

Predicting the Optimal Combination of Pattern Databases for Solving a Problem

Santiago Franco and Mike Barley

Computer Science Department
Auckland University

Abstract

Given a set of problems, a set of heuristics, and assuming that everything else is equal, it is never worse and usually better to solve each problem with the heuristic that is best for that problem than to use, for all problems, the heuristic that is best, on average, for the set of problems. Unfortunately, everything else is seldom equal. In particular, the cost of determining which heuristic is best, on average, is usually cheaper than determining for each problem, which is best. But, does this mean that when you include the costs of determining the best heuristic that the former approach (i.e. determining for each problem which is best for it) is always worse, on average, than the latter approach? This is not the case if the average cost of determining the best heuristic, for each problem, is lower than the average cost advantage of using it. However, how low can we bring this cost? This paper introduces two new data structures, minTrees and culprit counter lattices, that lower this cost and then we compare using this approach with these data structures against simply using a known effective compact pattern database variation. The experiments show that the key to success is finding cost-effective predictions of which heuristic will be best for a problem rather than precisely determining which is best. While our experiments do not provide conclusive proof that it is better on average to determine, for each problem, the best heuristic, it does provide evidence that suggests this can be the case.

Introduction

Problem-solvers often use heuristic search algorithms (e.g. A*(Hart, Nilsson, and Raphael 1968) and IDA*(Korf 1985) to find optimal solutions. These search algorithms use a cost function, F , to guide their search. The cost value for a node, $F(n)$, is the sum of the cost of the path from the root node to this node, $g(n)$, and an "underestimate"¹ of the minimal cost, $h(n)$, of any path from this node to its nearest goal node. $h(n)$ is usually called an heuristic evaluation function. Until recently, these heuristic evaluation functions have computed their values upon demand.(Culberson et al. 1994) introduced the idea of precomputing these values for a domain and storing them in a lookup table called a pattern database (PDB). Then during a search the heuristic function

need only lookup the cached h -value. The use of PDBs have enabled heuristic search algorithms to solve problems that were previously beyond their reach.

(Holte and Hernadvolygi 1999) showed that, in general, the larger the PDB used by a heuristic search algorithm, the less search that was needed to find a solution. Later, (Holte et al. 2006) showed that for a given amount of memory, it was usually better to divide up that memory among a number of smaller PDBs and use the max of their values. This result held even though the accuracy of the larger PDB was better than the accuracy of any one of the smaller PDBs. (Holte et al. 2006) also showed that using either too many or too few PDBs leads to suboptimal time performance. Given a set of PDBs, determining which combination of PDBs (out of the powerset of that PDB set) is best, is still an open problem.

There are two general approaches to tackling this problem. One is to determine the combination of PDBs, on average, best for a (usually uniform) distribution of problems in that domain. The other approach is to determine, for each specific problem, which combination is best. This paper explores that latter approach. The incentive for taking this approach is that (ignoring, for the moment, the cost of doing the determinations) this latter approach is never worse and is usually better than the former approach. The rest of this paper looks at how to determine the best combination of PDBs to use to solve a given problem.

This paper deals specifically with PDBs. However, they are simply one way of representing a heuristic. In the rest of this paper, we will simply use the term "heuristic", but this should be understood as referring to a PDB.

We assume there is no way to know *a priori* which combination of heuristics would be best for a given problem and we assume that it is pointless to determine this *a posteriori*. Thus we try to determine this *in situ*, i.e. use some of the initial problem-solving activity to gather enough data that a prediction can be made about which combination of heuristics will be best to use for solving the rest of the problem. Since the cost of the prediction is now part of the problem-solving costs, our system, called Reconfigurable-IDA*(RIDA*), needs to balance the cost of obtaining its prediction against the prediction's impact on the search costs. Currently, RIDA* uses a user-defined parameter, call the sample capping limit, which specifies how much data should be captured.

RIDA* uses a parameterised run-time formula to predict how long an iteration of IDA* will run on this problem for a given F-bound and a specific combination of heuristics. Some of the parameter values can be accurately determined *a priori*, while others can only be accurately be known *a posteriori*. RIDA* approximates these *a posteriori* values by analysing the data gathered during the first few iterations on the problem. This data is gathered simultaneously for all the heuristics.

We have tested RIDA* using the 15-puzzle domain and have adapted a combination of PDBs described in (Holte et al. 2006) to act as our benchmark against which we evaluated RIDA*'s performance.

This paper's main contribution is that it shows that it is possible for a search algorithm to determine (via an analysis of data gathered during the initial part of the search) which combination of heuristics should be used on the remaining portion of the search. In particular, this paper shows that the overhead of this determination (i.e. the data gathering and analysis) can be made low enough for the combined system, RIDA*, to be competitive against IDA* using our benchmark combination of heuristics.

This paper introduces two new data structures. One, called a minTree, allows RIDA* to cheaply gather the data on all the combinations of heuristics simultaneously. IDA* uses a value, called the F-bound (F), to determine which nodes to prune (i.e. not expand this iteration). MinTrees have a slightly different rule (than standard heuristic search trees) for using the F-bound when dealing with a combination of heuristics. In standard heuristic search trees (which we call maxTrees in this paper to distinguish them from minTrees), a node, n , is pruned if the maximum value returned by the heuristics (i.e. $h(n)$) is greater than value of $(F - g(n))$. In a minTree, a node is only pruned if the minimum value returned by the heuristics is greater than this difference $(F - g(n))$. This allows RIDA* to grow one search tree that simulates growing a separate search tree for every combination of heuristics. This reduces the overhead of gathering the data for every heuristic combination.

The other new data structure, called a culprit counter lattice, allows RIDA* to cheaply compute, for each heuristic combination, the size of the maxTrees that IDA* would generate for the sampled iterations on the given problem. A culprit heuristic is one whose value for a node, n , is less than $F - g(n)$. There is one culprit counter for every heuristic combination. Each culprit counter keeps track of every node expanded where the heuristics in the combination is exactly the set of culprit heuristics for that expanded node. The culprit counter lattice is updated while RIDA* is growing the minTree, and then after the data has been collected, it is used to compute the final maxTree sizes. The two main advantages of the culprit counter lattice are: only one counter is updated for each node expanded in the minTree and only the minimal number of additions are required to translate the culprit counter lattice into the final maxTree sizes for every heuristic combination.

Our paper is organised as follows: In the next section, we discuss our framework and related research. In the section following that, we describe RIDA*'s architecture and then in

the subsequent section we discuss our experiments to evaluate RIDA*. The final section summarises and suggests some future research

Related Research & Framework

In this section we discuss our research in the context of the existing literature. There are five main topics which we will discuss. Two of them are our main contributions, minTrees and culprit counters lattices, and one is an *ad hoc* mechanism.

When Run Time Formulas are the Appropriate Search Performance Measurement

The standard approach to comparing the performance of two search algorithms (including IDA*) has been to compare the number of nodes they expand before a solution is found. This is adequate when the cost of expanding a node is approximately the same for both algorithms. However, when the costs can be quite different then time need to be taken into account. If we are comparing the performance of two search algorithms running in very different environments (e.g. run on different hardware, implemented in different languages, compiled with different optimisation options, etc.), then it's not appropriate to compare the actual run times if it's possible that their relative performance might change radically if the environments change. Instead, algorithmic complexity formulas² (e.g. $O(n)$, etc.) are the appropriate measures. However, when the algorithms are being run in the same environment then it seems most appropriate to compare actual run times. This is the case when the system is attempting to select between combinations of heuristics. However, usually we won't know the actual run times. Instead, we need to rely on run-time formulas which allow us to predict the relative run-times of the two algorithms with respect to some variable (e.g. the optimal solution path length). These run-time formulas usually need to be parameterised to take into account how computationally expensive are the "primitive" statements and what are the initial conditions (e.g. in search problems, what is the initial state and what is the goal set).

In Situ Sampling

Given that there are two heuristics, h_1 and h_2 , if h_1 imposes a greater overhead per node but yields a greater pruning of the search space (either because it more greatly reduces the branching factor or the average depth) than h_2 , then there will likely be a brute force search space size where both heuristics will cause the search algorithm to spend roughly the same of amount of time to explore. This size is the crossover point for the two heuristics. H_2 will perform better on search spaces whose brute force sizes are smaller than the crossover point, while H_1 will perform better where the brute force sizes are larger.

Since we are only concerned in this paper with PDBs, the main overhead of creating a node will be its evaluation cost,

²Algorithmic complexity formulas attempt to capture the run-time complexity of algorithms at an abstract enough level that these environmental differences disappear.

and that cost will be directly proportional to the number of PDBs in the combination. Thus all the heuristic combinations with the same number of heuristics will have roughly the same overhead per node.

We assume that almost all of our heuristic combinations will exhibit different degrees of pruning of the search space and that the set of heuristic combinations can be partitioned into equivalence classes based on the number of heuristics in the combination. This means that there can be numerous crossover points and the size of the brute force space to be explored will affect which heuristic combination is best.

If we know the depth of the brute force search tree (and the domain's effective branching factor), then it is easy to calculate the size of that search tree. However, we seldom know *a priori* the length of the optimal solution path. Hence, we cannot predict *a priori* which heuristic combination will be best to use for the entire problem. However, since we are using IDA*, we can predict how large the brute force search tree will be for a given F-bound. Thus, if we can model how each heuristic combination affects the search space size, then we can predict which combination will be best to use on which iteration.

For many domains, heuristic performance depends on the problem being solved. For example there is no known obvious way to determine a priori which pattern database heuristic will do better on an N-puzzle instance. Korf described the following rule of thumb:

As a general rule, when partitioning the tiles, we want to group together tiles that are near each other in the goal state, since these tiles will interact the most with one another (Korf and Felner 2002, Section 3)

RIDA* does *in situ* heuristic selection which is the best option when *a priori* selection mechanism does not exist. *in situ* heuristic selection postpones the selection of heuristics until enough information has been gathered about all available heuristics to make a good choice but before the cost of acquiring this information outstrips future savings.

Searching with sampling is much more expensive than searching without sampling. This encourages stopping the sampling as early as possible. However, RIDA* needs to sample enough to adequately approximate the *a posteriori* values of the *in situ* variables so that RIDA*'s run-time predictions are accurate enough for RIDA* to decide which heuristic combination to use for the rest of the search.

Run-Time Formula

RIDA* uses a parameterised run-time formula to characterise search performance. The parameterised run-time formula allows RIDA* to model the search algorithm's actual run-time environment as well as the actual problem being solved. Because the different heuristic combinations have different time costs, their performance curves usually cross. The crossovers indicate where the impact of having better heuristic values overcomes the run-time penalty of looking at more heuristic values. RIDA* uses these crossover points to determine when it should shift from using one heuristic combination to using a different combination. We believe

the relative performance of the different heuristic combinations can vary considerably in different portions of the search space and consequently affect where the crossovers occur. This is one of the benefits of *in situ* sampling, RIDA* is deriving the values of its parameters from the actual part of the search space where it will be exploring. Similarly, changing the run-time environment (e.g. the compiler optimisation options, etc.) may also affect where the crossovers occur.

The run-time formula predicts the cost of using different heuristic combinations on future iterations. The cost of using a heuristic combination on an iteration is the number of nodes expected to be expanded for that iteration (when using that heuristic combination) multiplied by how expensive node expansions have been measured to be (when using that heuristic combination).

$$t(F) = N * t(\#h) \quad (1)$$

t(F) = Time to generate search space to F-bound;

N = # of generated nodes (from Eq3);

#h = # of available heuristics;

t(#h) = heuristic evaluation cost for #heuristics (table 1).

The number of nodes expanded in consecutive IDA* iterations grows exponentially and can be modeled in several ways. This paper does not claim to improve search modeling per se but uses instead existing models to choose a good heuristic combination for the problem instance being solved. We chose to represent the iterative growth of the IDA* search tree using the Heuristic Branching Factor (HBF) (Korf, Reid, and Edelkamp 2001). Korf et al defined the HBF as the constant growth factor between IDA* iterations once the problem depth is large enough. The HBF for a problem can vary significantly from iteration to iteration as the search space heuristic distribution changes from the initial state neighbourhood to the domain average.

$$HBF_i = \frac{N_i}{N_{i-1}} \quad (2)$$

N_{i-1} = Number of nodes generated previous iteration

N_i = Number of nodes generated current iteration

$$N_{i+1} = N_i \times HBF_i \quad (3)$$

N_{i+1} = Number of nodes generated next iteration

For alternative size prediction models please see Effective Depth reduction and heuristic distribution models (Korf, Reid, and Edelkamp 2001; Zahavi et al. 2008).

Eq (3) represents how we predict the size of the maxTree for future iterations. *N_i* and *HBF_i* parameters are calculated in situ. In order to calculate *HBF_i* we need to have grown at least two iterations for the current problem. So the earliest prediction RIDA* can make is for the third iteration.

We decided to use the HBF model, even though the heuristic distribution model of IDA* is more accurate than using the HBF (Zahavi et al. 2008), for two reasons. Firstly, keeping a heuristic distributions for each heuristic combination requires considerable more memory than just one

HBF per combination. As the number heuristic combinations grows exponentially, i.e. 2^h , and pattern databases get most of the available memory we decided the HBF model was more practical when selecting heuristic combinations. Secondly, as we do in situ sampling we believed the heuristic distribution obtained in such manner would be biased toward h values close to the initial state. Of course this bias effect would diminish with each passing iteration as the search space grows larger. But we want to do good predictions as soon as possible in order to keep most of the savings.

All existing size prediction models do well *a posteriori*, RIDA* challenge is to do a good prediction when we approximate *a posteriori* parameters with *in situ* sampling. Eq(3) makes good size predictions based on small sampling sizes but, as we have not explored all possible size prediction models, no claim is made that this size prediction model is the best option when predicting future iteration size from small samples.

MinTree

To minimize the costs of sampling different heuristic combinations we have created a data structure which we call a minTree. The minTree is the search tree composed of all nodes generated by all available heuristic combinations. Nodes in a minTree are assigned the minimum value of those returned by all the heuristics. MaxTrees will prune a node if any of the available heuristics prune the node and nodes are assigned the maximum value of those returned by all heuristics. For each iteration there are as many maxTrees as there are heuristic combinations. There is only one minTree for each iteration. All maxTrees for an iteration are subtrees of that iteration's minTree.

The minTree can be viewed as the union of all the heuristic combinations' maxTrees. It is more efficient than doing all possible maxTrees because any nodes common to two or more heuristic combinations will only be represented as one node in the minTree. Hence, the minTree of an iteration can be described as a lossless compression of all the maxTrees of that iteration put together.

MinTrees are never larger than the brute force search tree, but are usually substantially larger than any of the maxTrees for the same iteration. In order to achieve our objective of keeping the sampling costs lower than the possible savings we need to stop doing minTree sampling at least one iteration before the solution for the current problem instance is found. If the problem instance is solved while we are doing the minTree, then the resulting time performance will be much worse than the worst available heuristic combination.

The minTree also has the biggest overhead possible per node as all available heuristics are being used. But its overall computational effort is still significantly smaller than possible savings for most problems as long as the sampling stops before the solution distance. This is due to the exponential growth of the search effort in IDA*. IDA* is used on domains with exponential search growth, otherwise IDA* iterative nature would make it too expensive. MinTree savings are reported in the experiment section(table 3).

Culprit Counter Lattices

RIDA* uses culprit counters to efficiently calculate the size of all possible maxTrees out of an input heuristic set. RIDA* keeps one *culprit counter* for each heuristic combination. Any node expanded in the minTree updates one and only one of the culprit counters and its value is increased by the number of children generated. The culprit counter mapping function for each heuristic in the input set gives a value of 0 if the heuristic expands the node, 1 otherwise. For example culprit counter 000 would be associated to how many nodes were expanded by all three heuristics in the set at the same time. The mapping function is used to identify which culprit counter needs to get updated every time a node is expanded.

For our five heuristic set experiment this mean keeping a set of $2^h \rightarrow 2^5 = 32$ culprit counters starting at 00000 and finishing at 11111. The *final size counters* of any of the 32 maxTrees can be calculated as the addition of all culprit counters in which the heuristics that make up the maxTree generated nodes. Final size counters are calculated after the end of each iteration(Algorithm 1). For example to calculate the size of the maxTree resulting from heuristic combination 11100 (last two heuristics have been dropped) we would add up culprit counters: 00010, 00001, 00011 and 00000. These 4 culprit counters together represent nodes which only the fourth heuristic pruned, nodes who only the fifth heuristic pruned, nodes pruned only by the fourth & fifth heuristic and finally nodes generated by all heuristics.

Culprit Counters are efficient because if we have a large number of heuristic combinations(e.g. for 30 heuristics there are more than a billion heuristic combinations), we do not want to calculate which of the billion final size counters need to be updated every time a node is expanded in the minTree. This would not be a problem if instead of the minTree we were using Maxtrees. All nodes generated in a maxTree are assigned to the same heuristic combination so there is no need for culprit counters. But in the minTree the same node can be expanded by several heuristic combinations so we have an assignment problem. This is efficiently solved by using the culprit counters.

When a node is expanded in the minTree only a single culprit counter gets updated. We add the number of generated children to the corresponding culprit counter. This way we only have to do one addition every time an internal node in the minTree gets expanded. There is no extra bookkeeping for leaf nodes. As most nodes in trees are leaves this represents extra savings.

The culprit counters are kept in a lattice structure, where each node represents one of the heuristic combination subset out of the powerset of the set of heuristics. All edges between pairs of nodes mean that the set at one nodes is simply the union of the set at the other node and one of the other heuristics. The maximal node has the set containing all available heuristics and the minimal node has the empty set.

Capping

The purpose of using minTrees in RIDA* is to sample enough nodes for each heuristic combination maxTree so

that we can make good future size predictions with low sampling costs.

The worst heuristic combinations can create maxTrees whose size is orders of magnitude larger than their more informed counterparts for the same F-bound. Hence we have the problem of potentially investing much more sampling effort on the worst heuristics while making lower quality predictions, due to their low sampling frequency, on the potentially better heuristic combinations. We solve this by capping the minTree.

Capping the minTree means that once any of the individual heuristics used expands more nodes than an arbitrary number we call the *sampling cap* we will stop expanding any heuristic combinations using these heuristics for future minTree iterations(Algorithm:1). This way sampling effort is similarly split for all heuristic combinations. Also the overall prediction quality is improved by prioritising, i.e. improving weaker predictions over putting more sampling effort in already better predictions.

In the experiment section we show that by using the right sampling caps we can achieve a good prediction quality while keeping sampling costs reasonable.

Architecture of RIDA*

Architecture Overview

IDA* does not select its heuristic by itself so we had to modify standard IDA* adding both sampling and prediction modules. We call this algorithm Reconfigurable-IDA* (RIDA*).

The sampling module gathers data about the problem instance being solved. This is done *in situ*(i.e. while solving the problem). The sampling module also determines how much to sample for each heuristic combination by using the capping mechanism.

The prediction module uses the data gathered by the sampling module to predict the search effort of each heuristic combination for future iterations. The heuristic combination with the best expected performance is selected for the solving module.

Finally the solving module does the remaining search using the selected heuristic combination and regular IDA*. A detailed description of RIDA* modules follows.

Sampling Module

The objective for the sampling module is to obtain enough information to choose which heuristic combination is best at the lowest possible cost. We do not need to have perfect predictions for each heuristic combination's future performance. Predictions must be good enough to be able to choose a good heuristic combination with a reasonable sampling cost.

The HSTs being sampled need to be reasonably large for any prediction to be trustworthy. Even using minTree compression, culprit counters efficient assignment and capping mechanism this is not cheap.

If all the heuristic combinations maxTrees are above the sampling cap, then we stop sampling and choose the most promising combination. Currently the sampling cap is an

priori parameter, making the sampling cap an automatic *in situ* selection is work in progress.

Algorithm 1: Sampling Module Pseudocode

```

input : sampling cap (# of nodes),N heuristics
output: Size of each heuristic combination in powerset
        ( $2^N$ ) for sampled iterations

begin
  All N heuristics are uncapped and active;
  while at least one heuristic is over the sampling cap
  do
    while more nodes to expand for current iteration do // IDA* search with minTree:
      if (min(active heuristics)+g)  $\leq$  F-bound
      &&
      (At least one expanding heuristic is not capped) then
        Expand the node;
        Add to corresponding culprit counter
        number of children added;
      else // Finished or Node not expanded
        mark all pruning heuristics as inactive;
        // only evaluate expanding heuristics on descendant nodes
        backtrack as IDA* would do but
        reactivate inactive heuristics if
        backtracked past their pruning depth;
    All heuristics are capped;
    foreach heuristic combination in powerset do
      Calculate the final size counter for the
      heuristic combination adding corresponding
      culprit counters;
      if Final size of heuristic combination <
      sampling cap then
        uncap all heuristics  $\in$  heuristic
        combination;
  Update F-Bound;
end

```

Prediction Module

The Prediction Module attempts to predict which heuristic combination is likely to do best on future iterations. Its success depends on the quality of the prediction model, the amount of sampling done and the relative difference in the quality of the heuristics.

The prediction module generates a schedule of heuristics to be used by the solving module on the remaining iterations. Note that as the search space becomes bigger, heuristic combinations with more heuristics can become the best

Algorithm 2: Prediction Module Pseudocode

input : nodes generated per heuristic combination for last two iterations
input : $t(\#h)$ for the heuristics used(Eq:1)
output: Schedule of predicted heuristic performance
begin
 foreach *heuristic combination in powerset* **do**
 calculate HBF solving Eq:2 for last two iteration;
 predict size for next iterations using Eq:3;
 predict running time for next iterations using Eq:1;
 Now we can do a schedule predicting which is the expected best heuristic combination for future iterations;
end

option as their increased pruning power can compensate for their larger overhead.

Solving Module

The solving module does regular IDA* (maxTrees) with the scheduled best heuristic combination for the remaining iterations until a solution is found. Note that the best heuristic selection for a problem can change as the number of iterations increase and the search space becomes bigger, making more expensive but more accurate heuristics more competitive. If we made good predictions and the performance difference among heuristic combinations is bigger than the sampling effort, then the savings should outweigh the sampling costs.

Experiments

We present in this paper the first implementation of RIDA*. We are using the RIDA* framework to choose among five different possible Pattern Database heuristic combinations for the 15-puzzle.

Pattern Databases(PDBs)

Disjoint PDBs are the most effective admissible heuristics for N-puzzles as far as we know but they have a high computing cost. For our example, five compact 7-1-7 PDBs³ are available to solve each problem.

When non-additive PDBs are available, the standard heuristic combination mechanism (which preserves admissibility) is to maximize across all heuristic values(Korf and Felner 2002). This generates the smallest search space but

³(Holte et al. 2006) used five 7-1-7 disjoint PDBs to prove that maximizing across small PDBs could do better, in terms of the search space size only, than the best known PDB for the 15-puzzle(Holte et al. 2006). We firstly created our own PDBs as the original PDBs were not listed in the paper. The authors kindly provided us the original PDBs. When compared with our original databases their Max of Five was slightly better but, on a problem per problem basis, the best possible combination using our PDB was faster than theirs so we decided to keep our original PDBs for this paper.

also has the highest computing overhead cost per node. Maximizing across all available heuristics is not always the fastest heuristic combination. But finding the best heuristic combination for a problem *a priori* is too expensive as it requires solving the problem with all available heuristic combinations.

The heuristic overhead is specially large when PDBs are indexed to save memory(Korf and Felner 2002). Indexing is a computationally expensive operation and largely outweighs all other operations costs. Our experiments used indexed PDBs. We used the improved indexing technique presented by (Bonet 2008).

Parameterized time formula

Due to the very high heuristic indexing overhead (compared to other node expansion operations) we can simply predict time performance as in Eq(1). (López 2003) parametrised model of IDA* separated leaf nodes from internal nodes. This model is more accurate, in general, but because node expansion cost is so small compared to the heuristic evaluation cost for compact PDBs we can simplify the formula and assume time cost per node only depends on the number of compact PDBs used per node.

For the minTree, the overhead per node is the same as the maxTree of the five heuristics we are using. The additional computational costs per node due to RIDA* minTree logic are insignificant compared to the cost of indexing five compact PDBs per node.

Table 1: Time per node

Number of heuristics	Overhead per node($t(\#h)$ in Eq 1)
1	4.54 μ s
2	7.57 μ s
3	10.58 μ s
4	13.64 μ s
5	16.65 μ s

For our experiments, we can predict very accurately(less than 1 percent error) the run-time of an iteration if we know the maxTree size and treat the node overhead as a function of the number of heuristics being combined. The overhead per node will change if using different hardware or compiling/implementation options but should be a measurable parameter. If, instead of an expensive indexing operation, we had an efficient hashing function we would need a more sophisticated parametric model like those presented in (López 2003) or (Franco and Barley 2008).

We calculate the node overhead constant values via solving a large problem until the time overheads stabilize. Maximizing over five heuristics vs choosing a combination of those five heuristics is the example we use in this paper to showcase RIDA*.

Experiment Configuration

Holte et al(Holte et al. 2006) showed that, for the 15-puzzle, using five compact PDBs they could reduce the search space by a factor of 2.38 compared to the best known PDB at the

time. The five compact databases required the same memory as the larger PDB it was compared against. Interestingly this only translated to a time compression of 1.1 or 10% smaller due to the high overhead of evaluating five indexed heuristics compared to only one heuristic evaluation of the larger PDB.

We used, for our experiments, the five PDBs listed in figure 1. PDBs 1 and 4 are reflections over the diagonal and the rest are variations of PDB1 and PDB4. Each of the five PDBs is made up of 3 disjoint patterns. So to calculate the heuristic value of one of our PDBs we add together its 3 disjoint pattern stored distance. The PDBs themselves are not additive so to combine them the standard practice is to maximize over the PDBs in the combination(Korf and Felner 2002). We only choose patterns that follow Korf’s rule of thumb(Korf and Felner 2002, Section 3). We solved a 1000 random problems with the $2^5 PDBs = 32$ possible heuristic combinations. When using the best heuristic combination per problem for all 1000 random problems the total time saving is 39%(without sampling costs).

Which heuristic combination is the fastest depends on the problem being solved. It would be useful to know *a priori*, for a set of N heuristics, which heuristic combination, of the available 2^N , is best suited on a problem by problem basis. Unfortunately there is no generic mechanism to preselect the best PDBs for any domain(Holte, Grajkowski, and Tanner 2005). For the 15-puzzle we used Korf’s rule of thumb. RIDA* is well suited for this kind of problems as it can discover which heuristic combination is best for a problem as it solves it.

For our example, all possible 32 heuristic combinations are available to solve each problem instance. RIDA* will choose which, if any, of the five available PDBs to drop so that the problem is solved in the smallest possible time. This requires a delicate balance between sampling effort and prediction quality. The objective of RIDA* is to always keep sampling costs smaller than the savings.

Results

Table 2 shows our aggregated time performance for 1000 random problems as a function of the sampling cap chosen. We divided the 1000 random problems in 10 groups to study the mean savings and its standard deviation. The first sampling cap is “random”, this means no sampling so we select which heuristic combination to use at random. As we can see from the data it is worse to select heuristics at random than using the max of all available heuristics. In the next limit of “2 iterations” we do not use a sampling cap but instead do predictions as soon as we have done the 2 minimum iterations we need for Eq 1. It is better than selecting the heuristics at random but maximizing over all heuristics is still a better option due to poor prediction quality. The best savings are at a sampling cap limit of 1000, which forces all maxTrees to have at least 1000 nodes. For the next limits there is no improved savings as any optimality improvement is offset by the added sampling effort. As we raise the sampling caps the sampling costs increasingly outspend any savings.

The reason the sampling costs can get so high is that we

are picking arbitrary sampling caps. The higher sampling caps improve prediction accuracy but solve more problem instances on the expensive sampling phase before we make any prediction. Then instead of savings we get big losses for those problem instances.

If we could predict *a priori* (or *in situ*) the best sampling cap per problem instance, then we would keep more of the savings. Our preliminary experiments show us that the bigger the problem being solved the more savings we can get by adjusting the sampling cap on a problem by problem basis. But a good sampling cap for big problem instances tends to be too big for smaller problems. Hence the good optimality but poor overall results in table 2 once the sampling cap is over 5,000 nodes. Dynamically setting the cap limit is currently work in progress.

The second column on table 2 notes how many problems did not reach the sampling cap for all heuristic combinations before a solution was found. Note that even with sampling cap “2 iterations” two problems were solved with the minTree. These 2 problems did not have the minimum 2 iterations per heuristic combination necessary for calculating *in situ* the parameters for Eq 3. For the biggest sampling cap of 10^5 nodes only the 53 largest problems were solved with the selected heuristic combinations. The remaining 947 problems were solved in the sampling phase. The cost of solving 947 problems with the minTree is one order of magnitude bigger than the savings we get on the 6 maxTrees solved trees, hence the bad RIDA* running time. But the savings for the 6 maxTree solved problems were better than with the best average capping of 1000. A dynamic capping mechanism should take advantage of this.

Finally the biggest savings we got with any problem was 60%. Most problems had higher savings than the average reported on table 2. But the average performance suffers significantly every time we solved a problem while in the sampling phase. A dynamic capping mechanism should avoid solving problems in the sampling phase.

Table 3: minTree Savings

Size Compression	Time Compression
$\frac{\text{Sum of all 32 maxTrees Size}}{\text{Size of minTree}}$	$\frac{\text{Sum of all 32 maxTrees time}}{\text{Time of minTree}}$
4.04	3.85

For our problem and heuristic sets, the minTree size compression factor was 4.04(table 3). This means the number of nodes generated was approximately a quarter of those generated if all heuristic combination maxTrees had been individually generated. Even with only five heuristics the size compression is quite good, if we added more heuristics we would expect an even bigger compression factor. The time compression factor is 3.85, so the minTree takes approximately a quarter of the time compared to expanding the 32 heuristic combination maxTrees separately. Time compression is slightly worse than the size compression due to the additional computations we do for the minTree.

Figure 1: Our Five 7-1-7 PDBs

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	0 a a a a a a a b c c c c c c c	0 a a a a a a a c c c b c c c c	0 a c c a a c c a a c c a a b c	0 a b c a a c c a a c c a a c c	0 a c b a a c c a a c c a a c c
(a) 15-puzzle	(b) PDB1	(c) PDB2	(d) PDB3	(e) PDB4	(f) PDB5

Table 2: Results

Capping Limit (# nodes)	# Solved while sampling (# problems)	Avg Savings over 10 groups 100 problems each ($avg(100 \times (1 - \frac{selected\ time + sampling\ time}{Max\ of\ 5\ time}))$)	Standard Deviation	Sampling cost $avg(100 \times \frac{sampling\ time}{selected\ time})$	Optimality $avg(100 * \frac{best\ time}{chosen\ time})$
random combination	0	-22.70%	19.96%	0%	50.63%
2 iterations, no capping	32	-5.69%	17.58%	1.69%	59.73%
500	115	11.15%	10.40%	5.18%	71.95
1000	189	17.15%	9.3%	13.44%	82.41%
2500	341	7.60%	11.59%	38.22%	87.24
5000	487	-14.39%	11.28%	98.57%	95.32
10000	639	-67.54%	23.27%	216.07%	95.69
100000	947	-460.79%	77.03%	2174.74%	97.97

Conclusions and Future Research

Unfortunately, this paper does not provide definitive proof that the average cost of trying to determine the best heuristic (to use on that problem) can be made less than the average cost advantage of using that problem-best heuristic. However, this paper does provide evidence that suggests this can be the case.

This paper has introduced two new data structures, minTrees and culprit counter lattices, which lower the cost of predicting which heuristic will be best for specific iterations of IDA* on a given problem. The key issue was balancing the optimality of RIDA*'s predictions against the cost of obtaining those predictions. Capping was introduced as an *ad hoc* way of determining when to terminate RIDA*'s sampling. Given a good capping limit, RIDA* was able to, on average, determine which heuristic to use and solve the problem in less time than it took, on average, for a similar "reasonable" heuristic to just solve the problem.

RIDA*'s predictions of a heuristic's time to expand a problem's search tree to a given F-bound, were calculated by a run-time formula that used some *a priori* parameter values (e.g. the time it takes to expand a node using that heuristic) and some *in situ* parameter values (e.g. nodes generated in previous iterations, etc.). The *in situ* parameters were used as approximations of *a posteriori* parameters. The user-supplied capping limit determined the optimality of the prediction, where optimality is a measure of the impact on run-time of using the predicted best heuristic instead of using the true best heuristic for each problem.

Our next step is to strengthen our experiments. They need to directly compare the average cost advantage of using for each problem its predicted best heuristic (instead of using the best heuristic for the domain on each problem) against the average cost of predicting that heuristic for each problem. For this experiment, RIDA* will need to be extended so that it can automatically determine, for each problem, a capping limit that appropriately balances the optimality of the selected heuristic against the cost of the sampling used

to select that heuristic.

Finally (Zahavi et al. 2008) introduced a time saving mechanism. They propose to randomly apply one heuristic of the available set per node in the tree. This results in time savings as it maximizes the impact of using a single heuristic by reducing low h-values. We could apply RIDA* heuristic selection to preselect a reduced set of heuristics which would then be randomized as Zahavi et al proposed. This is future work as well.

References

- Bonet, B. 2008. Efficient algorithms to rank and unrank permutations in lexicographic order. In *Proceedings of the 1st International Symposium on Search Techniques in AI and Robotics*. Chicago, IL.
- Culberson, J.; Schaeffer, J.; of Computing Science, D.; and of Alberta, U. 1994. *Efficiently searching the 15-puzzle*. Dept. of Computing Science, University of Alberta.
- Franco, S., and Barley, M. 2008. In Situ Reconfiguration of Heuristic Search on a Problem Instance Basis. In *Proceedings of the 1st International Symposium on Search Techniques in AI and Robotics*. Chicago, IL.
- Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4(2):100–107.
- Holte, R., and Hernadvolgyi, I. 1999. A space-time tradeoff for memory-based heuristics. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, 704–709. JOHN WILEY & SONS LTD.
- Holte, R.; Felner, A.; Newton, J.; Meshulam, R.; and Furcy, D. 2006. Maximizing over multiple pattern databases speeds up heuristic search. *Artificial Intelligence* 170(16-17):1123–1136.
- Holte, R. C.; Grajkowski, J.; and Tanner, B. 2005. Hierarchical heuristic search revisited. In Zucker, J.-D., and

- Saitta, L., eds., *SARA*, volume 3607 of *Lecture Notes in Computer Science*, 121–133. Springer.
- Korf, R. E., and Felner, A. 2002. Disjoint pattern database heuristics. *Artif. Intell.* 134(1-2):9–22.
- Korf, R. E.; Reid, M.; and Edelkamp, S. 2001. Time complexity of iterative-deepening- a^* . *Artif. Intell.* 129(1-2):199–218.
- Korf, R. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*.
- López, C. L. 2003. On the heuristic performance of perimeter search algorithms. In Conejo, R.; Urretavizcaya, M.; and de-la Cruz, J.-L. P., eds., *CAEPIA*, volume 3040 of *Lecture Notes in Computer Science*, 445–456. Springer.
- Zahavi, U.; Felner, A.; Burch, N.; and Holte, R. C. 2008. Predicting the performance of ida^* with conditional distributions. In Fox, D., and Gomes, C. P., eds., *AAAI*, 381–386. AAAI Press.