On Learning In Agent-Centered Search

Nathan R. Sturtevant
Dept. of Computing Science
University of Alberta
Edmonton, Alberta, Canada
nathanst@cs.ualberta.ca

Vadim Bulitko
Dept. of Computing Science
University of Alberta
Edmonton, Alberta, Canada
bulitko@ualberta.ca

Yngvi Björnsson
School of Computer Science
Reykjavik University
Reykjavik, Iceland
yngvi@ru.is

ABSTRACT

Since the introduction of the LRTA* algorithm, real-time heuristic search algorithms have generally followed the same plan-act-learn cycle: an agent plans one or several actions based on locally available information, executes them and then updates (i.e., learns) its heuristic function. Algorithm evaluation has almost exclusively been empirical with the results often being domain-specific and incomparable across papers. Even when unification and cross-algorithm comparisons have been carried out in a single paper, there was no understanding of how efficient the learning process was with respect to a theoretical optimum. This paper addresses the problem with two primary contributions. First, we formally define a lower bound on the amount of learning any heuristic-learning algorithm needs to do. This bound is based on the notion of heuristic depressions and allows us to have a domain-independent measure of learning efficiency across different algorithms. Second, using this measure we propose to learn "costs-so-far" (g-costs) instead of "costs-to-go" (h-costs). This allows us to quickly identify redundant paths and dead-end states, thereby leading to asymptotic performance improvement as well as 1-2 orders of magnitude convergence speed-ups in practice.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms

Keywords

Search, Planning, Real-Time, Learning

1. INTRODUCTION

In this paper we study the problem of *agent-centered real-time heuristic search* [11]. The distinctive property of such search is that an agent must repeatedly plan and execute actions within a constant time interval that is independent of the size of the problem being solved. Furthermore, the planning must be restricted to the part of the domain around the current state of the agent. This restriction severely limits the range of applicable heuristic search algorithms. For instance, static search algorithms such as A* [6] and IDA* [16], re-planning algorithms such as D* [27], and anytime re-planning algorithms such as AD* [19] cannot guarantee a constant bound on planning time per action. TBA* [2], a real-time variant of A*, is also inapplicable as it does not restrict its planning to the vicinity of the agent.

As a motivating example, consider pathfinding in video games. A standard approach is to view a game map as a rectangular grid with some cells blocked due to obstacles. In such games, an agent can be tasked to go to any location on the map from its current location. Examples include real-time strategy games such as Warcraft 3, first-person shooters such as Doom, and role-playing games such as Baldur's Gate. Size and complexity of game maps as well as the number of simultaneously moving units on such maps continues to increase with every new generation of games. Nevertheless, each game unit or agent must react quickly to the user's command regardless of the map's size and complexity. Consequently, game companies impose a time-per-action limit on their pathfinding algorithms, often on the order of 1-3 ms per frame.

Real-time search algorithms compute (or plan) only the first few actions for the agent to take. This is usually done by conducting a lookahead search of a fixed depth (also known as "search horizon", "search depth" or "lookahead depth") around the agent's current state and using a heuristic (i.e., an estimate of the remaining travel cost) to select the next few actions. The actions are then taken and the planning-execution cycle repeats. Since the goal state is not reached by most such local searches, the agent runs the risk of heading into a dead end or, more generally, selecting suboptimal actions. To address this problem, real-time heuristic search algorithms update (or learn) their heuristic function with experience, however the learning process is not always efficient, and can produce behavior that seems irrational.

Since the introduction of the seminal real-time heuristic search algorithm (LRTA*) around two decades ago [17], most research in the area has focused on making the learning process more efficient. Most of the results share the same drawback – they are mostly empirical and domain-specific in nature. The lack of a domain-independent measure of learning efficiency has impeded a clear understanding of learning difficulty inherent in a problem as well as gauging learning efficiency delivered by the algorithms.

In this paper we make two primary contributions. First, we introduce a simple domain-independent measure of learning difficulty, which can be computed for any given state space and heuristic function. Furthermore, the concept behind the learning-difficulty measure provides an insight into learning problems that heuristic-learning agent-centered algorithms face. For a class of real-time search agents, we can show that increasing lookahead does not asymptotically improve performance. With this insight, we make

Cite as: On Learning In Agent-Centered Search, Nathan R. Sturtevant, Vadim Bulitko and Yngvi Björnsson, *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, van der Hoek, Kaminka, Lespérance, Luck and Sen (eds.), May, 10–14, 2010, Toronto, Canada, pp. XXX-XXX.

the second contribution by proposing a new learning mechanism that updates costs-so-far (g-costs) instead of costs-to-go (h-costs). This allows us to efficiently identify and prune out states extraneous to the problem at hand. We demonstrate that the new learning mechanism is asymptotically faster on an important class of learning problems. Finally, to demonstrate the efficiency of the new learning mechanism empirically we implement a real-time version of iterative deepening best-first search and show 1-2 orders of magnitude improvements across several domains.

The rest of the paper is organized as follows. We formalize the problem in Section 2, followed by a review of the related work in Section 3. The first contribution of this paper is presented in Section 4 followed by the new learning paradigm in Section 5. A practical implementation of the new learning paradigm comes in the form of a new algorithm in Section 6, complete with an extensive empirical evaluation in Section 7. Future work directions and conclusions close the paper in Section 8.

2. PROBLEM FORMULATION

We define a heuristic search problem as an undirected graph containing a finite set of states S and weighted edges E, with a state s_{start} designated as the *start state* and a state s_{goal} designated as the *goal state*. At every time step, a search agent has a single *current state*, vertex in the search graph, and takes an action by traversing an out-edge of the current state. Each edge has a positive cost $c(s, s/)$ associated with it. The total cost of edges traversed by an agent from its start state until it arrives at the goal state is called the *solution cost*.

We require algorithms to be *complete* and produce a path from start to goal in a finite amount of time if such a path exists. In order to guarantee completeness for real-time heuristic search we make the assumption of safe explorability of our search problems. Namely, all costs are finite and the goal state is reachable from any state that the agent can possibly reach from its start state.

The analysis in the paper applies to the *ant paradigm* [23]. Namely, our algorithms have an agent-centered view of the environment, having access to the search graph in a certain radius around their current state. In other words, they cannot access arbitrary vertices and edges of the search graph. Furthermore, any additional information they learn (e.g., a heuristic function) is subject to the same local access. This view is motivated by various applications ranging from routing in *ad hoc* wireless networks [18] to map streaming in video games [28].

Additionally, as motivated in the introduction, we consider *real-time* algorithms only. Specifically, a search algorithm is considered real-time if and only if the amount of planning it performs prior to traversing an edge in the search graph is upper-bounded by a constant independent of the total number of vertices (but possibly dependent on the vertex degree). As in most real-time heuristic search work, we adopt the plan-execute cycle where the agent does not plan while moving and does not move while planning.

Formally, all algorithms discussed in this paper are applicable to any such heuristic search problem. To keep the presentation focused and intuitive as well as to afford a large-scale empirical evaluation, we will use two popular domains: grid-based pathfinding and sliding-tile puzzles in the rest of the paper. Nevertheless, the contributions we make are, in principle, applicable to general planning and especially to real-time planning.

3. RELATED WORK

Most work in real-time heuristic search focused on *heuristic-learning* algorithms. The seminal example of such learning is the Learning Real-time A* (LRTA*) algorithm [17] shown in Figure 1.

```
LRTA*($s_{\text{start}}, s_{\text{goal}}, g_{\max}$)
1 $s \leftarrow s_{\text{start}}$
2 while $s \neq s_{\text{goal}}$ do
3     expand successor states up to cost $g_{\max}$ away
4     find a frontier state $s'$ with the lowest $g(s') + h(s')$
5     update $h(s)$ to $\max(h(s), g(s') + h(s'))$
6     change $s$ one step towards $s'$
7 end while
```

Figure 1: The LRTA* algorithm.

We will use the pseudo-code to illustrate the key points of this type of search. As long as the goal state is not reached (line 2), the agent follows the plan (lines 3-4), learn (line 5), execute (line 6) cycle. The planning consists of a fixed depth lookahead during which all unique states up to a certain cost cut-off ($g_{\max}$) are expanded. During the learning part of the cycle, the agent updates a numeric value $h(s)$ for its current state s. The heuristic function h is an estimate of the minimum total edge cost from a given state s to the goal state s_{goal}, denoted by $h^*(s)$. A heuristic is admissible if for any state s it does not exceed $h^*(s)$. A heuristic is consistent if for any two adjacent states s_1 and s_2 the difference in their heuristic values does not exceed the edge weight: $|h(s_1) - h(s_2)| \leq c(s_1, s_2)$. Finally, the agent moves by changing its current state towards the most promising state discovered in the planning stage.

Since LRTA*, research in the field of learning real-time heuristic search has resulted in several dozen algorithms with numerous variations. Most of them can be described by the following four attributes. The *local search space* is the set of states whose heuristic values are accessed in the planning stage. The two common choices are full-width limited-depth lookahead [17, 24, 26, 25, 5, 7, 21] and A*-shaped lookahead [12, 13, 15]. The *local learning space* is the set of states whose heuristic values are updated. Common choices are: the current state only [17, 24, 26, 25, 5, 3], all states within the local search space [12, 13] and previously visited states and their neighbors [7, 21]. A *learning rule* is used to update the heuristic values of the states in the learning space. The common choices are dynamic programming or mini-min [17, 26, 25, 7, 21], their weighted versions [24], max of mins [3], modified Dijkstra's algorithm [12], and updates with respect to the shortest path from the current state to the best-looking state on the frontier of the local search space [13]. Additionally, several algorithms learn more than one heuristic function [22, 5, 24]. The *control strategy* decides on the actions taken following the planning and learning phases. Commonly used strategies include: the first move of an optimal path to the most promising frontier state [17, 5, 7], the entire path [3], and backtracking moves [26, 25, 3].

Unfortunately, the evaluation of the numerous algorithms has been primarily empirical and, worse yet, with the results computed over incompatible sets of problems. While some researchers (e.g., [4]) compared a number of algorithms on the same problems, it remained unclear how well they perform with respect to a theoretical optimum. Indeed, the theoretical results on convergence have been limited to worst-case bounds that do not take into account the structure of a particular problem [9, 14, 10, 4].

As far as we know, the connection between convergence cost of LRTA* and the total amount of error in heuristic search was first made in [4]. The connection was purely empirical and no lower bound on the amount of learning was derived. A more recent analysis using total heuristic error applies to random maps only, and does not explain convergence difficulty via problem structure in a quantitative way [20]. Also, it provides no way to make quantitative predictions of the amount of learning for a given problem.

The first contribution of this paper is the introduction of a domain-independent measure of learning difficulty that takes problem structure into account. This allows us to formulate a theoretically minimal amount of learning that any heuristic search algorithm from a certain broad class must perform. This bound serves as a yardstick for convergence performance of the algorithms.

4. DIFFICULTY OF LEARNING

The first contribution of this paper is a domain-independent analysis of learning difficulty under the following assumptions. First, we assume that the heuristic being learnt by the algorithm is consistent at all times, which also implies admissibility when the h-cost of the goal is 0. Second, we assume that all edge weights are constant bounded. For simplicity we assume uniform edge costs of 1 in this analysis. Third, we assume that all algorithms use a Bellman update rule [1] (called *mini-min* in [17]), updating their heuristic in the current state as:

$$h(s) = \min_{s' \in \mathrm{Succ}(s)} \big(c(s,s') + h(s')\big) = \min_{s' \in \mathrm{Succ}(s)} \big(1 + h(s')\big)$$

where $\mathrm{Succ}(s)$ is the set of all states reachable from s in a single move[1] and $c(a,b)$ is the optimal cost between a and b. Fourth, we assume that the state s_{next} the agent travels to from its current state s is selected as (arg breaks ties randomly):

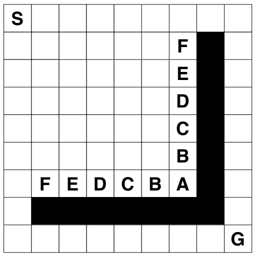
$$s_{\text{next}} = \arg\min_{s' \in \mathrm{Succ}(s)} \big(c(s,s') + h(s')\big) = \arg\min_{s' \in \mathrm{Succ}(s)} h(s')$$

Under these four assumptions we say that the learning process *converges* on a particular problem if, after repeatedly solving the same problem n times, the $(n+1)$-th solution cost incurred by the agent is guaranteed to be optimal regardless of the tie breaking. The *total amount of learning* is the cumulative magnitude of adjustments an algorithm made to its heuristic h during these n runs (also called *trials*). The cumulative cost of all edges traversed by the algorithm on the $n+1$ trials is called the *convergence cost*. These definitions, with the exception of tie-breaking, are in line with previously published work (e.g., [4]).

Given these assumptions, we can describe the total amount of learning that any agent must do to converge. First, an agent must learn a perfect heuristic on all optimal paths between the start and the goal. Second, for every node n_b which borders a node n_o on an optimal path, n_b must have a high enough value to prevent a greedy agent from leaving the optimal path. More precisely, $c(n_o, n_b) + h(n_b) > h(n_o)$. Finally, the heuristic in the rest of the state space must be consistent with the values bordering the optimal path.

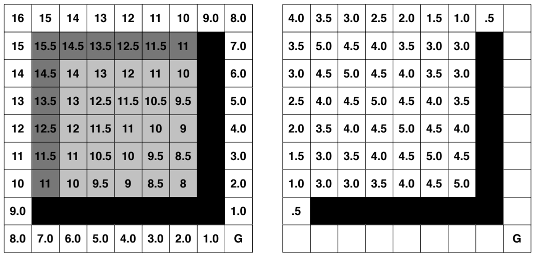
These three conditions are both necessary and sufficient. The first condition ensures, as ties are broken randomly, that all optimal paths could be followed. If the heuristic of a state on the optimal path is too high, it will be inadmissible and inconsistent, and if it is too low, an agent will necessarily revisit the state as other heuristic values exceed it. The second and third conditions together ensure that all nodes outside the optimal path will always have larger f-costs than the optimal nodes, and thus a suboptimal path will never be followed no matter the lookahead.

Given this definition there is a minimum amount of learning, L, that a learning algorithm must perform to converge on any particular problem. We use a simple argument to show that increasing the lookahead radius will not asymptotically reduce the time required to perform the minimum learning. Assume that a learning algorithm can look ahead k states prior to each action. Due to the consistency property, the heuristic can change at most by k in the space of the lookahead. This means that the heuristic value of the current state can be increased by at most $2k$. In order to maintain consistency, subsequent states in the lookahead will also have their heuristic values updated, but by progressively smaller values, with the final state not having its heuristic updated. As a result, the total changes made to the heuristic are bounded by $k(k+1)$ per underlying action of the agent. But, k must be constant relative to the size of the map. Thus learning must take at least $\lceil L/(k^2+k) \rceil$ steps, which is still $O(L)$.



S								
						F	■	
						E	■	
						D	■	
						C	■	
						B	■	
	F	E	D	C	B	A	■	
	■	■	■	■	■	■	■	
								G

Figure 2: 9×9 **sample learning map.**



16	15	14	13	12	11	10	9.0	8.0
15	15.5	14.5	13.5	12.5	11.5	11	■	7.0
14	14.5	14	13	12	11	10	■	6.0
13	13.5	13	12.5	11.5	10.5	9.5	■	5.0
12	12.5	12	11.5	11	10	9	■	4.0
11	11.5	11	10.5	10	9.5	8.5	■	3.0
10	11	10	9.5	9	8.5	8	■	2.0
9.0	■	■	■	■	■	■	■	1.0
8.0	7.0	6.0	5.0	4.0	3.0	2.0	1.0	G

(a) Required heuristic

4.0	3.5	3.0	2.5	2.0	1.5	1.0	.5	
3.5	5.0	4.5	4.0	3.5	3.0	3.0	■	
3.0	4.5	5.0	4.5	4.0	3.5	3.0	■	
2.5	4.0	4.5	5.0	4.5	4.0	3.5	■	
2.0	3.5	4.0	4.5	5.0	4.5	4.0	■	
1.5	3.0	3.5	4.0	4.5	5.0	4.5	■	
1.0	3.0	3.0	3.5	4.0	4.5	5.0	■	
.5	■	■	■	■	■	■	■	
								G

(b) Required learning

Figure 3: Measuring minimum learning required.

4.1 Example and Approximation

We approximate the metric, L, to measure learning on any given problem as follows. First, we place the start and goal on an *optimal path* queue. We iteratively take nodes off of this queue, placing them on a closed list and updating their h-cost to the perfect h-cost. Perfect h-costs are computed using a reverse A* search. Neighbors on the closed list are ignored and neighbors on optimal paths are put onto the optimal path queue; all other nodes are placed on a *neighbor* queue. When the optimal path queue is empty, all optimal paths have been found. Nodes in the neighbor queue are updated to their optimal h-cost, and then a pathmax step propagates the updated heuristic through the rest of the search space until the learned heuristic is consistent everywhere. The sum of the heuristic updates over all nodes is our approximation of L.

We demonstrate this in Figure 2. This is an 8-connected grid with diagonal edges having cost 1.5 and cardinal edges having cost 1. The initial heuristic is the cost of optimal travel between two points, assuming no walls. The goal is to travel between S and G. An agent greedily following the heuristic starting at S will travel first to A and then have to learn its way out of the local minimum. Our approximation computes the heuristic as follows.

First, the optimal costs between the goal and the start are computed, shown in Figure 3(a). There are two optimal paths corresponding with the outer edge of the map. The optimal heuristic values are shown in the figure. The start state must have a heuristic value of 16, but the initial heuristic value is only $8 \cdot 1.5 = 12$, or off by 4 as shown in Figure 3(b). Once all heuristic values on the optimal paths are known, the dark shaded cells are on the neighbor queue with their optimal cost to the goal. The heuristic values from

[1] With larger lookahead this rule is applied recursively.

these nodes are then propagated to the remaining nodes in the state space by ensuring that the values are consistent everywhere. The corner node next to the start state has a heuristic of 15.5. Because the cost to move diagonally towards the goal in the lower-right corner is 1.5, the heuristic of the neighbor must be at least 14 to ensure consistency. Note, however, that the heuristic of this corner state does not have to be 15.5. A value of $14.5 + \epsilon$ is sufficient to prevent an agent from leaving the optimal path and visiting this node. Thus, our approximation slightly overestimates the required learning, although the error per state is constant bounded.

5. ANALYSIS OF LEARNING

Learning h-values can be slow because inaccurate heuristics values are used to update other also inaccurate heuristic values. This is particularly problematic if a learning agent enters a heuristic depression [8], a localized area in the search space where it has to repeatedly revisit states to raise their heuristic values enough to be able to continue to explore other parts of the search space. The more frequent and deeper the depressions are, the more severely the problem manifests itself.

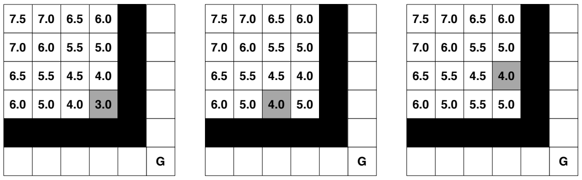
This is illustrated in Figure 4 with an example of LRTA* behavior on a portion of the same map we have previously been studying. States are marked with their initial heuristic values. Consider part (a) where the agent is in the shaded cell. Using a lookahead of one, the value of the corner heuristic can be updated from 3 to 5, because a neighbor distance 1 has heuristic cost 4. In part (b) the agent moves to the highlighted cell and makes a similar update, raising the h-cost to 5.5, before moving to the cell updated in part (c), where that cell will be updated to have a heuristic value of 5.5.

After three updates, considerable learning still remains. This is because the heuristic is being updated locally from neighboring heuristics, which, due to consistency, cannot be considerably larger. Thus, a state must be visited and updated many times before large changes in the heuristic can occur. As this learning begins far from the goal state, heuristic estimates are likely to be inaccurate.

Heuristic values are inaccurate near the start state, but g-costs near the start state are quite accurate. So, an alternate approach would be to learn g-costs, which can be 'learned' more accurately. At a first glance it is not clear how this is going to help the agent in finding the goal more efficiently, but a key observation is that the g-values can be helpful in escaping heuristic depressions. Not only can the they greatly reduce the number of times the agent must revisit states in a heuristic depression, but they are also useful in identifying dead states and redundant paths. Such identifications require accurate values and cannot be done with h-based learning.

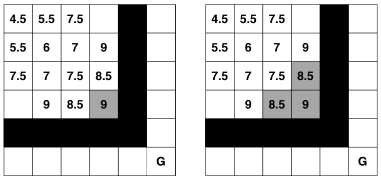
Excluding the start and goal, any node on an optimal path must have neighbor with both higher and lower g-costs. If a state has no neighbors with larger g-costs, then an optimal path to the goal cannot pass through this state. We thus define a *dead cell* as follows: Given a start state s and a node n, n is a dead cell if n is not the goal state and if for all non-dead neighbors of n, $n_1 \dots n_i$, $c(s,n) \geq c(n_i, s)$.

Consider the example in Figure 5, which shows g-cost estimates for the same problem. Upon reaching the corner, the agent can potentially mark each cell with the g-costs in the figure, which are upper-bounds on the actual cost to each node. In part (a) of the figure, the agent can see that the highlighted node in the corner is dead, because all neighbors can be reached by shorter paths through other nodes. After this node is marked dead, in part (b), two more cells can be marked dead. Learning that a cell is dead only requires visiting a state a single time, unlike learning a heuristic, which may take multiple visits. But, there is more that can be done if we know the optimal cost to each state.



(a) Step 1 (b) Step 2 (c) Step 3

Figure 4: Learning in a local minima.



(a) Step 1 (b) Step 2

Figure 5: Learning g-costs.

Consider Figure 6. In this case the cells in the corners can be marked as dead and removed once the optimal cost to these cells is discovered. Note, however, that even after removing the dead cells there are still many paths that lead out through this room. But, because there is only a single doorway to the room, these paths are all redundant. Detecting and ignoring nodes on such redundant paths offers additional saving. This requires two steps as illustrated in Figure 7. In the left portion of the figure, we show two possible optimal paths leading out of the room. We focus on nodes A and B, shown in detail in the center of the figure.

The first step is to mark all parents which are along optimal paths to a node. Each time a node is generated, if the parent is on an optimal path to the node, the parent is added to the list of optimal parents for that node. In the case of node B, nodes A and D can both reach B with the same cost, and so B maintains this information.

The second step occurs in the next iteration of search. Suppose that A is visited first. Then, at A we will notice that there is also an optimal path to B through D. Since there are no other optimal paths through A to a successor of A, A can be marked dead or redundant, and B is marked to have a single optimal parent of D. If D were visited first, D would be marked redundant, and only the path through A would be maintained.

The order in which parents are removed will influence the effectiveness of this approach. In each particular search problem there is an ordering which will effectively mark all redundant paths, but this ordering may not be known *a priori*. This can result in the removal of locally redundant paths, but prevent more global structures from being removed. Alternate ordering schemes we tried either shifted the problem elsewhere in the state space or had too much overhead to be effective.

In the right portion of Figure 7 we show an example of redundant cell removal from a real map. The light nodes are those which have been expanded and are still candidates for exploration. The darker

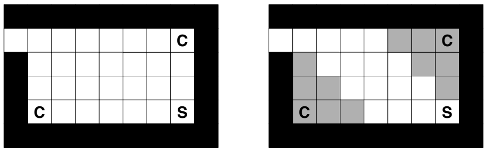


Figure 6: Removing dead cells.

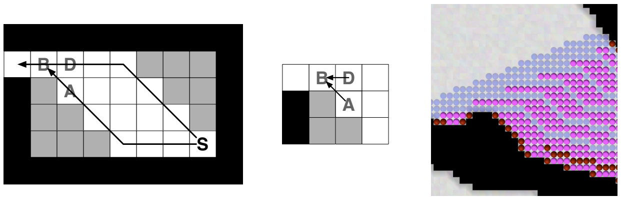


Figure 7: Detecting redundant cells.

nodes have been marked redundant. As can be seen, there are several different paths leading to the areas being explored. Some of these could be eliminated if a more global perspective was known. Ideally, there would be one long path leading to each area being explored, instead of multiple long paths, each of which must be traced in each iteration of the algorithm.

The effectiveness of dead cell and redundant cell pruning will depend on the problem being solved. But, we observe that if nodes can be marked dead and/or redundant at the same rate they are explored, then the g-cost learning approach will be linear in the number of states explored. In particular, for the main example used here (Figure 2), the path down the main diagonal can be marked redundant as soon as the neighbor cells have been explored. In each iteration the nodes from the two iterations previous will be marked as redundant. Thus, the cost of exploring this problem and finding an optimal solution is linear in the size of the map. The elimination of dead and redundant cells is the key idea that speeds the learning of the RIBS algorithm we present in the next section.

6. RIBS

We introduce a new real-time search algorithm, *Real-time Iterative-deepening Best-first Search (RIBS)*, which learns costs-so-far (g-costs) as described in the previous section. At a conceptual level the algorithm may be viewed as an ant-style variant of IDA* that has been adapted to: *(a)* expand successors in a best-first order, and *(b)* use the environment as a memory. Like IDA*, RIBS iteratively searches through the state space expanding all nodes of a given f-cost before returning to the start state and beginning a new iteration with next possible f-cost. It is guaranteed to be both optimal and complete. We describe the algorithm and evaluate it empirically on two disparate problem domains with respect to both lower-bounds on learning and performance.

6.1 Pseudo-code

Pseudo-code for RIBS is given in Figure 8. We use a dot notation to represent data that is stored in a state. For example, $s_n.g$ is the g-cost of state s_n. The top-level function is the main driver, executing the agent's steps in the environment, one at a time, until the goal is reached. The agent starts by progressively exploring the states in the vicinity of the start state that are within a given cost threshold f_{limit}, initially set to the h-cost of the start state. The agent must physically visit each of the states because of the ant-like paradigm of RIBS, backtracking its steps as necessary. When the agent returns to the start state and finds all successors explored ($s_{next} = null$), then the threshold f_{limit} is increased to the lowest f-value of seen but non-expanded states in the iteration just finishing ($f_{nextlimit}$), and the next iteration then begins. This is done until the goal is reached.

The control strategy for choosing the neighboring state to move to is implemented in the *next-state* function. It selects the neighbor of the current state s_c to which the agent should move next. Of the neighbors yet to be visited on the current iteration, the one with the lowest f-cost is chosen with tie-breaking on higher g-costs. Also, the g-value of the current state is updated if it can be reached via a shorter path than previously known from one of the neighbors. This allows the agent to learn the costs-so-far. If no neighbors are eligible then the agent returns to the parent node.

```
RIBS($s_{start}$, $s_{goal}$)
  $f_{nextlimit} \leftarrow \infty$
  $f_{limit} \leftarrow h(s_{start})$
  $s_c \leftarrow s_{start}$
  initialize($s_c$, $null$, 0, $f_{limit}$)
  while ( $s_c \neq s_g$ ) do
    $s_{next} \leftarrow$ next-state($s_c$, $f_{limit}$)
    if ($s_{next} = null$)
      $f_{limit} \leftarrow f_{nextlimit}$
      $f_{nextlimit} \leftarrow \infty$
    else
      $s_c \leftarrow s_{next}$
    end if
  end while
next-state($s_c$, $f_{limit}$)
  for each successor $s_n$ of $s_c$ do
    $g\prime \leftarrow s_n.g + c(s_c, s_n)$
    if (not initialized($s_n$))
      initialize($s_n$, $s_c$, $g\prime$, 0)
    end if
    if ($s_n.f \leq f_{limit}$)
      // Check if a shorter path is found to $s_n$ through $s_c$?
      if ($g\prime < s_n.g$) or ($g\prime = s_n.g$ and $s_n.lim \neq f_{limit}$)
        initialize($s_n$, $s_c$, $g\prime$, $f_{limit}$)
        add $s_n$ to eligible nodes
      end if
    else
      $f_{nextlimit} \leftarrow min(f_{nextlimit}, s_n.f)$
    end if
  end for
  if no eligible nodes
    $s_{next} \leftarrow s_c.parent$ // backtrack
  else
    // ... move to the most eligible node
    $s_{next} \leftarrow$ best eligible node (min $f$ max $g$)
  end if
  return $s_{next}$
initialize($s$, $parent$, $g$, $lim$)
  $s.parent \leftarrow parent$
  $s.g \leftarrow g$
  $s.f \leftarrow s.g + h(s)$
  $s.lim \leftarrow lim$
```

Figure 8: RIBS algorithm.

RIBS maintains all the properties of IDA*, so with an consistent, admissible heuristic the algorithm will be both optimal and complete, assuming that all costs are lower bounded by some fixed $\epsilon > 0$.

The two enhancements of eliminating dead-end and redundant states, although an integral part of the algorithm, are not shown in the pseudo-code as the implementation details are somewhat involved and would unnecessarily deflect from the presentation of the basic underlying algorithm (we refer the reader back to the discussion in the previous section).

6.2 Domain-Specific Analysis

In here we give a brief analysis of RIBS's learning complexity on both exponential and polynomial domains. Let d be the search depth and b the average branching factor.

First, we show a general worst-case bound. Assume that every node in the state space has a unique f-cost. Then, RIBS will expand one new node during each iteration, and the total number of nodes expanded will be $1 + 2 + \cdots + N = N(N + 1)/2$, or $O(N^2)$. Note that IDA* can blow up a graph exponentially, but RIBS uses the environment to detect cycles and so has better performance than IDA* in this regard.

In exponential domains it is usually assumed that the number of nodes expanded in each iteration is b times larger than the previous iteration. Under this assumption the cost of iterative deepening is dominated by the last iteration and the total cost is $O(b^d)$. Assume that all nodes in the last iteration must be expanded. Then, if dead and redundant cell pruning can reduce the size of the previous iterations, it will not asymptotically improve performance. Thus RIBS best-case and average-case performance will be the same – $O(b^d)$ nodes expanded and $O(b^d)$ distance travelled.

A polynomial domain is one where the number of nodes expanded in an iteration to depth d grows as d^k where k is a constant. Consider first an average-case analysis. Let the cost of an iteration to depth d be d^k. Then, the total number of nodes expanded over multiple iterations will be: $1^k + 2^k + 3^k + \cdots + d^k = \sum_{i=0}^{d} i^k$. The result is that the cost of iterations in an iterative deepening algorithm is in general not necessarily amortized by the cost of the last iteration. Instead, the degree of the polynomial is increased by one, for total work $O(d^{k+1})$. So, for a grid-based map where the map size grows as d^2 (where d is the dimension of one side of the map), in general d^3 work might be required to find an optimal solution with an iterative deepening approach, which is $N^{1.5}$, where N is the total number of states in the state space.

In the best case, RIBS will be able to reduce this to $O(N)$. This occurs when the dead and redundant cell pruning techniques are able to prune nodes at the same rate at which they are being explored, as will be the case in Figure 2. These techniques are the key to the asymptotic reduction of RIBS over previous techniques.

7. EMPIRICAL EVALUATION

In this section we evaluate algorithms on two domains, grid-based pathfinding and the sliding-tile puzzle. Experimental evidence supports our work measure as a good predictor for problem difficulty. We then look at how larger lookahead can reduce the learning required, and measure how these algorithms scale.

7.1 Scaling Experiments on Grids

We begin with experiments on grids. On the grid maps cardinal moves have cost 1.0, while diagonal moves have cost 1.5. We use the octile heuristic as the default heuristic, which is a perfect heuristic for 8-connected grids on an empty map without obstacles.

Our first experiment is a scaling experiment, performed on a class of maps like that shown in Figure 2. The start and goal are marked as S and G. This map is 9×9, but we will experiment with the same map from size 5×5 to 1000×1000.

Before discussing and presenting results, consider the problem facing a heuristic-learning agent. The heuristic will lead an agent towards state A, which then must learn enough to fill the local area with higher heuristic values before the goal can be found. This type of problem is typical in grid-based maps, but as far as we are aware, scaling experiments have not been previously performed.

It is difficult to quickly analyze the learning required for this problem, so we turn to our learning metric which is demonstrated in Figure 3. Assuming that the diameter of the map is d, the initial heuristic of the start state will be $1.5(d - 1)$, but should actually be $2(d - 1)$. Thus, the error $(d/2)$ is proportional to the size of the map, and the total heuristic error will be $O(d^3)$ ($O(d)$ error on each of d^2 cells). If there are $N = d^2$ states in the map, then the error, and hence the learning required will be $O(N^{1.5})$. As long as diagonal edges have constant cost less than 2, the learning required will always grow in this manner.

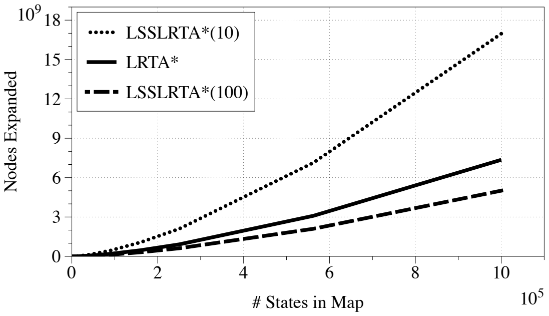


Figure 9: Nodes expanded by learning algorithms when exploring to convergence.

We used our learning measure to predict the amount of learning required for each problem. As our metric predicts, the amount of learning grows as $N^{1.5}$. On a 500×500 map, 4.15×10^7 learning is predicted. This works out to be an average update of 166.9 for every state which requires learning. If the map width and height are doubled, the number of states for which learning is required increases by a factor of 4. So, the total learning per state should increase by a factor of 2. On the 1000×1000 maps an update of 333.5 per state is predicted, which fits this analysis.

Next, we run several learning algorithms on this problem. We compare LRTA* with lookahead of radius 1 with Koenig's LSS-LRTA* [15] with a lookahead of both 10 and 100 nodes. LSS-LRTA* uses a k-node A*-shaped lookahead and updates the heuristic values of nodes within the lookahead based on unexpanded nodes on the border of the lookahead. LSS-LRTA*(1) is equivalent to LRTA*. We measured the total nodes expanded and touched, the total distance travelled, and the total learning. Nodes expanded, touched, and distance travelled all follow a curve that matches $kN^{1.5}$ for appropriate constants k. In Figure 9, because of space limitations, we show only the curves for the number of nodes expanded to convergence. The x-axis shows the number of states in the map ($\times 10^5$) and the y-axis the total number of nodes expanded ($\times 10^9$). LSS-LRTA* with larger lookahead converges faster than LRTA*, but expand more nodes per step, so for LSS-LRTA*(10) the total work (nodes expanded) is larger than LRTA*. According to our learning metric, LRTA* learns about 88% more than the minimum required, and LSS-LRTA* with lookahead of 10 or 100 learned 93% more than the minimum required. The algorithms learn more than required because they fill up the heuristic depression from the inside instead of learning from the optimal path on the outside.

As RIBS only requires a single trial to converge, we plot the first-trial performance for both RIBS with lookahead 1 and LSS-LRTA* with lookahead 100 in Figure 10. We plot performance on a log-log plot so the algorithms performance can be more clearly measured. As expected, on small problems LSS-LRTA* is able to use its lookahead to quickly find a solution to the problem on the first trial, while RIBS with lookahead one slowly explores. But, once the map size approaches 2000 nodes (a 45×45 local minima) RIBS begins to outperform LSS-LRTA*, even on the first trial. On a map with 1 million states, RIBS expands 7.9 million nodes to LSS-LRTA*'s 177 million. The nodes expanded by RIBS is growing at approximately $8N$, while the growth of the LSS-LRTA* curve

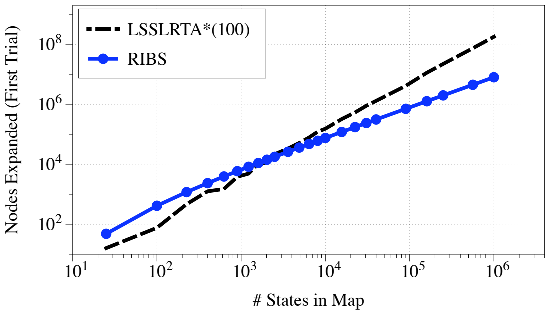


Figure 10: RIBS versus LSS-LRTA* first-trial performance.

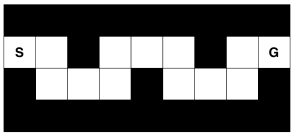


Figure 11: Corridor where heuristic underestimates actual cost.

correlates to $0.09 * x^{1.55}$ with a correlation coefficient of 0.999. Note that without redundant cell pruning RIBS exhibits the same asymptotic growth as the learning algorithms.

This result is clearly the best-case possible. We provide a worst-case example in Figure 11, which is a maze-like corridor with inaccurate heuristic values that will cause RIBS to perform multiple iterations before converging. If the length of the corridor is d, RIBS expands $O(d^2)$ nodes before converging. In this case, LRTA* will find the optimal path in d steps on each trial, but it will take exactly d trials and d^2 steps to converge.

In practice, any real-world problem may have a mix of problems that look like both Figure 2 and Figure 11. The exact mix will likely determine the performance of each algorithm type. Ideally, both of these approaches could be combined in a way that plays to the strengths of each approach, but this is a matter of future work.

7.2 Experiments on Game Maps

Given these results, we then perform experiments on maps from the game Baldur's Gate. We experiment on 75 maps, each of which is scaled to 512×512 in size. On each map we have 1280 problems, evenly distributed in length between 0 and 512. Over all problems the average learning per problem required is $278,644$, and the average state needs to have its heuristic increased by 15.8. On the hardest problems of length 508-512, the average minimum heuristic update is 54.1. We scaled the maps to size 1024×1024 and the learning per state on longs paths increased to 107.45. Thus, the learning required on these maps seems to be growing as $N^{1.5}$ as the maps scale.

On these problems we measured the number of trials, the distance travelled (both on the first trial and to convergence), the total nodes expanded (first trial and to convergence), as well as the ratio of the learning performed to the minimum learning required, according to our metric. Each data point is an average over all problems (75×1280).

Table 1: Comparison over all maps and problems.

	Algorithm			
	LRTA*	LL(10)	LL(100)	RIBS
Avg. Trials	2,707	650	109	1
Trial 1 Dist	44,627	10,288	1,680	210,377
Total Dist	1.56×10^6	360,581	55,548	210,377
Trial 1 Nodes	40,488	114,853	35,572	169,690
Total Nodes	1.3×10^6	2.43×10^6	598,716	169,690
Learn Ratio	3.36	2.80	2.21	-

Table 2: Comparison over longest problems.

		Algorithm	
Size	Measure	LSS-LRTA(100)	RIBS
512^2	Trial 1 Dist	8,326	1.00×10^6
1024^2	Trial 1 Dist	62,524 [7.5×]	6.74×10^6 [6.7×]
512^2	Trial 1 Nodes	192,216	811,001
1024^2	Trial 1 Nodes	1.58×10^6 [8.2×]	5.46×10^6 [6.7×]

We compare LRTA*, LSS-LRTA* (labelled LL) and RIBS in Table 1, noting the following trends. While we have shown that greater lookahead may not asymptotically reduce the amount of learning required, it does have the effect of reducing the actual learning performed in the problem. LRTA*, for instance, performs 3.37 times the minimum learning required, while LSS-LRTA*(100) only performs 2.21 times the minimum learning. Both algorithms are learning in areas of the map that may not be strictly required, but LSS-LRTA* with larger lookahead does a better job of limited the learning in these cases. RIBS has a higher first-trial cost than the other algorithms, but has far better convergence costs, expanding an order of magnitude fewer nodes than LRTA* and LSS-LRTA*(10). RIBS expands only 3.5 times fewer nodes as LSS-LRTA*(100), but this is still a strong result, especially considering that RIBS has a lookahead of only 1.

To gain some insights into how the algorithms behave as the problem difficulty grows, we focus on the longest problems (length $508 - 512$) as the maps grow in size. In Table 2 we show for those problems the distance travelled and nodes expanded on the first trial averaged over all maps as they are scaled up from size 512×512 to 1024×1024. We show the growth of each metric next to the figures. Based on this limited data it is difficult to draw strong conclusions, but the results suggest that the cost of learning required by LSS-LRTA* is growing faster than cost of searching by RIBS. The work required by RIBS, however, is no longer growing linearly, suggesting that there are portions of the map where RIBS has increased the asymptotic cost.

7.3 Experiments on Sliding-Tile Puzzle

In addition to experimenting on a polynomial domain, we also look at an exponential domain, the sliding-tile puzzle. In the sliding-tile puzzle, the cost of iterative deepening is amortized by the cost of the last iteration, so RIBS is expected to perform well in this domain. We tested the same algorithms as before on a set of 500 problems generated by taking a random walk length 100 from the goal state. According to our metric, the average learning required per problem is 154,953. The average learning per state is 2.44. Looking at a histogram for each problem, more than 66% of the states that require learning in each problem only require an update of 2 from the initial heuristic. Thus, in this domain the learning task does not grow as quickly as in pathfinding.

We collected the same kind of average statistics for the 500 sliding-tile puzzle problems as for the game maps. The results of our experiments are shown in Table 3. On the first trial, the learning algorithms have much better performance, traveling only a small fraction of the distance that a RIBS agent does. But, these algorithms take hundreds or thousands of trials to converge. Only LSS-LRTA* with a lookahead of 100 travels a comparable distances to RIBS when converging, but expands over 40 times the number of nodes that RIBS does in the process. Additionally, the learning al-

Table 3: Comparison over 500 sliding-tile puzzle problems.

	Algorithm			
	LRTA*	LL(10)	LL(100)	RIBS
Avg. Trials	6417	2840	455	1
Trial 1 Dist	945	389	130	76,147
Total Dist	3.50×10^7	3.82×10^6	111,151	76,147
Trial 1 Nodes	945	2,928	3,436	76,151
Total Nodes	3.50×10^7	2.97×10^7	3.23×10^6	76,151
Learn Ratio	244.6	97.9	8.8	-

gorithms are much less efficient with their learning in this domain. LRTA* learns 250 times as much as is required for convergence, while LSS-LRTA* learns 8.8 times the minimum required.

8. CONCLUSIONS AND FUTURE WORK

This paper advances the state-of-art in real-time agent-centered heuristic search in several ways. We propose a new learning metric which measures the minimum learning required to converge to an optimal solution for a given problem. Based on this, we show that learning algorithms like LRTA* cannot asymptotically reduce the amount of learning required by increasing their lookahead, although they can reduce the amount of learning they perform above and beyond the minimum required. We show that common problems in grid-based maps require learning that grows as $O(N^{1.5})$ in the number of states in the map. We introduce a new algorithm, RIBS, which can solve a class of these problems in $O(N)$ steps, and show that it outperforms the convergence cost of existing algorithms by several orders of magnitude. As far as we are aware, this is the first work which properly studies the properties of these algorithms as map sizes scale.

The work here suggests many new avenues of research. RIBS here is presented with only a lookahead of distance 1. The best way to scale RIBS to use a larger lookahead is an open question. Additionally, RIBS has poor first-trial performance on many ‘easy’ problems, because it always finds the optimal solution in the first trial. We are interested in studying the many possible extensions to RIBS to improve first-trial performance, including weighted-RIBS and interleaving the RIBS and LRTA*-style algorithms. Finally, our learning metric should be extended to a larger class of problems and adapted to predict first-trial performance in addition to convergence.

In summary, this work provides new analysis and algorithms that push forward the state-of-art in real-time heuristic search.

9. REFERENCES

[1] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[2] Y. Björnsson, V. Bulitko, and N. Sturtevant. TBA*: Time-bounded A*. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, Pasadena, California, 2009. AAAI Press.

[3] V. Bulitko. Learning for adaptive real-time search. Technical Report http: // arxiv. org / abs / cs.AI / 0407016, Computer Science Research Repository (CoRR), 2004.

[4] V. Bulitko and G. Lee. Learning in real time search: A unifying framework. *Journal of Artificial Intelligence Research (JAIR)*, 25:119–157, 2006.

[5] D. Furcy and S. Koenig. Speeding up the convergence of real-time search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 891–897, 2000.

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

[7] C. Hernández and P. Meseguer. LRTA*(k). In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1238–1243, Edinburgh, UK, 2005.

[8] T. Ishida. Moving target search with intelligence. In *Proceedings of the National Conference on Artificial Intelligence*, pages 525–532, 1992.

[9] T. Ishida and R. Korf. Moving target search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 204–210, 1991.

[10] S. Koenig. The complexity of real-time search. Technical Report CMU–CS–92–145, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1992.

[11] S. Koenig. Agent-centered search. *Artificial Intelligence Magazine*, 22(4):109–132, 2001.

[12] S. Koenig. A comparison of fast search methods for real-time situated agents. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 2*, pages 864 – 871, 2004.

[13] S. Koenig and M. Likhachev. Real-time adaptive A*. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 281–288, 2006.

[14] S. Koenig and R. G. Simmons. Complexity analysis of real-time reinforcement learning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 99–105, 1993.

[15] S. Koenig and X. Sun. Comparing real-time and incremental heuristic search for real-time situated agents. *Autonomous Agents and Multi-Agent Systems*, 18(3):313–341, 2009.

[16] R. Korf. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(3):97–109, 1985.

[17] R. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2-3):189–211, 1990.

[18] G. Lee, V. Bulitko, and I. Nikolaidis. nLRTS: Improving distance vector routing in sensor networks. In *Proceedings of the National Conference on Artificial Intelligence (AAAI), Workshop on Learning For Search*, pages 101–107, Boston, Massachusetts, 2006.

[19] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun. Anytime dynamic A*: An anytime, replanning algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2005.

[20] M. Mizusawa and M. Kurihara. Hardness measures for gridworld benchmarks and performance analysis of real-time heuristic search algorithms. *Journal of Heuristics*, 2008.

[21] D. C. Rayner, K. Davison, V. Bulitko, K. Anderson, and J. Lu. Real-time heuristic search with a priority queue. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2372–2377, Hyderabad, 2007.

[22] S. Russell and E. Wefald. *Do the right thing: Studies in limited rationality*. MIT Press, 1991.

[23] A. Shiloni, N. Agmon, and G. A. Kaminka. Of robot ants and elephants. In *AAMAS ’09: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 81–88, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[24] M. Shimbo and T. Ishida. Controlling the learning process of real-time heuristic search. *Artificial Intelligence*, 146(1):1–41, 2003.

[25] L.-Y. Shue, S.-T. Li, and R. Zamani. An intelligent heuristic algorithm for project scheduling problems. In *Proceedings of the Thirty Second Annual Meeting of the Decision Sciences Institute*, San Francisco, 2001.

[26] L.-Y. Shue and R. Zamani. An admissible heuristic search algorithm. In *Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems*, volume 689 of *LNAI*, pages 69–75. Springer Verlag, 1993.

[27] A. Stenz. The focussed D* algorithm for real-time replanning. In *Proceedings of the International Conference on Artificial Intelligence*, pages 1652–1659, 1995.

[28] Ubisoft Montreal. Far Cry 2, 2008.