Iterative-Deepening Search with On-line Tree Size Prediction

Ethan Burns and Wheeler Ruml

University of New Hampshire
Department of Computer Science
`eaburns` at `cs.unh.edu` and `ruml` at `cs.unh.edu`

Abstract. The memory requirements of best-first graph search algorithms such as A* often prevent them from solving large problems. The best-known approach for coping with this issue is iterative deepening, which performs a series of bounded depth-first searches. Unfortunately, iterative deepening only performs well when successive cost bounds visit a geometrically increasing number of nodes. While it happens to work acceptably for the classic sliding tile puzzle, IDA* fails for many other domains. In this paper, we present an algorithm that adaptively chooses appropriate cost bounds on-line during search. During each iteration, it learns a model of the search tree that helps it to predict the bound to use next. Our search tree model has three main benefits over previous approaches: 1) it will work in domains with real-valued heuristic estimates, 2) it can be trained on-line, and 3) it is able to make more accurate predictions with only a small number of training examples. We demonstrate the power of our improved model by using it to control an iterative-deepening A* search on-line. While our technique has more overhead than previous methods for controlling iterative-deepening A*, it can give more robust performance by using its experience to accurately double the amount of search effort between iterations.

Keywords: heuristic search, tree search, tree size prediction, reactive search, autonomous search

1 Introduction

Best-first search is a fundamental tool for automated planning and problem solving. One major drawback of best-first search algorithms, such as A* [6], is that they store every node that they generate. This means that for difficult problems in which many nodes must be generated, A* runs out of memory. If optimal solutions are required, iterative deepening A* (IDA*) [8] can often be used instead. IDA* performs a series of depth-first searches where each search expands all nodes whose estimated solution cost falls within a given bound. As with A*, the solution cost of a node n is estimated using the value $f(n) = g(n) + h(n)$ where $g(n)$ is the cost accrued along the path from the root to n and $h(n)$ is a lower-bound on the cost of the cheapest path to a goal node from n. We

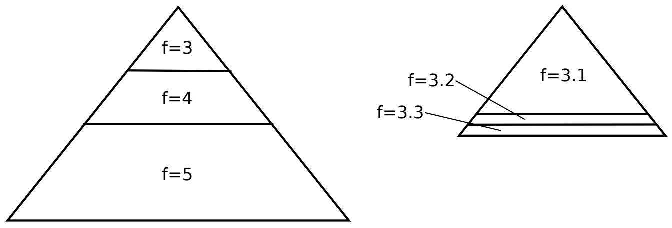

Fig. 1. Geometric versus non-geometric growth.

call $h(n)$ the *heuristic* value of n. After every iteration that fails to expand a goal, the bound is increased to the minimum f value of any node that was generated but not previously expanded. Because the heuristic estimator is defined to be a lower-bound on the true cost-to-go and because the bound is increased by the minimum amount, any solution found by IDA* is guaranteed to be optimal. Also, since IDA* uses depth-first search at its core, it only uses an amount of memory that is linear in the maximum search depth. Unfortunately, it performs poorly on domains with few nodes per f layer, because in that situation it will re-expand many interior nodes in order to expand only a very small number of new frontier nodes on each iteration.

One reason why IDA* performs well on classic academic benchmarks like the sliding tiles puzzle and Rubik's cube is that both of these domains have a geometrically increasing number of nodes in successive f layers. This means that each iteration of IDA* will re-expand not only all of the nodes of the previous iterations but also a significant number of new nodes that were previously out-of-bounds. Sarkar et al. [18] show that, in a domain with this geometric growth, IDA* will expand $\mathcal{O}(n)$ nodes where n is the number of nodes expanded by A* (the minimum required to prove a solution is optimal, short of tie-breaking nodes with cost equal to the optimal solution cost). They also show, however, that in a domain that does not exhibit geometric growth, IDA* may expand as many as $\mathcal{O}(n^2)$ nodes.

Figure 1 shows the two different behaviors graphically. The diagram on the left shows a tree with three f layers, each of an integer value and each successive layer adding a substantial portion of the tree, such that successive iterations of IDA* will each expand many new nodes that were not expanded previously. The right diagram in Fig 1, on the other hand, shows a tree with real-valued f layers. Each layer contains only a very small number of nodes, so IDA* will spend a majority of its time re-expanding nodes that it has expanded previously. Because domains with real-valued edge costs tend to have many distinct f values, they fall within this later category in which IDA* performs poorly.

The main contribution of this work is a new type of model that can be used to estimate the number of nodes expanded in an iteration of IDA*. As we will discuss in detail below, while the previous state-of-the-art approach to estimating search effort [23] is able to predict the number of expansion with surprising accuracy in several domains, it has two drawbacks: 1) it requires a large amount of off-line training to learn the distribution of heuristic values and

2) it does not extend easily to domains with real-valued heuristic estimates. Our new model, which we call an *incremental model*, is able to predict as accurately as the current state-of-the-art model for the 15-puzzle when trained off-line. Unlike the previous approaches, however, our incremental model can also handle domains with real-valued heuristic estimates. Furthermore, our model may be trained on-line during a search. We show that our model can be used to control an IDA* search by using information learned on completed iterations to determine a bound to use in the subsequent iteration. Our results show that our new model accurately predicts IDA* search effort. While IDA* guidance using our model tends to be expensive in terms of CPU time, the gain in accuracy allows the search to remain robust. Unlike the other IDA* variants which occasionally give very poor performance, IDA* using an incremental model is the only IDA* variant that can perform well over all of the domains used in our experiments. It also represents the first on-line use of detailed tree size prediction models to guide search.

2 Previous Work

Korf et al. [9] give a formula (henceforth abbreviated *KRE*) for predicting the number of nodes IDA* will expand with a given heuristic when searching to a given cost threshold. The KRE formula requires two components: the distribution of heuristic values in the search space and a function for predicting the number of nodes at a given depth in the brute-force search tree. They showed that off-line random sampling can be used to learn a sufficient estimate of the heuristic distribution. For their experiments, a sample size of ten billion states was used to estimate the distribution of the Manhattan distance heuristic on the 15-puzzle. Additionally, they demonstrate that a set of recurrence relations, based on a feature called the *type* of a node, can be used to compute the number of nodes at a given depth in the brute-force search tree for the tiles puzzle and Rubik's cube. The results of the KRE formula using these two techniques gave remarkably accurate predictions when averaged over a large number of initial states for each domain.

Zahavi et al. [23] provide a further refinement to the KRE formula called Conditional Distribution Prediction (CDP). The CDP formula replaces the heuristic distribution in the KRE formula with one that is conditioned on a set of features, such as the heuristic value and type of the parent and grandparent of each node. This extra information enables CDP to make predictions for individual initial states and to extend to domains with inconsistent heuristics. Using CDP, Zahavi et al. show that substantially more accurate predictions can be made on the sliding tiles puzzle and Rubik's cube given different initial states with the same heuristic value.

While the KRE and CDP formulas are able to give accurate predictions, their main drawback is that they require copious amounts of off-line training to estimate the heuristic distribution in a state space. Not only does this type of training take an excessive amount of time but it also does not make use of any instance-specific information. In addition, the implementation of these formulas as specified by Zahavi et al. [23] assumes that the heuristic estimates have integer values so that they can be used to index into a large multi-dimensional array. Many domains have real-valued edge costs and therefore these techniques are not directly applicable in those domains.

2.1 Controlling Iterative Search

The problem of node re-expansion in IDA* in domains with many distinct f values is well known and has been explored in past work. Vempaty et al. [20] present an algorithm called DFS* that is similar to IDA*, however, it increases the cost bound between iterations more liberally. DFS* performs branch-and-bound on its final iteration so that it can find a provably optimal solution. While the authors describe a sampling approach to estimate the bound increase between iterations, in their experiments, the bound is simply increased by doubling.

Wah et al. [21] present a set of three linear regression models to control an IDA* search. Unfortunately, this technique requires intimate knowledge of the growth properties of f in a domain for which it will be used. In many settings, such as domain-independent planning, this knowledge is not available in advance.

IDA* with Controlled Re-expansion (IDA*$_{CR}$) [18] uses a more liberal bound increase as in DFS*, however to determine its next bound, it uses a simple model. During an iteration of search, the model tracks the number of nodes that have each out-of-bound f value in a fixed-size histogram. Histograms are an appropriate choice over alternative techniques because fixed-size histograms provide constant-time operations whereas other methods, such as kernel density estimation, take linear time in the number of samples (which, in our case corresponds to the number of generated search nodes). When an iteration is complete, the histogram is used to estimate the f value that will double the number of nodes in the next iteration. The remainder of the search proceeds as in DFS*, by increasing the bound and performing branch-and-bound on the final iteration to guarantee optimality.

While IDA*$_{CR}$ is simple, the model that it uses to estimate search effort relies upon two assumptions about the search space to achieve good performance. The first is that the number of nodes that are generated outside of the bound must be at least as large as the number of nodes that were expanded. If there are an insufficient number of pruned nodes, IDA*$_{CR}$ sets the bound to the greatest pruned f value that it has seen. This value may be too small to significantly advance the search. The second assumption is that none of the children of the pruned frontier nodes of one iteration will fall within the bound on the next iteration. If this happens, then the next iteration may be much larger than twice the size of the previous. As we will see, this can cause the search to overshoot the optimal solution cost on its final iteration, giving rise to excessive search effort.

3 Incremental Models of Search Trees

To estimate the number of nodes that IDA* will expand when searching to a given cost threshold, one would ideally know the distribution of all of the f values in the search space. Assuming a consistent heuristic[1], all nodes with f values within the cost threshold will be expanded, so by using the f distribution one could simply find the bound for which the number of nodes with smaller f values matches the desired count. Our new incremental model estimates this distribution.

We will estimate the distribution of f values in two steps. In the first step, we learn a model of how the f values are changing from nodes to their offspring. In the second step, we extrapolate from the model of change in f values to estimate the overall distribution of all f values. This means that our incremental model manipulates two main distributions: we call the first one the Δf distribution and the second one the f distribution. In the next section, we describe the Δf distribution and give two techniques for learning it: one off-line and one on-line. Then, in Section 3.2, we describe how the Δf distribution can be used to estimate the distribution of f values in the search space.

3.1 The Δf Distribution

The goal of learning the Δf distribution is to predict how the f values in the search space change between nodes and their offspring. The advantage of storing these Δf increment values instead of storing the f values themselves is that it enables our model to extrapolate to portions of the search space for which it has no training data, a necessity when using the model on-line or with few training samples. We will use the information from the Δf distribution to build an estimate of the distribution of f values over the search nodes.

The CDP technique of Zahavi et al. [23] learns a conditional distribution of the heuristic value and node type of a child node c, conditioned on the node type and heuristic estimate of the parent node p, notated $P(h(c), t(c)|h(p), t(p))$. As described by Zahavi et al., this requires indexing into a multi-dimensional array according to $h(p)$ and so the heuristic estimate must be an integer value. Our incremental model also learns a conditional distribution, however in order to handle real-valued heuristic estimates, it uses the integer valued search-space-steps-to-go estimate d of a node instead of its cost-to-go lower bound, h. In unit-cost domains, d will often be the same as h, however in domains with real-valued edge costs they will differ. d is typically easy to compute while computing h [19] (for example, it is often sufficient to use the same procedure as for the heuristic but with a cost of 1 for each action). The distribution that is learned by the incremental model is $P(\Delta f(c), t(c), \Delta d(c)|d(p), t(p))$, that is, the distribution

[1] A heuristic is consistent when the change in the h value between a node and its successor is no greater than the cost of the edge between the nodes. If the heuristic is not consistent then a procedure called pathmax [13] can be used to make it consistent locally along each path traversed by the search.

over the change in f value between a parent and child, the child node type and the change in d estimate between a parent and child, given the distance estimate of the parent and the type of the parent node.

The only non-integer term used by the incremental model is $\Delta f(c)$. Our implementation uses a large multi-dimensional array of fixed-sized histograms (see Appendix A) over $\Delta f(c)$ values. Each of the integer-valued features is used to index into the array, resulting in a histogram of the $\Delta f(c)$ values. By storing counts, the model can also estimate the branching factor of the search space by dividing the total number of nodes with a given d and t by the total number of their offspring. This branching factor will be used below to estimate the number of successors of a node when estimating the f distribution.

Zahavi et al. [23] found that it is often important to take into account information about the grandparent of a node for the distributions used in CDP; we also found this to be the case for the incremental model. To accomplish this, we combine the node types of the parent and grandparent into a single type. For example, on the 15-puzzle, if the parent state has the blank in the center and it was generated by a state with the blank on the side, then the parent type would be a *side–center* node.

Learning Off-line. We can learn an incremental Δf model off-line using the same method as KRE and CDP. A large number of random states from a domain are sampled, and the children (or grandchildren) of each sampled state are generated. The change in distance estimate $\Delta d(c) = d(c) - d(p)$, node type $t(c)$ of the child node, node type $t(p)$ of the parent node, and the distance estimate $d(p)$ of the parent node are computed and a count of 1 is then added to the appropriate histogram for the (possibly real-valued) change in f, $\Delta f(c) = f(c) - f(p)$, between parent and child.

Learning On-line. An incremental Δf model can also be learned on-line during search. Each time a node is generated, the $\Delta d(c)$, $t(c)$, $t(p)$ and $d(p)$ values are computed for the parent node p and child node c and a count of 1 is added to the corresponding histogram for $\Delta f(c)$, as in the off-line case. In addition, when learning a Δf model on-line, the *depth* of the parent node in the search tree is also known. We have found that this feature greatly improves accuracy in some domains (such as the vacuum domain described below) and so we always add it as a conditioning feature when learning an incremental model on-line.

Learning a Back-off Model. Due to data sparsity, and because the Δf model will be used to extrapolate information about the search space for which it may not have any training data, a back-off version of the model may be needed. A back-off model is one that is conditioned on fewer features than the original model. When querying the model, if there is no training data for a given set of features, the more general back-off model is consulted instead. When learning a model on-line, because the model is learned on instance-specific data, we found that it was only necessary to learn a back-off model that ignores the depth feature. When training off-line, however, we learn a series of two back-off models, first eliminating the parent node distance estimate and then eliminating both the parent distance and type.

3.2 The f Distribution

Our incremental model predicts a bound that will result in expanding the desired number of nodes for a given start state by estimating the distribution of f values of the nodes in the search space. To accomplish this, the model uses the estimated f value distribution of one search depth in concert with the Δf model to generate an estimate of the f value distribution for the next depth. By beginning with the root node, which has a known f value, our procedure simulates the expansions of each depth layer to incrementally compute estimates of the f value distribution at the next layer. The accumulation of these depth-based f value distributions is an estimation of the f value distribution over the search space.

To increase accuracy, the estimated distribution of f values at each depth is conditioned on node type t and distance estimate d. We begin our simulation with a model of depth 0, which is simply a count of 1 for $f = f(root)$, $t = t(root)$ and $d = d(root)$. Next, the Δf model is used to find a distribution over Δf, t and Δd values for the offspring of the nodes at each combination of t and d values at the current depth. Recall that our incremental model stores $P(\Delta f(c), t(c), \Delta d(c) | d(p), t(p))$. Because we store Δ values, we can compute $d(c) = d(p) + \Delta d(c)$ and $f(c) = f(p) + \Delta f(c)$ for each child c with a parent p even if we have not seen nodes at this depth or f value before. This gives us an estimate of the number of nodes with each $\langle f, t, d \rangle$ combination at the next depth of the search.

Because the Δf values may be real numbers, they are stored as histograms by our Δf model. In order to add $f(p)$ and $\Delta f(c)$, we use a procedure called additive convolution [17, 16]. The convolution of two histograms ω_x and ω_y, where ω_x and ω_y are functions from values to weights, is a histogram ω_z, in which the count for a resulting f value k is the count at a parent f value i times the count of Δf at value $k - i$, summed over all possible parent values i. More formally,

$$\omega_z(k) = \sum_{i \in Domain(\omega_x)} \omega_x(i) \cdot \omega_y(k - i) \tag{1}$$

By convolving the f distribution of a set of nodes with the distribution of the change in f values between these nodes and their offspring, we get the f distribution of the offspring.

Because the maximum depth of a shortest-path search tree is typically unknown, our simulation must use a special criterion to determine when to stop. With a consistent heuristic the f values of nodes will be non-decreasing along a path [15]. This means that the Δf values in the model will always be non-negative, and the f values generated during the simulation will never decrease between layers. As soon as the accumulated f distribution has a weight that

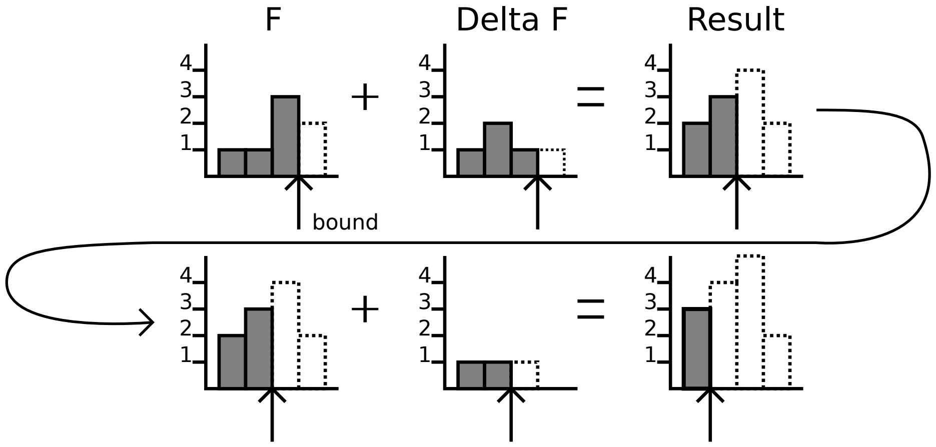

Fig. 2. Histogram pruning.

is greater than or equal to our desired node count, the maximum f value in the histogram can be fixed as an upper bound; selecting a greater f value can only give more nodes than desired. As the simulation proceeds and the weight in our accumulation histogram increases, we re-estimate the f value that gives our desired count. This upper bound will continue to decrease and the simulation will estimate fewer and fewer new nodes within the bound at each depth. When the expected number of new nodes is smaller than some ϵ the simulation can stop. In our experiments we use $\epsilon = 10^{-3}$. Additionally, because the d value of a node can never be negative, we can prune all nodes that would be generated with $d < 0$[2].

Example 1. Figure 2 shows a graphical example of histogram pruning. In this example, the desired number of nodes is 3. In the top row, the left-most histogram shows an accumulation histogram with a total weight of 7 (1 from each of the left two bars, 3 from the 3rd bar and 2 from the right-most bar). An arrow is drawn showing that the pruning bound is just after the 3rd bar; this is the first bar at which the total weight (aggregating from the left) surpasses the desired amount. The right-most histogram in this row shows the result of adding new nodes to this accumulation. Because the height of each bar has increased, the pruning bound has shifted to the left: only the first two bars are required to get our desired count. The next row shows a subsequent addition to this accumulation. As the bound moves further and further to the left, the total non-pruned weight in each of the histograms being added to the accumulation (the distribution of new nodes introduced at the current depth) will continue to decrease. When the count in one of these histograms is finally less than ϵ, the simulation will stop.

Figure 3 shows the pseudo-code for the procedure that uses the Δf distribution to estimate the f distribution. The entry point is the recursive function SIMULATE, which has the following parameters: the cost bound, desired number

[2] This could also be used to estimate the solution cost.

```
SIMULATE($bound, desired, depth, accum, nodes$)
 1. $nodes' = $ SIMEXPAND($depth, nodes$)
 2. $accum' = add(accum, nodes - nodes')$
 3. $bound' = find\_bound(accum', bound, desired)$
 4. if $weight\_left\_of(bound', nodes - nodes') > \epsilon$
 5.   $depth' = depth + 1$
 6.   SIMULATE($bound', desired, depth', accum', nodes'$)
 7. else return $accum'$

SIMEXPAND($depth, nodes$)
 8. $nodes' = $ new 2d histogram array
 9. for each $t$ and $d$ with $weight(nodes[t, d]) > 0$ do
10.   $fs = nodes[t, d]$
11.   SIMGEN($depth, t, d, fs, nodes'$)
12. return $nodes'$

SIMGEN($depth, t, d, fs, nodes'$)
13. for each type $t'$ and $\Delta d$
14.   $\Delta fs = delta\_f\_model[t', \Delta d, d, t]$
15.   if $weight(\Delta fs) > 0$ then
16.     $d' = d + \Delta d$
17.     $fs' = convolve(fs, \Delta fs)$
18.     $nodes'[t', d'] = add(nodes'[t', d'], fs')$
19. done
```

Fig. 3. Pseudo code for the simulation procedure used to estimate the f distribution.

of nodes, the current depth, a histogram that contains the accumulated distribution of f values so far and a 2-dimensional array of histograms that stores the conditional distribution of f values among the nodes at the current depth. SIMULATE begins by simulating the expansion of the nodes at the current depth (line 1). The result of this is the conditional distribution of f values for the nodes generated as offspring at the next depth. These f values are accumulated into a histogram of all f values seen by the simulation thus far (line 2). An upper bound is determined (line 3), and if more than ϵ new nodes are expected to be in the next depth then the simulation continues recursively (lines 4–6), otherwise the accumulation of all f values is returned as the final result.

The SIM-EXPAND function is used to build the conditional distribution of the f values for the offspring of the nodes at the current simulation depth. For each node type t and distance estimate d for which there exist nodes at the current depth, the SIM-GEN function is called to estimate the conditional f distribution of their offspring (lines 9–11). SIM-GEN uses the Δf distribution (line 14) to compute the number of nodes with each f value generated from parents with the specified combination of type and distance-estimate. Because this distribution is over Δf, t and Δd, we have all of the information that is needed to construct the conditional f distribution for the offspring (lines 16–18).

We have shown how to learn Δf, and use it to estimate the f distribution in a search tree. Now we turn to how an incremental model can be used to predict and control the performance of an IDA* search.

4 IDA*$_{IM}$

As we will see in Section 5.1, the incremental model can be used off-line for such tasks as distinguishing the more difficult of two search problems, a task that can be useful for automatically finding good heuristics [7]. One of the main features of the incremental model, however, is its ability to learn on-line, during search. In this section, we introduce IDA*$_{IM}$, a variant of IDA* that uses the incremental model to control how it increases bounds between iterations.

Like IDA* and IDA*$_{CR}$, IDA*$_{IM}$ uses a cost-bounded depth-first search. During each iteration, the Δf portion of an incremental model is learned as described in Section 3.1. When an iteration completes without finding a goal, the simulation procedure from Section 3.2 estimates a new bound that is expected to yield twice the number of expansions as the previous iteration. Given a good prediction, IDA*$_{IM}$ will increase the bound enough so that it does not degenerate into the $\mathcal{O}(n^2)$ worst-case of IDA*, and by a small enough amount that it does not greatly over-shoot the optimal cost and perform an extremely large final iteration.

With this approach, the goal may be found on an iteration where IDA*$_{IM}$ used a bound greater than the optimal cost. Like DFS* and IDA*$_{CR}$, IDA*$_{IM}$ uses branch-and-bound to complete its final iteration, expanding all nodes with f less than the current incumbent solution cost. If a cheaper goal is found, it becomes the new incumbent, and when all nodes with f less that the incumbent cost have been expanded, the search terminates with the optimal solution.

4.1 Implementation Details

Due to limited precision in the histograms and inaccuracies in the model, we found that IDA*$_{IM}$ occasionally estimates a bound for the subsequent iteration that is too low, i.e., the same or smaller than the previous bound. This tends to happen either very early in the search when the model has been trained on very few expansions or very late in the search when the histogram's precision becomes overly restrictive. In the latter case, the histogram size can be increased to increase its accuracy. In the former case, we track the minimum out-of-bound f value, just as IDA*, and use it as the minimum value for the next bound, ensuring that the search continues to progress.

Each iteration of IDA* search will expand a superset of the nodes expanded during the previous iteration. Instead of learning a new Δf model from scratch on each depth-first search, we learn one model and simply update it whenever a new node is encountered that was not generated on a previous iteration. We accomplish this by tracking the bound used in the previous iteration and only updating the model when expanding a node that would have been pruned by the previous bound. Additionally, the search spaces for many domains form graphs instead of trees. In domains with many cycles, it is beneficial to perform cycle checking—backtracking when the current node is one of the ancestors along the current path. In domains where this optimization is appropriate, our implementation uses a hash table of all of the nodes along the current path to locate and prune cycles. In order for our model to take this extra pruning into account, we only train the model on the successors of a node that do not create cycles.

As an iterative deepening search progresses, some of the shallower depths become *completely expanded*: no nodes are pruned at that depth or any shallower depth. All of the children of nodes in a completely expanded depth are *completely generated*. When learning the Δf distribution on-line, our incremental model has the exact *depth*, d and f values for all of the layers that have been completely generated. To speedup computation and improve accuracy, we "warm start" the simulation by initializing it with the perfect information for completed layers and begin it at the deepest completed depth instead of beginning at the root node. In some domains, such as the sliding tile domain, there are very few completed depth layers and warm starting has little effect. In other domains, such as the vacuum maze domains described in Section 5.2, many depth layers are completed during search and warm starting is extremely beneficial.

4.2 Theoretical Evaluation

In this section, we evaluate the theoretical properties of IDA*$_{IM}$.

Theorem 1. *IDA*$_{IM}$ is sound.*

Proof. IDA*$_{IM}$ never returns a solution that did not pass the goal test, therefore it will never return a non-solution. □

Theorem 2. *Given an admissible heuristic, IDA*$_{IM}$ is complete: if a solution exists then IDA*$_{IM}$ will find it.*

Proof. Suppose that there is a solution, it must have some cost, say c. IDA*$_{IM}$ always increases the cost bound between iterations, because it sets each subsequent bound to no less than the minimum out-of-bound f value. So, c will eventually be within the bound, and since the heuristic never overestimates, every node on the path to the goal will also be within the bound and the goal will be expanded. □

To prove the optimality of IDA*$_{IM}$, we use two helpful lemmata.

Lemma 1. *There always exists a deepest node along each optimal path to the goal that has been generated via an optimal path—it has an optimal g value.*

Proof. The proof is by induction on node expansions. To begin, the root is the first node along all optimal paths and is surely generated. At any point in the search, we have a deepest node expanded on each optimal path by the inductive hypothesis, and each of its successors is either: 1) not on an optimal path to a goal, 2) on an optimal path to a goal but has been generated via a suboptimal path or 3) is on an optimal path to some goal and was generated via an optimal path. In all three cases either the previous deepest node remains the deepest (1 and 2) or the newly generated node becomes the new deepest node on the path (3). Finally, the search never continues down a path after it expands a goal node. □

Lemma 2. *With a consistent heuristic (or one made consistent along each path from the root using pathmax) an iteration of IDA*$_{IM}$ expands all nodes with f less than or equal to the current bound.*

Proof. For sake of contradiction, suppose that IDA*$_{IM}$ completes an iteration with bound b but does not expand a node n with $f(n) \leq b$. In order for n not to be expanded it must have an ancestor m for which $f(m) > b$. However, $b \geq f(n) \geq f(m)$ since consistency implies that f strictly increases along each path [14]. □

Theorem 3. *Given an admissible heuristic, IDA*$_{IM}$ is optimal: if a solutions exists then IDA*$_{IM}$ will find the cheapest solution.*

Proof. For the sake of contradiction, suppose that IDA*$_{IM}$ returns a suboptimal solution sol. Heuristic admissibility implies $h(\cdot) = 0$ for all goal nodes, so by the suboptimality of sol, $g(opt) < g(sol)$, where opt is an optimal solution. In order for sol to have been expanded, the final iteration of IDA*$_{IM}$ must have used some cost bound $b \geq g(sol)$. Due to Lemma 1, there must be some node, say p, that is the deepest node along an optimal path to opt that was expanded along an optimal path. When p was expanded along its optimal path, the next node on this optimal path, q, must have been generated but not expanded. At this time, $f(q) \leq f(opt) = g(opt) < g(sol) \leq b$ due to heuristic consistency, and so q was within the bound and thus also expanded (Lemma 2): a contradiction. □

Lastly, we prove the correctness of the incremental model in an idealized setting.

Theorem 4. *Given an accurate Δf model, the admissible heuristic $h(\cdot) = 0$, and a uniform tree, i.e., a tree with a uniform branching factor and edge costs, additive convolution produces the correct f distribution for the next depth of the tree.*

Proof. Given a uniform tree, let $N(d)$ be the number of nodes and $\langle n_0, ..., n_{N(d)-1}\rangle$ be the nodes at depth d in the tree. Since the tree is uniform, each node $n_i, 0 \leq i < N(d)$ has children $\langle n'_{i,0}, ..., n'_{i,b-1}\rangle$ with costs c_j for $0 \leq j < b$ that are only dependent on the edge. Let $[p(\cdot)]$ be an indicator that has the value 1 when the predicate $p(\cdot)$ is true and 0 when $p(\cdot)$ is false [4], and let $delta_d(\cdot)$ be the Δf distribution for level d, i.e., it gives the number of successors of a node at level d with the given change in f. Since $h(\cdot) = 0$, $f(n'_{i,j}) - f(n_i) = c_j, 0 \leq i < N(d), 0 \leq j < b$. So, the number of nodes with f value of x is given by the equation:

$$\begin{aligned}
count_{d+1}(x) &= \sum_{i=0}^{N(d)-1} \sum_{j=0}^{b-1} [f(n_i) = x - c_j] \\
&= \sum_{j=0}^{b-1} \sum_{i=0}^{N(d)-1} [f(n_i) = x - c_j] \\
&= \sum_{j=0}^{b-1} count_d(x - c_j) \\
&= \sum_{\Delta f} \sum_{j=0}^{b-1} [c_j = \Delta f] \cdot count_d(x - \Delta f) \\
&= \sum_{\Delta f} delta_d(\Delta f) \cdot count_d(x - \Delta f)
\end{aligned}$$

This last formula is exactly the one computed by convolving the Δf and f distributions at depth d (c.f. Eqn. 1). □

We have presented a new variant of IDA* with attractive properties including the ability to make correct predictions in some ideal situations. We do not, however, have any guarantees about the accuracy of the model in the case of non-uniform search spaces; determining its usefulness in these cases is a matter for empirical study. In the next section, we assess the performance of our new algorithm in practice, and we show empirically that the model can give accurate predictions in a variety of domains.

5 Empirical Evaluation

In the following sections we show an empirical study of our new model and some of the related previous approaches. We begin by evaluating the accuracy of the incremental model when trained off-line. We then show the accuracy of the incremental model when used on-line to control an IDA* search.

5.1 Off-line Learning

We evaluate the quality of the predictions given by the incremental model when using off-line training by comparing the predictions of the model with the true node expansion counts. For each problem instance, the optimal solution cost is used as the cost bound. Because both CDP and the incremental model estimate all of the nodes within a cost bound, the true values are computed by running a full depth-first search of the tree bounded by the optimal solution cost. This search is equivalent to the final iteration of IDA*, assuming that the algorithm finds the goal node after having expanded all other nodes that fall within the cost bound.

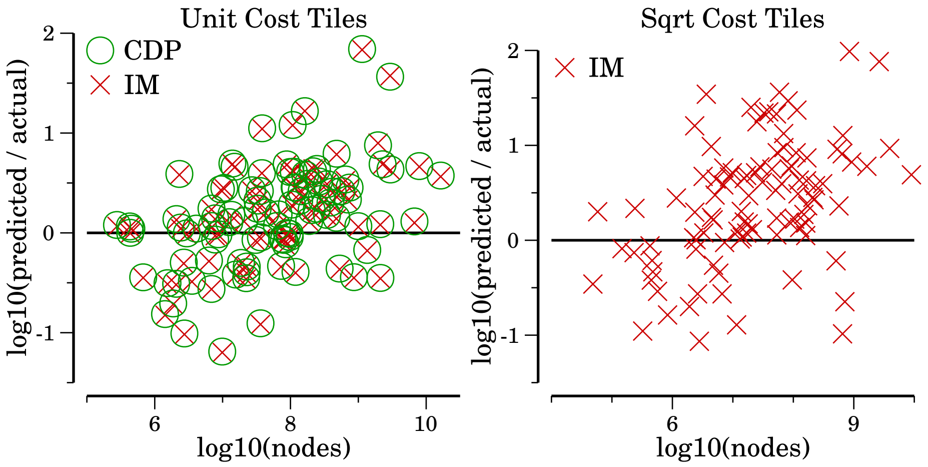

Fig. 4. Off-line training accuracy when predicting node expansions.

Estimation Accuracy. We trained both CDP [23] and an incremental model off-line on ten billion random 15-puzzle states using the Manhattan distance heuristic. We then compared the predictions given by each model to the true number of nodes within the optimal-solution-cost bound for each of the standard 100 15-puzzle instances due to Korf [8]. The left plot of Fig. 4 shows the results of this experiment. The x axis is on a log scale; it shows the actual number of nodes within the cost bound. The y axis is also on a log scale; it shows the ratio of the estimated number of nodes to the actual number of nodes, we call this metric the *estimation factor*. The closer that the estimation factor is to one (recall that $\log_{10} 1 = 0$) the more accurate the estimation was. The median estimation factor for the incremental model was 1.435 and the median estimation factor for CDP was 1.465 on this set of instances. From the plot we can see that, on each instance, the incremental model gave estimations that were nearly equivalent to those given by CDP, the current state-of-the-art predictor for this domain.

To demonstrate our incremental model's ability to make predictions in domains with real-valued edge costs and with real-valued heuristic estimates, we created a modified version of the 15-puzzle where each move costs the square root of the tile number that is being moved. We call this problem the square root tiles puzzle and for the heuristic we use a modified version of the Manhattan distance heuristic that takes into account the cost of each individual tile.

As presented by Zahavi et al. [23], CDP is not able to make predictions on this domain because of the real-valued heuristic estimates. The right plot in Fig. 4 shows the estimation factor for the predictions given by the incremental model trained off-line on fifty billion random square root tiles states. The same 100 puzzle states were used. Again, both axes are on a log scale. The median estimation factor on this set of puzzles was 2.807.

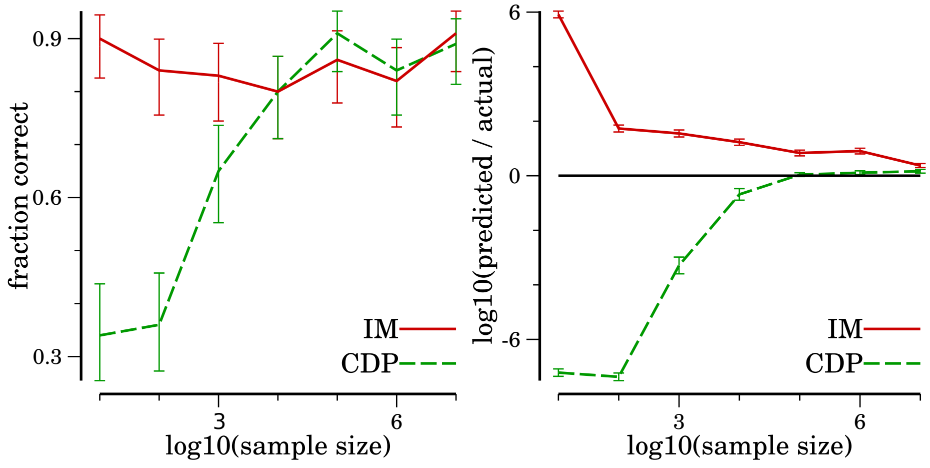

Fig. 5. Off-line training accuracy when predicting the harder instance.

Small Sample Sizes. Haslum et al. [7] use a technique loosely based on the KRE formula to select between different heuristics for domain independent planning. When given a choice between two heuristic lower bound functions, we would like to select the heuristic that will expand fewer nodes. Using KRE (or CDP) to estimate node expansions requires a very large off-line sample of the heuristic distribution to achieve accurate predictions, which is not achievable in applications such as Haslum et al.'s. Since the incremental model uses Δ values and a back-off model, however, it is able to make useful predictions with very little training data. To demonstrate this, we created 100 random pairs of instances from Korf's set of 15-puzzles. We used both CDP and the incremental model to estimate the number of expansions required by each instance when given its optimal solution cost. We rated the performance of each model based on the fraction of pairs for which it was able to correctly determine the more difficult of the two instances.

The left plot in Fig. 5 shows the fraction of pairs that were ordered correctly by each model for various sample sizes. Error bars represent 95% confidence intervals on the mean. We can see from this plot that the incremental model was able to achieve much higher accuracy when ordering the instances with as few as ten training samples. CDP required 10,000 training samples or more to achieve comparable accuracy. The right plot in this figure shows the $\log_{10}$ estimation factor of the estimates made by each model. While CDP achieved higher quality estimates when given 10,000 or more training instances, the incremental model was able to make much more accurate predictions when trained on only 10, 100 or 1,000 samples.

5.2 On-line Learning

In this section, we evaluate the incremental model when trained and used on-line during an IDA* search. As described in Section 4, the IDA$*_{IM}$ algorithm sets

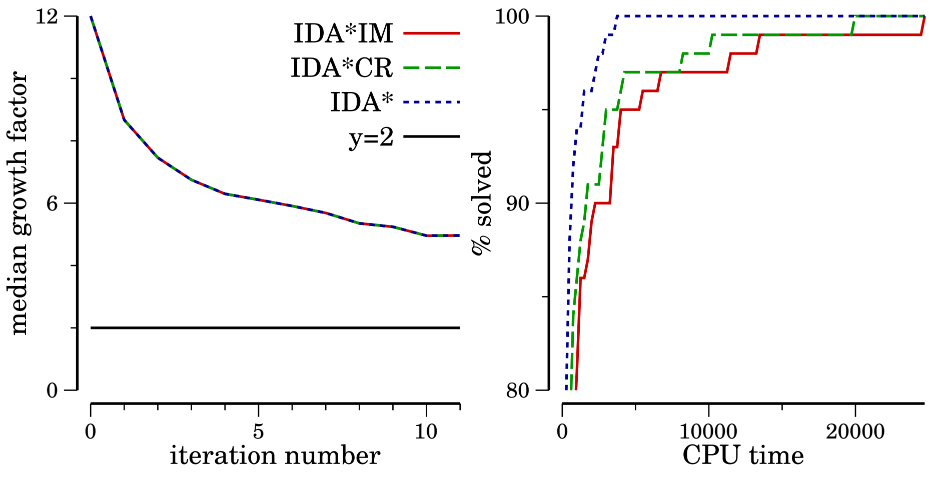

Fig. 6. Unit tiles: growth rates and number of instances solved.

the bound for the next iteration by consulting the incremental model to find a bound that is predicted to double the number of node expansions from that of the previous iteration. As we will see, because the model is trained on the exact instance for which it will be predicting, the estimations tend to be more accurate than the off-line estimations, even with a much smaller training set. In the following subsections, we evaluate the incremental model by comparing IDA*$_{IM}$ to the original IDA*[8] and IDA*$_{CR}$[18].

Sliding Tiles. The unit-cost sliding tiles puzzle is a domain where standard IDA* search works very well. The minimum cost increase between iterations is two and this leads to a geometric increase in the number of nodes between subsequent iterations.

The left panel of Fig. 6 shows the median *growth factor*, the relative size of one iteration compared to the next, on the y axis, for IDA* , IDA*$_{CR}$ and IDA*$_{IM}$. Ideally, all algorithms would have a median growth factor of two. All three of the lines for the algorithms are drawn directly on top of one another in this plot. While both IDA*$_{CR}$ and IDA*$_{IM}$ attempted to double the work done by subsequent iterations, all algorithms still achieved no less than 5x growth. This is because, due to the coarse granularity of f values in this domain, no threshold can actually achieve the target growth factor. However, the median estimation factor of the incremental model over all iterations in all instances was 1.029. This is very close to the optimal estimation factor of one. So, while granularity of f values made doubling impossible, the incremental model still predicted the amount of work with great accuracy. The right panel shows the percentage of instances solved within the time given on the x axis. Because IDA*$_{IM}$ and IDA*$_{CR}$ must use branch-and-bound on the final iteration of search they are unable to outperform IDA* in this domain. It should also be noted that, because IDA*$_{IM}$ and IDA*$_{CR}$ expand the exact same number of nodes on these

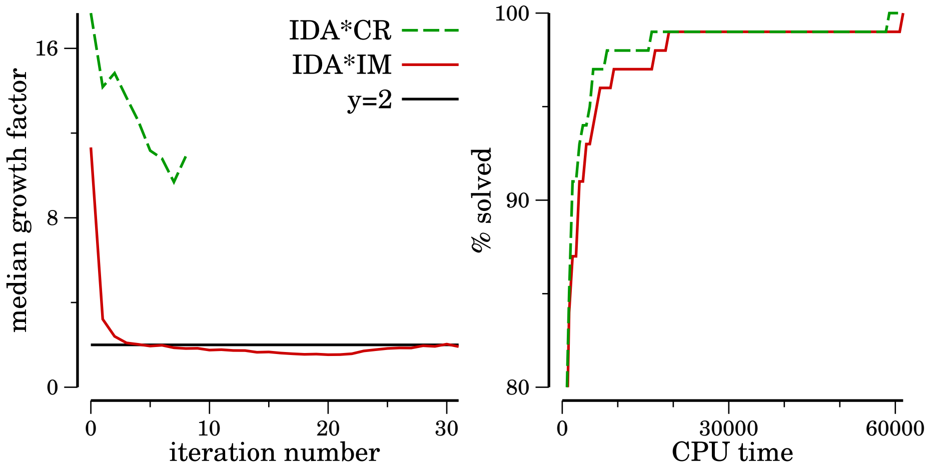

Fig. 7. Square root tiles: growth rates and number of instances solved.

instances (they use the same bounds for all iterations, and both use branch-and-bound on the final iteration), the difference in their performance, as seen in this plot, also shows the overhead incurred in learning the incremental model.

Square Root Tiles. While IDA* works well on the classic sliding tile puzzle, a trivial modification exposes its fragility: changing the edge costs. We examined the square root cost variant of the sliding tiles. This domain has many distinct f values, so when IDA* increases the bound to the smallest out-of-bound f value, it will visit a very small number of new nodes with the same f in the next iteration. We do not plot the results for IDA* on this domain because it gave extremely poor performance. IDA* was unable to solve any instances with a one hour timeout and at least one instance requires more than a week to solve.

Figure 7 presents the results for IDA*$_{IM}$ and IDA*$_{CR}$. Even with the branch-and-bound requirement, IDA*$_{IM}$ and IDA*$_{CR}$ easily outperform IDA* by increasing the bound more liberally between iterations. While IDA*$_{CR}$ gave slightly better performance with respect to CPU time, its model was not able to provide very accurate predictions. The growth factor between iterations for IDA*$_{CR}$ was no smaller than eight times the size of the previous iteration when the goal was to double. The incremental model, however, was able to keep the growth factor very close to doubling. The median estimation factor was 0.871 for the incremental model which is closer to the optimal estimation factor of one than when the model was trained off-line. We conjecture that the model was able to learn information specific to the instance for which it was predicting.

One reason why IDA*$_{CR}$ was able to achieve competitive performance in this domain is because, by increasing the bound very quickly, it was able to skip many iterations of search that IDA*$_{IM}$ performed. IDA*$_{CR}$ performed no more than 10 iterations on any instance in this set whereas IDA*$_{IM}$ performed up to 33 iterations on a single instance. Although the rapid bound increase was

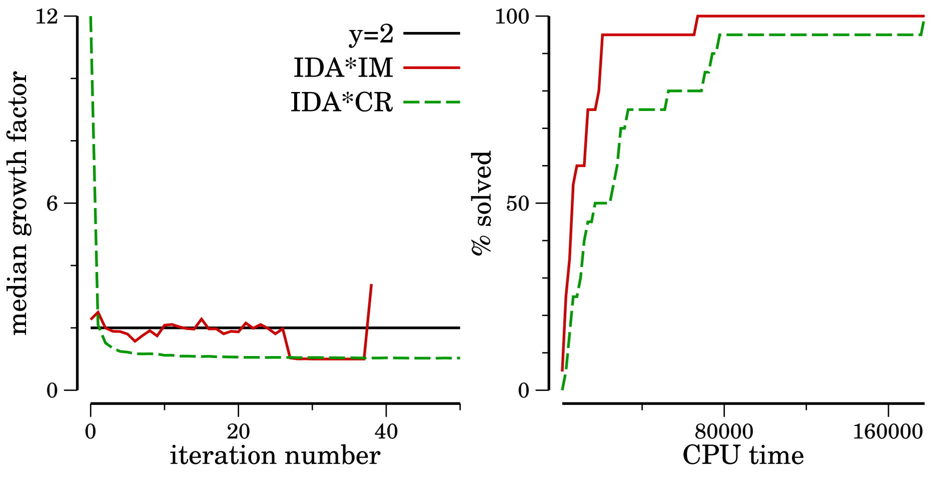

Fig. 8. Vacuum maze: growth rates and number of instances solved.

beneficial in the square root tiles domain, in a subsequent section we will see that increasing the bound too quickly can severely hinder performance.

Vacuum Maze. The objective of the vacuum maze domain is to navigate a robot through a maze in order to vacuum up spots of dirt. In our experiments, we used 20 instances of 500x500 mazes that were built with a depth-first search. Long hallways with no branching were then collapsed into single edges with a cost equivalent to the hallway length. Each maze contained 10 pieces of dirt and any state in which all dirt had been vacuumed was a goal. The median number of states per instance was 56 million and the median optimal solution cost was $28,927$. The heuristic was the size of the minimum spanning tree of the locations of the dirt and vacuum. The *pathmax* procedure [13] was used to make the f values non-decreasing along a path.

Figure 8 shows the median growth factor and number of instances solved by each algorithm for a given amount of time. Again, IDA* is not shown due to its very poor performance in this domain. Because there are many dead ends in each maze, the branching factor in this domain is very close to one. The model used by IDA*$_{CR}$ gave very inaccurate predictions and the algorithm often increased the bound by too small of an increment between iterations. IDA*$_{CR}$ performed up to 386 iterations on a single instance. With the exception of a dip near iterations 28–38, the incremental model was able to accurately find a bound that doubled the amount of work between iterations. The dip in the growth factors may be attributed to histogram inaccuracy on the later iterations of the search. The median estimation factor of the incremental model was 0.968, which is very close to the perfect factor of one. Because of the poor predictions given by the IDA*$_{CR}$ model, it was not able to solve instances as quickly as IDA*$_{IM}$ on this domain.

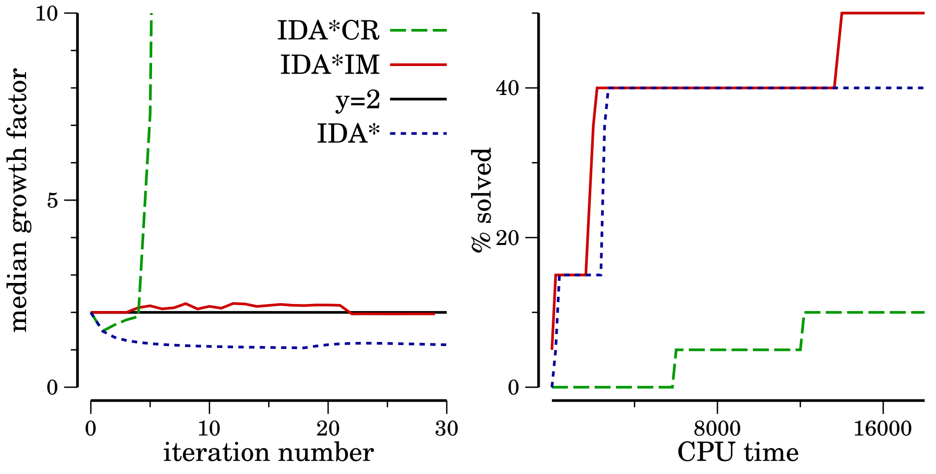

Fig. 9. Uniform Tree: growth rates and number of instances solved.

While our results demonstrate that the incremental model gave very accurate predictions in the vacuum maze domain, it should be noted that, due to the small branching factor, iterative searches are not ideal for this domain. A simple implementation of Frontier A* [10] (an A*-like search that elides the use of a closed list, and performs well in domains with low branching factors) was able to solve each instance in this set in no more than 1,887 CPU seconds.

Uniform Trees. We also designed a simple synthetic domain that illustrates the brittleness of IDA*$_{CR}$. We created a set of trees with 3-way branching where each node has outgoing edges of cost 1, 20 and 100. The goal node lies at a depth of 19 along a random path that is a combination of 1- and 20-cost edge and the heuristic $h = 0$ for all nodes. We have found that the model used by IDA*$_{CR}$ will often increase the bound extremely quickly due to the large 100-cost branches. Because of the extremely large searches created by IDA*$_{CR}$, we use a five hour time limit in this domain.

Figure 9 shows the growth factors and number of instances solved in a given amount of time for IDA*$_{IM}$, IDA*$_{CR}$ and IDA*. Again, the incremental model was able to achieve very accurate predictions with a median estimation factor of 0.978. IDA*$_{IM}$ was able to solve ten of twenty instances and IDA* solved eight within the time limit. IDA*$_{IM}$ solved every instance in less time than IDA*. IDA*$_{CR}$ was unable to solve more than two instances within the time limit. It grew the bounds in between iterations extremely quickly, as can be seen in the growth factor plot on the left in Fig. 9.

Although IDA* tended to have reasonable CPU time performance in this domain, its growth factors were very close to one. The only reason that IDA* achieved reasonable performance is because expansions in this synthetic tree domain required virtually no computation. This would not be observed in a domain where expansion required any reasonable computation.

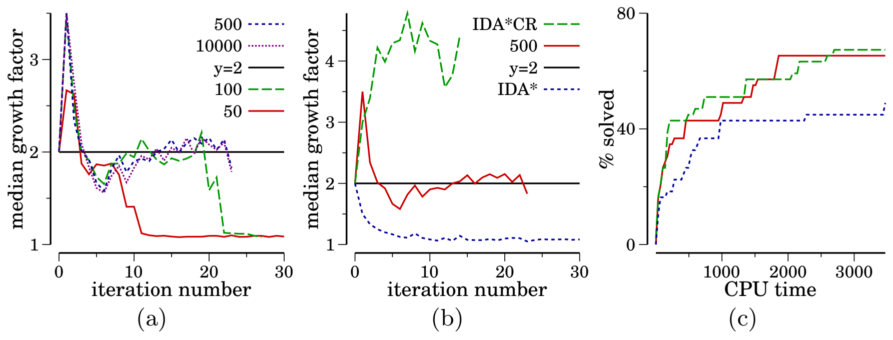

Fig. 10. Different histogram sizes (a), IDA*, IDA*$_{CR}$, and IDA*$_{IM}$ growth rates (b) and number of instances solved (c) on the pancake problem.

5.3 Effect of Histogram Size

Since the incremental model uses fixed-size histograms to represent distributions, it is useful to know the effect of the histogram size on the performance of the model. In our previous experiments we used a histogram size of 100. One limitation of the incremental model and thus IDA*$_{IM}$ is that its performance may rely on choosing a good histogram size. In this section we look at a domain for which the histogram size of 100 is too small.

In the *pancake problem*, a chef is given a stack of different size pancakes and a spatula. Unfortunately, the pancakes are all out of order, and to make for a beautiful presentation the chef must repeatedly stick his spatula into the stack and flip some of the top pancakes until the entire stack is sorted. At its core, the pancake problem is a permutation puzzle, where the goal is to sort an n-permutation using a sequence of prefix reversals. This domain differs from those of the previous sections because it has a fairly large branching factor; an n-pancake problem has a branching factor of $n-2$ if you disallow flipping the very top pancake (which has no effect on the ordering) and reversing the previous flip.

We ran IDA*$_{IM}$ on a non-unit-cost variant of the pancake problem where each flip costs the sum of the numbers of the pancakes being flipped, simulating the idea that flipping more or larger pancakes costs more than flipping fewer or smaller pancakes. This variant of the problem is challenging for standard IDA* as there are many distinct f values. In this experiment, we used 50 random 12-pancake instances with a 7-pancake pattern database [3] for the heuristic. Our initial results showed that IDA*$_{IM}$ gave poor performance on this domain; as the search progressed, the estimations from the incremental model became very inaccurate. We hypothesized that the large branching factor in the pancake problem was adding too many values to the fixed-size histograms leading to inaccuracy in the estimates. To test our hypothesis we ran IDA*$_{IM}$ with different histogram sizes on the pancake puzzle. The growth rates for histogram sizes of 50, 100, 500, and 10,000 are shown in Fig. 10a. With the histogram size of 50 the accuracy of the predictions from the model, and thus the growth rates of IDA*$_{IM}$ begun to significantly decrease just before 10 iterations. When the histogram size was increased to 100 the prediction accuracy begun to decrease at around 20 iterations. When we increased the histogram size to 500, the drop off in accuracy went away and the growth factor remained around the correct value of 2 until the very final iterations of search. Finally, we increased the histogram size much further to 10,000, the performance was very similar to the performance of 500, including the slight estimation inaccuracy during the final few iterations.

Figures 10b and 10c compare IDA*, IDA*$_{CR}$, and IDA*$_{IM}$ with a histogram size of 500 on the pancake problem. As before we can see that IDA* performs poorly as it requires many iterations, each of which is very similar to the previous. IDA*$_{CR}$, once again grows its iterations quite quickly, about quadrupling each subsequent iteration. The incremental model, using sufficiently large histograms, is rather accurate at doubling the size of its iterations, however, its performance is similar to IDA*$_{CR}$ with respect to the number of instances solved in the time limit. We suspect that IDA*$_{IM}$ does not outperform IDA*$_{CR}$ in this domain due to its increased overhead in maintaining larger histograms.

5.4 Summary

When trained off-line, the incremental model was able to make predictions on the 15-puzzle domain that were nearly indistinguishable from CDP, the current state-of-the art. In addition, the incremental model was able to estimate the number of node expansions on a real-valued variant of the sliding tiles puzzle where each move costs the square root of the tile number being moved. When presented with pairs of 15-puzzle instances, the incremental model trained with 10 samples was more accurately able to predict which instance would require fewer expansions than CDP when trained with 10,000 samples.

The incremental model made very accurate predictions across all domains when trained on-line and when used to control the bounds for IDA*, our model made for a robust search. While each of the alternative approaches occasionally gave extremely poor performance, IDA* controlled by the incremental model achieved the best performance of the IDA* searches in the vacuum maze and uniform tree domains and was competitive with the best search algorithms for both of the sliding tiles domains and the pancake puzzle. This provides an example of how a flexible tree model can be used in practice.

6 Discussion

In search spaces with small branching factors such as the vacuum maze domain, the back-off model seems to have a greater impact on the accuracy of predictions than in search spaces with larger branching factor such as the sliding tiles domains. Because the branching factor in the vacuum maze domain is small, however, the simulation must extrapolate out to great depths (many of which the model has not been trained on) to accumulate the desired number of expansions. The simple back-off model used here merely ignored depth. While this tended to give accurate predictions for the vacuum maze domain, a different model may be required for other domains.

We have looked at using an incremental model for predicting the number of nodes that will be expanded within a given cost bound. Another possible use for the incremental model is estimating the optimal solution cost for a problem. Since the distance-to-go estimate d is only equal to zero for goal nodes, the simulation procedure described in Section 3 may be used to determine the smallest f value among all of the nodes with $d = 0$. This f value is an estimate of the optimal solution cost. This would extend the results presented by Lelis, Stern, and Jabbari [11] to on-line learning and domains with real valued costs.

There has been much successful work in the area of exploiting instance-specific information for automatic algorithm selection and configuration [22, 12, 1]. In agreement with these results, we believe that instance-specific information is what allows the incremental model to make accurate predictions when trained online (cf Section 5.2, where our results show that the model was more accurate when trained online on each instance than when trained offline). Reactive and autonomous search [2, 5], in which optimization algorithms attempt to tune themselves online, have been show to be quite effective for constraint programming and combinatorial optimization. IDA*$_{IM}$ demonstrates that on-line learning can be used to control search for shortest-path problems too.

7 Conclusion

In this paper, we presented a new incremental model for predicting the distribution of solution cost estimates in a search tree and hence the number of nodes that bounded depth-first search will visit. Our new model is comparable to state-of-the-art methods in domains where those methods apply. The three main advantages of our new model are that it works naturally in domains with real-valued heuristic estimates, it is accurate with few training samples, and it can be trained on-line. We demonstrated that training the model on-line can lead to more accurate predictions. Additionally, we have shown that the incremental model can be used to control an IDA* search, giving a robust algorithm, IDA*$_{IM}$. Given the prevalence of real-valued costs in real-world problems, on-line incremental models are an important step in broadening the applicability of iterative deepening search.

8 Acknowledgements

We gratefully acknowledge support from NSF (grant IIS-0812141), the DARPA CSSG program (grant HR0011-09-1-0021), and the University of New Hampshire Dissertation Year Fellowship.

References

1. Arbelaez, A., Hamadi, Y., Sebag, M.: Continuous search in constraint programming. In: Twenty-second IEEE International Conference on Tools with Artificial Intelligence (ICTAI-10). vol. 1, pp. 53–60 (2010)
2. Battiti, R., Brunato, M., Mascia, F.: Reactive search and intelligent optimization, vol. 45. Springer (2008)
3. Culberson, J.C., Schaeffer, J.: Pattern databases. Computational Intelligence 14(3), 318–334 (1998)
4. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete Mathematics: A Foundation for Computer Science. Addison-Wesley (1998)
5. Hamadi, Y., Monfroy, E., Saubion, F.: What is autonomous search? Hybrid Optimization 45, 357–391 (2010)
6. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions of Systems Science and Cybernetics SSC-4(2), 100–107 (July 1968)
7. Haslum, P., Botea, A., Helmert, M., Bonte, B., Koenig, S.: Domain-independent construction of pattern database heuristics for cost-optimal planning. In: Proceedings of the Twenty-second Conference on Artificial Intelligence (AAAI-07) (Jul 2007)
8. Korf, R.E.: Iterative-deepening-A*: An optimal admissible tree search. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-85). pp. 1034–1036 (1985)
9. Korf, R.E., Reid, M., Edelkamp, S.: Time complexity of iterative-deepening-A*. Artificial Intelligence 129, 199–218 (2001)
10. Korf, R.E., Zhang, W., Thayer, I., Hohwald, H.: Frontier search. Journal of the ACM 52(5), 715–748 (2005)
11. Lelis, L., Stern, R., Jabbari Arfaee, S.: Predicting solution cost with conditional probabilities. In: Proceedings of the Fourth Annual Symposium on Combinatorial Search (SoCS-11) (2011)
12. Malitsky, Y.: Instance-Specific Algorithm Configuration. Ph.D. thesis, Brown University (2012)
13. Méro, L.: A heuristic search algorithm with modifiable estimate. Artificial Intelligence pp. 13–27 (1984)
14. Nilsson, N.J.: Principles of Artificial Intelligence. Tioga Publishing Co (1980)
15. Pearl, J.: Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley (1984)
16. Rose, K., Burns, E., Ruml, W.: Best-first search for bounded-depth trees. In: Proceedings of the Fourth Annual Symposium on Combinatorial Search (SoCS-11) (2011)
17. Ruml, W.: Adaptive Tree Search. Ph.D. thesis, Harvard University (May 2002)
18. Sarkar, U., Chakrabarti, P., Ghose, S., Sarkar, S.D.: Reducing reexpansions in iterative-deepening search by controlling cutoff bounds. Artificial Intelligence 50, 207–221 (1991)
19. Thayer, J., Ruml, W.: Using distance estimates in heuristic search. In: Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS-09) (2009)
20. Vempaty, N.R., Kumar, V., Korf, R.E.: Depth-first vs best-first search. In: Proceedings of AAAI-91. pp. 434–440 (1991)

21. Wah, B.W., Shang, Y.: Comparison and evaluation of a class of IDA* algorithms. International Journal on Artificial Intelligence Tools 3(4), 493–523 (October 1995)
22. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for SAT. Journal of Artificial Intelligence Research 32(1), 565–606 (2008)
23. Zahavi, U., Felner, A., Burch, N., Holte, R.C.: Predicting the performance of IDA* using conditional distributions. Journal of Artificial Intelligence Research 37, 41–83 (2010)

A Histograms

The incremental model makes use of fixed-size histograms to represent distributions over f and Δf values. To maintain as much accuracy as possible, our implementation uses two types of histograms: an exact *points* histogram for representing a small number of data points and an approximate *bins* histogram for representing a large number of data points. A points histogram is a list of the data points representing the distribution. Points histograms are completely accurate as they represents exactly every value in their distribution. When the number of data points added to the distribution reaches the histogram's size limit, it is converted to a bins histogram. A bins histogram is an array of fixed-width bins, where each entry is a floating point value that represents the amount of mass in the distribution for the range of values represented by the corresponding bin. A bins histogram is approximate but has the advantage of using a constant amount of memory to store and a constant amount of computation to manipulate. The size of the histogram represents the maximum number of points allowed before converting to bins and once converted to bins the size specifies the number of bins.

Our histogram implementation supports several operations: *add_mass* adds a point mass to the histogram; *add* returns a histogram representing the sum of a list of histograms; *convolve* returns a histogram that is the convolution two histograms as described in Section 3.2; *weight_left_of* returns the amount of mass in the histogram to the left of a given value; *total_weight* returns the total weight of the histogram; *normalize* normalizes the distribution such that the total weight is equal to a given value; and *prune_weight_right* prunes the mass in the histogram to the right of the given value.

When adding mass to a points histogram the new values are added as additional points. For a bins histogram, the additional mass is 'sprinkled' proportionally across the bins containing values for which mass is being added. If mass is added to a value that extends beyond the domain of a bins histogram then the bins of the histogram are recomputed by increasing the bin width by integral multiple of the current width. When the bin width is increased in this manner, adjacent bins are merged and the newly emptied bins are free to accommodate the new values. When convolving two histograms, the result is a bins histogram if either of the operands are a bins histogram, or if they are both points histograms but the domain of the result has more values than the histogram size.