Recursive Best-First Search with Bounded Overhead

Matthew Hatem and **Scott Kiesel** and **Wheeler Ruml**
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
mhatem and skiesel and ruml at cs.unh.edu

Abstract

There are two major paradigms for linear-space heuristic search: iterative deepening (IDA*) and recursive best-first search (RBFS). While the node regeneration overhead of IDA* is easily characterized in terms of the heuristic branching factor, the overhead of RBFS depends on how widely the promising nodes are separated in the search tree, and is harder to anticipate. In this paper, we present two simple techniques for improving the performance of RBFS while maintaining its advantages over IDA*. While these techniques work well in practice, they do not provide any theoretical bounds on the amount of regeneration overhead. To this end, we introduce RBFS_{CR}, the first method for provably bounding the regeneration overhead of RBFS. We show empirically that this improves its performance in several domains, both for optimal and suboptimal search, and also yields a better linear-space anytime heuristic search. RBFS_{CR} is the first linear space best-first search robust enough to solve a variety of domains with varying operator costs.

Introduction

Linear-space search algorithms only require memory that is linear in the depth of the search. Iterative Deepening A* (IDA*, Korf 1985) is a linear space analog to A* that does not keep an open or closed list. If its evaluation function f is monotonic (f is nondecreasing along every path), then IDA* expands nodes in best-first order. However, if f is non-monotonic, IDA* tends to expand nodes in depth-first order (Korf 1993).

Weighted A* (WA*, Pohl 1973) is a popular bounded suboptimal variant of A* that uses a non-monotonic evaluation function to find w-admissible solutions, solutions with cost bounded by a specified factor w of an optimal solution. Weighted IDA* (WIDA*) is the corresponding bounded suboptimal variant of IDA* and it uses the same non-monotonic function. Unlike WA*, which converges to greedy search as its suboptimality bound increases, WIDA* behaves more like depth-first search, resulting in longer, more costly solutions and in some cases, longer solving times than WA*. Thus WIDA* is not an ideal analog for WA*. Recursive Best-First Search (RBFS, Korf 1993) on the other hand, is a better linear space analog to WA*. It uses the same best-first search order as WA* and returns cheaper solutions than WIDA*.

Like IDA*, RBFS suffers from node regeneration overhead in return for its linear space complexity. While the node regeneration overhead of IDA* is easily characterized in terms of the heuristic branching factor (Korf 1985), the overhead of RBFS depends on how widely the promising nodes are separated in the search tree. Moreover, as we explain below, RBFS fails on domains that exhibit a large range of f values. The main contribution of this paper is a set of techniques for improving the performance of RBFS. We start with two simple techniques that perform well in practice but provide no theoretical guarantees on performance. We present a third technique that is more complex to implement but performs well in practice and comes with a bound on reexpansion overhead, making it the first linear space best-first search robust enough to solve a variety of domains with varying operator costs. Because of RBFS's best-first search order, it also leads to a better anytime algorithm. While IDA* enjoys widespread popularity, we hope this work encourages further investigation of linear-space techniques that maintain a best-first search order.

Previous Work

Iterative Deepening A*

IDA* performs iterations of bounded depth-first search where a path is pruned if $f(n)$ becomes greater than the bound for the current iteration. After each unsuccessful iteration, the bound is increased to the minimum f value among the nodes that were generated but not expanded in the previous iteration. Each iteration of IDA* expands a superset of the nodes in the previous iteration. If the size of iterations grows geometrically, then the number of nodes expanded by IDA* is $O(N)$, where N is the number of nodes that A* would expand (Korf 1985).

In domains with a wide range of edge costs, there can be few nodes with the same f value and the standard technique of updating the bound to the minimum f value of unexpanded nodes may cause IDA* to only expand a few new nodes in each iteration. The number of nodes expanded by IDA* can be $O(N^2)$ in the worst case when the number of new nodes expanded in each iteration is constant (Sarkar et al. 1991). To alleviate this problem, Sarkar et al. introduce

```
RBFS(n, B)
 1. if n is a goal
 2.    solution ← n; exit()
 3. C ← expand(n)
 4. if C is empty, return ∞
 5. for each child n_i in C
 6.    if f(n) < F(n) then F(n_i) ← max(F(n), f(n_i))
 7.    else F(n_i) ← f(n_i)
 8. (n_1, n_2) ← best_F(C)
 9. while (F(n_1) ≤ B and F(n_1) < ∞)
10.    F(n_1) ← RBFS(n_1, min(B, F(n_2)))
11.    (n_1, n_2) ← best_F(C)
12. return F(n_1)
```

Figure 1: Pseudo-code for RBFS.

IDA*$_{\text{CR}}$. IDA*$_{\text{CR}}$ maintains counts in a histogram using a fixed number of buckets estimating the distribution of f values of the pruned nodes during an iteration of search and uses it to find a good threshold for the next iteration. This is achieved by selecting the bound that is estimated to cause the desired number of pruned nodes to be expanded in the next iteration. If the successors of these pruned nodes are not expanded in the next iteration, then this scheme is able to accurately double the number of nodes expanded between iterations. If the successors do fall within the bound on the next iteration, then more nodes may be expanded than desired but this is often not harmful in practice (Burns and Ruml 2013). Since the threshold is increased liberally, branch-and-bound must be used on the final iteration of search to ensure optimality. To the best of our knowledge, this algorithm has never been proven to have bounded overhead, but it appears to perform well in practice.

Weighting the heuristic in IDA* results in Weighted IDA* (WIDA*): a bounded suboptimal linear-space search algorithm using the non-monotonic cost function $f'(n) = g(n)+w\cdot h(n)$. As the specified weight w increases, WIDA* prunes large portions of the search space and is often able to find w-admissible solutions quickly. However, those paths that are not pruned are searched in depth-first order, resulting in significantly more expensive solutions and, in some cases, longer solving times. In contrast, WA* performs more like greedy search as the bound increases, finding cheaper solutions quickly. Thus WIDA* is not an ideal analog for WA*.

Recursive Best-First Search

Unlike IDA*, RBFS expands nodes in best-first order even with a non-monotonic cost function and is thus a better linear-space analog to WA*. Pseudo-code is shown in Figure 1. Its arguments are a node n to be explored and a bound B that represents the best f value of an unexplored node found elsewhere in the search space so far. Each generated child node is given an f value, the usual $g(n_i) + h(n_i)$, and an F value, representing the best known f value of any node below n_i that has not yet been expanded. The F value of a child node is set to f the first time it is ever generated (line 7). We can determine that a child node is being generated for the first time by comparing its parent's f with its parent's backed-up value F (line 6). If $f(n) < F(n)$ then it must have already been expanded and the child nodes must have already been generated. If a child has been generated previously, its F value is set to the maximum of the F value of the parent or its own f value. Propagating the backed-up values down previously explored descendants of n improves efficiency by avoiding the backtracking that would otherwise be necessary when f decreases along a path.

RBFS orders all child nodes by their current F (line 8) and expands the child with the lowest value ($best_{\text{F}}$ returns the best two children according to F). Each child node with an F that is within B is visited by a recursive call using an updated threshold $min(B, F(n_2))$, the minimum cost of a node generated so far but not yet expanded. (If there is just one child then $F(n_2)$ returns ∞.) Setting the threshold this way for each recursive call allows RBFS to be robust to non-monotonic costs. If F goes down along a path, the threshold for the recursive calls will be lower than B to preserve a best-first expansion order. If all child costs exceed the threshold, the search backtracks (lines 9 and 12). When RBFS backtracks (line 10), it backs up either ∞ or the best f value of a node not yet expanded below node n_1, the child with the currently lowest F. Backing up a value of ∞ indicates that there are no fruitful paths below n_1 and prevents RBFS from performing any further search below it.

By tracking the value of the best unexplored node, and backtracking to reach it when necessary, RBFS achieves the same search order as WA* (ignoring ties). In contrast to IDA*, which uses a fixed threshold throughout each iteration of search, RBFS uses a locally adjusted threshold.

The best-first search order of RBFS has several advantages. Like WA*, RBFS can find w-admissible solutions that are often well within the suboptimality bound. Moreover, RBFS can concentrate search effort in the areas of the search space that appear most promising. While IDA* has to repeat search over all previously expanded nodes at each iteration, RBFS only needs to backtrack when a more promising node lies in a different region of the search space. If the most promising nodes are concentrated, not widely distributed, then RBFS can dominate IDA* in terms of the number of node expansions.

The disadvantages of RBFS are largely due to its overhead. RBFS must sort sibling nodes after each node expansion (lines 8 and 11). This work is not required in standard IDA*. When node expansion is fast, IDA* can outperform RBFS even though it performs many more node expansions. However, if expanding nodes is expensive, then RBFS can find solutions faster than IDA* by expanding fewer nodes.

Like IDA*, RBFS suffers from node regeneration overhead in return for its linear space complexity. There are two different sources of node regeneration overhead in both algorithms. First, since neither algorithm keeps a closed list, they are not able to avoid regenerating the same nodes through multiple paths. This limits IDA* and RBFS to problems where each state is reachable by relatively few paths. Second, the algorithms must regenerate the same nodes in each successive iteration of IDA* or each time a subtree is revisited in RBFS.

Recall that RBFS backtracks from a node n as soon as all descendants of n have an F greater than the threshold B.

In the worst case, every node may have a unique f value with successive values located in alternating subtrees of the root, in which case RBFS would backtrack at every node expansion and regenerate all previously generated nodes before expanding a new node, resulting in $O(N^2)$ expansions. Thus, its overhead depends on how widely promising nodes are separated in the search tree. This can be hard to anticipate, making the algorithm a risky proposition. The contributions we present in the next section address this source of overhead for RBFS, thereby making it more practical.

RBFS with Controlled Reexpansion

In this section, we present three new techniques for controlling the regeneration overhead of RBFS. We start with two simple techniques that work well in practice but lack any provable guarantees. The third, more complex technique, works well in practice and provides a provable guarantee of bounded overhead, resulting in the first practical version of RBFS for domains that exhibit a large range of f values.

RBFS$_\epsilon$ and RBFS$_{kthrt}$

In RBFS$_\epsilon$, the test against the upper bound (at line 9) is replaced by $F(n_1) \leq B + \epsilon$. Increasing ϵ relaxes the best-first search order of RBFS, causing it to persevere in each subtree. Backtracking becomes less frequent and the number of reexpansions decreases substantially. Because RBFS$_\epsilon$ relaxes the best-first search order, it must perform branch-and-bound once an incumbent solution is found if optimality is required. In general, RBFS$_\epsilon$ can guarantee that its solution is within a suboptimality bound w by terminating only when an incumbent goal node n satisfies $f(n) \leq w \cdot B$. As we will see below, this simple change to RBFS works well in practice. Unfortunately, it does not improve the theoretical upper bound of $O(N^2)$ total expansions. Moreover, the best choice of ϵ is likely to be domain or even instance specific.

Taking this idea further, when performing a bounded suboptimal RBFS, we can both loosen the optimality constraint and relax backtracking. For example, we can take the square root of w, yielding the evaluation function $f'(n) = g(n)+\sqrt{w}\cdot h(n)$, and multiply the upper bound B by $\sqrt{w}$ (at line 9). (In general, the evaluation function for RBFS$_{kthrt}$ is $f'(n) = g(n)+w^{\frac{k-1}{k}}\cdot h(n)$ and the threshold B is multiplied by $w^{\frac{1}{k}}$). Combining a portion of the suboptimality bound with B reduces the amount of backtracking while combining the remaining portion with the heuristic results in fewer node expansions to find a w-admissible solution. The advantage of this technique is that it does not require an additional user supplied parameter ϵ or branch-and-bound once a solution is found. However, it does not apply to optimal search.

Theorem 1 *When RBFS$_{kthrt}$ expands a goal node, it is w-admissible, assuming the heuristic is admissible.*

Proof: Let C^* be the cost of an optimal solution and C be the cost of the goal returned by RBFS$_{kthrt}$ and assume by contradiction that $C > w \cdot C^*$. Let B be the F value of the next best node on the frontier at the time the goal was expanded. Since the goal node was expanded it holds that $C \leq w^{\frac{1}{k}} \cdot B$. Also, since the optimal goal was not expanded then there is some node p that is on the optimal path to that goal with $f'(p) \geq B$. Therefore:

$$\begin{aligned} C &\leq w^{\frac{1}{k}} \cdot B \leq w^{\frac{1}{k}} \cdot f'(p) \\ &\leq w^{\frac{1}{k}} \cdot (g(p) + w^{\frac{k-1}{k}} \cdot h(p)) \\ &\leq w \cdot (g(p) + h(p)) \leq w \cdot C^* \end{aligned}$$

This contradicts the assumption that $C > w \cdot C^*$. □

The redeeming feature of RBFS$_\epsilon$ and RBFS$_{kthrt}$ is that they are simple to implement and RBFS$_{kthrt}$ does not require the branch-and-bound of RBFS$_\epsilon$. As we will see below, they outperform RBFS on the domains tested, especially domains with a wide range of f values.

RBFS$_{\text{CR}}$(n, B, B_{CR})
1. if $f(solution) \leq B$, return (∞, ∞)
2. if n is a goal,
3. $\quad solution \leftarrow n$; return ($\infty$, ∞)
4. $C \leftarrow expand(n)$
5. if C is empty, return (∞, ∞)
6. for each child n_i in C
7. $\quad$ if $f(n) < F_{\text{CR}}(n)$,
8. $\quad\quad F(n_i) \leftarrow max(F(n), f(n_i))$
9. $\quad\quad F_{\text{CR}}(n_i) \leftarrow max(F_{\text{CR}}(n), f(n_i))$
10. $\quad$ else
11. $\quad\quad F(n_i) \leftarrow f(n_i)$
12. $\quad\quad F_{\text{CR}}(n_i) \leftarrow f(n_i)$
13. $(n_{\text{CR1}}, n_{\text{CR2}}) \leftarrow best_{\text{CR}}(C)$
14. $(n_{\text{F1}}, n_{\text{F2}}) \leftarrow best_{\text{F}}(C)$
15. while ($F_{\text{CR}}(n_{\text{CR1}}) \leq B_{\text{CR}}$ and $F(n_{\text{F1}}) < f(solution)$)
16. $\quad (B', B'_{\text{CR}}) \leftarrow (min(B, F(n_{\text{F2}})), min(B_{\text{CR}}, F_{\text{CR}}(n_{\text{CR2}})))$
17. $\quad (F(n_{\text{CR1}}), F_{\text{CR}}(n_{\text{CR1}})) \leftarrow$ RBFS$_{\text{CR}}$(n_{CR1}, B', B'_{CR})
18. $\quad (n_{\text{CR1}}, n_{\text{CR2}}) \leftarrow best_{\text{CR}}(C)$
19. $\quad (n_{\text{F1}}, n_{\text{F2}}) \leftarrow best_{\text{F}}(C)$
20. for each child n_i in C
21. $\quad$ if n_i is a leaf, add $f(n_i)$ to the f-distribution for n
22. $\quad$ else merge the f-distributions of n_i and n
23. $F'_{\text{CR}} \leftarrow$ select based on f-distribution for n
24. return $(F(n_{\text{F1}}), F'_{\text{CR}})$

Figure 2: Pseudo-code for RBFS$_{\text{CR}}$.

RBFS$_{\text{CR}}$

We can provide provable guarantees of bounded reexpansion overhead in RBFS using a technique inspired by IDA*$_{\text{CR}}$. IDA*$_{\text{CR}}$ maintains a single histogram of f values of all nodes pruned during an iteration of IDA*. This is sufficient because IDA* always starts each iteration at the initial state. Extending this technique directly to RBFS is not straightforward because RBFS does not always backtrack to the root of the search tree before revisiting a subtree. Any visit to a subtree could consume the rest of the search. A top-down control strategy seems inappropriate for managing the intricate backtracking of RBFS.

Note that all we need in order to bound regeneration overhead is to guarantee that at least twice as many nodes are expanded below any node each time it is revisited. We can adapt the technique of IDA*$_{\text{CR}}$ by tracking the distribution of f values under each node in the search space. But, rather

than storing a histogram for every node, RBFS$_{CR}$ maintains linear space complexity by storing one histogram at each node along the currently explored path, which is still linear in the depth of the search. In the same way that f costs are backed up, the histogram counts are propagated to parent nodes as the search backtracks.

RBFS$_{CR}$ takes a more distributed, bottom-up approach to reexpansion control than IDA*$_{CR}$ by backing up inflated values. RBFS$_{CR}$ tracks the distribution of all f values pruned below any node p expanded during search and uses it to determine the backed up value of p. The value is selected such that at least twice as many nodes will be expanded below p the next time it is visited. As we will see, this bounds the number of times any node is expanded and bounds all node expansions by $O(N)$, where N is the number of nodes expanded by A*.

The pseudo-code for RBFS$_{CR}$ is given in Figure 2. RBFS$_{CR}$ stores an additional backed up value F_{CR} for every node and uses it as the threshold to perform its best-first search. For any node n, the value $F_{CR}(n)$ is determined according to the histogram of f values of all nodes generated but not expanded below n (lines 20-24). RBFS$_{CR}$ sorts the list of child nodes in order to determine the best children according to both F_{CR} and F (lines 13,14,18,19). The backed up F_{CR} is used to control backtracking according to the next best node not yet explored while the backed up F is used to guarantee admissibility (line 15). Aside from these differences, RBFS$_{CR}$ is essentially the same as RBFS.

In our implementation, we propagated f value statistics up the tree by following parent pointers and updating separate histograms at each parent node. In our experiments, this was more efficient than the alternative of merging the histograms of parent and child nodes as the search backtracks. We used the same bucketing technique of IDA*$_{CR}$ to implement the histograms but other implementations are possible (Burns and Ruml 2013). Like IDA*$_{CR}$, the backed up values are selected from the histogram (line 23) such that the number of nodes expanded on the next visit to node n is estimated to grow at least geometrically. We found doubling to be sufficient in our experiments but other schemes are possible. Because RBFS$_{CR}$ sets the thresholds liberally, it must perform branch-and-bound when an incumbent is found in order to guarantee admissibility. It uses the original F and B values for this purpose (lines 1-3, 15 and 17).

We will now show that, under certain assumptions, the overhead of RBFS$_{CR}$ is bounded. We refer to all nodes generated but not expanded so far during the search as the *virtual frontier*. The best and next best nodes on the virtual frontier are represented by nodes along the current search path with the lowest backed-up values.

Assumption 1 *The histograms maintained by RBFS$_{CR}$ accurately reflect the true distributions of f values below each subtree.*

To avoid the proof depending on the accuracy of the histogram data structure, we assume that there is no loss in precision.

Assumption 2 *The histograms maintained by RBFS$_{CR}$ always contain enough counts to select an appropriate backed-up value.*

This same assumption is made by IDA*$_{CR}$. It is reasonable since search spaces for which this assumption does not hold, would not be amenable to linear-space search.

Lemma 1 *Consider the search as it enters a subtree rooted at node p. Let the next best node on the virtual frontier be q. If $F_{CR}(q) \leq C^*$, then only nodes n such that $f(n) \leq C^*$ are expanded below p.*

Proof: The proof is analogous to that for Lemma 4.1 in Korf (1993). $F_{CR}(q)$ bounds the expansions performed in p. The search backtracks from p once all nodes n below p such that $f(n) \leq F_{CR}(q)$ have been expanded and it does not expand any nodes n such that $f(n) > F_{CR}(q)$. Since $F_{CR}(q) \leq C^*$, then all expanded nodes have $f(n) \leq C^*$. □

Lemma 2 *Consider any subtree rooted at a node p. The value $F_{CR}(p)$ always increases monotonically.*

Proof: Let $F'_{CR}(p)$ be the updated value for $F_{CR}(p)$. Only fringe nodes p' below p whose f values exceed $F_{CR}(p)$ are added to p's histogram and used to set $F'_{CR}(p)$. Thus, all of the fringe nodes in the histogram have an f that exceeds $F_{CR}(p)$. We set $F'_{CR}(p)$ such that there is at least one fringe node p with $f(p) \leq F'_{CR}(p)$. Therefore, $F_{CR}(p) < F'_{CR}(p)$. □

Lemma 3 *Consider the search as it enters a subtree rooted at node p. Every path whose nodes have f values $\leq F(p)$ will be explored and nodes that have already been expanded will be reexpanded once before reaching the frontier.*

Proof: Let the next best node on the virtual frontier be q. The value $F_{CR}(q)$ bounds the expansions performed in p. The best-first search order guarantees that $F_{CR}(p) \leq F_{CR}(q)$. The search will not backtrack from the subtree until all paths below p with nodes p' such that $f(p') \leq F_{CR}(p)$ have been expanded. Note that all such nodes p' below p that were already expanded previously have $F_{CR}(p') = max(F_{CR}(p), f(p'))$ (lines 7-9) and all recursive calls below node p have the form RBFS$_{CR}(p, B)$ for some B, where $F_{CR}(p) \leq F_{CR}(p') \leq B$. Since $F_{CR}(p) \leq F_{CR}(p')$ and $F_{CR}(p')$ is guaranteed to increase for each call to RBFS$_{CR}(p', B)$ by Lemma 2, it follows that any node p' below p will not qualify for reexpansion until the search reaches the frontier. □

Because RBFS$_{CR}$ sets B liberally, we will need to make an assumption on the number of extra nodes that qualify for expansion.

Assumption 3 *The number of unique nodes n such that $C^* < f(n) \leq f(q)$ is $O(N)$ where q is the node with the largest f generated during a best-first search and N is the number of nodes expanded by A* when no duplicates are pruned and with worst-case tie-breaking.*

This assumption is reasonable, for example, in domains in which g and h values are computed in terms of the costs of a fixed set of operators. This often implies that f values increase by bounded amounts and that the number of nodes in successive f layers are related by a constant (known as the heuristic branching factor (Korf and Reid 1998)).

Lemma 4 *: The total number of unique nodes visited during search is bounded by $O(N)$.*

Proof: Consider all nodes n expanded by the search and let node q be the generated node with the highest f value. Either $f(n) \leq C^*$ or $C^* < f(n) \leq f(q)$. The total number of unique nodes n with $f(n) \leq C^*$ is at most N. The total number of unique nodes with $C^* < f(n) \leq f(q)$ is $O(N)$ by Assumption 3. Thus, the total number of unique nodes visited by the search is $N + O(N) = O(N)$. □

Lemma 5 *At least as many nodes are expanded below a subtree p as we anticipated when selecting $F_{CR}(p)$.*

Proof: Let $F'_{\text{CR}}(p)$ be the newly backed up value the next time node p is expanded. When p is expanded with $F'_{\text{CR}}(p)$ we will expand all the nodes expanded when p was expanded with $F_{\text{CR}}(p)$ (by Lemma 3 and $F(p) \leq F_{\text{CR}}(p)$). Furthermore, the fringe nodes p', whose f values were added to the distribution last time and were used to compute $F'_{\text{CR}}(p)$, and that we anticipated would fall below $F_{\text{CR}}(p)$, will be initialized with $F_{\text{CR}}(p') = f(p')$ in line 12. Thus, they will pass the test in line 15 and be expanded in the call at line 17. □

Lemma 6 *RBFS$_{CR}$ terminates with a solution if one exists.*

Proof: From Lemma 2 and Lemma 5 the bound at each subtree is always increasing and all nodes below the bound are expanded during search. Eventually the bound when exploring any subtree is larger than or equal to the cost of a solution and a goal node is expanded. □

Theorem 2 *If solution exists and A* expands N nodes before terminating with a solution, then RBFS$_{CR}$ expands $O(N)$ nodes before terminating with a solution.*

Proof: From Lemma 6 we already proved completeness so we need only show an upper bound on reexpansion overhead. $F_{\text{CR}}(p)$ is set when backtracking to cause twice as many nodes to be expanded the next time p is expanded and by Lemma 5 at least this many nodes will be expanded. This implies at least doubling the number of nodes expanded each time we visit a subtree. Then by Lemma 3 each node previously expanded below p is expanded one more time before reaching the frontier when we renter the subtree at p. Note that this implies that we can't expand a node a second time without expanding at least one new node for the first time. Further, when we expand a node that had been expanded twice for a third time, we also expand all the nodes below it that had been expanded once for a second time, and then expand a number of nodes for the first time that is at least the sum of the number of nodes expanded for a third or second time. In general, let s_i be the set of nodes that have been expanded i times and let $n_i = |s_i|$. The sizes of these sets form a series in which $n_{i+1} \leq n_i/2$.

Because every unique node expanded is expanded some number of times, the total number of unique nodes expanded equals $\sum_i^\infty n_i$. The total number of expansions (including reexpansions) is:

$$\begin{aligned} \textstyle\sum_{i=1}^{\infty} i(n_i) &\leq 1(n_1) + 2(\tfrac{n_1}{2}) + 3(\tfrac{n_1}{4}) + 4(\tfrac{n_1}{8}) + ... \\ &\leq \textstyle\sum_{i=1}^{\infty} i(n_1/2^{(i-1)}) \\ &\leq 2n_1 \textstyle\sum_{i=1}^{\infty} i(2^{-i}) < 4n_1 \end{aligned}$$

	Time	Exp.	Exp./Sec.	Reopened
IDA*$_{\text{CR}}$	18,394	2,044m	112k	2,042m
RBFS$_{\text{CR}}$	1,824	188m	103k	87m
RBFS$_{\epsilon=1}$	3,222	472m	147k	4m
RBFS$_{\epsilon=2}$	1,795	251m	140k	5m
RBFS$_{\epsilon=3}$	1,094	154m	141k	14m

Table 1: Solving all Dockyard instances optimally. Times reported in seconds.

By Lemma 4, $n_1 = O(N)$, so the total number of expansions is also $O(N)$. □

Experiments

We compared the new algorithms to standard RBFS and WIDA* (and WIDA*$_{\text{CR}}$) on a variety of domains with different edge costs. All algorithms were written in Java and compiled with OpenJDK 1.6.0.24. All of our experiments were run on a machine with a dual-core CoreDuo 3.16 GHz processor and 8 GB of RAM running Linux.

Sliding-Tile Puzzle

First, we evaluated RBFS$_\epsilon$, RBFS$_{kthrt}$ and RBFS$_{\text{CR}}$ on Korf's 100 15-puzzles (Korf 1985) using unit edge costs and the Manhattan distance heuristic. Our techniques have no advantage over standard RBFS or WIDA* on this domain. WIDA* does not have to sort child nodes at each node expansion and is thus 4 times faster than the RBFS-based algorithms on average. However, the RBFS-based algorithms always return cheaper solutions.

To demonstrate the advantage of our techniques, we change the domain slightly. The cost to move a tile is now the square root of the number on the face of the tile. This provides a wide range of edge costs. Standard WIDA* and RBFS are unable to solve instances in a reasonable amount of time, motivating IDA*$_{\text{CR}}$ and RBFS$_{\text{CR}}$. In this setting RBFS$_{\text{CR}}$ solves all instances at all suboptimality bounds tested (1.5-4) and is just 3.8 times slower than IDA*$_{\text{CR}}$. However, RBFS$_{\text{CR}}$ expands fewer nodes than all other algorithms and provides significantly cheaper solutions than IDA*$_{\text{CR}}$. RBFS$_\epsilon$ with $\epsilon = 16$ and RBFS$_{kthrt}$ with $k = 5$ are 1.7 times faster than RBFS$_{\text{CR}}$ on average because they have less overhead and also return cheaper solutions than IDA*$_{\text{CR}}$.

Dockyard Robot Planning

To evaluate the performance of RBFS$_{\text{CR}}$ on a domain where node expansion is slow, we implemented a planning domain inspired by the dockyard robot example used throughout the textbook by Ghallab, Nau, and Traverso (2004). In this domain, containers must be moved from their initial locations to their destinations via a robot that can carry only a single container at a time. The containers at each location form a stack and only the top container can be moved by using a crane. Actions in this domain have real-valued costs and provide a wide range of f values. We conducted these experiments on a configuration with 5 locations, cranes, piles and 8 containers. Unlike the sliding-tile puzzle, this domain has many paths to the same state, making algorithms that do not

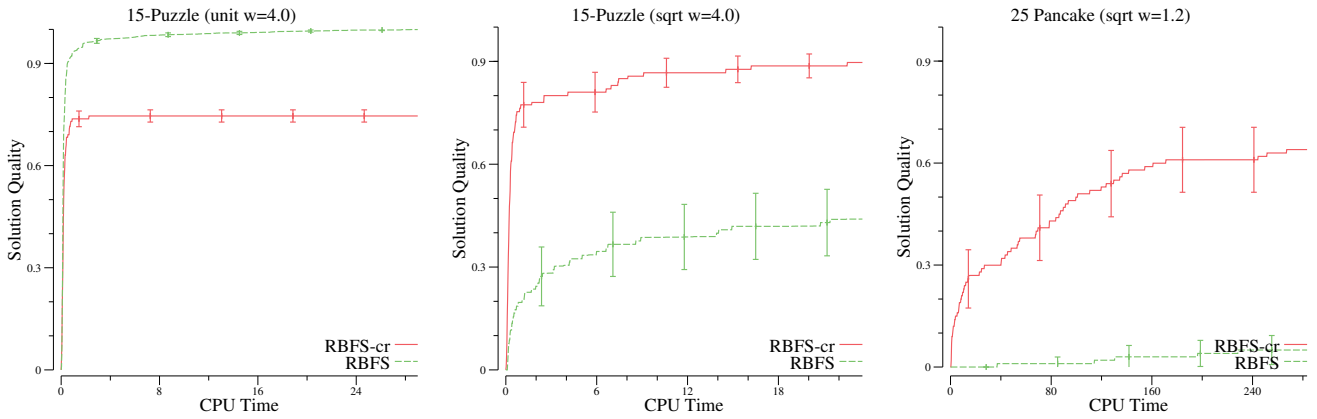

Figure 3: Anytime Search Profiles for the 15-Puzzle and the Pancake Puzzle.

keep a closed list impractical. One popular technique is to combine IDA* with a transposition table which stores a portion of the closed list and can significantly improve the performance of IDA* on domains with many duplicate states. For these experiments we combined IDA*$_{CR}$ and RBFS$_{CR}$ with a transposition table. We used an admissible and consistent heuristic and an unlimited table, thus avoiding the pitfalls reported by Akagi, Kishimoto, and Fukunaga (2010).

Table 1 summarizes the results for dockyard robots. These results show that IDA*$_{CR}$ has a slightly faster node expansion rate; however, it performs more than 10 times as many node expansions, leading to significantly slower solving times. In the last column, we show the number of times a node was *reopened*. A node is reopened when the search regenerates an expanded state through a better path. IDA*$_{CR}$ suffers from having to reopen the same nodes more often because its search order is depth first. The RBFS-based algorithms are best-first and often generate nodes through an optimal path first, avoiding the cost of reopening them.

Anytime Heuristic Search

Hansen and Zhou (2007) propose a simple method for converting a suboptimal heuristic search into an anytime algorithm: simply continue searching after finding the first solution. They show positive results for using RBFS as an anytime algorithm. We compared the anytime profiles of RBFS and RBFS$_{CR}$. Each algorithm was given a total of 5 minutes to find and improve a solution. In Figure 3 the x-axis is CPU time and the y-axis is solution quality. Each data point is computed in a paired manner by determining the best solution found on each instance by any algorithm and dividing this value by the incumbent solution's cost at each time value on the same instance (Coles et al. 2012). Incumbents are initialized to infinity, which allows for comparisons between algorithms at times before all instances are solved. The lines show the mean over the instance set and the error bars show the 95% confidence interval on the mean.

We used the same Korf 100 15-Puzzle instances with unit and square root cost. The left plot in Figure 3 summarizes the results for unit costs. An initial suboptimality bound of 4.0 was selected to minimize time to first solution for both algorithms. RBFS and RBFS$_{CR}$ are able to find solutions quickly. However, because RBFS$_{CR}$ is not strictly best-first, it finds solutions of worse quality than RBFS and is slow to improve. RBFS$_{CR}$ has no advantage over RBFS in this unit-cost setting. The middle plot in Figure 3 summarizes the results for sqrt costs with the same initial suboptimality bound of 4.0. In this plot we clearly see that RBFS$_{CR}$ finds higher quality solutions faster than RBFS.

The right plot shows similar results for a third domain, the pancake puzzle. In this domain we must order a permutation of $\{1, ..., N\}$, where N is the number of pancakes, by reversing a contiguous prefix. We used a stack of 25 pancakes with a similar sqrt cost function and adapted the gap heuristic for sqrt costs. The gap heuristic is a landmark heuristic that counts the number of non-adjacent pancakes or "gaps" for each pancake in the stack (Helmert 2010). We used an initial bound of 1.2. Again, we clearly see that RBFS$_{CR}$ outperforms standard RBFS, making it a more robust algorithm in an anytime setting with non-unit costs.

Conclusion

We presented three techniques for controlling the overhead caused by excessive backtracking in RBFS. RBFS$_{\epsilon}$ and RBFS$_{kthrt}$ are simple although they do not provide provable guarantees on performance. RBFS$_{CR}$ is a more complex technique that provides provable guarantees of bounded re-expansion overhead. We showed that these new algorithms perform well in practice, finding solutions faster than RBFS on non-unit cost domains and solutions that are significantly cheaper than WIDA*. RBFS$_{CR}$ is the first linear space best-first search capable of solving problems with a wide variety of f values, especially when node expansion is expensive or when cheap solutions are useful, as in anytime search. While IDA* enjoys widespread popularity, we hope this work encourages further investigation of linear-space techniques that maintain a best-first search order.

Acknowledgments

We gratefully acknowledge support from the NSF (grant 1150068). We thank Rich Korf, Arial Felner and Roni Stern for helpful discussions that started us on this path of research.

References

Akagi, Y.; Kishimoto, A.; and Fukunaga, A. 2010. On transposition tables for single-agent search and planning: Summary of results. In *Proceedings of the Symposium on Combinatorial Search (SoCS-10)*.

Burns, E., and Ruml, W. 2013. Iterative-deepening search with on-line tree size prediction. *Annals of Mathematics and Artificial Intelligence* S68:1–23.

Coles, A.; Coles, A.; Olaya, A. G.; Jiménez, S.; López, C. L.; Sanner, S.; and Yoon, S. 2012. A Survey of the Seventh International Planning Competition. *AI Magazine* 33(1):83–88.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. San Francisco, CA: Morgan Kaufmann.

Hansen, E. A., and Zhou, R. 2007. Anytime heuristic search. *Journal of Artificial Intelligence Research* 28:267–297.

Helmert, M. 2010. Landmark heuristics for the pancake problem. In *Proceedings of the Symposium on Combinatorial Search (SOCS-10)*.

Korf, R., and Reid, M. 1998. Complexity analysis of admissibe heuristic search. In *Proceedings of the National Conference on Artificial Intelligence*, 305–310.

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

Korf, R. E. 1993. Linear-space best-first search. *Artificial Intelligence* 62(1):41–78.

Pohl, I. 1973. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computation issues in heuristic problem solving. In *Proceedings of IJCAI-73*, 12–17.

Sarkar, U.; Chakrabarti, P.; Ghose, S.; and Sarkar, S. D. 1991. Reducing reexpansions in iterative-deepening search by controlling cutoff bounds. *Artificial Intelligence* 50:207–221.