ARTICLE IN PRESS
ARTINT:3039

Artificial Intelligence ••• (••••) •••–•••

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Rational deployment of multiple heuristics in optimal state-space search

Erez Karpas [a,*], Oded Betzalel [b], Solomon Eyal Shimony [b], David Tolpin [b], Ariel Felner [c]

[a] *Faculty of Industrial Engineering and Management, Technion, Haifa 32000, Israel*
[b] *CS Department, Ben-Gurion University of the Negev, Beer-Sheva, Israel*
[c] *ISE Department, Ben-Gurion University of the Negev, Beer-Sheva, Israel*

A R T I C L E   I N F O

Article history:
Received 21 August 2016
Received in revised form 9 October 2017
Accepted 3 November 2017
Available online xxxx

Keywords:
Heuristic search
A^*
Admissible heuristics
Rational metareasoning

A B S T R A C T

The obvious way to use several admissible heuristics in searching for an optimal solution is to take their maximum. In this paper, we aim to reduce the time spent on computing heuristics within the context of A^* and IDA^*. We discuss *Lazy* A^* and *Lazy* IDA^*, variants of A^* and IDA^*, respectively, where heuristics are evaluated lazily: only when they are essential to a decision to be made in the search process. While these lazy algorithms outperform naive maximization, we can do even better by intelligently deciding when to compute the more expensive heuristic. We present a new rational metareasoning based scheme which decides whether to compute the more expensive heuristics at all, based on a myopic regret estimate. This scheme is used to create *rational lazy* A^* and *rational lazy* IDA^*. We also present different methods for estimating the parameters necessary for making such decisions. An empirical evaluation in several domains supports the theoretical results, and shows that the rational variants, rational lazy A^* and rational lazy IDA^*, are better than their non-rational counterparts.

1. Introduction

Introducing rational metareasoning techniques [1] into search is a research direction that has recently proved worthwhile. All search algorithms have decision points about which search actions to perform. Traditionally, tailored rules are hard-coded into the algorithms. However, applying metareasoning techniques based on value of information or other ideas can significantly speed up the search. This was shown for depth-first search in CSPs [2] as well as for Monte-Carlo tree search [3]. In the heuristic search literature, the greatest speedups using metareasoning techniques have been achieved when they were used to trade off solution quality for search time [4–6], where the search algorithm attempts to choose a node which will also minimize expected search effort, rather than just expected solution cost.

Taking advantage of metareasoning when the search is for an optimal solution is different than in satisficing planning, as the time vs. quality tradeoff is not available. Nevertheless, optimal search algorithms use heuristics, and when more than one such heuristic is available, metareasoning can be used to speed up search, trading off different aspects of search

* Corresponding author.
E-mail addresses: karpase@technion.ac.il (E. Karpas), odedbetz@cs.bgu.ac.il (O. Betzalel), shimony@cs.bgu.ac.il (S.E. Shimony), tolpin@cs.bgu.ac.il (D. Tolpin), felner@bgu.ac.il (A. Felner).

https://doi.org/10.1016/j.artint.2017.11.001

time, such as time spent for computing heuristics vs. time spent in node expansion, to achieve an overall speedup without jeopardizing optimality. In this paper, we examine how this can be done for A^* and IDA^*.

The A^* algorithm [7] and its derivatives, such as IDA^* [8] and RBFS [9] are best-first heuristic search algorithms guided by the cost function $f(n) = g(n) + h(n)$. A^* is often described as being 'optimal', in that it *expands* the minimum number of unique nodes. If the heuristic $h(n)$ is consistent[1] then the set of nodes expanded by A^* is both necessary and sufficient to find the optimal path to the goal with a unidirectional search [11].[2]

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. 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 scheme can be repeated as needed if we have more than two heuristics. LA^* expands no more nodes than A^* using the maximum. While LA^* may have the extra overhead of inserting a node into Open more than once, it has the potential to significantly reduce search time, as we may bypass computation of h_2 for many nodes. LA^* was briefly mentioned in the context of the MAXSAT heuristic for planning domains [12].

One major drawback for using A^* is that its memory consumption is linear in the number of generated nodes, which is typically exponential in the problem description size, and that may be unacceptable. In contrast to A^*, IDA^* is a linear-space algorithm which emulates A^* by performing a series of depth-first searches from the root, each with increasing costs, thus re-expanding nodes multiple times. IDA^* is typically used in domains and problem instances where A^* requires more than the available memory and thus cannot be run to completion. Similarly to A^*, the first thing to consider for IDA^* is lazy evaluation of the heuristics. In order to reduce the time spent on heuristic computations, Lazy IDA^* evaluates the heuristics one at a time, lazily. When h_1 causes a cutoff there is no need to evaluate h_2. Unlike Lazy A^*, where lazy evaluation must pay an overhead (re-inserting into the OPEN list), Lazy IDA^* ($LIDA^*$) is straightforward and has no immediate overhead.

As our 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 informed heuristic on only one node. *Selective max* (Sel-MAX), an online learning scheme which chooses one heuristic to compute at each node, is based on this idea [13]. 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 the computation times of the heuristics and of the average branching factor.

Similarly, previous work showed that randomizing a heuristic and applying *bidirectional pathmax* (BPMX) might sometimes be faster than evaluating all heuristics and taking the maximum [10]. This technique is only useful in undirected search spaces, and is therefore not applicable to some of the domains we examine in this paper. Both Selective max and Random compute the resulting heuristic *once*, before each node is added to Open, while LA^* and $LIDA^*$ compute 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.

In this paper, we combine the ideas of lazy heuristic evaluation and of trading off more node expansions for less heuristic computation time. We introduce a new variant of LA^* called *Rational Lazy* A^* (RLA^*), as well as a new variant of $LIDA^*$ called *Rational Lazy* IDA^* ($RLIDA^*$). These new rational algorithms are based on rational metareasoning in the sense of [1], and use a myopic *regret* criterion to decide whether to compute $h_2(n)$ or to bypass the computation of h_2 and expand n instead. They aim to reduce search time, even at the expense of more node expansions than A^* or IDA^* with the maximum of the heuristics. Empirical results on several heuristic search problems, as well as on numerous planning domains demonstrate that RLA^* and $RLIDA^*$ lead to better performance than their non-rational versions in many cases.

Perhaps the most closely related work, the RA* [14] and GHS [15] algorithms, have a similar objective — minimizing search time when given access to multiple heuristics. However they approach this problem in a different way, by choosing a subset of the heuristics to maximize over during search. On the other hand, we assume there are exactly two heuristics given, and attempt to minimize search time using these heuristics. An interesting direction for future work would be choosing a set of heuristics to combine using RLA^* or $RLIDA^*$.

Preliminary papers appeared on these ideas, introducing the A* variants [16] and the IDA* variants [17]. This paper unifies the presentation of RLA^* and $RLIDA^*$ into one coherent whole, while providing more experimental results. In addition, this paper further describes several technical optimizations for LA^* and $LIDA^*$. Finally, we extend the previous versions of RLA^* and $RLIDA^*$ by relaxing one of the assumptions originally made, as well as describing new techniques for estimating the parameters used in deciding when to compute the more expensive heuristic.

This paper is organized as follows. We begin (Section 2) by reintroducing LA^* and $LIDA^*$, and analyze the potential savings of LA^* over A^* and of $LIDA^*$ over IDA^*. We then consider the effects of some common additional enhancements to LA^* and $LIDA^*$ (Section 3). The main contribution of the paper is Section 4, which introduces the principles behind the decisions made by Rational LA^* and Rational $LIDA^*$, and Section 5, which presents different methods of using these

[1] A heuristic (in undirected graphs) is *consistent* if for any two nodes n and m, $|h(n) - h(m)| \leq cost(n, m)$ [10].

[2] A^* has similar guarantees on the set of nodes expanded with an inconsistent heuristic but may perform many unnecessary re-expansions [10]. In addition, we are neglecting the tie breaking in the last f-layer.

principles in practice. Our approach is then extensively evaluated empirically in Section 6, in several puzzle domains as well as in numerous domains from past planning competitions. We then discuss possible directions for future work in section 7, and conclude in section 8.

2. Lazy A^* and IDA^*

In this section we study LA^* and $LIDA^*$. The idea behind these algorithms is simple and was probably used by others. In fact, it was specifically mentioned in work on using the MAXSAT heuristic for planning [12]. Nevertheless, we study this technique in more depth in the context of A* and IDA*, and point out its strengths and weaknesses. Additionally, LA^* and $LIDA^*$ serve as a basis for our enhanced algorithms, RLA^* and $RLIDA^*$, which add metareasoning to the lazy technique.

2.1. Definitions and assumptions

Throughout this paper we assume for clarity that we have two available admissible heuristics, h_1 and h_2.

- 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 $g(n)+\max(h_1(n), h_2(n))$. We denote the cost of the optimal solution by C^*.
- 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 $t_1 + t_o$. Thus, our main objective is to reduce computations of h_2. Note that t_1, t_2, and t_o are not necessarily constants, as heuristic computation times could vary between different nodes, and t_o could depend on the size of Open. Nevertheless, treating them as constants is sometimes a useful approximation, and has been done before [13].

2.2. Lazy A^*

Algorithm 1: Rational lazy A^*.

1 Apply all heuristics to Start
2 Insert Start into Open
3 **while** Open *not empty* **do**
4     $n \leftarrow$ best node from Open **(update statistics)**
5   **if** *Goal(n)* **then**
6   **return** trace(n)
7   **if** *h_2 was not applied to n* **and opt-cond then**
8   Apply h_2 to n **(update statistics)**
9   insert n into Open
10   continue   //next node in OPEN
11   **foreach** *child c of n* **do**
12   Delete-duplicates(c, Open, Closed)
13   Apply h_1 to c **(update statistics)**
14   insert c into Open
15   Insert n into Closed
16 **return** FAILURE

We begin with a formal treatment of LA^*. 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 11–14), 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 $f_{max}(n)$. The optional condition, **opt-cond** in Line 7, as well as the statistics collected by **update statistics** in Lines 4, 8, and 13, are used by the Rational variant of LA^* which is introduced in Section 4. For the basic variant of LA^* discussed in this section **opt-cond** is simply assumed to be always TRUE, and thus ignored.

We use A^*_{MAX} to denote the variant of A* which evaluates both heuristics and uses their maximum. It is easy to see that LA^* is as informed as A^*_{MAX}, in the sense that a node n is expanded both by A^*_{MAX} and by LA^* only if $f_{max}(n)$ is the best f-value in Open. Therefore, LA^* and A^*_{MAX} generate and expand 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 [18,19], 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.

Please cite this article in press as: E. Karpas et al., Rational deployment of multiple heuristics in optimal state-space search, Artif. Intell. (2017), https://doi.org/10.1016/j.artint.2017.11.001

Table 1
Time spent on each node for A^*_{MAX} and for LA^*.

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^*	$t_1 + \mathbf{t_2} + 4t_o$	$t_1 + \mathbf{t_2} + 3t_o$	$t_1 + t_o$

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 ties are broken in Open in favor of small h-values, the goal node with $f = C^*$ could 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 (from the view point of saving computation time). Other nodes, those with $f_1(n) < C^*$ (and some with $f_1(n) = C^*$) are called *regular nodes*, as we need to compute both heuristics for them.

A^*_{MAX} computes both h_1 and h_2 for all generated nodes, spending time $t_1 + t_2$ on all generated nodes, as well as the time spent on inserting all nodes into the open list (t_o), and an extra t_o spent on removing the nodes that were expanded from the open list. In contrast, for *good* nodes LA^* only spends t_1 time computing heuristic estimates (saving t_2 time), as well as extra overhead on open list operations. In the basic 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. Both A^*_{MAX} and LA^* spend $t_1 + t_2$ time computing heuristic estimates for each of these nodes. A^*_{MAX} inserts and removes each of these nodes from the open list, for an extra $2t_o$ time, while LA^* inserts and removes each of these nodes from the open list twice, for an extra $4t_o$ time.
- **(2)** *surplus regular* (SR) — nodes for which h_2 was computed but are still in Open when the goal was found. Both A^*_{MAX} and LA^* spend $t_1 + t_2$ time computing heuristic estimates for each of these nodes. A^*_{MAX} inserts each of these nodes to the open list, for an extra t_o time, while LA^* inserts each of these nodes into the open list twice, and removes them once, for an extra $3t_o$ time.
- **(3)** *surplus good* (SG) — nodes for which only h_1 was computed when the goal was found. A^*_{MAX} spends $t_1 + t_2$ time computing heuristic estimates for each of these nodes, and t_o time inserting each of them into the open list. On the other hand, LA^* only spends t_1 time computing heuristic estimates for each of these nodes, as well as an extra t_o time inserting each of them into the open list.

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 then, as seen in the SG column, there is a clear advantage for LA^*. This advantage grows when the number of SG nodes increases.

2.3. Lazy IDA^*

We now present $LIDA^*$, the lazy variant of IDA^*. Recall that IDA^* works in iterations, with an increasing cutoff threshold T at each iteration. After $h(n)$ is evaluated, if $f(n) = g(n) + h(n) > T$, then n is pruned and IDA^* backtracks to n's parent. Given both h_1 and h_2, a naive implementation of IDA^*, denoted as IDA^*_{MAX}, will evaluate them both and use their maximum in comparing against T. Lazy IDA^* ($LIDA^*$) is based on the simple fact that when you have an *or* condition in the form of *cond*1 *or* *cond*2 then if *cond*1 $= True$ then *cond*2 becomes irrelevant ("don't-care") and does not need to be computed, as the entire *or* condition is surely true. In the context of IDA^*, if $f_1(n) > T$ then the search can backtrack without the need to compute h_2. This simple observation is probably recognized by most implementers of IDA^*. Thus, it is likely that $LIDA^*$ is already a popular way to implement IDA^* when more than one heuristic is present.

The pseudo-code for $LIDA^*$ (and its enhanced version, Rational $LIDA^*$, which is discussed in Section 4), is depicted as Algorithm 2. In Lines 8–10 we check whether f_1 is already above the threshold, in which case search backtracks. h_2 is only calculated (in Lines 13–14) if $f_1(n) \leq T$. The "optional condition" in Line 13, as well as the updating of statistics in Lines 8 and 14, are needed for the Rational Lazy IDA^* algorithm, and will be explained in Section 4. In the standard version of Lazy IDA^*, the "optional condition" in line 13 is always true, and the respective heuristics are always evaluated at this juncture.

While Lazy A^* was always as informed as A^* using the maximum of the heuristics, this is not the case for Lazy IDA^*. This is because, in rare cases, $LIDA^*$ can cause extra iterations of the algorithm compared to IDA^*. Suppose that the current threshold is T and the current value of the *next threshold* (NT) is $T + 3$ as some node m seen in the current iteration has $f(m) = T + 3$. Now we generate node n with $f_1(n) = T + 1$ and thus set $NT = T + 1$ and bypass h_2. However, if $f_2(n) = T + 2$ then consulting h_2 would have caused $NT = T + 2$. With $LIDA^*$, we may now start a new and redundant iteration with threshold $T + 1$, rather than with $T + 2$ — which would have been the case with IDA^*_{MAX}.

However, in order for an extra iteration #i with value v to happen, all nodes that have an h_2 value of v in iteration #$i - 1$ have to be pruned by their h_1 value. As the number of nodes grows between iterations (potentially exponentially), this case

Please cite this article in press as: E. Karpas et al., Rational deployment of multiple heuristics in optimal state-space search, Artif. Intell. (2017), https://doi.org/10.1016/j.artint.2017.11.001

Algorithm 2: Rational lazy IDA^*.

Lazy-IDA^*(root) **{**
  Thresh ← max(h_1(root), h_2(root))
  solution ← null
  while *solution = null and Thresh* $< \infty$ **do**
    solution, Thresh = Lazy-DFS(root, Thresh)
  return solution

Lazy-DFS(n, Thresh) **{**
  Compute h_1 **and update statistics**
  if $g(n) + h_1(n) >$ *Thresh* **then**
    return null, $g(n) + h_1(n)$
  if *goal-test(n)* **then**
    return n, Thresh
  if ***opt-cond*** **then**
    Compute h_2 **and update statistics**
    if $g(n) + h_2(n) >$ *Thresh* **then**
      return null, $g(n) + h_2(n)$
  next-Thresh ← ∞
  for n' *in successors(n)* **do**
    solution, temp-Thresh ← Lazy-DFS(n', Thresh)
    if *solution* $\neq$ *null* **then**
      return *solution, temp-Thresh*
    else
      next-Thresh ← min(temp-Thresh, next-Thresh)
  return *null, next-Thresh*

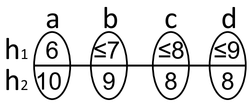


Fig. 1. Example of HBP.

becomes less likely in later iterations. Our empirical evaluation (Section 6.2.4 in particular), where Lazy IDA^* outperforms regular IDA^*, corroborates this. Furthermore, experiments on various domains where a random heuristic was selected (out of several heuristics) showed that such cases are very rare [10].

3. Enhancements to Lazy A^* and Lazy IDA^*

Having described the basic lazy algorithms (LA^* and $LIDA^*$), we now describe two enhancements of these algorithms. These enhancements are effective especially if t_1 and t_o are not negligible.

3.1. Heuristic bypassing

Heuristic bypassing (HBP) is a technique that allows us to compute the maximum, $\max(h_1(n), h_2(n))$, without evaluating one of the two heuristics for some nodes. HBP is probably used by many implementers of A^*_{MAX}, although to the best of our knowledge, it never appeared in the literature. HBP works for a node n under the following two conditions: **(1)** the operator between n and its parent p is bidirectional, and **(2)** both heuristics are *consistent*.

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 $h(n)$: $h(p) - C \leq h(n) \leq h(p) + C$. 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 $\overline{h_1(n)} \leq \underline{h_2(n)}$, or similarly, we skip the computation of h_2 if $\overline{h_2(n)} \leq \underline{h_1(n)}$. For example, consider node a in Fig. 1, where all operators cost 1, $h_1(a) = 6$, and $h_2(a) = 10$. Based on our bounds $\overline{h_1(b)} \leq 7$ and $\underline{h_2(b)} \geq 9$. Thus, there is no need to compute $h_1(b)$ 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 $h_1(d) \leq 9$ then $h_1(d)$ can potentially return the maximum and should thus be evaluated.

HBP can be applied 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 add n to Open with $f(n) = g(n) + \underline{h_2(n)}$ and continue as in LA^*. In this case,

Please cite this article in press as: E. Karpas et al., Rational deployment of multiple heuristics in optimal state-space search, Artif. Intell. (2017), https://doi.org/10.1016/j.artint.2017.11.001

we saved t_1 time by not computing h_1, used $\underline{h_2(n)}$ which is more informative than $h_1(n)$, while still having the option to compute h_2 later, if needed. If, however, $\overline{h_1(n)} \geq \underline{h_2(n)}$, then we compute $h_1(n)$ and continue regularly.

HBP can also be applied in $LIDA^*$, where one only needs to know whether the f-value is below or above the threshold T. Again, assume that node n was generated, that p is the parent of n, and that the cost of the edge is C. If it happens to be the case that $f_1(p) + C \leq f_2(p)$, then we can deduce that $f_1(p) + C \leq T$. This is because p was expanded, and thus $f_2(p) \leq T$. Since the heuristics are consistent, we know that $f_1(n) \leq f_1(p) + C \leq T$. Thus, in such cases, one can skip the computation of $h_1(n)$ and go directly to h_2.

While HBP can save some computation of h_1, 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. Finally, we remark that when the heuristic is inconsistent then a mechanism called bidirectional pathmax (BPMX) [10] can be used to propagate heuristic values from parents to children and vice versa. Using exhaustive evaluations of all heuristics, even if $h_1(n)$ already exceeded the threshold, can potentially help in propagating larger heuristic values to the neighborhood of n. Nevertheless, experiments showed that even in this context, lazy evaluation of heuristics is faster than exhaustive evaluation [10].

3.2. Open bypassing

Another optimization, which is relevant only for LA^*, is called Open bypassing (OB). Suppose that 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 $f_1(n) < f_{best}$, then n can immediately reach the top of Open and h_2 will be computed. In such cases where $f_1(n) < f_{best}$ we can choose to compute $h_2(n)$ right away (after Line 13 in Algorithm 1), thus saving the overhead of inserting n into Open and popping it again at the next step $(= 2 \times t_o)$.[3] For such nodes, LA^* is identical to A^*_{MAX}, as both heuristics are computed before the node is added to Open. This enhancement is reminiscent of the *immediate expand* technique applied to a generated node [20,21]. The same technique can be applied when n again reaches the top of Open when evaluating $h_2(n)$; if $f_2(n) < f_{best}$, expand n right away and bypass open. When applying OB, LA^* will incur the extra overhead of two Open cycles only for nodes n where both $f_1(n) > f_{best}$ and then later $f_2(n) > f_{best}$. As $LIDA^*$ does not keep an open list, this enhancement is only applicable to LA^*.

In our earlier paper [16] we showed that on some unit-edge cost domains such as the 15-puzzle, if t_1 and t_2 are very similar then HBP and OB are particularly useful, and they save heuristic computation for many of the nodes. For example, the number of good nodes dropped from 38% to 11% when adding HBP on top of LA^*. This leaves little room for further improvement for LA^*. Thus, timing results did not show a significant difference between the different versions.

3.3. Extending lazy A^* and IDA^* to multiple heuristics

Given a set $\{h_1, h_2, \ldots, h_n\}$ of heuristics, it is straightforward to extend either Lazy A^* or Lazy IDA^* to handle multiple heuristics. Simply repeat the code snippet used for h_2 in either algorithm, and apply it to each h_i, for all $3 \leq i \leq n$. This assumes that we have already ordered the heuristics in some reasonable way, although this ordering itself is far from trivial, as discussed in Section 7.

4. Rational lazy A^* and IDA^*

LA^* provides a very strong guarantee, of expanding the same set of nodes as A^*_{MAX}. While $LIDA^*$ can potentially result in extra iterations, it is also guaranteed to expand the same set of nodes as IDA^* with the maximum of the two heuristics. However, often we would prefer to expand more nodes, if it means reducing search time [13]. This will be possible, for example, if we skip the computation of h_2 for a given node n and expand it whenever we believe that expanding it and generating its children will consume less CPU time than calculating $h_2(n)$. We now present *Rational Lazy* A^* (RLA^*) as well as *Rational Lazy* IDA^* ($RLIDA^*$) — two algorithms which attempt to optimally manage this tradeoff, based on the principle of rational metareasoning.

We begin by reviewing rational metareasoning in the context of optimal search in Section 4.1. We then discuss how we can compute the regret of our possible meta-level decisions in Section 4.2. Using these regret values, we derive a rational meta-level policy in Section 4.3. Finally, in Section 5 we describe some techniques for implementing this policy in practice.

4.1. Rational metareasoning for optimal search

Previous work has presented a general theory of rational metareasoning in search [1]. In rational metareasoning, theoretically every computational action (heuristic function evaluation, node expansion, open list operation) should be treated

[3] The only "risk" in this enhancement is that n might have a sibling node n' with an even lower f value; in this case, n' would have been removed from Open before n by basic LA^*. However, this does not affect the optimality of the solution returned, as both h_1 and h_2 are admissible heuristics, so n still ends up in Open with an admissible estimate.

Please cite this article in press as: E. Karpas et al., Rational deployment of multiple heuristics in optimal state-space search, Artif. Intell. (2017), https://doi.org/10.1016/j.artint.2017.11.001

as an action in a sequential decision-making meta-level problem: actions should be chosen so as to achieve the greatest expected utility (hence the term "rational"). For algorithms guaranteed to deliver an optimal solution, maximizing expected utility translates into minimizing the expected search time. However, the appropriate general metareasoning problem is extremely hard to parametrize precisely, and when fully parametrized, results in an intractable metareasoning problem. Therefore, typically simplifying assumptions of two types are made in order to allow for a practical approximation to rational metareasoning: myopic assumptions and independence assumptions [1].

In this paper we focus on just one decision type, made in the context of LA^* and $LIDA^*$— that of deciding whether to evaluate or to bypass the computation of h_2 for some node n. We have two options: **(1)** Evaluate the second heuristic $h_2(n)$, and proceed using the value $f_{max}(n)$, or **(2)** bypass the computation of $h_2(n)$ and use $f_1(n)$, thereby saving time by not computing h_2, at the risk of additional expansions and evaluations of h_1.

In the context of LA^*, we must make this decision when n first emerges from Open (Line 7 in Algorithm 1). This is done by the optional condition **opt-cond**. If we choose to bypass the computation of $h_2(n)$ (opt-cond=FALSE), n is expanded right away. Otherwise (opt-cond=TRUE) $h_2(n)$ is computed, and n is enqueued back in Open with $f_{max}(n)$.

In the context of $LIDA^*$, we must make this decision when we evaluate a node n and $f_1(n)$ was within the current IDA^* threshold T (Line 13 in Algorithm 2). If we choose to bypass the computation of $h_2(n)$ (opt-cond=FALSE), n is expanded right away. Otherwise (opt-cond=TRUE), $h_2(n)$ is evaluated and IDA^* proceeds with $f_{max}(n)$, that is, the search backtracks if $f_{max}(n) > T$.

We would like the algorithm to make the decision on whether to evaluate h_2 rationally. Therefore, we define a criterion based on the regret for bypassing $h_2(n)$ in this context. We define regret here as the value lost (in terms of expected increased run time) due to bypassing the computation of $h_2(n)$, i.e., how much runtime is increased due to bypassing the computation. We wish to compute $h_2(n)$ only if this regret is positive. Next, we estimate the regret for each of these possible decisions.

4.2. Computing the regret

Let us now consider our two possible decisions (compute h_2 or bypass h_2). Suppose that we choose to compute h_2 — this results in one of the following outcomes:

h_2 **helpful:** That is, the computation of $h_2(n)$ will prevent the expansion of n. For LA^*, this means that n is re-inserted into Open, and the goal is found without ever expanding n, which is only possible if $f_{max}(n) \geq C^*$. For $LIDA^*$, since we already know that $g(n) + h_1(n) \leq T$, "helpful" means $g(n) + h_2(n) > T$, and therefore n is not expanded in the current IDA^* iteration.

h_2 **not helpful:** n is still expanded despite the computation of $h_2(n)$. For LA^*, this means either now or eventually, while for $LIDA^*$ this means n is expanded in the current iteration.

In estimating the gain due to the computation of h_2, we rely on the subtree independence assumption [1] — that a computation in one node contributes information only to itself or one of its ancestors, which is tantamount to assuming that the search space is a tree, and that there is no dependency between nodes at different branches of the tree. Thus, we assume that the information gathered by computing h_2 for node n is used solely for pruning n.

Under these assumptions, computing h_2 could be beneficial only in the first outcome, where the potential time savings due to computing h_2 are due to pruning a search subtree, at the expense of time t_2. However, for a given node n, which outcome will take place after evaluating $h_2(n)$ is not known to the algorithm when it makes that decision, and thus it must decide whether to evaluate $h_2(n)$ according to what it *believes to be* the probability of each of the outcomes.

In order to estimate the regret, we make the following additional simplifying assumptions:

I The decision is made *myopically*: we work under the assumption that the algorithm continues to behave like Lazy A^* or Lazy IDA^* starting with the children of n, and will never bypass another h_2 computation after the current decision is made (i.e., opt-cond=TRUE for all future decisions).

II h_2 is *consistent*. Thus, if evaluating h_2 is helpful on n, it is also helpful on any successor of n, due to the fact that f (specifically, f_2) increases monotonically for consistent heuristics.

III As a first approximation (relaxed later on), we also assume that all successors of n would be expanded if we used only h_1 for them.

Note that these metareasoning assumptions are made in order to derive decisions, and as is common in research on metareasoning, the assumptions do *not* actually hold in practice [1]. Nevertheless, if the violation of the assumptions is not "too severe", the resulting algorithms still show significant improvements over their non-rational counterparts. Without such assumptions the model becomes far too complicated and one cannot move ahead at all. Nevertheless, the assumptions make sense: if our rational algorithms (RLA^* and $RLIDA^*$) are better than their lazy (non-rational) counterparts (LA^* and $LIDA^*$, respectively), the first assumption results in an upper bound on the regret, because regret considers future runtime, and the rational algorithm should be faster than its non-rational version. Unfortunately, we can not provide such a statement

Please cite this article in press as: E. Karpas et al., Rational deployment of multiple heuristics in optimal state-space search, Artif. Intell. (2017), https://doi.org/10.1016/j.artint.2017.11.001

Table 2
Regret in rational lazy A^*.

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

regarding the two other assumptions, and in fact Assumption III is clearly off the mark in certain domains and we attempt to relax it later on.

In order to derive a rational policy, we begin by analyzing our two possible decisions: to compute h_2 or to bypass h_2, under the two (unknown) possible future outcomes: h_2 is helpful or not. 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 for RLA^*, thus delaying the expansion of n. Note that if we use the Open bypassing optimization described in Section 3.2, t_d could be lower. For the sake of simplicity we assume this does not happen, although our analysis could be easily extended if the fraction of nodes for which this optimization applies is known (or estimated). t_e is the time to expand n, and evaluate h_1 on each of its successors (as well as the time to remove n from Open and insert the successors into the open list for RLA^*). $b(n)$ denotes the "local branching factor", i.e., the number of successors of n, and t_c the time to generate the children of n (i.e., to have a copy of the states representing the children at hand). We thus have:

$$\begin{aligned} t_d &= t_2 + t_o \\ t_e &= t_o + t_c + b(n)t_1 + b(n)t_o \end{aligned} \tag{1}$$

Computing h_2 needlessly wastes time t_d. Bypassing h_2 computation when h_2 would have been helpful means generating all successors of n, and computing h_2 for them (assumption I). From assumption III, h_1 is not going to be enough to prune any of these successors, but because h_2 is consistent (assumption II) h_2 will be helpful on the successors and prune them. Therefore, we have expanded one "extra" level of the search tree, and computed h_1 and h_2 on $b(n)$ successors, instead of computing h_2 for n only. This wastes $t_e + b(n)t_d$ time, but because computing h_2 would have cost t_d we need to subtract t_d and thus the regret is $t_e + (b(n) - 1)t_d$. Using Table 2, we can now derive a rational policy for deciding whether to compute or bypass h_2. Finally, for simplicity, we will assume $t_o = 0$ for $RLIDA^*$.

4.3. Deriving a rational policy

Now that we have the regret of each possible action under each possible future outcome, we can derive a rational policy, which will tell us which decision we should make. Let us denote the probability that h_2 is helpful by p_{h_2}.[4] The expected regret of computing h_2 is thus $(1 - p_{h_2})t_d$. On the other hand, the expected regret of bypassing h_2 is $p_{h_2}(t_e + (b(n) - 1)t_d)$. As we wish to minimize the expected regret, we should thus evaluate h_2 (i.e., **opt-cond** = **TRUE**) just when:

$$(1 - p_{h_2})t_d < p_{h_2}(t_e + (b(n) - 1)t_d) \tag{2}$$

or equivalently when:

$$(1 - b(n)p_{h_2})t_d < p_{h_2}t_e \tag{3}$$

If $b(n)p_{h_2} \geq 1$, then the expected regret is minimized by always evaluating h_2, regardless of the values of t_d and t_e. In these cases, the rational algorithms cannot be expected to do better than their lazy counterparts. 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 the search algorithm evaluates $h_2(n)$, then one should simply use LA^* or $LIDA^*$.

For $p_{h_2}b(n) < 1$, the decision of whether to evaluate h_2 (i.e., **opt-cond**) depends on the values of t_d and t_e:

$$\textbf{evaluate } h_2 \textbf{ if } t_d < \frac{p_{h_2}}{1 - p_{h_2}b(n)} t_e \tag{4}$$

By substituting (1) into (4) we obtain the following **opt-cond**: **evaluate** $h_2(n)$ **if**:

$$t_2 + t_o < \frac{p_{h_2}}{1 - p_{h_2}b(n)} (t_c + b(n)t_1 + (b(n) + 1)t_o) \tag{5}$$

The factor $\frac{p_{h_2}}{1 - p_{h_2}b(n)}$ depends on the potentially unknown probability p_{h_2}, making it difficult to reach the optimum decision. However, if our goal is just to do better than LA^* or $LIDA^*$, then it is safe to replace p_{h_2} by an upper bound on p_{h_2}. In Section 5, we describe practical methods for estimating p_{h_2}. However, we first describe a variant of our decision rule, which relaxes our third assumption.

[4] This quantity was denoted by p_h in previous papers.

4.4. Relaxing Assumption III

Our third assumption, that all successors of n would not be pruned solely by h_1, is obviously frequently violated, especially in the context of IDA^*. A relaxation we use here is that there is some probability $p_{h_1}(n)$ that such pruning occurs, i.e., $p_{h_1}(n)$ is the probability that h_1 will prune each of n's *successors*, thereby also making the simplifying assumption that these probabilities are i.i.d.

Under this relaxed assumption, the regret values are the same as in Table 2, except for that of bypassing h_2 when it is helpful. In this case instead of "wasting" $b(n)$ calculations of h_2 in each successor of n, we only waste it with a probability of $1 - p_{h_1}(n)$. Thus, the regret accounts for expanding the node, computing h_1 on all $b(n)$ children and then h_2 on the $b(n)(1 - p_{h_1}(n))$ children which were not pruned by h_1, less the time spent computing h_2 on the parent node — yielding a regret of $t_e + (b(n) - 1)(1 - p_{h_1}(n))t_d$. Note that if $p_{h_1}(n) = 0$, i.e., calculation of h_1 in n's successors is never helpful, we get the same regret as in Table 2. Finally, we remark that using this more fine grained equation requires estimating both p_{h_2} and p_{h_1}. The next section describes several techniques for estimating p_{h_2}, and one technique which can also estimate p_{h_1}.

5. Using the rational policy in practice

Despite the simplicity of equation (5), it is still not clear how to use it in practice. This is because all of the quantities $b(n)$, t_1, t_2, t_c, t_o and especially $p_{h_1}(n)$ and $p_{h_2}(n)$ may actually be unknown. Furthermore, these quantities might change between different nodes, or as search progresses. Estimating the times t_1, t_2, t_c, t_o is usually easy, as we can measure these during search and take the average measurement as the estimate. The number of successors, $b(n)$, is also often readily available to the search algorithm. However, we must still also estimate p_{h_2} and p_{h_1}.

Note that RLA^* and $RLIDA^*$ are meant to improve upon their non-rational counterparts, so we would like to be highly confident that we will not be worse than LA^* and $LIDA^*$, respectively. Since LA^* and $LIDA^*$ always choose to compute h_2, we can guarantee being no worse than them by always computing h_2. Thus, we should only decide to skip h_2 computation when we are highly confident it is the right decision. Using an *upper bound* on p_{h_2} would mean we err on the side of caution, and might compute h_2 in some cases where the right decision might be to skip h_2. However, this achieves our purpose of not doing any worse than LA^* and $LIDA^*$ with high probability.

One possible approach is to use concentration measures to derive such a probabilistic upper bound on p_{h_2} [17,22]. However, in practice this bound is too loose, and our decision rule almost always chooses to compute h_2. In the remainder of this section, we describe other approaches for estimating p_{h_2}, which perform better in practice.

5.1. Domain-specific parameter settings

If we expect to only solve problems from one specific domain, we can treat p_{h_2} and p_{h_1} as setting-specific constants (that is, hyper-parameters), and tune them on a given set of benchmarks. Plugging those into the very crude model above is sufficient to achieve improved performance in some cases, as shown in our experimental evaluation in Section 6. Furthermore, if we know something about the times t_1, t_2, and t_o, we can simplify the decision rule in Equation (5).

For example, if we are using RLA^* in domains where evaluating h_1 is cheap, (e.g., the Manhattan distance heuristic), t_o is the most significant part of t_e. Open in A^* is frequently implemented as a priority queue, and thus we have, approximately, $t_o = \tau \log N_o$ for some constant τ, where the size of Open is N_o. For such domains rule (5) can be approximated as:

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

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 PDDL 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:

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

Note that the right hand side of (7) grows with $b(n)$, and thus it is beneficial to evaluate h_2 only for nodes with a sufficiently large branching factor. Also recall that if $p_{h_2} b(n) \geq 1$, then the expected regret is minimized by always evaluating h_2 (as mentioned in Section 4.3), which is consistent with this conclusion.

5.2. Empirical frequencies

The above approach relies on a set of benchmarks on which we can tune p_{h_2} and p_{h_1}. However, such a set of benchmarks is not always available. Furthermore, often search problems are very different from each other, and even problem instances in the same domain are of varying size. Thus getting a single set of values for p_{h_2} and p_{h_1} which works well across many problems is difficult. Instead, we would like to adaptively estimate these parameters during search.

Please cite this article in press as: E. Karpas et al., Rational deployment of multiple heuristics in optimal state-space search, Artif. Intell. (2017), https://doi.org/10.1016/j.artint.2017.11.001

The first adaptive method we present focuses on estimating p_{h_2} from empirical frequencies (it is safe to assume $p_{h_1}=0$, as this will also result in an upper bound on regret). One important distinction between RLA^* and $RLIDA^*$ is the immediacy of the feedback about whether h_2 was helpful. $RLIDA^*$ can tell whether h_2 was helpful immediately after evaluating it — h_2 is helpful iff $g(n)+h_2(n)>T$, thereby causing a cutoff. Thus, we simply estimate p_{h_2} by the frequency of observed helpful evaluations of h_2 so far.

For RLA^*, things are a bit more complicated, as we do not have immediate feedback about whether h_2 was helpful. Once a node for which we computed h_2 is expanded, we know that h_2 was *not* helpful on that node, which is why in the pseudo-code for RLA^*, we call **update statistics** whenever a node is removed from Open (Line 4 in Algorithm 1). However, we can only be certain that h_2 was helpful after search terminates.

Nevertheless, we can still *estimate* p_{h_2}. 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. After the search completes, this is the exact definition of h_2 being helpful, but during the search, this is a necessary condition for n to be potentially helpful. Let us denote by A the number of nodes for which we computed h_2 that were not yet expanded, and are thus still in Open. Let us denote by B the number of nodes for which we computed h_2. Then $\frac{A}{B}$ can be used as an estimate of p_{h_2}. In fact, this method will tend to overestimate p_{h_2}, which is consistent with our goal of trying to use an upper bound to ensure we are no worse than LA^* or $LIDA^*$.

One minor issue is that using the empirical frequency is not likely to be a stable estimate at the beginning of the search, such as after seeing only 1 example. If we use this estimate directly, then we could get an estimate of $p_{h_2}=0$, which means we would never evaluate h_2 again, and never learn that it might be helpful. To overcome this problem, we "imagine" we have observed k examples, which give us an estimate of $p_{h_2}=p_{init}$, and use a weighted average between these k examples, and the observed examples — that is, we estimate p_{h_2} by $(\frac{A}{B}\cdot B+p_{init}\cdot k)/(B+k)$. In our empirical evaluation, we used $k=1000$ and $p_{init}=0.5$.

5.3. Type systems

The approaches we used above only estimate p_{h_2}, the probability that h_2 is helpful. The final approach we present can estimate both p_{h_2} and p_{h_1}, and is based on the use of type systems [23]. A *type system* partitions the state-space nodes into "similar valued" classes of nodes, called *types*, according to some features of each node. Thus, each type system is defined by a set of features. Although type systems were originally used to predict the number of nodes generated by search algorithms, we use them in order to estimate p_{h_2} and p_{h_1}. In other words, we use a set of features to define a type system, and use that type system to compute conditional probabilities corresponding to p_{h_2} and p_{h_1}, as described below.

5.3.1. Estimating p_{h_2}

In order to estimate p_{h_2} we learn a distribution of h_2 values for each *type*. In every call to **update statistics** in Algorithms 1 and 2, we compute the type of the current node, and update the distribution of h_2 values for nodes of that type. We consider two type systems here:

Type System 1: (TS1) TS1 consists of one feature only, the value of h_1.
Type System 2: (TS2) TS2 includes not just the value of h_1, but also the value of h_2 in the closest ancestor for which h_2 was computed, and the distance to that ancestor.

In other words, TS1 simply learns the conditional distribution table of h_2 as a function of h_1. However, TS1 ignores important information, by assuming that the distribution of h_2 given h_1 is constant in the entire search tree, which is unlikely to be true. TS2 attempts to remedy that by also looking at the "closest" source of information regarding h_2.

Both of the above type systems give us a distribution on the value of $h_2(n)$, conditioned on the values of the features of the type system. This distribution is learned online, and is updated as the search progresses. In $RLIDA^*$, this distribution can be used with the threshold T to directly estimate p_{h_2}, the probability that h_2 will be helpful for the current node. Since we want $g(n)+h_2(n)>T$, we simply want to estimate the probability that $h_2(n)>T-g(n)$, which is given by:

$$p_{h_2}(n)=\sum_{i=T-g(n)+1}^{\infty}\Pr(h_2(n)=i) \tag{8}$$

where $\Pr(h_2(n)=i)$ is obtained from our type system.

Note that the threshold T plays a very important role here — it tells us exactly how high the value of $h_2(n)$ needs to be in order for h_2 to be helpful. However, for RLA^*, the threshold is not available. In order to apply this estimation method with RLA^*, we must use some other quantity instead of T. We use the highest f-value expanded so far during search, which, like the current threshold T, serves as a lower-bound on the cost of an optimal solution. If $f_2(n)$ is less than the highest f-value expanded so far, then h_2 is definitely not helpful. On the other hand, it could be the case that $f_2(n)$ is greater than the highest f-value expanded so far, and h_2 still turns out not to be helpful. This is a conservative estimate, which is again consistent with trying to guarantee we do not do worse than LA^* or $LIDA^*$. Nevertheless, the empirical results show that is useful.

Please cite this article in press as: E. Karpas et al., Rational deployment of multiple heuristics in optimal state-space search, Artif. Intell. (2017), https://doi.org/10.1016/j.artint.2017.11.001

5.3.2. Estimating p_{h_1}

We have come up with only one viable technique to estimate p_{h_1}. This technique is also based on a type system, which we call **Type System 3 (TS3)**. Recall that $p_{h_1}(n)$ is the probability that h_1 will be helpful for n's successors. Thus, TS3 keeps a distribution of the h_1 values of a node's *successors*, as a function of the node's h_1 value and distance to the last h_2 computation.

In order to use TS3, whenever we compute h_1 for some node n with parent p, we update the distribution of h_1 values of p's successors. We estimate the probability that h_1 is helpful on the successors of n, using an equation similar to Equation (8), except that the g value of the successors is $g(n)+1$[5]:

$$p_{h_1}(n) = \sum_{i=T-g(n)}^{\infty} \Pr(h_1(succ_n) = i) \tag{9}$$

Where $\Pr(h_1(succ_n) = i)$ is the probability that the successors of n will have an h_1 value of i, which is obtained from the statistics table we keep. We remark that we tried several other type systems for estimating p_{h_1} that yielded similar results.

Finally, note that TS3 is only used to estimate p_{h_1}. In order to use RLA^* or $RLIDA^*$, we must also estimate p_{h_2}. Thus, we combine TS3 with either TS1 or TS2, and refer to these combinations as TS1+TS3 or TS2+TS3, respectively. For example, in TS1+TS3, TS1 is used to estimate p_{h_2} and TS3 is used to estimate p_{h_1}.

6. Empirical evaluation

We have described two new algorithms, RLA^* and $RLIDA^*$, which are based on rational metareasoning. These algorithms have a decision rule based on some parameters, most importantly p_{h_2} (the probability that h_2 is helpful) and p_{h_1} (the probability that h_1 is helpful). We have also described different ways of estimating these parameters in practice:

- Domain-specific parameter estimation (Section 5.1)
- Estimation from empirical frequencies (Section 5.2)
- Estimation using type systems (Section 5.3). We presented three different type systems (TS1, TS2, and TS3), and four different ways to combine them: TS1, TS2, TS1+TS3, and TS2+TS3.

We now examine these algorithms and their different parameter estimation methods empirically. As these two algorithms are very different from each other (for example, in their memory requirements), we divide our empirical evaluation into two parts: one comparing RLA^* to A^* and its variants, and the other comparing $RLIDA^*$ to IDA^* and its variants. We evaluate all of our algorithms on a large set of PDDL planning domains from all previous International Planning Competitions (IPC), as well as on some combinatorial puzzles.

6.1. Evaluation of RLA^*

We begin with an empirical evaluation of RLA^*, which we compare to A^*-based search algorithms. Results are for a set of planning domains, as well as for two variants of the 15-puzzle.

6.1.1. Planning domains

We implemented LA^* and RLA^* on top of the Fast Downward planning system [24], and used two state-of-the-art heuristics: the admissible landmarks heuristic h_{LA} (as h_1) [25], and the landmark cut heuristic h_{LMCUT} [26] (as h_2). On average, h_{LMCUT} computation is about 8 times more expensive than that of h_{LA}. 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 negligible compared to the cost of heuristic evaluations, especially with the heuristics we used.

We experimented with 57 planning domains: the optimal versions of all IPC planning domains in the Fast Downward repository, as well some of those from IPC 2014. We had to exclude the CITYCAR domain and most (17 out of 20) instances of the CAVEDIVING domain, because they included conditional effects, and the heuristics we used did not support conditional effects in the version of Fast Downward we used.[6] We compare the performance of A^* using each of the heuristics individually (where lm denotes h_{LA} and lmcut denotes h_{LMCUT}), as well as with their maximum (denoted by max). We also evaluate selective max (selmax) [13], LA^* (denoted lazy), and RLA^* using two different methods of estimating p_{h_2}: empirical frequencies (*emp*), and the TS1 type system (*T1*).

We did not implement the TS2 and TS3 type systems, as they require the value of h_2 in the closest ancestor for which h_2 was computed, and the distance to it. There are two ways to obtain this information for each node: either follow the

[5] With non-uniform action costs, we can either generate the successors to get their exact g-values, or use an estimate of the average action cost.

[6] It is worth noting that the Fast Downward translator managed to get rid of conditional effects in the 3 instances of CAVEDIVING as well as in the MAINTENANCE domain, even though they were present in the PDDL domain.

Please cite this article in press as: E. Karpas et al., Rational deployment of multiple heuristics in optimal state-space search, Artif. Intell. (2017), https://doi.org/10.1016/j.artint.2017.11.001

Table 3
Results for A^* algorithms in planning domains.

Algorithm	Coverage	Avg search time score	Expansions	Total memory (KB)	h_2 ratio
lm	797	32.98	12,057.25	106,135,188	
lmcut	826	33.87	1,978.56	**17,626,512**	
max	859	34.18	**1,551.71**	29,415,652	
selmax	873	34.61	3,581.64	57,311,016	
lazy	**875**	35.55	1,564.77	30,464,880	0.66
emp	874	**35.74**	1,640.27	38,113,656	0.59
T1	872	35.35	2,028.68	50,850,408	0.54

parent pointers until you find a node for which h_2 was computed (requiring time overhead), or store this extra information for each node (requiring memory overhead). Using the first approach would hinder our objective of speeding up the search, while using the second approach would increase the memory overhead. We also did not use domain-specific parameter settings, because we believe the other approaches are better equipped to handle the diversity of IPC domains we used. The search was limited to 3GB memory, and 30 minutes of CPU time on a single core of an Intel E5-2680 CPU with 64-bit Linux OS.

Table 3 summarizes the results of our empirical evaluation across all domains. Detailed tables including the results for each domain are relegated to the appendix, but references to these tables are provided here.

We will first examine the coverage — the number of problems solved by each search algorithm in 30 minutes (Table A.14 provides detailed, per-domain, coverage results). First, note that all of the intelligent combination methods (selective max, LA^*, and the 2 variants of RLA^*) solve over 870 problems, while each heuristic alone solves less than 830, and A^*_{MAX} solves less than 860. This indicates that selectively choosing which heuristic to compute has demonstrable benefits. LA^* solves one more problem than RLA^* with empirical frequencies, which is due to the extra overhead required by RLA^*.

Furthermore, looking at search time, we see the benefits of using RLA^*. We compare time score (Table A.15 provides per-domain results), which is computed from the time it took an algorithm to solve a problem. If the search time is less than 1 second, then the score is the maximum possible score — 100. The score then decreases logarithmically, until it reaches 0 at 1800 seconds. Note that the time score is a standard measure computed by DOWNWARD-LAB [27], Fast Downward's experiment running tool. Thus, the time score rewards fast solutions, and does not distinguish between solving a problem in 1800 seconds and a timeout. As the results show, RLA^* with empirical frequencies achieves a better (higher) time score than all other algorithms.

Fig. 2 shows the anytime performance of the different algorithms — the number of instances solved under different time limits. Note that both axes are in logscale. As the figure shows, the advantage of the RLA^* variants is even more evident for shorter timeouts, and LA^* only becomes the better algorithm for longer planning times (at 1773 seconds, to be exact).

To further understand the differences between the search algorithms, we compare the geometric mean of the number of expanded nodes in each search algorithm (Table A.16 shows per-domain results). As expected, A^*_{MAX} has the fewest expanded nodes, and LA^* is very close (the differences are due to tie-breaking). The two variants of RLA^* have more expanded nodes, while selective max expands more nodes than all of the RLA^* variants. This shows that these algorithms do behave differently.

We also compare peak memory usage for these algorithms (Table A.17 shows per-domain results). The results show that using only h_{LMCUT} is the most memory efficient. This is due to the extra memory overhead involved with using the h_{LA} heuristic, which is necessary to keep track of the achieved landmarks at each state. However, when looking at the different ways of combining both of these heuristics, the results are very similar to those in Table A.16 — A^*_{MAX} uses the least memory, LA^* is very close, the variants of RLA^* use more memory, and selective max uses almost twice the memory as A^*_{MAX}.

Fig. 3 shows the speedup vs. memory overhead for RLA^* vs. LA^*. Each point in the plots represents an instance, solved by RLA^* and LA^*. The x-coordinate of the instance is the relative speedup of RLA^* vs. LA^*, and the y-coordinate is the ratio of expanded states of RLA^* vs. LA^*, i.e., the extra memory. Points to the right of the dotted line at $x = 1$ are the instances where RLA^* is faster. Fig. 3a compares RLA^*(emp) to LA^*, while Fig. 3b compares RLA^*(T1) to LA^*. As these plots show, RLA^* is typically faster (more points to the right of the dotted line), but comes at the price of an increased number of expanded nodes (except for 4 instances in Fig. 3b, which are due to tie-breaking). The plots also show that RLA^*(T1) differs more from LA^* than RLA^*(emp). Finally, we can see that an increase in speedup does not necessarily correlate to an increase in the number of expanded nodes.

To complete the picture, we look at the fraction of nodes for which h_2 (h_{LMCUT} in this case) was computed for LA^* and RLA^* (per-domain results are in Table A.18). We do not include selective max here, as selective max can choose not to compute h_1 for some nodes, while the algorithms we compare all do. Unsurprisingly, LA^* has the highest numbers here. Finally, note the correlation between a high proportion of h_2 computations with decreasing number of expanded nodes.

Another important conclusion of this empirical evaluation is that RLA^* with empirical frequency estimation beats using type system TS1. Recall that TS1 relies on a threshold in the decision rule, which is only available in $RLIDA^*$. In RLA^* we use the highest f-value expanded so far instead of the threshold, while empirical frequency estimation does not rely on a

Please cite this article in press as: E. Karpas et al., Rational deployment of multiple heuristics in optimal state-space search, Artif. Intell. (2017), https://doi.org/10.1016/j.artint.2017.11.001

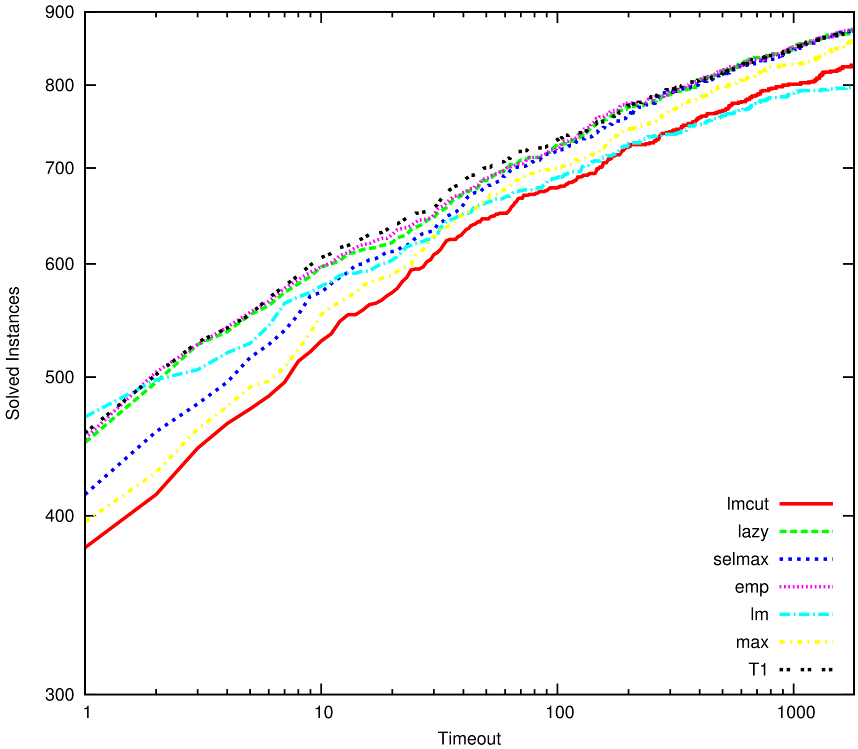


Fig. 2. Anytime plot for A^* algorithms in planning domains (both axes are in logscale).

threshold. One possible explanation for these results is that our estimate of the threshold is too conservative. In future work we will examine a better proxy for the threshold.

6.1.2. Weighted 15 puzzle

We now provide an empirical evaluation on the weighted 15-puzzle [4], a variant of the 15-puzzle where the cost of moving each tile is equal to the number on the tile. For consistency of comparison, we used a subset of 36 problem instances out of the set of 100 instances by [8], keeping the problems which could be solved with 2Gb of RAM and 15 minutes timeout using the Weighted Manhattan Distance heuristic (WMD) for h_1. As the expensive and informative heuristic h_2 we use a heuristic based on lookaheads [20]. 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.

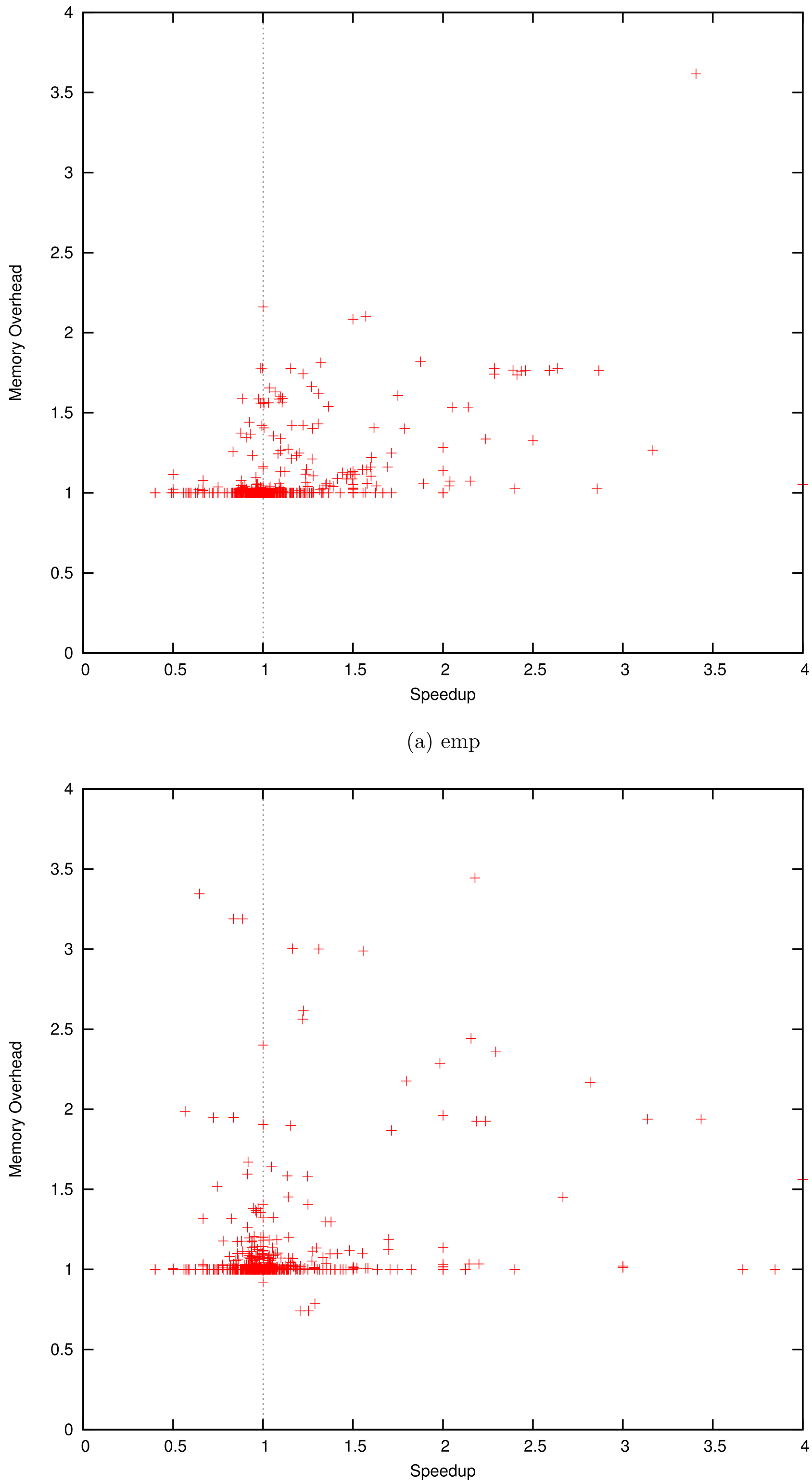
Table 4 presents the results averaged on all instances solved. The runtimes are reported relative to the time of A^* with WMD (with no lookahead), which generated 1,886,397 nodes (not reported in the table). The first 3 columns of Table 4 show the results for A^* with the lookahead heuristic for different 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 overhead of computing the heuristic eventually outweighs the savings achieved by 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^*. We used domain-specific parameter estimation to set good values for τ, p_{h_2}, t_2 for each lookahead depth.

The parameters were tuned manually on 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.

Please cite this article in press as: E. Karpas et al., Rational deployment of multiple heuristics in optimal state-space search, Artif. Intell. (2017), https://doi.org/10.1016/j.artint.2017.11.001



(a) emp

(b) T1

Fig. 3. Speedup vs. memory overhead for RLA^* vs. LA^*.

Please cite this article in press as: E. Karpas et al., Rational deployment of multiple heuristics in optimal state-space search, Artif. Intell. (2017), https://doi.org/10.1016/j.artint.2017.11.001

Table 4
Weighted 15 puzzle: comparison of $A^*_{\max}$, lazy A^*, and rational lazy A^*.

Lookahead	A^*		LA^*				RLA^* (using Eq. (6))				
	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 5
Summary of results for IDA^* algorithms in planning domains.

Algorithm	Coverage	Avg search time score	Expansions	h_2 ratio
lm	440	15.9	10,614.63	
lmcut	467	18.52	1,574.53	
max	497	19.24	**999.24**	
selmax	461	17.05	3,951.16	
lazy	505	20.09	999.86	0.54
emp	501	19.92	1,055.68	0.46
T1	**508**	**20.19**	1,009.96	0.48

6.2. Evaluation of $RLIDA^*$

We now turn to $RLIDA^*$, which we compare to IDA^* based search algorithms. We provide results for a set of planning domains, as well as for sliding tile puzzles (including the 15-puzzle) and for a container relocation problem.

6.2.1. Planning domains

As in the evaluation for RLA^*, we implemented $LIDA^*$ and $RLIDA^*$ on top of the Fast Downward planning system [24], and experimented with the admissible landmarks heuristic h_{LA} (used as h_1) [25], and the landmark cut heuristic h_{LMCUT} [26] (used as h_2). We also used the same set of domains.

We compare the performance of IDA^* using each of the heuristics individually (where lm denotes h_{LA} and lmcut denoted h_{LMCUT}), as well as with their maximum (denoted by max). We also evaluate selective max (selmax) [13], $LIDA^*$ (denoted lazy), and $RLIDA^*$ using the same methods of estimating p_{h_2}: empirical frequencies (*emp*), and the TS1 type system (*T1*). The search was limited to 3GB memory, and 30 minutes of CPU time on a single core of an Intel E5-2680 CPU with 64-bit Linux OS.

Table 5 provides a summary of the results, while per-domain results appear in the appendix, and are referenced here. We begin by looking at coverage — the number of planning problems solved by each algorithm in 30 minutes (per-domain results in Table A.19). These results show that $RLIDA^*$ using the TS1 type system solves more problems than any other search algorithm. Furthermore, $LIDA^*$ as well as $RLIDA^*$ using both parameter estimation methods solve more problems than any other approach. Note that selmax does extremely poorly here. As the decision rule that selmax uses was built for A^*, it is not surprising that it does not perform well in IDA^*.[7] Finally, it is worth mentioning that IDA^* and its variants are ill suited for the IPC benchmark domains, solving around 500 problems compared to over 800 solved by A^* and its variants. This is due to the large number of paths which reach the same state, which make IDA^* explore the same subtree repeatedly.

Next, we look at the time score for each search algorithm (per-domain results are available in Table A.20). Recall that the time score is computed from the time it took an algorithm to solve a problem. If the search time is less than 1 second, then the score is the maximum possible score — 100. The score then decreases logarithmically, until it reaches 0 at 1800 seconds. As the results show, RLA^* with the TS1 type system achieves a better (higher) time score than all other algorithms.

Fig. 4 shows the anytime performance of the different algorithms — the number of instances solved under different time limits. Note that both axes are in logscale. As the figure shows, the advantage of the $RLIDA^*$ variants is even more evident for shorter timeouts.

Examining the geometric mean of the number of expanded nodes in each search algorithm (Table A.21 provides per-domain results), we can see that, IDA^*_{MAX} has the fewest expanded nodes, and $LIDA^*$ is very close. The two variants of $RLIDA^*$ have more expanded nodes.

Finally, we look at the fraction of nodes for which h_2 (h_{LMCUT} in this case) was computed for $LIDA^*$ and $RLIDA^*$ (Table A.22 shows the per-domain results). Unsurprisingly, $LIDA^*$ has the highest numbers here.

[7] Adapting the decision rule of selmax to IDA^* is a non-trivial task, and is outside the scope of this paper.

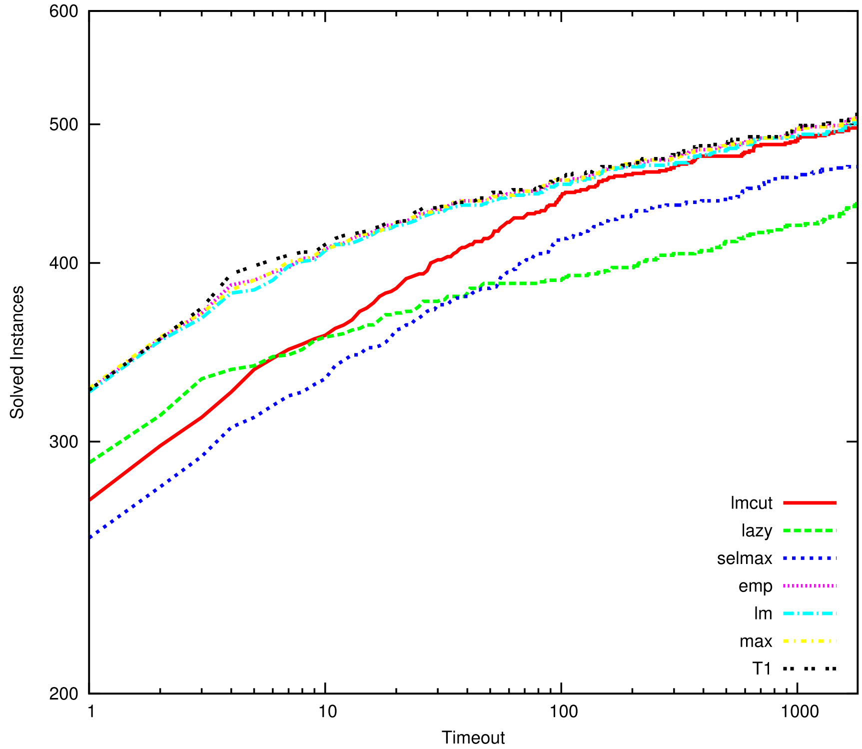


Fig. 4. Anytime plot for IDA^* algorithms in planning domains (both axes are in logscale).

Table 6
IDA* variants: results for 15 puzzle.

Algorithm	Time	Generated	h_2 total	h_2 helpful	PA
IDA^* (MD)	58.84	268,163,969			
IDA^* (LC)	40.08	**30,185,881**			
$LIDA^*$	32.85	**30,185,881**	21,886,093	6,561,972	30%
$RLIDA^*$ ($p_{h_2} = 0.3$)	**20.09**	47,783,019	8,106,832	4,413,050	54%
Clairvoyant	**12.66**	**30,185,881**	6,561,972	6,561,972	100%
$RLIDA^*$+TS1	33.90	49,314,132	14,265,984	9,315,480	65%
$RLIDA^*$+TS2	27.49	39,466,460	11,508,429	7,313,390	64%
$RLIDA^*$+TS1+TS3	35.08	65,269,319	7,908,331	5,351,080	68%
$RLIDA^*$+TS2+TS3	30.23	57,518,574	6,719,180	4,625,750	69%

Note that, unlike with RLA^*, the more informative parameter estimation method — using the TS1 type system — does better than the others. This is likely because with IDA^* we do have the exact threshold needed, and can exploit this information better.

6.2.2. Sliding-tile puzzles

We now examine the results on the 15-puzzle. We used as test instances the 98 out of Korf's 100 instances [8] that were solved in less than 20 minutes with standard IDA^* using the Manhattan Distance (MD) heuristic. As using the lookahead heuristic saves time only on OPEN list insertions and deletions, using it in IDA^* won't reduce runtime at all (as there is no OPEN list and nodes are expanded in a DFS manner). Thus the h_2 heuristic was the *linear-conflict heuristic* (LC) [28] which adds a value of 2 to MD for pairs of tiles that are in the same row (or the same column) as their respective goals but in a reversed order. One of these tiles will need to move away from the row (or column) to let the other pass.

Results for this problem set are shown in Table 6, listing average runtime in seconds, number of generated nodes, number of h_2 evaluations, number of *helpful* h_2 evaluations and the prediction accuracy (PA), which is the percentage of times calculating h_2 was indeed helpful. For $RLIDA^*$ we found the domain-specific parameter setting of $p_{h_2} = 0.3$ by manually testing different values for p_{h_2} on a set of problems and choosing the best one. Indeed, $RLIDA^*$ ($p_{h_2} = 0.3$) outperforms all other algorithms, an exception being the unrealizable "Clairvoyant" algorithm, which (using hindsight) evaluates h_2 only if it turned out to be helpful (the results for this "algorithm" are obtained by subtracting the time spent on non-helpful h_2 evaluations from the total time). The reason for presenting the clairvoyant algorithm is methodological: it is meant as a

Please cite this article in press as: E. Karpas et al., Rational deployment of multiple heuristics in optimal state-space search, Artif. Intell. (2017), https://doi.org/10.1016/j.artint.2017.11.001

Table 7
Weighted 15 puzzle.

Algorithm	Time	Generated	h_2 total	h_2 helpful	PA
IDA^* (WMD)	184.46	822,898,188			
IDA^* (WLC)	155.35	104,943,867			
IDA^*_{MAX}	149.50	104,943,867			
$LIDA^*$	112.74	104,943,890	65,660,207	12,549,104	19%
$RLIDA^*$ ($p_{h_2} = 0.3$)	**63.08**	137,881,842	21,564,188	8,871,727	**41%**
Clairvoyant	**40.36**	**104,943,890**	**12,549,104**	**12,549,104**	**100%**

Table 8
Weighted 3 by 5 puzzle.

Algorithm	Time	Generated	h_2 total	h_2 helpful	PA
IDA^* (WMD)	134.27	518,625,911			
IDA^* (WLC)	68.65	53,073,488			
IDA^*_{MAX}	71.28	53,073,488			
$LIDA^*$	59.89	53,073,499	36,000,253	8,218,490	23%
$RLIDA^*$ ($p_{h_2} = 0.3$)	**38.31**	77,199,730	12,104,449	6,564,049	**54%**
Clairvoyant	**27.99**	**53,073,499**	**8,218,490**	**8,218,490**	**100%**

Table 9
Weighted 3 by 6 puzzle.

Algorithm	Time	Generated	h_2 total	h_2 helpful	PA
IDA^* (WMD)	17.76	66,655,434			
IDA^* (WLC)	30.11	17,098,738			
IDA^*_{MAX}	31.08	17,098,738			
$LIDA^*$	21.99	17,098,746	10,308,664	1,473,548	14%
$RLIDA^*$ ($p_{h_2} = 0.3$)	**10.68**	21,053,303	2,882,141	1,007,129	**35%**
Clairvoyant	**7.17**	**17,098,746**	**1,473,548**	**1,473,548**	**100%**

yardstick for measuring performance of $RLIDA^*$ variants; once performance approaches that of an algorithm with perfect foreknowledge, there is presumably little room for further improvement.

Using type systems in $RLIDA^*$ increases the fraction of helpful evaluations of h_2 (Table 6), but the overhead and the added number of generated nodes results in an overall worse runtime performance. The additional improvements are therefore contra-indicated for the sliding tile problem. The rule used by $RLIDA^*$ with $p_{h_2} = 0.3$ was tantamount to having **opt-cond** being true only for nodes with $b(n) = 4$, which is in essence a very simple type system based on the branching factor. A more complicated type system is not justified here.

We have also experimented with the weighted version of the 15-puzzle, where the cost of moving each tile is equal to the number on the tile [4]. Table 7 shows similar results for 82 of the previous problem instances of the weighted 15 puzzle that were solved in 20 minutes by IDA^* (the weighted 15 puzzle is harder). In this domain, Rational Lazy IDA^* also achieves a significant speedup of a factor of 2 and is much closer to Clairvoyant than to $LIDA^*$. Attempts to improve upon this by adding a more complicated type system and relaxing Assumption III did not achieve any improvement, as shown in [29]. A complication in this variant compared to the unweighted version is that there are too many types, as the number of possible values of h_1 and h_2 is very large, but even limiting this number by binning did not achieve good results.

Finally, we ran the same algorithms on sliding tile puzzles with a different fraction of $b(n) = 4$ nodes, by "flattening" the 15-puzzle into a 3 by 5 puzzle (a "14-puzzle"), and into a 3 by 6 puzzle (a "17-puzzle"). Results for these variants of the weighted 15-puzzle (Tables 8, 9) show similar improvements for $RLIDA^*$.

6.2.3. Container relocation problem

The container relocation problem is an abstraction of a planning problem encountered in retrieving stacked containers for loading onto a ship in sea-ports [30]. We are given S stacks of containers, where each stack consists of up to T containers. The initial state has $N \leq S \times T$ containers, arbitrarily numbered from 1 to N. The rules of stacking and of moving containers are the same as for blocks in the blocks-world domain. The goal is to "retrieve" all containers in order of number, from 1 to N, i.e., to place them on a freight truck that takes the container away to be loaded onto a ship. The objective function to minimize is the number of container moves until all containers are gone. The complication comes from the fact that we can only "retrieve" a container if it is at the top of one of the stacks. Optimally solving this problem is NP-hard [30]. We use the version of the problem where each container ("block" in blocks-world terminology) is uniquely numbered, that a stack s that currently has T containers is "full" and no additional containers can be placed on s until some container is moved away from the top of s.

We used the LB_1 and LB_3 heuristics [30] as h_1 and h_2, respectively. For the sake of completeness, we review these heuristics here: Every container numbered X which is above at least one container Y with a number smaller than X must

Please cite this article in press as: E. Karpas et al., Rational deployment of multiple heuristics in optimal state-space search, Artif. Intell. (2017), https://doi.org/10.1016/j.artint.2017.11.001

Table 10
Container relocation: 49 "small" instances.

Small instances					
Algorithm	Time	Generated	h_2 total	h_2 helpful	PA
IDA^* (LB1)	336	853,094,579			
IDA^* (LB3)	967	**128,798,338**			
$LIDA^*$	366	**128,798,338**	44,527,029	19,564,237	44%
$RLIDA^*$ ($p_{h_2} = 0.3$)	337	233,077,220	27,628,566	13,575,017	49%
Clairvoyant	228	**128,798,338**	19,564,237	19,564,237	100%
$RLIDA^*$+TS1	327	166,781,023	35,931,245	21,292,089	59%
$RLIDA^*$+TS2	292	159,923,334	29,460,335	19,250,841	65%
$RLIDA^*$+TS1+TS3	207	318,146,242	9,001,091	6,653,964	74%
$RLIDA^*$+TS2+TS3	**201**	300,623,173	8,751,578	6,876,705	79%
Enhanced clairvoyant	138	**182,659,873**	44,527,029	9,737,977	22%

Table 11
Container relocation: all 63 instances.

All instances					
Algorithm	Time	Generated	h_2 total	h_2 helpful	PA
IDA^* (LB1)	1641	3,811,296,602			
IDA^* (LB3)	5761	**715,385,239**			
$LIDA^*$	2770	1,050,197,101	262,718,267	108,780,900	41%
$RLIDA^*$ ($p_{h_2} = 0.3$)	2764	1,073,191,297	254,291,856	106,366,804	42%
Clairvoyant	1656	1,050,197,101	108,780,900	108,780,900	100%
$RLIDA^*$+TS1	1924	1,502,283,957	138,031,927	87,720,923	64%
$RLIDA^*$+TS2	1967	1,337,796,749	146,545,466	94,988,940	65%
$RLIDA^*$+TS1+TS3	1311	2,378,791,883	24,727,136	17,317,818	70%
$RLIDA^*$+TS2+TS3	**1304**	2,342,370,343	23,816,116	18,299,499	77%
Enhanced clairvoyant	989	1,498,935,597	43,431,488	43,431,488	100%

be moved from its stack in order to allow Y to be retrieved. The number of such containers in a state can be computed quickly, and forms an admissible heuristic called LB_1. LB_3 adds one relocation for each container that must be relocated a second time as any place to which it is moved will block some other container. Computing LB_3 requires much more computation time (at least quadratic in the number of containers) than LB_1 (roughly linear time), and additionally its runtime depends heavily on the state.

In the experiments, we used the hardest tests out of those that were solved in less than 20 minutes with the LB_1 heuristic, from the CVS test suite described in [31,32].[8] Results are shown in Table 10. In this domain Rational Lazy IDA^* shows some performance improvement over Lazy IDA^*, even when p_{h_2} was assumed to be constant with $p_{h_2} = 0.3$. Furthermore, as these results show, using type systems to estimate p_{h_2} and p_{h_1} yields even better results. Specifically, the most significant improvement in this domain results from estimating p_{h_1}, which is only possible due to relaxing Assumption III.

We then conjectured that the timing differences should increase when we include harder problem instances, and added an additional 14 instances with a runtime greater than 20 minutes in IDA^*. The results (Table 11), including both the smaller and larger instances, were quite surprising, in some respects. First, $RLIDA^*$ was better than $LIDA^*$ as before, but now was actually substantially worse than just using IDA^* with only h_1. The reason is evident from examining the line "IDA^*(LB3)".[9] Although h_2 drastically reduces generated node numbers, its runtime with these larger problem instances outweighed its usefulness to such an extent that $RLIDA^*$ can at best approach IDA^* by evaluating h_2 very rarely.

But lifting Assumption III, that h_1 does not cause a cutoff in the children, achieves further speedup and the best performance of all. The difference between TS1 and TS2 does not appear significant: TS2 achieves better accuracy, but has higher overhead for metareasoning, so in the overall runtime performance TS2 is usually only slightly better than TS1, and sometimes slightly worse. An important thing to notice is that the PA also increases from 41% for $LIDA^*$ to 65% for TS2 and to 77% for TS2+TS3. That means that $RLIDA^*$ indeed makes "better" decisions than $LIDA^*$ and that relaxing Assumption III leads to even better decisions.

Note that in both container relocation results $RLIDA^*$ with the TS3 type system performs better than the clairvoyant scheme, which seems surprising. However, upon deeper examination, it turns out that even if h_2 cuts off a node n after h_1 fails to do so, it does not follow that one should evaluate h_2. For example, consider a node n with $b(n)$ children, where h_2 cuts off the search at node n, where h_1 would cut off the search at all of its $b(n)$ children. Then, if evaluating $h_2(n)$ is more expensive than computing h_1 for all $b(n)$ children, then bypassing h_2 may be better than evaluating it. $RLIDA^*$ takes such

[8] Retrieved from http://iwi.econ.uni-hamburg.de/IWIWeb/Default.aspx?tabId=1083&tabindex=4.

[9] In the larger instances, there appears to be a discrepancy in number of generated node numbers in the table. The discrepancy is due to timeouts when running "IDA^*(LB3)".

Please cite this article in press as: E. Karpas et al., Rational deployment of multiple heuristics in optimal state-space search, Artif. Intell. (2017), https://doi.org/10.1016/j.artint.2017.11.001

Table 12
Weighted 15 puzzle – number of extra iterations in $LIDA^*$.

Number of extra iterations	Number of instances
0	49
1	19
2	5
3	7
4	2

Table 13
Weighted 15 puzzle – locations of extra iterations in $LIDA^*$.

Part/16	Percentage of locations
1	31%
2	76%
4	100%

cases into account, whereas the clairvoyant scheme does not. In other words, knowing the future alone is insufficient if you do not use the information to reduce search time (rather than only to evaluate h_2 iff it is helpful), and this version of the clairvoyant algorithm fails to provide the needed yardstick.

We thus implemented an enhanced clairvoyant algorithm, which computes h_2 if it is helpful *and* if it is faster than computing h_1 for all of n's successors and h_1 will prune the successors. In other words, the enhanced clairvoyant applies the rational decision rule given perfect knowledge of not just node n but also its successors, that is, a 1-step lookahead. This new clairvoyant algorithm is a better yardstick for measuring performance of $RLIDA^*$.

6.2.4. Extra iterations for $LIDA^*$

In Section 2.3, we mentioned that $LIDA^*$ can lead to performing more iterations than IDA^*_{MAX}, although we expect this to happen rarely and have little impact. We now examine this claim empirically. We compared the number of iterations and threshold values between a problem solved by $LIDA^*$ and a problem solved by IDA^*_{MAX}. For the 15-puzzle and the container relocation problems, the number of iterations was equal, and the iteration thresholds were identical between the solvers (if a solver timed out we only compared threshold values up to that point). This is not surprising, as the threshold values in these problems are all relatively small integers, and therefore "gaps" resulting in potential additional iterations are unlikely to occur.

However, for the weighted 15-puzzle the number of iterations was different, as expected, due to the much larger number of possible threshold values. Table 12 shows the difference in the number of iterations between the two algorithms. As one can see in most of the instances there were no extra iterations, yet in a significant number of instances, there was at least one extra iteration.

It is important to know in which part of the search such extra iterations took place, as this tends to have a major impact on the runtime. We analyzed all the extra iterations – iterations with thresholds that did not exist in IDA^* but did exist in $LIDA^*$. It is worth mentioning that other than these iterations, all other threshold values were identical, i.e., there were no threshold values that existed in $LIDA^*$ but did not exist in IDA^*. Table 13 shows the extra iteration locations. 31% of the extra iterations were in the first 1/16 of the search (if a search process has 96 iterations, the first 1/16 of the search is the first 6 iterations), almost all of the extra iterations were in the first 1/8 of the search (76%), and all the extra iterations were in the first 1/4 of the search.

As previously mentioned in Section 2.3, in order for an extra iteration $\#i$ with value v to happen, all nodes that have an h_2 value of v in iteration $\#i-1$ have to be pruned by their h_1 value. The extra iterations did not appear to result in significant additional runtime, likely because the number of nodes grows exponentially between iterations in the problem we considered here.

7. Future work

A natural direction to extend the work done in this paper is to handle more than two heuristics, as done for the Lazy algorithms in Section 3.3. Non trivial issues arise, and multiple heuristics deserve a study beyond the scope of this paper.

Consider the simple setting, where we are actually committed to doing LA^*, but are free to choose the order of evaluating the heuristics. Suppose further that we know the exact runtime t_i of each heuristic, and the probability p_{h_i} that it will be helpful, given the current state of the search (including, possibly, results obtained by previous evaluations of heuristics). Suppose furthermore that the time to insert and remove a node from the open list is negligible. Under all these simplifying assumptions, the problem of optimally ordering the heuristics is equivalent to optimal test ordering, which is NP-hard [33, 34]. However, if we additionally restrict the distribution over helpfulness of heuristics to be independent, only then is there a simple optimal ordering, which is to compute the heuristics in non-increasing order of $\frac{p_{h_i}}{t_i}$. This latter scheme is what we recommend when running LA^* with multiple heuristics.

Please cite this article in press as: E. Karpas et al., Rational deployment of multiple heuristics in optimal state-space search, Artif. Intell. (2017), https://doi.org/10.1016/j.artint.2017.11.001

The above discussion on ordering heuristics is relevant in two ways. First, trying to define a meta-level optimized RLA^* and $RLIDA^*$ in such settings will result in metareasoning problems harder than the optimal test ordering problem, which is already NP-hard. Thus, ad-hoc schemes such as the naive one just suggested may be the only *practical* way to proceed. Second, note how the optimal ordering argument may affect even the case of two heuristics. We assumed that $t_1 < t_2$, and that $p_{h_1} < p_{h_2}$. Then we stated that it makes sense to start off with h_1. However, it is still possible that $\frac{p_{h_2}}{t_2} > \frac{p_{h_1}}{t_1}$, in which case it is better to start with h_2 if they are independent, as well as in cases where h_2 dominates h_1.

One naive way of generalizing RLA^* and $RLIDA^*$ to multiple heuristics is to use the same rational decision rule on pairs of consecutive heuristics, where heuristics are ordered according to their runtime. Once one of the heuristics is bypassed, we automatically bypass all the more expensive heuristics. It is important to note that in such a simple setting our decision rule must now consider that bypassing a heuristic could lead to potential losses from bypassing an even more expensive (and, most likely, informative) heuristic down the line.

Furthermore, even if we were able to define a rational decision rule, other practical issues arise when trying to estimate parameters when there are n heuristics. For example, a naive generalization of our type systems based approach to n heuristics would require n^2 type systems. This could be alleviated by defining the type systems differently, for example by only using the value of h_1 as a feature to predict whether $h_2 \dots h_n$ are helpful, or using only h_{i-1} to predict whether h_i will be helpful. However, it is not clear what the best solution would be, and whether there is significant benefit in allowing type systems to use multiple heuristics as features.

Another question is whether our methods can be effective if only *one* nontrivial admissible heuristic h is available. It seems possible, by using 0 as the value of h_1, and h as h_2, running RLA^* this way would make it behave like uniform cost search (AKA Dijkstra's algorithm) at some points in the search. This is actually in agreement with the known observation [35] that sometimes it is more efficient to start out with uniform cost search when using A^*, thereby saving time needed to compute h in many nodes that would be expanded anyway, especially at the beginning of the search.

Some questions regarding possible improvements to RLA^* include incorporating information about the memory limit into the decision rule, obtaining a better proxy for the threshold, incorporating cost predictors (e.g., [36,37]) into the decision rule, and looking at the problem of predicting whether h_2 will be helpful as online learning with delayed feedback (e.g., [38]). Other interesting problems include using rational metareasoning to control decisions in other variants of A^*, and adapting RA* [14] and GHS [15] to choose a set of heuristics to combine using RLA^* or $RLIDA^*$, instead of A^*_{MAX}.

8. Conclusions

We discussed two schemes for decreasing the computational resources used to evaluate heuristics. LA^* and $LIDA^*$ are very simple to implement, and are as informed as A^*_{MAX} and IDA^*_{MAX}, respectively, with the caveat that $LIDA^*$ could lead to extra iterations. While these can significantly speed up the search in some cases, such as when t_2 dominates the other time costs, additional benefit can be gained by using the rational metareasoning framework [1] to decide when computing the expensive heuristic is worth the time spent on it. The resulting algorithms, RLA^* and $RLIDA^*$, achieve better performance than their non-rational counterparts on many different problems.

In particular, RLA^* is simpler to implement than its direct competitor, selective 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 $f_1(n) \leq C^*$. In contrast, although selective max uses a much more complicated decision rule, it chooses which heuristic to compute when n is first generated, and does not know whether h_1 will be informative enough to prune n. RLA^* outperforms selective max in our planning experiments.

Furthermore, $RLIDA^*$ can make even better decisions than RLA^*, because it knows the "target value" for f_2 — the current threshold, T, in addition to the value of $f_1(n)$. This also means that $RLIDA^*$ knows whether h_2 is helpful immediately after evaluating it, while RLA^* can only know that h_2 was not helpful for a node it expands, but will know if h_2 is helpful only when the search terminates.

Additionally, the decision rule for RLA^* and $RLIDA^*$ only considers search time, not memory. This not an issue for $RLIDA^*$, which only requires linear memory, but could cause RLA^* to expand too many nodes and exhaust available memory.

Our analysis and empirical evaluation also shed some light on the question of when using rational metareasoning is worthwhile: Whenever we have multiple heuristics, where one of the heuristics is informative but expensive to compute, using rational metareasoning is likely a good idea. In fact, in some cases the informative heuristic is so expensive that using it only becomes beneficial in conjunction with rational metareasoning. However, if we only have heuristics which are relatively cheap to compute, the overhead of rational metareasoning, as well as the probability of making a mistake, are not worth the potential benefit. In such cases, LA^* or $LIDA^*$ are probably better choices.

Acknowledgements

This research was supported by the Israeli Science Foundation (ISF) under grants #417/13 and #844/17 to Ariel Felner and Solomon Eyal Shimony.

Appendix A. Detailed empirical results for planning domains

Table A.14
Coverage for A^* algorithms in planning domains.

coverage	lm	lmcut	max	selmax	lazy	emp	T1
airport (50)	30	28	30	30	30	30	**31**
barman-opt11-strips (20)	**4**	**4**	**4**	**4**	**4**	**4**	**4**
blocks (35)	19	**28**	**28**	**28**	**28**	**28**	**28**
depot (22)	6	**7**	**7**	**7**	**7**	**7**	**7**
driverlog (20)	10	13	**14**	**14**	**14**	**14**	**14**
elevators-opt08-strips (30)	12	**22**	**22**	**22**	**22**	**22**	**22**
elevators-opt11-strips (20)	10	**18**	**18**	**18**	**18**	**18**	**18**
floortile-opt11-strips (20)	2	**7**	**7**	**7**	**7**	**7**	2
freecell (80)	**61**	15	43	59	49	49	**61**
grid (5)	**2**	**2**	**2**	**2**	**2**	**2**	**2**
gripper (20)	**7**	**7**	**7**	**7**	**7**	**7**	**7**
ipc2014-opt-Barman (14)	**0**	**0**	**0**	**0**	**0**	**0**	**0**
ipc2014-opt-CaveDiving (3)	**3**	**3**	**3**	**3**	**3**	**3**	**3**
ipc2014-opt-ChildSnack (20)	**0**	**0**	**0**	**0**	**0**	**0**	**0**
ipc2014-opt-Floortile (20)	0	**5**	**5**	**5**	**5**	**5**	0
ipc2014-opt-GED (20)	**15**	**15**	**15**	14	**15**	**15**	**15**
ipc2014-opt-Hiking (20)	**11**	9	9	9	9	9	10
ipc2014-opt-Maintenance (5)	**5**	**5**	**5**	**5**	**5**	**5**	**5**
ipc2014-opt-Openstacks (20)	**3**	**3**	**3**	**3**	**3**	**3**	**3**
ipc2014-opt-Parking (20)	0	3	3	3	**4**	**4**	**4**
ipc2014-opt-Tetris (17)	**9**	5	5	**9**	6	6	6
ipc2014-opt-Tidybot (20)	**8**	7	7	7	7	7	7
ipc2014-opt-Transport (20)	**6**	**6**	**6**	**6**	**6**	**6**	**6**
ipc2014-opt-Visitall (20)	4	**5**	**5**	**5**	**5**	**5**	**5**
logistics00 (28)	10	**20**	**20**	**20**	**20**	**20**	19
logistics98 (35)	2	**6**	**6**	**6**	**6**	**6**	**6**
miconic (150)	**143**	141	141	**143**	142	142	142
mprime (35)	21	**22**	**22**	**22**	**22**	**22**	**22**
mystery (30)	15	**17**	**17**	**17**	**17**	**17**	**17**
nomystery-opt11-strips (20)	18	14	18	18	18	18	**19**
openstacks-opt08-strips (30)	20	**21**	19	19	19	19	19
openstacks-opt11-strips (20)	15	**16**	14	15	14	14	14
openstacks-strips (30)	**7**	**7**	**7**	**7**	**7**	**7**	**7**
parcprinter-08-strips (30)	15	**18**	**18**	15	**18**	**18**	**18**
parcprinter-opt11-strips (20)	11	**13**	**13**	11	**13**	**13**	**13**
parking-opt11-strips (20)	1	2	2	3	**4**	3	3
pathways-noneg (30)	4	**5**	**5**	**5**	**5**	**5**	**5**
pegsol-08-strips (30)	**27**	**27**	**27**	**27**	**27**	**27**	**27**
pegsol-opt11-strips (20)	**17**	**17**	**17**	**17**	**17**	**17**	**17**
pipesworld-notankage (50)	**19**	17	17	**19**	17	17	17
pipesworld-tankage (50)	**13**	11	12	12	12	12	12
psr-small (50)	**49**	**49**	**49**	**49**	**49**	**49**	**49**
rovers (40)	8	7	8	8	**9**	8	8
satellite (36)	7	7	7	7	**8**	**8**	**8**
scanalyzer-08-strips (30)	9	**15**	**15**	13	**15**	**15**	**15**
scanalyzer-opt11-strips (20)	6	**12**	**12**	10	**12**	**12**	**12**
sokoban-opt08-strips (30)	24	28	29	29	29	**30**	25
sokoban-opt11-strips (20)	19	**20**	**20**	**20**	**20**	**20**	19
tidybot-opt11-strips (20)	**14**	**14**	**14**	**14**	**14**	**14**	**14**
tpp (30)	**6**	**6**	**6**	**6**	**6**	**6**	**6**
transport-opt08-strips (30)	**11**	**11**	**11**	**11**	**11**	**11**	**11**
transport-opt11-strips (20)	**6**	**6**	**6**	**6**	**6**	**6**	**6**
trucks-strips (30)	9	**10**	**10**	**10**	**10**	**10**	**10**
visitall-opt11-strips (20)	**10**	**10**	**10**	**10**	**10**	**10**	**10**
woodworking-opt08-strips (30)	15	16	16	15	**17**	**17**	**17**
woodworking-opt11-strips (20)	10	11	11	10	**12**	**12**	**12**
zenotravel (20)	9	**13**	12	12	**13**	**13**	**13**
Sum (1615)	797	826	859	873	**875**	874	872

Table A.15
Search time score for A^* algorithms in planning domains.

score	lm	lmcut	max	selmax	lazy	emp	T1
airport (50)	**51.23**	44.38	45.61	49.96	47.62	47.45	50.79
barman-opt11-strips (20)	**7.00**	4.01	3.71	4.22	4.01	4.45	4.36
blocks (35)	49.97	**65.38**	64.93	65.18	65.25	64.99	65.15
depot (22)	14.78	18.09	18.40	18.18	19.96	19.86	**20.19**
driverlog (20)	43.87	53.84	54.70	54.35	**56.95**	56.88	56.82
elevators-opt08-strips (30)	26.88	38.89	38.09	38.66	**42.77**	42.70	42.60
elevators-opt11-strips (20)	30.25	45.83	44.69	45.78	**51.27**	50.92	50.72
floortile-opt11-strips (20)	7.16	**18.48**	17.94	18.16	18.09	18.07	8.97
freecell (80)	**57.44**	9.47	36.70	50.91	43.73	43.81	57.38
grid (5)	**34.76**	23.83	26.35	34.44	27.75	27.55	28.92
gripper (20)	**27.96**	23.72	23.76	27.12	23.90	23.99	24.54
ipc2014-opt-Barman (14)	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
ipc2014-opt-CaveDiving (3)	**8.77**	5.95	5.64	7.49	6.42	6.36	7.94
ipc2014-opt-ChildSnack (20)	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
ipc2014-opt-Floortile (20)	0.00	**7.82**	7.24	7.58	7.37	7.27	0.00
ipc2014-opt-GED (20)	57.61	56.64	55.37	53.56	55.85	**60.75**	56.58
ipc2014-opt-Hiking (20)	**38.01**	28.48	28.87	32.59	31.07	30.85	31.98
ipc2014-opt-Maintenance (5)	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
ipc2014-opt-Openstacks (20)	3.80	**4.19**	2.67	3.22	2.95	2.93	2.92
ipc2014-opt-Parking (20)	0.00	2.04	2.55	**5.21**	4.60	4.55	4.41
ipc2014-opt-Tetris (17)	**31.36**	14.93	15.54	26.84	17.87	17.90	18.54
ipc2014-opt-Tidybot (20)	**14.49**	6.03	6.35	10.57	6.73	6.51	6.50
ipc2014-opt-Transport (20)	**15.32**	14.74	14.56	14.54	15.23	15.13	14.98
ipc2014-opt-Visitall (20)	16.14	**17.54**	17.17	17.51	17.43	17.48	17.50
logistics00 (28)	32.66	56.69	55.98	51.67	**58.00**	57.74	56.18
logistics98 (35)	5.03	13.23	13.08	12.89	13.71	**13.73**	11.82
miconic (150)	**93.45**	81.98	82.02	87.11	92.31	92.34	92.32
mprime (35)	53.25	51.95	53.16	**55.69**	54.84	54.87	54.77
mystery (30)	52.96	54.83	54.60	**54.86**	54.65	54.49	54.57
nomystery-opt11-strips (20)	**78.18**	56.87	67.26	77.54	75.30	75.23	77.10
openstacks-opt08-strips (30)	39.87	**41.80**	36.79	39.03	37.84	37.68	37.42
openstacks-opt11-strips (20)	34.92	**38.20**	30.25	32.95	32.07	31.54	31.39
openstacks-strips (30)	**22.03**	19.08	19.98	21.67	20.24	20.26	21.98
parcprinter-08-strips (30)	39.35	**54.40**	53.89	38.10	53.97	54.01	51.95
parcprinter-opt11-strips (20)	39.06	**56.40**	55.90	37.33	56.11	55.94	53.13
parking-opt11-strips (20)	2.74	2.77	3.22	4.34	5.06	**5.17**	5.06
pathways-noneg (30)	13.33	**14.82**	14.76	14.74	14.79	14.77	14.75
pegsol-08-strips (30)	78.20	**78.52**	76.66	77.43	76.70	77.43	76.53
pegsol-opt11-strips (20)	64.23	**64.95**	62.43	63.25	62.74	63.16	62.13
pipesworld-notankage (50)	**28.73**	22.91	23.60	25.98	24.43	24.50	24.56
pipesworld-tankage (50)	**17.79**	13.20	14.10	14.67	15.54	15.58	15.47
psr-small (50)	**95.69**	92.54	92.12	93.67	92.92	93.28	92.81
rovers (40)	**16.89**	15.31	15.44	16.28	16.34	16.27	16.49
satellite (36)	17.66	16.74	17.07	17.11	18.02	**18.14**	18.03
scanalyzer-08-strips (30)	22.10	36.08	35.76	29.13	**36.82**	35.90	36.43
scanalyzer-opt11-strips (20)	18.06	39.63	39.54	29.03	40.52	**40.68**	**40.68**
sokoban-opt08-strips (30)	54.45	64.19	63.88	64.56	64.16	**68.29**	59.16
sokoban-opt11-strips (20)	63.13	73.10	71.86	72.46	72.22	**77.02**	68.98
tidybot-opt11-strips (20)	**49.39**	35.34	35.11	41.52	36.19	35.99	35.81
tpp (30)	**19.26**	19.10	19.05	18.97	19.10	19.07	19.04
transport-opt08-strips (30)	30.28	30.39	30.49	29.76	30.79	30.78	**30.87**
transport-opt11-strips (20)	20.24	20.64	20.53	19.39	**21.43**	21.15	21.14
trucks-strips (30)	19.36	**22.52**	22.17	22.29	22.30	22.32	22.28
visitall-opt11-strips (20)	45.51	45.24	45.11	45.39	**45.52**	45.44	45.42
woodworking-opt08-strips (30)	36.95	41.34	40.98	35.93	42.29	42.20	**42.33**
woodworking-opt11-strips (20)	30.45	37.03	36.64	28.59	**38.59**	38.16	38.44
zenotravel (20)	40.68	48.21	47.95	48.13	49.39	**49.57**	49.46
Average (1615)	32.98	33.87	34.18	34.61	35.55	**35.74**	35.35

Table A.16
Number of expansions for A^* algorithms in planning domains.

expansions	lm	lmcut	max	selmax	lazy	emp	T1
airport (28)	601.94	**204.45**	222.65	554.43	222.65	232.98	436.50
barman-opt11-strips (4)	5,621,156.49	1,265,904.14	1,263,198.17	1,420,926.76	**1,263,186.90**	1,459,457.38	1,872,354.58
blocks (19)	679.55	124.15	124.08	188.15	**123.34**	123.44	123.80
depot (6)	147,557.91	6,468.31	**6,087.91**	8,543.10	6,165.93	6,165.93	6,401.78
driverlog (10)	8,157.38	274.62	261.58	366.79	**256.60**	**256.60**	256.86
elevators-opt08-strips (12)	105,352.48	3,256.96	**3,137.88**	3,261.25	**3,137.88**	**3,137.88**	**3,137.88**
elevators-opt11-strips (10)	210,471.85	5,946.75	**5,722.65**	5,929.04	**5,722.65**	**5,722.65**	**5,722.65**
floortile-opt11-strips (2)	196,061.77	**1,739.60**	**1,739.60**	1,866.64	1,740.42	1,740.42	30,089.22
freecell (15)	19.43	16,361.15	**19.40**	100.25	**19.40**	**19.40**	19.43
grid (2)	2,946.87	2,039.90	1,210.33	2,946.87	**1,209.75**	1,210.31	1,269.65
gripper (7)	**47,081.71**	49,623.60	**47,081.71**	47,103.22	**47,081.71**	**47,081.71**	**47,081.71**
ipc2014-opt-CaveDiving (3)	1,229,877.94	**502,662.88**	**502,662.88**	1,229,877.94	**502,662.88**	714,798.94	1,187,293.37
ipc2014-opt-GED (14)	21,163.09	**4,615.32**	**4,615.32**	5,103.63	**4,615.32**	7,291.41	4,699.90
ipc2014-opt-Hiking (9)	21,633.25	20,455.50	**17,845.41**	20,372.84	17,845.50	17,849.29	18,468.61
ipc2014-opt-Maintenance (5)	39.91	33.15	**30.15**	36.37	32.12	32.12	38.78
ipc2014-opt-Openstacks (3)	3,300,439.58	**2,346,255.21**	**2,346,255.21**	2,968,191.35	**2,346,255.21**	**2,346,255.21**	**2,346,255.21**
ipc2014-opt-Tetris (5)	5,939.03	3,919.73	3,533.40	5,792.20	**3,533.33**	3,533.34	3,608.36
ipc2014-opt-Tidybot (7)	87,502.35	6,161.11	**6,090.07**	27,674.20	**6,090.07**	6,236.75	6,607.96
ipc2014-opt-Transport (6)	994,777.88	15,269.17	15,157.34	499,709.02	**15,153.55**	**15,153.55**	18,608.97
ipc2014-opt-Visitall (4)	132,624.53	24,900.35	22,183.60	32,363.17	**22,183.51**	**22,183.51**	25,416.35
logistics00 (10)	2,576.88	**160.05**	**160.05**	280.69	**160.05**	**160.05**	165.15
logistics98 (2)	6,153.31	**89.85**	**89.85**	106.99	**89.85**	**89.85**	**89.85**
miconic (141)	**109.14**	**109.14**	**109.14**	**109.14**	**109.14**	**109.14**	**109.14**
mprime (21)	670.81	78.70	**47.52**	78.89	55.13	55.13	55.13
mystery (18)	319.11	27.28	**26.45**	33.56	26.96	26.99	27.87
nomystery-opt11-strips (14)	247.44	423.16	**162.61**	234.35	165.12	166.11	176.03
openstacks-opt08-strips (19)	136,014.49	**117,130.49**	**117,130.49**	123,795.93	**117,130.49**	117,922.64	**117,130.49**
openstacks-opt11-strips (14)	606,248.94	**503,402.68**	**503,402.68**	548,449.83	**503,402.68**	504,085.78	**503,402.68**
openstacks-strips (7)	2,613.79	8,862.84	**2,572.71**	2,769.60	**2,572.71**	2,613.79	2,601.09
parcprinter-08-strips (15)	4,851.49	198.36	**198.30**	4,850.14	**198.30**	207.59	539.18
parcprinter-opt11-strips (11)	36,882.02	536.50	**536.29**	36,868.02	**536.29**	540.62	2,098.20
parking-opt11-strips (1)	57,211.00	3,391.00	**2,420.00**	5,603.00	**2,420.00**	**2,420.00**	2,586.00
pathways-noneg (4)	1,033.05	**38.48**	**38.48**	**38.48**	**38.48**	39.26	195.89
pegsol-08-strips (27)	18,162.30	4,590.93	4,567.51	4,895.04	**4,562.72**	7,384.24	5,124.26
pegsol-opt11-strips (17)	180,026.89	43,444.92	43,143.88	43,800.81	**43,086.65**	64,381.33	48,925.92
pipesworld-notankage (17)	8,064.14	3,339.01	**2,092.49**	6,470.42	2,115.62	2,119.18	2,381.33
pipesworld-tankage (11)	6,850.86	4,683.98	2,807.97	3,685.90	**2,804.41**	2,902.31	2,898.73
psr-small (49)	502.23	**386.24**	**386.24**	454.87	386.30	418.27	387.81
rovers (7)	657.09	439.94	302.98	351.82	**289.88**	**289.88**	**289.88**
satellite (7)	1,189.83	308.29	**233.35**	292.18	234.12	234.12	234.31
scanalyzer-08-strips (8)	8,370.39	158.49	**158.23**	286.11	**158.23**	**158.23**	158.79
scanalyzer-opt11-strips (5)	70,002.61	1,095.74	**1,092.91**	1,742.76	**1,092.91**	**1,092.91**	1,099.10
sokoban-opt08-strips (23)	171,029.66	11,097.34	11,070.85	23,182.92	**11,070.81**	12,773.15	53,072.97
sokoban-opt11-strips (19)	354,406.00	20,290.70	20,239.81	41,511.12	**20,239.73**	23,735.55	126,902.32
tidybot-opt11-strips (14)	11,766.14	1,729.18	**1,694.68**	5,329.62	**1,694.68**	1,798.96	1,850.97
tpp (6)	107.20	**63.15**	**63.15**	72.96	**63.15**	**63.15**	**63.15**
transport-opt08-strips (11)	8,676.74	420.58	**418.23**	8,336.64	418.24	418.24	433.61
transport-opt11-strips (6)	225,933.52	3,687.90	**3,650.25**	231,426.07	3,650.36	3,650.36	3,898.44
trucks-strips (9)	152,021.34	4,556.40	4,493.97	4,677.25	**4,477.82**	**4,477.82**	4,496.86
visitall-opt11-strips (10)	466.77	256.85	**204.74**	259.06	204.76	204.76	216.96
woodworking-opt08-strips (14)	7,029.98	**146.60**	**146.60**	6,711.05	153.88	153.88	150.66
woodworking-opt11-strips (9)	46,944.68	**618.39**	**618.39**	46,921.77	657.37	657.37	636.12
zenotravel (9)	402.49	39.65	**39.63**	57.51	45.50	45.83	45.51
Geometric mean (726)	12,057.25	1,978.56	**1,551.71**	3,581.64	1,564.77	1,640.27	2,028.68

Table A.17
Peak memory usage (in KB) for A^* algorithms in planning domains.

memory	lm	lmcut	max	selmax	lazy	emp	T1
airport (28)	1,201,096	**209,432**	417,628	1,205,824	419,932	432,144	1,035,472
barman-opt11-strips (4)	1,909,588	**391,292**	621,032	792,180	655,676	879,732	1,086,840
blocks (19)	3,482,060	**64,980**	77,224	85,088	77,960	81,516	81,540
depot (6)	5,365,860	**120,320**	175,076	308,028	182,212	227,356	230,892
driverlog (10)	846,320	**35,460**	41,012	44,396	41,552	43,384	43,460
elevators-opt08-strips (12)	5,362,360	**106,284**	147,412	151,676	151,124	181,940	182,044
elevators-opt11-strips (10)	5,351,576	**99,436**	139,888	144,148	143,300	174,124	174,304
floortile-opt11-strips (2)	135,392	**7,620**	9,060	9,348	9,104	9,864	35,496
freecell (15)	**169,936**	236,244	175,720	177,524	176,060	176,208	176,312
grid (2)	37,816	**28,416**	29,356	38,508	29,688	31,564	32,216
gripper (7)	1,084,240	**637,468**	1,083,980	1,083,252	1,132,916	1,480,576	1,481,440
ipc2014-opt-CaveDiving (3)	610,120	**262,060**	432,244	609,824	453,828	599,144	801,340
ipc2014-opt-GED (14)	2,794,896	**535,776**	880,840	918,804	923,472	1,564,792	1,182,096
ipc2014-opt-Hiking (9)	246,756	**153,852**	213,308	267,756	221,368	278,936	290,524
ipc2014-opt-Maintenance (5)	17,044	**15,768**	17,604	18,640	17,308	17,560	18,780
ipc2014-opt-Openstacks (3)	3,780,600	**2,067,120**	3,679,784	3,839,624	3,827,444	4,849,528	4,849,440
ipc2014-opt-Tetris (5)	163,488	**88,860**	151,884	170,596	153,080	161,020	162,328
ipc2014-opt-Tidybot (7)	958,576	**342,440**	835,960	895,520	836,668	839,496	840,084
ipc2014-opt-Transport (6)	4,467,016	**90,284**	127,740	4,466,648	131,172	157,676	190,680
ipc2014-opt-Visitall (4)	392,964	**32,040**	45,100	57,132	46,808	57,196	59,116
logistics00 (10)	102,004	**31,036**	33,964	34,880	34,096	34,468	34,592
logistics98 (2)	22,300	**6,188**	6,716	6,720	6,712	6,712	6,712
miconic (141)	828,924	**609,820**	848,984	851,928	858,916	923,008	923,604
mprime (21)	3,062,088	**193,360**	259,272	334,368	261,924	277,472	277,372
mystery (18)	1,653,804	**162,984**	259,388	282,732	265,860	300,300	364,232
nomystery-opt11-strips (14)	87,472	**76,500**	91,324	92,044	91,500	92,044	92,168
openstacks-opt08-strips (19)	6,254,764	**3,317,952**	5,906,772	6,075,136	6,150,576	7,837,824	7,837,760
openstacks-opt11-strips (14)	6,233,184	**3,300,360**	5,885,652	6,057,452	6,128,628	7,814,524	7,814,224
openstacks-strips (7)	**49,520**	76,920	50,256	52,264	51,164	58,768	58,812
parcprinter-08-strips (15)	3,421,360	**156,164**	207,964	3,420,940	213,760	242,588	862,748
parcprinter-opt11-strips (11)	3,408,036	**144,060**	194,540	3,407,496	200,292	229,172	849,672
parking-opt11-strips (1)	63,228	**8,352**	14,400	17,048	14,424	14,976	15,184
pathways-noneg (4)	20,632	**12,364**	13,164	13,288	13,284	13,288	15,808
pegsol-08-strips (27)	1,372,876	**399,208**	631,276	705,376	658,508	807,480	845,132
pegsol-opt11-strips (17)	1,390,852	**379,348**	614,008	689,960	641,876	802,972	839,100
pipesworld-notankage (17)	1,842,584	**343,156**	466,744	1,944,356	471,400	562,940	1,164,124
pipesworld-tankage (11)	955,832	**236,704**	321,432	494,656	332,688	433,348	420,508
psr-small (49)	1,090,280	**600,972**	936,316	1,048,060	968,844	1,231,136	1,212,084
rovers (7)	110,892	**48,504**	52,264	60,832	53,776	63,192	63,124
satellite (7)	241,328	80,984	**61,024**	139,300	62,924	74,572	74,532
scanalyzer-08-strips (8)	1,558,484	**217,312**	359,768	361,460	376,500	482,200	486,820
scanalyzer-opt11-strips (5)	1,543,008	**207,180**	347,300	348,888	363,920	469,596	474,292
sokoban-opt08-strips (23)	7,348,856	**241,528**	379,184	1,577,924	394,220	545,292	5,554,144
sokoban-opt11-strips (19)	5,691,372	**222,652**	354,444	1,473,264	369,348	516,840	5,520,228
tidybot-opt11-strips (14)	1,013,196	**378,960**	889,688	961,956	890,696	897,164	898,084
tpp (6)	60,188	**23,248**	28,276	46,020	28,824	31,860	31,840
transport-opt08-strips (11)	961,068	**60,516**	71,428	966,364	71,564	76,704	78,096
transport-opt11-strips (6)	942,420	**44,360**	54,228	947,820	54,148	59,228	60,556
trucks-strips (9)	3,464,788	**109,348**	171,288	173,772	178,900	221,696	221,824
visitall-opt11-strips (10)	752,476	**93,964**	123,560	227,144	128,448	157,472	221,140
woodworking-opt08-strips (14)	4,593,712	**151,040**	230,816	4,598,148	239,328	286,076	279,352
woodworking-opt11-strips (9)	4,573,672	**135,308**	212,352	4,577,964	220,772	267,600	260,852
zenotravel (9)	3,042,284	**31,008**	36,008	36,940	36,388	37,388	37,284
Sum (726)	106,135,188	**17,626,512**	29,415,652	57,311,016	30,464,880	38,113,656	50,850,408

Table A.18
Fraction of nodes in which h_2 was evaluated for A^* algorithms in planning domains.

h_2 **ratio**	**lazy**	**emp**	**T1**
airport (30)	**0.50**	0.31	0.33
barman-opt11-strips (4)	**1.00**	0.79	0.65
blocks (28)	**0.81**	0.80	**0.81**
depot (7)	**0.74**	**0.74**	0.69
driverlog (14)	**0.54**	**0.54**	0.53
elevators-opt08-strips (22)	**0.57**	**0.57**	**0.57**
elevators-opt11-strips (18)	**0.58**	**0.58**	**0.58**
floortile-opt11-strips (2)	**0.88**	**0.88**	0.11
freecell (49)	**0.18**	0.17	0.05
grid (2)	**0.64**	**0.64**	0.54
gripper (7)	**0.93**	**0.93**	0.65
ipc2014-opt-CaveDiving (3)	**0.69**	0.55	0.04
ipc2014-opt-GED (15)	**0.96**	0.27	0.86
ipc2014-opt-Hiking (9)	**0.71**	**0.71**	0.49
ipc2014-opt-Maintenance (5)	**0.16**	**0.16**	0.13
ipc2014-opt-Openstacks (3)	**0.61**	**0.61**	**0.61**
ipc2014-opt-Parking (3)	**0.37**	**0.37**	0.26
ipc2014-opt-Tetris (6)	**0.53**	**0.53**	0.46
ipc2014-opt-Tidybot (7)	**0.98**	0.96	0.92
ipc2014-opt-Transport (6)	**0.93**	**0.93**	0.85
ipc2014-opt-Visitall (5)	**0.87**	**0.87**	0.77
logistics00 (19)	**0.49**	**0.49**	**0.49**
logistics98 (6)	0.54	0.54	**0.62**
miconic (142)	**0.14**	0.13	**0.14**
mprime (22)	**0.42**	**0.42**	**0.42**
mystery (21)	**0.57**	0.55	0.53
nomystery-opt11-strips (18)	**0.29**	0.28	0.22
openstacks-opt08-strips (19)	**0.55**	**0.55**	**0.55**
openstacks-opt11-strips (14)	**0.56**	**0.56**	**0.56**
openstacks-strips (7)	**0.65**	0.49	0.10
parcprinter-08-strips (18)	**0.82**	0.74	0.62
parcprinter-opt11-strips (13)	**0.85**	0.83	0.56
parking-opt11-strips (3)	**0.35**	**0.35**	0.34
pathways-noneg (5)	**0.73**	**0.73**	0.47
pegsol-08-strips (27)	**0.97**	0.55	0.85
pegsol-opt11-strips (17)	**0.96**	0.60	0.84
pipesworld-notankage (17)	**0.56**	**0.56**	0.51
pipesworld-tankage (12)	**0.48**	0.45	0.44
psr-small (49)	**0.84**	0.51	0.82
rovers (8)	**0.33**	**0.33**	0.29
satellite (8)	**0.34**	**0.34**	**0.34**
scanalyzer-08-strips (15)	**0.95**	**0.95**	**0.95**
scanalyzer-opt11-strips (12)	**0.95**	**0.95**	**0.95**
sokoban-opt08-strips (25)	**0.95**	0.61	0.27
sokoban-opt11-strips (19)	**0.97**	0.57	0.20
tidybot-opt11-strips (14)	**0.90**	0.85	0.83
tpp (6)	**0.63**	0.55	**0.63**
transport-opt08-strips (11)	**0.82**	**0.82**	0.80
transport-opt11-strips (6)	**0.89**	**0.89**	0.86
trucks-strips (10)	**0.92**	**0.92**	0.91
visitall-opt11-strips (10)	**0.68**	**0.68**	0.62
woodworking-opt08-strips (17)	**0.44**	**0.44**	0.43
woodworking-opt11-strips (12)	**0.46**	**0.46**	**0.46**
zenotravel (13)	**0.55**	0.53	**0.55**
Average (860)	**0.66**	0.59	0.54

Table A.19
Coverage for IDA^* algorithms in planning domains.

coverage	lm	lmcut	max	selmax	lazy	emp	T1
airport (50)	18	**21**	**21**	19	**21**	**21**	**21**
barman-opt11-strips (20)	**0**	**0**	**0**	**0**	**0**	**0**	**0**
blocks (35)	18	**19**	**19**	**19**	**19**	**19**	**19**
depot (22)	**2**	**2**	**2**	**2**	**2**	**2**	**2**
driverlog (20)	6	**8**	**8**	**8**	**8**	**8**	**8**
elevators-opt08-strips (30)	**0**	**0**	**0**	**0**	**0**	**0**	**0**
elevators-opt11-strips (20)	**0**	**0**	**0**	**0**	**0**	**0**	**0**
floortile-opt11-strips (20)	0	**1**	**1**	**1**	**1**	**1**	0
freecell (80)	**32**	5	25	31	28	28	**32**
grid (5)	**1**	**1**	**1**	**1**	**1**	**1**	**1**
gripper (20)	**3**	2	2	**3**	2	2	**3**
ipc2014-opt-Barman (14)	**0**	**0**	**0**	**0**	**0**	**0**	**0**
ipc2014-opt-CaveDiving (3)	**0**	**0**	**0**	**0**	**0**	**0**	**0**
ipc2014-opt-ChildSnack (20)	**0**	**0**	**0**	**0**	**0**	**0**	**0**
ipc2014-opt-Floortile (20)	**0**	**0**	**0**	**0**	**0**	**0**	**0**
ipc2014-opt-GED (20)	12	**13**	**13**	12	**13**	**13**	**13**
ipc2014-opt-Hiking (20)	**2**	1	1	**2**	**2**	**2**	**2**
ipc2014-opt-Maintenance (5)	**5**	**5**	**5**	**5**	**5**	**5**	**5**
ipc2014-opt-Openstacks (20)	**0**	**0**	**0**	**0**	**0**	**0**	**0**
ipc2014-opt-Parking (20)	**0**	**0**	**0**	**0**	**0**	**0**	**0**
ipc2014-opt-Tetris (17)	**2**	**2**	**2**	**2**	**2**	**2**	**2**
ipc2014-opt-Tidybot (20)	**0**	**0**	**0**	**0**	**0**	**0**	**0**
ipc2014-opt-Transport (20)	**0**	**0**	**0**	**0**	**0**	**0**	**0**
ipc2014-opt-Visitall (20)	**3**	**3**	**3**	**3**	**3**	**3**	**3**
logistics00 (28)	**8**	7	7	**8**	**8**	**8**	**8**
logistics98 (35)	2	**3**	**3**	**3**	**3**	**3**	**3**
miconic (150)	**138**	**138**	**138**	**138**	**138**	**138**	**138**
mprime (35)	17	**20**	**20**	**20**	**20**	**20**	**20**
mystery (30)	12	**16**	**16**	**16**	**16**	**16**	**16**
nomystery-opt11-strips (20)	**14**	12	**14**	**14**	**14**	**14**	**14**
openstacks-opt08-strips (30)	**4**	**4**	**4**	**4**	**4**	**4**	**4**
openstacks-opt11-strips (20)	**1**	**1**	**1**	**1**	**1**	**1**	**1**
openstacks-strips (30)	**5**	0	**5**	**5**	**5**	**5**	**5**
parcprinter-08-strips (30)	6	**13**	**13**	6	**13**	**13**	**13**
parcprinter-opt11-strips (20)	2	**8**	**8**	2	**8**	**8**	**8**
parking-opt11-strips (20)	**1**	0	**1**	**1**	**1**	**1**	**1**
pathways-noneg (30)	2	**4**	**4**	2	**4**	**4**	**4**
pegsol-08-strips (30)	21	**24**	**24**	**24**	**24**	22	**24**
pegsol-opt11-strips (20)	7	**14**	**14**	**14**	**14**	12	**14**
pipesworld-notankage (50)	**11**	10	**11**	**11**	**11**	**11**	**11**
pipesworld-tankage (50)	**7**	5	6	6	**7**	**7**	**7**
psr-small (50)	**29**	28	28	28	28	28	28
rovers (40)	**4**	**4**	**4**	**4**	**4**	**4**	**4**
satellite (36)	4	5	5	4	**6**	**6**	**6**
scanalyzer-08-strips (30)	4	**12**	**12**	4	**12**	**12**	**12**
scanalyzer-opt11-strips (20)	1	**9**	**9**	1	**9**	**9**	**9**
sokoban-opt08-strips (30)	**0**	**0**	**0**	**0**	**0**	**0**	**0**
sokoban-opt11-strips (20)	**0**	**0**	**0**	**0**	**0**	**0**	**0**
tidybot-opt11-strips (20)	3	3	3	3	**4**	**4**	3
tpp (30)	**5**	**5**	**5**	**5**	**5**	**5**	**5**
transport-opt08-strips (30)	1	**3**	**3**	1	**3**	**3**	**3**
transport-opt11-strips (20)	**0**	**0**	**0**	**0**	**0**	**0**	**0**
trucks-strips (30)	2	**3**	**3**	1	**3**	**3**	**3**
visitall-opt11-strips (20)	**9**	**9**	**9**	**9**	**9**	**9**	**9**
woodworking-opt08-strips (30)	7	**10**	**10**	7	**10**	**10**	**10**
woodworking-opt11-strips (20)	2	**5**	**5**	2	**5**	**5**	**5**
zenotravel (20)	7	**9**	**9**	**9**	**9**	**9**	**9**
Sum (1615)	440	467	497	461	505	501	**508**

Table A.20
Search time score for IDA^* algorithms in planning domains.

coverage	lm	lmcut	max	selmax	lazy	emp	T1
airport (50)	34.32	37.71	37.88	35.95	38.03	**38.18**	37.95
barman-opt11-strips (20)	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
blocks (35)	44.77	45.57	45.09	45.46	46.00	45.76	**46.10**
depot (22)	6.28	**9.09**	9.00	8.93	**9.09**	**9.09**	**9.09**
driverlog (20)	18.56	29.14	29.52	29.53	32.32	32.20	**32.39**
elevators-opt08-strips (30)	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
elevators-opt11-strips (20)	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
floortile-opt11-strips (20)	0.00	0.62	0.47	0.59	0.66	**0.69**	0.00
freecell (80)	**32.18**	1.65	27.77	31.90	30.14	30.12	32.14
grid (5)	**20.00**	18.52	19.22	**20.00**	**20.00**	**20.00**	**20.00**
gripper (20)	**9.77**	8.56	8.68	9.43	9.23	9.30	9.50
ipc2014-opt-Barman (14)	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
ipc2014-opt-CaveDiving (3)	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
ipc2014-opt-ChildSnack (20)	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
ipc2014-opt-Floortile (20)	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
ipc2014-opt-GED (20)	39.54	46.48	45.61	44.55	45.59	45.24	**47.05**
ipc2014-opt-Hiking (20)	3.00	2.09	1.97	2.60	3.76	**3.81**	**3.81**
ipc2014-opt-Maintenance (5)	81.99	**88.47**	87.05	80.70	88.21	88.19	88.37
ipc2014-opt-Openstacks (20)	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
ipc2014-opt-Parking (20)	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
ipc2014-opt-Tetris (17)	**9.09**	7.08	7.15	8.74	8.31	8.32	8.29
ipc2014-opt-Tidybot (20)	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
ipc2014-opt-Transport (20)	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
ipc2014-opt-Visitall (20)	6.51	6.21	6.64	5.72	7.30	**7.40**	7.37
logistics00 (28)	**21.79**	20.18	19.22	21.09	21.06	21.02	21.24
logistics98 (35)	3.40	**8.57**	8.48	8.54	**8.57**	**8.57**	**8.57**
miconic (150)	**90.12**	76.11	76.20	89.81	89.03	89.09	89.01
mprime (35)	41.21	46.74	48.71	49.03	**51.17**	51.09	51.16
mystery (30)	40.85	48.98	49.58	49.93	**50.41**	50.33	50.26
nomystery-opt11-strips (20)	**64.60**	46.77	54.34	64.20	61.92	62.37	62.72
openstacks-opt08-strips (30)	**13.33**	**13.33**	**13.33**	**13.33**	**13.33**	**13.33**	**13.33**
openstacks-opt11-strips (20)	**5.00**	**5.00**	**5.00**	**5.00**	**5.00**	**5.00**	**5.00**
openstacks-strips (30)	**14.20**	0.00	10.55	13.44	12.00	12.03	13.74
parcprinter-08-strips (30)	17.62	**43.33**	**43.33**	20.00	**43.33**	**43.33**	**43.33**
parcprinter-opt11-strips (20)	6.45	**40.00**	**40.00**	10.00	**40.00**	**40.00**	**40.00**
parking-opt11-strips (20)	0.09	0.00	0.19	1.18	1.22	1.24	**1.25**
pathways-noneg (30)	6.67	**13.33**	**13.33**	6.67	**13.33**	11.05	12.93
pegsol-08-strips (30)	44.94	**54.68**	52.19	53.48	52.96	49.65	52.86
pegsol-opt11-strips (20)	12.88	**25.68**	22.60	23.62	23.36	19.38	23.41
pipesworld-notankage (50)	**16.37**	12.04	14.57	15.72	16.03	16.03	16.16
pipesworld-tankage (50)	**10.37**	7.65	9.07	9.92	10.35	10.34	10.35
psr-small (50)	**50.04**	48.60	47.96	48.22	49.38	49.45	49.51
rovers (40)	**10.00**	**10.00**	**10.00**	**10.00**	**10.00**	**10.00**	**10.00**
satellite (36)	10.30	13.83	13.77	9.99	**13.88**	13.85	**13.88**
scanalyzer-08-strips (30)	11.13	33.43	33.36	11.04	33.64	33.62	33.62
scanalyzer-opt11-strips (20)	5.00	35.25	34.94	5.00	35.34	35.36	**35.42**
sokoban-opt08-strips (30)	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
sokoban-opt11-strips (20)	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
tidybot-opt11-strips (20)	**9.46**	8.25	8.29	9.37	8.75	8.74	8.81
tpp (30)	14.04	16.52	16.44	13.81	**16.53**	16.49	16.49
transport-opt08-strips (30)	3.33	**7.17**	7.12	3.33	7.14	7.11	7.16
transport-opt11-strips (20)	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
trucks-strips (30)	2.52	**7.85**	7.59	2.35	7.59	7.61	7.50
visitall-opt11-strips (20)	37.18	38.66	38.86	36.51	39.34	39.36	**39.43**
woodworking-opt08-strips (30)	15.78	31.28	**31.30**	17.42	31.11	31.12	31.12
woodworking-opt11-strips (20)	5.87	**21.97**	21.90	5.48	21.63	21.66	21.67
zenotravel (20)	31.56	37.81	37.39	37.47	39.02	38.97	**39.07**
Average (1615)	15.90	18.52	19.24	17.05	20.09	19.92	**20.19**

Please cite this article in press as: E. Karpas et al., Rational deployment of multiple heuristics in optimal state-space search, Artif. Intell. (2017), https://doi.org/10.1016/j.artint.2017.11.001

Table A.21
Number of expansions for IDA^* algorithms in planning domains.

coverage	lm	lmcut	max	selmax	lazy	emp	T1
airport (18)	163.76	**130.84**	**130.84**	163.76	**130.84**	143.92	131.19
blocks (18)	6,263.01	1,770.50	**1,769.53**	3,700.41	**1,769.53**	1,945.70	1,776.18
depot (2)	104,540.03	**932.42**	**932.42**	1,321.51	**932.42**	**932.42**	**932.42**
driverlog (6)	484,403.54	14,532.94	**10,804.04**	13,572.15	**10,804.04**	**10,804.04**	10,804.69
freecell (5)	**16.69**	529,646.82	**16.69**	**16.69**	**16.69**	**16.69**	**16.69**
grid (1)	6,560.00	1,678.00	**1,283.00**	6,555.00	**1,283.00**	1,295.00	1,297.00
gripper (2)	**13,109.12**	23,527.29	**13,109.12**	13,109.74	**13,109.12**	**13,109.12**	**13,109.12**
ipc2014-opt-GED (11)	58,997.60	**6,682.47**	**6,682.47**	8,806.94	**6,682.47**	8,675.63	**6,682.47**
ipc2014-opt-Hiking (1)	5,121,116.00	2,268,620.00	**2,094,746.00**	5,121,116.00	**2,094,746.00**	2,094,942.00	2,097,682.00
ipc2014-opt-Maintenance (5)	5,627.41	1,457.33	**1,231.02**	4,953.68	**1,231.02**	**1,231.02**	**1,231.02**
ipc2014-opt-Tetris (2)	26,190.46	17,510.38	**16,381.08**	26,190.46	**16,381.08**	**16,381.08**	**16,381.08**
ipc2014-opt-Visitall (3)	5,412,921.66	1,619,635.88	**923,706.97**	5,412,871.76	**923,706.97**	1,007,303.58	1,023,168.51
logistics00 (7)	37,055.43	**36,900.09**	**36,900.09**	37,055.43	**36,900.09**	**36,900.09**	**36,900.09**
logistics98 (2)	1,796,135.34	**19,676.79**	**19,676.79**	36,650.71	**19,676.79**	**19,676.79**	19,680.92
miconic (138)	**1,807.37**	**1,807.37**	**1,807.37**	**1,807.37**	**1,807.37**	**1,807.37**	**1,807.37**
mprime (17)	6,527.81	864.75	**365.42**	1,200.74	**365.42**	**365.42**	**365.42**
mystery (14)	420.74	41.08	**32.29**	46.89	**32.29**	**32.29**	**32.29**
nomystery-opt11-strips (12)	6,630.27	3,830.61	**2,458.62**	6,457.16	**2,458.62**	2,466.55	2,541.87
openstacks-opt08-strips (4)	235.34	**208.85**	**208.85**	235.34	**208.85**	**208.85**	**208.85**
openstacks-opt11-strips (1)	73.00	**61.00**	**61.00**	73.00	**61.00**	**61.00**	**61.00**
parcprinter-08-strips (6)	2,247.40	**23.18**	**23.18**	112.77	**23.18**	24.35	**23.18**
parcprinter-opt11-strips (2)	772,685.36	**22.49**	**22.49**	646.49	**22.49**	23.00	**22.49**
pathways-noneg (2)	1,221.91	**32.76**	**32.76**	890.23	**32.76**	**32.76**	**32.76**
pegsol-08-strips (21)	146,957.15	20,746.26	**20,555.69**	24,658.29	**20,555.69**	41,502.91	21,802.72
pegsol-opt11-strips (7)	3,224,793.17	325,747.05	**319,776.43**	328,031.15	**319,776.43**	702,021.95	327,586.91
pipesworld-notankage (10)	33,865.14	48,148.35	**13,922.18**	29,835.16	**13,922.18**	13,922.82	14,512.34
pipesworld-tankage (5)	8,492.64	14,063.65	**4,272.29**	6,999.00	**4,272.29**	4,273.40	4,325.75
psr-small (28)	23,054.69	**19,329.28**	**19,329.28**	22,490.74	19,386.26	19,767.85	19,465.97
rovers (4)	234.60	268.86	**127.84**	233.66	**127.84**	**127.84**	**127.84**
satellite (4)	4,427.88	233.24	**215.18**	4,411.43	**215.18**	**215.18**	**215.18**
scanalyzer-08-strips (4)	2,635.85	**15.24**	**15.24**	742.88	**15.24**	**15.24**	**15.24**
scanalyzer-opt11-strips (1)	184.00	**10.00**	**10.00**	34.00	**10.00**	**10.00**	**10.00**
tidybot-opt11-strips (3)	12,348.98	**3,020.36**	**3,020.36**	12,345.51	**3,020.36**	3,048.60	3,085.71
tpp (5)	327.29	**86.16**	**86.16**	327.29	**86.16**	**86.16**	**86.16**
transport-opt08-strips (1)	**10.00**	16.00	**10.00**	**10.00**	**10.00**	**10.00**	**10.00**
trucks-strips (1)	740,948.00	3,543.00	**3,431.00**	740,862.00	**3,431.00**	**3,431.00**	3,521.00
visitall-opt11-strips (9)	2,089.45	1,093.24	**686.96**	2,053.41	**686.96**	707.09	719.81
woodworking-opt08-strips (7)	218,483.05	**106.19**	**106.19**	66,241.56	108.56	108.56	109.89
woodworking-opt11-strips (2)	773,198.06	**164.27**	**164.27**	772,864.88	**164.27**	**164.27**	**164.27**
zenotravel (7)	4,527.89	**202.32**	**202.32**	483.75	**202.32**	203.13	**202.32**
Geometric mean (398)	10,614.63	1,574.53	**999.24**	3,951.16	999.86	1,055.68	1,009.96

Please cite this article in press as: E. Karpas et al., Rational deployment of multiple heuristics in optimal state-space search, Artif. Intell. (2017), https://doi.org/10.1016/j.artint.2017.11.001

Table A.22
Fraction of nodes in which h_2 was evaluated for IDA^* algorithms in planning domains.

h_2 **ratio**	**lazy**	**emp**	**T1**
airport (21)	**0.91**	0.44	0.88
blocks (19)	**0.63**	0.54	0.55
depot (2)	**0.63**	**0.63**	0.62
driverlog (8)	**0.36**	**0.36**	0.35
freecell (28)	**0.27**	0.26	0.24
grid (1)	**0.58**	0.56	0.53
gripper (2)	**0.25**	**0.25**	0.13
ipc2014-opt-GED (13)	**1.00**	0.59	0.76
ipc2014-opt-Hiking (2)	**0.10**	**0.10**	0.09
ipc2014-opt-Maintenance (5)	**0.20**	**0.20**	**0.20**
ipc2014-opt-Tetris (2)	**0.24**	**0.24**	0.22
ipc2014-opt-Visitall (3)	**0.59**	0.54	0.51
logistics00 (8)	**0.15**	**0.15**	0.14
logistics98 (3)	**0.34**	**0.34**	**0.34**
miconic (138)	**0.12**	0.11	0.11
mprime (20)	**0.56**	**0.56**	0.55
mystery (18)	**0.71**	**0.71**	**0.71**
nomystery-opt11-strips (14)	**0.24**	0.23	0.20
openstacks-opt08-strips (4)	**1.00**	0.86	0.99
openstacks-opt11-strips (1)	**1.00**	0.66	0.98
openstacks-strips (5)	**0.46**	0.42	0.04
parcprinter-08-strips (13)	**0.85**	0.72	0.81
parcprinter-opt11-strips (8)	**0.85**	0.78	0.81
parking-opt11-strips (1)	**0.19**	**0.19**	**0.19**
pathways-noneg (4)	**0.84**	0.59	0.45
pegsol-08-strips (22)	**0.87**	0.43	0.78
pegsol-opt11-strips (12)	**0.85**	0.44	0.76
pipesworld-notankage (11)	**0.29**	**0.29**	0.26
pipesworld-tankage (7)	**0.16**	**0.16**	0.15
psr-small (28)	**0.37**	0.26	0.24
rovers (4)	**0.22**	**0.22**	0.18
satellite (6)	**0.39**	**0.39**	**0.39**
scanalyzer-08-strips (12)	**0.85**	**0.85**	0.80
scanalyzer-opt11-strips (9)	**0.84**	**0.84**	0.82
tidybot-opt11-strips (3)	**0.57**	0.56	0.55
tpp (5)	**0.63**	0.49	0.58
transport-opt08-strips (3)	**0.78**	**0.78**	0.69
trucks-strips (3)	**0.83**	**0.83**	0.81
visitall-opt11-strips (9)	**0.65**	0.63	0.57
woodworking-opt08-strips (10)	**0.42**	**0.42**	0.41
woodworking-opt11-strips (5)	**0.39**	**0.39**	0.38
zenotravel (9)	**0.36**	0.35	0.35
Average (501)	**0.54**	0.46	0.48

References

[1] S. Russell, E. Wefald, Principles of metareasoning, Artif. Intell. 49 (1991) 361–395.
[2] D. Tolpin, S.E. Shimony, Rational deployment of CSP heuristics, in: T. Walsh (Ed.), IJCAI, IJCAI/AAAI, 2011, pp. 680–686.
[3] N. Hay, S. Russell, D. Tolpin, S.E. Shimony, Selecting computations: theory and applications, in: N. de Freitas, K.P. Murphy (Eds.), UAI, AUAI Press, 2012, pp. 346–355.
[4] J.T. Thayer, W. 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.
[5] E. Burns, W. Ruml, M.B. Do, Heuristic search when time matters, J. Artif. Intell. Res. (JAIR) 47 (2013) 697–740.
[6] D. O'Ceallaigh, W. Ruml, Metareasoning in real-time heuristic search, in: Lelis and Stern [39], pp. 87–95.
[7] P.E. Hart, N.J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Trans. Syst. Sci. Cybern. SCC 4 (2) (1968) 100–107.
[8] R.E. Korf, Depth-first iterative-deepening: an optimal admissible tree search, Artif. Intell. 27 (1) (1985) 97–109.
[9] R.E. Korf, Linear-space best-first search, Artif. Intell. 62 (1) (1993) 41–78.
[10] A. Felner, U. Zahavi, R. Holte, J. Schaeffer, N. Sturtevant, Z. Zhang, Inconsistent heuristics in theory and practice, Artif. Intell. 175 (9–10) (2011) 1570–1603.
[11] R. Dechter, J. Pearl, Generalized best-first search strategies and the optimality of A*, J. ACM 32 (3) (1985) 505–536.
[12] L. Zhang, F. Bacchus, MAXSAT heuristics for cost optimal planning, in: AAAI, 2012.
[13] C. Domshlak, E. Karpas, S. Markovitch, Online speedup learning for optimal planning, J. Artif. Intell. Res. 44 (2012) 709–755.
[14] M.W. Barley, S. Franco, P.J. Riddle, Overcoming the utility problem in heuristic generation: why time matters, in: S.A. Chien, M.B. Do, A. Fern, W. Ruml (Eds.), Proc. ICAPS 2014, AAAI, 2014.
[15] L.H.S. Lelis, S. Franco, M. Abisrror, M. Barley, S. Zilles, R.C. Holte, Heuristic subset selection in classical planning, in: S. Kambhampati (Ed.), Proc. IJCAI 2016, IJCAI/AAAI Press, 2016, pp. 3185–3191.
[16] D. Tolpin, T. Beja, S.E. Shimony, A. Felner, E. Karpas, Toward rational deployment of multiple heuristics in A*, in: IJCAI, 2013.

Please cite this article in press as: E. Karpas et al., Rational deployment of multiple heuristics in optimal state-space search, Artif. Intell. (2017), https://doi.org/10.1016/j.artint.2017.11.001

[17] D. Tolpin, O. Betzalel, A. Felner, S.E. Shimony, Rational deployment of multiple heuristics in IDA*, in: T. Schaub, G. Friedrich, B. O'Sullivan (Eds.), ECAI 2014, in: Frontiers in Artificial Intelligence and Applications, vol. 263, IOS Press, 2014, pp. 1107–1108.
[18] A. Felner, M. Goldenberg, G. Sharon, R. Stern, T. Beja, N.R. Sturtevant, J. Schaeffer, R. Holte, Partial-expansion A* with selective node generation, in: AAAI, 2012, pp. 471–477.
[19] M. Goldenberg, A. Felner, R. Stern, G. Sharon, N.R. Sturtevant, R.C. Holte, J. Schaeffer, Enhanced partial expansion A, J. Artif. Intell. Res. (JAIR) 50 (2014) 141–187.
[20] R. Stern, T. Kulberis, A. Felner, R. Holte, Using lookaheads with optimal best-first search, in: AAAI, 2010, pp. 185–190.
[21] X. Sun, W. Yeoh, P. Chen, S. Koenig, Simple optimization techniques for A*-based search, in: AAMAS, 2009, pp. 931–936.
[22] D. Tolpin, O. Betzalel, A. Felner, S.E. Shimony, Rational deployment of multiple heuristics in IDA*, CoRR arXiv:1411.6593, http://arxiv.org/abs/1411.6593.
[23] L.H.S. Lelis, S. Zilles, R.C. Holte, Predicting the size of IDA*'s search tree, Artif. Intell. 196 (2013) 53–76.
[24] M. Helmert, The fast downward planning system, J. Artif. Intell. Res. 26 (2006) 191–246.
[25] E. Karpas, C. Domshlak, Cost-optimal planning with landmarks, in: IJCAI, 2009, pp. 1728–1733.
[26] M. Helmert, C. Domshlak, Landmarks, critical paths and abstractions: what's the difference anyway?, in: ICAPS, 2009, pp. 162–169.
[27] J. Seipp, F. Pommerening, S. Sievers, M. Helmert, Downward-lab 2.0, 2017, https://doi.org/10.5281/zenodo.399255.
[28] R.E. Korf, L.A. Taylor, Finding optimal solutions to the twenty-four puzzle, in: AAAI, 1996, pp. 1202–1207.
[29] O. Betzalel, A. Felner, S.E. Shimony, Type system based rational lazy IDA, in: Lelis and Stern [39], pp. 151–155, http://www.aaai.org/ocs/index.php/SOCS/SOCS15/paper/view/11015.
[30] H. Zhang, S. Guo, W. Zhu, A. Lim, B. Cheang, An investigation of IDA* algorithms for the container relocation problem, in: Proc. of the 23rd Inter. Conf. on Industrial Engineering and Other Applications of Applied Int. Systems – Part I, IEA/AIE'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 31–40.
[31] M. Caserta, S. Voß, M. Sniedovich, Applying the corridor method to a blocks relocation problem, OR Spektrum 33 (4) (2011) 915–929.
[32] B. Jin, A. Lim, W. Zhu, A greedy look-ahead heuristic for the container relocation problem, in: IEA/AIE, in: LNCS, vol. 7906, Springer, 2013, pp. 181–190.
[33] D. Berend, S. Cohen, S.E. Shimony, S. Zucker, Optimal ordering of tests with extreme dependencies, in: Modelling, Computation and Optimization in Information Systems and Management Sciences – Proceedings of the 3rd International Conference on Modelling, Computation and Optimization in Information Systems and Management Sciences, MCO 2015, Metz, France, May 11–13, 2015, Part I, 2015, pp. 81–92.
[34] D. Berend, R. Brafman, S. Cohen, S. Shimony, S. Zucker, Optimal ordering of independent tests with precedence constraints, Discrete Appl. Math. 162 (2014) 115–127.
[35] R.E. Korf, A. Felner, Recent progress in heuristic search: a case study of the four-peg towers of Hanoi problem, in: Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI 2007, Hyderabad, India, January 6–12, 2007, 2007, pp. 2324–2329.
[36] L.H.S. Lelis, R. Stern, A. Felner, S. Zilles, R.C. Holte, Predicting optimal solution cost with conditional probabilities – predicting optimal solution cost, Ann. Math. Artif. Intell. 72 (3–4) (2014) 267–295.
[37] L.H.S. Lelis, R. Stern, S.J. Arfaee, S. Zilles, A. Felner, R.C. Holte, Predicting optimal solution costs with bidirectional stratified sampling in regular search spaces, Artif. Intell. 230 (2016) 51–73.
[38] P. Joulani, A. György, C. Szepesvári, Online learning under delayed feedback, in: ICML, 2013, pp. 1453–1461.
[39] L. Lelis, R. Stern (Eds.), Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015, 11–13 June 2015, Ein Gedi, the Dead Sea, Israel, AAAI Press, 2015.