Bucket Aggregation in Bucket-Based Priority Queues for Heuristic Search

Ryan D. Goodwin[1], **Eric A. Hansen**[1], **Garrett M. Fereday**[2]

[1]Department of Computer Science and Engineering, Mississippi State University, USA
[2]Map Engineering and Cartography Department, Garmin International, Inc., Olathe, Kansas, USA
rdg291@msstate.edu, hansen@cse.msstate.edu, Rhett.Fereday@garmin.com

Abstract

Bucket-based priority queues can reduce the overhead of priority queue operations for A* and related bounded-suboptimal and anytime variants, especially when transition costs are small integers. The buckets are usually indexed by both f-cost (primary) and h-cost (secondary), which enables efficient tie-breaking and priority ordering. However, when solution costs are large and the heuristic is informative, the number of secondary buckets within a primary bucket can become very large, leading to bucket sparsity that degrades performance. To address this, we introduce a simple aggregation method that restores bucket density while preserving standard solution-quality guarantees. We demonstrate its effectiveness in several benchmark domains.

Code — https://github.com/RDG0818/BucketAggregation

Introduction

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 a binary heap (Dial 1969; Burns et al. 2012). Recently, a *bucket heap* data structure has been proposed that generalizes this approach to support bounded-suboptimal and anytime variants of A* that use a non-admissible evaluation function (Fereday and Hansen 2025). Both structures group nodes with equal priorities into buckets, allowing expansion in any order and thus reducing priority queue overhead compared to binary heaps. Often, they allow near-constant-time operations in favorable cases.

In this paper, we address a complication affecting bucket queues and bucket heaps when buckets are indexed by both node f-cost and h-cost. In this setup, a *primary bucket* contains nodes with the same f-cost, while a *secondary bucket* contains the subset sharing the same h-cost. For A*, this two-level indexing enables efficient tie-breaking on h-cost. For bounded-suboptimal and anytime variants, two-level indexing is required when the evaluation function weights g-cost and h-cost differently. Thus, a common two-level bucket-based priority queue can support a wide range of heuristic search algorithms.

The efficiency of bucket-based priority queues, however, depends on the distribution of nodes among buckets. When A* solves search problems with integer edge costs bounded by a small constant, the f-costs of nodes in the priority queue remain within a small bounded range. Nevertheless, when the heuristic h is informative, the number of distinct h-costs within a single f-level can grow proportionally to the optimal solution cost. Thus, for problems with very large solution costs, an informative heuristic can, counterintuitively, increase priority queue overhead by generating many distinct h-costs within a single f-level. The resulting bucket sparsity can negate the constant-time benefits of the bucket structure. This issue is further exacerbated when the active range of f-costs grows, either because transition costs are larger or because bounded-suboptimal and anytime algorithms use non-admissible node evaluation functions.

To limit secondary bucket sparsity, we propose a simple approach to aggregating secondary buckets. By trading a small loss of resolution in h-based ordering for improved bucket density, we reduce overhead without loss of solution quality guarantees. For integer-cost search domains with high solution cost, our experiments show that bucket aggregation improves the performance of bucket-based priority queues when bucket density would otherwise be sparse.

Background

Best-first search algorithms, including A* and its bounded-suboptimal and anytime variants, use a priority queue to organize the search, ordered by a function of both $g(n)$, the cost of the best path found so far from the start node to node n, and $h(n)$, a lower bound on the cost of a path from n to a goal node. Algorithms in this class primarily differ in the node evaluation function used to order nodes in the queue.

The priority queue has three basic operations: *enqueue*, which inserts a node; *dequeue*, which removes the next node in order of priority; and *changePriority*, which reorders a node n in the queue when a shorter path to n is found that decreases $g(n)$. For bounded-suboptimal and anytime variants of A* that change the priority function during the search, an additional *reorder* operation rebuilds the priority queue.

When implemented as a binary heap, *dequeue* and *changePriority* each take $O(\log n)$ time, while *enqueue* has $O(\log n)$ worst-case complexity but is often close to constant time in practice. *Reorder* takes $O(n)$ time.

Two-Level Bucket-Based Priority Queues

We consider two-level bucket-based priority queues that index nodes by both f-cost and h-cost, where f-cost is the primary index and h-cost the secondary index. Nodes with the same f-cost belong in the same primary bucket and nodes with the same f- and h-cost belong to the same secondary bucket. For A*, as already noted, this two-tiered bucketing scheme supports tie-breaking among nodes with equal f-cost in favor of nodes with smaller h-cost. For bounded-suboptimal and anytime variants of A*, it supports node expansion in order of a non-admissible node evaluation function that weights h-cost differently than g-cost.

Many implementations of A* use a two-level bucket queue (Burns et al. 2012). A two-level bucket queue also underlies the bucket heap used by bounded-suboptimal and anytime variants of A* (Fereday and Hansen 2025).

Bucket Queue. In a two-level bucket queue, primary buckets are stored in an array indexed by f, and each primary bucket maintains an array of secondary buckets indexed by h. The least non-empty primary bucket is tracked by a pointer $f_{\min}$, while each primary bucket maintains a pointer $h_{\min}$ to its least non-empty secondary bucket. Because $f_{\min}$ is monotonic, it can be updated in amortized constant time. In contrast, $h_{\min}$ is not monotonic and may require scanning to locate the next non-empty secondary bucket after a bucket becomes empty.

With array-based buckets implemented as dynamic arrays, *enqueue* takes amortized $O(1)$ time. *Dequeue* also takes $O(1)$ time once the active secondary bucket (indexed by $h_{\min}$) is known; but when that bucket becomes empty, updating $h_{\min}$ may require scanning over empty secondary buckets. Thus the worst-case cost of *dequeue* is linear in the number of secondary buckets, although in practice it depends strongly on bucket density. Similarly, *changePriority* is amortized $O(1)$ except when removing the last node from a secondary bucket forces an update of $h_{\min}$.

When many secondary buckets are empty, locating the next non-empty bucket may require scanning over a large range of h-values. Sparse secondary buckets also reduce cache locality by spreading nodes across a larger set of bucket structures. Consequently, priority queue performance depends not only on the number of nodes in the queue, but also on the density of secondary buckets.

Bucket Heap. A bucket heap augments the two-level bucket queue with a binary heap whose elements are the non-empty primary buckets. The heap orders primary buckets according to the node evaluation function of the search algorithm. For each primary bucket, the evaluation function is applied to the g- and h-costs of its active secondary bucket, where $h = h_{\min}$ and $g = f - h_{\min}$. Consequently, the priority of a primary bucket changes only when its active secondary bucket changes. Since only non-empty primary buckets appear in the heap, maintaining heap priorities adds little overhead compared to maintaining the underlying bucket queue. Thus, the performance of a bucket heap depends largely on the efficiency of the underlying two-level bucket queue. The dominant overhead remains maintaining the secondary buckets and updating $h_{\min}$.

Secondary Bucket Aggregation

Two-level bucket-based priority queues have an underappreciated failure mode. When solving search problems with both a high solution cost and a sufficiently informative heuristic, the range of h-values within a primary bucket can become large relative to the number of nodes it contains, creating many empty or near-empty secondary buckets. Bucket sparsity increases memory usage and allocation overhead, reduces cache locality, and increases the cost of finding the next non-empty secondary bucket.

To mitigate these effects, we propose the aggregation of nearby secondary buckets into coarser buckets that cover ranges of h-values. For simplicity, we partition the h-value range into intervals of size α, where α is a tunable parameter. Note that secondary bucket aggregation requires no modifications of the search algorithm itself. All that changes is assignment of a representative h-value $\hat{h}(b)$ to each aggregated bucket b. We first consider how to make that assignment in a way that maintains solution quality guarantees, and then consider how aggregation affects node expansion order.

Solution Quality Guarantees. To preserve solution quality guarantees, the representative value $\hat{h}(b)$ assigned to an aggregated bucket must be chosen so that the bucket priority is no greater than the priority of any node it contains, ensuring that no node is expanded ahead of its true priority.

For search algorithms that expand nodes in increasing order of a weighted evaluation $f_w(n) = g(n) + w \cdot h(n) = f(n) + (w-1)h(n)$, including A* (for which $w = 1$), we set $\hat{h}(b)$ to the minimum possible h-value of any node in the aggregated bucket b, ensuring that $\hat{f}_w(b) = f + (w-1)\hat{h}(b)$ is a lower bound on $f_w(n)$ for every node n in b. Expanding nodes in nondecreasing order of a lower bound on f_w preserves the standard suboptimality guarantee.

For algorithms such as ANA* (Stern et al. 2014) and Dynamic Potential Search (Gilon, Felner, and Stern 2016), which expand nodes in *decreasing* order of a *potential function*, the analysis is reversed because their priority functions decrease as $h(n)$ increases. ANA* expands nodes in decreasing order of the potential $u(n) = (C - g(n))/h(n)$, where C is the cost of the best solution found so far. Using $g(n) + h(n) = f$, this can be written as $u(n) = 1 + (C-f)/h(n)$, which is decreasing in $h(n)$ (since $C \geq f$ for any open node). Therefore, we set $\hat{h}(b)$ to the *maximum* possible h-value in bucket b, and the resulting bucket priority is a lower bound on the potential of any node in the bucket. Similar reasoning applies to Dynamic Potential Search.

Priority Resolution and Expansion Order. Although aggregation preserves solution quality guarantees, it reduces priority resolution by assigning the same priority to nodes that would otherwise receive different priorities. As a result, expansion order is less informed, and the expansion of nodes with higher true priority within an aggregated bucket may be delayed until other nodes in the same bucket are expanded.

For bounded-suboptimal and anytime variants of A*, the effect of bucket aggregation on node expansion order is limited (though not eliminated) because aggregation does not affect pruning, which depends on cost bounds rather than the

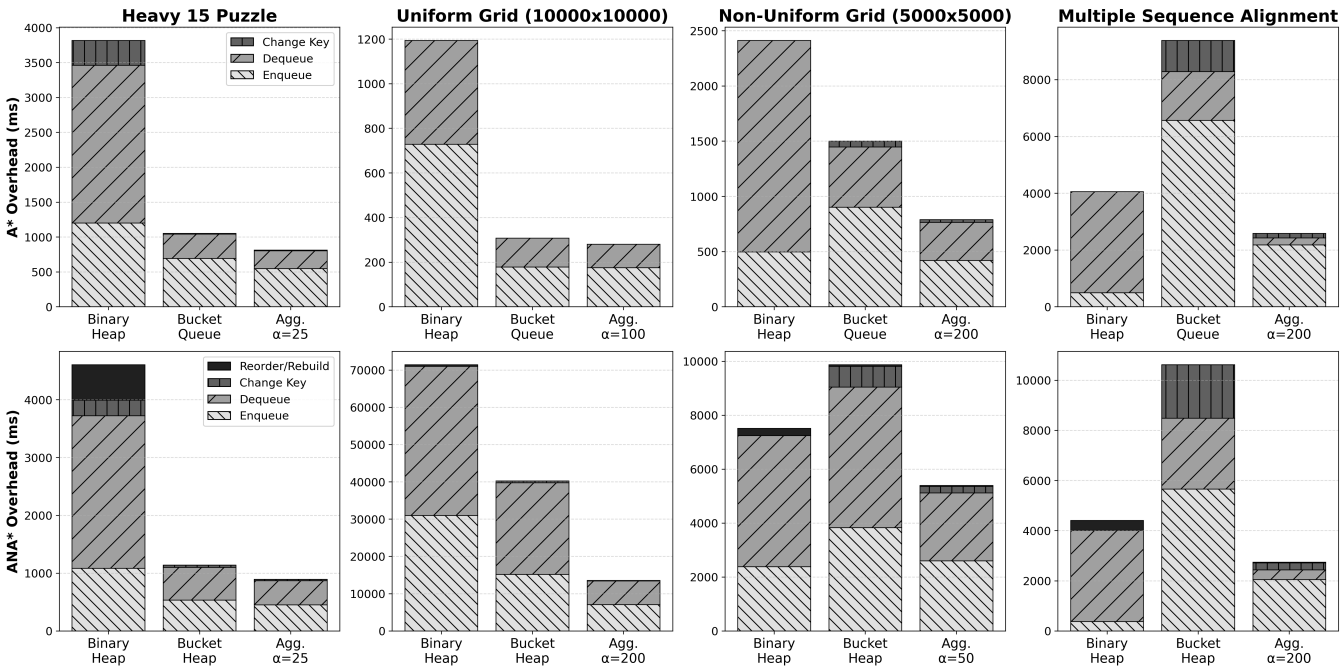

Figure 1: Priority queue overhead (in milliseconds) for A* (above) and ANA* (below) using a binary heap, a standard bucket queue ($\alpha = 1$), and a bucket queue/heap with secondary bucket aggregation based on the indicated aggregation factor α.

fine-grained ordering of nodes within an aggregated bucket. For A*, a similar observation can be used to mitigate the loss of tie-breaking resolution. Once a goal node is generated, A* can terminate as soon as it begins expanding a primary bucket with the same f-value. This works because all nodes n in that bucket have $f(n) \geq g(\text{goal})$.

Experiments and Analysis

We evaluated bucket aggregation in the following domains, all with integer transition costs, high optimal solution costs, and relatively informative heuristics.

- The heavy 15-puzzle, where tile movement cost equals the tile's label, using the heavy Manhattan distance heuristic, averaged over Korf's (1985) 100 instances. Optimal solution costs range from 400 to 450.
- Path planning in a $10{,}000 \times 10{,}000$ grid with cardinal moves, unit costs, and Manhattan distance heuristic, averaged over 25 random instances with 20% obstacles. Optimal solution costs are around 20,000.
- Path planning in a $5{,}000 \times 5{,}000$ grid with the same setup but non-uniform integer costs in $[1, 5]$, averaged over 25 instances. Optimal solution costs average around 37,000.
- Five-sequence protein alignment averaged over the same dataset used by Stern et al. (2014), but with PAM250 costs (Lermen and Reinert 2000) and the sum-of-pairs heuristic. Optimal solution costs average around 64,000.

All experiments were conducted on an Ubuntu 22.04 LTS system using an Intel Core i9-14900K processor with 32 GB of RAM. The software was compiled with GCC 15.2.1 (-O3) and executed in a single-threaded configuration.

A* and ANA*. The top and bottom rows of Figure 1 compare priority queue overhead incurred by A* and ANA*, respectively, in solving these search problems using a binary heap, a two-level bucket queue without aggregation ($\alpha = 1$), and a bucket queue with secondary buckets aggregated based on an aggregation factor α tuned for each problem.

The domains are ordered from left to right by increasing solution cost. As solution cost increases, the best aggregation factor α generally increases as well, reflecting the larger range of h-values that must be represented within a primary bucket. More importantly, the benefit of aggregation increases substantially. For the heavy 15-puzzle and uniform grid, an unaggregated bucket queue already outperforms a binary heap, and aggregation improves performance further. In contrast, for the non-uniform grid and Multiple Sequence Alignment (MSA)—the domains with the largest solution costs—a bucket queue without aggregation can perform even worse than a binary heap. In these domains, aggregation not only recovers the performance lost to sparsity, but reestablishes a substantial advantage over the binary heap baseline. These results confirm that the failure mode for bucketing is tied to solution cost: as solution cost increases from left to right, the penalty for using an unaggregated bucket queue grows, and the benefit of aggregation grows correspondingly.

For the non-uniform grid, the unaggregated bucket heap performs much worse for ANA* than it does for A*, which is consistent with the tendency of non-admissible expansion orders to maintain more active primary buckets simultaneously, further reducing bucket density and amplifying the overhead associated with sparse secondary buckets.

Domain	Queue	Time (s)	Page F.	LLC M.	Branch M.
Grid	Unaggregated	97.02	234K	109M	1.62B
	Aggregated	**47.22**	**8K**	**40M**	**0.49B**
MSA-5	Unaggregated	60.60	1.24M	1.13B	142M
	Aggregated	**49.90**	**55K**	**0.94B**	**90M**

Table 1: Hardware profiling of priority queue overhead for ANA* on non-uniform grid and MSA problem instances. Bucket aggregation reduces page faults, improves cache locality (fewer LLC misses), and improves CPU predictability (fewer branch mispredictions).

Profiling Performance. Figure 1 shows that aggregation reduces overhead across all priority queue operations. Table 1 helps explain why. For ANA* on the non-uniform grid and MSA domains, aggregation reduces page faults by more than an order of magnitude and substantially decreases both last-level cache (LLC) misses and branch mispredictions.

Aggregation reduces *dequeue* overhead by shortening scans for the next non-empty secondary bucket and thereby reducing the branch mispredictions caused by sparse buckets. It reduces *enqueue* overhead by decreasing the number of memory allocations. More generally, the large reduction in page faults is consistent with the observation that aggregation eliminates many small allocations associated with sparsely populated secondary buckets, while the reduction in LLC misses reflects improved cache locality. Together, the results suggest that aggregation improves the overall memory behavior of the priority queue, not just bucket scanning.

Tuning the Aggregation Factor. Table 2 examines the sensitivity of performance to the aggregation factor α on the non-uniform grid. For A*, bucket aggregation reduces priority queue overhead substantially, but has little effect on search order because it affects only tie-breaking among nodes with the same f-cost. Thus, runtime improves significantly once severe bucket sparsity is eliminated, but is relatively insensitive to the exact value of α thereafter. In fact, A* expands exactly the same set of nodes for all values of α in this domain. Thus, aggregation acts primarily as a priority queue optimization for A*, reducing overhead without changing the search itself.

ANA* presents a different tradeoff. Increasing α initially produces large reductions in priority queue overhead, but it also reduces priority resolution, which leads to progressively more node expansions. As a result, the range of near-optimal α values is smaller; larger values of α further increase node expansions. This also explains why aggregation benefits A* more than ANA* in this domain: for ANA*, reduced priority resolution leads to additional node expansions that partially offset the savings from improved bucket density.

For grid path-planning problems, ANA* expands many more nodes than A* not only due to its non-admissible expansion order, but because the large number of alternative paths to the same state in grid graphs makes the search unusually sensitive to node re-openings and re-expansions when nodes are expanded in non-admissible order (Chen et al. 2019; Fereday and Hansen 2025).

	A* running time				ANA* running time		
Queue, α	**Overall**	**PQ**	**Enq.**	**Deq.**	**Overall**	**PQ**	**Nodes**
bin. heap	4303	2479	497	1916	11149	6925	47.0M
$\alpha = 1$	3228	1501	902	600	15721	10812	48.3M
$\alpha = 25$	2452	833	452	381	**10043**	5462	57.4M
$\alpha = 50$	**2439**	813	438	375	10487	5366	67.7M
$\alpha = 100$	2443	797	426	371	12114	5856	86.4M
$\alpha = 200$	2477	790	420	370	15403	7173	116.7M

Table 2: Sensitivity of performance to aggregation factor α on the non-uniform grid. All quantities are in milliseconds except for the number of nodes expanded by ANA*.

Dynamic Potential Search. To evaluate the effectiveness of secondary bucket aggregation in bounded-suboptimal search, we used it in Dynamic Potential Search (Gilon, Felner, and Stern 2016) to solve a difficult 6-sequence MSA instance, consisting of the six sequences in the dataset used in our 5-sequence alignment experiments. At a suboptimality bound of $\epsilon = 1.04$, an unaggregated bucket heap performed substantially worse than a binary heap due to severe bucket sparsity. With aggregation ($\alpha = 2000$), however, the bucket heap achieved more than twice the priority queue throughput of the binary heap and reduced overall search time by a factor of $1.85\times$. The solution cost was 96,206, consistent with our observation that larger solution costs tend to require larger values of α to offset increasing bucket sparsity.

Conclusion

Our results confirm the effectiveness of bucketing in reducing priority queue overhead when solving challenging search problems, but also identify a previously uncharacterized failure mode: for problems with high solution cost and an informative heuristic, secondary bucket sparsity can substantially degrade performance. Secondary bucket aggregation addresses this by trading a minor loss of priority resolution for a restoration of memory and cache density, improving the scalability of bucket-based search.

Interestingly, secondary bucket aggregation creates an intermediate data structure between a standard two-level bucket queue for A* and Dial's one-level bucket queue, since the latter essentially aggregates all secondary buckets into a single bucket. From this perspective, our approach offers a principled tradeoff between these two designs.

Related work on multi-level bucket queues and radix heaps (Edelkamp and Schrödl 2012) also coarsens buckets to improve efficiency. However, those approaches coarsen primary buckets while preserving exact ordering, whereas we aggregate secondary buckets and preserve correctness through lower-bound priority assignments. The relationship between these approaches is worth exploring further.

Aggregating primary as well as secondary buckets could further reduce priority queue overhead, especially in domains with non-uniform transition costs, and would also be necessary to extend bucketing to search problems with real-valued transition costs. Exploring the interaction between primary- and secondary-bucket aggregation may therefore provide a path toward even more scalable bucket-based priority queues.

Acknowledgments

The authors gratefully acknowledge detailed feedback from the anonymous reviewers that helped to improve this paper.

References

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

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. AAAI press.

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

Edelkamp, S.; and Schrödl, S. 2012. *Heuristic Search*. San Francisco: Morgan Kaufmann.

Fereday, G.; and Hansen, E. 2025. A Bucket-Based Priority Queue for Bounded-Suboptimal and Anytime A* Search. In *Proc. of the 18th International Symposium on Combinatorial Search*, 56–64. AAAI press.

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.

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

Lermen, M.; and Reinert, K. 2000. The Practical Use of the A* Algorithm for Exact Multiple Sequence Alignment. *Journal of Computational Biology*, 7: 655 – 671.

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.