Open List Implementations in A*

Remo Christen, Florian Pommerening, Malte Helmert

University of Basel, Switzerland
{remo.christen, florian.pommerening, malte.helmert}@unibas.ch

Abstract

A* is the most important algorithm in optimal heuristic search. For its implementation, details beyond the scope of textbook descriptions become relevant. One such detail is how A* deals with duplicate states. A common approach is to modify a state's open list entry when a cheaper path to that state is found, updating its key. A binary heap often serves as the underlying data structure despite its suboptimal asymptotic runtime. More complex structures with better asymptotic runtime such as Fibonacci heaps are said to not pay off. A wide-spread alternative is to delay duplicate detection until expansion, allowing multiple entries for the same state on the open list. So far, the effect of these choices has not been studied in isolation for domain-independent planning. We compare early duplicate detection using a binary heap, Fibonacci heap, pairing heap, or bucket priority queue to delayed duplicate detection in order to evaluate whether early duplicate detection can be worth it.

Introduction

From its inception in 1968 to a fundamental concept in any artificial intelligence textbook today, A* (Hart, Nilsson, and Raphael 1968) has served as the backbone of optimal heuristic search. It manages a priority queue of search nodes (open list). Each node consists of a state s, the cost with which s was reached (g), the node from which s was reached (parent pointer), and a lower bound on the cost from s to a goal (h). A* explores nodes from the open list in order of increasing values of $f = g + h$. While the details of the algorithm's description have remained similar over the years, they do not agree on all aspects. One interesting question is how to handle the case where a cheaper path to a state that is already on the open list is found.

Because it is sufficient to remember the cheaper of the two paths, typical presentations of A* (e.g., Hart, Nilsson, and Raphael 1968; Pearl 1984; Russell and Norvig 2009; Edelkamp and Schroedl 2011) update the existing open list entry with the new g- and thus f-value. This ensures that the open list contains at most one entry per state at any given time, avoiding duplicate entries[1] upon insertion.

[1]Duplicates are search nodes that refer to the same state. Other parts of the node, such as g-value or parent pointer, may differ.

Updating an existing open list entry decreases its f-value and thus increases its priority. This mechanism can inform the choice of the open list data structure. Simple binary heaps are often used, but they do not have a constant-time `decrease_key` operation to update an entry's priority. Other heap implementations such as Fibonacci heaps support this operation in constant time but incur a memory overhead and additional code complexity. Folk wisdom says that more complex heaps do not pay off, even though the suboptimal asymptotic runtime of `decrease_key` means that A* as a whole also has suboptimal worst-case performance. To the best of our knowledge this has not been studied in the context of domain-independent planning (Hoffmann 2011), which is the application of A* we consider.

Instead of updating open list entries we can insert duplicates, as Russell and Norvig (2020) do in the most recent edition of their textbook. If the duplicate with higher f-value is popped later, it can be safely ignored since the duplicate with lower f-value must have been popped and expanded earlier.[2] Thus, duplicates are not prevented directly at insertion, but only later at expansion. We refer to the former strategy as *early* and to the latter as *delayed duplicate detection*.

While optimality is not affected by this choice (Christen et al. 2025), it may well have an impact on performance. Both variants exist in prominent projects implementing A* search, such as Fast Downward (Helmert 2006) using delayed and HOG2[3] using early duplicate detection.

Burns et al. (2012) previously studied implementation details of A* on the sliding tile problem. Among other optimizations, they compare early and delayed duplicate detection. They conclude that early duplicate detection does not pay off, because it requires an additional hash table lookup operation to find a given state on the open list. In their setting, generating a successor state is much faster than a hash table lookup, but they hypothesize that the trade-off would be different when state generation is more expensive such as in domain-independent planning. Their other recommendations are to store states in a packed representation, use pool allocations to reserve memory for both states and search nodes, and avoid virtual function calls with C++ templates.

[2]Russell and Norvig (2020) technically still expand the duplicate, but add none of its successors to the open list.

[3]https://github.com/nathansttt/hog2, accessed on 2026-05-28.

Burns et al. also investigate intrusive data structures that include additional data in the search node to avoid wrapper data structures. They evaluate using an intrusive pointer for the next closed list entry to avoid a wrapper class consisting of a pointer to the search node and one to the next element, thereby saving one pointer per node. Their experiments show only a mild benefit with added complexity, so they do not recommend this optimization. When using early duplicate detection, intrusive pointers can also be used to identify a node's position on the open list during a `decrease_key` operation. We evaluate this use later.

Finally, Burns et al. recommend to group the open list into buckets indexed by f and g value in a type of *2-level bucket priority queue* (Dial 1969). Fereday and Hansen (2025) recently explored such bucket priority queues for use in anytime and bounded-suboptimal A* variants. They exploit the bucket structure inside the algorithms in a way that is not directly applicable to plain A*, do not consider heap structures other than binary heaps, and do not evaluate on planning benchmarks. Costa, Castro, and de Freitas (2025) also recently studied bucket priority queues and other heap data structures for Dijkstra's search in purely theoretical work. Thus neither of these address the hypothesis by Burns et al., which we aim to do.

Fast Downward's implementation aligns well with the recommendations by Burns et al. For one, it stores states in a packed representation with pooled memory allocation. This data structure acts as a hash set, assigning a unique ID to each state while search nodes (parent pointer, creating operator, g-value, and state ID) are stored in another pooled data structure indexed by state IDs. While this means that there is exactly one node for each state, the open list only stores the node's priority and state ID, so it may still contain multiple entries for the same state. Fast Downward uses delayed duplicate detection with a 2-level bucket priority queue, so intrusive data structures are not needed. Virtual function calls exist to access domain-level details, but the open list is templated to avoid such calls. The main drawback of early duplicate detection identified by Burns et al., the costly hash table lookup, Fast Downward requires anyway to access state IDs. Early duplicate detection could thus be beneficial, but the question which data structure to use remains.

We try answering this question by first testing if heap implementations with more efficient `decrease_key` operations pay off compared to a binary heap with early duplicate detection. We compare binary heaps to Fibonacci heaps (Fredman and Tarjan 1987) and pairing heaps (Fredman et al. 1986) as well as bucket priority queues, to which we add an efficient `decrease_key` operation, and analyze their speed, memory usage, and effect on the number of expansions. We then compare early to delayed duplicate detection to see whether detecting duplicates early is beneficial in automated planning.

Comparing Heap Implementations

Early duplicate detection with a binary heap did not pay off in the experiments by Burns et al. since a comparatively expensive hash table lookup was necessary in their setting. However, there is another possible reason for a binary heap to perform worse. Updating an element in the heap uses the `decrease_key` operation, which has a worst case runtime of $O(\log n)$ for binary heaps. Fibonacci heaps take constant time (Fredman and Tarjan 1987) and Pairing heaps take a sub-logarithmic time in $O(2^{2\sqrt{\log \log n}})$ (Pettie 2005).

To analyze whether an advanced heap implementation can improve performance, we first try to find which heap implementation is best. We implement an alternative open list in Fast Downward, using either a custom implementation of binary heaps, or the implementation of Fibonacci or Pairing heaps from the Boost libraries.[4] The open list replaces Fast Downward's default bucket priority queue and we adapt the A* implementation to use `decrease_key` operations instead of pushing duplicate entries to the open list. As with the baseline implementation, the open list entries consist of only the state ID and its priority. For the latter, we use tuples of f- and h-value to break ties in favor of lower h-values.

We evaluate all three variants with the LM-cut heuristic (Helmert and Domshlak 2009) and with the zero heuristic as blind search on 1847 STRIPS instances across 66 domains from the classical optimal tracks of all past international planning competitions.[5] Each configuration ran on a single core of an Intel Xeon Silver 4114 2.2 GHz Processor with a CPU time limit of 1800 s and a memory limit of 3.5 GiB. Source code, benchmarks, and experiment data are available online (Christen, Pommerening, and Helmert 2026). A summary of the results discussed in the following sections is shown in Tables 2 and 3.

Even though all three versions break ties in the f-value in favor of lower h-value, there is still ambiguity when both f and h are identical. The experiments show a noticeable difference in the number of expanded states in the last f-layer between the different heap implementations both with and without a heuristic. The difference is within an order of magnitude except for the `openstacks` domains where Fibonacci heaps perform up to 55 times more expansions.

We expected this to be random but a Wilcoxon signed rank test (Wilcoxon 1945), with ties handled according to Pratt (1959), on commonly solved instances of the blind search shows that the pairing heap has significantly better tie-breaking behavior than the other two heaps, with $p < 10^{-4}$ for the binary and $p < 10^{-2}$ for the Fibonacci heap. Between these two heaps, there is a trend in favor of the Fibonacci heap but the significance is less clear ($p = 0.06$). This does not translate to better coverage, though. Due to better memory efficiency, blind search with the binary heap solves 14 more tasks than with the pairing heap and 22 more than with the Fibonacci heap. Using LM-cut, the binary and pairing heaps perform the same while searching with the Fibonacci heap solves 7 fewer tasks.

Tie-breaking in A* has been investigated in detail before (Asai and Fukunaga 2017; Corrêa, Pereira, and Ritt 2018) and even though we think it is interesting that the tie-breaking strategies have a significant impact, our goal is not to evaluate this effect. Instead we are interested in the perfor-

[4] https://www.boost.org/, accessed on 2026-05-28.

[5] https://www.icaps-conference.org/competitions/, accessed on 2026-05-28.

mance independent of tiebreaking so we control for it by fixing one specific order: we introduce an additional integer to the stored hash key and the search node. This running number is assigned to each node upon creation. Breaking ties on it makes the tiebreaking unique and identical for all heaps, roughly corresponding to FIFO among nodes with equal f- and h-values. We picked FIFO instead of LIFO as this is more comparable to Fast Downward's existing search. Historically, Fast Downward uses FIFO among otherwise equal nodes because this order was more useful for other search algorithms which share the open list implementation of A*.

With all three heap implementations, fixing the tiebreaking leads to significantly more expansions ($p < 10^{-18}$). However, since all methods now perform exactly the same expansions, we can attribute differences in time and memory usage exclusively to heap internals.

With blind search, all failures are due to hitting the memory limit. Since the binary heap uses the least memory for any task using at least 55 MiB (by a factor of up to 2), it consequently also achieves higher coverage than the Fibonacci and pairing heap by 16 and 12, respectively (see Table 2). For search with LM-cut, where time is the primary reason for failure, coverage is very similar despite the pairing heap being significantly faster than both binary and Fibonacci (with $p < 10^{-6}$ and $p < 10^{-4}$).

In conclusion, out of the three heap structures, the binary heap solves most tasks due to better memory efficiency. The pairing heap has an advantage in the more time-constrained LM-cut setting, but not enough to improve coverage. The Fibonacci heap is also slightly faster than the binary heap, but uses the most memory. Since memory is an important factor in the success of the binary heap, even when using a comparatively expensive heuristic, we take a closer look at the memory composition of A* in Fast Downward.

Memory Usage

We measure the memory usage of Fast Downward in two ways. First we use the reported amount of maximal heap memory used by the process, which gives an upper bound of the *total* memory usage. We also compute lower bound size estimates of individual large data structures, in particular for

- The static memory requirement s_{static} of the planner: we measure this as the used heap memory after reading in the task but before starting the search. This varies with instance size but is between 40 and 108 MiB for all instances that we can solve.
- A hash set containing state IDs that is part of the data structure for associating a unique key with each state: its size is $s_{\text{IDs}} = 8c_{\text{IDs}}$ bytes, where c_{IDs} is the capacity of the underlying vector. Upon becoming full, the vector's capacity is doubled, making half its entries empty. We denote the percentage of present entries the *load factor*; it ranges from 0.4 to 1 with an average of 0.67.
- The pool of packed state data containing all generated states: its size is $s_{\text{s-pool}} = 8c_{\text{s-pool}} + 8192n_{\text{s-pool}}$ bytes, where $c_{\text{s-pool}}$ and $n_{\text{s-pool}}$ are the capacity and the size of the underlying allocator. It stores states in segments of 8192 bytes. The size of a state, and thus the number of states in one segment, varies by instance.
- The pool of search nodes: its size is $s_{\text{n-pool}} = 8c_{\text{n-pool}} + 8192n_{\text{n-pool}}$ bytes, analogous to $s_{\text{s-pool}}$. Each node requires 16 bytes, so 512 of them are stored in each entry.
- A heuristic cache that Fast Downward uses to avoid recomputing heuristic values: if a heuristic is used, its size is $s_{\text{h-pool}} = 8c_{\text{h-pool}} + 8192n_{\text{h-pool}}$ bytes with 2048 4-byte entries in each 8192-byte segment. If no heuristic is used, its size is $s_{\text{h-pool}} = 0$ bytes.
- The open list s_{open}: its memory requirement varies by implementation. For the binary heap, the underlying data structure again is a pool allocator using $s_{\text{b-pool}} = 8c_{\text{b-pool}} + 8192n_{\text{b-pool}}$ bytes with 512 16-byte entries per segment. Additionally, we store an integer (4 bytes) in the search node as an intrusive pointer to the heap position of this state. The number of required segments $n_{\text{b-pool}}$ depends on the maximal number of states that are on the open list at any one time, which we log during the search. For the Fibonacci and pairing heap, the memory management is handled by Boost and we only measure their sizes indirectly (see below).

Comparing these lower bounds to the upper bound of measured heap memory for the binary heap shows that the above formulas account for 97% of the memory usage on average. For the other two heap implementations, we get a rough estimate of their memory requirements by subtracting $s_{\text{static}} + s_{\text{IDs}} + s_{\text{s-pool}} + s_{\text{n-pool}} + s_{\text{h-pool}}$ from the measured heap memory. For the Fibonacci heap this results in over 80 bytes effectively used for each heap entry and for the pairing heap in over 64 bytes. Additionally, theses heaps need 8 bytes per state to store the heap position in the search node as an intrusive pointer (see Table 1).

In our experiments, we use the actual vector capacities and sizes of the involved data structures to compute the bounds. To get a better feeling for the memory usage, we can further approximate them with the number N of unique states generated, the load factor $L = N/c_{\text{IDs}}$ of the hash set, and the size S of a single state in bytes. Assuming $S \leq 16$, we can drop the terms $8c_x$ for all pool allocators x as they contribute at most 4% to the overall size, even if the capacity of the pool is twice its size. The memory usage of a configuration with a heuristic then is roughly

$$s_{\text{static}} + (20 + 8/L + S)N + s_{\text{open}}.$$

The size s_{open} of the open list depends on its maximal size O at any point during the search, the size p_{open} of the intrusive pointer used in the search node to identify the node's position, and the size h_{open} required for each heap node on average

$$p_{\text{open}}N + h_{\text{open}}O.$$

Table 1 makes it clear that the more advanced heap structures require much more memory. Even though the size per state only increases by 4 from using a pointer rather than an integer to identify heap locations, the space requirement per heap node increases drastically. If a large fraction of states reside on the open list, this can dominate the memory usage

	Binary	Fibonacci	Pairing	Bucket
p_{open}	4	8	8	4
h_{open}	16	≥ 80	≥ 64	≥ 4

Table 1: Size of intrusive pointers into the heaps (p_{open}) and the memory required per heap node (h_{open}), both in bytes.

where a binary heap with 16 bytes per heap entry will use less than half of the available memory (20 bytes including the intrusive pointer, the same amount as the search nodes and the heuristic cache combined). We thus look at bucket priority queues as a potential way to safe memory and time.

Bucket Priority Queues

For better performance Burns et al. recommend using 2D bucket priority queues (Dial 1969), a data structure indexed by f- and g-value where each bucket is a stack. State IDs can be inserted in their respective bucket in (amortized) constant time and do not require additional information to store alongside the heap entry. To remove the entry with the smallest key, the data structure keeps track of the non-empty bucket with the smallest key and simply pops an element from the respective bucket. The only non-constant time operation is updating the pointer to the minimal bucket once a bucket becomes empty (Burns et al. 2012).

Fast Downward uses such a bucket priority queue by default, however, it uses dequeues instead of stacks for the buckets for a FIFO order inside the bucket compared to the LIFO order provided by a stack. Without further modification, Dial's bucket priority queue has large areas of unused memory when f-values are sparse, which is common in some planning domains. Fast Downward thus uses an ordered map (usually implemented as a red-black tree) indexed with $\langle f, h \rangle$ tuples instead of maintaining a 2D-array of buckets. This gives up constant time access to the buckets in order to save memory.

Fast Downward's bucket priority queue does not have a `decrease_key` operation but this can be easily added: we introduce an additional integer in the search node to identify the position inside its bucket. When decreasing a key, we remove it from its bucket and add it to the end of another.

We consider two strategies for removing an entry from a bucket. The efficient implementation swaps the element to be removed with the start of its bucket (updating the integers pointing to the positions of both nodes), then removes it from the start. The inefficient implementation shifts all elements before the removed node to maintain the order within the bucket. While expensive, this strategy has the same expansion order as our heap-based implementations in their tie-breaking-controlled (FIFO) variants.

To understand the difference between the two removal strategies, we compare them directly. While swapping an element to the front of its bucket breaks FIFO order, this only has an effect on the number of expansion for 78 tasks and the effect is an increase within 0.5%. We can thus safely disregard this difference and compare the efficient bucket implementation to the heap-based open lists.

The binary heap uses more memory than the bucket priority queue on most tasks but only about 2% more on average. Both use a 4 byte integer per state to identify an open list entry position. The bucket priority queue only stores 4 bytes for the state ID in each heap entry, compared to the 16 bytes used by the binary heap. However, the map itself has an overhead, in particular if there are a large number of buckets. Overallocation inside each bucket and the overhead of the red-black tree which scales with the amount of distinct $\langle f, h \rangle$ buckets can further increase the memory requirement. If we estimate the memory effectively used for each heap entry on average, analogous to our estimations for the Fibonacci and pairing heaps, we see they require between the ideal 4 and up to 24 bytes per state.

With blind search, the bucket queue is faster by up to 40% on 528 tasks and only slower on 40 tasks. With LM-cut, where a large part of the time is spent on heuristic computations, the difference between the run times becomes insignificant. The bucket queue solves 5 additional tasks with blind search and 2 with LM-cut (see Table 2).

We can also compare the bucket priority queue to the binary heap without a fixed tie-breaking order. We have seen before that this tie-breaking order is not ideal, and enforcing it requires additional memory both per state and per heap node. With its default order, the binary heap outperforms the bucket priority queue, solving 6 additional tasks with both blind search and LM-cut. However, this seems to be an effect mostly of the better tiebreaking and less of the more memory efficient data structure. For our comparison to delayed duplicate detection, we thus stick with the bucket priority queue, as it matches Fast Downward's current implementation most closely.

Delayed Duplicate Detection

Now that we analyzed different early duplicate detection alternatives, we compare to delayed duplicate detection to determine if early duplicate detection pays off in domain-independent planning.

We use the bucket priority queue enhanced with an efficient `decrease_key` operation as the representative configuration for the early duplicate detection approach. Since delayed duplicate detection employs FIFO by default, the efficient bucket priority queue does again not match the tiebreaking exactly, but with the same 0.5% deviation mentioned in the previous section.

In a direct comparison, search time does not vary significantly ($p > 0.09$) with no relative difference greater than 60%, but memory usage for the delayed queue is significantly lower for tasks of sufficient size ($> 55\,\text{MiB}$; $p < 10^{-35}$). For smaller tasks, early duplicate detection often has a constant advantage of around 30 bytes, skewing the significance analysis. This shows that early duplicate detection's theoretical memory advantage of having to store fewer entries on the open list is more than offset by fact that additional pointers into the open list have to be maintained, even with this more compact structure compared to heap implementations (see Table 1). We can understand this by seeing that delayed duplicate detection, in the worst observed case, stores $2\,000\,000 \cdot 4$ bytes of additional entries on the open

	Binary	Fibo.	Pairing	Bucket	Delayed
Blind	711	695	699	716	**720**
LM-cut	962	959	961	**964**	963

Table 2: Coverage summary for early duplicate detection with different heaps (enforced FIFO tiebreaking), early duplicate detection with a bucket queue (default near-FIFO tiebreaking), and delayed duplicate detection.

list, which amounts to less than 8 MiB and less than 2% of the total memory required to solve the task.

This result is also consistent with the prediction of our memory estimation. If we compare the estimated memory usage on the same instance with the same tiebreaking, both early and delayed duplicate detection require $s_{\text{static}} + (20 + 8/L + S)N$ bytes of memory for the static part of the program, the pool of search nodes, the hash set containing state IDs, and the pool of packed state data but they differ in our estimate for the size of the open list $s_{\text{open}} = p_{\text{open}}N + h_{\text{open}}O$. With early duplicate detection, $p_{\text{open}} = 4$ (see Table 1) and h_{open} is roughly 4 as well. For delayed duplicate detection, $p_{\text{open}} = 0$ as we do not need an intrusive pointer, and h_{open} again is roughly 4. The number of generated states N is the same in both cases but the maximal open list size O could be different. Thus, we would expect that delayed duplicate detection uses more memory than early duplicate detection if $4N + 4O_{\text{early}} < 4O_{\text{delayed}}$ or equivalently if $O_{\text{delayed}} - O_{\text{early}} > N$. That is, we expect early duplicate detection to pay off regarding memory if the maximal number of open list entries of delayed duplicate detection is higher by at least the number of generated states N. This can only happen if many alternative paths to states are discovered before they are expanded (at least 2 for every state on the open list, because $O_{\text{early}} < N$). In our experience the number of duplicates is typically only a small fraction of O, so this is a very specific corner case.

Conclusion

We have evaluated the performance of different open list implementations for A* with the zero and LM-cut heuristic. Tables 2 and 3 recap coverage, time, and memory results.

Simple binary heaps require less memory than Fibonacci and pairing heaps. This advantage leads to better coverage, despite higher theoretical and practical runtime.

Based on Burns et al.'s recommendation and supported by Fast Downward' default, we evaluated a bucket-based open list with added `decrease_key` operation as a possible optimization over heaps. After disregarding tie-breaking effects, this does indeed speed up search while requiring less memory compared to binary heaps. It does however not offer an improvement over Fast Downward's baseline delayed duplicate detection.

Beyond the experimental evaluation, we analyzed Fast Downward's memory composition and derived a simple formula to estimate memory usage in terms of basic search parameters. This insight can be used to compare implementations on a theoretical level.

Time for Blind
Delayed $<$ Bucket $<$ Pairing $<$ Fibonacci $<$ Binary
Time for LM-cut
Pairing $<$ Binary $\sim$ Bucket $\sim$ Delayed $\sim$ Fibonacci
Memory for both Blind and LM-cut
Delayed $<$ Bucket $<$ Binary $<$ Pairing $<$ Fibonacci

Table 3: Time and memory summary of the configurations from Table 2, where $X < Y$ means that X uses significantly ($p < 0.05$) fewer resources than Y on tasks solved by both, and $X \sim Y$ that there is no significant difference. The relations are transitive and memory only considers tasks requiring at least 55 MiB.

Finally, we can refute the hypothesis posed by Burns et al: detecting duplicates early does not benefit search in domain-independent planning.

Acknowledgments

This work was funded by the Swiss National Science Foundation (SNSF) as part of the project "Unifying the Theory and Algorithms of Factored State-Space Search" (UTA).

References

Asai, M.; and Fukunaga, A. 2017. Tie-Breaking Strategies for Cost-Optimal Best First Search. *Journal of Artificial Intelligence Research*, 58: 67–121.

Burns, E.; Hatem, M.; Leighton, M. J.; and Ruml, W. 2012. Implementing Fast Heuristic Search Code. In Borrajo, D.; Felner, A.; Korf, R.; Likhachev, M.; Linares López, C.; Ruml, W.; and Sturtevant, N., eds., *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SoCS 2012)*, 25–32. AAAI Press.

Christen, R.; Pommerening, F.; Büchner, C.; and Helmert, M. 2025. A Formalism for Optimal Search with Dynamic Heuristics. In Lipovetzky, N.; Sardina, S.; Harabor, D.; and Ramirez, M., eds., *Proceedings of the Thirty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2025)*, 30–39. AAAI Press.

Christen, R.; Pommerening, F.; and Helmert, M. 2026. Code, benchmarks, and experiment data for the SoCS 2026 paper "Open List Implementations in A*". https://doi.org/10.5281/zenodo.20480289.

Corrêa, A. B.; Pereira, A. G.; and Ritt, M. 2018. Analyzing Tie-Breaking Strategies for the A^* Algorithm. In Lang, J., ed., *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI 2018)*, 4715–4721. IJCAI.

Costa, J.; Castro, L.; and de Freitas, R. 2025. Exploring monotone priority queues for Dijkstra optimization. *RAIRO - Operations Research*, 59(5): 2419–2436.

Dial, R. B. 1969. Algorithm 360: shortest-path forest with topological ordering [H]. *Communications of the ACM*, 12(11): 632–633.

Edelkamp, S.; and Schroedl, S. 2011. *Heuristic Search: Theory and Applications*. Morgan Kaufmann.

Fereday, G. M.; and Hansen, E. A. 2025. A Bucket-Based Priority Queue for Bounded-Suboptimal and Anytime A^* Search. In Likhachev, M.; Rudová, H.; and Scala, E., eds., *Proceedings of the 18th Annual Symposium on Combinatorial Search (SoCS 2025)*, 56–64. AAAI Press.

Fredman, M. L.; Sedgewick, R.; Sleator, D. D.; and Tarjan, R. E. 1986. The Pairing Heap: A New Form of Self-Adjusting Heap. *Algorithmica*, 1: 111–129.

Fredman, M. L.; and Tarjan, R. E. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3): 596–615.

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

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26: 191–246.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 162–169. AAAI Press.

Hoffmann, J. 2011. Everything You Always Wanted to Know About Planning (But Were Afraid to Ask). In Bach, J.; and Edelkamp, S., eds., *Proceedings of the 34th Annual German Conference on Artificial Intelligence (KI 2011)*, volume 7006 of *Lecture Notes in Artificial Intelligence*, 1–13. Springer-Verlag.

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Pettie, S. 2005. Towards a final analysis of pairing heaps. In *Proceedings of the Fourty-Sixth Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005)*, 174–183. IEEE Computer Society.

Pratt, J. W. 1959. Remarks on Zeros and Ties in the Wilcoxon Signed Rank Procedures. *Journal of the American Statistical Association*, 54(287): 655–667.

Russell, S.; and Norvig, P. 2009. *Artificial Intelligence: A Modern Approach*. Pearson.

Russell, S.; and Norvig, P. 2020. *Artificial Intelligence: A Modern Approach*. Pearson.

Wilcoxon, F. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6): 80–83.