Tunable Suboptimal Heuristic Search

Stephen Wissow[1], Fanhao Yu[2], Wheeler Ruml[1]

[1] University of New Hampshire, Durham, NH, USA
[2] Nashua High School South, Nashua, NH, USA
sjw@cs.unh.edu, yufanhao12@gmail.com, ruml@cs.unh.edu

Abstract

Finding optimal solutions to state-space search problems often takes too long, even when using A* with a heuristic function. Instead, practitioners often use a tunable approach, such as weighted A*, that allows them to adjust a trade-off between search time and solution cost until the search is sufficiently fast for the intended application. In this paper, we study algorithms for this problem setting, which we call 'tunable suboptimal search'. We introduce a simple baseline, called Speed*, that uses distance-to-go information to speed up search. Experimental results on standard search benchmarks suggest that 1) bounded-suboptimal searches suffer overhead due to enforcing a suboptimality bound, 2) beam searches can perform well, but fare poorly in domains with dead-ends, and 3) Speed* provides robust overall performance.

Introduction

A wide variety of intractable planning problems can be formulated as state-space search problems, so it is no surprise that many search problems take too long to solve optimally, even when using an optimally efficient algorithm such as A* (Hart, Nilsson, and Raphael 1968; Dechter and Pearl 1988). A* is a best-first search that orders its search frontier on increasing $f(n) = g(n) + h(n)$, where $g(n)$ represents the cost of the path to node n from the root search node, corresponding to the problem's initial state, and $h(n)$ represents a heuristic estimate of the remaining cost-to-go from n to a node corresponding to a state satisfying the problem's goal predicate $goal(s)$. While optimal solutions are of course always preferred, when they are infeasible to compute many practitioners use methods that relax A*'s solution optimality guarantee in favor of reduced computation time, methods that we refer to broadly as *suboptimal*. At the opposite extreme from A* is the *agile* (also referred to as satisficing or pure suboptimal) search algorithm, whose objective is to find any solution at all as fast as possible. However, what is often desired is a *tunable* suboptimal algorithm, which allows the practitioner to adjust the trade-off between solution cost and search time, ideally spanning from optimal to as-fast-as-possible.

Weighted A* (wA*) is a popular best-first search that orders its search frontier on increasing $f'(n) = g(n) + w \cdot h(n) : w \geq 1$ (Pohl 1970). (In the implementation tested below, we break ties in favor of low h.) wA* is a *bounded suboptimal* algorithm, because it guarantees that the solution it returns will have a cost that is within a factor w of the cost C^* of an optimal solution. Several other bounded suboptimal algorithms have been proposed, including A^*_ϵ (Pearl and Kim 1982), Explicit Estimation Search (Thayer and Ruml 2011, EES), Dynamic Potential Search (Gilon, Felner, and Stern 2016, DPS), and Round-Robin-d (Fickert, Gu, and Ruml 2022, RR-d). Researchers have also proposed *bounded cost* search algorithms, such as Potential Search (Stern, Puzis, and Felner 2011), that guarantee returning a solution whose cost is less than a provided budget or failing if no such solution exists.

However, using bounded algorithms for tunable suboptimal search, where we don't really care about guaranteed cost but just want the best empirical performance, raises two issues. The first is that many bounded search algorithms, such as wA* and DPS, focus on $g(n)$ cost-so-far and $h(n)$ cost-to-go to guide their search, and they converge in the limit of large bounds to Greedy Search (also known as Greedy Best-First Search, GBFS) (Michie and Ross 1969), which is a best-first search that orders its search frontier on increasing $h(n)$, breaking ties on low $g(n)$. It is well-known that, for problems that feature actions with different costs (also known as 'non-unit costs'), guiding search on $d(n)$ distance-to-go (in terms of number of state-space transitions, or equivalently number of action applications or number of arcs in the state-space graph) can be much faster (Thayer, Ruml, and Kreis 2009). While some bounded-suboptimal algorithms, such as EES and RR-d, use $d(n)$ in concert with $h(n)$ to guide their search and do converge in the limit of large bounds to Speedy Search, a best-first search on increasing $d(n)$ that breaks ties on low h (Ruml and Do 2007), they often have high overheads from maintaining multiple orderings of the open list and coordinating between them in order to enforce the suboptimality bound. We are not aware of a previously-proposed simple, low-overhead tunable algorithm that can span the spectrum between A* and Speedy.

Second, while the bounded suboptimal and bounded-cost search settings have received attention, recent work has not explicitly addressed the problem setting of tunable suboptimal search—even though, in practice, this is the problem setting to which algorithms like wA* are often applied. It

Copyright © 2024, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

is unclear whether methods might exist that find solutions of similar (or lower) cost in similar (or less) time if they do not need to enforce bounds. In other words, previous work has not evaluated 'the price of bounded suboptimality' in heuristic search. Do these guarantees come with performance overheads than can be avoided when the guarantees are not needed in practice? How might they fare empirically against approaches like beam search (Bisiani 1987) that do not guarantee bounds?

In this paper, we explicitly study algorithms for the tunable suboptimal search setting. We present a simple and complete tunable suboptimal algorithm, Speed*, that offers performance trade-offs spanning from A* to Speedy. We find through experimental evaluation that Speed* often outperforms not just wA* but also RR-d, the state-of-the-art bounded suboptimal algorithm. We also see that Speed* is robust to domains with dead-ends, unlike beam-based search approaches such as Bead (Lemons et al. 2022) and Rectangle search (Lemons et al. 2024). This work is the first to evaluate RR-d and Rectangle in the tunable suboptimal setting. The results suggest that Bead search offers the best trade-off between cost vs. time for domains without dead-ends and that Speed* is the preferred approach where robustness to dead-ends is required. We hope this work draws researchers' attention to the tunable suboptimal search setting, since it often seems to be what practitioners really want.

Previous Work

Greedy and Speedy are agile heuristic search algorithms designed to find any (unboundedly suboptimal) solution as quickly as possible. Greedy search can be seen as the limiting case of wA* as w is increased. Speedy search often finds solutions more quickly than Greedy in non-unit-cost settings.

Many bounded suboptimal methods have been proposed, starting with wA*, and including EES and DPS. The current state-of-the-art is RR-d (Fickert, Gu, and Ruml 2022). In addition to a 'cleanup' list that orders the entire search frontier on increasing $f(n)$, RR-d maintains two additional queues, 'open' ordered on increasing $\hat{f}(n)$, and 'focal' ordered on increasing $d(n)$, that contain a subset of nodes from the cleanup list: $\{n : f(n) \leq w \cdot f_{min}\} \subseteq \text{cleanup}$, where f_{min} is the f-value of the node at the front of the cleanup list. $\hat{f}$ is a potentially inadmissible 'best estimate' of f^*. In the implementation tested below, $\hat{f}$ is computed from f using online error correction and a global error model, following Thayer, Dionne, and Ruml (2011). To select the next node to expand, RR-d alternates among its three queues in a round-robin fashion. With an admissible heuristic, and since g increases over the course of the search, f_{min} will never exceed the optimal solution cost, thus guaranteeing a goal selected from any of the three queues will always have a cost within the suboptimality bound. The intuition behind the different orderings of the three queues is that (1) a node expansion from the cleanup (f-ordered) list raises f_{min}, admitting additional nodes to the other queues, (2) a node expansion from the open ($\hat{f}$-ordered) list makes progress toward low-cost solutions (and enqueues them), and (3) a node expansion from the focal (d-ordered) list pursues nearby solutions that can be found quickly. In their empirical evaluation, RR-d performed better than other queue alternation schemes that Fickert, Gu, and Ruml (2022) tested, so we do not compare against them.

Beam search is a tunable search that does not offer a bound. It can be understood as a variant of breadth-first search that also proceeds by depth layer. Instead of expanding every node at each depth like bread-first search, beam search selects for expansion only a constant number of nodes k at each depth layer; k is referred to as the beam's 'width'. Any nodes not selected for expansion at a given depth are pruned, making beam search incomplete. Nodes at a given depth layer are evaluated based on some given static evaluator function, and the k best are selected for expansion. Their successors at the next depth layer of the search tree are evaluated and their k best selected for expansion, and so on. Bead search (Lemons et al. 2022, Bead) is a beam search that selects nodes preferring low $d(n)$. It was found to return lower-cost solutions faster than beam search using f or h.

Hill-climbing is a very simple heuristic search algorithm that commits to a single successor from each node that is expanded. According to a given state evaluation function, Hill-climbing selects *and commits to* the most promising successor s of the initial state s_i, then to the most promising successor s' of s, then the most promising successor s'' of s', and so on, until generating a goal state. Hill-climbing can thus be seen as the limiting case of any beam search with a beam width of $k = 1$, and it is also not complete. The implementation tested below uses a closed list to avoid cycles, and drops duplicate states.

Rectangle is a state-of-the-art anytime algorithm based on beam search (Lemons et al. 2024). Rectangle can be thought of as a beam search whose width increases as the search tree depth increases, according to an 'aspect ratio' specified at runtime. Unlike beam search, Rectangle does not permanently prune nodes that are not selected for expansion, but maintains them in a collection of queues, one for each depth reached so far. As the search progresses deeper and the beam grows wider, Rectangle returns to each previous (shallower) depth layer to select additional nodes for expansion, ensuring that the same number of nodes has been expanded at every depth so far. For example, with an aspect ratio of 1, Rectangle at depth d will expand d nodes, as well as 1 additional node at each previous (shallower) depth. Rectangle is complete and converges to optimal in the limit of running time. To convert Rectangle from an anytime algorithm to a tunable suboptimal algorithm, we terminate the anytime search either (a) when Rectangle finds its jth anytime solution, where $j \in \mathbb{Z}^+$ is specified at runtime, or (b) when rectangle only finds $i : 0 < i < j$ solutions but subsequently also empties all the depth-based queues (pruned based on the incumbent solution), in which case the algorithm has proved the optimality of the ith solution.

The Speed* Search Algorithm

Speed* is a best-first search that considers cost-so-far, cost-to-go, and weighted distance-to-go information in ordering its search frontier. It is extremely simple to implement. It

is designed to interpolate between A* and Speedy searches depending on the value of its speed parameter $s \in [1, \inf)$. Speed* does not guarantee monotonic change in the cost vs. time relationship throughout this interpolation, but wA* shares this behavior (Wilt and Ruml 2012). Given the well-known improvement both in cost and running time of searching on d instead of on h in agile search, it is surprising that Speed* has never been tried before for tunable suboptimal search.

In explaining Speed*'s state evaluation function, $f^\dagger$, we begin with a simpler version of the algorithm, called Speed*5000, which orders its search frontier on $f^{\dagger 5000}(n) = g(n) + h(n) + s \cdot d(n) : s \in \mathbb{R}^{\geq 0}$, with ties broken in favor of low h. First note that in unit cost domains Speed*5000 with $s = 0.1$ implies search behavior similar to that of wA* with $w = 1.1$, so the first modification in Speed* is to require $s \in [1, \inf)$ and to subtract 1 from it, so that Speed* and wA* behave similarly for $w = s > 1$ on unit cost domains, with differences in behavior resulting solely from their goal detection policies (discussed below). Second, note that the effect of s in Speed*5000 depends on the relative magnitudes of h and d in each specific domain, unlike the effect of w in wA* which benefits from g and h being in the same units. To mitigate this at least somewhat, Speed*'s state evaluation function $f^\dagger$ scales $s-1$ by $\frac{h(n_i)}{d(n_i)}$, where n_i is the initial state. This gives us Speed*'s state evaluation function:

$$f^\dagger(n) = g(n) + h(n) + s' \cdot d(n) : s' = (s-1) \cdot \frac{h(n_i)}{d(n_i)}$$

where $s \in [1, \inf)$. Note that s' is fully determined once $h(n_i)$ and $d(n_i)$ are computed and is held constant for the remainder of the search. Finally, with no suboptimality bound to guarantee, Speed* immediately returns the first solution generated when $s > 1$, rather than following wA*'s approach of enqueuing all generated solutions onto the open list and waiting for one to be selected for expansion. When $s = 1$, we special case goal detection in Speed* to occur at expansion instead of generation, in order to behave identically to A*.

There is only a single open list, so we expect overhead to be no greater than wA*'s. In the implementation tested below, Speed* was implemented with a single basic binary heap for the open list and a hash table for the closed list.

The Behavior of Speed*

Speed* is complete in both finite and, under reasonable assumptions, also infinite state spaces. In particular, we assume that (A1) h is admissible; (A2) both h and d are goal-aware, i.e., $h(n) = d(n) = 0$ iff n is a goal state; (A3) d is bounded; (A4) edge costs are bounded away from zero by some fixed finite $\epsilon > 0$; and (A5) the state space is 'locally finite', meaning every node has a finite number of neighbors, thus implying that a single expansion takes a finite amount of time.

Lemma 1 If a solution path exists, then $\exists\, n \in \text{OPEN} : n$ lies on a solution path.

Proof. We proceed by induction.

initialization: if there exists a solution path $p(n_i, n_g)$ from initial state n_i to some goal state n_g, then n_i is on path p and is inserted into OPEN when Speed* begins execution.

maintenance: $\exists\, n \in p \rightarrow \exists\, n' \in succ(n) : n' \in p$. If and when n is extracted from OPEN and expanded, n' will be generated and, because Speed* does not prune, n' will be inserted into OPEN. □

Lemma 2 If a solution p of finite cost C exists, s' is finite.

Proof. The initial state n_i must be on p, so $h(n_i) \leq C$ by (A1). Since s' need only be calculated when n_i is not a goal state, $d(n_i) > 0$ by (A2), making $\frac{h(n_i)}{d(n_i)}$ finite. Therefore $s' = (s-1) \cdot \frac{h(n_i)}{d(n_i)}$ is finite. □

Theorem 1 Speed* is complete: if a finite cost solution p exists, Speed* will find it and terminate.

Proof. If p exists, then by Lemma 1 at any time $\exists\, n_p \in \text{OPEN} : n_p \in p$. Let C denote the finite cost of p. We have that

$$\begin{aligned} f^\dagger(n_p) &= g(n_p) + h(n_p) + s' \cdot d(n_p) \\ &\leq C + s' \cdot d(n_p) \end{aligned}$$

is finite and bounded by Lemma 2 and (A3).

Let k denote the length of p. By (A4) and because C in finite, k is finite. At any time, finite $j \leq k$ expansions along p are required to reach p's goal. Let X represent the finite set of all states on OPEN that are not part of any path that leads to a goal within k steps, and let $X_{front} \subseteq \left\{n : n \in X \wedge f^\dagger(n) \leq f^\dagger(n_p)\right\}$ be the largest subset of X that is ordered before n_p on OPEN. If $X_{front} = \emptyset$, then n_p is selected for expansion and j decrements. If $X_{front} \neq \emptyset$, then some $n_X \in X_{front} : f^\dagger(n_X) \leq f^\dagger(n_p)$ is selected for expansion. By (A5) n_X will have a finite number of successor states, and the effect on X_{front} is as follows:

1. if n_X has no successors, then the cardinality of X_{front} decrements;
2. otherwise, by (A4) we have that $g(n'_X) \geq \epsilon + g(n_X)\ \forall\, n'_X \in succ(n_X)$ for some fixed finite $\epsilon > 0$.

In particular, for any path x that does not reach a goal within k steps from n_i, the number of expansions from any n_x along x that are possible before $g(n'_x) > f^\dagger(n_p)$ for some $n'_x \in x \cap T_{n_x}$, where T_{n_x} denotes the subtree under n_x, is bounded from above by

$$i_{n_p} := \left\lfloor \frac{f^\dagger(n_p)}{\epsilon} \right\rfloor + 1.$$

Let $t_{n_x} \subseteq T_{n_x}$ be the portion, with depth bounded by i_{n_p}, of T_{n_x} that may possibly be selected for expansion before n_p is selected for expansion. Because each t_{n_x} is of bounded depth, $|t_{n_x}|$ is always finite by (A5). We can then represent all the states that may possibly be selected for expansion prior to n_p as the following union:

$$X_{before} := \bigcup_{n_X \in X_{front}} t_{n_X}.$$

Because X_{front} is always of finite size, and because t_{n_X} is finite for any $n_X \in X_{front}$, the overall number of expansions that may occur prior to n_p being selected for expansion is therefore bounded from above by finite $|X_{before}|$.

Since only a finite number of expansions may occur prior to any particular n_p being selected for expansion, and since finite k many n_p must be expanded before generating n_g, Speed* will run for a finite length of time before generating a solution and terminating. □

Even though Speed* does not adhere to an a priori suboptimality bound, it can nonetheless provide one post hoc. Let C be the cost of the solution returned by Speed*, with OPEN retaining its state at the moment of termination. Note that $f_{\min} := \min_{n \in \text{OPEN}} f(n)$ is a lower bound on the unknown cost C^* of an optimal solution. Then we have the post hoc suboptimality bound

$$b := \frac{C}{f_{\min}} \geq \frac{C}{C^*}.$$

Experimental Evaluation

We compared a variety of popular and state-of-the-art tunable suboptimal algorithms, both bounded and unbounded, both complete and incomplete, and with both priority-queue-based and beam-based (breadth-based) search frontiers: wA*, RR-d, Bead, Rectangle, and Speed*. We also include Greedy, Speedy, and Hill-climbing, as they are useful limiting cases. We used five standard search benchmarks: 15-puzzle, traffic, racetrack, blocks world, and pancake. On the 15-puzzle and pancake, we used both unit- and non-unit cost models, and traffic and racetrack enabled us to examine the algorithms' behavior in state spaces with dead-ends.

Tunable algorithms were run with a range of parameter values. We also compared whether dropping vs. reopening duplicate nodes affected performance for wA* and Speed*.

All algorithms were implemented in Rust and run on machines with Intel Core i3-12100 4.3 GHz processors and 64 GB RAM. Only one algorithm run was performed at a time on each machine to minimize cache and memory bus interference from other processes. All algorithms were given a 60 second limit and a 62 GB memory limit (which was never reached by any of the suboptimal algorithms). All CPU times were verified to be within 1% of their corresponding wall times as a post hoc guarantee of no significant preemption by other tasks on the system.

15-Puzzle

The 15-puzzle is a sliding tile puzzle with a 4x4 grid. We use Korf's 100 instances (Korf 1985) and three cost models: unit-cost, heavy tiles where the cost to move each tile is equal to its number (e.g., the cost to move tile number 15 is 15), and inverse tiles, where the cost to move each tile is the multiplicative inverse of its number (e.g., the cost to move tile number 15 is $\frac{1}{15}$).

Results are shown in Figure 1, comparing each tunable algorithm's cost vs. time trade-off over its range of parameter values. Data are averaged over a set of commonly solved instances that were solved by every algorithm (and, if tunable, at every parameter value) that is included in the plot.

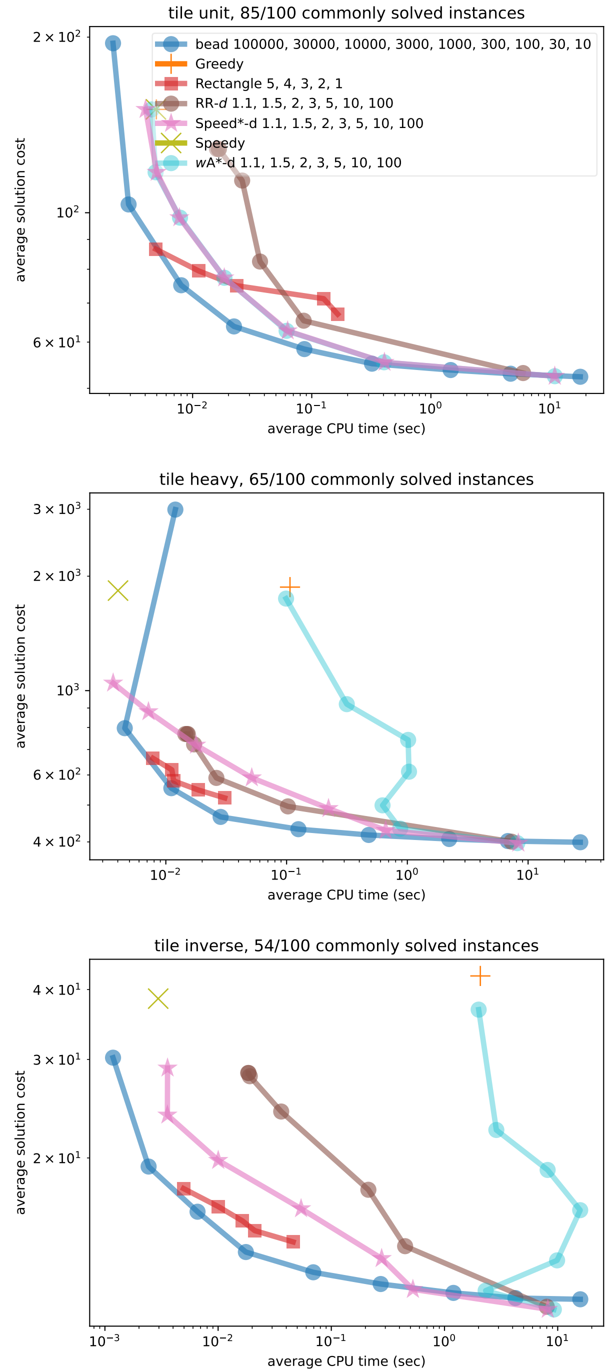


Figure 1: Cost vs. time on 15-puzzle with unit- (top), heavy- (middle), and inverse-cost (bottom) models.

The legends include the different parameter values that were used to sweep out each performance envelope: each point corresponds to a different beam width (Bead), solution number (Rectangle), w (RR-d and wA*), or s (Speed*), averaged across the commonly solved problem instances. We list the 'aspect' parameter of Rectangle as a part of the name ('Rect-

angle' = 1, 'Rectangle500' = 500). The suffixes '-d' vs. '-r' denote the duplicate handling policies used for wA* and Speed* (dropping or reopening). To induce a set of commonly solved problem instances with coverage greater than 40% in a given domain, we incrementally excluded the single algorithm (and, if applicable, only at the single parameter value) that had the lowest coverage in that domain. The resulting number of commonly solved problem instances is displayed in each plot's title. The algorithms' glyphs and colors are constant across all plots.

Bead matches or outperforms every other algorithm against which we compared on all three tiles cost models. Rectangle(1) nearly matches Bead in all three cost models, while Rectangle(500) returns much higher cost solutions (and so is omitted to better visualize the better performing algorithms). This is consistent with Figures 66 (a,b) and 67 (b) of Lemons et al. (2023). As expected given Thayer, Ruml, and Kreis (2009), Speed* and wA* are indistinguishable on unit tiles while Speed* solves problems faster and at lower cost than wA* on heavy and inverse tiles. On unit and inverse tiles, RR-d is outperformed by Speed*, while on heavy tiles they crisscross, neither clearly dominates the other.

We note that wA*'s dominance of RR-d on unit tiles in the tunable suboptimal setting is not inconsistent with Fickert, Gu, and Ruml (2022)'s determination of RR-d's dominance over wA* in the bounded suboptimal setting. For a given suboptimality bound, RR-d runs faster but returns a higher cost solution than wA*, so it is not clear that RR-d should be expected to dominate wA* in terms of cost vs. time in the tunable suboptimal setting.

While Bead performs very well on the 15-puzzle, Hill-climbing fails to solve more than a single instance on any of the three cost models. Hill-climbing is equivalent to beam search with a beam width of 1, so this highlights the brittleness of a beam search's aggressive commitment.

We shall see that results in the other domains without dead-ends generally follow the patterns observed in the 15-puzzle results.

Blocks World

In blocks world, a table supports one or more towers of uniquely numbered but otherwise identical blocks (Slaney and Thiébaux 2001). A block is clear if there is no block stacked on top of it. A goal state is a complete specification of which block is stacked on which other block, or directly on the table. We use the 'shallow' action model wherein an applicable action consists of picking up a clear block and placing it either onto another clear block or onto the table (as opposed to the model introduced by Lelis, Zilles, and Holte (2013) that separates picking and placing into different actions and leads to deeper solutions). We randomly generated instances with 15, 20, and 50 blocks using the same method as Lemons et al. (2022), where the start and goal states are each a random assortment of towers.

Results on 20-blacks are shown in Figure 2; these results are representative of 15-blocks, and 50-blocks results are omitted due to low coverage under the 60 sec experimental time limit. As on the 15-puzzle, Bead again performs best

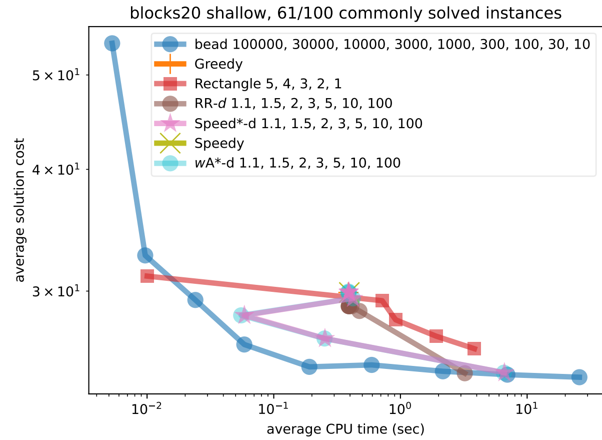


Figure 2: Cost vs. time on 20-blocks world.

overall, finding solutions before any other algorithm tested and almost always finding lower cost solutions. Speed* does not perform as well as Bead, but is also not dominated by any other algorithm. Rectangle(1) does manage to find a first (least optimal) solution with even lower cost, but fails to find sufficiently cheaper subsequent solutions to match the performance of any other algorithm at greater computation times, yielding a poor cost vs. time trade-off. Rectangle(500) is off the top of the plot with the most expensive solutions, running a bit slower than Rectangle. Again RR-d suffers, taking longer to find solutions, and usually at higher cost.

As on non-unit-cost tiles, we can observe the pathological behavior of wA*, which very closely matches (and is plotted behind) that of Speed*. This non-monotonic relationship between running time and suboptimality bound was previously noted by Wilt and Ruml (2012). However, while Speed* can suffer from this as well, it may not always do so in each domain that wA* does (see non-unit cost tiles, Figure 1, middle and bottom). Hill-climbing again performed poorly, solving no more than 5 instances out of 100 on each problem size.

Pancake

In the pancake domain (Helmert 2010), a single stack of different sized pancakes must be rearranged such that no pancake lies on top of a smaller pancake. An action consists of inserting the spatula in between two adjacent pancakes of the stack, and flipping the sub-stack of pancakes above the spatula, reversing their order. We use two cost models: unit-cost and heavy-cost (Hatem and Ruml 2014), where the cost to make a flip is equal to the size of the pancake above the spatula (at the bottom of the sub-stack prior to its being flipped). We use instances with 50, 70, and 100 pancakes. We use the gap heuristic (Helmert 2010) for unit-cost and adapt it for heavy-cost.

Results on 70-pancake are shown in Figure 3 and are representative of results on 50-pancake and 100-pancake. Whereas on the 15-puzzle and blocks world Bead was often first to find solutions, Bead takes longer to start finding solutions on pancake relative to most of the other algorithms.

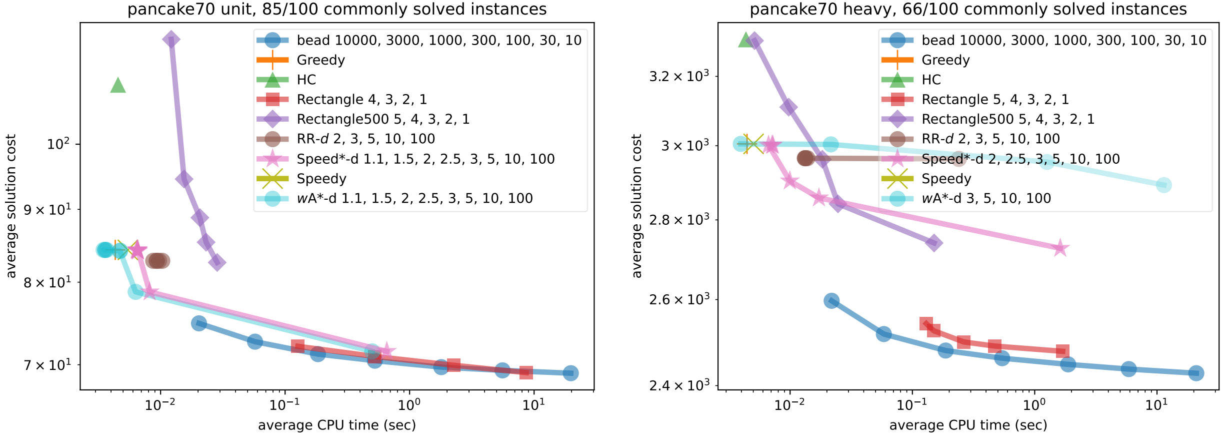


Figure 3: Cost vs. time on 70-pancake with unit- (left) and heavy-cost (right) models.

However, once it does start finding solutions, Bead returns the lowest cost solutions on average, nearly matched only by Rectangle(1). In contrast to almost all other domains tested, Hill-climbing achieved complete coverage on pancake, returning relatively high cost solutions very quickly. While wA* and Speed* found solutions the soonest on unit pancake, and wA* and Rectangle(500) on heavy pancake, wA* and Speed* returned solutions of higher cost than Bead by at least one order of magnitude on heavy pancake. Under both cost models RR-d again suffers, and Rectangle(500) fails to achieve costs as low as any of the other algorithms until its fifth solution.

On the non-dead-end domains of the 15-puzzle, blocks world, and pancake, Bead always found the lowest cost solutions and usually was the first algorithm to find any solutions at all. Rectangle(1) usually matched Bead in cost but took longer to start finding solutions. Speed* usually dominated RR-d and excelled in non-unit-cost models in comparison to wA*.

Traffic

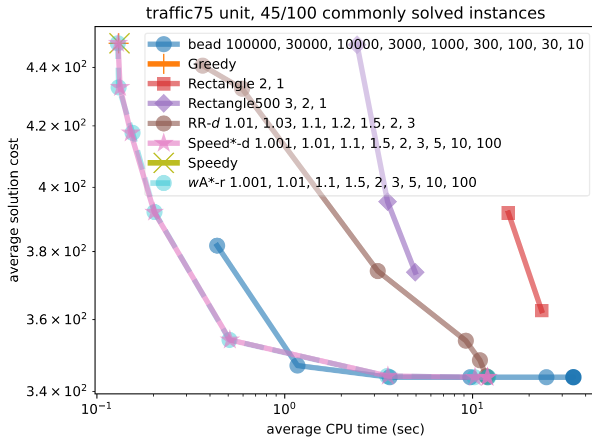
Traffic takes place on a grid with discrete time steps (Kiesel, Burns, and Ruml 2015). The agent's goal is to move from its start location to a goal location elsewhere on the grid without colliding with any dynamic obstacles. Dynamic obstacles have random initial starting locations and horizontal and vertical velocities $dx, dy \in [-1, 0, 1]$, and upon collision with the grid border reverse direction in the dimension perpendicular to the border's edge. At each time step, the dynamic obstacles' positions are updated and the agent takes an action by either moving to an adjacent cell via 4-way movement or remaining in its current cell until the next time step (no-op). We consider unit cost actions, including no-op. While dynamic obstacles pass through each other, the agent may not occupy a cell that contains any dynamic obstacles, so an applicable action in the current time step is one that moves the agent to, or holds the agent in, a cell where no dynamic obstacles will be located at the next time step. Instances are 100x100, with 5k, 7.5k, 8.5k, or 9.5k obstacles, with the agent's start location in the upper-left corner and goal location in the lower-right, and are guaranteed solvable. With these densities of obstacles, we can expect the domain to exhibit dead-end states that have no successors.

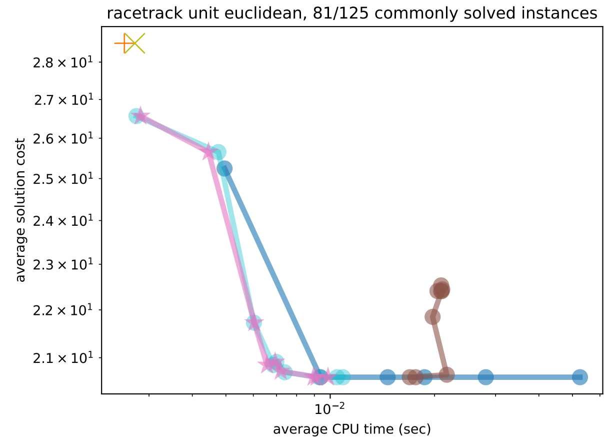
Results with 7.5k obstacles are shown in Figure 4 and are representative of performance on 8.5k and 9.5k obstacles. Speed* and wA* dominate the other algorithms in terms of cost vs. time (top panel). As expected, Bead is not robust in this domain with dead-ends. While Bead did find some solutions at a width of 2, the coverage was so low (bottom panel) that this point was omitted from the cost vs. time plot. To achieve near-complete coverage at a width of 5, Bead took longer than Speed* and wA* did to achieve complete coverage about half an order of magnitude earlier. Again we see RR-d suffers, both in terms of relative delay until first finding solutions and also in returning much higher cost solutions per computation time than Speed*, wA*, or Bead. Even though complete, Rectangle(1) and Rectangle(500) also appear to have difficulty handling traffic's dead-ends, with the better Rectangle taking the longest to find any solution at all compared to the other algorithms. Hill-climbing failed utterly at $\geq$7.5k obstacles with zero coverage, terminating well before the 60 sec time limit.

Racetrack

In racetrack, the agent moves in a grid by applying 1, 0, or -1 acceleration horizontally and vertically at every time step (Gardner 1973; Barto, Bradtke, and Singh 1995). The objective is to reach a goal state on the finish line as quickly as possible. Like Gardner and unlike Barto, Bradtke, and Singh, we use the formulation where successor states are not generated if they fall on blocked grid cells or beyond the map's borders, thus causing dead-end states with no successors. We call this variant New Hampshire Racetrack.[1] We use two heuristics. The Euclidean heuristic admissibly estimates the time to reach the finish line as the the Euclidean distance from the agent's location to the nearest point on

[1] See the state's official motto.





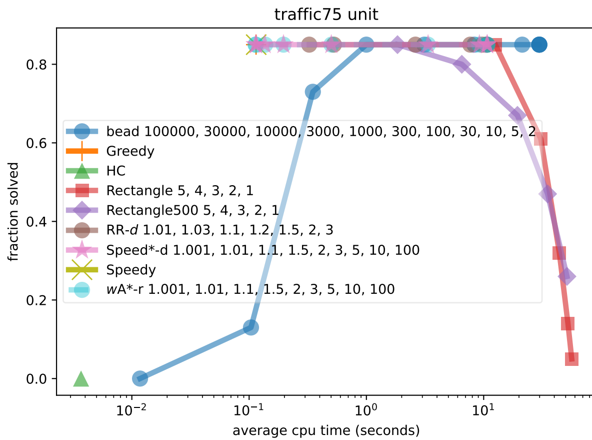


Figure 4: Cost vs. time (top) and coverage vs. time (bottom) on traffic with 7.5k obstacles.

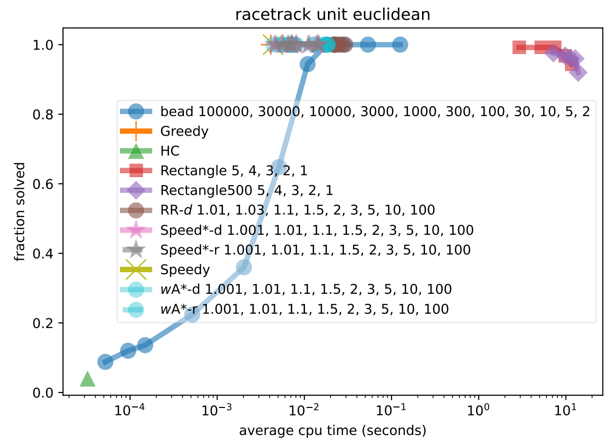


Figure 5: Cost vs. time (top) and coverage vs. time (bottom) on racetrack with the Euclidean heuristic.

the finish line divided by the greater of the maximum velocity achievable in each dimension. The Dijkstra heuristic accounts for obstacles created by the shape of the track by calculating the shortest path from the agent's location to the nearest point on the finish line using 4-way movement. The horizontal and vertical components of the path length are stored separately, and each is divided by the maximum velocity achievable in the respective dimension. The maximum of the two resulting values is returned as an admissible estimate of the time to reach a goal state. Heuristic precomputation occurs before the search algorithm begins and so is excluded from the CPU time measurement.

We use the Barto map (Barto, Bradtke, and Singh 1995) at two scales, as well as the Uniform map at two scales and the Hansen-Barto combined map (Cserna et al. 2018, a longer form of the Barto map), with 25 random start locations that are at least 95% of the way back from the finish line for each map-scale.

Results are shown in Figures 5–6. The coverage vs. time plots (bottom of each of Figures 5–6) show that, while Bead is able to start solving some problems sooner than the other tunable algorithms, it takes Bead more than an order of magnitude longer than Speed* and wA* to achieve full coverage. Bead also cannot boast lower solution costs when the Dijkstra heuristic is used and with the less informative Euclidean heuristic Bead returns significantly higher cost solutions than Speed* and wA* (cost vs. time plots, top of each of Figures 5–6). With both heuristics Speed* finds solutions faster and at lower cost than RR-d, which again suffers. Rectangle(1) and Rectangle(500) are again slower and return higher cost solutions than the other tunable algorithms, so much so under the Euclidean heuristic that they are omitted from that plot (they would appear above and to the right). Hill-climbing performed extremely poorly, solving fewer than 5 instances under each heuristic.

Discussion

In domains without dead-ends, Bead appears to return lower cost solutions more quickly than the other algorithms considered here. In domains with dead-ends, Speed* was robust, finding solutions sooner than Bead and at lower cost, and in non-unit-cost domains Speed* dominates wA*. Interestingly, Hill-climbing almost always utterly fails, even though it is equivalent to beam search with a width of 1.

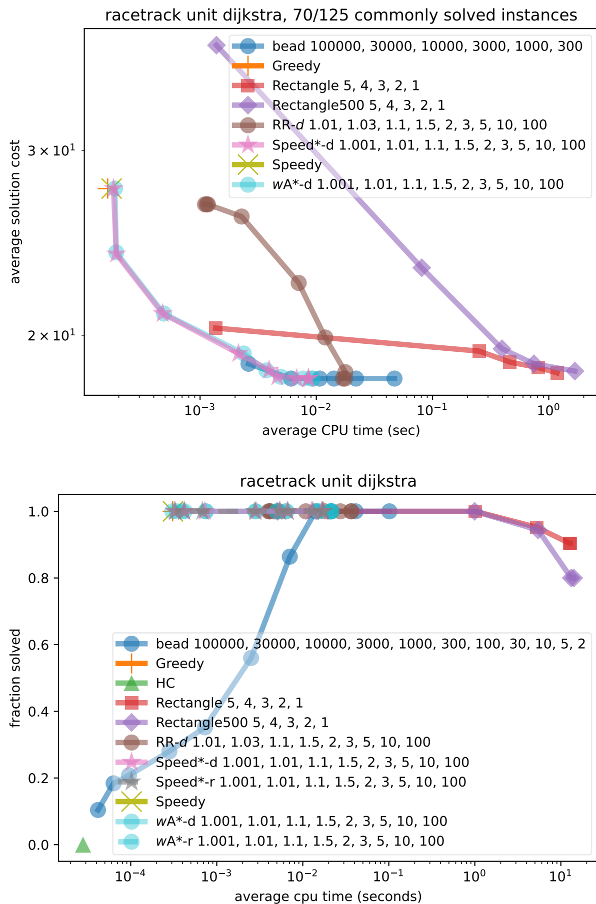


Figure 6: Cost vs. time (top) and coverage vs. time (bottom) on racetrack with the Dijkstra heuristic.

We limited problem instance size in order to be able to compare many algorithms across many domains. However, there are no obvious reasons why our conclusions would not hold as problem difficulty scales up. Our work extends previous studies that have compared suboptimal search algorithms (Wilt, Thayer, and Ruml 2010), but our evaluation is the first to consider the state-of-the-art methods Rectangle and RR-d in the tunable suboptimal problem setting and the first to include a leading anytime algorithm.

As with other best-first searches and any search that keeps a closed list, Speed* stores every generated node, regardless of how poor its evaluation is, so its memory use is proportional to its runtime. If memory consumption is a concern, s can be increased to encourage the algorithm to find solutions more quickly and hence use less memory. It would be an interesting direction for future research to integrate Speed* with bounded-memory algorithms such as A*+IDA* (Bu and Korf 2019).

Conclusions

We present a study of the tunable suboptimal setting, in which the user adjusts an algorithm-specific parameter to achieve an informal trade-off between running time and solution cost, without concern for guaranteeing a specific suboptimality bound. We found experimentally that algorithms that guarantee a bound do often fare worse in cost vs. time performance than algorithms that do not. We evaluated both traditional wA* and state-of-the-art RR-d bounded suboptimal algorithms, and presented a simple, new tunable suboptimal algorithm based on best-first search, Speed*. We also compared against Bead (beam-search on d), and found that Bead is overall the best algorithm for tunable suboptimal search in domains without dead-ends, but that Speed* is preferred otherwise: it is robust to dead-ends, is often faster than wA* under non-unit-cost models, and is almost always (with the exception of on the heavy-cost 15-puzzle) faster than state-of-the-art bounded suboptimal RR-d under both unit- and non-unit cost models.

More generally, we hope this work draws further research attention to the tunable suboptimal setting.

Acknowledgments

We gratefully acknowledge support for this work from the NSF-BSF program via NSF grant 2008594.

References

Barto, A. G.; Bradtke, S. J.; and Singh, S. P. 1995. Learning to Act using Real-Time Dynamic Programming. *Artificial Intelligence*, 72(1): 81–138.

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

Bu, Z.; and Korf, R. E. 2019. A*+IDA*: A Simple Hybrid Search Algorithm. In Kraus, S., ed., *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*, 1206–1212. ijcai.org.

Cserna, B.; Doyle, W. J.; Ramsdell, J. S.; and Ruml, W. 2018. Avoiding Dead Ends in Real-time Heuristic Search. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*.

Dechter, R.; and Pearl, J. 1988. The Optimality of A*. In Kanal, L.; and Kumar, V., eds., *Search in Artificial Intelligence*, 166–199. Springer-Verlag.

Fickert, M.; Gu, T.; and Ruml, W. 2022. New Results in Bounded-Suboptimal Search. In *Proceedings of the Thirty-sixth AAAI Conference on Artificial Intelligence (AAAI-22)*.

Gardner, M. 1973. Mathematical Games. *Scientific American*, 228(5): 102–107.

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 2016*, 36–44. AAAI Press.

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

Hatem, M.; and Ruml, W. 2014. Bounded Suboptimal Search in Linear Space: New Results. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search (SoCS-14)*.

Helmert, M. 2010. Landmark Heuristics for the Pancake Problem. In Felner, A.; and Sturtevant, N. R., eds., *Proceedings of the Third Annual Symposium on Combinatorial Search, SOCS 2010, Stone Mountain, Atlanta, Georgia, USA, July 8-10, 2010*, 109–110. AAAI Press.

Kiesel, S.; Burns, E.; and Ruml, W. 2015. Achieving goals quickly using real-time search: experimental results in video games. *Journal of Artificial Intelligence Research*, 54: 123–158.

Korf, R. E. 1985. Iterative-Deepening-A*: An Optimal Admissible Tree Search. In *Proceedings of IJCAI-85*, 1034–1036.

Lelis, L. H. S.; Zilles, S.; and Holte, R. C. 2013. Stratified tree search: a novel suboptimal heuristic search algorithm. In Gini, M. L.; Shehory, O.; Ito, T.; and Jonker, C. M., eds., *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '13*, 555–562.

Lemons, S.; Linares López, C.; Holte, R. C.; and Ruml, W. 2022. Beam Search: Faster and Monotonic. In *Proceedings of the Thirty-second International Conference on Automated Planning and Scheduling (ICAPS-22)*.

Lemons, S.; Ruml, W.; Holte, R. C.; and Linares López, C. 2023. Rectangle Search: An Anytime Beam Search (Extended Version). arXiv:2312.12554.

Lemons, S.; Ruml, W.; Holte, R. C.; and Linares López, C. 2024. Rectangle Search: An Anytime Beam Search. In *Proceedings of AAAI-24*. AAAI Press.

Michie, D.; and Ross, R. 1969. Experiments with the Adaptive Graph Traverser. In *Machine Intelligence 5*, 301–318.

Pearl, J.; and Kim, J. H. 1982. Studies in Semi-Admissible Heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4(4): 391–399.

Pohl, I. 1970. Heuristic Search Viewed as Path Finding in a Graph. *Artificial Intelligence*, 1: 193–204.

Ruml, W.; and Do, M. B. 2007. Best-first Utility-guided Search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-07)*, 2378–2384.

Slaney, J. K.; and Thiébaux, S. 2001. Blocks World revisited. *Artif. Intell.*, 125(1-2): 119–153.

Stern, R.; Puzis, R.; and Felner, A. 2011. Potential Search: A Bounded-Cost Search Algorithm. In *Proceedings of the Twenty-first International Conference on Automated Planning and Scheduling (ICAPS-11)*.

Thayer, J. T.; Dionne, A.; and Ruml, W. 2011. Learning Inadmissible Heuristics During Search. In *Proceedings of the Twenty-first International Conference on Automated Planning and Scheduling (ICAPS-11)*.

Thayer, J. T.; and Ruml, W. 2011. Bounded Suboptimal Search: A Direct Approach Using Inadmissible Estimates. In *Proceedings of the Twenty-second International Joint Conference on Artificial Intelligence (IJCAI-11)*.

Thayer, J. T.; Ruml, W.; and Kreis, J. 2009. Using Distance Estimates in Heuristic Search: A Re-evaluation. In *Proceedings of the Symposium on Combinatorial Search (SoCS-09)*.

Wilt, C.; and Ruml, W. 2012. When Does Weighted A* Fail? In *Proceedings of SoCS*.

Wilt, C.; Thayer, J.; and Ruml, W. 2010. A Comparison of Greedy Search Algorithms. In *Proceedings of the Symposium on Combinatorial Search (SoCS-10)*.