Towards Rational Deployment of Multiple Heuristics in A*

David Tolpin
Tal Beja
Solomon Eyal Shimony
CS Department
Ben-Gurion University
Israel
tolpin,bejat,shimony}@cs.bgu.ac.il

Ariel Felner
ISE Department
Ben-Gurion University
Israel
felner@bgu.ac.il

Erez Karpas
Faculty of IE&M
Technion
Israel
karpase@gmail.com

Abstract

The obvious way to use several admissible heuristics in A* is to take their maximum. In this paper we aim to reduce the time spent on computing heuristics. We discuss Lazy A*, a variant of A* where heuristics are evaluated lazily: only when they are essential to a decision to be made in the A* search process. We present a new rational meta-reasoning based scheme, rational lazy A*, which decides whether to compute the more expensive heuristics at all, based on a myopic value of information estimate. Both methods are examined theoretically. Empirical evaluation on several domains supports the theoretical results, and shows that lazy A* and rational lazy A* are state-of-the-art heuristic combination methods.

1 Introduction

The A* algorithm [Hart et al., 1968] is a best-first heuristic search algorithm guided by the cost function $f(n) = g(n) + h(n)$. If the heuristic $h(n)$ is admissible (never overestimates the real cost to the goal) then the set of nodes expanded by $A^*$ is both necessary and sufficient to find the optimal path to the goal [Dechter and Pearl, 1985].

This paper examines the case where we have several available admissible heuristics. Clearly, we can evaluate all these heuristics, and use their maximum as an admissible heuristic, a scheme we call $A_{MAX}^*$. The problem with naive maximization is that all the heuristics are computed for all the generated nodes. In order to reduce the time spent on heuristic computations, Lazy A* (or LA*, for short) evaluates the heuristics one at a time, lazily. When a node $n$ is generated, LA* only computes one heuristic, $h_1(n)$, and adds $n$ to OPEN. Only when $n$ re-emerges as the top of OPEN is another heuristic, $h_2(n)$, evaluated; if this results in an increased heuristic estimate, $n$ is re-inserted into OPEN. This idea was briefly mentioned by Zhang and Bacchus (2012) in the context of the MAXSAT heuristic for planning domains. LA* is as informative as $A_{MAX}^*$, but can significantly reduce search time, as we will not need to compute $h_2$ for many nodes. In this paper we provide a deeper examination of LA*, and characterize the savings that it can lead to. In addition, we describe several technical optimizations for LA*.

LA* reduces the search time, while maintaining the informativeness of $A_{MAX}^*$. However, as noted by Domshlak et al. (2012), if the goal is to reduce search time, it may be better to compute a fast heuristic on several nodes, rather than to compute a slow but informative heuristic on only one node. Based on this idea, they formulated selective max (SelMAX), an online learning scheme which chooses one heuristic to compute at each state. SelMAX chooses to compute the more expensive heuristic $h_2$ for node $n$ when its classifier predicts that $h_2(n) - h_1(n)$ is greater than some threshold, which is a function of heuristic computation times and the average branching factor. Felner et al. (2011) showed that randomizing a heuristic and applying bidirectional path-max (BFMx) might sometimes be faster than evaluating all heuristics and taking the maximum. This technique is only useful in undirected graphs, and is therefore not applicable to some of the domains in this paper. Both Sel-MAX and Random compute the resulting heuristic once, before each node is added to OPEN while LA* computes the heuristic lazily, in different steps of the search. In addition, both randomization and Sel-MAX save heuristic computations and thus reduce search time in many cases. However, they might be less informed than pure maximization and as a result expand a larger number of nodes.

We then combine the ideas of lazy heuristic evaluation and of trading off more node expansions for less heuristic computation time, into a new variant of LA* called rational lazy A* (RLA*). RLA* is based on rational meta-reasoning, and uses a myopic value-of-information criterion to decide whether to compute $h_2(n)$ or to bypass the computation of $h_2$ and expand $n$ immediately when $n$ re-emerges from OPEN. RLA* aims to reduce search time, even at the expense of more node expansions than $A_{MAX}^*$.

Empirical results on variants of the 15-puzzle and on numerous planning domains demonstrate that LA* and RLA* lead to state-of-the-art performance in many cases.

2 Lazy A*

Throughout this paper we assume for clarity that we have two available admissible heuristics, $h_1$ and $h_2$. Extension to multiple heuristics is straightforward, at least for LA*. Unless stated otherwise, we assume that $h_1$ is faster to compute than $h_2$ but that $h_2$ is weakly more informed, i.e., $h_1(n) \leq h_2(n)$ for the majority of the nodes $n$, although counter cases where $h_1(n) > h_2(n)$ are possible. We say that $h_2$ dominates $h_1$, if such counter cases do not exist and $h_2(n) \geq h_1(n)$ for all nodes $n$. We use $f_1(n)$ to denote $g(n) + h_1(n)$. Likewise, $f_2(n)$ denotes $g(n) + h_2(n)$, and $f_{max}(n)$ denotes...
Algorithm 1: Lazy $A^*$

Input: L AZY-$A^*$
1 Apply all heuristics to Start
2 Insert Start into OPEN
3 while OPEN not empty do
4 \[ n \leftarrow \text{best node from OPEN} \]
5 if Goal$(n)$ then
6 \[ \text{return trace}(n) \]
7 if $h_2$ was not applied to $n$ then
8 \[ \text{Apply } h_2 \text{ to } n \]
9 \[ \text{insert } n \text{ into OPEN} \]
10 continue /*next node in OPEN
11 foreach child $c$ of $n$ do
12 \[ \text{Apply } h_1 \text{ to } c. \]
13 \[ \text{insert } c \text{ into OPEN} \]
14 Insert $n$ into CLOSED
15 return FAILURE

\[
g(n) + \max(h_1(n), h_2(n)) \]

We denote the cost of the optimal solution by $C^*$. Additionally, we denote the computation time of $h_1$ and of $h_2$ by $t_1$ and $t_2$, respectively, and denote the overhead of an insert/pop operation in OPEN by $t_o$. Unless stated otherwise we assume that $t_2$ is much greater than $t_1 + t_o$. $A^*$ thus mainly aims to reduce computations of $h_2$.

The pseudo-code for $A^*$ is depicted as Algorithm 1, and is very similar to $A^*$. In fact, without lines 7–10, $A^*$ would be identical to $A^*$ using the $h_1$ heuristic. When a node $n$ is generated we only compute $h_1(n)$ and $n$ is added to OPEN (Lines 11–13), without computing $h_2(n)$ yet. When $n$ is first removed from OPEN (Lines 7–10), we compute $h_2(n)$ and reinsert it into OPEN, this time with $f_{\text{max}}(n)$.

It is easy to see that $A^*$ is as informative as $A^*_{\text{MAX}}$, in the sense that both $A^*_{\text{MAX}}$ and $A^*$ expand a node only if $f_{\text{max}}(n)$ is the best $f$-value in OPEN. Therefore, $A^*$ and $A^*_{\text{MAX}}$ generate and expand the same set of nodes, up to differences caused by break-taking.

In its general form $A^*$ generates many nodes that it does not expand. These nodes, called surplus nodes [Felner et al., 2012], are in OPEN when we expand the goal node with $f = C^*$. All nodes in OPEN with $f > C^*$ are surely surplus but some nodes with $f = C^*$ may also be surplus. The number of surplus nodes in OPEN can grow exponentially in the size of the domain, resulting in significant costs.

$A^*$ avoids $h_2$ computations for many of these surplus nodes. Consider a node $n$ that is generated with $f_1(n) > C^*$. This node is inserted into OPEN but will never reach the top of OPEN, as the goal node will be found with $f = C^*$. However, if OPEN breaks ties in favor of small $h$-values, the goal node with $f = C^*$ will be expanded as soon as it is generated and such savings of $h_2$ will be obtained for some nodes with $f_1 = C^*$ too. We refer to such nodes where we saved the computation of $h_2$ as good nodes. Other nodes, those with $f_1(n) < C^*$ (and some with $f_1(n) = C^*$) are called regular nodes as we apply both heuristics to them.

$A^*_{\text{MAX}}$ computes both $h_1$ and $h_2$ for all generated nodes, spending time $t_1 + t_2$ on all generated nodes. By contrast, for good nodes $A^*$ only spends $t_1$, and saves $t_2$. In the basic implementation of $A^*$ (as in algorithm 1) regular nodes are inserted into OPEN twice, first for $h_1$ (Line 13) and then for $h_2$ (Line 9) while good nodes only enter OPEN once (Line 13). Thus, $A^*$ has some extra overhead of OPEN operations for regular nodes. We distinguish between 3 classes of nodes:

1. expanded regular (ER) — nodes that were expanded after both heuristics were computed.
2. surplus regular (SR) — nodes for which $h_2$ was computed but are still in OPEN when the goal was found.
3. surplus good (SG) — nodes for which only $h_1$ was computed by $A^*$ when the goal was found.

<table>
<thead>
<tr>
<th>Alg</th>
<th>ER</th>
<th>SR</th>
<th>SG</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A^*_{\text{MAX}}$</td>
<td>$t_1 + t_2 + 2t_o$</td>
<td>$t_1 + t_2 + 4t_o$</td>
<td>$t_1 + t_2 + 3t_o$</td>
</tr>
<tr>
<td>$A^*$</td>
<td>$t_1 + t_2 + t_o$</td>
<td>$t_1 + t_2 + 3t_o$</td>
<td>$t_1 + t_2 + t_o$</td>
</tr>
</tbody>
</table>

The time overhead of $A^*_{\text{MAX}}$ and $A^*$ is summarized in Table 1. $A^*$ incurs more OPEN operations overhead, but saves $h_2$ computations for the SG nodes. When $t_2$ (boldface in table 1) is significantly greater than both $t_1$ and $t_o$, there is a clear advantage for $A^*$, as seen in the SG column.

3 Enhancements to Lazy $A^*$

Several enhancements can improve basic $A^*$ (Algorithm 1), which are effective especially if $t_1$ and $t_o$ are not negligible.

3.1 OPEN bypassing

Suppose node $n$ was just generated, and let $f_{\text{best}}$ denote the best $f$-value currently in OPEN. $A^*$ evaluates $h_1(n)$ and then inserts $n$ into OPEN. However, if $f_1(n) \leq f_{\text{best}}$, then $n$ will immediately reach the top of OPEN and $h_2$ will be computed. In such cases we can choose to compute $h_2(n)$ right away (after Line 12 in Algorithm 1), thus saving the overhead of inserting $n$ into OPEN and popping it again at the next step ($= 2 \times t_o$). For such nodes, $A^*$ is identical to $A^*_{\text{MAX}}$, as both heuristics are computed before the node is added to OPEN. This enhancement is called OPEN by-passing (OB). It is a reminiscent of the immediate expand technique applied to generated nodes [Stern et al., 2010; Sun et al., 2009]. The same technique can be applied when $n$ again reaches the top of OPEN when evaluating $h_2(n)$; if $f_2(n) \leq f_{\text{best}}$, expand $n$ right away. With OB, $A^*$ will incur the extra overhead of two OPEN cycles only for nodes $n$ where $f_1(n) > f_{\text{best}}$ and then later $f_2(n) > f_{\text{best}}$.

3.2 Heuristic bypassing

Heuristic bypassing (HBP) is a technique that allows $A^*_{\text{MAX}}$ to omit evaluating one of the two heuristics. HBP is probably used by many implementers, although to the best of our knowledge, it never appeared in the literature. HBP works for a node $n$ under the following two preconditions: (1) the operator between $n$ and its parent $p$ is bidirectional, and (2) both heuristics are consistent [Felner et al., 2011].

Let $C$ be the cost of the operator. Since the heuristic is consistent we know that $|h(p) - h(n)| \leq C$. Therefore, $h(p)$
provides the following upper- and lower-bounds on \( h(n) \) of 
\[ h(p) - C \leq h(n) \leq h(p) + C. \]
We thus denote 
\[ h(n) = h(p) - C \text{ and } \overline{h(n)} = h(p) + C. \]

To exploit HB in \( A^*_\text{MAX} \), we simply skip the computation of \( h_1(n) \) if \( h_1(n) \leq h_2(n) \), and vice versa. For example, consider node \( a \) in Figure 1, where all operators cost 1, \( h_1(a) = 6 \), and \( h_2(a) = 10 \). Based on our bounds \( h_1(b) \leq 7 \) and \( h_2(c) \geq 9 \). Thus, there is no need to check \( h_1(b) \) as \( h_2(b) \) will surely be the maximum. We can propagate these bounds further to node \( c \). \( h_2(c) = 8 \) while \( h_1(c) \leq 8 \) and again there is no need to evaluate \( h_1(c) \). Only in the last node \( d \) we get that \( h_2(d) = 8 \) but since \( h_1(c) \leq 9 \) then \( h_1(c) \) can potentially return the maximum and should thus be evaluated.

HBP can be combined in \( LA^* \) in a number of ways. We describe the variant we used. \( LA^* \) aims to avoid needless computations of \( h_2 \). Thus, when \( h_1(n) < h_2(n) \), we delay the computation of \( h_2(n) \) and add \( n \) to \( OPEN \) with \( f(n) = g(n) + h_2(n) \) and continue as in \( LA^* \). In this case, we saved \( t_1 \), delayed \( t_2 \) and used \( h_2(n) \) which is more informative than \( h_1(n) \). If, however, \( h_1(n) \geq h_2(n) \), then we compute \( h_1(n) \) and continue regularly. We note that HBP incurs the time and memory overheads of computing and storing four bounds and should only be applied if there is enough memory and if \( t_1 \) and especially \( t_2 \) are very large.

4 Rational Lazy \( A^* \)

\( LA^* \) offers us a very strong guarantee, of expanding the same set of nodes as \( A^*_\text{MAX} \). However, often we would prefer to expand more states, if it means reducing search time. We now present Rational Lazy \( A^* \) (RLA*), an algorithm which attempts to optimally manage this tradeoff.

Using principles of rational meta-reasoning [Russell and Wefald, 1991], theoretically every algorithm action (heuristic function evaluation, node expansion, open list operation) should be treated as an action in a sequential decision-making meta-level problem: actions should be chosen so as to achieve the minimal expected search time. However, the appropriate general meta-reasoning problem is extremely hard to define precisely and to solve optimally.

Therefore, we focus on just one decision type, made in the context of \( LA^* \), when \( n \) re-emerges from \( OPEN \) (Line 7). We have two options: (1) Evaluate the second heuristic \( h_2(n) \) and add the node back to \( OPEN \) (Lines 7-10) like \( LA^* \), or (2) bypass the computation of \( h_2(n) \) and expand \( n \) right way (Lines 11-13), thereby saving time by not computing \( h_2 \), at the risk of additional expansions and evaluations of \( h_1 \). In order to choose rationally, we define a criterion based on value of information (VOI) of evaluating \( h_2(n) \) in this context.

The only addition of RLA* to \( LA^* \) is the option to bypass \( h_2 \) computations (Lines 7-10). Suppose that we choose to compute \( h_2 \) — this results in one of the following outcomes:

1: \( n \) is still expanded, either now or eventually.
2: \( n \) is re-inserted into \( OPEN \), and the goal is found without ever expanding \( n \).

Computing \( h_2 \) is helpful only in outcome 2, where potential time savings are due to pruning a search subtree at the expense of the time \( t_2(n) \). However, whether outcome 2 takes place after a given state is not known to the algorithm until the goal is found, and the algorithm must decide whether to evaluate \( h_2 \) according to what it believes to be the probability of each of the outcomes. We derive a rational policy for when to evaluate \( h_2 \), under the myopic assumption that the algorithm continues to behave like \( LA^* \) afterwards (i.e., it will never again consider bypassing the computation of \( h_2 \)).

The time wasted by being sub-optimal in deciding whether to evaluate \( h_2 \) is called the regret of the decision. If \( h_2(n) \) is not helpful and we decide to compute it, the effort for evaluating \( h_2(n) \) turns out to be wasted. On the other hand, if \( h_2(n) \) is helpful but we decide to bypass it, we needlessly expand \( n \). Due to the myopic assumption, RLA* would evaluate both \( h_1 \) and \( h_2 \) for all successors of \( n \).

<table>
<thead>
<tr>
<th>( h_2 ) helpful</th>
<th>Bypass ( h_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( t_e + (b(n) - 1)t_d )</td>
</tr>
<tr>
<td>( h_2 ) not helpful</td>
<td>( t_d )</td>
</tr>
</tbody>
</table>

Table 2: Regret in Rational Lazy \( A^* \)

Table 2 summarizes the regret of each possible decision, for each possible future outcome; each column in the table represents a decision, while each row represents a future outcome. In the table, \( t_d \) is the time to compute \( h_2 \) and re-insert \( n \) into \( OPEN \) thus delaying the expansion of \( n \), \( t_e \) is the time to remove \( n \) from \( OPEN \), expand \( n \), evaluate \( h_1 \) on each of the \( b(n) \) (“local branching factor”) children \{\( n' \)\} of \( n \), and insert \{\( n' \)\} into the open list. Computing \( h_2 \) needlessly wastes time \( t_d \). Bypassing \( h_2 \) computation when \( h_2 \) would have been helpful wastes \( t_e + b(n)t_d \) time, but because computing \( h_2 \) would have cost \( t_d \), the regret is \( t_e + (b(n) - 1)t_d \).

Let us denote the probability that \( h_2 \) is helpful by \( p_h \). The expected regret of computing \( h_2 \) is thus \((1-p_h)t_d \). On the other hand, the expected regret of bypassing \( h_2 \) is \( p_h(t_e + (b(n) - 1)t_d) \). As we wish to minimize the expected regret, we should thus evaluate \( h_2 \) just when:

\[
(1-p_h)t_d < p_h(t_e + (b(n) - 1)t_d)
\]

or equivalently:

\[
(1-b(n)p_h)t_d < p_h t_e
\]

If \( p_h b(n) \geq 1 \), then the expected regret is minimized by always evaluating \( h_2 \), regardless of the values of \( t_d \) and \( t_e \). In these cases, RLA* cannot be expected to do better than \( LA^* \). For example, in the 15-puzzle and its variants, the effective branching factor is \( \approx 2 \). Therefore, if \( h_2 \) is expected to be helpful for more than half of the nodes \( n \) on which \( LA^* \) evaluates \( h_2(n) \), then one should simply use \( LA^* \).

For \( p_h b(n) \leq 1 \), the decision of whether to evaluate \( h_2 \) depends on the values of \( t_d \) and \( t_e \):

\[
\text{evaluate } h_2 \text{ if } t_d < \frac{p_h}{1-p_h b(n)} t_e \tag{3}
\]

Denote by \( t_c \) the time to generate the children of \( n \). Then:

\[
t_d = t_2 + t_o \tag{4}
\]

\[
t_e = t_o + t_c + b(n)t_1 + b(n)t_o \tag{4}
\]

By substituting (4) into (3), obtain: evaluate \( h_2 \) if:

\[
t_2 + t_o < \frac{p_h [t_c + (b(n)t_1 + (b(n) + 1)t_o]}{1-p_h b(n)} \tag{5}
\]
The factor \( \frac{p_h}{1 - p_h(b(n))} \) depends on the potentially unknown probability \( p_h \), making it difficult to reach the optimum decision. However, if our goal is just to do better than \( LA^* \), then it is safe to replace \( p_h \) by an upper bound on \( p_h \). Note that the values \( p_h, t_1, t_2, t_e \) may actually be variables that depend in complicated ways on the state of the search. Despite that, the very crude model we use, assuming that they are setting-specific constants, is sufficient to achieve improved performance, as shown in Section 5.

We now turn to implementation-specific estimation of the runtimes. \( OPEN \) in \( A^* \) is frequently implemented as a priority queue, and thus we have, approximately, \( t_o = \tau \log N_o \) for some \( \tau \), where the size of \( OPEN \) is \( N_o \). Evaluating \( h_1 \) is cheap in many domains, as is the case with Manhattan Distance (MD) in discrete domains, \( t_o \) is the most significant part of \( t_e \). In such cases, rule (5) can be approximated as 6:

\[
evaluate h_2 \text{ if } t_2 < \frac{\tau p_h}{1 - p_h(b(n))} (b(n) + 1) \log N_o
\]

(6)

Rule (6) recommends to evaluate \( h_2 \) mostly at late stages of the search, when the open list is large, and in nodes with a higher branching factor.

In other domains, such as planning, both \( t_3 \) and \( t_2 \) are significantly greater than both \( t_o \) and \( t_e \), and terms not involving \( t_3 \) or \( t_2 \) can be dropped from (5), resulting in Rule (7):

\[
evaluate h_2 \text{ if } \frac{t_2}{t_1} < \frac{p_h(b(n))}{1 - p_h(b(n))}
\]

(7)

The right hand side of (7) grows with \( b(n) \), and here it is beneficial to evaluate \( h_2 \) only for nodes with a sufficiently large branching factor.

5 Empirical evaluation

We now present our empirical evaluation of \( LA^* \) and \( RLA^* \), on variants of the 15-puzzle and on planning domains.

5.1 Weighted 15 puzzle

We first provide evaluations on the weighted 15-puzzle variant [Thayer and Ruml, 2011], where the cost of moving each tile is equal to the number on the tile. We used a subset of 36 problem instances (out of the 100 instances of Korf (1985)) which could be solved with 2Gb of RAM and 15 minutes timeout using the Weighted Manhattan heuristic (WMD) for \( h_1 \). As the expensive and informative heuristic \( h_2 \) we use a heuristic based on lookaheads [Stern et al., 2010]. Given a bound \( d \) we applied a bounded depth-first search from a node \( n \) and backtracked when we reached leaf nodes \( l \) for which \( g(l) + WMD(l) > g(n) + WMD(n) + d \). \( f \)-values from leaves were propagated to \( n \).

Table 3 presents the results averaged on all instances solved. The runtimes are reported relative to the time of \( A^* \) with WMD (with no lookahead), which generated 1,886,397 nodes (not reported in the table). The first 3 columns of Table 3 show the results for \( A^* \) with the lookahead heuristic for different lookahead depths. The best time is achieved for lookahead 6 (0.588 compared to \( A^* \) with WMD). The fact that the time does not continue to decrease with deeper lookaheads is clearly due to the fact that although the resulting heuristic improves as a function of lookahead depth (expanding and generating fewer nodes), the increasing overheads of computing the heuristic eventually outweigh savings due to fewer expansions.

The next 4 columns show the results for \( LA^* \) with WMD as \( h_1 \), lookahead as \( h_2 \), for different lookahead depths. The \( Good1 \) column presents the number of nodes where \( LA^* \) saved the computation of \( h_2 \) while the \( h_2 \) column presents the number of nodes where \( h_2 \) was computed. Roughly 28\% of nodes were \( Good1 \) and since \( t_2 \) was the most dominant time cost, most of this saving is reflected in the timing results. The best results are achieved for lookahead 8, with a runtime of 0.527 compared to \( A^* \) with WMD.

The final columns show the results of \( RLA^* \), with the values of \( \tau, p_h, t_2 \) calibrated for each lookahead depth using a small subset of problem instances. The \( Good2 \) column counts the number of times that \( RLA^* \) decided to bypass the \( h_2 \) computation. Observe that \( RLA^* \) outperforms \( LA^* \), which in turn outperforms \( A^* \), for most lookahead depths. The lowest time with \( RLA^* \) (0.371 of \( A^* \) with WMD) was obtained for lookahead 10. That is achieved as the more expensive \( h_2 \) heuristic is computed less often, reducing its effective computational overhead, with some adverse effect in the number of expanded nodes. Although \( LA^* \) expanded fewer nodes, \( RLA^* \) performed much fewer \( h_2 \) computations as can be seen in the table, resulting in decreased overall runtimes.

5.2 Planning domains

We implemented \( LA^* \) and \( RLA^* \) on top of the Fast Downward planning system [Helmert, 2006], and experimented with two state of the art heuristics: the admissible landmarks heuristic \( h_{LA} \) (used as \( h_1 \)) [Karpas and Domshlak, 2009], and the landmark cut heuristic \( h_{LMCUT} \) [Helmert and Domshlak, 2009] (used as \( h_2 \)). On average, \( h_{LMCUT} \) computation is 8.36 times more expensive than \( h_{LA} \) computation. We did not implement HBP in the planning domains as the heuristics we use are not consistent and in general the operators are not invertible. We also did not implement OB, as the cost of \( OPEN \) operations in planning is trivial compared to the cost of heuristic evaluations.

We experimented with all planning domains without conditional effects and derived predicates (which the heuristics we used do not support) from previous IPCs. We compare the performance of \( LA^* \) and \( RLA^* \) to that of \( A^* \) using each
Instead, we vary our estimate of \( p \) value for the same domain are of varying size, and thus getting a single heuristics, which we measure as search progresses.

The rule involves two unknown quantities: \( h \) and \( k \), the ratio between heuristic computations times, and \( p \), the probability that \( h2 \) is helpful. Estimating \( \frac{h}{k} \) is quite easy — we simply use the average computation times of both heuristics, which we measure as search progresses.

Estimating \( p \) is not as simple. While it is possible to empirically determine the best value for \( p \), as done for the weighted 15 puzzle, this does not fit the paradigm of domain-independent planning. Furthermore, planning domains are very different from each other, and even problem instances in the same domain are of varying size, and thus getting a single value for \( p \) works well for many problems is difficult. Instead, we vary our estimate of \( p \) adaptively during search. To understand this estimate, first note that if \( n \) is a node at which \( h2 \) was helpful, then we computed \( h2 \) for \( n \), but did not extend it. Thus, we can use the number of states for which we computed \( h2 \) that were not yet expanded (denoted by \( A \)), divided by the number of states for which we computed \( h2 \) (denoted by \( B \)), as an approximation of \( p \). However, \( \frac{h}{k} \) is not likely to be a stable estimate at the beginning of the search, as \( A \) and \( B \) are both small numbers. To overcome this problem, we “imagine” we have observed \( k \) examples, which give us an estimate of \( p_{*} = \frac{p_{init}}{k} \), and use a weighted average between these \( k \) examples, and the observed examples — that is, we estimate \( p \) by \( \frac{h}{k} \cdot B + p_{init} \cdot k / (B + k) \). In our empirical evaluation, we used \( k = 1000 \) and \( p_{init} = 0.5 \).

Table 4 depicts the experimental results. The leftmost part of the table shows the number of solved problems in each domain. As the table demonstrates, \( RLA* \) solves the most problems, and \( LA^* \) solves the same number of problems as \( Sel-MAX \). Thus, both \( LA^* \) and \( RLA^* \) are state-of-the-art in cost-optimal planning. Looking more closely at the results, note that \( Sel-MAX \) solves 10 more problems than \( LA^* \) and \( RLA^* \) in the freecell domain. Freecell is one of only three domains in which \( h \) is more informed than \( h_{LMCUT} \) (the other two are nonmystry-opt11 and visitall-opt11), violating the basic assumptions behind \( LA^* \) that \( h2 \) is more informed than \( h1 \). If we ignore these domains, both \( LA^* \) and \( RLA^* \) solve more problems than \( Sel-MAX \).

The middle part of the Table 4 shows the geometric mean of planning time in each domain, over the commonly solved problems (i.e., those that were solved by all 6 methods). \( RLA^* \) is the fastest overall, with \( LA^* \) second. It is important to note that both \( LA^* \) and \( RLA^* \) are very robust, and even in cases where they are not the best they are never too far from the best. For example, consider the miconic domain. Here, \( h_{LMCUT} \) is very informative and thus the variant that only computed \( h_{LMCUT} \) is the best choice (but a bad choice overall). Observe that both \( LA^* \) and \( RLA^* \) saved \( 86\% \) of \( h_{LMCUT} \) computations, and were very close to the best algorithm in this extreme case. In contrast, the other algorithms that consider

<table>
<thead>
<tr>
<th>Domain</th>
<th>Problems Solved</th>
<th>Planning Time (seconds)</th>
<th>GOOD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Problems Solved</td>
<td>Planning Time (seconds)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( h_{LMCUT} )</td>
<td>( h_{LMCUT} )</td>
<td>( h_{LMCUT} )</td>
</tr>
</tbody>
</table>

Table 4: Planning Domains — Number of Problems Solved, Total Planning Time, and Fraction of Good Nodes
both heuristics (max and Sel-MAX) performed very poorly here (more than three times slower).

The rightmost part of Table 4 shows the average fraction of nodes for which LA* and RLA* did not evaluate the more expensive heuristic, \( h_{LMCUT} \), over the problems solved by both these methods. This is shown in the good columns. Our first observation is that this fraction varies between different domains, indicating why LA* works well in some domains, but not in others. Additionally, we can see that in domains where there is a difference in this number between LA* and RLA*, RLA* usually performs better in terms of time. This indicates that when RLA* decides to skip the computation of the expensive heuristic, it is usually the right decision.

Finally, Table 5 shows the total number of expanded and generated states over all commonly solved problems. LA* is indeed as informative as \( A_{MAX} \) (the small difference is caused by tie-breaking), while RLA* is a little less informed and expands slightly more nodes. However, RLA* is much more informative than its “intelligent” competitor - Sel-MAX, as these are the only two algorithms in our set which selectively omit some heuristic computations. RLA* generated almost half of the nodes compared to Sel-MAX, suggesting that its decisions are better.

### 5.3 Limitations of LA*: 15 puzzle example

Some domains and heuristic settings will not achieve time speedup with LA*. An example is the regular, unweighted 15-puzzle. Results for \( A_{MAX} \) and LA* with and without HBP on the 15-puzzle are reported in Table 6. HBP1 (HBP2) count the number of nodes where HBP pruned the need to compute \( h_1 \) (resp. \( h_2 \)). OB is the number of nodes where OB was helpful. Bad is the number of nodes that went through two OPEN cycles. Finally, Good is the number of nodes where computation of \( h_2 \) was saved due to LA*.

In the first experiment, Manhattan distance (MD) was divided into two heuristics: \( \Delta x \) and \( \Delta y \) used as \( h_1 \) and \( h_2 \). Results are averaged over 100 random instances with average solution depth of 26.66. As seen from the first two lines, HBP when applied on top of \( A_{MAX} \) saved about 36% of the heuristic evaluations. Next are results for LA* and LA*+HBP. Many nodes are pruned by HBP, or OB. The number of good nodes dropped from 28% (Line 3) to as little as 11% when HBP was applied. Timing results (in ms) show that all variants performed equally. The reason is that the time overhead of the \( \Delta x \) and \( \Delta y \) heuristics is very small so the saving on these 28% or 11% of nodes was not significant to outweigh the HBP overhead of handling the upper and lower bounds.

The next experiment is with MD as \( h_1 \) and a variant of the additive 7-8 PDBs [Korf and Felner, 2002], as \( h_2 \). Here we can observe an interesting phenomenon. For LA*, most nodes were caught by either HBP (when applicable) or by OB. Only 4% of the nodes were good nodes. The reason is that the 7-8 PDB heuristic always dominates MD and is always the maximum among the two. Thus, 7-8 PDB was needed at early stages (e.g. by OB) and MD itself almost never caused nodes to be added to OPEN and remain there until the goal was found.

These results indicate that on such domains, LA* has limited merit. Due to uniform operator cost and the heuristics being consistent and simple to compute, very little space is left for improvement with good nodes. We thus conclude that LA* is likely to be effective when there is significant difference between \( t_1 \) and \( t_2 \), and/or operators that are not bidirectional and/or with non-uniform costs, allowing for more good nodes and significant time saving.

### 6 Conclusion

We discussed two schemes for decreasing heuristic evaluation times. LA* is very simple to implement and is as informative as \( A_{MAX} \). LA* can significantly speed up the search, especially if \( t_2 \) dominates the other time costs, as seen in weighted 15 puzzle and planning domains. Rational LA* allows additional cuts in \( h_2 \) evaluations, at the expense of being less informative than \( A_{MAX} \). However, due to a rational tradeoff, this allows for an additional speedup, and Rational LA* achieves the best overall performance in our domains.

RLA* is simpler to implement than its direct competitor, Sel-MAX, but its decision can be more informed. When RLA* has to decide whether to compute \( h_2 \) for some node \( n \), it already knows that \( f_1(n) \leq C^* \). By contrast, although Sel-MAX uses a much more complicated decision rule, it makes its decision when \( n \) is first generated, and does not know whether \( h_1 \) will be informative enough to prune \( n \). Rational LA* outperforms Sel-MAX in our planning experiments.

RLA* and its analysis can be seen as an instance of the rational meta-reasoning framework [Russell and Wefald, 1991]. While this framework is very general, it is extremely hard to apply in practice. Recent work exists on meta-reasoning in DFS algorithms for CSP) [Tolpin and Shimony, 2011] and in Monte-Carlo tree search [Hay et al., 2012]. This paper applies these methods successfully to a variant of A*.

There are numerous other ways to use rational meta-reasoning to improve A*, starting from generalizing RLA* to handle more than two heuristics, to using the meta-level to control decisions in other variants of A*. All these potential extensions provide fruitful ground for future work.

### 7 Acknowledgments

The research was supported by the Israeli Science Foundation (ISF) under grant #305/09 to Ariel Felner and Eyal Shimony and by Lynne and William Frankel Center for Computer Science.
References


