand n and insert children into $open$
8. while($open \neq \{\}$)
9. $f_{min} \leftarrow n \in open$ with minimum $f(n) = g(n) + h(n)$
10. $\widehat{f}'_{min} \leftarrow n \in open$ with minimum $\widehat{f}'(n) = g(n) + w \cdot \widehat{h}(n)$
11. if $w \cdot f(f_{min}) \geq g(incumbent)$
12. return $incumbent$
13. otherwise, if $\widehat{f}'(\widehat{f}'_{min}) \leq g(incumbent)$
14. if $\widehat{f}'_{min}$ is a goal
15. $incumbent \leftarrow min(\widehat{f}'_{min}, incumbent)$
16. otherwise, remove $\widehat{f}'_{min}$ from $open$, expand it, and insert its children.
17. otherwise, remove f_{min} from $open$, expand it and insert children into $open$
18. return $incumbent$

Figure 5: Skeptical Search pseudo-code

if $f(f_{min})$ is within a factor b of the cost of the incumbent solution, we know that the incumbents quality is within a bounded factor of the cost of an optimal solution.

Pseudo code for the algorithm is provided in Figure 4. In lines 3 through 7, weighted A* using a weight w (presumably higher than the bound b) is used to find an initial solution. The remainder of the code is focused on proving that the incumbent is within the desired suboptimality bound (lines 11, 12, and 17) or opportunistically improving the quality of the incumbent solution. In lines 11 and 12, we test to see if the incumbent solution can be shown to be within the current bound, and if it is, then we return it. In line 17, we remove f_{min} from open and expand it. This may raise the lower bound on the cost of an optimal solution to the problem, allowing us to return the current incumbent in the next iteration. Lines 13–16 seek to improve the current incumbent solution. If it ever appears that a node might lead to a better incumbent solution, it is pursued. In practice, this case is rarely, if ever, used. For a node to be expanded by this case, it must first be generated by an f_{min} expansion, otherwise it would have been expanded before an incumbent was found in lines 1–7. This can happen if w and b are selected such that the solution initially found is outside of the bound. In practice, we prove the quality of a solution long before such a node becomes a candidate for expansion in line 13.

⟨FIX: DESCRIBE THE DIFFERENCES BETWEEN SKEPTICAL AND OPTIMISTIC IN MORE DETAIL THAN YOU DO CURRENTLY. BELABOR THE POINT THAT YOU'RE GETTING A PRINCIPLED INADMISSIBLE ESTIMATE OF COST-TO-GO, RATHER THAN JUST SKEWING H UPWARD TO DEAL WITH ADMISSIBILITY.⟩

When searching for an incumbent solution, optimistic search can use any inadmissible heuristic and still retain its guarantees of bounded suboptimality as long as an admissible heuristic is available for proving that the incumbent was within the desired bound. While, at first glance, it may not be obvious that optimistic search is using an inadmissible heuristic, we can show that it is by closely examining line 4. Rather than writing $f'(n) = g(n) + w \cdot h(n)$, we could instead write $f'(n) = g(n) + b \cdot \frac{w}{b} \cdot h(n)$. We can think of $\frac{w}{b} \cdot h(n)$ as an inadmissible heuristic which attempts to correct for the under-estimating nature of $h(n)$ by scaling it up uniformly (recall that $w > b$). We can replace the weighted admissible heuristic from the first phase of optimistic search with any learned heuristic. We call this modification of optimistic search *skeptical search*, and we provide pseudo-code for it in Figure 5. It is skeptical in that it does not place absolute trust in the base heuristic, but rather learns to temper its predictions based on experience. Note that the adhoc additional weight parameter of optimistic search has been removed, and so skeptical only accepts two parameters instead of three. As we will see in the following evaluation, skeptical search offers two benefits over optimistic search. It removes the need for parameter tuning and provides improved performance in several benchmark domains.

The pseudo-code makes no attempt to specially handle duplicate states, that is states re-encountered by a cheaper path. Avoiding re-expanding duplicate states often improves the performance of weighted A* (Likhachev, Gordon, & Thrun, 2003; Thayer, Ruml, & Kreis, 2009). If the heuristic being used is consistent, dropping duplicates has no impact on the suboptimality bound. (If the heuristic is inconsistent, dropping duplicates forces us to loosen the suboptimality bound dramatically, see Ebendt and Drechsler (2009) for details.) In skeptical search, we cannot drop duplicates entirely. They must be retained so that $f(f_{min})$ provides an accurate lower bound on optimal solution cost. At best, we can choose to delay duplicates during the first iteration of skeptical search, when we are looking for a potential solutions. This leads us to find potential solutions faster, but they tend to be of lower quality. This makes the step of proving solution quality take longer. Preliminary experiments showed that delaying duplicate expansions until the cleanup phase provided better performance, and this is the approach taken in the results reported here.

Figures 6, 7, 8, and 9 compare several optimism settings, the factor by which w exceeds b, for the original optimistic search (Thayer & Ruml, 2008), weighted A* and skeptical search. The x-axis of the plot is the suboptimality bound, the desired guarantee on solution quality. The y-axis represents the amount of time needed to solve problems for the given bound. We show results for skeptical with path-based correction as it produced the best results.

Although many of the algorithms are often difficult to distinguish in detail, what is clear is that skeptical search is always at least competitive with optimistic search for any of the optimism settings examined. On the fifteen puzzle (Figure 6), and dynamic robot navigation (Figure 8) because the confidence intervals on the search time between skeptical search and the best configuration for optimistic search overlap. For life cost grids (Figure 7), we see that skeptical search takes between half and a third of the time needed by any optimistic search and and it is three times faster in vacuum world (Figure 9).

In addition to out-performing optimistic search, skeptical search removes the need for parameter tuning. Optimistic search requires two parameters, the desired suboptimality bound and an optimism factor. The optimism factor tells optimistic search how aggressive it should be in pursuing the initial solution. If it is set too high, the incumbent solution will be outside of the desired bound, and the performance of the algorithm will suffer. If it is set too low, finding the initial solution will take too long, pulling down overall algorithm performance. Skeptical search has only

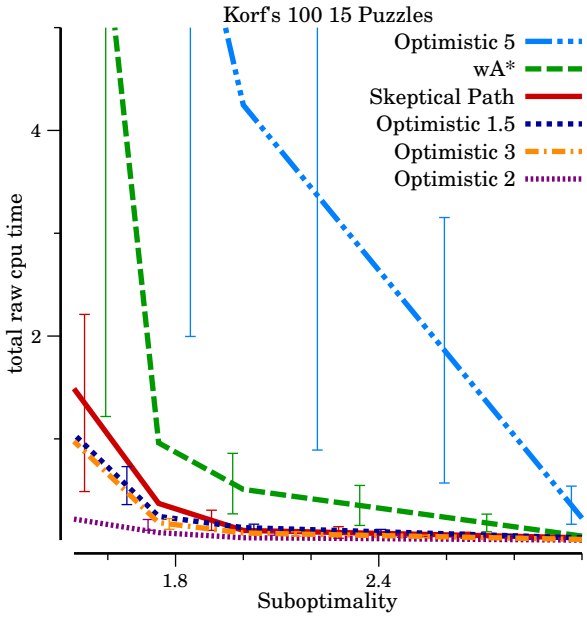


Figure 6: Performance of bounded suboptimal search on the 15-puzzle

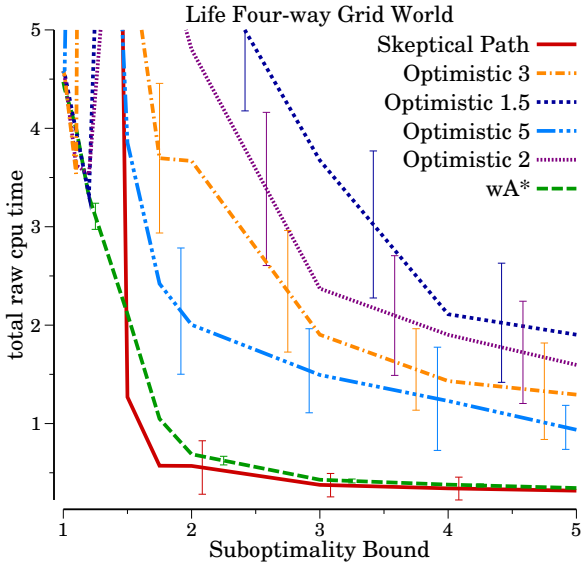


Figure 7: Performance of bounded suboptimal search on life grid navigation

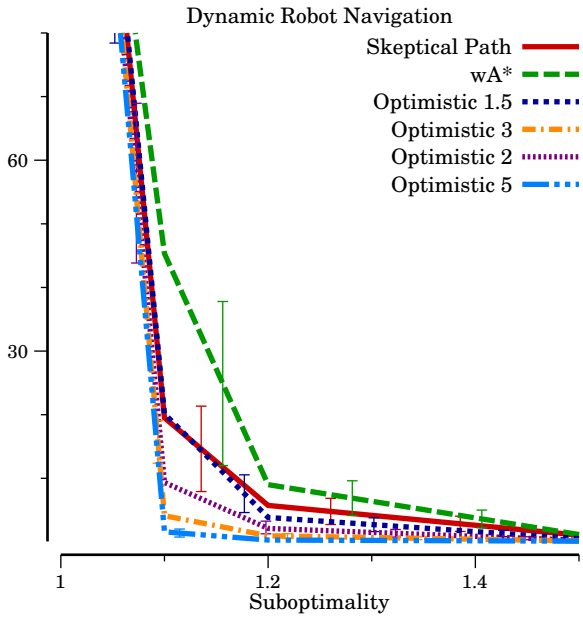


Figure 8: Performance of bounded suboptimal search on dynamic robot navigation

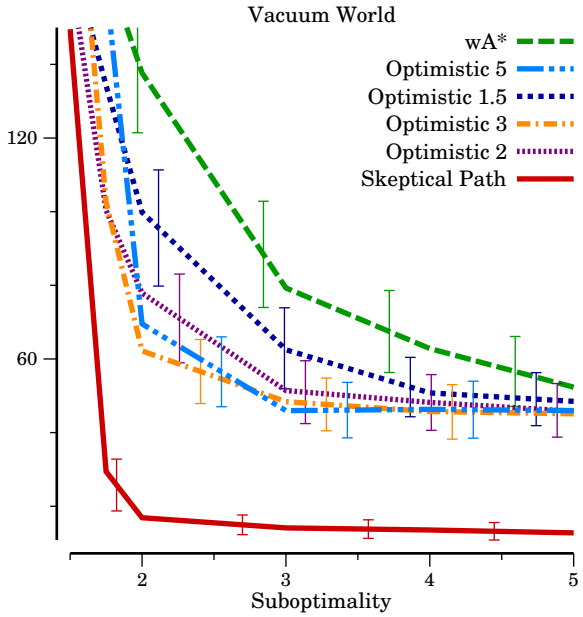


Figure 9: Performance of bounded suboptimal search on the vacuum problems

	8-Puzzle		Life Grids		Small Vacuums	
	nodes generated	cost	$\frac{sec}{1000}$	$\frac{cost}{1000}$	secs	cost
Greedy Baseline	582	*128*	*169*	2993	*0.990*	*2673*
Greedy SS Path	**463**	**33**	**71**	**2795**	**0.260**	**2100**
A* SS Path						

Table 5: Performance of single-step corrections in greedy search on domains from accuracy study

	Fifteen Puzzle		Dynamic Robot		Dock Robot		Large Vacuums	
	$\frac{secs}{1000}$	cost	$\frac{secs}{1000}$	cost	secs	cost	secs	cost
Greedy Baseline	29	*302*	60	522	*169*	*Failed 55*	*9.07*	*9635*
Greedy SS Path	**15**	90	**14**	**47**	**0.38**	**29**	**1.22**	6063
A* SS Path	810	**63**	813	Failed #	33.5	Failed #	8.51	**5011**

Table 6: Performance of single-step corrections when used in greedy search on larger problems

the desired suboptimality bound as a parameter. Rather than requiring an explicit optimism factor, skeptical search constructs $\widehat{h}$ using its experience during problem solving. It's best suited to domains where expanding nodes and computing heuristics is relatively inexpensive. If computing heuristics and generating successors are very expensive, more complicated techniques like explicit estimation search (Thayer & Ruml, 2010) are more appropriate. Of the domains presented here, explicit estimation search only outperforms skeptical search in vacuum world.

2.7 Performance of Heuristics in A*

2.8 Summary

As we have just seen, our approach to learning heuristic corrections online during the solving of a single instance produces heuristics with strong guidance but poor overall accuracy. We saw that the strong guidance led to good performance in both suboptimal and bounded suboptimal search, improving substantially on the performance of the base heuristics. Tests of transfer provided evidence that on-line learning learns something specific to the instance being solved. This may be particularly useful when the instances of interest have substantially different properties despite being from the same domain.

Finally, we should note that we make no assumptions about the characteristics of the heuristics used as the basis for learning, This allows our technique to be as general as possible. The equations showing that the learning of single-step corrections is theoretically sound rely on only two assumptions about the basic nature of the underlying heuristics: $h(n)$ estimates the cost-to-go from n to a goal, and $d(n)$ estimates the number of actions in that solution. We did not make, nor do we need make, any assumption as to the consistency, admissibility, or accuracy of the underlying heuristic.

3. Alternate Approaches to Learning During Search

The single-step corrections presented in the previous section are not the only way that we can learn improved heuristics on-line. This section of the paper focuses on alternative approaches that can be

used on-line and while similarly justified and natural, do not appear to work as well in practice, as we will see in the accompanying evaluations.

3.1 Single-step Correction Without Distance Estimates

We might naturally wonder how much the distance-to-go heuristic $d(n)$ is contributing to the singles-step correction process. To evaluate this we altered the single-step error model to use only cost-to-go estimates, removing the need for distance-to-go estimates entirely. Rather than measuring the error in $h(n)$ per-step, we measure it per-cost:

$$\epsilon_{h_p}^{cost} = \frac{(h(bc(p)) + c(p, bc(p))) - h(p)}{c(p, bc(p))} \tag{21}$$

This can also be rewritten using Equation 2:

$$\epsilon_{h_p}^{cost} = \frac{\epsilon_{h_p}}{c(p, bc(p))} \tag{22}$$

Then, we compute the mean cost-step error at p as:

$$\bar{\epsilon}_{h_p}^{cost} = \frac{\sum_{n \in p \rightsquigarrow goal} \epsilon_{h_n}^{cost}}{h^*(p)} \tag{23}$$

We then compute the corrected heuristic as:

$$\widehat{h}^{cost}(n) = \frac{h(n)}{1 - \bar{\epsilon}_{h_n}^{cost}} \tag{24}$$

using, as we did in Equations 18 and 20, either a path-based or global average to estimate $\bar{\epsilon}_{h_p}^{cost}$. The following proof shows that this is a legitimate correction:

Theorem 3 *For any node p with a goal beneath it:*

$$h^*(p) \quad = \quad h(p) + h^*(p) \cdot \bar{\epsilon}_{h_p}^{cost} \tag{25}$$

where $\bar{\epsilon}_{h_p}^{cost}$ is the average per-cost error in the cost-to-go estimate $h(p)$.

Proof: The proof is by induction over the nodes in $p \rightsquigarrow goal$, the optimal path from p to a goal node. For our base case, we show that when $bc(p)$ is a goal, the theorem holds:

$$\begin{array}{rcll}
h^*(p) & = & c(p, bc(p)) & \text{because } bc(p) \text{ is the goal} \\
& = & h(p) + c(p, bc(p)) - h(p) & \text{by algebra} \\
& = & h(p) + c(p, bc(p)) \cdot \frac{c(p,bc(p))) - h(p)}{c(p,bc(p))} & \text{by algebra} \\
& = & h(p) + c(p, bc(p)) \cdot \frac{(h(bc(p)) + c(p,bc(p))) - h(p)}{c(p,bc(p))} & \text{h(bc(p)) = 0} \\
& = & h(p) + c(p, bc(p)) \cdot \bar{\epsilon}_{h_p}^{cost} & \text{by Equation 21} \\
& = & h(p) + h^*(p) \cdot \bar{\epsilon}_{h_p}^{cost} & \text{because } bc(p) \text{ is the goal}
\end{array}$$

For the inductive case we show that, assuming that Equation 25 holds for $bc(p)$, we can show that it holds for its parent p as well:

$$\begin{aligned}
h^*(p) &= c(p, bc(p)) + h^*(bc(p)) && \text{by Equation 1} \\
&= c(p, bc(p)) + h(bc(p)) + h^*(bc(p)) \cdot \bar{\epsilon}^{cost}_{h_{bc(p)}} && \text{by inductive assumption} \\
&= h(p) + \epsilon_{h_p} + h^*(bc(p)) \cdot \bar{\epsilon}^{cost}_{h_{bc(p)}} && \text{by Equation 2} \\
&= h(p) + \epsilon_{h_p} + \textstyle\sum_{n \in bc(p) \rightsquigarrow goal} \epsilon^{cost}_{h_n} && \text{by Equation 23} \\
&= h(p) + \textstyle\sum_{n \in p \rightsquigarrow goal} \epsilon^{cost}_{h_n} && \text{by definition of } \rightsquigarrow \\
&= h(p) + h^*(p) \cdot \frac{\sum_{n \in p \rightsquigarrow goal} \epsilon^{cost}_{h_n}}{h^*(p)} && \text{by algebra} \\
&= h(p) + h^*(p) \cdot \bar{\epsilon}^{cost}_{h_p} && \text{by Equation 23}
\end{aligned}$$

□

Solving Equation 25 for $h^*(p)$, we can arrive at something nearly identical to Equation 24. The only difference is that here we have the exact single-step error, and in Equation 24 single-step error is being estimated.

$$\begin{aligned}
h^*(p) &= h(p) + h^*(p) \cdot \bar{\epsilon}^{cost}_{h} \\
h^*(p) - h^*(p) \cdot \bar{\epsilon}^{cost}_{h} &= h(p) \\
h^*(p) \cdot (1 - \bar{\epsilon}^{cost}_{h}) &= h(p) \\
h^*(p) &= \frac{h(p)}{(1 - \bar{\epsilon}^{cost}_{h})}
\end{aligned}$$

As with the single-step model, there are many ways we could choose to aggregate the observed error in the heuristic. In this work we evaluate two:

Cost Global Computes $\widehat{h}$ based on the cost-based error in $h(n)$, computed as in Equation 24 using a global average to estimate the error in h. The best-child is estimated as in the global model.

Cost Path Computes $\widehat{h}$ based on the cost-based error in $h(n)$, computed as in Equation 24. Error in the cost-to-go heuristic is aggregated along paths as in the previous path-based model.

We now evaluate the cost-step model. This will allow us to see the influence of distance estimates on our single-step corrections.

3.1.1 ACCURACY

Figure 10 shows the absolute accuracy of the cost-step models on "life" grid navigation and small vacuum problems. The 8-puzzle is omitted because it has unit cost, and the cost-step models are identical to the single-step models for such domains. Additionally, the cost-based global model is omitted from Figure 10 as it occasionally estimated the heuristic to be infinitely large. The figure shows that, like the single-step approach to learning, the heuristics constructed online using cost-step error are less accurate than the base heuristic that they are being built from. As we saw in the previously presented distance based corrections, the global model appears to be less accurate in general than the path based corrections.

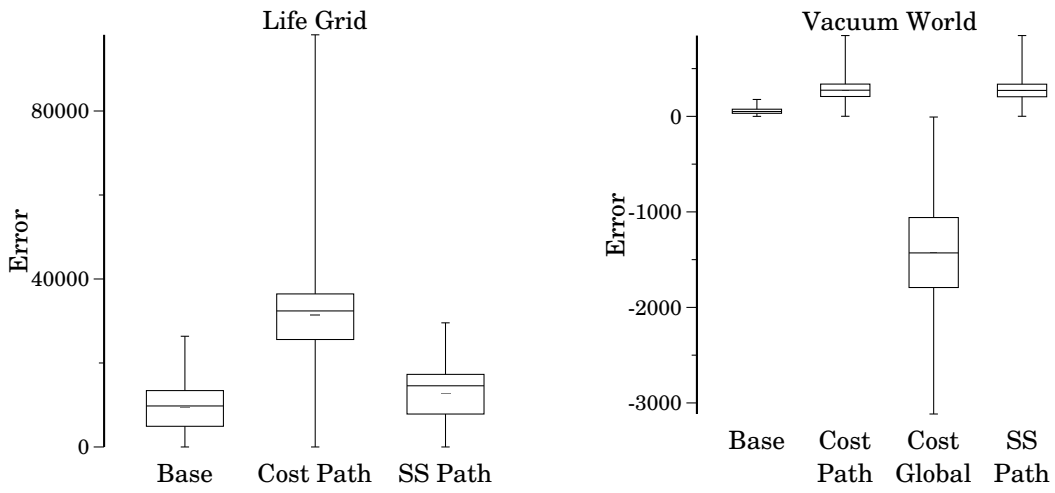


Figure 10: Accuracy of cost-step model on "life" grids (left) and small vacuum problems (right)

	Life Grids		Small Vacuums	
	$\frac{sec}{1000}$	$\frac{cost}{1000}$	secs	cost
Baseline	169	2993	0.990	2673
Cost-Global	*5140*	*9846*	1.042	*2786*
Cost-Path	2509	3246	*1.725*	1910
SS Path	**71**	**2795**	**0.260**	**2100**

Table 7: Performance in greedy search on domains from accuracy study

	Dynamic Robot		Dock Robot		Vacuums	
	$\frac{secs}{1000}$	cost	secs	cost	secs	cost
Baseline	60	522	169	Failed 55	*9.07*	9635
Cost-Global	*600000*	*Failed 40*	*349*	*Failed 73*	1.75	Failed 1
Cost-Path	**14**	**46**	11.8	Failed 2	1.94	*Failed 22*
SS Path	**14**	47	**0.385**	**29**	**1.22**	**6063**

Table 8: Comparing cost-step corrections to single-step corrections on larger problems

3.1.2 GUIDANCE

Table 7 shows the performance of the cost-step heuristics in a greedy best-first search on the domains used in the accuracy study. While we might expect that, like the single-step models, cost-step heuristics would provide better guidance than the baseline, the experiments reveal that they do not. We see that, for these domains, both cost-based approaches are often worse than the base heuristic in terms of time and solution cost.

Table 8 shows the performance of the cost-step heuristics on larger benchmark problems. We see that although the global version of the cost-step approach is consistently worse than the base heuristic, the path-based approach often makes substantial improvements, solving problems faster and providing solutions of lower cost. The single-step path-based heuristic is still substantially better in that it is never slower and it never failed to solve one of our benchmark instances. From this, we can conclude that using the distance-to-go estimate $d(n)$ is important to the good performance of our corrected heuristics.

3.2 Comparison to Generic Regression Algorithms

The techniques we have considered so far were derived specifically for learning heuristic values online. We also evaluated the use of generic linear regression techniques. These can be applied, noting that if our corrected heuristics were perfect, we would see that the estimated cost of the parent, $\widehat{f}$, was exactly that of the estimated cost of the best child. If we were computing the corrected heuristic as a feature vector $\phi(n)$ weighted by a vector $\vec{w}$, we could expand this equation to be:

$$\begin{aligned}
\widehat{f}(p) &= \widehat{f}(bc(p)) && (26)\\
g(p) + \widehat{h}(p) &= g(bc(p)) + \widehat{h}(bc(p)) && (27)\\
\widehat{h}(p) &= g(bc(p)) - g(p) + \widehat{h}(bc(p)) && (28)\\
\widehat{h}(p) &= \widehat{h}(bc(p)) + c(p, bc(p)) && (29)\\
\widehat{h}(p) - \widehat{h}(bc(p)) &= c(p, bc(p)) && (30)\\
(\vec{\phi(p)} - \vec{\phi(bc(p))}) \times \vec{w} &= c(p, bc(p)) && (31)
\end{aligned}$$

This shows that, so long as we can determine which node is the best child, we can use linear regression to compute an improved estimate of cost-to-go. To do this, we use the difference of a set of features between a parent and its best child and learn a function from them onto the cost of the transition between them. The same function for estimating the cost of the transition from the differences in features will be an estimator of the full cost-to-go from any node, as shown by the above algebra.

Unfortunately, this does not work for all regression algorithms. If the learned function is not a linear combination of the features, then we cannot perform the transformation in between Equation 30 and Equation 31. We can still use regression techniques in these situations, so long as we are willing to assume that the heuristic values of nodes deeper in the search tree are more likely to be accurate than that of nodes higher in the tree. Equation 1 suggests that we can approximate $h^*(p)$ as $h(bc(p)) + c(p, bc(p))$. It may be reasonable to assume that the heuristic of the child has a more accurate heuristic because the best child is one step closer to a goal, and therefor has less to be uncertain about. What this effectively provides us is a target value for standard regression techniques that can be used during the search itself. For all nodes (save the root), we can collect a

set of features of the parent and then train them to estimate the heuristic of the best child plus the cost of arriving at that child, which should be more accurate than the original heuristic.

To provide a fair comparison with our previous techniques, we limit ourselves to the following four features for learning:

$g(n)$ the cost of arriving at n from the root

$h(n)$ an estimate of the cost-to-go from n to a goal along a cost-optimal path from n to the goal

$depth(n)$ the number of actions between the root and n

$d(n)$ an estimate of the number of actions along a cost-optimal path from n to the goal

We take care to try to normalize the features between 0 and 1 based on an estimate of their range (using the h and d values of the root), as this typically improves the performance of learning. We cannot always normalize the values between 0 and 1 because we do not always know what the maximum value for a feature is a priori. For example the maximum $depth(n)$ and $g(n)$ are tied to the execution of search and thus unknowable. We evaluate the following learning techniques:

LMS Least means squared linear regression can be used to train an improved estimator of cost-to-go. In the offline setting, this is typically done with batched regression using a library like LAPACK. However, in our online setting, batched regression is impractically slow. We use streamed regression which will still converge *provided the data are presented in a random order*. Since an online approach will present the data to the learner in an order related to the search order, we are violating one of the assumptions that guarantees our learning will converge. Therefore we can only make observations as to the empirical performance of online regression, not its correctness.

ANN We trained a three layer neural network with three hidden nodes and used it to compute $\widehat{h}$. This learning technique was also used by Jabarri Arfaee, Zilles, and Holte (2010). We used a back-propagation learning rate of 0.01. To initialize the network, we collected the first 100 training pairs and performed a batch regression for 1000 epochs or until the network converged. Doing the batched regression any shorter or longer had a negative impact on performance. After this initial period, we began streaming subsequent features and target values to the learner.

ANN Offline We used the same network architecture and training algorithm as before, but now in the offline setting. We used at least 500,000 feature-target pairs taken from 10 random instances, with the exact number of pairs varying by domain. We used $h^*(n)$ as the target value and used $g^*(n)$, the optimal cost of arriving at a node from the initial state, as features in addition to $d(n)$, $h(n)$, $depth(n)$, and a constant. We trained the network for 10,000 epochs or until it converged.

LMS Offline Using the same data as we did when training the offline ANN, we optimally solved a least mean squared linear regression using $h^*(n)$ as the target value and $g^*(n)$, $d(n)$, $h(n)$, $depth(n)$ and a constant as features.

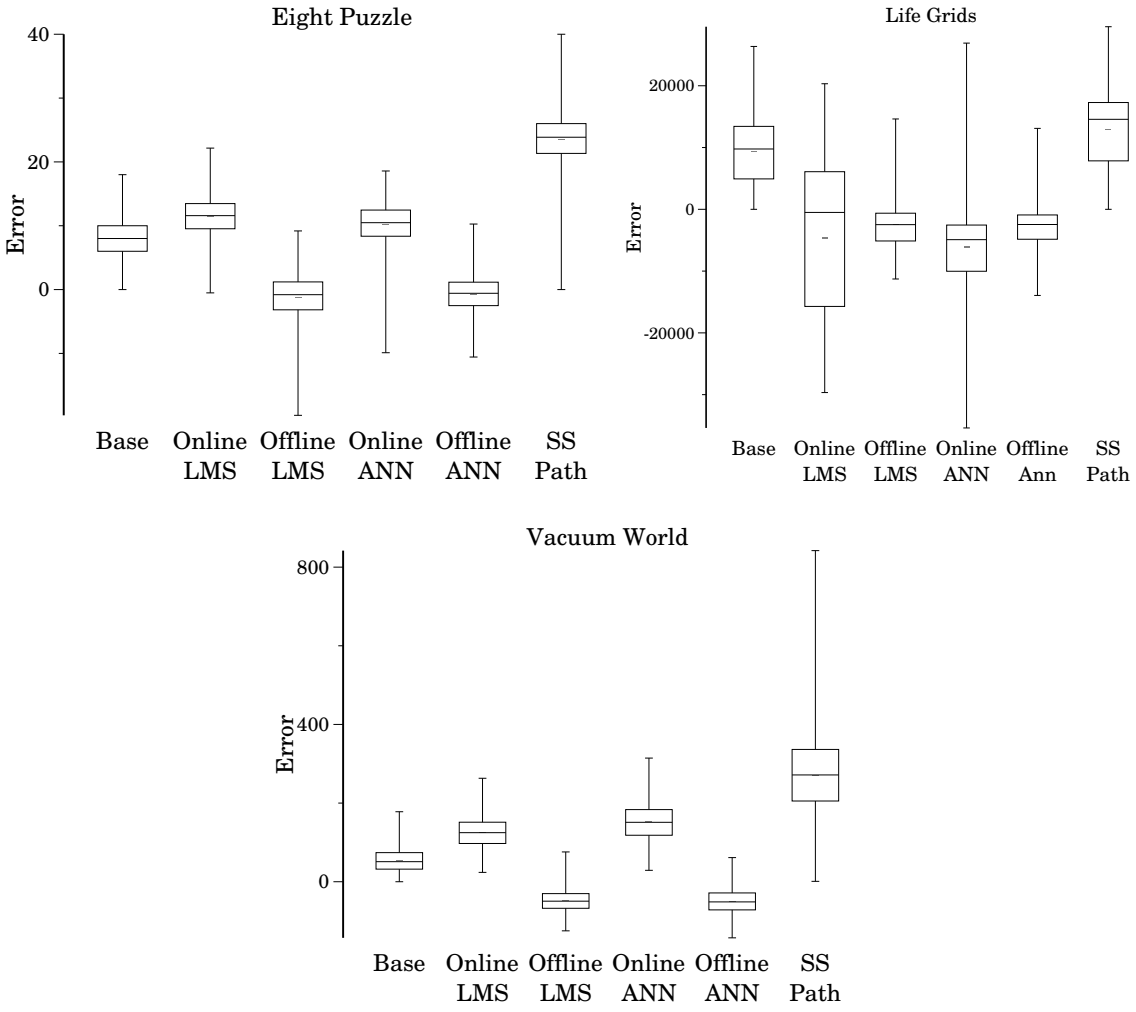


Figure 11: Accuracy of heuristics constructed with standard regression techniques on 8-puzzles, "life" grids, and small vacuum problems

	8-Puzzle		Life Grids		Small Vacuums	
	generated	cost	$\frac{sec}{1000}$	$\frac{cost}{1000}$	secs	cost
Baseline	582	*128*	169	2993	0.990	2673
Offline LMS	**337**	113	275	3967	*6.266*	1573
Online LMS	514	108	216	2993	**0.158**	**1368**
Offline ANN	798	**31**	323	2809	0.390	2459
Online ANN	610	56	919	5056	0.995	*5415*
SS Path	463	33	**71**	**2795**	0.260	2100

Table 9: Performance in greedy search on small domains

3.2.1 ACCURACY

Figure 11 shows the performance of the regression techniques in terms of absolute accuracy on the 8 puzzle, "life" grid navigation, and our small vacuum benchmark. We see, most notably in the 8 puzzle plot, that the offline estimators are more accurate predictors of cost-to-go than the base heuristic or their online counterparts. We also see that online LMS corrections has varied performance. As before, we now need to see how well accuracy translates into search performance.

3.2.2 GUIDANCE

Table 9 shows the performance of the regression-based heuristics in greedy search for the same domains that we used in the accuracy study. We see that the offline ANN tends to outperform its online counterpart. This isn't particularly surprising. The offline learners have better data available as they are learning against true cost-to-go values.

For the 8-puzzle, Offline LMS finds solutions faster than any other approach, while the Offline ANN finds the best solutions but requires more expansions. For permutation puzzles like the 8 and 15-puzzle, the state space for all problems is identical and a heuristic learned on one instance of the problem transfers perfectly to new instances of that problem. These two offline techniques benefit by knowing the "correct" answer at the beginning of search while the online technique must learn the improved heuristic on the fly. That is, the offline techniques have already performed all of their learning and converged on a set of weights to produce $\widehat{h}$. This function will be used on all nodes in search. In contrast, the online techniques are learning their weights, and so $\widehat{h}$ will fluctuate over time leading to potentially unfair comparisons of nodes.

We see that for these small benchmarks, the online LMS correction is competitive with the single-step path corrections. It is nearly as efficient for the 8-Puzzle, and produces better solutions in less time on the small Vacuum World benchmarks. It is interesting to note that online LMS performs best when it is least accurate on these benchmark domains. It is, however, just over three times slower on the life grid benchmarks. We now turn to larger benchmarks.

Table 10 shows the performance of the learned heuristics in greedy best-first search on problems that are too large to enumerate. As these problems are so large, we can not perform offline learning directly. The results show that single step corrections are much more effective that the regression-based techniques. The LMS heuristics now outperform the ANN heuristics which had less variance and a better mean. Why is this? First, recall that the target values for both learners are very different. The offline techniques are allowed to see truth, while the online techniques must approximate the

	Fifteen Puzzle		Dynamic Robot		Dock Robot		Vacuums	
	$\frac{secs}{1000}$	cost	$\frac{secs}{1000}$	cost	secs	cost	secs	cost
Baseline	29	302	60	522	*169*	*Failed 55*	9.07	9635
Online LMS	**8**	520	75	522	73	Failed 17	9.42	9635
Online ANN	*2444*	719	3418	*881*	135	Failed 47	7.40	6155
SS Path	15	**90**	**14**	**47**	**0.385**	**29**	**1.22**	**6063**

Table 10: Comparing online LMS and ANNs to single-step corrections on larger problems

target value for learning using the f values of their children. We posit that the ANN may be more sensitive to noise in the target values and may be more prone to over-fitting. Thus, it may be learning to predict the noise in our prediction of the true cost-to-go instead of predicting h^* as we would desire.

⟨FIX: EXPAND UPON THIS AS REQUESTED BY REVIEWER 1⟩ That linear regression and neural network-based heuristics perform so poorly is especially surprising considering how well these techniques have performed in previous work on learning in heuristic search and their high accuracy in our own evaluation. Our explanation is that previous work has mostly focused (with the notable exception of Xu, Fern, and Yoon (2007), discussed in Section 5) on learning heuristics for optimal search algorithms, namely iterative deepening A*. The role, and therefore the desired properties, of the heuristic in IDA* and greedy best-first search differ substantially. IDA* uses heuristics primarily for pruning, and in many implementations only pruning, while greedy best-first search uses the heuristic solely for guidance. IDA* works by expanding all nodes within a cost bound, and iteratively increasing this cost bound until a solution is contained within it. In all but the final iteration, the relative ordering of nodes is of no consequence, with the exception of the final iteration, and many implementations ignore child ordering [1]. The child ordering is of limited consequence because, excepting the final iteration, IDA* must exhaust the entire f-layer to show that no solution exists within the current bound. This, along with the way the bound is updated, guarantee that when a solution is found it will be an optimal solution.

If our goal is to exhaust all nodes with some property and not, instead, to find a goal, then we don't care what order we expand the nodes in. Accurate cost estimates allow IDA* to prune unpromising nodes early, dramatically reducing the size of these exhausted layers, and therefore dramatically reducing the search effort. In contrast, greedy search cares not one whit for accuracy in the absolute sense. Any heuristic that can correctly sort the set of all open nodes so that nodes leading to good solutions are explored earliest is acceptable, even if it is inaccurate. By way of example, the following heuristic results in perfect performance despite being infinitely inaccurate: the heuristic returns 1 on any optimal path from the root to the goal, and infinity for any other state.

3.3 Estimating $h^*(n)$ Using Backwards-looking Heuristics

If we find ourselves in a domain where the heuristic estimate of cost can be computed between two arbitrary points, we have an alternate technique for gathering information about heuristic error: we can compare the heuristic estimate of the cost-to-go from a node n to the initial state with the cost

1. The current state-of-the-art is to run IDA* with multiple action orderings in parallel (Valenzano, Sturtevant, Schaeffer, Buro, & Kishimoto, 2010),which takes advantage of child ordering, but doesn't use the heuristic to order the children.

	8-Puzzle		Life Grids		Small Vacuums	
	generated	cost	$\frac{sec}{1000}$	$\frac{cost}{1000}$	secs	cost
Baseline	582	*128*	169	2993	0.990	2673
Online LMS	514	108	216	2993	0.**0.158**	**1368**
Reverse LMS	623	36	168	**2763**	1.065	2956
Online ANN	610	56	919	5056	0.995	*5415*
Reverse ANN	*5032*	83	*1996*	*6829*	1.884	4590
SS Path	463	33	**71**	2795	0.260	2100

Table 11: Performance in greedy search on domains from accuracy study

of arriving at that node from the initial state during this search, $g(n)$. This would be especially appealing if we knew that we had arrived at a node by an optimal path, as we would have if we were performing uniform cost search or A* search with a consistent heuristic (Pearl, 1984). $g(n)$ is very likely to be suboptimal in the kinds of searches we consider in this paper, but we can still use it as an approximation of the true cost between an arbitrary node n and the root.

We can learn $\widehat{h}(n)$ as a weighted combination of features pointing from n to the initial search state using any of the previously described regression techniques. The target value of these weighted features is $g(n)$, an approximation of the optimal cost of navigating between a given node n and the initial state. When we want to produce a forward looking estimate (ie from n to the goal), we simply use features that relate n to the goal rather than to the root. If our forwards and backwards looking features are similarly informed, as we would suspect them to be if they were heuristics computed using the same relaxation, then this should produce a reasonable estimate for $\widehat{h}(n)$.

More concretely, assume that we have a cost-to-go and distance-to-go heuristic that can be computed between arbitrary states, $h(n, m)$ and $d(n, m)$ respectively. When we present training examples to these learning algorithms, we present $g(n)$ as the target value, and $h(n, root)$ and $d(n, root)$ as features. When we want to compute $\widehat{h}(n, goal)$, then we use $h(n, goal)$ and $d(n, goal)$ as features. All of the previously used features have a corresponding backwards looking feature. $g(n)$ can be estimated by $h(n, goal)$, $h(n)$ can be mapped to $h(n, root)$, $depth(n)$ as $d(n, goal)$, and $d(n)$ as $d(n, root)$. It should be noted that such an approach is not nearly as general as those discussed previously. It limits us to domains where we can efficiently compute heuristics between arbitrary states.

3.3.1 EVALUATION: BACKWARDS LOOKING HEURISTICS

We evaluated two heuristic learning techniques based on heuristics that look towards the initial state of the search space. We use backward looking features (the heuristics computed towards the root for h and d, and the heuristics computed towards the goal for g and $depth$) and $g(n)$ as a target value. We examine the following regression techniques:

Reverse LMS Least mean squared linear regression.

Reverse ANN Estimating the remaining cost-to-go using an Artificial Neural Network. The ANN is constructed as before, with the same random weights and the same initial training period.

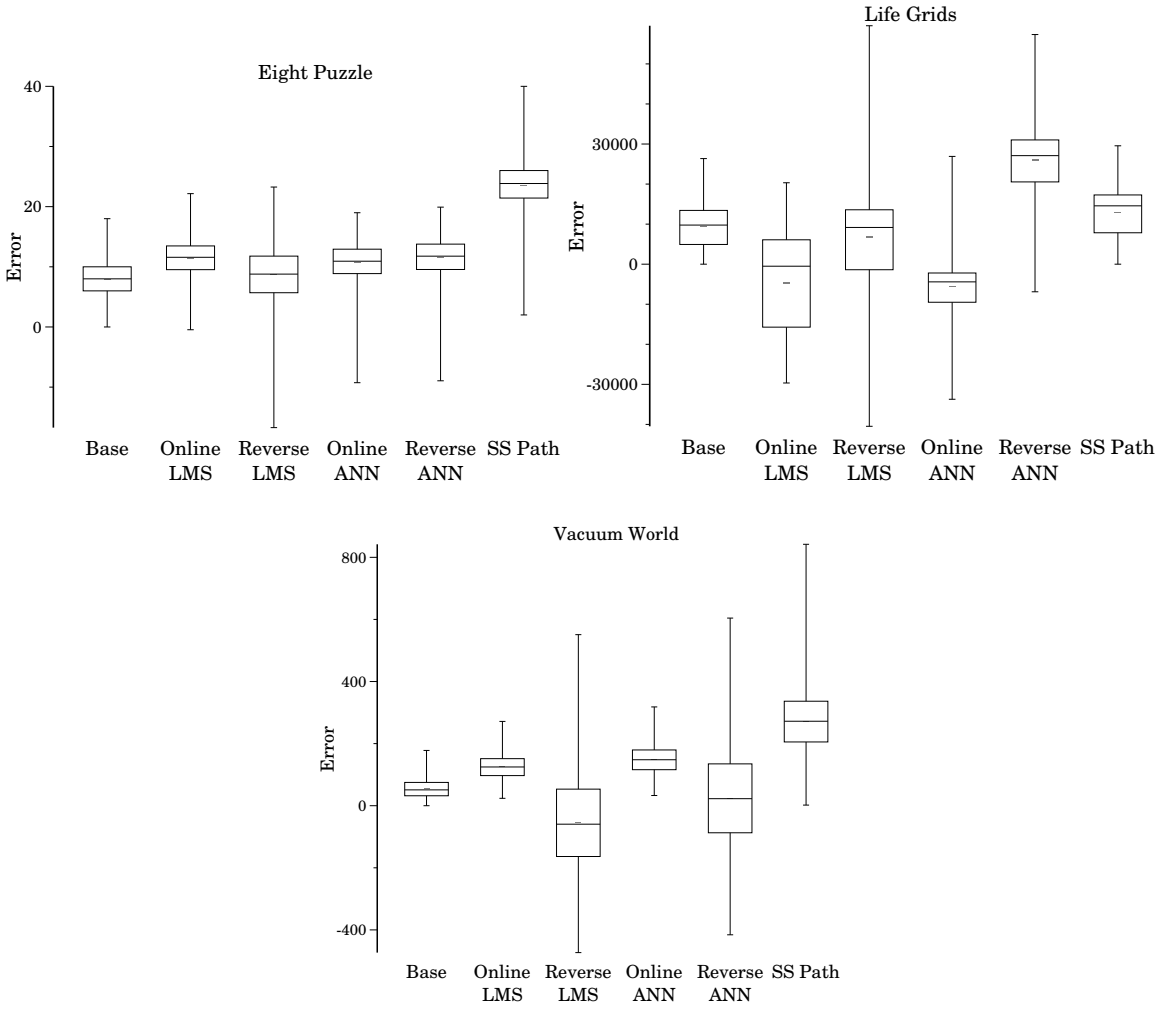


Figure 12: Accuracy of heuristics constructed with standard machine learning techniques and backwards looking heuristics on 8-puzzles, "life" grids, and small vacuum problems

	Fifteen Puzzle		Dynamic Robot		Dock Robot		Vacuums	
	$\frac{secs}{1000}$	cost	$\frac{secs}{1000}$	cost	secs	cost	secs	cost
Baseline	29	302	60	522	*169*	*Failed 55*	9.07	9635
Online LMS	**8**	520	75	522	73	Failed 17	9.42	9635
Reverse LMS	25	150	17128	95	–	–	6.556	9648
Online ANN	*2444*	719	3418	*881*	135	Failed 47	7.40	6155
Reverse ANN	531	*831*	*465133*	254	–	–	*14.28*	*13525*
SS Path	15	**90**	**14**	**47**	**0.385**	**29**	**1.22**	**6063**

Table 12: Comparing online LMS and ANN's to single-step corrections on larger problems

Figure 12 shows the absolute accuracy of the backwards looking regression approaches over three benchmark domains. While they can produce more accurate estimates, most noticeable in the vacuum world domain where both reverse LMS and reverse ANN heuristics have better means than their forward looking counterparts, they tend to have a much wider variance than the other techniques, something that holds for all three domains. While they can produce better estimates, they don't always, as is the case for life grids where the reverse looking ANN heuristic produces a substantially less accurate estimator than its forward looking counterpart.

Table 11 shows the performance of the backwards-looking heuristics in terms of absolute accuracy on the 8 puzzle, "life" grid navigation, and our small vacuum benchmark. We see that, perhaps surprisingly, they do not perform substantially better than similar techniques that look forwards. This is likely because the target values being used for training, the g-values of the nodes being expanded, are much higher than their optimal values. When a node is expanded by an A* search on an admissible and consistent heuristic, we know it is expanded with its optimal g-value. Greedy search on potentially inadmissible heuristics enjoys no such guarantee. It appears that, empirically, this harms the performance of the algorithm. When we consider the additional overhead of computing the backwards looking heuristics together with the large variance of the resulting estimators, it is unsurprising that they perform worse when used in search.

We see similar results for the larger domains in Table 12. The learning algorithms that rely on heuristics that look towards the root are omitted for the dock robot domain because we cannot construct similarly informed heuristics in both directions, highlighting a limitation of the approach. The backwards looking corrections rely on our ability to compute a *similarly informed heuristic* between arbitrary states in the space efficiently. The base heuristic we use in this domain isn't from state to state, but from one state to a set of states, since many states satisfy the goal. Thus it is asymmetric.

3.4 Summary

One might ask what we lose, in terms of guidance and accuracy, by restricting ourselves to only the information available during search. In this section we compared the performance of the online techniques to heuristics similarly trained offline. We found that the offline techniques generally produced heuristics that were more accurate than those learned during the course of the search itself. Despite being more accurate, these heuristics actually produced worse performance when used in best-first heuristic search algorithms. This was especially surprising considering the success such approaches have enjoyed in previous work on learning heuristics for optimal or near optimal search. We pointed out that the purpose of a heuristic in an optimal search is substantially different than that in a suboptimal search. Specifically, in optimal search we need the heuristic to be accurate so that we can effectively prune away unpromising portions of the space early allowing us to prove solution optimality. In suboptimal search we merely need the heuristic to guide us towards a goal, and the accuracy of the estimations with respect to truth is a secondary concern at best.

4. Learning Interleaved with Search

This paper is primarily concerned with the problem of learning heuristics online during search on a single instance. A strongly related problem is that of learning heuristics while solving a large set of problems. Techniques for this setting are closely related for two reasons. First, single-instance methods can be directly applied to multiple instances individually or, as discussed in Section 2.5,

heuristics learned while solving one instance can sometimes be transferred to other instances. In our case, the learned single-step errors $\bar{\epsilon}_h$ and $\bar{\epsilon}_d$ can be passed between instances. Second, any technique that learns an improved heuristic while solving multiple instances can be made to work on a single instance by first constructing a training set. We now discuss two techniques designed specifically for the multiple-instance setting.

4.1 Bootstrap Learning

Jabarri Arfaee et al. (2010) showed that the process of solving a set of instances can be shortened by interleaving learning with solving. Their bootstrapping method attempts to solve all of the instances in a set within a time bound using a base heuristic, h_0. It then uses information from the solved instances, including the true cost-to-go for states along optimal paths and a set of features, to train a new heuristic using an ANN. This process then iterates, using the newly constructed heuristic as a feature, over the unsolved instances until all instances are solved. In addition to solving the instances, this procedure also results in new heuristics. If an insufficient number of the instances are solved in any given iteration, new easy-to-solve instances are automatically generated by random walks backwards from the goal.

Unlike the techniques discussed previously, bootstrapping learns in between episodes of search, not concurrently with it. When faced with a single target instance, bootstrapping generates a set of instances of progressively increasing difficulty to solve along with the target instance. Effectively, it takes the single problem setting and reduces it to the multi-problem setting by generating a set of instances to solve and learn from. The actual generation process cleverly constructs a set of problems that are almost guaranteed to be of increasing difficulty, a property that bootstrapping finds beneficial. It does this by using a series of longer and longer random walks backwards from the goal state of the problem. Further details are given by Jabarri Arfaee, Zilles, and Holte (2011).

While bootstrapping avoids the need for a set of training instances, it still assumes that the instances are similar enough for the learning to transfer effectively. It also makes two additional assumptions that may not be immediately obvious. The first is that there is some function that allows us to expand nodes backwards. In domains with reversible actions, this exists trivially, in others we must construct such a function. The second assumption is that a fixed goal state exists. There are some problems, such as STRIPS planning, for which the goal is only partially specified, leading to a potentially huge set of goal states from which we must regress in order to generate training instances. It is also implicitly assumed that the base heuristic is too weak to solve the instances we care about, as otherwise no learning ever occurs.

4.1.1 COMPARISON OF BOOTSTRAPPING AND SINGLE-STEP CORRECTIONS

Table 13 compares the performance of bootstrapping and single-step corrections on the 24-puzzle (a 5x5 sliding tile puzzle). The results for Bootstrapping are taken from Jabarri Arfaee et al. (2011) and personal communications with the authors. The table is split into two halves. The top shows results for the search algorithm when solving 500 random instances of the 24-puzzle, the second shows results for a larger set of 5000 instances. In both cases, we use the same instances used in Jabarri Arfaee et al. (2011). The columns show the time consumed while solving all instances, and the cost of all solutions summed together appears in the final column. We must take care to note that the algorithms were implemented in different languages and run on different machines, so the timing results are not directly comparable. This table reveals two huge disparities between these two

	Total Time	Total Cost
500 instances		
IDA* Bootstrapping	*42180 seconds	*73878
Greedy Bootstrapping	322 seconds	167484
Manhattan Distance	1921 seconds	Failed 1
SS Path	87 seconds	139674
5000 instances		
IDA* Bootstrapping	*421200 seconds	*575402
Greedy Bootstrapping	2263 seconds	Failed 1 (1687677)
Manhattan Distance	21596 seconds	Failed 17
SS Path	828 seconds	1387004

Table 13: Comparison of bootstrapping and single-step corrections on the 24-puzzle. * denotes results taken from (Jabarri Arfaee et al., 2011)

approaches to learning for heuristic search. The path-based corrections are three orders of magnitude faster than bootstrapping, but they produce solutions of much higher cost. Bootstrapping takes nearly 12 hours to solve 500 random instances of the 24-puzzle, whereas path-based corrections take around 90 seconds. For 5000 random instances, this gap widens proportionally with bootstrapping taking several days and path-based corrections solving all 5000 instances in 14 minutes. While the timing results are not directly comparable, the gaps in solving time are so large that we can reasonably conclude that greedy search on path-based corrections is much faster than bootstrapping on the 24-puzzle.

The huge disparities in solving time and solution quality reflect a fundamental difference in the goals of the two approaches. This difference is clearly outlined by the choice of search algorithm the learned heuristic is used in. Bootstrapping relies on a search algorithm designed for finding optimal solutions (although by using the bootstrapped heuristic such guarantees are abandoned); it was proposed using IDA*. The near optimality of the solutions returned by a search in bootstrapping are fundamental to the technique; we assume that the solutions returned are optimal (or near optimal) and are thus suitable for use as target values for learning better informed heuristics. Using wildly inflated costs would lead to inaccurate heuristics, which goes against the intent of the technique.

In contrast, we evaluate our approaches in suboptimal and bounded suboptimal algorithms. As we discussed in Section 3.2.2, the desired qualities of a heuristic differ for these two search paradigms. Optimal solvers like IDA* want very accurate heuristics, the type of heuristics that the learning in bootstrapping tends to produce. In suboptimal search, accuracy is unimportant, and ordering is key. In end effect, the two approaches are solving distinct problems: Bootstrapping wants to find nearly optimal (but not provably optimal) solutions quickly, and build an accurate estimator as a side effect, and our approaches seek to find any solution as quickly as possible, with quality being a secondary consideration. In another sense, the techniques are directly comparable as neither provides any guarantee on suboptimality bounds before search begins.

4.1.2 COMBINING BOOTSTRAPPING WITH GREEDY SEARCH

As we previously note, it is possible to use any algorithm to solve instances in the bootstrapping approach described above. While IDA* and A* have several properties that make them particularly appealing for bootstrapping, any complete search would suffice. In this section we consider using a greedy best-first search instead of the optimal IDA* search for finding solutions in the bootstrapping algorithm. Greedy search was chosen because it provides for a more direct comparison with the results presented in Table 13 and because it approaches the problem from an alternate perspective: solve all instances as quickly as possible as opposed to solving all instances as cheaply as possible.

Results are presented in Table 13. The original bootstrapping results, taken from Jabarri Arfaee et al. (Jabarri Arfaee et al., 2011), are listed as Bootstrapping IDA*, and marked with a * to denote that the cpu results are not directly comparable. Bootstrapping with greedy search being used to solve instances is denoted as Bootstrapping Greedy. Greedy search with single-step heuristic corrections and with the base Manhattan distance heuristic are labeled as before. We show the best result of several configurations of greedy bootstrapping. Bootstrapping has, as a parameter, the initial time allotment to solve an instance. We tested with values of 0.001, 0.1, 0.5, 0.75, 1, and 5, but found that a timeout of 0.1 provided the best results. For larger timeouts, the bulk of all the instances were solved before any learning occurred, leading to bad poor performance. The smallest timeout caused learning to occur frequently, resulting in a longer overall solving time without substantial gains in aggregate solution cost.

Again, the results are presented in Table 13. Here, we can see two major trends. First, the greedy results, be they the base heuristic or based on learning, are consistently faster but more expensive than the approach based around IDA*. Considering the differing goals of IDA* and greedy search algorithms, such differences in performance should be expected. The second major trend we see is that learning is consistently beneficial. Both bootstrapping greedy search and SS Path based corrections in greedy search outperform greedy search on the base heuristic. In the table, the results for greedy bootstrapping are not as good as those for SS Path based heuristic learning. The difference in nodes expanded, although not shown in the table, is also substantially different between the two approaches. This suggests that the difference in performance is not a result of overhead.

4.2 Interleaved Linear Regression

Bramanti-Gregor and Davis (1993) also proposed a technique, called SACH, that iteratively improves a heuristic used for solving a batch of problems. Using the current heuristic, they attempt to solve all of the problems in a set of instances within a given expansion bound using A* search. Any instances that are solved are used to train a new heuristic using linear regression against h^*. The process then repeats until all instances are solved. If all of the remaining instances are too difficult to solve using the current heuristic, it applies a weight to the current heuristic. Again, we must be able to assume that all of the instances we are trying to solve are similar enough to one another to allow learning to transfer across instances.

In addition to the interleaved approach proposed in Bramanti-Gregor and Davis (1993), a related paper shows how to perform SACH online for a single instance (Bramanti-Gregor & Davis, 1991). To learn during a search, SACH looks at the nodes on the search frontier. It uses parent pointers to trace backwards from these nodes to the root of the search. For each state along the path from the fringe to the root, it records the difference in g-values and a set of features. It uses these to learn

	Fifteen Puzzle		Dynamic Robot		Dock Robot		Vacuums	
	$\frac{secs}{1000}$	cost	$\frac{secs}{1000}$	cost	secs	cost	secs	cost
Baseline	29	302	60	522	169.297	Failed 55	9.073	9635
SACH	*149613*	*Failed 21*	*236815*	*Failed 6*	*238.142*	*Failed 83*	*85.824*	*Failed 2*
SS Path	**15**	**90**	**14**	**47**	**0.385**	**29**	**1.218**	**6063**

Table 14: Performance of SACH compared to other search algorithms

an estimate of the cost-to-go from arbitrary nodes to the goal. The technique used for learning by SACH can learn from arbitrary states, and so it does not need to completely solve an instance to perform learning in the same way that bootstrapping does. The learning is very similar to what we proposed in Equation 31, except that instead of using differences between a parent and its best child, it uses differences between a fringe node and all of its ancestors to create training data.

Table 14 shows the performance of online, single-instance SACH on the larger benchmark domains from our evaluations. SACH doesn't perform very well when compared to the other algorithms, especially the single-step path-based corrections shown in the table. Again, a large part in the difference in performance is due to the underlying search algorithm. At the heart of SACH is a search algorithm intended to find optimal solutions, A*.

5. Learning Search Orderings Directly

The previously discussed techniques attempt to learn an improved estimate of cost-to-go to be used in guiding the search towards goals. While learning cost estimates is quite popular (Samadi, Felner, & Schaeffer, 2008a; Jabarri Arfaee et al., 2010; Bramanti-Gregor & Davis, 1993; Fink, 2007), Xu et al. (2007) point out that it is not the only approach. They propose two search algorithms, LaSO-BR and LaSO-BST, that rely on a technique that directly learns an ordering over nodes based on the performance of that ordering in a beam search.

⟨FIX: RESPONDS TO REVIEWER 1: RE-WRITE NEXT BIT TO DISCUSS BOTH VARIANTS OF BEAM SEARCH AT LENGTH. BE SURE TO POINT OUT THE DRAWBACKS OF A BEST FIRST BEAM SEARCH AT LENGTH, CITING WILT AND OTHERS. THE MOTIVATION OF WINDOW SEARCH MAY BE INFORMATIVE FOR THAT DISCUSSION⟩

A beam search is a form of breadth-first search where the size of the open list, the nodes which have been generated but not yet expanded, is kept to a fixed size. This size is referred to as the beam width of the search, typically denoted b. All nodes on the beam are expanded simultaneously, all children are added to the open list, and then the open list is pruned until it is no larger than the beam width. Plain beam search is a form of memory limited search; by controlling the width of the beam, you can limit how many nodes need to be considered at any time, thereby limiting the maximum amount of memory consumed by a beam search. For domains with many duplicate paths to the same state and many potential cycles, beam searches need to implement a closed list to be effective (Wilt, Thayer, & Ruml, 2010). Having a closed list removes the limited-memory property of beam search algorithms, but allows them to solve a wider variety of problems.

Rather than performing a linear regression from the features of a node to truth, the LaSO technique learns a weighting over the features that would prevent a beam search with a given beam width b from pruning away all nodes leading to optimal solutions. In essence, the algorithm works by simulating a beam search forward from the root of the search problem. It repeatedly expands

Update-BR$(\mathbf{S_i}, \mathbf{P_i}, \mathbf{b}, \mathbf{w})$
//$S_i = \langle I_i, s_i(\cdot), f(\cdot), <_i \rangle$ and $P_i = \{P_{i,0}, ..., P_{i,maxdepth}\}$
//I_i is the root node, $s_i(\cdot)$ is the successor function
//$f_i(\cdot)$ generates features of a node
//P_i is the set of all nodes along a desirable path to the goal
1. $B \leftarrow I_i$
2. for $depth = 1$ to $maxdepth$
3. $C \leftarrow$ **BreadthExpand**$(B, s_i(\cdot))$
4. for every $v \in C$
5. $H(v) \leftarrow w \cdot f(v)$ // compute heuristic value of v
6. Order C according to H and the total ordering $<_i$
7. $B \leftarrow$ the first b nodes in C
8. if $B \cap P_{i,depth} = \emptyset$ then
9. $w \leftarrow w + \alpha \cdot (\frac{\sum_{v^* \in P_{i,depth} \cap C} f(v^*)}{|Pi,depth \cap C|} - \frac{\sum_{v \in B} f(v)}{b})$
10. $B \leftarrow P_{i,depth} \cap C$

Figure 13: Update Rule For LaSO-BR

all nodes in the current beam and sorts them based on the current weight vector and features of the node. If, when forming the next beam based on the expansion of the previous beam, all nodes that lead to an optimal solution have been pruned, the weights are updated. The weights are updated to promote nodes on optimal paths that could have been in the beam but were not because of the weight vector. Then, the current beam is set to be the remaining optimal nodes on open. This process is performed offline before the algorithm is used to solve problems. It requires a set of training instances that can be optimally solved by some other technique such as A* or IDA*.

This ranking function can be learned from either best-first beam search or breadth-first beam search. We refer to these approaches as LaSO-BST and LaSO-BR respectively. Training for LaSO-BST often takes far longer than training for LaSO-BR. The first reason for this is that it often takes best-first beam search longer to solve a problem than breadth-first beam search. ⟨FIX: CITE EVIDENCE OF THE POOR PERFORMANCE OF BEST-FIRST BEAM SEARCH HERE AS WELL.⟩ The second is that it is rarer for a best-first beam search to prune away all nodes leading to an optimal solution because it only expands a single node at a time. If there are multiple paths to an optimal solution, as there are in all of the domains considered in this paper, it is likely that several optimal nodes exist in the beam. Unless the children of the node being expanded manage to drive them all out, LaSO-BST will perform no learning in this step. LaSO-BR, on the other hand, expands all nodes at once. Presumably, the optimal nodes are a small portion of the existing beam and they likely only have one or two children on the optimal path. Thus, the optimal nodes must beat out many competitors to be included in the next beam, they often don't, and so learning occurs more frequently in practice.

Pseudo-code for updating the weights in the breadth-first beam search variant of LaSO is provided in Figure 13. In essence, the algorithm works by simulating a breadth-first beam search. It repeatedly expands all nodes in the current beam (line 3) and sorts them based on the current weight

Algorithm	Unit 15-puzzle $\frac{sec}{1000}$	Unit 15-puzzle Cost	Vacuum World sec	Vacuum World Cost	Dock Robot sec	Dock Robot Cost
Base	*29*	*302*	9.07	9635	169	Failed 55
LaSO-BR	85	392	*142*	*Failed 7*	*577*	*Failed 98*
SS Path	**15**	**90**	**1.22**	**6063**	**0.385**	**29**

Table 15: Heuristic performance in greedy search

vector and features of the node (lines 5 & 6). If, when forming the next beam based on the expansion of the previous beam, all nodes that lead to a good solution have been pruned (lines 7 & 8), the weights are updated (line 9). The weights are updated to promote nodes on good paths that could have been in the beam, $P_{i,j} \cap C$, but were not because of the weight vector. The code for LaSO-BST is similar, but the breadth-first beam search is replaced with a best-first beam search.

Note that the beam width to be trained for is a parameter of the LaSO learning technique (line 7 of Figure 13). This makes adapting the learning technique of LaSO to general heuristic search difficult. How to set the beam width to get the best performance for our learned heuristic in a different search algorithm is an open question. For our evaluation, we tried multiple beam widths, 1, 3, 5, 10, 50, 100, 500, and 1000, and then report results for the best-performing beam width for the algorithm.

5.1 Evaluation: Greedy Search

Table 15 shows the performance of the learned heuristics in greedy best-first search across three benchmark domains. It shows that greedy best-first search on single-step path corrections performs best in terms of time and solution quality for the examined domains. It never fails to solve an instance whereas the baseline and LaSO heuristic do in heavy vacuums and dockyard robots respectively. We should also note that the heuristic learned for use in LaSO-BR performs substantially worse than the baseline in two domains. There are two reasons behind this. The first is that in domains where the LaSO heuristic is performing poorly, the learning is unlikely to generalize well. Consider the tiles domains, where the LaSO heuristic substantially outperforms the baseline. Here, the underlying state space is identical (unit-cost) or incredibly similar (inverse cost) across problems, and therefore the learned ordering generalizes well. Contrast that with the dockyard robot domain, where the goal configuration and the cost of transition between depots changes across instances. Here the learned node ranking performs poorly.

Secondly, the LaSO heuristic was trained to be used in beam search, not a best-first search. The role of the heuristic is different in these two kinds of search algorithms, just as the role of the heuristic in greedy search and IDA* differs. In best-first search, we want to push goals, or nodes leading to goals, all the way to the front of the open list. In a beam search the heuristic need only prevent us from pruning away all promising nodes. We can see in line 8 of Figure 13 that is exactly what we are training the LaSO heuristic to do. The weights are only updated when all of the promising nodes are pruned away from the beam. In light of that, we shouldn't expect the LaSO heuristic to perform well in greedy best-first search because it isn't designed to provide the right kind of guidance.

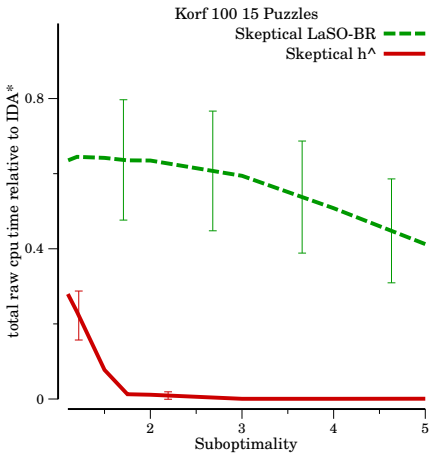


Figure 14: Single-step path corrections versus LaSO-BR in skeptical search on the 15-puzzle

	Beam Width					
	10		100		1000	
Ordering	Time	Cost	Time	Cost	Time	Cost
15-puzzle						
Base	**0.0029**	**173**	**0.0249**	**73**	**0.2999**	**59**
SS Path	*0.0076*	*204*	*0.0360*	*74*	*0.4286*	**59**
LaSO BR	0.0061	191	0.0296	74	0.3345	**59**
Vacuum World						
Base	*576.0024*	*Failed 145*	*41.3884*	*Failed 17*	*32.9744*	**Failed 1**
SS Path	576.0022	Failed 141	3.5336	Failed 2	32.6410	**Failed 1**
LaSO BR	**372.1086**	**Failed 104**	**3.5102**	**Failed 1**	**30.9520**	**Failed 1**
Dock Robot						
Base	*97.1460*	*Failed 8*	24.2612	Failed 1	25.2046	Failed 1
SS Path	**0.1460**	**112**	**0.3962**	**29**	**1.6446**	**15**
LaSO BR	97.1544	Failed 6	*151.6342*	*Failed 10*	*299.7842*	*Failed 21*

Table 16: Learning techniques in breadth-first beam search

5.2 Evaluation: Bounded Suboptimal Search

Figure 14 shows the relative performance of LaSO-BR and the best single-step correction technique in skeptical search, a bounded suboptimal search. Since LaSO-BR does not learn a cost-to-go estimate, we perform the initial search on the learned heuristic directly, and then perform cleanup on $f(n)$. We see that for all suboptimality bounds shown, skeptical search on single-step path corrections outperforms skeptical search relying on the LaSO heuristic. Investigation of the results showed that, while skeptical search can construct an initial solution much faster when using the LaSO heuristic, the solution found is much more expensive. Even if this incumbent is within the bound, proving this requires more effort than showing that a solution with lower cost is within the same bound. Results for the other domains are similar.

5.3 Evaluation: Beam Search

⟨FIX: FOR REVIEWER 1: REITERATE THAT IN GENERAL, BEAM SEARCH IS THE WRONG SOLUTION TO SOLVING MOST PROBLEMS IN PRACTICE. BETTER (COMPLETE!) TECHNIQUES EXIST THAT SHOULD BE USED. THE EVALUATION IS HERE SO THAT WE MIGHT HAVE SOME SENSE OF THE FIRST ITERATION OF BEAM STACK, ETC, AND FOR THE CURIOUS.⟩

The previous evaluations of LaSO were, in some sense, unfair because LaSO wasn't designed to be used in general search algorithms. It was designed to be used in beam search. Table 16 shows the relative performance of the learned heuristics in breadth-first beam search for differing beam widths and domains. In the table, rows are the heuristic used to sort the beam, and major columns show the beam width. As before, each major column is divided into two minor columns that report the time required to find a solution and the solution cost respectively.

The first results, those showing the performance on fifteen puzzle (first row of Table 16), are particularly surprising because the techniques which learn their heuristics appear to be dominated by search on the base heuristic. However, the lower mean time to solution is primarily a result of reduced overhead rather than significantly fewer node expansions. The solution costs for each beam are within noise of one another, meaning that the depths to which each beam search is going in this domain are incredibly similar and therefore the number of nodes generated are similar. They are in fact not statistically distinct for many of the beam widths. We see a similar phenomenon for the vacuum world domain, where LaSO BR appears to outperform search on single-step path corrections, but the values are statistically indistinguishable from one another (the confidence intervals overlap significantly).

Table 16 also reveals that, as was the case in greedy search, LaSO-BR fails to solve many of the instances in the dockyard robot domain. Again, we attribute this to the fact that the underlying instances are very different from one another. This impedes the performance of techniques which perform all of their learning offline. Note that greedy search on the LaSO heuristic (Table 15, second row) also performs poorly, and so this performance is likely the fault of the heuristic and not the search algorithm itself.

Both LaSO-BR and single step corrections generally improve the performance of beam search. At worst, they do not harm performance. Single-step path corrections provides better guidance in beam search than the LaSO heuristic. From Table 16, we see that it solves more instances and it tends to have lower mean solving time and better mean solution quality. While these times are not always distinguishable from search performed on the LaSO heuristic, in the dockyard robot domain search on single-step path corrections is clearly better than search on the LaSO heuristic.

5.4 Summary

Suboptimal search algorithms need functions that can effectively discriminate between nodes to guide search. Learning exactly what we need is very appealing. The previous technique for learning search orders directly, LaSO, is designed for beam search. Unfortunately it does not appear to work as well when used in other kinds of search algorithms. Although the single step techniques proposed in this work provide the largest advantage in best-first search algorithms, they can be competitive with LaSO techniques when used in beam search.

6. Other Related Work

We now present alternative and complementary techniques for learning before any search begins, and in between multiple runs on a single instance. We say that the techniques are complementary as the single-step error corrections presented here could be added to these techniques to improve performance.

The most popular, or at least the most frequently proposed in the literature, technique for learning heuristics for search is to learn those heuristics before any search of the target instances begins, offline, from training data. All such techniques assume that training instances are abundant, or at least that they are easily generated. Further, several of the following approaches make use of strong domain-specific features to use for the learning of heuristics. Both of these assumptions limit the applicability of the techniques.

Samuel's checker playing program (1959) used learning techniques to construct good static evaluators to be used in his alpha-beta pruning game tree search and it is the earliest to make use of learning techniques for constructing heuristic evaluation functions. Positional strength is not the same quality as cost-to-go, so this technique is not directly comparable, or even easily combined, with those presented here.

Sarkar, Chakrabarti, and Ghose (1998) learn to identify sets of nodes with interesting properties, such as nodes that are likely to lie on a path to a solution or nodes that are more likely to be near to solutions. They then use this classification to perform efficient tie-breaking in optimal search algorithms. Learning is performed offline, from training data, before any search over the target instances begins.

Samadi et al. (2008a) present a technique for combining an arbitrary number of features into a single cost-to-go estimate. In their implementation, these features are pre-computed pattern databases, powerful heuristics in their own right. They train an artificial neural network (ANN) to map these values to an estimate of the cost-to-go using h^* as the target value. When problems are too large to solve optimally, they substitute the optimal solution of a relaxed problem for h^*. Naturally, this lessens the quality of the training data and leads to slightly worse estimates as a result. Samadi, Siabani, Felner, and Holte (2008b) provide a technique for compressing pattern databases efficiently that could be used here to ensure admissibility. While we do not rely on admissibility, such a powerful cost-to-go estimate would likely make a good starting heuristic for our online technique.

Fink (2007) proposes a technique that learns an improved heuristic for multiple searches over the same instance of a pathfinding problem. Specifically, he assumes that the same graph is being searched every time, but that the start and goal nodes may change. A cost-to-go heuristic is learned in between search episodes using information recorded during the previous search. Features of a node are recorded and a heuristic is learned by performing a regression from these features to the true cost-to-go. As more problems are solved, more data becomes available and the quality of the heuristic improves as a result of that. While bootstrapping and the original implementation of SACH were exclusively evaluated on permutation puzzles, where each solution shares the same underlying search space, it can be run without alteration on problems where the underlying state space differs between instances. This isn't obviously the case for the technique proposed by Fink.

7. Discussion

There are three times when learning can happen: before any search, in between solving instances of a batch, or during the execution of a search. We do not thoroughly investigate the possibility of combining offline or interleaved learning with online learning in this paper. As we've shown that the online technique works with the base heuristic and generally improves when the accuracy of the underlying heuristic improves, it is likely that a combination of the techniques would be very beneficial.

Nearly all of the previous work has focused on finding optimal or near optimal solutions. There have been very few techniques that consider speed as the primary figure of merit. Learning heuristics generally leads to a certain amount of inadmissibility, preventing us from guaranteeing cost-optimality. There are many applications of search, and while many demand solutions of the highest possible quality there are important settings that require us to solve problems quickly.

Online techniques for improving heuristics allows us to take advantage of the information present in every expansion. By definition, search algorithms tend to spend a majority of their time searching. Every expansion, of which there will in the worst case for search (but the best case for learning) be many, provides an opportunity to learn a potentially improved evaluation function.

A point that arose several times in our investigation is that different kinds of search algorithms have differing requirements for their heuristics. For finding optimal search, the problem that nearly all previous work focuses on, we need the heuristic to be extremely accurate in terms of absolute error. That is, the heuristic must be able to very accurate predict the true cost-to-go, h^*. This is because in optimal search the heuristic is used to prove that the returned solution is optimal (ie expand all nodes where $f(n) \leq g(opt)$). The absolute magnitude of the heuristic determines what portion of the search space we must exhaustively search before we can prove that the solution we find is of a sufficient quality. If we are unconcerned with proving quality bounds, or if time is at a larger premium than quality, we should use search algorithms that rely on the guiding power of a heuristic. Here we are not exhausting large portions of the space to prove quality and the limiting factor of the search is how quickly we can guide the algorithm into goals. Any heuristic that assigns a node close to a solution a relatively smaller value than one far away will work well here, regardless of how far away its estimates are from truth.

8. Conclusions

Learning for heuristic search had previously been considered primarily in two settings: learning an improved heuristic offline, before any search begins, and learning an improved heuristic between the solving of instances in a large batch. The technique presented in this paper, learning corrections from single-step error, learns during the execution of the search itself. It can be easily combined with either, or both, of the other two settings to improve performance. Our technique has the advantage of making few assumptions. Specifically, we do not assume a training set or the ability to generate one, we do not assume we can solve problems optimally, and we don't assume that all of the instances being solved are similar. We merely require that a heuristic search algorithm is being used, and we need a cost-to-go and a distance-to-go heuristic. Both are likely to exist for any given domain. This allows the described approach to be widely and immediately applicable. In our evaluation, we found that our technique produces better solutions faster than the base heuristics when used in greedy best-first search across a wide range of benchmark domains. The technique also proved to

be beneficial in bounded suboptimal search, improving upon the performance of previous state of the art algorithms while removing the need for parameter tuning.

Acknowledgments

A preliminary version of this work was published by Thayer, Dionne, and Ruml (2011). The authors would like to thank Shahab Jabarri Arfaee, Sandra Zilles, and Rob Holte for their helpful comments and personal communications related to bootstrapping. This work was supported by NSF (grant IIS-0812141) and the DARPA CSSG program (grant N10AP20029).

References

Benton, J., Talamadupula, K., Eyerich, P., Mattmueller, R., & Kambhampati, S. (2010). G-value plateuas: A challenge for planning. In *Proceedings of the Twentieth International Conference on Automated Plannin and Scheduling*.

Bramanti-Gregor, A., & Davis, H. (1993). The statistical learning of accurate heuristics. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pp. 1079–1085.

Bramanti-Gregor, A., & Davis, H. W. (1991). Learning admissible heuristics while solving problems. In *Proceedings of the Twelth International Joint Conference on Artificial Intelligence*, pp. 184–189.

Christensen, J., & Korf, R. (1986). A unified theory of heuristic evaluation functions and its application to learning. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pp. 148–152.

Dechter, R., & Pearl, J. (1988). The optimality of A*. In Kanal, L., & Kumar, V. (Eds.), *Search in Artificial Intelligence*, pp. 166–199. Springer-Verlag.

Doran, J. E., & Michie, D. (1966). Experiments with the graph traverser program. In *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, pp. 235–259.

Ebendt, R., & Drechsler, R. (2009). Weighted A* search - unifying view and application. *Artificial Intelligence*, *173*, 1310–1342.

Fink, M. (2007). Online learning of search heuristics. In *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics*.

Ghallab, M., & Allard, D. (1983). A_ϵ: An efficient near admissible heuristic search algorithm. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*.

Ghallab, M., Nau, D., & Traverso, P. (2004). *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers.

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

Helmert, M., & Röger, G. (2007). How good is almost perfect?. In *Proceedings of the ICAPS-2007 Workshop on Heuristics for Domain-independent Planning: Progress, Ideas, Limitations, Challenges*.

Holte, R. C. (2010). Common misconceptions concerning heuristic search. In *Proceedings of the Third Annual Symposium on Combinatorial Search*, pp. 46–51.

Jabarri Arfaee, S., Zilles, S., & Holte, R. (2010). Bootstrap learning of heuristic functions. In *Proceedings of the Third Annual Symposium on Combinatorial Search*.

Jabarri Arfaee, S., Zilles, S., & Holte, R. (2011). Learning heuristic functions for large state spaces. *Artificial Intelligence*.

Korf, R., & Felner, A. (2002). Disjoint pattern database heuristics. *Artificial Intelligence*, *134*, 9–22.

Korf, R. E. (1985). Iterative-deepening-A*: An optimal admissible tree search. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp. 1034–1036.

Likhachev, M., Gordon, G., & Thrun, S. (2003). ARA*: Anytime A* with provable bounds on sub-optimality. In *Proceedings of the Seventeenth Annual Conference on Neural Information Processing Systems*.

Likhachev, M., Gordon, G., & Thrun, S. (2003). ARA*: Formal analysis. Tech. rep. CMU-CS-03-148, Carnegie Mellon University School of Computer Science.

Nilsson, N. J. (1998). *Artificial Intelligence: A New Synthesis*, pp. 172–175. Morgan Kaufmann Publishers, Inc.

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Pearl, J., & Kim, J. H. (1982). Studies in semi-admissible heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *PAMI-4*(4), 391–399.

Pohl, I. (1973). The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computation issues in heuristic problem solving. In *Proceedings of the Third International Joint Conference on Artificial Intelligence*, pp. 12–17.

Russell, S., & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach* (Second edition). Prentice Hall, Upper Saddle River, New Jersey.

Samadi, M., Felner, A., & Schaeffer, J. (2008a). Learning from multiple heuristics. In *Proceedings of the Twenty-Third Conference on Artificial Intelligence*.

Samadi, M., Siabani, M., Felner, A., & Holte, R. (2008b). Compressing pattern databases with learning. In *Proceedings of the Eighteenth European Conference on Artificial Intelligence*.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*.

Sarkar, S., Chakrabarti, P., & Ghose, S. (1998). A framework for learning in search-based systems. In *IEEE Transactions on Knowledge and Data Engfineering*, Vol. 10, pp. 563–575.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, *3*, 9–44.

Thayer, J. T., Dionne, A., & Ruml, W. (2011). Learning inadmissible heuristics during search. In *Proceedings of the Twenty-First International Conference on Automated Planning and Scheduling*.

Thayer, J. T., & Ruml, W. (2008). Faster than weighted A*: An optimistic approach to bounded suboptimal search. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*.

Thayer, J. T., & Ruml, W. (2009). Using distance estimates in heuristic search. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*.

Thayer, J. T., & Ruml, W. (2010). Finding acceptable solutions faster using inadmissible information. Tech. rep. 10-01, University of New Hampshire.

Thayer, J. T., Ruml, W., & Kreis, J. (2009). Using distance estimates in heuristic search: A re-evaluation. In *Proceedings of the Second Symposium on Combinatorial Search*.

Valenzano, R., Sturtevant, N., Schaeffer, J., Buro, K., & Kishimoto, A. (2010). Simultaneously searching with multiple settings: An alternative to parameter tuning for suboptimal single-agent search algorithms. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling*.

Wilt, C., & Ruml, W. (2011). Cost-based heuristic search is sensitive to the ratio of operator costs. In *Proceedings of the Fourth Symposium on Combinatorial Search*.

Wilt, C., Thayer, J., & Ruml, W. (2010). A comparison of greedy search algorithms. In *Proceedings of the Third Symposium on Combinatorial Search*.

Xu, Y., Fern, A., & Yoon, S. (2007). Discriminative learning of beam-search heuristics for planning. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*.