Bounded-Suboptimal Beam Search

Devin W. Thomas, Stephen Wissow, Michael Bauer, Paige McAfee, and Wheeler Ruml
University of New Hampshire, USA
dwt@cs.unh.edu, sjw@cs.unh.edu, Michael.Bauer@unh.edu, pdmcafee@unh.edu, ruml@cs.unh.edu

Abstract

Beam search is a popular heuristic search method based on breadth-first search. It often finds a solution quickly, but no bound is provided on the solution's cost, so it can potentially be much more costly than an optimal one. Several heuristic search methods with bounded suboptimality have been proposed, but to date, all of them have been based on best-first search. In this paper, we present two approaches for adapting beam searches to be bounded-suboptimal. In a preliminary empirical comparison to state-of-the-art methods, we find that bounded-suboptimal beam search outperforms state-of-the-art bounded-suboptimal heuristic search algorithms in the 15 puzzle domain. To our knowledge, these are the first state-of-the-art bounded-suboptimal methods that are not based on best-first search.

Introduction

Heuristic search is frequently used in many real-world applications. These include many applications of planning, including games (Bulitko et al. 2011), extraterrestrial exploration (Mudgal et al. 2005), autonomous warehouses (Wurman, D'Andrea, and Mountz 2008), and self-driving cars (Dolgov et al. 2008). The traditional objective of search is to return an optimal (minimal cost) plan. However, many planning problems are so challenging that it is impractical to find an optimal plan, due to constraints on computation time or memory. A bounded-suboptimal (BSO) search algorithm returns a plan with a cost within a fixed factor of the cost of the optimal solution. This allows a faster search that generates fewer states, while still providing a guarantee on the quality of the returned solution.

The traditional approach to optimal heuristic search is A* (Hart, Nilsson, and Raphael 1968), which finds optimal plans by exploring every possible plan that could cost less than an optimal plan. Often, much of the search effort is spent proving the optimality of the plan, rather than on finding it, even when A* has an almost-perfect heuristic (Helmert and Röger 2008). Suboptimal search, which is allowed to return a plan with any cost, is often faster because it is not required to find (or prove that it has found) an optimal plan. A common suboptimal search setting, called greedy (or sometimes satisficing or unboundedly suboptimal) search, has the objective of minimizing the time spent computing a plan.

Existing approaches to bounded-suboptimal search, such as Weighted A* (Pohl 1973), Φ_{XDP} (Chen and Sturtevant 2019), DPS (Gilon, Felner, and Stern 2016), and RR-d (Fickert, Gu, and Ruml 2022) are all based on best-first search. However, this is not the case in the greedy setting, where Beam Search (Newell 1980; Bisiani 1987) is a popular approach based on breadth-first search. Beam search explores layer by layer in increasing depth, but limits itself to considering only a fixed-size portion of each depth layer. While beam search is incomplete, it is empirically performant (Wissow, Yu, and Ruml 2024), particularly the beam search variant Bead (Lemons et al. 2022). In Bead, depth layers are pruned by distance-to-goal rather than cost-to-goal, meaning it prefers partial plans with fewer remaining steps (that are thus easier to find) rather than those with less remaining cost. Given beam search's success in suboptimal search, it is natural to wonder if it could be adapted to the bounded-suboptimal setting. Rectangle Search (Lemons et al. 2024) is an adaptation of beam search to the anytime setting that exhibits state-of-the-art performance, and so it is also natural to wonder if that performance could be translated to the bounded-suboptimal setting.

A non-best-first approach is particularly interesting because beam searches do not need to fill local minima to escape them, and therefore have the potential to be faster with the same heuristic. Furthermore, the method we use to adapt beam search to be bounded-suboptimal shows how any algorithm can be made bounded-suboptimal and the method we use to adapt rectangle search to be bounded-suboptimal shows how any anytime algorithm can be made bounded-suboptimal.

In this paper, we present two algorithmic approaches for using beam search in the bounded-suboptimal search setting. These approaches use beam-based searches to explore the state space within the search's current estimate of the user's requested solution suboptimality bound. The search can fall back to A*-style expansions of the least f node in order to raise the estimate of the optimal solution cost before continuing the beam search. We compare the empirical performance of these beam-search based methods against state-of-the-art bounded-suboptimal heuristic search algorithms on the classic 15-puzzle domain. We find that the new methods significantly outperform existing approaches. While further investigation in additional domains is necessary, it ap-

pears that breadth-first search-based bounded-suboptimal algorithms are quite promising.

Background

These methods solve search problems where each problem instance can be represented by a tuple $\langle S, s_i, succ(s), c(s, s'), isG \rangle$, where S is the set of states, $s_i \in S$ the initial state, $succ$ the successor function, and $c(s, s')$ the cost from state $s \in S$ to its successor $s' \in succ(s)$. A sequence of states $s_0, \ldots, s_n$ is a valid solution if $isG(s_n)$ (it ends at a goal), and $\forall i \in 1 \ldots n : s_i \in succ(s_{i-1})$ (the actions are valid starting at s_0). A heuristic is admissible if it is a non-overestimate of the cost to a goal. Search algorithms generally make use of node data structures to represent a (partial) plan. For a node n, $s(n)$ is the final state along the plan represented by n, $g(n)$ is the cost of the plan so far, $h(n)$ is the admissible heuristic cost to reach a goal from $s(n)$, $f(n) = g(n)+h(n)$ is the heuristic admissible estimate of the plan cost, d is the distance to go, which is similar to h except that it measures how many steps the plan has remaining rather than cost, along with $\hat{h}$, $\hat{f}$, and $\hat{d}$ which are estimates (not necessarily admissible) of h, f, and d.

An algorithm is complete if it is guaranteed to find a valid solution if one exists and optimal if it provides a guarantee that there is no solution with less cost than the one returned. An algorithm is called bounded-suboptimal if it is instead guaranteed to return a solution with cost at most $w \times C^*$, where $w \geq 1$ is the suboptimality bound and C^* is the cost of an optimal plan. Anytime algorithms quickly return a (likely suboptimal) solution and continue emitting better solutions until an optimal solution is emitted. The algorithms we discuss in this paper are designed for either optimal, anytime, bounded-suboptimal, or unboundedly suboptimal search.

Beam Search

Beam search (Newell 1980; Bisiani 1987) is a family of algorithms that can be used to solve unboundedly suboptimal search problems. Beam search is a variant of breadth-first search where only up to a constant number of nodes are generated at each depth level of the search; this parameter is the beam width, $width \in \mathbb{Z}^+$ (not to be confused with the suboptimality bound w). Different variants of beam search prioritize candidate nodes for the beam based on different criteria. We will call the beam search variant using the traditional f priority Beam. However, because beam search is intended for the unboundedly suboptimal search setting, Bead (Lemons et al. 2022), a variant that prioritizes low $\hat{d}$ nodes performs better. Bead's good performance can be understood as prioritizing nodes with fewer search steps to reach the goal, which aligns with the objective of unboundedly suboptimal search, rather than prioritizing based on f, the plan cost, which aligns with the objective of optimal search, to prove that no lesser cost solution exists.

Pseudocode for beam search is shown in Algorithm 1. The key data structures are: the beam, which is a fixed-sized array containing the search frontier, and the closed list (cl),

Algorithm 1: Beam Search

```
1: function BEAM(s_i, succ(s), c(s, s'), isG(s), width)
2: n_i ← {s : s_i, g : 0}
3: beam ← {n_i}
4: cl ← {s_i ↦ n_i}
5: while True do
6:     cand ← ∅                                  ▷ Beam Candidates
7:     for n ∈ beam do
8:         for s' ∈ succ(n.s) do
9:             if isG(s') then return Success
10:            g' ← g(n) + c(s, s')
11:            if (s' ∈ cl ∧ g' < g(cl[s'])) then
12:                n' ← {s : s', g : g'}
13:                cand ← (cand \ cl[s']) ∪ {n'}
14:                cl[s'] ← n'
15:            else if s' ∉ cl then
16:                n' ← {s : s', g : g'}
17:                cand ← cand ∪ {n'}
18:                cl[s'] ← n'
19:    beam ← best width from cand       ▷ (e.g., ↓ f, ↓ d̂)
```

a mapping of generated states to their corresponding search node (if one has been generated). For simplicity of presentation, our pseudocode builds the beam lazily from the set of candidate nodes ($cand$). If desired, the new beam can instead be constructed incrementally. Some details have been omitted to be concise (e.g., reconstructing the solution) or because they are domain-specific (e.g., calculating $h, d \ldots$).

We initialize the beam with a single node corresponding to the state s_i with g-value 0 (Line 3), and closed with the mapping from the initial state s_i to the corresponding search node n (Line 4). Then, until a goal is generated (Line 9), we generate (or update) the successors of the beam (Lines 7-18), and select the best $width$ nodes from among them for inclusion in the next beam (Line 19). When used as an unboundedly suboptimal search, beam variants may instead choose to drop duplicates rather than reopening nodes when a lesser cost plan is found. As we will be using beam search as part of a bounded-suboptimal search, we do not drop duplicates because it would make the search not bounded-suboptimal (unlike Weighted A* (Thayer, Ruml, and Kreis 2009)).

Because the search frontier is constrained to be at most $width$ nodes, beam search generates a number of nodes that is linear in the depth of the found solution, in contrast to best-first search methods, which can generate a number of nodes exponential in the solution depth (depending on the domain). Beam search often performs well. However, it is incomplete in most settings, and in practice, it can encounter issues in domains with dead ends.

Rectangle Search

Rectangle Search (Lemons et al. 2024)[1] is an anytime beam search. While beam search explores deeper while maintaining a fixed search width, Rectangle Search explores deeper and wider simultaneously. This enforced exploration at dif-

[1]Rectangle is a refinement of Triangle (Lemons et al. 2023).

ferent depths allows it to escape large local minima that would trap a best-first search and makes the algorithm complete, unlike ordinary beam search.

Algorithm 2 shows pseudocode for Rectangle. The frontier $rect$ is a 0-origin depth-indexed array of priority queues of search nodes. In each iteration it of the outermost loop, Rectangle attempts to expand nodes from every depth level l in $rect$, from shallowest to deepest. When a node is selected from $rect[l]$ (Line 10), it is pruned if it cannot help improve on the incumbent σ (Line 11). When a node is expanded (Line 13), that depth level's expansion counter $ec[l]$ is incremented (Line 12). Successors are also pruned on the incumbent (Lines 15). Unpruned successors that are not detected, stored, and reported as new anytime solutions (Line 16–17) are recorded in cl and enqueued in the next depth layer (Lines 19–20) only if they represent novel states or lower-g paths to existing states (Line 18). Immediately prior to attempting to attempting to expand nodes from the current deepest layer, a new depth level queue is created for any successors that may be generated, along with a new expansion counter (Line 8).

The number of expansions performed at any given (it, l) is limited both by $|rect[l]|$ and by the depth and width allowances $(\delta_\downarrow, \delta_\leftrightarrow)$ computed from the given aspect ratio a (Line 4). In particular, further expansions at (it, l) are only allowed while $ec[l] < it \times \delta_\leftrightarrow$ and $rect[l]$ is non-empty, and new levels are only created at depths $l < it \times \delta_\downarrow$. The original Rectangle Search (Lemons et al. 2024) differs slightly from this method. In each iteration, it performs up to one new expansion from the shallowest to the deepest non-empty open list, and then up to $width$ expansions at each of up to $aspect$ new depth levels. Our new approach to Rectangle enables flexible control over search behavior through the allowance function, which may take any arguments and have any definition, subject only to the requirement that its scalar return value represent an inclusive upper bound on $ec[l]$ at iteration it. The specific allowance function we use in this paper preserves the effective per-(it, l) expansion bounds of the original Rectangle for aspect ratios $a > 1$. For aspect $0 < a < 1$, however, the original Rectangle only adds a depth layer every $\frac{1}{a}$ iterations, whereas our allowance function always one new depth layer and allows up to a additional expansions (increasing width), per iteration. The result is that Algorithm 2 honors the given aspect a by increasing the appropriate dimension by a, and the other dimension by 1, in each iteration (Line 4), so that every iteration may benefit from search in both dimensions.

Algorithm 2: Rectangle Search

1: **function** RECTANGLE($s_i, succ(s), isG(s), c(s, s'), a$)
2: $n_i \leftarrow \{s : s_i, g : 0\}, cl \leftarrow \{s_i \mapsto n_i\}, \sigma \leftarrow \varnothing$
3: $rect \leftarrow \{0 \mapsto \{n\}\}, ec \leftarrow \{0 \mapsto 0\}, it \leftarrow 1, l \leftarrow 0$
4: $(\delta_\downarrow, \delta_\leftrightarrow) \leftarrow (a, 1)$ **if** $a \geq 1$ **else** $(1, \frac{1}{a})$
5: **while** $rect$ has a non-empty level **do**
6: **while** $l < it \times \delta_\downarrow$ **do**
7: **if** $l \geq (it - 1) \times \delta_\downarrow$ **then**
8: $rect \leftarrow rect \cup \{l{+}1 \mapsto \emptyset\}, ec \leftarrow ec \cup \{l{+}1 \mapsto 0\}$
9: **while** $ec[l] < it \times \delta_\leftrightarrow \wedge rect[l] \neq \emptyset$ **do**
10: $n \leftarrow$ extract a lowest-d node from $rect[l]$
11: **if** $\sigma \neq \varnothing \wedge f(n) \geq g(\sigma)$ **then** continue
12: $ec[l] \leftarrow ec[l] + 1$
13: **for** $s' \in succ(s(n))$ **do**
14: $n' \leftarrow \{s : s', g : g(n) + c(s(n), s')\}$
15: **if** $\sigma = \varnothing \vee f(n') < g(\sigma)$ **then**
16: **if** $isG(s')$ **then**
17: $\sigma \leftarrow n'$, report σ
18: **else if** $s' \notin cl \vee g(n') < g(cl[s'])$ **then**
19: $cl[s'] \leftarrow n'$
20: $rect[l + 1] \leftarrow rect[l + 1] \cup \{n'\}$
21: $l \leftarrow l + 1$
22: $it \leftarrow it + 1, l \leftarrow 0$

cause it is commonly used and so it can be a useful point of reference. As w increases, the search behavior converges to the behavior of Greedy Best-first search. Weighted A* has two notable idiosyncrasies: it generally finds solutions that are much lower cost than the suboptimality bound requires (Holte et al. 2019) and the runtime and solution quality can be non-monotonic in w. In practice, this can lead to larger w leading to slower search when h is not well correlated with the steps to goal (Wilt and Ruml 2021).

Bounded-Suboptimal Search

In this section we will describe several existing approaches to bounded-suboptimal search: Weighted A* is a simple and popular approach, Focal Search is a useful framework for analyzing theoretical properties of our methods, and EES and RR-d are state-of-the-art bounded-suboptimal searches.

Weighted A* Weighted A* (Pohl 1973) is a straightforward modification of the A* algorithm where the priority function is modified from $f(n) = g(n) + h(n)$ to $f(n) = g(n) + w \times h(n)$. We have included it in our discussion because it is commonly used and so it can be a useful point of reference. As w increases, the search behavior converges to the behavior of Greedy Best-first search.

Focal Search Focal Search (Pearl and Kim 1982) is a family of bounded-suboptimal search algorithms where the search utilizes two queues: an open list like that used by A*, which is the set of nodes that have been generated but not yet expanded, and the focal list, which may be ordered however the algorithm designer wishes but contains only the nodes n with $f(n) \leq w \times f_{\min}$. Because we maintain the open list, the lowest f value of open nodes ($f_{\min}$) can be quickly queried. Generally, Focal Search places all qualifying nodes onto focal and searches by expanding the best node on focal (traditionally the lowest d node). Focal search is bounded-suboptimal because f is admissible, so $f_{\min} \leq C^*$, and all nodes on focal (and thus all nodes expanded by focal search) have $f(n) \leq w \times f_{\min} \leq w \times C^*$.

The empirical performance of focal search can suffer because the focal priority (low d) focuses exclusively on finding solutions within the current proven suboptimality bound ($w \times f_{\min}$) rather than raising $f_{\min}$ (Thayer, Ruml, and Kreis 2009).

Multi-queue Methods

Bounded-suboptimal searches have two simultaneous objectives. The first is to find a solution as quickly as possible,

which can be viewed as exploiting the search's current estimate of $f_{\min}$, while the second objective is to raise $f_{\min}$ by exploring the search space to prove that the solution is bounded-suboptimal. Explicit Estimation Search (EES) (Thayer and Ruml 2011) and Round-Robin-d (RR-d) (Fickert, Gu, and Ruml 2022) are bounded-suboptimal search algorithms that explicitly try to advance those two objectives. They utilize three queues: an A*-style open list ordered on f and two focal lists, one ordered on $\hat{f}$ and a second ordered on $\hat{d}$. They differ in how the search decides which queue to expand from. RR-d takes a simple round-robin approach, where the search cycles between the three queues in turn. EES tries to make a more reasoned choice: it expands the least $\hat{d}$ node if its expected cost is less than the search's current lower bound on the suboptimality bound ($\hat{f}(best_{\hat{d}}) \leq w \times f_{\min}$), otherwise it tries the least $\hat{f}$ node ($\hat{f}(best_{\hat{f}}) \leq w \times f_{\min}$), and finally if neither of those qualifies, it expands a $\min_f$ node, with the hope that this increases the lower bound $f_{\min}$.

Two Approaches

We now turn to the main contributions of this paper: two approaches to adapting a suboptimal search like beam search to be bounded-suboptimal. Both are conceptually straightforward: the first follows the architecture of focal search, with a beam search that is limited to nodes within the bound, and the second adapts an anytime beam search. The first approach can be used to make any search bounded-suboptimal, while the second can be used to make any anytime search bounded-suboptimal.

Bounded-Suboptimal Beam Search

Conceptually, our approach to bounded-suboptimal beam search can be seen as a variant of focal search, where the focal list is a beam with only some of the qualifying nodes, rather than all. TOur first approach to bounded-suboptimal beam search uses Bead, a performant unboundedly suboptimal search, to try to find a solution within $w \times f_{\min}$, and if that fails we start a new Bead search from a $\min_f$ frontier node. This explores the state space within a focal style envelope of $f(n) \leq w \times f_{\min}$ nodes, but that exploration comes in the form of successive beams to the edge of the envelope, rather than in $g(n) + w \times h(n)$ layers.

The pseudocode for Bounded-Suboptimal Beam Search (BSBS) is shown in Algorithm 3. In addition to the beam, it uses an open list ordered on f to allow for quickly querying $f_{\min}$ and a corresponding node. Every node that is generated is placed on open (Line 23), but only those that are guaranteed to meet the suboptimality bound are placed onto the beam (Line 15-24). If there are no candidates for the beam, then a new beam search is started from a $\min_f$ node (Line 25-26). All nodes are expanded as part of a (suboptimality-bounded) beam search (Line 7). When a node is expanded it is also removed from open (Line 9). As with beam search, the priority ordering used to select which candidates are put on the beam can be anything (Line 27); unless otherwise noted we use $\hat{d}$.

Algorithm 3: Bounded-Suboptimal Beam Search

```
1: function BSBS(s_i, succ(s), c(s, s'), isG(s), w, width)
2: n ← {s : s_i, g : 0}
3: open ← {n}
4: beam ← {n}
5: while True do
6:     cand ← {}                                   ▷ Beam Candidates
7:     for n ∈ beam do
8:         if isG(n) then return Success
9:         open ← open \ {n}
10:        for s' ∈ succ(s(n)) do
11:            g' ← g(n) + c(s, s')
12:            f_min ← f(top(open))
13:            if s' ∈ cl ∧ g' < g(cl[s']) then
14:                n' ← {s : s', g : g'}
15:                if f(n') ≤ w × f_min then
16:                    cand ← (cand \ cl[s']) ∪ {n'}
17:                open ← (open \ cl[s']) ∪ {n'}
18:                cl[s'] ← n'
19:            else if s' ∉ cl then
20:                n' ← {s : s', g : g'}
21:                if f(n') ≤ w × f_min then
22:                    cand ← cand ∪ {n'}
23:                open ← open ∪ {n'}
24:                cl[s'] ← n'
25:    if |cand| = 0 then
26:        cand ← {top(open)}
27:    beam ← best width from cand   ▷ (e.g., ↓ f or ↓ d̂)
```

Properties of BSBS

While Bounded-Suboptimal Beam Search is motivated by the empirical performance of Bead, for the purpose of theoretical analysis it is easiest to view it as a form of focal search. Like a focal list, the beam never contains a node that violates the suboptimality bound. In this subsection, we prove two desirable properties of Bounded-Suboptimal Beam Search: 1) it is complete, and 2) it is bounded-suboptimal.

We will make the following assumptions:

A1 the state space is finite,

A2 costs are positive, and

A3 h is admissible.

For completeness there are two concerns: first, that no plans are incorrectly pruned:

Lemma 1. *Always (prior to Algorithm 3, Line 8) for every possible plan p that could reach a goal, there is a node n_e on open corresponding to a plan p_e that could also reach that goal and plan p_e has cost less than or equal to the cost of p.*

Proof. If there is a node corresponding to p on open, then select it as n_e (a node corresponds to a plan p if it represents p, or a sub-plan of p that starts at the same state as p). Otherwise, every node that is generated is placed on open, and a node n corresponding to p will only not be generated for a successor s' if it is dominated (including tie-breaking) by an

existing node n_e corresponding to the same state s' (Algorithm 3, Line 13). The plan p_e represented by n_e will have cost less than or equal to the plan represented by n that was not generated because the partial plan to reach s' has the same or less cost, and they share the same remaining plan after s. □

Second, any node on open will eventually be expanded by a beam search:

Lemma 2. *In finite time, any node n on open will be expanded by a beam search, unless a goal is expanded first.*

Proof. The state space is finite, and there are not negative cost cycles (A2), so there are a finite number of plans, with corresponding search node n_p, that have $f(n_p) \leq w \times f(n)$ (A1, A2). Because n is on open, $f_{\min} \leq f(n)$, and only nodes with $f(n_p) \leq w \times f_{\min} \leq w \times f(n)$ can be expanded prior to n (see Algorithm 3, Line 15). Because there are a finite number of such nodes (A1, A2), n will eventually be expanded (unless a goal is expanded first and the search terminates) because it is the only candidate. □

Therefore,

Theorem 3. *Bounded-Suboptimal Beam search is complete.*

Proof. In order to be incomplete, the search would either need: 1) not to generate a node corresponding to a plan p that could reach the goal, or 2) to generate but never expand p. By Lemma 1 either p will be generated, or a plan at least as good as p will be. Consider whichever is generated to be p: unless the search terminates earlier by finding a goal (Algorithm 3, Line 8), p will eventually be expanded (Lemma 2). □

Theorem 4. *Bounded-Suboptimal Beam search is bounded-suboptimal.*

Proof. Bounded-Suboptimal Beam search is complete, so if a solution exists it will eventually return one, and because the state space is finite if there is no solution it will eventually terminate (Theorem 3). Call the cost of an optimal solution C^*, h is admissible, so always $w \times f_{\min} \leq C^*$. No node is expanded, except as part of a beam search, and no node n' is placed on the beam unless it has $f(n') \leq w \times f_{\min} \leq C^*$ so $f(n') \leq C^*$. □

Unlike Weighted A*, and like RR-d and EES, bounded-suboptimality is not necessarily retained if duplicates are dropped. Doing so in a focal-style search can violate $w \times f_{\min} \leq C^*$ because the lower cost duplicate of the $\min_f$ node may have been dropped.

Bounded-Suboptimal Rectangle Search

We now turn to our second approach. The approach we use to make Rectangle bounded-suboptimal would also work with any other anytime approach. Whereas termination in the anytime setting is by definition unknown at runtime, we instead terminate only when a solution guaranteed to meet the suboptimality bound has been found. Applying this method to Rectangle results in an algorithm we call Bounded-Suboptimal Rectangle Search (BSOR). BSOR requires efficient access to the least f-value on the search frontier, so we maintain an A*-style priority queue of open nodes along with the rectangle. This makes it easy to interleave $\min_f$ expansions with the rectangle expansions, an additional variant we call Round-Robin Rectangle Search (RRR). Like the $\min_f$ expansions in RR-d, the alternating $\min_f$ expansions in RRR force the search to devote half its effort to raising the estimate of the suboptimality bound.

Algorithm 4: Bounded Suboptimal Rectangle Search

```
1:  function BSOR(s_i, succ(s), isG(s), c(s,s), a, w, rr)
2:    n_i ← {s : s_i, de : 0, g : 0}, cl ← {s_i ↦ n_i}
3:    rect ← {0 ↦ {n}}, ec ← {0 ↦ 0}, open ← {n_i}
4:    (δ↓, δ↔) ← (a, 1) if a ≥ 1 else (1, 1/a)
5:    it ← 1, l ← 0, σ ← ∅
6:    while σ = ∅ ∨ w × f(peek(open)) < g(σ) do
7:      if rr then
8:        repeat
9:          if open = ∅ then return σ
10:         n ← extract a lowest-f node from open
11:         extract n from rect[de(n)]
12:       until σ = ∅ ∨ f(n) < g(σ)
13:       EXPAND(n)                          ▷ Algorithm 5
14:     repeat
15:       if ¬NEXT(rect, ec, δ↓, δ↔, i, l) then return σ
16:       n ← extract a lowest-d node from rect[l]
17:       extract n from o
18:     until σ = ∅ ∨ f(n) < g(σ)
19:     if l ≥ (it − 1) × δ↓ then
20:       rect ← rect ∪ {l+1 ↦ ∅}, ec ← ec ∪ {l+1 ↦ 0}
21:     ec[l] ← ec[l] + 1
22:     EXPAND(n)                            ▷ Algorithm 5
23:   return σ
24: function NEXT(rect, ec, δ↓, δ↔, i, l)
25:   loop
26:     if rect[l] ≠ ∅ ∧ ec[l] < i × δ↔ then return true
27:     if l < i × δ↓ − 1 then l ← l + 1
28:     else if rect has a non-empty level then i ← i+1, l ←
        0
29:     else return false
```

Algorithm 4 shows pseudocode for BSOR and RRR, which are selected by false and true values of the rr toggle, respectively. Both BSOR and RRR maintain the A*-style open list $open$; RRR selects from $open$ for every other expansion. In order to iterate over the rectangle in the same order as Rectangle Search, which benefits from the simplicity of nested loops, RRR by contrast requires the rectangle expansion order to be interruptible, so iteration management is offloaded to the NEXT function (Line 24). In short, for each rectangle expansion, NEXT controls whether BSOR and RRR select from the current depth l, the next depth, or start a new iteration by selecting from the top at $l = 0$; if the rectangle is empty, NEXT signals for the search to terminate (Lines 28–29). RRR also terminates when $open$ is found to be empty (Line 9); the invariant that a node is either in both $open$ and $rect$ or in neither, never just in one or the other, preserves completeness.

Both BSOR and RRR prune selected and generated nodes

Algorithm 5: BSOR: Successor Expansion

```
1: function EXPAND(n)
2:   for s' ∈ succ(s(n)) do
3:     n' ← {s : s', de : de(n) + 1, g : g(n) + c(s(n), s')}
4:     if σ ≠ ∅ ∧ f(n') ≥ g(σ) then continue
5:     if isG(s') then σ ← n', report σ, continue
6:     pre ← cl[s']
7:     if pre = ∅ then
8:       cl[s'] ← n', open ← open ∪ {n'}
9:       rect[de(n')] ← rect[de(n')] ∪ {n'}
10:    else if g(n') < g(pre) then
11:      g(pre) ← g(n')
12:      de_pre ← de(pre), de(pre) ← de(n')
13:      if pre ∉ open then
14:        open ← open ∪ {pre}
15:        rect[de(pre)] ← rect[de(pre)] ∪ {pre}
16:      else
17:        repair open
18:        if de_pre ≠ de(pre) then
19:          extract pre from rect[de_pre]
20:          rect[de(pre)] ← rect[de(pre)] ∪ {pre}
21:        else repair rect[de(pre)]
```

that cannot improve on the incumbent solution σ (Algorithm 4 Lines 12,18, and Algorithm 5 Line 4). Like Rectangle, BSOR and RRR lazily create the next depth level immediately before it is needed (Lines 19–20). Like Rectangle, BSOR and RRR prune duplicate states without a lower g-value (Algorithm 5 Line 10).

Properties of BSOR and RRR

Unlike most bounded-suboptimal searches, the breadth-first nature of BSOR and RRR may lead them to expand nodes that it has not proven to be within the suboptimality bound (for another example of such optimism see Thayer and Ruml (2008)). Essentially, where other searches must first prove the bound, then find a corresponding solution (such as Weighted A*), BSOR and RRR can find a solution first and then prove its bounded suboptimality by raising $f_{\min}$; BSOR and RRR thus cannot be viewed as focal searches. In this subsection, we prove two desirable properties of BSOR and RRR: 1) they are complete, and 2) they are bounded-suboptimal. We make the same assumptions A1-A3 (finite state space, positive costs, admissible h) as for BSBS.

Lemma 5. *BSOR and RRR prune a node only when it is dominated by another node corresponding to the same state, meaning an equal or better partial solution remains on the open list open and rectangle rect, or when it cannot possibly lead to a better solution than the incumbent.*

Proof. The proof follows from Algorithm 4 Lines 8–12 and 14–18 and Algorithm 5 Lines 4 and 10. □

Lemma 6. *BSOR and RRR terminate in finite time.*

Proof. Finite work is performed in each iteration of BSOR/RRR. All explored paths will be acyclic because edge costs are positive (A2) and duplicate detection is performed with cl and duplicates without lower g values are pruned. Because the state space is finite (A1), only a finite number of acyclic paths exist to any given state, so each state will be reopened a finite number of times. Therefore, Algorithm 4 Lines 8–12 and 14–18 and Algorithm 5 Line 2 will always take finite time to complete. Since cycles are prevented and the state space is finite, the paths along which BSOR and RRR expand states are necessarily of finite length (regardless of whether those paths are solutions), so the loop in Algorithm 4 Line 6 will exit after finite time. □

Theorem 7. *BSOR and RRR are complete.*

Proof. For BSOR/RRR to be incomplete, it would need either to run forever, which contradicts Lemma 6, or to prune all existing solutions, which contradicts Lemma 5. In particular, since the existence of a solution implies the existence of a bounded suboptimal solution, pruning all solutions means that at some point BSOR or RRR pruned at least one node n along every solution p. Pruning only occurs when $f(n) \geq g(\sigma)$, however, which contradicts the assumption that all solutions could be pruned since the incumbent is already non-null. □

Theorem 8. *BSOR and RRR are bounded suboptimal.*

Proof. By Theorem 7, if a solution exists, BSOR/RRR will return a solution when it terminates. When BSOR/RRR has an incumbent solution and terminates, either the open list $open$ and rectangle $rect$ are 1) non-empty or 2) empty.

Case 1) In this case, $w \times f(peek(open)) < g(\sigma)$ must be false, and by A3 we have $w \times f_{\min} \leq w \times C^*$. Therefore we have $g(\sigma) \leq w \times f(peek(open)) \leq w \times C*$, so the solution returned is within the suboptimality bound.

Case 2) When at least one solution exists, then at least one bounded suboptimal solution must also exist; call one such solution p. From initialization, $open$ and $rect$ must contain at least one node corresponding to a state s along p, since if a bounded suboptimal (partial-)solution were pruned, it could only be in favor of a better and thus still boundedly suboptimal (partial-)solution; let p refer to such a preferred solution. If $open$ and $rect$ are empty at termination, then p must have already been found and stored in σ. Otherwise, at some earlier point BSOR or RRR must have pruned p, but that would either require that a better incumbent had been identified (in which case call it p) or it would contradict Lemma 5. □

Experimental Evaluation

To evaluate our algorithms we compare their performance on the Korf 100 classic sliding tiles search benchmark (Korf 1985). The experiments were run serially on machines with i3-12100 CPUs, each with a time budget of 5 minutes and memory budget of 60 GB. With this state space, we used three action cost models: unit, heavy (moving tile i costs i), and inverse cost ($1/i$), using the appropriate weighting of the Manhattan distance heuristic. The different costs vary the local minima in the state space, changing the behavior of the algorithms. For comparison, we have included Weighted A*, RR-d, and EES. Algorithms were written in either C++ or

Rust[2]. Despite similar levels of overhead and optimization, when comparing algorithms between codebases we refrain from CPU time comparisons and concentrate on node expansions, which are comparable between implementations.

As noted by Hami and Sturtevant (2025), the original domain-specific implementation for RR-d contained a bug that prevented the search from ever expanding from the $\hat{f}$ queue. Our RR-d code is newly written and, as an additional contribution of this paper, we compare RR-d to variants that entirely remove the $\hat{f}$ and d queues, respectively. Due to space limitations, in several places we present only a representative plot; unless otherwise noted, the other experiments found similar results.

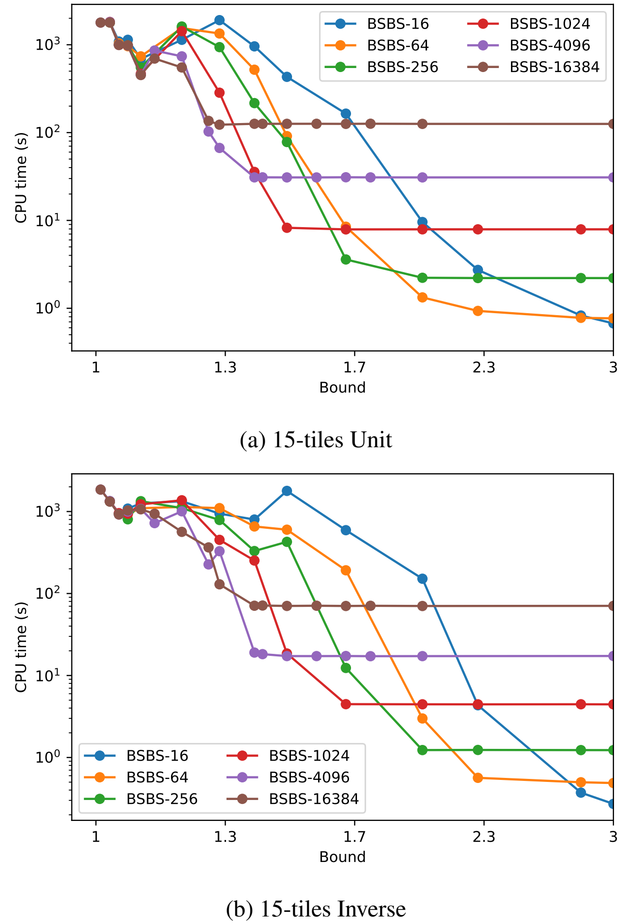
The Effect of Beam Width on BSBS

We begin by exploring the performance of BSBS as the width of the beam is changed. There are two regimes when considering the performance of BSBS. The first is where the suboptimality is easy enough to satisfy that it can be solved by a single (or perhaps a small number) of beam searches in BSBS. In this regime, we would expect any additional beam width beyond that needed to solve the problem to lead to a proportional increase in the runtime. We will call this first regime the loose-bound regime, because a sufficiently loose sub-optimality will make it more likely for this to occur. The second regime, which we will call the tight-bound regime, considers the remaining bounds, where tuning the beam width trades off between the number of beams and the size of those beams.

Figure 1 shows the runtime of BSBS variants with a variety of widths and bounds. The loose-bound regime is seen as a runtime plateau beginning at about suboptimality bound 1.5 for width 16384 and suboptimality bound 3.0 for width 16. The runtime at these plateaus approaches the runtimes of the corresponding (unboundedly suboptimal) Bead search, with some overhead caused by BSBS maintaining the open list. These results show that, as would be expected, for high suboptimality bounds (> 1.5), the looser the suboptimality bound, the skinnier the beam can be. In contrast, for the tightest bounds (< 1.1), the runtime performance is not sensitive to the width of the beam. However, in the intermediate zone, $(1.1 - 1.5)$, wider beams can result in faster runtimes. In general, wide beams (e.g., 16384) are dominated on almost all suboptimality bounds by more moderate beam widths (e.g., 4096). BSBS with smaller beam widths shows a non-monotonic relationship between suboptimality bound and runtime for intermediate bounds (1.1-1.5) for some cost models (e.g., unit tiles).

Variations of BSBS

In this subsection, we explore the empirical performance of two additional variants of BSBS. In BSBS (Algorithm 3) when the beam search dies, a new beam search is started by expanding a single $\min_f$ node. We define two alternate variants of BSBS with slightly different behavior. First, BSBS-fill instead expands $width$ successive $\min_f$ nodes, effectively starting $width$ beam searches that compete for the $width$ spots on the beam. Second, we define BSBS-f-layer, which attempts to raise $f_{\min}$ before restarting the beam. It does this by expanding $\min_f$ nodes until a higher f node is expanded. Qualitatively, this approach performed extremely poorly in our initial testing (perhaps because it would expand the entire final f-layer), so we instead evaluate a variant that is constrained to only do up to $width$ $\min_f$ expansions, then restarts the beam from a $\min_f$ node. This approach is different from BSBS-fill because the children of the $width$ $\min_f$ expansions are not placed on the beam (though they could be the starting state of a later beam).



(a) 15-tiles Unit

(b) 15-tiles Inverse

Figure 1: BSBS Beam Widths

Figure 2 shows the runtime results for these BSBS variants. We selected a width of 1024 for Figure 2 as an example because BSBS with width 1024 performed well for a wide range of suboptimality bounds, however, we tested the full range of widths for all three variants and found a similar performance pattern for all widths. All three perform similarly, so we will use BSBS for the remaining comparisons as it is the baseline approach.

[2] github.com/dwthomas/bsbs and github.com/sjwo/2026-hsdip-bsor.

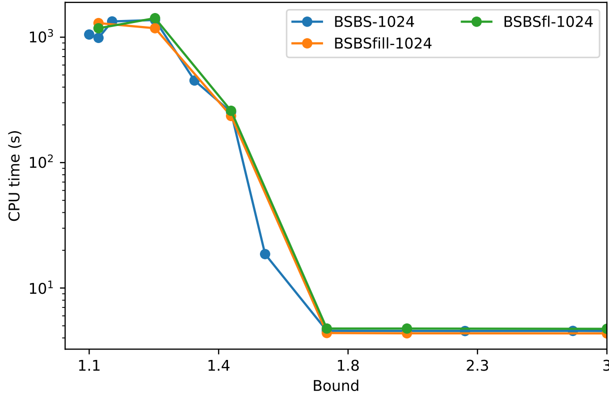


Figure 2: BSBS variants, 15-tile Inverse

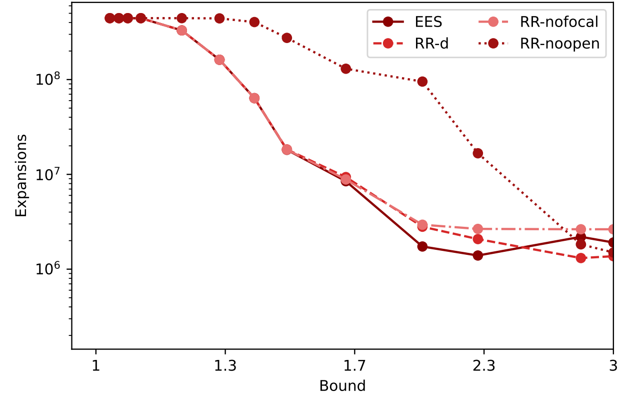
Comparison of EES and RR-d variants

We now conduct an ablation study of the three viable RR-d variants. RR-d uses three queues, one ordered on d, one on $\hat{f}$, and one ordered on f. The f queue is required to compute $\min_f$, so we have tested the performance of two additional variants of RR-d. RR-d-no-focal skips the focal (d) list, while RR-d-no-open skips the open ($\hat{f}$) list. RR-d-no-open is similar to the buggy original domain-specific implementation of RR-d, though our implementation entirely gets rid of focal/open, rather than just not expanding from them, which potentially decreases the overhead by maintaining one less queue.

Figure 3 shows the expansion and runtime performance of the three RR-d variants along with EES. All three variants expand about the same number of nodes for tight suboptimality bounds ($w < 1.1$), and RR-d and RR-nofocal expand a similar number until $w \approx 2$. In runtime, the variants benefit from needing to maintain only two queues. In tight-suboptimality bound problems, we find that RR-d-no-open can perform better, likely because the challenge of the search is in proving the suboptimality of the solution, rather than finding a satisficing solution. On the other hand, RR-d and RR-d-no-focal perform similarly, with RR-d-no-focal performing slightly better with intermediate suboptimality bounds. EES performs very similarly to RR-d and RR-d-no-focal, performing better for two bounds, but worse for the least-tight bounds.

Comparison of BSO Rectangle Algorithms

Figure 4 shows the expansion performance of BSOR and RRR. The aspect ratios 1 and 500 were selected to match Lemons et al. (2024). RRR performs better for suboptimality bounds up to 1.7, then BSOR performs better. In both cases, the aspect ratio 1 outperformed 500, in contrast with the empirical performance of rectangle in anytime search, where 500 performs better (Lemons et al. 2024). The forced raising of $f_{\min}$ in RRR appears to help the most for low (but not so low that you would be better off doing A*) suboptimality bounds.



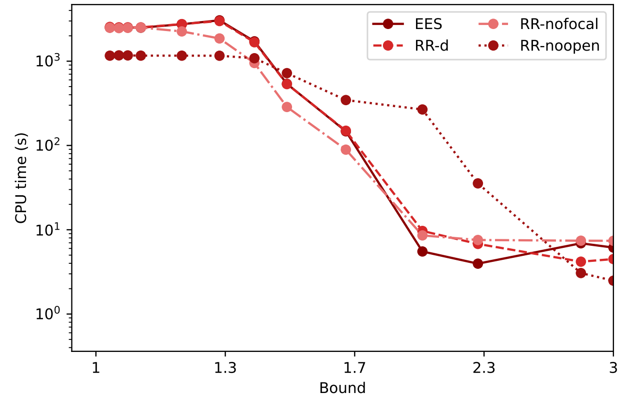


Figure 3: RR-d variants, 15-tile Inverse

Comparison of State of the Art Algorithms

Figure 5 shows the runtime performance of BSBS, BSOR, and RRR in comparison to Weighted A* and RR-d-no-focal. For all three cost models the general trend is for RRR-1 to dominate for tight bounds, then BSOR-1, then BSBS-16 for loose bounds. On unit and heavy BSBS-64 also has a window of dominance. RRR-1 individually dominated both Weighted A* and RR-d-no-focal.

Table 1 quantifies the statistical significance of a subset of these results. We use the nonparametric Wilcoxon signed-rank test (Wilcoxon 1945) with the null hypothesis that the difference in the log-expansions (or log-runtime) is non-negative, the logarithm is used because the Wilcoxn test assumes the difference is symmetric, and search expansions (and runtime) are exponential in solution depth. For example, BSBS-64 $<$ WA*, $\geq$ 1.5, UHI signifies that we are quantifying if BSBS-64 was faster than Weighted A* for the tested suboptimality bounds $w \geq 1.5$ with unit, heavy, or inverse cost. Because we perform multiple comparisons, we adjusted the p-values using the Bonferroni correction. However, post correction all reported p-values remained orders of magnitude less than 0.001. We quantified only exactly the pre-selected subset of hypotheses shown in Table 1, and exclusion should not be construed to imply that a result would not be significant if it had been tested.

Considering RRR-1, BSOR-1 and BSBS-16, we find that:

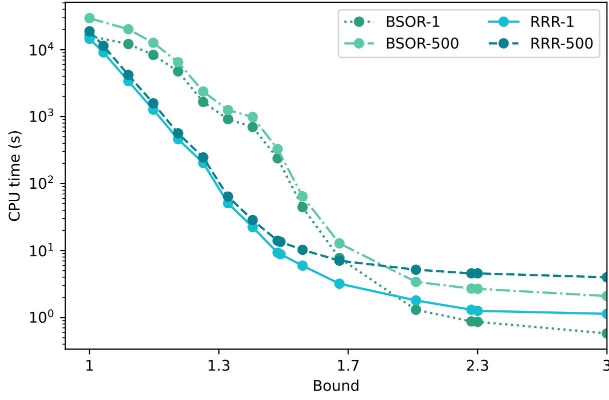


Figure 4: BSOR Algorithms Compared

Hypothesis	Bounds	Costs	Adj. p
RRR-1 $<$ BSOR-1	≤ 1.3	HI	< 0.001
BSOR-1 $<$ RRR-1	≥ 1.7	UHI	< 0.001
BSBS-16 $<$ BSOR-1	≥ 2.25	HI	< 0.001
BSBS-64 $<$ WA*	≥ 1.5	UHI	< 0.001
BSBS-64 $<$ RR-nofocal	≥ 1.7	UHI	< 0.001
BSOR-1 $<$ WA*	≥ 1.1	UHI	< 0.001
BSOR-1 $<$ RR-nofocal	≥ 1.0	UHI	< 0.001
RRR-1 $<$ WA*	≥ 1.09	UHI	< 0.001
RRR-1 $<$ RR-nofocal	≥ 1.0	UHI	< 0.001

Table 1: Statistical Results

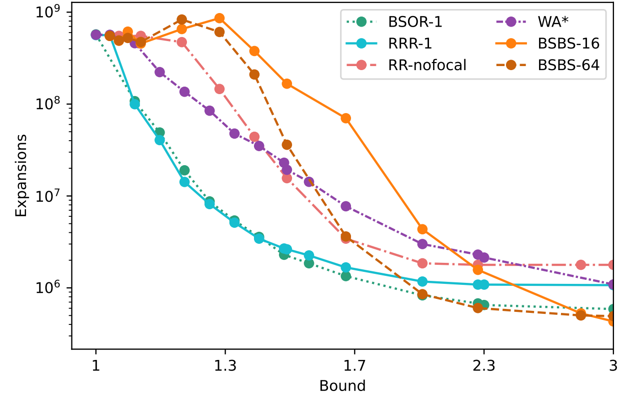
1) RRR-1 is significantly faster than BSOR-1 for $w \leq 1.3$ with heavy and inverse costs, 2) BSOR-1 is significantly faster than RRR-1 for $w \geq 1.7$ for all three costs tested, and 3) BSBS-16 expand significantly fewer nodes than BSOR-1 (and therefore RRR-1) for $w \geq 2.7$ with heavy and inverse costs. These approaches are significantly faster than Weighted A* and RR-nofocal. RRR-1 and BSOR-1 are both faster than Weighted A* and expands significantly fewer nodes than RR-nofocal in all tested settings except when $w = 1$. Similarly, BSBS-64 is significantly faster than Weighted A* and RR-nofocal for looser bounds. Therefore, RRR-1, BSOR-1 and BSBS-16 significantly outperform Weighted A* and RR-nofocal for the non-optimal subset of bounded-suboptimal search, representing a new state of the art (for the 15-Puzzle).

Discussion

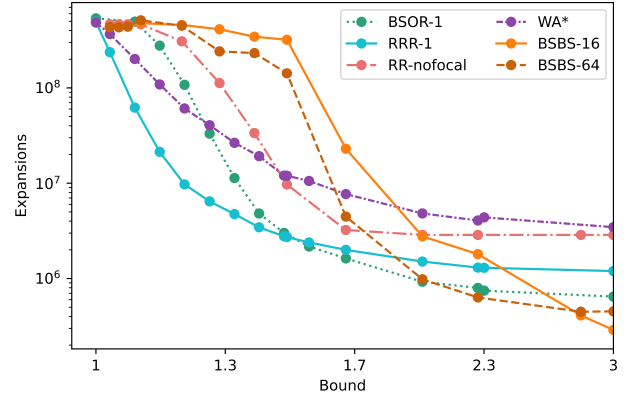
We find that BSOR-1, RRR-1, and BSBS singificantly outperform other approaches in this domain.

The chief limitations of BSBS as a bounded-suboptimal heuristic search are the dependence on beam width as an additional parameter, and the sometimes (relatively) poor and non-monotonic performance of BSBS in the tight regime. Similarly, BSOR and RRR require the user to set the aspect ratio parameter.

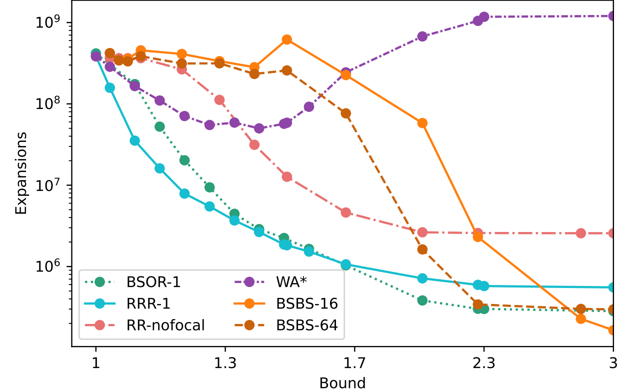
In future work, we will consider additional benchmark domains, including those that feature dead-ends, which can be difficult for beam-based approaches (Wissow, Yu, and Ruml 2024). We will also include additional comparators, such as DPS (Gilon, Felner, and Stern 2016) and Ψ_{XDP} (Chen and Sturtevant 2019).



(a) 15 Tile, Unit



(b) 15 Tile, Heavy



(c) 15 Tile, Inverse

Figure 5: Non-dominated Algorithms Compared

Conclusions

BSBS, BSOR, and RRR are, to our knowledge, the first non-best-first heuristic search algorithms explicitly designed for the bounded-suboptimal problem setting. BSBS is the simplest example of a new class of bounded-suboptimal methods based on beam search, while BSOR is a straightforward adaptation of Rectangle Search from an anytime approach to a bounded-suboptimal search. RRR interleaves $f_{\min}$ expansions into BSOR, which improves performance on tight suboptimality bounds by forcing the search to raise $f_{\min}$.

We proved that all three algorithms are complete and bounded-suboptimal. We found empirically that together these three approaches significantly outperform existing approaches for a range of suboptimality bounds. For tight bounds ($w < 1.6$), RRR-1 performs the best, for intermediate bounds ($1.6 \leq w < 2$) BSOR performs the best, and for loose suboptimality bounds ($w \geq 2$) BSBS performs the best. Weighted A* (effectively A*) remains the best for very tight bounds ($w \simeq 1$).

We hope this work stimulates investigation of non-best-first approaches to the many different suboptimal heuristic search problem settings.

References

Bisiani, R. 1987. Beam search. In Shapiro, S. C., ed., *Encyclopedia of Artificial Intelligence*, 56–58. John Wiley and Sons.

Bulitko, V.; Björnsson, Y.; Sturtevant, N. R.; and Lawrence, R. 2011. Real-time heuristic search for pathfinding in video games. In *Artificial Intelligence for Computer Games*, 1–30. Springer.

Chen, J.; and Sturtevant, N. R. 2019. Conditions for Avoiding Node Re-expansions in Bounded Suboptimal Search. *International Joint Conference on Artificial Intelligence (IJCAI)*.

Dolgov, D.; Thrun, S.; Montemerlo, M.; and Diebel, J. 2008. Practical search techniques in path planning for autonomous driving. *AAAI Search Workshop*, 1001(48105): 18–80.

Fickert, M.; Gu, T.; and Ruml, W. 2022. New Results in Bounded-Suboptimal Search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, 10166–10173.

Gilon, D.; Felner, A.; and Stern, R. 2016. Dynamic Potential Search — A New Bounded Suboptimal Search. In *Proceedings of the Ninth Annual Symposium on Combinatorial Search (SoCS-16)*.

Hami, M.; and Sturtevant, N. R. 2025. Suboptimal search with dynamic distribution of suboptimality. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 26991–26999.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107.

Helmert, M.; and Röger, G. 2008. How good is almost perfect? In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI'08, 944–949. AAAI Press. ISBN 9781577353683.

Holte, R.; Majadas, R.; Pozanco, A.; and Borrajo, D. 2019. Error analysis and correction for weighted A*'s suboptimality. In *Proceedings of the International Symposium on Combinatorial Search*, volume 10, 135–139.

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

Lemons, S.; López, C. L.; Holte, R. C.; and Ruml, W. 2022. Beam search: faster and monotonic. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 32, 222–230.

Lemons, S.; Ruml, W.; Holte, R.; and López, C. L. 2024. Rectangle Search: An Anytime Beam Search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 20751–20758.

Lemons, S.; Ruml, W.; López, C. L.; and Holte, R. 2023. Triangle search: An anytime beam search. In *ICAPS 2023 Heuristics and Search for Domain-Independent Planning Workshop*.

Mudgal, A.; Tovey, C.; Greenberg, S.; and Koenig, S. 2005. Bounds on the travel cost of a mars rover prototype search heuristic. *SIAM Journal on Discrete Mathematics*, 19(2): 431–447.

Newell, A. 1980. Harpy, production systems, and human cognition. *Perception and production of fluent speech*, 289–380.

Pearl, J.; and Kim, J. H. 1982. Studies in Semi-Admissible Heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4(4): 392–399.

Pohl, I. 1973. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, 12–17. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Thayer, J. T.; and Ruml, W. 2008. Faster than Weighted A*: An Optimistic Approach to Bounded Suboptimal Search. In *ICAPS*, 355–362.

Thayer, J. T.; and Ruml, W. 2011. Bounded suboptimal search: A direct approach using inadmissible estimates. In *IJCAI*, volume 2011, 674–679.

Thayer, J. T.; Ruml, W.; and Kreis, J. 2009. Using distance estimates in heuristic search: A re-evaluation. In *Symposium on Combinatorial Search*. Citeseer.

Wilcoxon, F. 1945. Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6): 80–83.

Wilt, C.; and Ruml, W. 2021. When Does Weighted A* Fail? *Proceedings of the International Symposium on Combinatorial Search*, 3(1): 137–144.

Wissow, S.; Yu, F.; and Ruml, W. 2024. Tunable suboptimal heuristic search. In *Proceedings of the International Symposium on Combinatorial Search*, volume 17, 170–178.

Wurman, P. R.; D'Andrea, R.; and Mountz, M. 2008. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI magazine*, 29(1): 9–9.