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 optmizations 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* (Sel-MAX), an online learning scheme which chooses one heuristic to compute at each state. Sel-MAX 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 pathmax* (BPMX) 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: LAZY-A*
1  Apply all heuristics to Start
2  Insert Start into OPEN
3  while OPEN not empty do
4      n ← best node from OPEN
5      if Goal(n) then
6          return trace(n)
7      if h2 was not applied to n then
8          Apply h2 to n
9          insert n into OPEN
10         continue        //next node in OPEN
11     foreach child c of n do
12         Apply h1 to c.
13         insert c 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 $t_1 + t_o$. LA^* thus mainly aims to reduce computations of h_2.

The pseudo-code for LA^* is depicted as Algorithm 1, and is very similar to A^*. In fact, without lines 7 – 10, LA^* 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 $f_{max}(n)$.

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

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.

LA^* 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 $f = C^*$. In fact, 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^*_{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 LA^* only spends t_1, and saves t_2. In the basic

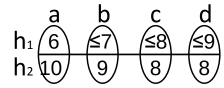

Figure 1: Example of HBP

implementation of LA^* (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, LA^* 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 LA^* when the goal was found.

Alg	ER	SR	SG
A^*_{MAX}	$t_1 + \mathbf{t_2} + 2t_o$	$t_1 + \mathbf{t_2} + t_o$	$t_1 + \mathbf{t_2} + t_o$
LA^*	$t_1 + \mathbf{t_2} + 4t_o$	$t_1 + \mathbf{t_2} + 3t_o$	$t_1 + t_o$

Table 1: Time overhead for A^*_{MAX} and for LA^*

The time overhead of A^*_{MAX} and LA^* is summarized in Table 1. LA^* 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 LA^*, as seen in the SG column.

3 Enhancements to Lazy A^*

Several enhancements can improve basic LA^* (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_{best} denote the best f-value currently in OPEN. LA^* evaluates $h_1(n)$ and then inserts n into OPEN. However, if $f_1(n) \leq f_{best}$, 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, LA^* is identical to A^*_{MAX}, as both heuristics are computed before the node is added to OPEN. This enhancement is called *OPEN bypassing* (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 $f_2(n) \leq f_{best}$, expand n right away. With OB, LA^* will incur the extra overhead of two OPEN cycles only for nodes n where $f_1(n) > f_{best}$ and then later $f_2(n) > f_{best}$.

3.2 Heuristic bypassing

Heuristic bypassing (HBP) is a technique that allows A^*_{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 $\underline{h(n)} = h(p) - C$ and $\overline{h(n)} = h(p) + C$.

To exploit HBP in A^*_{MAX}, we simply skip the computation of $h_1(n)$ if $\overline{h_1(n)} \leq \underline{h_2(n)}$, 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 $\overline{h_1(n)} < \underline{h_2(n)}$, we delay the computation of $h_2(n)$ and add n to OPEN with $f(n) = g(n) + \underline{h_2(n)}$ and continue as in LA^*. In this case, we saved t_1, delayed t_2 and used $\underline{h_2(n)}$ which is more informative than $h_1(n)$. If, however, $\overline{h_1(n)} \geq \underline{h_2(n)}$, 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^*_{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.

	Compute h_2	Bypass h_2
h_2 helpful	0	$t_e + (b(n) - 1)t_d$
h_2 not helpful	t_d	0

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 to time 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:

$$(1 - p_h)t_d < p_h(t_e + (b(n) - 1)t_d) \tag{1}$$

or equivalently:

$$(1 - b(n)p_h)t_d < p_h t_e \tag{2}$$

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 ≈ 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 $p_h b(n) < 1$, the decision of whether to evaluate h_2 depends on the values of t_d and t_e:

$$\textbf{evaluate } h_2 \textbf{ if } t_d < \frac{p_h}{1 - p_h b(n)} t_e \tag{3}$$

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

$$\begin{aligned} t_d &= t_2 + t_o \\ t_e &= t_o + t_c + b(n)t_1 + b(n)t_o \end{aligned} \tag{4}$$

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

$$t_2 + t_o < \frac{p_h \left[t_c + b(n)t_1 + (b(n) + 1)t_o\right]}{1 - p_h b(n)} \tag{5}$$

	A^*		LA^*				RLA^*(Using Eq. 6)				
lookahead	generated	time	generated	Good1	h_2	time	generated	Good1	Good2	h_2	time
2	1,206,535	0.707	1,206,535	391,313	815,213	0.820	1,309,574	475,389	394,863	439,314	0.842
4	1,066,851	0.634	1,066,851	333,047	733,794	0.667	1,169,020	411,234	377,019	380,760	0.650
6	889,847	**0.588**	889,847	257,506	632,332	0.533	944,750	299,470	239,320	405,951	0.464
8	740,464	0.648	740,464	196,952	543,502	**0.527**	793,126	233,370	218,273	341,476	0.377
10	611,975	0.843	611,975	145,638	466,327	0.671	889,220	308,426	445,846	134,943	**0.371**
12	**454,130**	0.927	**454,130**	95,068	359,053	0.769	807,846	277,778	428,686	**101,378**	0.429

Table 3: Weighted 15 puzzle: comparison of $A^*_{\max}$, Lazy A^*, and Rational Lazy A^*

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_c 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 τ, 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:

$$\textbf{evaluate } h_2 \textbf{ if } t_2 < \frac{\tau p_h}{1 - p_h b(n)}(b(n)+1)\log N_o \qquad (6)$$

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_1 and t_2 are significantly greater than both t_o and t_c, and terms not involving t_1 or t_2 can be dropped from (5), resulting in Rule (7):

$$\textbf{evaluate } h_2 \textbf{ if } \frac{t_2}{t_1} < \frac{p_h b(n)}{1 - p_h b(n)} \qquad (7)$$

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 outweights 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 τ, 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

	Problems Solved						Planning Time (seconds)						GOOD	
Domain	h_{LA}	lmcut	max	selmax	LA^*	RLA^*	h_{LA}	lmcut	max	selmax	LA^*	RLA^*	LA^*	RLA^*
airport	25	24	26	25	**29**	**29**	**0.29**	0.57	0.5	0.33	0.38	0.38	0.48	0.67
barman-opt11	**4**	0	0	0	0	3	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
blocks	26	27	27	27	**28**	**28**	1.0	**0.65**	0.73	0.81	0.67	0.67	0.19	0.21
depot	**7**	6	5	5	6	6	**2.27**	2.69	3.17	3.14	2.73	2.75	0.06	0.06
driverlog	10	**12**	**12**	**12**	**12**	**12**	2.65	**0.29**	0.33	0.36	0.3	0.31	0.09	0.09
elevators-opt08	12	**18**	17	17	17	17	14.14	4.21	4.84	4.85	**3.56**	3.64	0.27	0.27
elevators-opt11	10	**14**	**14**	**14**	**14**	**14**	26.97	8.03	9.28	9.28	**6.64**	6.78	0.28	0.28
floortile-opt11	2	**6**	**6**	**6**	**6**	**6**	8.52	**0.44**	0.6	0.58	0.5	0.52	0.02	0.02
freecell	**54**	10	36	51	41	41	**0.16**	7.34	0.22	0.24	0.18	0.18	0.86	0.86
grid	**2**	**2**	1	**2**	**2**	**2**	**0.1**	0.16	0.18	0.34	0.15	0.15	0.17	0.17
gripper	**7**	6	6	6	6	6	**0.84**	1.53	2.24	2.2	1.78	1.25	0.01	0.4
logistics00	**20**	17	16	**20**	19	19	**0.23**	0.57	0.68	0.27	0.47	0.47	0.51	0.51
logistics98	3	**6**	**6**	**6**	**6**	**6**	0.72	**0.1**	0.1	0.11	**0.1**	**0.1**	0.07	0.07
miconic	**141**	140	140	**141**	**141**	**141**	**0.13**	0.55	0.58	0.57	0.16	0.16	0.87	0.88
mprime	16	20	20	20	**21**	20	1.27	0.5	0.51	0.5	**0.44**	0.45	0.25	0.25
mystery	13	**15**	**15**	**15**	**15**	**15**	0.71	**0.35**	0.38	0.43	0.36	0.37	0.3	0.3
nomystery-opt11	**18**	14	16	**18**	**18**	**18**	**0.18**	1.29	0.58	0.25	0.33	0.33	0.72	0.72
openstacks-opt08	15	**16**	14	15	**16**	**16**	2.88	**1.68**	3.89	3.03	2.62	2.64	0.44	0.45
openstacks-opt11	10	**11**	9	10	**11**	**11**	13.59	**6.96**	19.8	14.44	12.03	12.06	0.43	0.43
parcprinter-08	14	**18**	**18**	**18**	**18**	**18**	0.92	**0.36**	0.37	0.38	0.37	0.37	0.17	0.26
parcprinter-opt11	10	**13**	**13**	**13**	**13**	**13**	2.24	**0.56**	0.6	0.61	0.58	0.59	0.14	0.17
parking-opt11	1	1	1	**3**	2	2	9.74	22.13	17.85	7.11	**6.33**	6.43	0.64	0.64
pathways	4	**5**	**5**	**5**	**5**	**5**	0.5	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	0.1	0.12
pegsol-08	**27**	**27**	**27**	**27**	**27**	**27**	1.01	**0.84**	1.2	1.1	1.06	0.95	0.04	0.42
pegsol-opt11	**17**	**17**	**17**	**17**	**17**	**17**	4.91	**3.63**	5.85	5.15	4.87	4.22	0.04	0.38
pipesworld-notankage	**16**	15	15	**16**	15	15	**0.5**	1.48	1.12	0.85	0.9	0.91	0.42	0.42
pipesworld-tankage	**11**	8	9	9	9	9	**0.36**	2.24	1.02	0.47	0.69	0.71	0.62	0.62
psr-small	**49**	48	48	**49**	48	48	**0.15**	0.2	0.21	0.19	0.19	0.18	0.17	0.49
rovers	6	**7**	**7**	**7**	**7**	**7**	0.74	**0.41**	0.45	0.45	0.41	0.42	0.47	0.47
scanalyzer-08	6	**13**	**13**	**13**	**13**	**13**	0.37	**0.25**	0.27	0.27	0.26	0.26	0.06	0.06
scanalyzer-opt11	3	**10**	**10**	**10**	**10**	**10**	**0.59**	0.64	0.75	0.73	0.67	0.68	0.05	0.05
sokoban-opt08	23	25	25	24	26	**27**	3.94	1.76	2.19	2.96	1.9	**1.32**	0.04	0.4
sokoban-opt11	**19**	**19**	**19**	18	**19**	**19**	7.26	2.83	3.66	5.19	3.1	**2.02**	0.03	0.46
storage	14	**15**	14	14	**15**	**15**	**0.36**	0.44	0.49	0.45	0.44	0.42	0.21	0.28
tidybot-opt11	**14**	12	12	12	12	12	**3.03**	16.32	17.55	9.35	15.67	15.02	0.11	0.18
tpp	**6**	**6**	**6**	**6**	**6**	**6**	0.39	**0.22**	0.23	0.23	0.22	0.22	0.32	0.4
transport-opt08	**11**	**11**	**11**	**11**	**11**	**11**	1.45	**1.24**	1.41	1.54	1.25	1.26	0.04	0.04
transport-opt11	**6**	**6**	**6**	**6**	**6**	**6**	12.46	**8.5**	10.38	11.13	8.56	8.61	0.0	0.0
trucks	7	**9**	**9**	**9**	**9**	**9**	4.49	**1.34**	1.52	1.44	1.41	1.42	0.07	0.07
visitall-opt11	12	10	**13**	12	**13**	**13**	0.2	0.34	0.19	**0.18**	0.18	0.18	0.38	0.38
woodworking-opt08	12	**16**	**16**	**16**	**16**	**16**	1.08	0.71	0.75	0.75	**0.66**	0.67	0.56	0.56
woodworking-opt11	7	**11**	**11**	**11**	**11**	**11**	5.7	2.86	3.15	3.01	**2.55**	2.58	0.52	0.52
zenotravel	8	**11**	**11**	**11**	**11**	**11**	0.38	**0.14**	0.14	0.14	0.14	0.14	0.17	0.19
OVERALL	698	697	722	747	747	**750**	1.18	0.98	0.98	0.89	0.79	**0.77**	0.27	0.34

Table 4: Planning Domains — Number of Problems Solved, Total Planning Time, and Fraction of Good Nodes

of the heuristics individually, as well as to their max-based combination, and their combination using selective-max (Sel-MAX) [Domshlak *et al.*, 2012]. The search was limited to 6GB memory, and 5 minutes of CPU time on a single core of an Intel E8400 CPU with 64-bit Linux OS.

When applying RLA^* in planning domains we evaluate rule (7) at every state. This rule involves two unknown quantities: $\frac{t_2}{t_1}$, the ratio between heuristic computations times, and p_h, the probability that h_2 is helpful. Estimating $\frac{t_2}{t_1}$ is quite easy — we simply use the average computation times of both heuristics, which we measure as search progresses.

Estimating p_h is not as simple. While it is possible to empirically determine the best value for p_h, 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_h which works well for many problems is difficult. Instead, we vary our estimate of p_h adaptively during search. To understand this estimate, first note that if n is a node at which h_2 was helpful, then we computed h_2 for n, but did not expand n. Thus, we can use the number of states for which we computed h_2 that were not yet expanded (denoted by A), divided by the number of states for which we computed h_2 (denoted by B), as an approximation of p_h. However, $\frac{A}{B}$ 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 $p_h = p_{init}$, and use a weighted average between these k examples, and the observed examples — that is, we estimate p_h by $(\frac{A}{B} \cdot B + p_{init} \cdot k)/(B + k)$. 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_{LA} is more informed than h_{LMCUT} (the other two are nomystery-opt11 and visitall-opt11), violating the basic assumptions behind LA^* that h_2 is more informed than h_1. 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_{LA} is very informative and thus the variant that only computed h_{LA} 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

	Expanded	Generated
h_{LA}	183,320,267	1,184,443,684
lmcut	23,797,219	114,315,382
A^*_{MAX}	22,774,804	108,132,460
selmax	54,557,689	193,980,693
LA^*	22,790,804	108,201,244
RLA^*	25,742,262	110,935,698

Table 5: Total Number of Expanded and Generated States

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, unweighed 15-puzzle. Results for A^*_{MAX} and LA^* with and without HBP on the 15-puzzle are reported in Table 6. HBP_1 (HBP_2) 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: Δx and Δy 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 Δx and Δy 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.

Alg.	Generated	HBP1	HBP2	OB	Bad	Good	time
$h1 = \Delta X, h2 = \Delta Y$, Depth = 26.66							
A*	1,085,156	0	0	0	0	0	**415**
A*+HBP	1,085,156	216,689	346,335	0	0	0	417
LA*	1,085,157	0	0	734,713	37,750	312,694	417
LA*+HBP	1,085,157	140,746	342,178	589,893	37,725	115,361	416
$h1 =$ Manhattan distance, $h2 =$ 7-8 PDB, Depth 52.52							
A*	43,741	0	0	0	0	0	34.7
A*+HBP	43,804	30,136	1,285	0	0	0	33.6
LA*	43,743	0	0	42,679	47	1,017	34.2
LA*+HBP	43,813	7,669	1,278	42,271	21	243	**33.3**

Table 6: Results on the 15 puzzle

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 informed 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

[Dechter and Pearl, 1985] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM*, 32(3):505–536, 1985.

[Domshlak *et al.*, 2012] Carmel Domshlak, Erez Karpas, and Shaul Markovitch. Online speedup learning for optimal planning. *JAIR*, 44:709–755, 2012.

[Felner *et al.*, 2011] A. Felner, U. Zahavi, R. Holte, J. Schaeffer, N. Sturtevant, and Z. Zhang. Inconsistent heuristics in theory and practice. *Artificial Intelligence*, 175(9-10):1570–1603, 2011.

[Felner *et al.*, 2012] A. Felner, M. Goldenberg, G. Sharon, R. Stern, T. Beja, N. R. Sturtevant, J. Schaeffer, and Holte R. Partial-expansion a* with selective node generation. In *AAAI*, pages 471–477, 2012.

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

[Hay *et al.*, 2012] Nicholas Hay, Stuart Russell, David Tolpin, and Solomon Eyal Shimony. Selecting computations: Theory and applications. In Nando de Freitas and Kevin P. Murphy, editors, *UAI*, pages 346–355. AUAI Press, 2012.

[Helmert and Domshlak, 2009] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *ICAPS*, pages 162–169, 2009.

[Helmert, 2006] Malte Helmert. The Fast Downward planning system. *JAIR*, 26:191–246, 2006.

[Karpas and Domshlak, 2009] Erez Karpas and Carmel Domshlak. Cost-optimal planning with landmarks. In *IJCAI*, pages 1728–1733, 2009.

[Korf and Felner, 2002] R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134(1-2):9–22, 2002.

[Korf, 1985] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[Russell and Wefald, 1991] Stuart Russell and Eric Wefald. Principles of metereasoning. *Artificial Intelligence*, 49:361–395, 1991.

[Stern *et al.*, 2010] Roni Stern, Tamar Kulberis, Ariel Felner, and Robert Holte. Using lookaheads with optimal best-first search. In *AAAI*, pages 185–190, 2010.

[Sun *et al.*, 2009] X. Sun, W. Yeoh, P. Chen, and S. Koenig. Simple optimization techniques for A*-based search. In *AAMAS*, pages 931–936, 2009.

[Thayer and Ruml, 2011] Jordan T. Thayer and Wheeler Ruml. Bounded suboptimal search: A direct approach using inadmissible estimates. In *Proceedings of the Twenty-second International Joint Conference on Artificial Intelligence (IJCAI-11)*, 2011.

[Tolpin and Shimony, 2011] David Tolpin and Solomon Eyal Shimony. Rational deployment of CSP heuristics. In Toby Walsh, editor, *IJCAI*, pages 680–686. IJCAI/AAAI, 2011.

[Zhang and Bacchus, 2012] Lei Zhang and Fahiem Bacchus. Maxsat heuristics for cost optimal planning. In *AAAI*, 2012.