Overcoming the Utility Problem in Heuristic Generation: Why Time Matters

Mike Barley, Santiago Franco, and Pat Riddle
Department of Computer Science
University of Auckland
Private Bag 92019
Auckland, 1142, New Zealand
barley@cs.auckland.ac.nz, Santiago.franco@gmail.com, pat@cs.auckland.ac.nz

Abstract

Progress has been made recently in developing techniques to automatically generate effective heuristics. These techniques typically aim to reduce the size of the search tree, usually by combining more primitive heuristics. However, simply reducing search tree size is not enough to guarantee that problems will be solved more quickly. We describe a new approach to automatic heuristic generation that combines more primitive heuristics in a way that can produce better heuristics than current methods. We report on experiments using 14 planning domains that show our system leads to a much greater reduction in search time than previous methods. In closing, we discuss avenues for extending this promising approach to combining heuristics.

Introduction and Motivation

In the last few decades, Artificial Intelligence has made significant strides in domain-independent planning. Some of the progress has resulted from adopting the heuristic search approach to problem-solving, where use of an appropriate heuristic often means substantial reduction in the time needed to solve hard problems. Initially heuristics were hand-crafted. These heuristics required domain-specific expertise and often much trial-and-error effort.

Recently, techniques (Haslum et al. 2007; Haslum, Bonet, and Geffner 2005; Edelkamp 2007; Nissim, Hoffmann, and Helmert 2011; Helmert, Haslum, and Hoffmann 2007) have been developed to automatically generate heuristics from domain and problem specifications. In this paper, we call the components that generate these heuristics *heuristic generators*. Many of these heuristic generators work by creating abstractions of the original problem spaces. Usually there are a number of different ways to abstract the problem space, and the generators search for a good abstraction to create their heuristic. These searches are often guided by predictions about the impact that different choices will have on the size of the problem's search space. The assumption is that the smaller the search tree the shorter the search runtime.

If heuristics have comparable evaluation times, using search tree sizes to compare heuristics makes sense because then smaller trees do translate into shorter search times. However, evaluation times can differ by many orders of magnitude[1]. Large differences in evaluation time can cause a system that reduces the search tree size to actually increase the search time. As Domshlak, Karpas et al. (2010) point out, effective heuristic generators need to consider heuristic's evaluation costs as well as impacts on search tree size.

In this paper, we describe a system, RA*, which chooses heuristics based upon reasoning about how those choices will impact the problem-solver's search time. RA* is able to reason about a heuristic's impact on search time because it measures both per node generation time and per node heuristic evaluation time. RA* then uses a runtime formula to predict the impact of a heuristic upon search time by using its estimates of the heuristic's impact on the average time per node and on the tree size. RA* can base its choice of heuristics upon their predicted impact on search time. We compare RA* against 7 other systems in the setting of the 2011 International Planning Competition (IPC). We claim that:

1. Creating better heuristics involves reasoning about their impact upon the search time as well as reasoning about their impact on the search tree size.
2. RA* produces the best heuristics, on average, in our experiments, precisely because it explicitly models the impact of heuristics upon total time per node and tree size.
3. RA* solves more of the IPC problems than any of the other seven systems in our experiments.

Current heuristic generators generate their heuristics from the domain and problem specifications. However, RA* approaches this differently by being given its heuristics and heuristic generators. Specifically, the problem that RA* solves is the following: Given a problem p and a set of heuristics H, find a subset, S, of H which, when maxed over, solves p in the shortest time. This approach allows RA* to use any heuristic. To include a new heuristic (generator), one simply passes it as input to RA*. The heuristic generators which are passed to RA* are run to generate their heuristics. These generated heuristics are added to any other heuristics (that were passed to RA*) and form the set of *primitive* heuristics. This set of primitive heuristics is the set that RA* will be selecting from.

[1]In our experiments, for a given state, one heuristic's evaluation time can be a million-fold greater than another heuristic's.

In the next section we discuss the background for this research. Following that, we describe the heuristic generation utility problem and both a simple approach and RA*'s approach to dealing with this utility problem. After that we describe experiments to evaluate RA*'s performance. In the final section we summarize our results and suggest the direction of future research.

Background

Throughout this paper we will be referring to some terms that are not universally understood, so we will define them here. Since the individual primitive heuristics, used in this paper, are consistent then so, too, is the maxed combination of these heuristics. This means that the f-values monotonically increase along the search paths. Because we are using A* with consistent heuristics, we can look upon A* as expanding layers of nodes, where each layer consists of all the nodes with the same f-value. This layer is called an *f-level* and the leaves of a layer are called an *f-boundary*.

There are a number of different bases for creating heuristics: abstraction (e.g., pattern databases (PDBs) (Culberson and Schaeffer 1996)), delete relaxations (e.g., h^{max} (Bonet and Geffner 2001)), and landmarks (e.g., LM-cut (Helmert and Domshlak 2009)). While we focus on PDBs in this section, our approach applies equally well to all types of heuristics, see Experiments section. The one proviso is that our approach relies on being able to measure the heuristic's evaluation time easily and that the measurement is a good approximation to the average evaluation time for that heuristic.

Early work by Holte and Hernadvolgyi (1999) on PDBs found that, on average, for a single PDB the larger the PDB (i.e., the less abstract the pattern), the smaller the search tree. Culberson and Schaeffer (1998) mention the possibility of combing patterns for smaller problems to produce heuristics for larger problems, e.g., using PDBs for the 8-puzzle and/or the 15-puzzle to serve as a heuristic estimate for the 24-puzzle. Instead of using PDBs from smaller problems, as suggested above, Holte et al. (2004) looked at combining smaller PDBs for the same problem. They found that maxing[2] over many small PDBs usually produced smaller search trees than a larger PDB whose size (in number of entries) was the sum of the sizes of the smaller PDBs. Holte found that while the search trees may be smaller, the search times can be larger. This was because when maxing over multiple PDBs, the per node evaluation cost is the sum of the individual PDB evaluation costs. Zahavi and Felner et al. (2007) looked at using a "random"[3] combiner instead of a max combiner for a set of symmetric PDBs. Zahavi showed that using random combiner over a large[4] set of PDBs can reduce the search tree size more than using max over a smaller subset of those PDBs. Another advantage of the random combiner was its smaller per node evaluation time.

[2]"maxing" in this context is evaluating all the individual heuristics (PDBs) and returning their maximum value.

[3]The random combiner selects a PDB at random from a set of PDBs.

[4]If max and random were done over the same of PDBs, then obviously max would produce smaller search trees.

This idea of not using all heuristics to evaluate every node has appeared several times recently (*selmax* (Domshlak, Karpas, and Markovitch 2010), Lazy A* (LA*) and Rational Lazy A* (RLA*) (Tolpin et al. 2013)). These decide which of a set of heuristics should be used to evaluate a given node. They can be explained most easily as choosing between a cheap and less accurate heuristic, h1, and an accurate but expensive heuristic, h2. *selmax* chooses which to use by estimating whether more time will be saved by using h1 and possibly expanding that node's subtree or by using h2 and possibly not expanding it. *selmax* learns online a classifier that predicts when it should use h2 at a node. *selmax* may expand more nodes than max but its overall problem solving time should be lower than either max or random.

When LA* creates a node, it computes h1 for the node, and puts it in the open list. When it is taken from the open list, h2 is computed and it is put back in the open list. When the problem is solved there are a number of nodes in the open list for which only h1 was computed thus saving the cost of the expensive h2 computations. Thus LA* expands exactly the same nodes as max does, but at a lower cost. RLA* is a variation of LA*. When a node is picked from the open list and has not yet been evaluated by h2, RLA* decides if it should be evaluated using h2 or just expanded immediately. If it will be expanded eventually anyway then using h2 to evaluate it is a waste of time. If evaluating it using h2 would prevent it from being expanded then the time to expand it and to evaluate all of its children with h1 would be a waste of time. So either decision could lead to wasted time. If we know how likely evaluating h2 is to be the correct decision then we could compute whether it was more wasteful, on average, to evaluate n with h2 or just expand n. RLA* uses this to decide which to do. RLA* may expand more nodes than LA* but should solve the problem quicker.

The field of heuristic search has been moving from an era of heuristics handcrafted for specific domains to one where heuristics are being automatically generated for specific problems. In the recent past, heuristics would be developed for a domain and then evaluated over a number of problems in that domain. If the heuristic seemed useful then it would be used by the problem solver to solve further problems in that domain. More recent work, (Edelkamp's GA-PDB (2007), Haslum and Botea et al.'s iPDB (2007), and Rayner et al's (2013)) in automatically generating heuristics, views heuristic construction as a search through a space of combinations of PDBs. Edelkamp searches through the space of combinations using genetic algorithms and Haslum searches through the space using hill climbing. Both approaches predict which combinations will most reduce the size of the search trees. Edelkamp uses the average h value of the heuristic combination as the fitness function, if one combination's average h value is greater than another combination's then the former will probably produce the smaller search tree. In Haslum et al's case, iPDB's hill climbing search needs a heuristic that predicts which modification of a combination will be best. iPDB simply counts the number of elements whose h-values are higher in the new PDB, the higher the count the better. Rayner et al view the construction of a new heuristic as selecting the best n-element subset from H, a set of heuristics, where both n and H are given. ”Best” means the subset that has the least loss for a given domain. The loss formula adds the subset heuristic's estimated distance between pairs of states. Like the two previous approaches, Rayner et al does not take into account the evaluation times associated with the individual primitive heuristics in the combination. Consequently, while the search tree may be smaller the search time may be greater.

In the work on maxing and randomizing over multiple PDBs, we often encounter the problem that simply reducing the search tree does not necessarily decrease the search time. This is a form of the *utility* problem. In the 1980's there was work on learning search control rules for planners that has many parallels with the current work on multiple PDBs. The goal of both is to add new sources of knowledge to speedup the search for solutions. In the 1980's, learning methods automated the acquisition of search control rules. However, Minton (1990) found that sometimes adding more search control rules would reduce the search space but could also increase the search time. Minton's approach to this problem was to estimate the utility of a new search control rule and only add those rules whose benefits outweighed their costs.

Recent work on combining multiple PDBs has encountered the same problem. Adding another PDB to max over, reduces the size of the search tree but increases the per node evaluation costs. Sometimes the benefits of having a smaller search tree are outweighed by the additional per node costs. Before adding another PDB to the combination, its utility must be estimated. Estimating a PDB's utility involves not only reasoning about its impact on search tree size but also on per node evaluation costs. These two impacts must be evaluated in light of how they both affect search time.

A simple runtime formula for a search-based system is:

$$srchTime(p,h) = nodesInTree(p,h) * timePerNode(h) \quad (1)$$

where p is a problem, h is a heuristic, $srchTime$ is the time spent searching for a solution to p using h, $nodesInTree$ is the number of nodes generated in p's search tree when using h, and $timePerNode$ is the average amount of time spent generating a node. Computing $timePerNode(h)$ is straight forward, it is the cost of generating that node plus the average time spent using h to evaluate the node's state. Computing $nodesInTree(p,h)$ for A* is an open problem.

While there are much better formulas for computing $nodesInTree(p,h)$ for IDA* (Lelis, Zilles, and Holte 2012), there are usually many paths to the same state, making IDA* is much less efficient than A*. In this paper, we use A* and consequently cannot make use of this more accurate prediction method. A common formula (Russell and Norvig 2009) for predicting the size of the A* search tree is:

$$nodesInTree(p,h) = \sum_{i=0}^{d} b_{p,h}^{i} \quad (2)$$

where p and h are as before, d is the length[5] of optimal paths for p, and b is h's effective branching factor for p. The effective branching factor, b is defined as $\sqrt[d]{nodesInTree(p,h)}$.

[5]We assume all operators have unit cost in this paper.

Overcoming the Utility Problem

In the last section we briefly discussed the utility problem in the context of heuristics. Namely, that while adding more heuristics to a max combination will decrease the size of the search tree, it may actually increase the search time. In this section, we will discuss two approaches to ameliorating this problem. The first approach is very simple and the second approach is our own.

The Space of Combinations

There are two spaces that are being searched. One is the space of combinations of heuristics and the other is the problem's search space. The latter space is the standard heuristic search space used by most planners. We will now briefly describe the former search space.

As we have seen, one common approach to creating a new heuristic is to combine two already existing heuristics. In this paper we will only be concerned with finding the utility of max combinations. Focussing on max combinations allows us to look at this problem as finding the best subset of a given set of heuristics. Looking at combinations as simple subsets immediately suggests one way of organizing the size of combinations, as a search space where nodes are the subsets, the root node is the empty set, and edges indicate the heuristic that is added to the parent's node's set of heuristics. The number of states in this search space is 2^n, where n is the number of heuristics in the given set. For large n, this space is too large to be searched exhaustively.

Simple Approach

We will look at a relatively simple approach to using a time based utility formula to select which subset of heuristics to use to solve a problem. This approach computes each combination's utility by using that combination to grow a search tree for the problem to sufficient depth[6] to give a good estimate of the combination's effective branching factor, b. The optimal solution depth, d, can be approximated by taking the maximum $h(initialState)$ over all the given heuristics and doubling it. Now both variables for computing tree size have been given values and assuming the per node times have been approximated, then the combination's utility can be calculated. We pick the combination with the best utility estimate and then continue solving the problem with that combination.

Our Approach: RA*

While we believe the previous approach might be good at selecting the most useful combination for a given f-level, it has a very high cost to calculate the combinations' effective branching factors (EBFs). Especially since for n primitive heuristics, there would be 2^n EBFs to calculate. Since the calculation of an EBF involves exploring the initial part of the problem's search tree, much of that tree would be grown 2^n times. So, while the chosen combinations might be very useful, the approach might not be.

[6]For RA*, we assume that when intervals take 3 seconds or more the search has gone “deep enough”.

Our approach's focus is on reducing the overall system's time to solve the problem. Our approach trades the quality of the selected combination against the cost of making that selection. We try to reduce this selection cost primarily by reducing the time to predict each combination's effective branching factor.

Like the simple approach above, our approach predicts the time for the current f-level. Unlike the approach above, we only make one selection and continue with that selection to solve the rest of problem. In our experiments, our current implementation of RA*'s heuristic generation process takes almost 70% of the total problem solving time. The generation time includes sampling time and utility calculation time.

We will first look at the different phases that RA* uses in solving a problem, then at our utility formula, next at the combination pruning heuristics used to reduce the combinatorics of the selection process and finally at the mechanisms that underlie our approach to reducing the time to approximate the combinations' effective branching factors: the min combiner and culprit counters.

RA* Phases RA* inputs a set of heuristics and heuristic generators and outputs a heuristic. To do this it goes through a sequence of phases (which will be described in the next few sections):

- Run heuristic generators to produce their "primitive" heuristics.
- Time how long it takes to generate a node.
- Time how long each primitive heuristic takes to evaluate a state.
- Initial Growth Period: Expand the initial part of the problem's search tree to an adequate frontier size to start sampling.
- Sample Period: Nodes are taken from the frontier to obtain information on each candidate combination's EBFs.
- Estimate the utility for each candidate heuristic combination.
- Finish solving the problem with the best combination.

Utility Formula The utility formula for a combination c is:

$$U_{c,f} = \mid Front_f \mid * EBF_{c,f} * (eRN_c + gRN) \qquad (3)$$

where f is the current f-level, $Front_f$ is the set of frontier nodes at the beginning of f, $EBF_{c,f}$ is the effective branching factor at f for combination c, eRN_c is the per node evaluation time, and gRN is the per node generation time. We obtain $EBF_{c,f}$ from sampling this f-level as described in the rest of this section. $EBF_{c,f}$ is simply all the sampled nodes that would have been generated by c in that f-level divided by the number of sampled frontier nodes. eRN_c is simply the sum of the evaluation times of all the heuristics in the combination.

Pruning Heuristics For any non-trivial set of heuristics, the number of combinations is large. In our experiments, we use a set of 45 heuristics. Thus there are 2^{45} (approximately $32 * 10^{12}$ combinations). Far too many to predict times for. Instead, we want to eliminate as many of the clearly inferior combinations as possible. To do this we use some rules of thumb to identify and prune away less useful primitive heuristics. We now discuss some of these rules.

At the end of the initial growth period, we randomly select frontier nodes[7] to grow during the sampling period. We call these nodes the *sample roots*. We classify the primitive heuristics into strong, medium, and weak by evaluating every sample root using every heuristic. We count the times each heuristic had the highest value for a sample root. If a heuristic's count is high enough[8], it is called a *strong* heuristic. Of the remaining heuristics, with associated non-zero counts, the top user-specified percentage[9]are called *medium* heuristics. The remaining ones are called *weak*. A *combination candidate* is defined to be one which has at least one strong primitive heuristic, and the remainder are at least medium primitive heuristics. We generate the set of combination candidates in the ascending subset size order and prune away the ones that do not meet our criteria (specified above). We stop generating after a user-specified number[10], *MaxComb*, of candidate combinations.

Min Combiner and the Search Tree Our goal is to grow only one search tree, a *MinTree*, rather than grow a separate tree for every combination. We do this by using a *min* combiner. A min combiner is very much like a max combiner, except that the heuristic value of a min combiner over a set of heuristics is the minimum value produced by that set of heuristics.

As in the simple approach, we need to grow the search tree deep enough, that our approximations will be reasonable. We call this period, the ***initial growth period*** . While we are in the initial growth period, we use the max combiner to keep the tree size small. When the tree is deep enough, at the end of a f-boundary, we switch to using the min combiner for one f-level. We call this f-level the ***sampling period***. Using the max combiner, a node is expanded only if all the heuristics agree to expand it. However, for a min combiner, a node is expanded as long as one of the heuristics wants to expand it.

Culprit Counters and Combination Counters In this section we give a very brief overview of how RA* computes the number of nodes generated by each heuristic combination during the sampling period. Franco and Barley et al. (2013) provide a more detailed description of this process.

During the sampling period, culprit counters are used to compute the effective branching factors for each combination. The min combiner allows us to only grow one search tree and the culprit counters allow us to only update one counter (the culprit counter) per node expanded.

When we expand a node, the primitive heuristics that agreed to the expansion are call the *culprits*. The culprits are a subset of the full set of heuristics. We associate a counter

[7]In our experiments the number of nodes RA* selects to sample is 1% of the previous f-level.

[8]In our experiments, the count must be higher than is 50% of the number of samples taken.

[9]In our experiments, the user-specified percentage is 30%.

[10]In our experiments, this was set to 20,000

with each culprit set[11], the *culprit counter*. When a node is expanded in the sampling period, we add its number of children to the counter associated with its culprit set.

When the sampling period is over, we now need to calculate the number of nodes generated for each combination. During the final search to find a solution, RA* will be using the max combiner, which means that all the heuristics in the combination must agree to generate a node for it to be generated. This means that for each combination of heuristics, we compute how many nodes would have been generated for this f-level. For a given combination, we sum all the culprit counters whose culprit set heuristics are a subset of the combination's heuristics. After the counts for all the combinations have been summed, we can compute the effective branching factor for each combination by dividing its count by the number of sampled roots. We can also calculate the per node evaluation time for that combination of heuristics by adding together the per node evaluation time of each heuristic in that combination.

Once, the effective branching factors and the per node evaluation times have been calculated, then the utility for each combination is calculated and the combination with the highest utility is selected to finish the search for the solution.

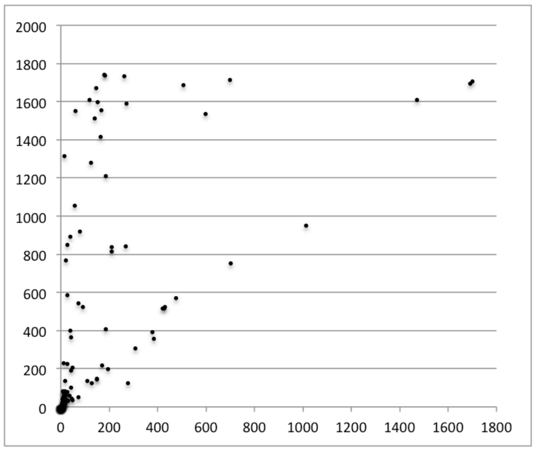

Figure 1: Comparing Max and RA* Heuristics' Search Times in Seconds, x-axis is RA*'s Search Time and y-axis is Max's Search Time

Experiments

In this section we will try to answer two questions: (1) how do RA*'s heuristics compare with state-of-the-art heuristics; (2) how does the RA* system compare with other state-of-the-art systems. The requirements we used for state-of-the-art systems are: admissible, be integrated into the Fast Downward's system (Helmert 2006), and be interruptible[12]. We choose iPDB (Haslum et al. 2007) because it is currently one of the best PDB generators. We choose LM-cut (Helmert and Domshlak 2009) because it is one of the best landmark heuristics. We choose h^{max} (Bonet and Geffner 2001) because it was not dominated by any single heuristic and while it was seldom among the best heuristics for any problem it was better than most of the heuristics for many of the problems. We choose GA-PDB (Edelkamp 2007), because while it was not one of the best individually, it can easily generate a set of heuristics to max over.

Besides having a good set of heuristics to select from, we also wanted to compare how well RA* did against two default approaches, namely, Max, i.e., taking the max over the set, and Random, i.e., randomly selecting which heuristic to use to evaluate a given state. These two defaults represent the two extreme approaches of minimizing the search tree size versus minimizing the per node time cost.

In all the experiments, the same A* search algorithm is run, the only difference is the particular heuristic being used. The code for the heuristics LM-cut, h^{max}, and iPDB come from the Fast Downward's code repository. Both GA-D and GA-ND also come from the Fast Downward's code repository, they both use the same code, the only difference is the setting of the "disjoint_patterns" parameter, which controls whether the patterns within a collection need to be disjoint (GA-D) or not (GA-ND).

LM-cut and h^{max} are online heuristics that compute their values by search in an abstraction space. iPDB, GA-D and GA-ND are heuristic generators that produce PDBs[13]. iPBD produces a single PDB. We have created GA-D and GA-ND by using Edelkamp's heuristic pattern generator and varied its mutation rate from 0 to 1 in steps of .05. Thus each of GA-D and GA-ND produce 21 individual PDBs, which they max over. The seed was changed for each of the 42 GA-PDB[14] generations to increase the variety between heuristics. For every problem there were 45 heuristics provided by these 5 systems. The Max and the Random systems combine these 45 heuristics using the max and random combiners respectively. RA* inputted those 45 heuristics and selected a subset to max over to solve the current problem. When RA* is selecting heuristics it can choose any subset out of those 45 heuristics including any combination of the 42 GA-PDB heuristics. However, in our tables below, when we talk about GA-D or GA-ND, we are talking about the max over the 21 heuristics produced by that generator.

The 2011 IPC setting for the deterministic optimization track has 14 domains which have 20 problems each. There are also both time (30 minutes) and memory limits (6 GB of RAM). The competition metric is simply the number of problems solved within those constraints. Out of the 280 problems, only 192 of them were solved by any of our 8 systems. We have limited our analysis to only those problems which were solved by at least one of our systems.

[11]We only store counters for the culprit sets actually encountered during the sampling period.

[12]This requirement was for heuristic generators. Some generators can run a long time, so RA* sets a time limit for how long a generator can run and kills any generator that exceeds that limit.

[13]Due to a bug with the resetting of the random seed which was discovered while writing this paper, the heuristics used by GA-D and the GA-D heuristics and used by Random are not always the same. Only the GA-D heuristics used by RA* and by Max are the same. No other system in these experiments uses GA-D heuristics.

[14]When we refer GA-PDB, we are referring to both GA-D and GA-ND.

Evaluating RA*'s Heuristics

RA*, Max, and Random all work with the same set of heuristics and consequently have the same variety of heuristics available. The three systems just deal with them differently. The other five systems have less variety available. We will first compare RA*'s search time with Max and Random's and then compare it with the remaining five systems.

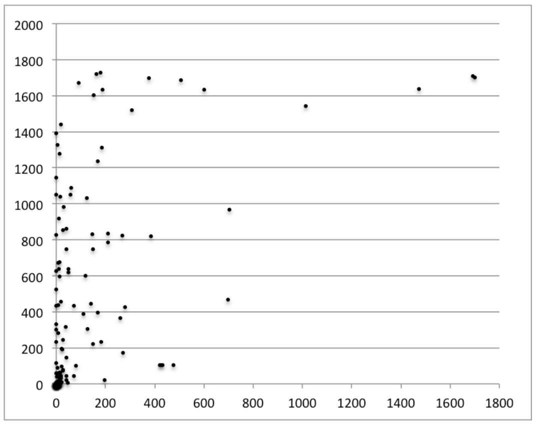

Figure 2: Comparing Random and RA* Heuristics' Search Times in Seconds, x-axis is RA*'s Search Time and y-axis is Random's Search Time

Comparing RA*'s Heuristic with Max's and Random's If we look at Figure 1, we see that RA*'s heuristics reduce search a lot more than Max. Why is this? Max maxes over all 45 heuristics, and RA* chooses a subset of the 45 heuristics and maxes over that subset. RA*'s per node time is lower than Max's but its search tree is larger. If we look at Tables 1 and 3, we see that while RA* produces 9 times as many nodes as Max, Max's time per node (TPN) is 26 times as long as RA*'s[15]. Resulting in Max's average search time being 3 times (26/9 to be precise) as long as RA*'s. While RA* almost always beats Max, it does not always. There are a few problems where Max has a shorter search time.

Since LM-cut's evaluation time is so high, perhaps including it was a mistake. If we did not have LM-cut in the set of heuristics being maxed over, then would Max do even better? To test this, we reran the experiments on Max with and without LM-cut. The machine where we had originally run our experiments had been replaced by a new, faster machine, this means that our results for the Max's can no longer be meaningfully compared to the other results. On the new machine, both Max with and without LM-cut solved the same number of problems, namely 181. However, of those 181 problems, only 171 were solved by both. Thus, each solved a different set of ten problems that the other did not. Taking LM-cut out of the set of maxed over heuristics, did indeed make a difference in which problems were solved, but not as far as the number of problems solved.

Looking at the scatter plot for RA* and Random, Figure 2, looks similar to the one for RA* and Max. However, there are a few more problems where Random does better than RA*; for the rest, RA* is much faster than Random. RA*'s average search time is 88 seconds, while Random's is 306 seconds. For every search node, Random chooses one of the 45 heuristics to evaluate that node's state. This allows Random to sometimes solve a problem faster than RA*. However, RA* generates fewer nodes than Random and RA*'s average per node overhead is also less than Random's.

It is evident that RA* does much better, on average, than using either Max or Random. However, it is unlikely to be true for all domains and settings. Work remains to be done to characterize when each of these approaches is best and why.

	Avg.	Std. Dev.	Sum
RA*	88.46	240.74	17,000
GA-ND	123.85	280.17	23,800
GA-D	133.83	314.39	24,900
iPDB	212.01	455.71	40,700
MAX	260.91	500.40	50,100
RAND	321.23	505.60	59,600
LM-cut	446.62	699.15	85,800
h^{max}	706.12	820.01	136,000

Table 1: Search Time For Each System

Comparing RA*'s Heuristic with the Individual Member Heuristics Another question is: Has the state-of-the-art reached the point where they can produce a heuristic which cannot be improved by RA* maxing over them and others? In other words, have the individual heuristics gotten so good that using RA* is superfluous?

When comparing the average search times per problem in Table 1, we see that RA* does quite well. The two next best systems, GA-ND and GA-ND, are .4 and .5 times slower respectively, while the next system, iPDB, is almost 1.5 times slower. The answer would seem to be that the field still has some way to go in improving our heuristic generators.

RA*'s heuristics significantly reduce the average search time. Why is this? We hope it is because RA* is better at finding good tradeoffs. However there are at least three obvious alternative possibilities: (1) more variety of heuristics; (2) picking the right heuristic; and (3) smaller search trees.

RA* has more heuristics available. This greater variety might mean that the heuristics are more complementary, i.e., where one heuristic underestimates a distance another one will be more accurate. If we assume that RA* is simply using all of the heuristics, and that this variety was the main reason for RA*'s heuristics' better performance then, since Max has the same variety as RA*, it should also perform better than iPDB, GA-D and GA-ND. However, from Table 1 we can see that this is not true. All three of these have shorter average search times than Max. So, it is not just that RA's is using a wider variety of heuristics that accounts for its shorter search times.

The second possibility is that tradeoffs have nothing to do with RA* doing better than iPDB, LM-cut, h^{max}, or the GAPDB's. Perhaps, the reason for RA* doing better is that for some problems iPDB is the best, for some others, the GAPDBs are best, and for yet others LM-cut is best and RA* is simply picking the right heuristic for the right problems. If this were true then we would expect to see most of

[15]Max's TPN=260.91/456,430.9=.000572 sec/node and RA*'s TPN = 88.46/4,100,515.7 = .000022 sec/node and their ratio is 26.

RA*'s improvement, over the other heuristic generators, in the cases where singleton heuristics were chosen. Table 2 shows the number of heuristics used by RA* to solve problems. Note that a lot of the problems, 97 to be precise, were solved before the end of the sampling period and so were solved before having selected a combination. The problems in this table represent the 90 problems solved after having selected a heuristic combination. Over half of the problems (56%) were solved using more than one heuristic. In fact, there was one problem where all 45 heuristics were used which resulted in substantial time savings over either just using either h^{max} or LM-cut. This is even though both of these two heuristics were part of the 45 heuristics being used. So, the reason for RA* doing better than any of the individual heuristics seems unlikely to simply be that RA* is picking, for each problem, the heuristic that is best for that problem.

Table 2 could be explained if RA* picked singleton subsets for hard problems and multiple heuristics subsets for much easier problems. To check this we looked at the average search time of min(h^{max}, LM-cut) on these sets of problems. While there are definitely some combination sizes that were used on easier problems, specifically, combination sizes 5 and 6 (where the average min(h^{max} , LM-cut) search time is much lower than for the singleton heuristics), the rest are almost as hard or much harder. In fact, the problem RA* solved with 9 heuristics in less than 3 minutes, both h^{max} and LM-cut could not solve within their 30 minute time limit. It seems clear that RA* did not chose multiple heuristics subsets for the easier problems, that it did indeed choose them for the hardest of the problems.

The last possibility is that RA* selects heuristics that will produce the smallest search trees. In other words, RA* might not really need to bother about estimating search times, it would have worked just as well just estimating tree sizes. If we look at Table 3, we see that Max has the smallest average tree size with 456,430 nodes generated[16] with LM-cut following with 841,706 nodes. RA* is 4th with 4,100,515 nodes, almost nine times as many nodes as Max. It is clear that the reason that RA*'s heuristics are the fastest is not because they generate the smallest search trees.

The most likely explanation, for why RA*'s heuristics are best, is that they are better at finding tradeoffs between per node times and the size of the search tree. Obviously, RA* sometimes gets this wrong or it would always take less time to solve a problem than any of the other methods. One would expect, that since RA* explicitly reasons about the search time, that RA* would be better at finding good tradeoffs than systems that simply reason about search tree sizes.

Evaluating the RA* System

In evaluating RA*, it is not enough to look at how quickly it is able to solve problems after it picked a heuristic combination. We must also look at, how long it took RA* to

# h's	# probs	RA* Time Avg	RA* Time. Std. Dev.	Avg min(h^{max},LM-cut)
1	40	173.80	297.61	656.33
2	15	139.94	172.00	744.47
3	9	159.85	188.83	845.16
4	6	42.22	70.83	943.14
5	4	50.57	79.06	51.22
6	8	43.91	94.95	116.25
7	5	50.192	58.23	607.66
9	1	168.79		1800
10	1	140.2		954.3
45	1	48.51		753.34

Table 2: Impact of Singleton Heuristics vs Multiple Heuristics on Average Search Time

pick that combination. We do this in two ways. One is to use the IPC[17] criteria to evaluate different domain-independent planners. In our experiments we use exactly the same planner but with different heuristic generators. Thus, we can use the IPC criteria as one way to evaluate RA*. Another way is to use the total time that each system uses in attempting to solve all the problems. We will look at both approaches.

	Avg.	Std. Dev.	Sum
MAX	456,430.9	1,314,724	87,634,731
LM-cut	841,706.8	2,579,406	134,000,000
GA-D	3,614,383.8	12,985,491	621,674,005
RA*	4,100,515.7	319,394,651	783,198,505
RAND	7,265,320.7	20,769,307	1,039,000,000
GA-ND	7,868,931.4	19,408,850	1,511,000,000
h^{max}	14,679,327	45,239,202	1,806,000,000
iPDB	15,826,878	43,379,822	3,039,000,000

Table 3: Generated Nodes For Each System

IPC's Evaluation Criteria The IPC evaluation criteria for the optimal deterministic track was a count of the number of problems solved within the time and memory limits. The time limit is applied to the total time the system takes to solve a problem, including time taken to generate heuristics.

Looking at table 4 we see that RA* was able to solve 187 of the 280 problems. This was better than the other systems. RA*'s heuristics failed to solve 5 of the 192 solved[18] problems. In each of these 5 cases, RA*'s heuristic caused A* to exceed the memory limit rather than the time limit.

Using the IPC criteria, RA* would be considered the best heuristic generator. However, this seems a very arbitrary criteria, because if we changed either resource limit or had different problems or domains, we might get very different results. If we look at the fourteen domains, we see that in nine of the domains, RA* solved as many problems as the best of the other seven systems. In three domains RA* solved one less problem than the best of the other systems and in two domains, it solved more problems than the others. In Scanalyzer, it solved three more than best of the other systems and in Visitall if just solved one more than the others. The three

[16]We only include nodes in our counts for problems that were solved by that system. Additionally, the node generated counts do not count the number of nodes generated until a solution was found. Rather, it is the count of nodes until A* hit the optimal solution's f-boundary. Thus, our generated node counts are the minimal node count for any A* search that uses the same heuristic.

[17]An International Planning Competition (IPC) is run biannually. These competitions provide a number of domains and problems that serve as benchmarks for evaluating planners.

[18]These are the problems solved by at least one of the systems.

	Problems Solved									Avg. # Heuristics		
	RA* problems solved (problems solved after sampling phase)									Used By RA*		
	RA*	GA-ND	GA-D	MAX	iPDB	LM-cut	RAND	h^{max}	total	GA	iPDB	LM-cut
Barman	4(4)	4	4	4	4	4	4	4	4	1.333	0.25	1
Elevators	19(9)	19	19	18	16	18	16	13	19	2.22	0	0
Floortile	6(4)	3	4	6	2	6	2	6	6	3.5	0	0.5
Nomystery	20(4)	20	20	20	18	14	20	8	20	3.25	0	0
Openstack	15(10)	16	16	13	16	16	15	16	16	1	0	0
Parcprinter	13(3)	13	13	13	11	13	10	11	13	1.67	0	0.33
Parking	7(7)	1	1	1	7	1	0	0	7	0	1	0
Pegsol	19(14)	19	19	17	18	17	17	17	19	6.86	0	0
Scanalyzer	14(4)	10	9	11	10	11	6	6	14	5	0	0
Sokoban	20(9)	20	20	20	20	20	20	20	20	0.67	0.67	0
Tidybot	13(11)	12	11	11	13	12	7	5	13	0.09	1	0
Transport	9(4)	10	10	8	6	6	7	6	10	1.25	0	0
Visitall	18(2)	16	17	17	16	10	13	9	18	7	0	0
Woodworking	10(5)	10	9	11	7	11	5	4	13	1.8	0	0
Total	187(92)	173	172	170	164	159	143	123	192	2.9	weighted average	
Ratio	.97(.49)	.89	.88	.88	.85	.83	.74	.64				

Table 4: Number of Problems Solved by Each System

columns on the right-hand side of Table 4 show for each domain the average number of heuristics used by RA* on that domain's problems that RA* solved after having selected a heuristic combination (this is the number of problems shown in parentheses in the RA* column on the left-hand side of the table). RA* never used h^{max} on any of these problems and it does not have a column on the right-hand side of the table. If we do a weighted sum (weighted by the number of problems RA* solved using a heuristic combination in that domain) of the number of heuristics used in each domain, we find that RA*, on average, uses 2.9 heuristics on each problem. In the two domains where RA* did better than the others, it had a larger heuristic combination than normal. In scanalyzer, it used 5 heuristics on average, and in Visitall, it used 7 heuristics on average. In the three domains where it did poorer (though only by one problem), RA* had smaller heuristic combinations than normal. In those 3 domains, on average it used less than 2 heuristics (specifically, on average it used just 1, 1.25, and 1.8 heuristics). RA* does better when using non-singleton heuristic combinations.

	Avg.	Std. Dev.	Sum
GA-D	205.79	371.40	38,700
iPDB	227.68	469.94	43,700
GA-ND	232.68	333.67	44,700
RA*	284.97	319.38	54,700
LM-cut	446.62	699.15	85,800
MAX	453.31	571.95	87,000
RAND	517.68	607.15	99,400
h^{max}	706.12	820.01	136,000

Table 5: The Total Times for Solving Problems

Total Time Evaluation Criteria All time used by a system to solve a problem is included in its total time figures. This includes the time to run the heuristic generators as well as the time used by RA* to make its selection. Thus, RA*'s, Max's, and Random's total time includes the time taken by iPDB, GA-D, and GA-ND to generate their heuristics. The total time is subject to the time limit.

If we look at Table 5, we see GA-D with the shortest average total time of 206 seconds and RA* with the 4th shortest average time of 285 seconds, 40% longer than GA-D. Using this criteria, RA* does worse. This indicates that RA* is not a clear winner. RA* does not do any meta-level reasoning about its own tradeoffs between its reductions in the time to solve a problem and the time it spends determining the heuristic combination to use to reduce that problem solving time. The goal is to reduce this total time of both determining the combination to use and using that combination to solve the problem. The last column of Table 6 shows the percentage of the total time that each system spends on generating PDBs. If we look at RA*, we see that it spends the highest percentage of its time generating PDBs, almost 70%. While the next system, GA-ND, spends less than half of its total time producing PDBs. What is amazing is that iPDB only spends 6% of its total time producing its PDB. RA*'s generation of all 45 heuristics is obviously a performance issue. This is an area for future research.

	Avg	Std. Dev.	Sum	Ratio
RA*	196.51	200.63	37,729.59	69.0%
GA-ND	108.84	111.04	20,896.48	46.8%
MAX	192.40	201.25	36,940.87	42.4%
RAND	196.45	208.44	37,700	37.9%
GA-D	71.96	98.90	12,815.52	35.6%
iPDB	15.67	25.24	3,008.4	6.6%
h^{max}	0	0	0	0%
LM-cut	0	0	0	0%

Table 6: Runtime for Generating PDBs

Conclusions

Our main claim is that heuristic generators can suffer from the utility problem. The utility problem is when the generator's attempts to create heuristics that generate smaller search trees also makes those heuristics take a longer time.

Creating better heuristics involves reasoning about the heuristics' impact upon search time. Smaller search trees do not guarantee shorter search times. What is necessary, however, is for the heuristic combiner to make the best tradeoffs between the heuristic's reductions in the search tree size and any additional per node evaluation costs. We have discussed RA*, a system that explicitly reasons about the heuristics' impact on the system's search time. Our experiments are done in the 2011 IPC setting. These experiments use a set of five state-of-the-art admissible heuristics and compare RA*'s performance with two default approaches (Max and Random). RA* is clearly superior to both approaches. These experiments also compare RA*'s performance against the individual heuristics. RA*'s heuristics, on average, reduce the search time more than any of the individual approaches. Our analyses of the experiments indirectly support our claim that RA*'s superiority comes from its explicit reasoning about the heuristics' impact on search time.

While RA* generates better heuristic combinatins, how does it compare to other systems that generate heuristics? We analysed this in two ways. We used the 2011 IPC criteria and we also compared the systems based on their total time. The 2011 IPC criteria for deterministic optimal planners was the total number of problems solved under given time and memory constraints. RA* did best with 187 problems solved out of 280, while the next best system, GA-ND, solved 173 problems. However, when we compared RA* against the seven other systems, it came in 4th by running 40% longer than the fastest system, GA-D. This is unsurprising as RA* generates all 43 PDBs (using GA-D, iPDB, and GA-ND), while the top 3 systems (GA-D, iPDB, and GA-ND) only generated 21, 1, and 21 PDBs, respectively. The PDB generation time took up 70% of RA*'s total time.

The greatest improvement for RA* would likely come from better handling of PDB generation. Currently, all the candidate PDBs are generated and then they are evaluated at one time. It seems plausible that a better approach is to adopt the approaches of iPDB and GA-PDB to integrate the generation and evaluation of heuristics into an incremental search. This integration would have the drawback of making it more difficult to add new heuristic generators to RA*.

Acknowledgements

This material is based on research sponsored by the Air Force Research Laboratory, under agreement number FA2386-12-1-4018. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

We would also like to thank Malte Helmert, et al, for making available the Fast Downward code repository and the 2011 International Planning Competition domains and problems. Both of these have made doing research a lot easier. We would also like to thank Malte Helmert for his generous and prompt technical guidance whenever we had doubts on how FD worked.

References

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

Culberson, J., and Schaeffer, J. 1996. Searching with pattern databases. In *11th Conference of the Canadian Society for the Computational Study of Intelligence*.

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

Domshlak, C.; Karpas, E.; and Markovitch, S. 2010. To max or not to max: Online learning for speeding up optimal planning. In *AAAI*, 1701–1706.

Edelkamp, S. 2007. Automated creation of pattern database search heuristics. In *Model Checking and Artificial Intelligence*, volume 4428 of *LNCS*. 35–50.

Franco, S.; Barley, M.; and Riddle, P. 2013. A new efficient in situ sampling model for heuristic selection in optimal search. In *Australasian Joint Conference on Artificial Intelligence*.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *AAAI*.

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New admissible heuristics for domain-independent planning. In *AAAI*.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway. In *ICAPS*.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *ICAPS*, 176–183.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26(1):191–246.

Holte, R. C., and Hernádvölgyi, I. T. 1999. A space-time tradeoff for memory-based heuristics. In *AAAI*, 704–709.

Holte, R.; Newton, J.; Felner, A.; Meshulam, R.; and Furcy, D. 2004. Multiple pattern databases. *ICAPS* 122–131.

Lelis, L.; Zilles, S.; and Holte, R. 2012. Fast and accurate predictions of IDA*'s performance. In *AAAI*, 514–520.

Minton, S. 1990. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence* 42(2):363–391.

Nissim, R.; Hoffmann, J.; and Helmert, M. 2011. Computing perfect heuristics in polynomial time: On bisimulation and merge-and-shrink abstraction in optimal planning. In *IJCAI*, volume 3, 1983–1990.

Rayner, C.; Sturtevant, N.; and Bowling, M. 2013. Subset selection of search heuristics. In *IJCAI*, 637–643.

Russell, S., and Norvig, P. 2009. *Artificial Intelligence: A Modern Approach.* Prentice Hall.

Tolpin, D.; Beja, T.; Shimony, S. E.; Felner, A.; and Karpas, E. 2013. Towards rational deployment of multiple heuristics in A*. In *IJCAI*, 674–680.

Zahavi, U.; Felner, A.; Schaeffer, J.; and Sturtevant, N. 2007. Inconsistent heuristics. In *AAAI*, 1211.