A Bucket-Based Priority Queue for Bounded-Suboptimal and Anytime A* Search

Garrett M. Fereday and **Eric A. Hansen**

Dept. of Computer Science and Eng., Mississippi State University, USA
gmfereday@gmail.com, hansen@cse.msstate.edu

Abstract

We introduce a priority queue data structure, called a *bucket heap*, which generalizes the *bucket queue* commonly used to accelerate A* search for shortest-path problems with a small range of integer transition costs. Unlike a bucket queue, a bucket heap speeds up priority queue operations for bounded-suboptimal and Anytime A* algorithms guided by non-admissible node evaluation functions. It also provides direct access—without any additional overhead—to the underlying bucket queue of A*, which we show can be used to improve search performance in further ways.

Introduction

For complex shortest-path problems that are not easily solved by the classic A* algorithm (Hart, Nilsson, and Raphael 1968), especially under time constraints, bounded-suboptimal and anytime variants of A* that expand nodes in order of a non-admissible node evaluation function can find a solution path more quickly in exchange for allowing the solution to have a bounded degree of suboptimality.

Bounded-suboptimal search algorithms such as *Weighted A** (Pohl 1970), and related bounded-suboptimal algorithms that use a non-weighted node evaluation function (Gilon, Felner, and Stern 2016), terminate after the first solution is found. *Anytime A** algorithms also expand nodes in order of a non-admissible node evaluation function. But instead of stopping after the first solution is found, they continue to search for improved solutions until eventual convergence to an optimal solution, which allows them to be stopped at any time with the best solution found so far. The efficiency of this approach is further improved by using lower bounds given by an admissible node evaluation function and an upper bound given by the best solution found so far to discard suboptimal nodes that will never be expanded. Examples of the Anytime A* approach include algorithms that expand nodes in order of a weighted node evaluation function (Hansen and Zhou 2007; Likhachev et al. 2008; Richter and Westphal 2010; Thayer, Benton, and Helmert 2012; Flerova, Marinescu, and Dechter 2017; Djukanovic, Raidl, and Blum 2020; Wan and Schweitzer 2021), as well as algorithms that expand nodes in order of related non-weighted node evaluation functions (Stern et al. 2014).

For search problems with a small number of integer transition costs, it is well-known that the performance of both Dijkstra's algorithm and A* can be improved by using a *bucket queue* data structure for the priority queue, instead of the widely-used binary heap (Dial 1969; Burns et al. 2012). By grouping nodes with the same cost into buckets from which they can be expanded in any order, and thus with less overhead, a bucket queue reduces the complexity of priority queue operations from logarithmic to constant time in the number of nodes in the priority queue. However, a bucket queue is not well-suited for bounded-suboptimal and anytime search algorithms that expand nodes in order of a non-admissible node evaluation function. To address this, we introduce a generalization of the bucket queue data structure, called a *bucket heap*, designed specifically for this class of algorithms. We prove that its operations are asymptotically faster than those of a binary heap and show empirically that it can lead to substantial speedups.

Because a bucket heap is built on top of the same bucket queue data structure used by A*, it provides—without added overhead—two priority queues: one ordered by a non-admissible evaluation function and the other by its related consistent (and thus admissible) evaluation function. We conclude by showing how to use the second priority queue to further improve search performance, such as by tightening suboptimality bounds and reducing node re-openings.

Background

The A* algorithm uses a priority queue, also called an Open list, to expand nodes in order of an admissible node evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the best path found so far from the start node to node n and $h(n)$ is a lower bound on the cost of any path from n to a goal node. When used by A*, a priority queue has three basic operations: *enqueue*, which inserts a node, *dequeue*, which removes a node with least f-cost, and *changePriority*, which reorders a node n on the queue when a shorter path is found to n, which decreases $g(n)$ and thus $f(n)$.

When a priority queue is implemented as a binary heap, the *enqueue*, *dequeue* and *changePriority* operations each take worst-case logarithmic time in the size of the queue. The average-case complexity of *dequeue* and *changePriority* is also logarithmic in the size of the queue, although the average-case complexity of *enqueue* is constant.

Dial's Bucket Queue

For shortest-path problems in graphs with a limited range of non-negative integer edge costs, Dial (1969) showed that the worst-case complexity of *enqueue*, *dequeue*, and *changePriority* operations can be reduced from logarithmic to (amortized) constant time by using a bucket queue to implement the priority queue. A bucket queue is an array of "buckets," indexed by integer f-cost, where each bucket is a doubly-linked list of nodes with the same cost.

Updating the f_{min} Index. Each operation on Dial's bucket queue takes constant time except *dequeue*, which takes amortized constant time for a *monotone* priority queue, where a priority queue is said to be *monotone* when nodes are removed in increasing order of f-cost. For A*, the priority queue is monotone when the heuristic h is consistent.

The bucket queue maintains an index f_{min} into the least-cost non-empty bucket from which the next node will be removed. When the *dequeue* operation removes the last node from this bucket, f_{min} is increased to the index of the next non-empty bucket by searching forward in the array of buckets until a non-empty bucket is found. Since this may involve skipping over empty buckets, the worst-case complexity of this step is not constant. However, when the priority queue is monotone—that is, when f_{min} increases monotonically, the amortized complexity of this step is constant.

Two-Level Bucket Queue for A* with Tie Breaking

Although A* can be implemented using Dial's bucket queue, its performance can be improved by expanding nodes with the same f-cost in increasing order of h-cost. Indeed, this tie-breaking rule is particularly effective for problems in which many nodes share the same f-cost, which are precisely the problems for which bucketing is useful.

Therefore, instead of having a bucket queue that is indexed by f-cost only, a two-level bucket queue is more often used for A*, where the first level is indexed by f and the second is indexed by h (Burns et al. 2012, p. 31). This structure can be viewed as a two-dimensional array of buckets, indexed first by f and then by h, where each bucket is a list of nodes with the same f, g and h costs. In this approach, we consider all nodes with the same f-cost to belong to the same *primary bucket*, as in Dial's bucket queue, while all nodes with the same f, g and h costs belong to the same *secondary bucket*. When expanding nodes with the least f-cost, secondary buckets are expanded in increasing order of h-cost for efficient tie breaking.

Updating the h_{min} Index. In a two-level bucket queue, each primary bucket has an h_{min} index that keeps track of the non-empty secondary bucket with least h-cost. When the *dequeue* operation removes the last node from this bucket, h_{min} must be increased to the index of the next non-empty secondary bucket. It is increased by searching forward in the array of secondary buckets for the next non-empty bucket, similar to how f_{min} is increased when the *dequeue* operation removes the last node from the primary bucket indexed by f_{min}. But in contrast to f_{min}, the value of h_{min} is not guaranteed to increase monotonically. It can be decreased when an *enqueue* operation inserts a node into a previously-empty secondary bucket with a lower h-cost than h_{min}. Therefore the *dequeue* operation for a two-level bucket queue does not have amortized constant complexity. Instead, if implemented as we have described, it has worst-case linear complexity in the range of secondary buckets.

The worst-case complexity of *dequeue* can be reduced from linear to logarithmic in the range of secondary buckets by associating with each primary bucket a min-heap of its non-empty secondary buckets. When the *dequeue* operation removes the last node from a secondary bucket indexed by h_{min}, the index of this secondary bucket is removed from the min-heap and the index of the next non-empty secondary bucket can be found in logarithmic time.

Although this approach to updating h_{min} indices has much better worst-case complexity, it does not necessarily perform better in practice. When there are few or no empty secondary buckets, the *average-case* complexity of using a min-heap to update h_{min} is logarithmic compared to constant for linear search. Thus for the test problems in this paper, linear search performs slightly better than a min-heap. In general, the best average-case performance can be achieved by monitoring problem characteristics and adopting the approach that works best for a given problem.

Bucket Heap Data Structure

A bucket queue as described above is not suitable for bounded-suboptimal and anytime search algorithms because it assumes nodes are expanded in order of admissible f-cost. Therefore we propose a more suitable priority queue data structure, called a *bucket heap*, that is built atop the two-level bucket queue described above. Specifically, a bucket heap is a binary heap of indices to the non-empty primary buckets of the two-level bucket queue. The buckets in the heap are ordered by a non-admissible priority function while the same buckets in the bucket queue are indexed by f-cost.

We focus on two classes of non-admissible node evaluation functions that can be used to order a bucket heap.

Bucketing Based on Weighted Evaluation Function

A bucket heap can be used to implement a priority queue that is ordered by the weighted node evaluation function used by Weighted A* (WA*), as well as by Anytime A* algorithms based on it, which is defined as

$$f_w(n) = g(n) + w \cdot h(n), \tag{1}$$

where the parameter $w \geq 1$ is the weight.

Because not all nodes in the same primary bucket of a two-level bucket queue have the same f_w-cost, the priority assigned to a primary bucket when it is in the bucket heap must be a *dynamic priority* that can change over time. It is the smallest f_w-cost of any node currently in the bucket, that is, the f_w-cost of the node or nodes in the secondary bucket indexed by h_{min}. Let b denote a primary bucket. Its dynamic priority in the bucket heap is defined as

$$f_w(b) = f(b) - h_{min}(b) + w \cdot h_{min}(b), \tag{2}$$

where $f(b) - h_{min}(b)$ is the g-cost of the secondary bucket indexed by h_{min}, and $h_{min}(b)$ is its h-cost. Naturally, any change in w necessitates reordering the priority queue.

6-DOF robot	# Expansions	451,277	1,425,126	3,101,184	6,492,269	11,228,621	20,820,068	35,853,787
(non-uniform	# nodes	513,966	1,503,340	2,752,394	4,709,245	6,828,730	9,882,166	1,682,416
cost)	# buckets	41	34	27	21	14	7	1
15-Puzzle	# Expansions	74,004	121,384	205,540	2,104,994	2,951,428	17,314,945	86,782,695
(instance	# nodes	48,274	83,394	154,307	1,705,100	2,336,742	13,424,870	43,128,501
#100)	# buckets	23	14	11	7	5	3	1

Table 1: For two test problems, the effectiveness of bucketing is illustrated by comparing the number of nodes in the priority queue to the number of (primary) buckets in the bucket heap at various points during ANA* search, based on node expansions.

Bucketing Based on Potential Function

A bucket heap can also be used to implement a priority queue that is ordered by what Stern et al. (2014) call a *potential function*. A potential function takes two slightly different forms depending on whether it is used by an anytime search algorithm or a bounded-suboptimal search algorithm. Recall that Anytime A* prunes a node n if it cannot lead to an improved solution, that is, if $g(n) + h(n) \geq C$, where C is the cost of the best solution found so far. That is easily shown to be equivalent to pruning a node n if $u(n) \leq 1$, where

$$u(n) = \frac{C - g(n)}{h(n)}. \tag{3}$$

Stern et al. (2014) call $u(n)$ a *potential* function because the greater its value, the greater the potential (or probability) that its expansion leads to an improved solution. They propose an Anytime A* algorithm, called *Anytime Non-Parametric A** (ANA*), that expands nodes in *decreasing* order of $u(n)$.

Gilon et al. (2016) propose a closely-related bounded-suboptimal search algorithm called *Dynamic Potential Search* (DPS) that expands nodes in decreasing order of the similar node evaluation function

$$ud(n) = \frac{\epsilon \cdot f_{min} - g(n)}{h(n)}, \tag{4}$$

where the constant $\epsilon \geq 1$ is the desired suboptimality bound of the solution. Note that $u(n)$ and $ud(n)$ are related by the simple equation $C = \epsilon \cdot f_{min}$.

When a priority queue ordered by either of these potential functions is implemented as a bucket heap, the dynamic priority assigned to a primary bucket b is

$$u(b) = \frac{C - (f(b) - h_{min}(b))}{h_{min}(b)}, \tag{5}$$

in the case of ANA*, and it is

$$ud(b) = \frac{\epsilon \cdot f_{min} - (f(b) - h_{min}(b))}{h_{min}(b)}, \tag{6}$$

in the case of DPS. Thus ANA* must reorder the priority queue whenever the upper bound C decreases, while DPS must reorder it whenever the lower bound f_{min} increases.

Conditions For Using a Bucket Heap

As these two examples illustrate, our approach to bucketing is designed for non-admissible node evaluation functions with the following properties: (i) the value is based on g-cost, h-cost, and usually at least one other parameter that may change during the search, and (ii) nodes in the same primary bucket are expanded in increasing order of h-cost.

Complexity of Bucket Heap Operations

In addition to maintaining the underlying two-level bucket queue, maintaining a bucket heap requires the following operations: (i) accessing the primary bucket at the root of the bucket heap in order to get the next node to expand, (ii) removing the primary bucket from the root of the heap if it becomes empty after removing a node, (iii) adding a primary bucket to the heap if a node is added to the bucket when it is empty, (iv) updating the priority of a bucket when the h_{min} index of the bucket changes (which also requires moving the bucket up or down in the heap), and (v) updating the priority of *all* primary buckets when a parameter of the node evaluation function changes. A changed parameter could be a changed weight w of the weighted evaluation function of Equation (2), for example. Changing all bucket priorities requires reordering the priority queue, as discussed below.

The complexity of the *enqueue*, *dequeue* and *changePriority* operations of a bucket heap is the sum of the complexity of the corresponding operations in the underlying two-level bucket queue and the complexity of managing the bucket heap. If h_{min} indices are updated using a min-heap, the worst-case complexity of *enqueue*, *dequeue*, and *changePriority* for the two-level bucket queue is logarithmic in the range of non-empty secondary buckets. Managing the bucket heap requires recalculating the dynamic priority (e.g., $f_w(b)$, $u(b)$, or $ud(b)$) of a primary bucket b when its h_{min} index changes as the result of one of these operations, and then moving the bucket up or down in the heap, which takes logarithmic time in the number of non-empty primary buckets. Because the range of non-empty secondary buckets cannot be less than the range of non-empty primary buckets, the worst-case asymptotic complexity of the *enqueue*, *dequeue* and *changePriority* operations of a bucket heap is the same as the worst-case asymptotic complexity of the corresponding operations of a two-level bucket queue for A*. However, average-case complexity may differ depending on bucketing efficiency—often favoring the simpler bucket queue.

As noted above, when the priority of all buckets changes, as when the weight w of the weighted evaluation function of Equation (2) changes, the priority queue must be reordered. For a binary heap, the *reorder* operation takes linear time in the number of nodes in the priority queue (Floyd 1964). For a bucket heap, *reorder* takes linear time in the number of non-empty primary buckets. Therefore, use of a bucket heap dramatically improves the efficiency of this operation.

Pseudocode for the bucket heap operations and additional implementation details are provided by Fereday (2025).

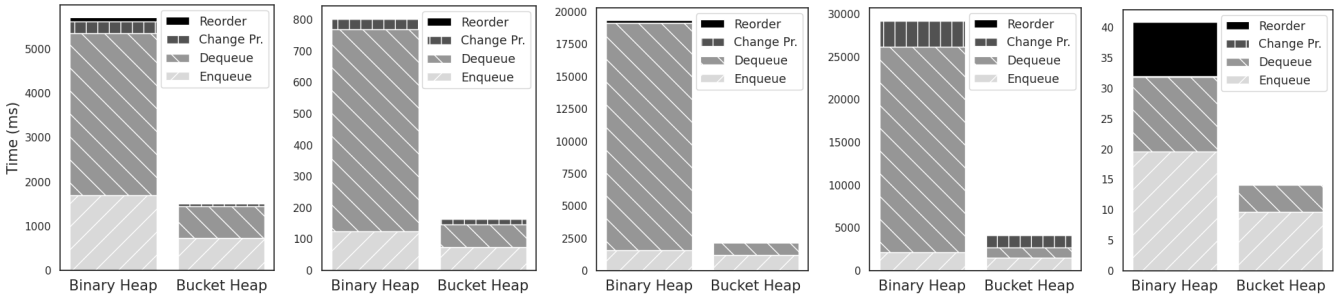


Figure 1: Priority queue overhead (in milliseconds) for ANA* using a binary heap compared to a bucket heap in solving: (a) the 15-puzzle, averaged over Korf's 100 instances, (b) the unit-cost 6-DOF robot arm, (c) the non-uniform cost 6-DOF robot arm, (d) five-sequence alignment, averaged over 6 instances, and (e) the 48-pancake puzzle, averaged over 100 instances.

Experimental Comparison and Analysis

We evaluate our approach using the following well-known search benchmarks:

- *Fifteen puzzle* using the *Manhattan distance heuristic*, averaged over the standard 100 instances (Korf 1985);
- uniform-cost and non-uniform cost *trajectory planning* for 6 and 20 degree-of-freedom (DOF) simulated robotic arms, using the same heuristic as Likhachev et al. (2008);
- *multiple sequence alignment* of 5 sequences of length about 450, averaged over $\binom{6}{5} = 6$ selections of 5 out of 6 sequences from the same set used by Stern et al. (2014), with the same cost function and *Sum-of-Pairs heuristic*;
- *path planning* in a 5000×5000 grid with movement in each cardinal direction and unit transition costs, using the Manhattan distance heuristic. The start state is in the upper left corner and the goal state is in the bottom right corner, with obstacles generated by blocking 20% of cells randomly. Averaged over 25 random instances;
- 48-*pancake puzzle* (Gates and Papadimitriou 1979), where a permutation of numbers ("pancakes") from 1 to 48 is ordered in ascending sequence via a minimal number of prefix reversals ("flips"), using the *gap heuristic* (Helmert 2010). Averaged over 100 random instances;
- and finally, the *heavy* 16-*pancake puzzle*, a challenging extension of the n-pancake puzzle where the cost of a flip is the maximum of the first and last pancakes in the prefix, using the same gap heuristic. Averaged over 25 random instances.

Experiments were performed on a laptop with a 3.2 GHz AMD Ryzen 7 5800H processor with 32 GB RAM.

So far, we have described a bucket as a linked list of nodes. However, linked lists require dynamic memory allocation and suffer from poor cache locality. Therefore, our implementation uses dynamic arrays instead. It improves performance by storing nodes in contiguous memory, which reduces allocation overhead. Removing nodes in LIFO order further enhances cache performance.

Our initial experiments use the Anytime Non-Parametric A* (ANA*) algorithm, which uses the potential function of Equation (3) to order the priority queue (Stern et al. 2014).

Examples of Bucketing. Table 1 illustrates the effectiveness of our approach to bucketing when used by ANA* to solve both an especially difficult instance of the 15-puzzle and the 6-DOF non-uniform cost robot arm planning problem. The number of nodes in the priority queue is compared to the number of buckets in the bucket heap at different times in the search, measured by node-expansion counts. Early in the search, using a non-admissible heuristic increases the number of distinct f-costs on the queue. But as the search continues, and especially as the upper bound C and lower bound f_{min} approach each other, the number of non-empty buckets rapidly decreases and bucketing becomes increasingly effective. The algorithm converges to an optimal solution when $C = f_{min}$, at which point any remaining nodes in the last non-empty bucket are left unexpanded.

Priority Queue Overhead. The bar graphs in Figure 1 compare the priority queue overhead when ANA* uses a bucket heap and a binary heap in solving five test problems.

All priority queue operations are faster using a bucket heap, but the speedup is most striking for *dequeue* because it takes average-case logarithmic time in a binary heap, while in a bucket heap it's effectively constant time. Although *enqueue* is performed at least as many times as *dequeue*, it takes average-case constant time in a binary heap, and so there is less room for speedup. The difference is clearly visible in the bar graphs for the first four problems. For the 48-pancake problem, the *enqueue* operation incurs much more overhead because it is performed many more times than *dequeue* due to (i) the problem's high branching factor and (ii) frequent node re-openings caused by transpositions and the use of an inconsistent (non-admissible) evaluation function.

The *changePriority* operation also takes average-case logarithmic time in a binary heap. But for four of the five test problems, it is not performed enough times to incur significant overhead.

The *reorder* operation has linear complexity in the size of the priority queue. But similarly, it is not performed enough times to incur significant overhead, except for the 48-pancake problem. For the other problems, *enqueue* and *dequeue* are performed tens of millions of times, whereas *reorder* is invoked only tens of times.

	one heap		two heaps	
	PQ	total	PQ	total
15-Puzzle	3.82x	1.73x	5.98x	2.39x
robot (uniform)	4.95x	1.44x	8.17x	1.72x
robot (non-uniform)	8.80x	1.41x	11.67x	1.62x
5-sequence align.	7.78x	1.31x	10.49x	1.51x
48-pancake	2.88x	1.22x	3.44x	1.30x
5000×5000 grid	1.36x	1.14x	1.49x	1.25x

Table 2: Priority queue (PQ) speedup and total search speedup for ANA* when implemented using a bucket heap instead of (i) a single binary heap or (ii) two binary heaps. (Robot results are for the 6-DOF simulated robot arm.)

Two Priority Queues

We next consider another advantage of use of a bucket heap. Because it is built on top of the same two-level bucket queue used by A*, it provides two priority queues: one ordered by a non-admissible priority function and the other by f-cost.

Interestingly, both the bounded-suboptimal search algorithm *Focal Search* (Pearl and Kim 1982) and its anytime extension, called *Anytime Focal Search* (Cohen et al. 2018), use two priority queues: one ordered by f-cost (called the Open list) and the other ordered by a non-admissible priority function (called the Focal list). Several bounded-suboptimal and anytime search algorithms can be viewed as special cases of the focal search paradigm (Ebendt and Drechsler 2009; Gilon, Felner, and Stern 2016). In fact, Cohen et al. (2018) argue that if ANA* is given a second priority queue ordered by f-cost, it can be viewed as a special case of Anytime Focal Search, with the advantage that the second priority queue gives ANA* immediate access to f_{min}. With this lower bound, ANA* can compute a tighter suboptimality bound, C/f_{min}, than the bound provided by the original ANA* algorithm, which has just one priority queue.

The bar graphs in Figure 1 compare the priority queue overhead of ANA* using a bucket heap to ANA* using a single binary heap. However, the bucket heap implementation benefits from immediate access to f_{min}, allowing it to compute the suboptimality bound, C/f_{min}, which is tighter than the bound provided by the original ANA* algorithm. Therefore, to ensure a fair comparison, we also consider an implementation of ANA* using two binary heaps, with one ordered by f-cost, so that both implementations maintain the same bound. Table 2 shows the results of both comparisons. Although the priority queue overhead of using two binary heaps is greater than the overhead of using one binary heap, it is not double, in part because operations on the two binary heaps involve the same nodes, improving cache efficiency.

The reduction in priority queue overhead that is shown in Table 2 is not perfectly correlated with the total speedup shown in the same table. For example, although the priority queue speedup for the Fifteen puzzle is lower than for the robot problems, its total speedup is greater. This is because node-generation overhead is much lower for the Fifteen puzzle, making priority queue overhead a larger share of total search time.

Multiple Ways to Reduce Priority Queue Overhead

In Table 2, there is a lot of variation in the priority queue speedup for different problems. As a rule, the speedup from bucketing is greater when the priority queue is larger, and the priority queue tends to be larger when the heuristic h is weak. A strong heuristic can reduce the size of the priority queue—and thus the potential speedup from bucketing—in two ways. First, a stronger heuristic reduces the search space that must be explored, resulting in fewer nodes inserted into the priority queue. Second, ANA* (and other Anytime A* algorithms) discard any node n from the priority queue if $f(n) = g(h) + h(n) \geq C$, and the stronger the heuristic h, the more effective this pruning technique.

For the problems in Table 2, the search heuristic is weakest for the robot and sequence alignment problems. Since the same heuristic is used for both uniform and non-uniform cost robot planning, it is clearly less informative for non-uniform costs. As a result, the priority queue is larger and there is more speedup from bucketing. The heuristic is much stronger for the pancake and grid pathfinding problems, which reduces the size of the priority queue. Nevertheless, bucketing remains effective. In short, bucketing complements other ways of reducing priority queue overhead.

Comparison to Alternative Bucketing Scheme

A similar approach to bucketing is described in the literature. Gilon, Felner and Stern (2016) propose a bucket-based priority queue for the bounded-suboptimal search algorithm Dynamic Potential Search (DPS). They use bucketing primarily to reduce the overhead of reordering the priority queue each time the lower bound f_{min} increases, since it is a parameter in the DPS node evaluation function given by Equation (4).

Their discussion of bucketing is otherwise brief, however. They do not give implementation details; they do not consider bucketing for any other search algorithm; and they do not experimentally compare the performance of DPS using their bucketing scheme to its performance using binary heaps. We try to fill in the details in the following.

First, we note a key difference between their approach and ours. In their approach, each bucket contains nodes with the same g-cost and h-cost. That is, their "bucket heap" (not a name they use) is a heap of what we call secondary buckets. By contrast, our bucket heap is a heap of primary buckets, containing nodes with the same f-cost. To compare the two approaches experimentally, we assume that the other details of their approach—although not described in their paper—are the same as in our implementation.

Clearly, any difference in performance between the two approaches depends on the number of non-empty secondary buckets versus the number of non-empty primary buckets. Since each non-empty primary bucket must have at least one non-empty secondary bucket, the number of non-empty secondary buckets cannot be less than the number of non-empty primary buckets. Of course, it can be more. In the worst case, C bounds the number of non-empty secondary buckets per primary bucket, and so the total number of non-empty secondary buckets can be greater than the total number of non-empty primary buckets by the factor C.

	ϵ	Dynamic Potential Search (DPS) PQ speedup BH	GFS	total speedup BH	GFS	Optimistic Search (OPS) PQ speedup BH	GFS	total speedup BH	GFS
15-Puzzle	1.05	6.82x	3.37x	2.54x	1.99x	5.79x	3.03x	2.22x	1.82x
6-DOF robot arm (uniform)	1.02	9.00x	6.55x	1.88x	1.69x	6.11x	4.55x	1.71x	1.48x
6-DOF robot arm (non-uniform)	1.02	14.34x	13.93x	1.70x	1.68x	10.06x	9.67x	1.52x	1.50x
5-sequence alignment	1.02	5.65x	3.28x	1.41x	1.33x	3.18x	2.61x	1.25x	1.24x
48-Pancake	1.02	3.40x	2.89x	1.27x	1.24x	2.52x	2.22x	1.15x	1.13x
5000×5000 grid	1.02	1.41x	0.32x	1.30x	0.40x	1.26x	0.28x	1.14x	0.37x

Table 3: For two bounded-suboptimal search algorithms, the table shows both the priority queue speedup and the total speedup using our new bucket heap (BH) and the bucketing scheme of Gilon, Felner and Stern (GFS) relative to using two binary heaps.

Two Bounded-Suboptimal Algorithms Compared. To compare the performance of these two bucketing approaches experimentally, we consider their use in two different bounded-suboptimal search algorithms. *Dynamic Potential Search* (DPS) is similar to ANA*, but uses the slightly different potential function of Equation (4). *Optimistic Search* (Thayer and Ruml 2008) relies on a weighted node evaluation function instead of a potential function. It first runs Weighted A* (WA*) with the weight $w = 2\epsilon - 1$, where $\epsilon \geq 1$ is the desired suboptimality bound. The rationale for setting the weight of WA* greater than the target bound ϵ is that WA* usually finds a solution with a bound much better than the assigned weight w. Once a solution is found, Optimistic Search expands nodes in order of f-cost until it verifies it has a solution with the desired suboptimality bound ϵ.

Both DPS and Optimistic Search use a priority queue ordered by f-cost alongside one ordered by the non-admissible node evaluation function that primarily guides the search. As a result, both benefit from being implemented using a bucket heap. DPS needs a priority queue ordered by f-cost so it has access to f_{min} to compute its node evaluation function. Optimistic Search needs the second priority queue in order to expand nodes in order of f-cost after WA* terminates, as well as to compute the bound C/f_{min}.

Results. Table 3 shows priority queue performance and speedup for the two bounded-suboptimal search algorithms. As in Table 2, results for the heavy 16-Pancake and 20-DOF robot arm problems are omitted, since these problems are too difficult to solve for the target suboptimality bounds.

In Table 3, the columns labeled "BH" report the speedup from using our bucket heap relative to two binary heaps. The columns labeled "GFS" report the speedup from using the bucketing approach of Gilon, Felner, and Stern (2016), also relative to two binary heaps. Gilon et al. advocate the use of bucketing to reduce the overhead of *reordering* the priority queue. Thus it is worth noting that most of the speedup in our experiments comes from reducing the overhead of the *enqueue* and *dequeue* operations, as shown in Figure 1.

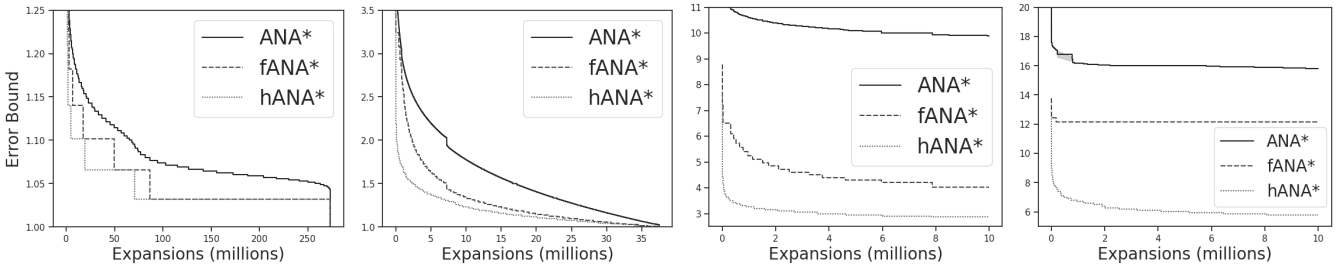
The results in Table 3 show that both approaches to bucketing are effective. The smaller the target suboptimality bound, the longer the search and the larger the priority queue becomes. As a result, the speedup from bucketing is somewhat less for Optimistic Search because it inflates the target suboptimality bound when running its initial WA* search.

Although both approaches to bucketing lead to speedup, more speedup is obtained with our approach. The explanation is simple: the time complexity of bucket heap operations is proportional to the number of buckets in the heap, and the number of non-empty primary buckets is less than the number of non-empty secondary buckets, and often much less. For example, for the 15-puzzle, priority queue operations are approximately twice as fast using our bucket heap as with the bucketing approach of Gilon et al.

For some test problems, the speedup difference between the two approaches is less than for others. For example, the difference is less for the robot planning problems, especially in the non-uniform cost case. That reflects the weakness of the heuristic for these problems. In particular, a weak heuristic reduces the number of non-empty secondary buckets per primary bucket. To help understand why, note that in the extreme case where $h(n) = 0$ for all nodes, which is the weakest possible heuristic, each f-cost corresponds to a unique g-cost, resulting in exactly one secondary bucket per primary bucket. In that case, there would be no speedup difference.

When the heuristic is weak, the priority queue also tends to be larger. Thus bucketing leads to more speedup for the robot arm planning problems than for the other problems. By contrast, in grid pathfinding the heuristic is highly accurate, which both reduces the number of nodes in the priority queue and increases the number of secondary buckets relative to primary buckets. In this case, our approach to bucketing still leads to modest speedup. However, the approach of Gilon et al. significantly degrades search performance. For this search problem, its performance can be up to three times slower than simply using two binary heaps. The performance degradation is caused by an excessive number of secondary buckets compared to nodes in the priority queue, due to both an accurate heuristic and a high optimal solution cost. As a result, nodes become sparsely distributed among many non-empty secondary buckets. When that happens, the resulting overhead for managing the heap of secondary buckets (including reduced cache locality and increased likelihood for dynamic memory allocation) is significantly greater than any benefit from bucketing.

These results suggest that our bucketing approach is not only at least as efficient in all cases, but often significantly more efficient—and more robust across a variety of domains—than the approach proposed by Gilon et al.



(a) (b) (c) (d)

Figure 2: Comparison of the error (suboptimality) bound for the best solution found so far by ANA*, fANA* and hANA* as a function of the number of nodes expanded in solving (a) instance #100 of Korf's 15-Puzzle instances, (b) the non-uniform cost 6-DOF robot arm, (c) a random instance of the heavy 16-pancake problem, and (d) the non-uniform cost 20-DOF robot arm.

Hybrid Expansion Strategy

As we have seen, a second priority queue ordered by f-cost gives bounded-suboptimal and Anytime A* algorithms immediate access to f_{min} and so a better suboptimality bound. We next consider another way in which a second priority queue ordered by f-cost can be used to improve search performance: it allows nodes to be expanded in order of f-cost.

Expanding Nodes in Order of f-cost

The idea of a bounded-suboptimal or anytime search algorithm that sometimes expands nodes in order of f-cost is not new. Recall that *Optimistic Search* proceeds in two stages. First, it uses WA* to quickly find a solution that has a suboptimality bound equal to the weight of the weighted heuristic. Then it expands nodes in order of f-cost to further improve this bound. Because it uses two priority queues to expand nodes in these two different orders, it can be more efficiently implemented using a bucket heap, as Table 3 already shows.

We next consider a strategy that *interleaves* expansion of nodes in order of a non-admissible node evaluation function and expansion of nodes in order of f-cost, instead of separating these expansions into two different stages. In fact, we consider the simplest interleaving strategy possible. We simply alternate expansion of a node in order of the non-admissible node evaluation function with expansion of a node in order of f-cost. We show that this very simple strategy has two advantages. First, it often significantly increases the rate at which the suboptimality bound tightens. Second, it can reduce the number of node re-openings and re-expansions that occur due to use of a non-admissible (and thus inconsistent) node evaluation function.

Faster Convergence of the Suboptimality Bound

First consider alternating expansions from a priority queue ordered by the potential function of Equation (3) and expansions from a priority queue ordered by f-cost. Because this approach alternates expansions of ANA* with expansions of A*, it is essentially a hybrid algorithm of ANA* and A*. Thus we now have three versions of ANA*: the original version that uses a single priority queue, the "focal search" version that uses a second priority queue to compute the tighter suboptimality bound C/f_{min}, and the hybrid algorithm.

To distinguish among these three variants of the search algorithm, we let ANA* denote the original version of the algorithm with its associated bound; we let fANA* denote the "focal search" version of ANA* that expands nodes in exactly the same order, but uses a second priority queue ordered by f-cost to compute the improved bound C/f_{min}; and we let hANA* denote the hybrid algorithm.

Figure 2 compares the rates at which the suboptimality bound converges for these three versions of ANA* in solving four test problems. At first, it may seem surprising that hANA* tightens its suboptimality bound so much faster than fANA* as well as ANA*. To help understand this result, note that the potential function of Equation (3) is defined so that it improves the currently-available solution as fast as possible. By contrast, expanding nodes in order of f-cost improves the lower bound as fast as possible. Thus alternating these two expansion strategies appears to be especially effective in improving the suboptimality bound because it takes turns improving the upper bound and the lower bound.

The increased rate at which the hybrid algorithm improves the suboptimality bound is especially striking for the heavy 16-pancake and non-uniform-cost 20-DoF robot arm problems, which are our two most challenging test problems. Because the heuristics used to solve these two problems are very weak, no version of ANA* can find an optimal solution for either problem under reasonable time and memory constraints. Thus our results suggest that the hybrid strategy is especially effective when the heuristic is weak, and for search problems that are more difficult to solve.

The hybrid strategy can also be used in bounded-suboptimal search. Figure 3a compares the performance of DPS and a hybrid algorithm that alternates expansion in order of the potential function of DPS with expansion in order of f-cost. For various target suboptimality bounds, Figure 3a shows the number of instances of the heavy 16-pancake problem—out of 25 randomly-generated instances—it can solve within 5 million node expansions. Clearly, the hybrid strategy is *much* more effective. Indeed, for difficult-to-solve search problems, our results show that the hybrid strategy not only tightens the suboptimality bound faster, it may be the only approach that can attain a target suboptimality bound within given time and memory constraints.

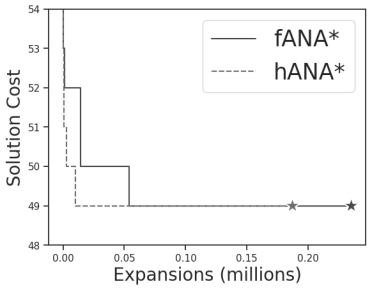
DPS

Target bound	5.0	4.5	4.0	3.5	3.0
Number solved	25	23	15	6	1

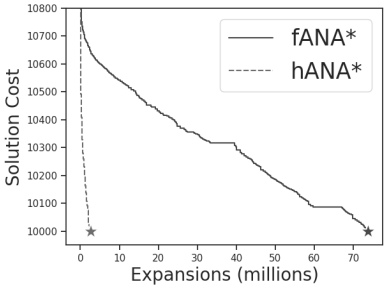
Hybrid of DPS and A*

Target bound	3.0	2.75	2.5	2.25	2.0
Number solved	25	22	7	2	1

(a)



(b)



(c)

Figure 3: For DPS and its hybrid, (a) shows how many out of 25 randomly-generated instances of the heavy 16-pancake problem are solved within 5 million node expansions for various target bounds. For fANA* and hANA*, (b) shows the convergence of the upper bound on solution cost for an instance of the 48-Pancake problem and (c) shows the same for the 5000×5000 grid.

Limiting Node Re-Openings and Re-Expansions

For search problems in graphs with many transpositions, such as grid pathfinding problems, the performance of a bounded-suboptimal or Anytime A* algorithm can be adversely affected by excessive node re-openings and re-expansions due to use of a node evaluation function that is inconsistent (Chen et al. 2019). We next show how to use a hybrid expansion strategy to counteract this problem.

It is well-known that when A* is guided by a consistent node evaluation function, it never re-expands a node. We consider a natural extension of this result: suppose a hybrid Anytime A* algorithm expands every kth node in order of a consistent f-cost, while using an arbitrary (potentially non-admissible) order for the others. Then, assuming the same tie-breaking rule, the total number of nodes expanded by the hybrid algorithm to reach an optimal solution is at most k times the number expanded by A*. This is because, under these conditions, every node expanded by A* will eventually be selected by the hybrid algorithm—within at most k expansions. In this way, adopting a hybrid expansion strategy can bound the number of node re-expansions.

To help characterize problems for which this approach could be useful, we use the term *expansion ratio* to refer to the number of node expansions performed by an Anytime A* algorithm when run until convergence to an optimal solution, divided by the number of node expansions performed by A*. For graph-search problems with many transpositions and an inconsistent node evaluation function, the expansion ratio can be high due to node re-openings and re-expansions. Among our test problems, the 48-pancake and grid pathfinding problems are especially susceptible to this.

	expansion ratio		hybrid
	fANA*	hANA*	speedup
48-Pancake	1.64x	1.06x	1.58x
5000×5000 grid	57.59x	1.90x	27.29x

Table 4: For fANA* and its hybrid variant, hANA*, the table shows the expansion ratio in solving the problem relative to A* and the overall search speedup from the hybrid approach.

Table 4 compares the expansion ratios for fANA* and hANA* when solving the 48-pancake and grid pathfinding problems. It shows a dramatic reduction in expansion ratio when the hybrid strategy is adopted. In fact, by reducing re-expansions, the hybrid algorithm solves the grid pathfinding problem more than 27 times faster than fANA*. (This speedup is *in addition* to the speedup from the bucket heap.) The hybrid strategy also leads to moderate speedup in solving the 48-pancake problem. The hybrid strategy does not improve performance on the other test problems, where node re-expansion is minimal. However, it does not slow convergence either, since every node expanded in order of f-cost must eventually be expanded to ensure optimality.

One might suppose the hybrid strategy leads to faster convergence in solving the pancake and grid problems only because it improves the lower bound faster. But as shown in Figures 3b and 3c, the hybrid strategy can also lead to faster improvement of the upper bound. The explanation is simple: occasionally expanding nodes in f-cost order can improve the upper bound by discovering better partial paths to states already included in the current best solution path.

Conclusion

We have provided the first comprehensive analysis of how to use bucketing to reduce priority queue overhead in bounded-suboptimal and Anytime A* search. Although the observed speedup is sometimes modest, the approach asymptotically improves priority queue performance, which means speedup increases with priority queue size. Moreover, bucketing complements other methods for enhancing search efficiency and it can be especially useful in time-critical applications.

A bucket heap also enables an efficient implementation of two priority queues: one ordered by a non-admissible node evaluation function and the other by f-cost. The second priority queue gives a tight lower bound on solution cost. It also supports a hybrid strategy that alternates between expanding nodes from each of the two queues. We have shown that this hybrid strategy can both improve the rate at which the suboptimality bound tightens and reduce node re-openings and re-expansions caused by use of a non-admissible heuristic.

Acknowledgments

The authors are grateful for feedback from the anonymous reviews that helped improve this paper. Partial support for this research was provided by NSF Award IIS:RI #1718384.

References

Burns, E.; Hatem, M.; Leighton, M.; and Ruml, W. 2012. Implementing Fast Heuristic Search Code. *Proc. of the 5th Annual Symposium on Combinatorial Search*, 25–32.

Chen, J.; Sturtevant, N. R.; Doyle, W. J.; and Ruml, W. 2019. Revisiting Suboptimal Search. In *Proc. of the 12th Annual Symposium on Combinatorial Search*, 18–25.

Cohen, L.; Greco, M.; Ma, H.; Hernandez, C.; Felner, A.; Kumar, T.; and Koenig, S. 2018. Anytime focal search with applications. In *Proc. of the 27th International Joint Conference on Artificial Intelligence*, 1434–1441.

Dial, R. 1969. Shortest-path forest with topological ordering. *Communications of the ACM*, 12(11): 632–633.

Djukanovic, M.; Raidl, G.; and Blum, C. 2020. Finding Longest Common Subsequences: New Anytime A* search results. *Applied Soft Computing*, 95: 106499.

Ebendt, R.; and Drechsler, R. 2009. Weighted A* search – unifying view and application. *Artificial Intelligence*, 173(14): 1310–1342.

Fereday, G. 2025. *Bucket-based priority queues for A* and related bounded-suboptimal and anytime search algorithms: Theoretical and practical advancements*. Ph.D. thesis, Mississippi State University.

Flerova, N.; Marinescu, R.; and Dechter, R. 2017. Weighted heuristic anytime search: New schemes for optimization over graphical models. *Annals of Mathematics and Artificial Intelligence*, 79(1–3): 77–128.

Floyd, R. 1964. Algorithm 245: Treesort 3. *Communications of the ACM*, 7: 701.

Gates, W.; and Papadimitriou, C. 1979. Bounds for sorting by prefix reversal. *Discrete Mathematics*, 27(1): 47–57.

Gilon, D.; Felner, A.; and Stern, R. 2016. Dynamic potential search – a new bounded suboptimal search algorithm. In *Proc. of the 9th Annual Symposium on Combinatorial Search*, 36–44. AAAI press.

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

Hart, P.; Nilsson, N.; 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. 2010. Landmark Heuristics for the Pancake Problem. In *Proc. of the Third Annual Symposium on Combinatorial Search*.

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

Likhachev, M.; Ferguson, D.; Gordon, G.; Stentz, A.; and Thrun, S. 2008. Anytime search in dynamic graphs. *Artificial Intelligence*, 172: 1613–1643.

Pearl, J.; and Kim, J. 1982. Studies in semi-admissible heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4: 391–399.

Pohl, I. 1970. First results on the effect of error in heuristic search. *Machine Intelligence*, 5: 219–236.

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research*, 39: 127–177.

Stern, R.; Felner, A.; van den Berg, J.; Puzis, R.; Shah, R.; and Goldberg, K. 2014. Potential-based bounded-cost search and Anytime Non-Parametric A*. *Artificial Intelligence*, 214: 1–25.

Thayer, J.; Benton, J.; and Helmert, M. 2012. Better Parameter-free Anytime Search by Minimizing Time Between Solutions. In *Proc. of the Fifth Annual Symposium on Combinatorial Search (SOCS-12)*, 120–128.

Thayer, J.; and Ruml, W. 2008. Faster than Weighted A*: An Optimistic Approach to Bounded Suboptimal Search. In *Proc. of the 18th International Conference on Automated Planning and Scheduling*, 355–362.

Wan, G.; and Schweitzer, H. 2021. Accelerated Combinatorial Search for Outlier Detection with Provable Bound on Sub-Optimality. *Proc. of the AAAI Conference on Artificial Intelligence*, 34(14): 12436–12444.