Implementation notes

From Amit's Thoughts on Pathfinding

Sketch

The A* algorithm, stripped of all the code, is fairly simple. There are two sets, OPEN and CLOSED. The OPEN set contains those nodes that are candidates for examining. Initially, the OPEN set contains only one element: the starting position. The CLOSED set contains those nodes that have already been examined. Initially, the CLOSED set is empty. Graphically, the OPEN set is the "frontier" and the CLOSED set is the "interior" of the visited areas. Each node also keeps a pointer to its parent node so that we can determine how it was found.

There is a main loop that repeatedly pulls out the best node `n` in OPEN (the node with the lowest `f` value) and examines it. If `n` is the goal, then we're done. Otherwise, node `n` is removed from OPEN and added to CLOSED. Then, its neighbors `n'` are examined. A neighbor that is in CLOSED has already been seen, so we don't need to look at it (1). A neighbor that is in OPEN is scheduled to be looked at, so we don't need to look at it now. Otherwise, we add it to OPEN, with its parent set to `n`. The path cost to `n'`, `g(n')`, will be set to `g(n) + movementcost(n, n')`.

I go into a lot more detail here[1], with interactive diagrams.

(1) I'm skipping a small detail here. You do need to check to see if the node's `g` value can be lowered, and if so, you re-open it.

```
OPEN = priority queue containing START
CLOSED = empty set
while lowest rank in OPEN is not the GOAL:
  current = remove lowest rank item from OPEN
  add current to CLOSED
  for neighbors of current:
```

```
    cost = g(current) + movementcost(current, neighbor)
    if neighbor in OPEN and cost less than g(neighbor):
      remove neighbor from OPEN, because new path is better
    if neighbor in CLOSED and cost less than g(neighbor): (2)
      remove neighbor from CLOSED
    if neighbor not in OPEN and neighbor not in CLOSED:
      set g(neighbor) to cost
      add neighbor to OPEN
      set priority queue rank to g(neighbor) + h(neighbor)
      set neighbor's parent to current

reconstruct reverse path from goal to start
by following parent pointers
```

(2) This should never happen if you have an consistent admissible heuristic. However in games we often want inadmissible heuristics[2].

See Python and C++ implementations here[3].

Connectivity

If your game has situations in which the start and goal are not connected at all by the graph, A* will take a long time to run, since it has to explore every node connected from the start before it realizes there's no path. Calculate the Connected Components[4] first and only use A* if the start and goal are in the same region.

Performance

The main loop of A* reads from a priority queue, analyzes it, and inserts nodes back into the priority queue. In addition, it tracks which nodes have been visited. To improve performance, consider:

- Can you decrease the size of the graph? This will reduce the number of nodes that are processed, both those on the path and those that don't end up on the final path. Consider navigation meshes instead of grids. Consider hierarchical map representations.

- Can you improve the accuracy of the heuristic? This will reduce the number of nodes that are not on the final path. The closer the hheuristic to the actual path length (not the distance), the fewer nodes A* will explore. Consider these heuristics for grids. Consider ALT (A*, Landmarks, Triangle Inequality) for graphs in general (including grids).
- Can you make the priority queue faster? Consider other data structures for your priority queue. Consider processing nodes in batches, as fringe search[5] does. Consider approximate sorting.
- Can you make the heuristic faster? The heuristic function is called for every open node. Consider caching its result. Consider inlining the call to it.

For grid maps, see these suggestions[6].

Source code and demos

Demos

These demos run in your browser:

- **I have written an introduction to A*[7] with interactive demos.**
- I have written Flash demos for square grids[8], hexagonal grids[9], and triangular grids[10]. The Actionscript 3 code for these demos is available here[11] (see Pathfinder.as for the main algorithm, and Graph.as for the abstract interface to graphs).
- This Javascript library and demo[12] is very fast, works on graphs, and also includes bidirectional A*.
- This site[13] has demos of A*, Breadth-First Search, Dijkstra's Algorithm, and Greedy Best-First Search on road maps (not grids)
- This Javascript library and demo[14] has lots of optimizations for grid maps.
- This Javascript A* demo[15] lets you change the road weight; source code on github[16], using the MIT open source license. The calculation is interruptible so you can run a few iterations per frame.

- James Macgill's Java applet[17].
- This interactive demo lets you choose A* or Dijkstra's Algorithm[18].
- This demo[19] is nice and has Javascript source code[20].
- This demo[21] is nice and also has code available.
- Another Javascript demo[22]
- This page[23] describes Jump Point Search and also has an online demo.
- This Actionscript A* tutorial[24] has a demo near the end.
- This demo[25] is in Javascript with readable source but I don't know the license for the source code.
- This A* demo[26] and this Jump Point Search demo[27] use Unity.

Code

If you're using C++, be sure to look at Recast[28], from Mikko Mononen[29].

If you're planning to implement graph search yourself, **here's my guide to Python and C++ implementations**[30]. My code is much shorter and simpler than most other A* code I have found, and works on any graphs, not only on grids.

}

There's a list of pathfinding implementations in this gist[31]. I have not looked at or evaluated them.

Set representation

What's the first thing you'll think of using for the OPEN and CLOSED sets? If you're like me, you probably thought "array". You may have thought "linked list", too. There are many different data structures we can use, but to pick one we should look at what operations are needed.

There are three main operations we perform on the OPEN set:

- **Remove-best**: the main loop finds the best node and removes it from the set
- **Membership**: the neighbor loop will check whether a node is in the set
- **Insert**: the neighbor loop will also insert new nodes

These operations are typical of a priority queue[32].

The choice of data structure depends not only on the operations but on the number of times each operations runs. The membership test runs once for each neighbor for each node visited. Insertion runs once for each node being considered. Remove-best runs once for each node visited. Most nodes that are considered will be visited; the ones that are not are the *fringe* of the search space. When evaluating the cost of operations on these data structures, we need to consider the maximum size of the fringe (F).

In addition, *there's a fourth operation*, which is relatively rare but still needs to be implemented. If the node being examined is already in the OPEN set (which happens frequently), and if its `f` value is better than the one already in the OPEN set (which is rare), then the value in the OPEN set must be adjustment. The adjustment operation involves removing the node (which is not the best `f`) and re-inserting it. These two steps may be optimized into an increase-priority operation that moves the node (this is also called decrease-key).

> Do you really need the priority-adjustment operation? When I wrote this document in 1997 I believed you did. I have since come to believe that you don't always need it, and in many cases you're better off not implementing it. See the "what happens if you don't reprioritize?" paragraph[33] on my newer A* page.

My recommendation: The best generic choice is a binary heap. If you have a binary heap library available, use it. If not, start with unsorted arrays, and switch to binary heaps if you want more performance. If you have more than 10,000 elements in your OPEN set, then consider more complicated structures such as a bucketing system. Read this paper about priority queue

performance for modern CPUs[34].

Unsorted arrays or linked lists

The simplest data structure is an unsorted array or list. Membership test is slow, O(F) to scan the entire structure. Insertion is fast, O(1) to append to the end. Remove-best element is slow, O(F) to scan the entire structure, then O(1) to remove it. Although removing an element from an array is usually O(F), in this case we can swap the best element with the last element, then remove the last element. The increase-priority operation is O(F) to find the node and O(1) to change its value.

Since membership test is common, it's sometimes useful to use a faster data structure for it and used the unsorted array for the rest.

When I wrote this in 1997, the cache behavior wasn't a big concern. The O(F) scans should have good cache behavior, so unsorted arrays would be my second choice after binary heaps, especially for smaller F.

Sorted arrays

Sorted arrays seem like they might be better than unsorted arrays.

If we sort by *key* we can make membership fast using binary search, O(log F). But insertion is slow, O(F) to move all the elements to make space for the new one. Remove-best is also slow, O(F). Increase-priority is fast, O(log F) to find and update the node.

If we sort by *priority* we can make remove-best fast because the best element is always at the end. But insertion is slow, O(F) to move elements to make space. Membership is also slow, O(F). Increase-priority is slow as well, O(F).

I think sorting by priority is better than sorting by key. Use some other data structure to make membership fast.

Binary heaps

A binary heap[35] (not to be confused with a memory heap) is a tree structure that is stored in an array. Unlike most trees, which use pointers to refer to children, the binary heap uses indexing to find children.

In a binary heap, membership is O(F), as you have to scan the entire structure. Insertion is O(log F) and remove-best is O(log F).

The increase-priority operation is tricky, with O(F) to find the node and surprisingly, only O(log F) to increase-priority it. Unfortunately most priority queue libraries don't include this operation. Fortunately, *it's not strictly necessary*, as described in this paper[36]. So I recommend not worrying about it unless you absolutely need to. Instead of increasing priority, insert a new element into the priority queue. You'll potentially end up processing the node twice but that's relatively cheap compared to implementing increase-priority.

In C++, use the priority_queue[37] class, which doesn't have increase-priority, or Boost's mutable priority queue[38], which does. In Python, use the heapq library[39].

You can combine a hash table or indexed array for membership and a priority queue for managing priorities; see the hybrid section below.

As part of my newer A* tutorial[40], I have a complete A* implementation in Python and C++[41] using binary heaps for the priorities and hash tables for the for membership. I *do not implement increase-priority* and explain why in the optimization section. I also have some C# code there.

A variant of the binary heap is a d-ary heap[42], which has more than 2 children per node. Inserts and increase-priority become a little bit faster, but removals become a little bit slower. They may have better cache performance. B-heaps are also worth a look if your frontier is large[43].

I have only used binary heaps and bucket approaches for my own pathfinding projects. If binary heaps aren't good enough, then consider pairing heaps, sequence heaps, or a bucket based approach. The paper Priority Queues and Dijkstra's Algorithm[44] is worth a read if you are unable to shrink the graph and need a faster priority queue.

Sorted skip lists

Searching an unsorted linked list is slow. We can make that faster if we use a skip list[45] instead of a linked list. With a skip list, membership is fast if you have the sort key: O(log F). Insertion is O(1) like a linked list if you know where to insert. Finding the best node is fast if the sort key is `f`, O(1), and removing a node is O(1). The increase-priority operation involves finding a node, removing it, and reinserting it.

If we use skip lists with the map location as the key, membership is O(log F), insertion is O(1) after we've performed the membership test, finding the best node is O(F), and removing a node is O(1). This is better than unsorted linked lists in that membership is faster.

If we use skip lists with the `f` value as the key, membership is O(F), insertion is O(1), finding the best node is O(1), and removing a node is O(1). This is no better than sorted linked lists.

Indexed arrays

If the set of nodes is finite and reasonably sized, we can use a direct indexing structure, where an index function `i(n)` maps each node `n` to an index into an array. Unlike the unsorted and sorted arrays, which have a size corresponding to the largest size of OPEN, with an indexed array the array size is always `max(i(n))` over all `n`. If your function is dense (*i.e.*, there are no indices unused), then `max(i(n))` will be the number of nodes in your graph. Whenever your map is a grid, it's easy to make the function dense.

Assuming `i(n)` is O(1), membership test is O(1), as we merely have to check whether `Array[i(n)]` contains any data. Insertion is O(1), as we set `Array[i(n)]`. Find and remove best is O(numnodes), since we have to search the entire structure. The increase-priority operation is O(1).

Hash tables

Indexed arrays take up a lot of memory to store all the nodes that are *not* in the OPEN set. An alternative is to use a hash table, with a hash function `h(n)` that maps each node `n` into a hash code. Keep the hash table twice as big as N to keep the chance of collisions low. Assuming `h(n)` is O(1), membership test is expected O(1), insertion is expected O(1), and remove best is O(numnodes), since we have to search the entire structure. The increase-priority operation is O(1).

Hash tables are best for set membership but not for managing priorities. In my my newer A* tutorial[46], I use hash tables for membership and binary heaps for priorities[47]. I combined the OPEN and CLOSED sets into one, which I call VISITED.

Splay trees

Heaps are a tree-based structure with expected O(log F) time operations. However, the problem is that with A*, the common behavior is that you have a low cost node that is removed (causing O(log F) behavior, since values have to move up from the very bottom of the tree) followed by low cost nodes that are added (causing O(log F) behavior, since these values are added at the bottom and bubble up to the very top). The *expected case* behavior of heaps here is equivalent to the *worst case* behavior. We may be able to do better if we find a data structure where *expected case* is better, even if the *worst case* is no better.

Splay trees are a self adjusting tree structure. Any access to a node in the tree tends to bring that node up to the top. The result is a "caching" effect, where rarely used nodes go to the bottom and don't slow down operations. It doesn't matter how big your splay tree is, because your operations are only as slow as your "cache size". In A*, the low cost nodes are used a lot, and the high cost nodes aren't used for a long time, so those high cost nodes can move to the bottom of the tree.

With splay trees, membership, insertion, remove-best, and increase-priority are all expected O(log F), worst case O(F). Typically however, the caching keeps the worst case from occurring. Dijkstra's Algorithm and A* with an underestimating heuristic however have some peculiar characteristics that may keep splay trees from being the best. In particular, `f(n') >= f(n)` for nodes `n` and neighboring node `n'`. When this happens, it may be that the insertions all occur on one side of the tree and end up putting it out of balance. I have not tested this.

Bucketing

A bucketing strategy (Radix heap[48] or bucket queue[49] is useful when the priorities in the priority queue are restricted in range instead of being arbitrary numbers. There are often situations where knowing the range of values can lead to faster algorithms, such as sorting on arbitrary values being O(N log N) time, whereas sorting a fixed range of numbers being O(N) time using bucket or radix sorts.

Consider Dijkstra's Algorithm: the frontier has priorities `p` through `p+k` where `k` is the largest movement cost. For example if your movement costs are 1, 2, or 3, then everything in the frontier has priority `p`, `p+1`, `p+2`, `p+3`. Use 4 buckets, one for each bucket. There's no need to sort within a bucket because the priorities are all the same. And there's no need to sort the buckets. That means insertion is O(1) and remove-best is O(1).

Note that Breadth First Search can be viewed as using a bucketed priority queue with exactly two buckets.

With A* it's a little more complicated because you also have to look at the effect of the heuristic on the priority. There will be more buckets than with Dijkstra's Algorithm.

HOT queues

HOT Queues[50] are a variant of bucket queues that accept a larger range of values. Instead of each bucket having *exactly* one priority, each bucket has a range of priorities. Since we're only removing elements from the top bucket, only the top bucket has to be ordered.

HOT queues make the topmost bucket use a binary heap. All other buckets are unsorted arrays. Membership test is O(F) because we don't know which bucket the node is in. Insertion and remove-best in the top bucket are O(log (F/K)), for K buckets. Insertion into other buckets is O(1), and remove-best never runs on other buckets. If the top bucket empties, then we need to convert the next bucket, an unsorted array, into a binary heap. It turns out this operation ("heapify") can be run in O(F/K) time. The increase-priority operation is best treated as a O(F/K) removal followed by an O(log (F/K)) or O(1) insertion.

In A*, many of the nodes we put into OPEN we never actually need. HOT Queues are a big win because the elements that are not needed are inserted in O(1) time. Only elements that are needed get heapified (which is not too expensive). The only operation that is more than O(1) is node deletion from the heap, which is only O(log (F/K)).

One person reported that HOT queues are as fast as heaps for at most 800 nodes in the OPEN set, and are 20% faster when there are at most 1500 nodes. I would expect that HOT queues get faster as the number of nodes increases.

A cheap variant of a HOT queue is a two-level queue: put good nodes into one data structure (a heap or an array) and put bad nodes into another data structure (an array or a linked list). Since most nodes put into OPEN are "bad", they are never examined, and there's no harm putting them into the big array.

Pairing heaps

Fibonacci heaps are good priority queues for A*, in theory. However, in practice they're not used. A pairing heap[51] can be thought of as a simplified Fibonacci heap. They are said to work well in practice; I have never used them.

Here's the original paper describing them[52].

Soft heaps

A soft heap[53] is a type of heap that gives the nodes in approximately the right order. By approximating, it can provide results faster than a regular heap.

I haven't tried soft heaps. The algorithms described in this paper[54] seem fairly short and straightforward to implement. However this answer on stackoverflow[55] says it's useful in theory but "unlikely to be useful in practice".

For pathfinding in games, we often do not need the *exact* shortest path and usually would prefer to have a reasonably short path computed quickly. So this may be one of the applications where it *is* useful in practice. I don't know yet; if you have studied this data structure, please email me.

Sequence heaps

I haven't looked into them. See Peter Sanders's paper Fast Priority Queues for Cached Memory[56].

"Sequence heaps may currently be the fastest available data structure for large comparison based priority queues both in cached and external memory This is particularly true if the queue elements are small and if we do not need deletion of arbitrary elements or decreasing keys."

Data Structure Comparison

It is important to keep in mind that we are not merely looking for asymptotic ("big O") behavior. We also want to look for a low constant. To see why, consider an algorithm that is O(log F) and another that is O(F), where F is the number of elements in the heap. It may be that on your machine, an implementation of the first algorithm takes 10,000 * log(F) seconds, while an implementation of the second one takes 2 * F seconds. For F = 256, the first would take 80,000 seconds and the second would take 512 seconds. The "faster" algorithm takes more time in this case, and would only start to be faster when F > 200,000.

You cannot merely compare two algorithms. You should also compare the implementations of those algorithms. You also have to know what size your data might be. In the above example, the first implementation is faster for F > 200,000, but if in your game, F stays under 30,000, then the second implementation would have been better.

None of the basic data structures is entirely satisfactory. Unsorted arrays or lists make insertion very cheap and membership and removal very expensive. Sorted arrays or lists make membership somewhat cheap, removal very cheap and insertion very expensive. Binary heaps make insertion and removal somewhat cheap, but membership is very expensive. Splay trees make everything somewhat cheap. HOT queues make insertions cheap, removals fairly cheap, and membership tests somewhat cheap. Indexed arrays make membership and insertion very cheap, but removals are incredibly expensive, and they can also take up a lot of memory. Hash tables perform similarly to indexed arrays, but they can take up a lot less memory in the common case, and removals are merely expensive instead of extremely expensive.

For a good list of pointers to more advanced priority queue papers and implementations, see Lee Killough's Priority Queues page[57].

Hybrid representations

To get the best performance, you will want a hybrid data structure. For my A* code, I used an indexed array for O(1) membership test and a binary heap for O(log F) insertion and O(log F) remove-best. For increase-priority, I used the indexed array for an O(1) test whether I really needed to perform the change in priority (by storing the `g` value in the indexed array), and then in those rare cases that I did need to increase priority, I used the O(F) increase-priority on the binary heap. You can also use the indexed array to store the location in the heap of each node; this would give you O(log F) for increase-priority.

In my newer A* tutorial[58], I use a hash table for membership and a binary heap for priorities[59]. I further simplified by combining OPEN and CLOSED into the same set, which I call Visited.

Interaction with the game loop

Interactive (especially real-time) games introduce requirements that affect your ability to compute the best path. It may be more important to get *any* answer than to get the *best* answer. Still, all other things being equal, a shorter path is better than a longer one.

In general, computing the part of the path close to the starting point is more important than the path close to the goal. The principle of *immediate start*: get the unit moving as soon as possible, even along a suboptimal path, and then compute a better path later. In real-time games, the *latency* of A* is often more important than its *throughput*.

Units can be programmed to follow either their instincts (simple movement) or their brains (a precalculated path). Units will follow their instincts unless their brains tell them otherwise. (This approach is used in nature and also in Rodney Brook's robot architecture.) Instead of calculating all the paths at once, limit the game to finding one path every one, two, or three game cycles. Then let the units start walking according to instinct (which could simply be moving in a straight line towards the goal), and come back later to find paths for them. This approach allows you to even out the cost of pathfinding so that it doesn't occur all at once.

Early exit

It is possible to exit early from the A* main loop and get a partial path. Normally, the loop exits when it finds the goal node. However, at any point before that, it can return a path to the currently best node in OPEN. That node is our best chance of getting to the goal, so it's a reasonable place to go.

Candidates for early exit include having examined some number of nodes, having spent some number of milliseconds in the A* algorithm, or exploring a node some distance away from the starting position. When using path splicing, the spliced path should be given a smaller limit than a full path.

Interruptible algorithm

If few objects need pathfinding services or if the data structures used to store the OPEN and CLOSED sets are small, it can be feasible to store the state of the algorithm, exit to the game loop, then continue where A* left off.

Group movement

Path requests do not arrive evenly distributed. A common situation in a real-time strategy game is for the player to select multiple units and order them to move to the same goal. This puts a high load on the pathfinding system.

See Also: Putting paths together is called path splicing in another section of these notes.

In this situation, it is very likely that the path found for one will be useful for other units. One idea is to find a path `P` from the center of the units to the center of the destinations. Then, use most of that path for all the units, but replace the first 10 steps and the last 10 steps by paths that are found for each individual unit. Unit *i* will receive a path from its starting location to `P[10]`, followed by the shared path `P[10..len(P)-10]`, followed by a path from `P[len(P)-10]` to the destination.

The paths found for each unit are short (approximately 10 steps on average), and the long path is shared. Most of the path is found only once and shared among all the units. However, the user may not be impressed if he sees all the units moving on the same path. To improve the appearance of the system, make the units follow slightly different paths. One way to do this is to alter the paths themselves, by choosing adjacent locations.

Another approach is to make the units aware of each other (perhaps by picking a "leader" unit randomly, or by picking the one with the best sense of what's going on), and only find a path for the leader. Then use a flocking algorithm[60] to make them move in a group.

There are some variants of A*[61] that handle a moving destination, or updated knowledge of the destination. Some of them might be adapted to handle multiple units going to the same destination, by performing A* in reverse (finding a path from the destination to the unit).

Refinement

If the map contains few obstacles, but instead contains terrain of varying costs, then an initial path can be computed by treating terrain as being cheaper than normal. For instance, if grasslands are cost 1, hills are cost 2, and mountains are cost 3, then A* will consider walking through 3 grasslands to avoid 1 mountain. Instead, compute an initial path by treating grasslands as 1,

hills as 1.1, and mountains as 1.2. A* will then spend less time trying to avoid the mountains, and it will find a path quicker. (It approximates the benefits of an exact heuristic.) Once a path is known, the unit can start moving, and the game loop can continue. When spare CPU is available, compute a better path using the real movement costs.

Email me redblobgames@gmail.com, or tweet @redblobgames, or comment:

Endnotes

[1]: https://www.redblobgames.com/pathfinding/a-star/introduction.html
[2]: http://realtimecollisiondetection.net/blog/?p=56
[3]: https://www.redblobgames.com/pathfinding/a-star/implementation.html
[4]: http://en.wikipedia.org/wiki/Connected_component_(graph_theory)
[5]: http://en.wikipedia.org/wiki/Fringe_search
[6]: https://www.redblobgames.com/pathfinding/grids/algorithms.html
[7]: https://www.redblobgames.com/pathfinding/a-star/introduction.html
[8]: http://theory.stanford.edu/~amitp/game-programming/a-star-flash/main-square.swf
[9]: http://theory.stanford.edu/~amitp/game-programming/a-star-flash/main-hexagon.swf
[10]: http://theory.stanford.edu/~amitp/game-programming/a-star-flash/main-triangle.swf
[11]: http://www-cs-students.stanford.edu/~amitp/game-programming/a-star-flash/
[12]: https://github.com/anvaka/ngraph.path
[13]: http://kevanahlquist.com/osm_pathfinding/
[14]: http://mikolalysenko.github.io/l1-path-finder/www/
[15]: http://easystar.nodejitsu.com/demo.html
[16]: https://github.com/prettymuchbryce/easystarjs

[17]: http://www.vision.ee.ethz.ch/~cvcourse/astar/AStar.html
[18]: http://www.growingwiththeweb.com/projects/pathfinding-visualiser/
[19]: http://qiao.github.com/PathFinding.js/visual/
[20]: https://github.com/qiao/PathFinding.js
[21]: http://will.thimbleby.net/a-shortest-path-in-javascript/
[22]: http://www.briangrinstead.com/files/astar/
[23]: http://zerowidth.com/2013/05/05/jump-point-search-explained.html
[24]: http://www.untoldentertainment.com/blog/2010/08/20/introduction-to-a-a-star-pathfinding-in-actionscript-3-as3-2/
[25]: http://cs.williams.edu/~morgan/codeheartjs/examples/pathfinding/play.html
[26]: http://singul4rity.com/misc/astar/
[27]: http://singul4rity.com/2013/04/jump-point-search-unity3d-online-demo/
[28]: http://code.google.com/p/recastnavigation/
[29]: http://digestingduck.blogspot.com/
[30]: https://www.redblobgames.com/pathfinding/a-star/implementation.html
[31]: https://gist.github.com/systemed/be2d6bb242d2fa497b5d93dcafe85f0c
[32]: http://en.wikipedia.org/wiki/Priority_queue
[33]: https://www.redblobgames.com/pathfinding/a-star/implementation.html#optimizations
[34]: https://arxiv.org/pdf/1403.0252.pdf
[35]: http://en.wikipedia.org/wiki/Heap_(data_structure)
[36]: http://www.cs.sunysb.edu/~rezaul/papers/TR-07-54.pdf
[37]: http://en.cppreference.com/w/cpp/container/priority_queue
[38]: http://www.boost.org/doc/libs/1_50_0/doc/html/heap/concepts.html#heap.concepts.mutability
[39]: https://docs.python.org/3/library/heapq.html
[40]: https://www.redblobgames.com/pathfinding/a-star/introduction.html
[41]: https://www.redblobgames.com/pathfinding/a-star/implementation.html
[42]: http://en.wikipedia.org/wiki/D-ary_heap
[43]: https://queue.acm.org/detail.cfm?id=1814327

[44]: http://www3.cs.stonybrook.edu/~rezaul/papers/TR-07-54.pdf
[45]: http://en.wikipedia.org/wiki/Skip_list
[46]: https://www.redblobgames.com/pathfinding/a-star/introduction.html
[47]: https://www.redblobgames.com/pathfinding/a-star/implementation.html
[48]: https://en.wikipedia.org/wiki/Radix_heap
[49]: https://en.wikipedia.org/wiki/Bucket_queue
[50]: http://www.star-lab.com/goldberg/pub/neci-tr-97-104.ps
[51]: http://en.wikipedia.org/wiki/Pairing_heap
[52]: https://www.cs.cmu.edu/~sleator/papers/pairing-heaps.pdf
[53]: https://en.wikipedia.org/wiki/Soft_heap
[54]: http://epubs.siam.org/doi/pdf/10.1137/1.9781611973068.53
[55]: https://stackoverflow.com/a/26126781
[56]: http://algo2.iti.kit.edu/sanders/papers/falenex.ps.gz
[57]: http://www.leekillough.com/heaps/
[58]: https://www.redblobgames.com/pathfinding/a-star/introduction.html
[59]: https://www.redblobgames.com/pathfinding/a-star/implementation.html
[60]: http://www.red3d.com/cwr/boids/
[61]: http://en.wikipedia.org/wiki/Incremental_heuristic_search

---

From Red Blob Games

I started writing this in 1997; last modified: 22 Jan 2022