17

Pathfinding Architecture Optimizations

Steve Rabin and Nathan R. Sturtevant

17.1 Introduction
17.2 Orders of Magnitude Difference in Performance
17.3 Optimization #1: Build High-Quality Heuristics
17.4 Optimization #2: Using an Optimal Search Space Representation
17.5 Optimization #3: Preallocate All Necessary Memory
17.6 Optimization #4: Overestimating the Heuristic
17.7 Optimization #5: Better Heuristics
17.8 Optimization #6: Open List Sorting
17.9 Optimization #7: Don't Backtrack during the Search
17.10 Optimization #8: Caching Successors
17.11 Bad Ideas for Pathfinding
17.12 Conclusion

17.1 Introduction

Agent path requests are notorious for devouring huge proportions of the AI's CPU cycles in many genres of games, such as real-time strategy games and first-person shooters. Therefore, there is a large need for AI programmers to all be on the same page when it comes to optimizing pathfinding architectures. This chapter will cover in a priority order the most significant steps you can take to get the fastest pathfinding engine possible.

All game developers understand that A* is the pathfinding search algorithm of choice, but surprisingly, or not so surprisingly, it is not a panacea. There is a huge realm of knowledge that is crucial to crafting the fastest engine. In fact, even if a large number of pathfinding design choices have already been made, there is still much you can do.

17.2 Orders of Magnitude Difference in Performance

What is the difference between the fastest and slowest A* implementations?

At the DigiPen Institute of Technology video game university, the introductory AI course has students program a simple A* implementation on fixed regular grid as one of the first assignments. As extra credit for the assignment, there is a contest held to see who can write the fastest A* implementation. So if you were to guess the difference between the fastest and slowest solutions, what would you guess? Would you guess that the best solution is several times faster than the slowest?

The true answer is quite surprising. Given hundreds of students who have taken the course over the years, the fastest implementations are 2 orders of magnitude faster than the slowest implementations (a 100× difference). The fastest implementations are also 1 order of magnitude faster than the average implementation (a 10× difference). To put concrete numbers behind this example, on a given map, the fastest implementation finds a path in ~200 us, the average takes ~2500 us, and the slowest implementations take upwards of 20,000 us. Given that these are junior, senior, and master's students, how do you think you would rank if given the same task? It's a strange question, since as a professional game programmer you would never be put in such a position. Wherever you might rank, it is a scary thought that you might be 1 to 2 orders of magnitude slower than the best solution.

Although with fewer students, the second author has had similar experiences with his students in both regular assignments and competitions. The insights of both authors have been distilled here. Thus, you might want to scour this chapter for the nuggets of wisdom that will keep you within spitting distance of the best implementations.

17.3 Optimization #1: Build High-Quality Heuristics

This first optimization is the epitome of the classic programming trade-off between memory and speed. There are many ways that heuristics can be built; we will go through several useful approaches here.

17.3.1 Precompute Every Single Path (Roy–Floyd–Warshall)

While at first glance it seems ridiculous, it is possible to precompute every single path in a search space and store it in a look-up table. The memory implications are severe, but there are ways to temper the memory requirements and make it work for games.

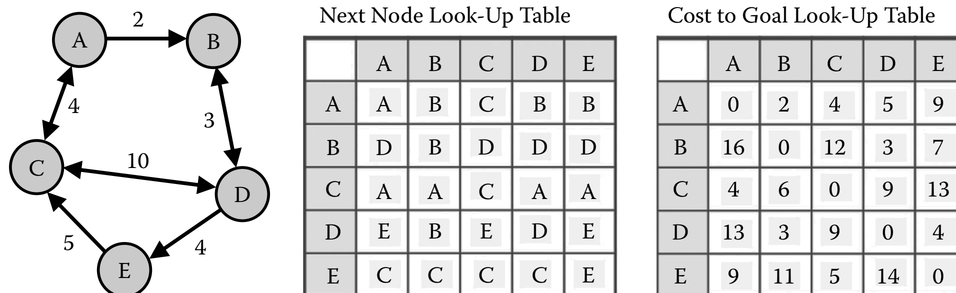
The algorithm is known in English-speaking circles as the Floyd–Warshall algorithm, while in Europe it is better known as Roy–Floyd. Since the algorithm was independently discovered by three different mathematicians, we'll give credit to each and refer to it as the Roy–Floyd–Warshall algorithm [Millington 09].

While we won't explain the algorithm in enough detail to implement it, you should be aware of its basic advantages and properties so that you can make an informed choice whether to pursue implementing it for your game. Here are the facts:

- Roy–Floyd–Warshall is the absolute fastest way to generate a path at runtime. It should routinely be an order of magnitude faster than the best A* implementation.
- The look-up table is calculated offline before the game ships.
- The look-up table requires $O(n^2)$ entries, where n is the number of nodes. For example, for a 100 by 100 grid search space, there are 10,000 nodes. Therefore, the memory required for the look-up table would be 100,000,000 entries (with 2 bytes per entry, this would be ~200 MB).
- Path generation is as simple as looking up the answer. The time complexity is $O(p)$, where p is the number of nodes in the final path.

Figure 17.1 shows a search space graph and the resulting tables generated by the Roy–Floyd–Warshall algorithm. A full path is found by consecutively looking up the next step in the path (left table in Figure 17.1). For example, if you want to find a final path from B to A, you would first look up the entry for (B, A), which is node D. You would travel to node D, then look up the next step of the path (D, A), which would be node E. By repeating this all the way to node A, you will travel the optimal path with an absolute minimum amount of CPU work. If there are dynamic obstacles in the map which must be avoided, this approach can be used as a very accurate heuristic estimate, provided that distances are stored in the look-up table instead of the next node to travel to (right table in Figure 17.1).

As we mentioned earlier, in games you can make the memory requirement more reasonable by creating minimum node networks that are connected to each other [Waveren 01, van der Sterren 04]. For example if you have 1000 total nodes in your level, this would normally require $1000^2 = 1{,}000{,}000$ entries in a table. But if you can create 50 node zones of 20 nodes each, then the total number of entries required is $50 \times 20^2 = 20{,}000$ (which is 50 times fewer entries).



Next Node Look-Up Table

	A	B	C	D	E
A	A	B	C	B	B
B	D	B	D	D	D
C	A	A	C	A	A
D	E	B	E	D	E
E	C	C	C	C	E

Cost to Goal Look-Up Table

	A	B	C	D	E
A	0	2	4	5	9
B	16	0	12	3	7
C	4	6	0	9	13
D	13	3	9	0	4
E	9	11	5	14	0

Figure 17.1

A search space with its corresponding Roy–Floyd–Warshall path look-up table on the left and a very accurate heuristic cost look-up table on the right.

17.3.2 Lossless Compression of Roy–Floyd–Warshall

Another approach to reducing the memory requirement is to compress the Roy–Floyd–Warshall data. Published work [Botea 11] has shown the effectiveness of compressing the data, and this approach fared very well in the 2012 Grid-Based Path Planning competition (http://www.movingai.com/GPPC/), when sufficient memory was available.

An alternate way to compress the Roy–Floyd–Warshall data is to take advantage of the structure of the environment. In many maps, but not all maps, there are relatively few optimal paths of significant length through the state space, and most of these paths overlap. Thus, it is possible to find a sparse number of "transit nodes" through which optimal paths cross [Bast et al. 07]. If, for every state in the state space, we store the path to all transit nodes for that state, as well as the optimal paths between all transit nodes, we can easily reconstruct the shortest path information between any two states, using much less space than when storing the shortest path between all pairs of states. This is one of several methods which have been shown to be highly effective on highway road maps [Abraham et al. 10].

17.3.3 Lossy Compression of Roy–Floyd–Warshall

The full Roy–Floyd–Warshall data results in very fast pathfinding queries, at the cost of memory overhead. In many cases you might want to use less memory and more CPU, which suggests building strong, but not perfect heuristics.

Imagine if we store just a few rows/columns of the Roy–Floyd–Warshall data. This corresponds to keeping the shortest paths from a few select nodes. Fortunately, improved distance estimates between all nodes can be inferred from this data. If $d(x, y)$ is the distance between node x and y, and we know $d(p, z)$ for all z, then the estimated distance between x and y is $h(x, y) = |\, d(p, x) - d(p, y)\,|$, where p is a pivot node that corresponds to a single row/column in the Roy–Floyd–Warshall data. With multiple pivot nodes, we can perform multiple heuristic lookups and take the maximum. The improved estimates will reduce the cost of A* search.

This approach has been developed in many contexts and been given many different names [Ng and Zhang 01, Goldberg and Harrelson 05, Goldenberg et al. 11, Rayner et al. 11]. We prefer the name *Euclidean embedding*, which we will justify shortly. First, we summarize the facts about this approach:

- Euclidean embeddings can be far more accurate than the default heuristics for a map, and in some maps are nearly as fast as Roy-Floyd-Warshall.
- The look-up table can be calculated before the game ships or at runtime, depending on the size and dynamic nature of the maps.
- The heuristic requires $O(kn)$ entries, where n is the number of nodes and k is the number of pivots.
- Euclidean embeddings provide a heuristic for guiding A* search. Given multiple heuristics, A* should usually take the maximum of all available heuristics.

Why do we call this a Euclidean embedding? Consider a map that is wrapped into a spiral, such as in Figure 17.2. Points A and B are quite close in the coordinates of the map, but quite far when considering the minimal travel distance between A and B. If we could just unroll the map into a straight line, the distance estimates would be more accurate. Thus, the central problem is that the coordinates used for aesthetic and gameplay

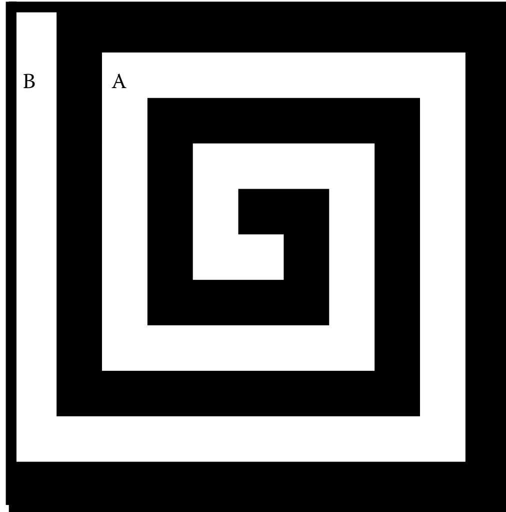


Figure 17.2

A map where straight-line distances are inaccurate.

purposes are not the best for A* search purposes. That is, they do not provide accurate heuristic estimates. If we could provide a different set of coordinates optimized for A* search, we could use these coordinates to estimate distances between nodes and have a higher quality heuristic. This process of transforming a map into a new state space where distance estimates are (hopefully) more accurate is called an *embedding*. A single-source shortest-path search from a pivot node is equivalent to performing a one-dimensional embedding, as each node gets a single coordinate, and the heuristic in this embedding is the distance between the embedded points. Other types of embeddings are possible, just not yet well understood.

The key question of this approach is how the pivots should be selected. In general, a pivot should not be at the center of the map, but near the edges. The heuristic from pivot p between nodes x and y will be most accurate when the optimal path from p to x goes through y. In many games, there are locations where characters will commonly travel, which suggests good locations for pivots. In a RPG, for instance, entrance and exit points to an area are good locations. In a RTS, player bases would be most useful. In a capture-the-flag FPS, the location of the flag would probably work well.

17.4 Optimization #2: Using an Optimal Search Space Representation

If you have to search for a path at runtime, then the number one optimization you can make is to use an efficient search space representation. The reason is that the time spent looking for a path is directly proportional to the number of nodes that must be considered. Fewer nodes equates to less time searching. A more in-depth discussion on choosing a search space representation can be found within this book [Sturtevant 13].

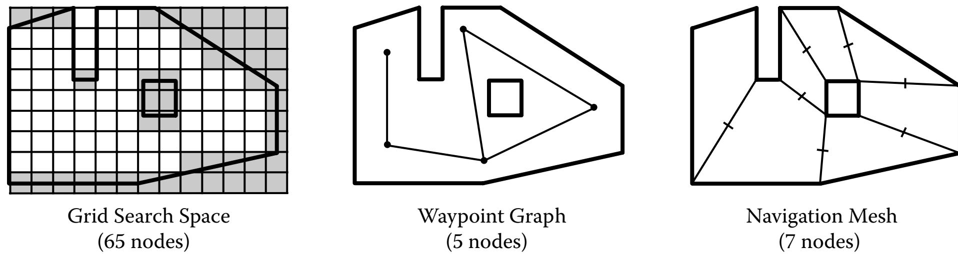


Figure 17.3

Three search space representations. Note the number of nodes in each representation, which affects search speed.

Figure 17.3 shows the three primary search space representations available. As you can plainly see, the most nodes are used by a grid search space, with an order of magnitude fewer nodes in the waypoint graph and the navigation mesh (navmesh).

If you have a large world, none of the three search space representations will be sufficient to keep CPU load to a minimum. Instead, you need to resort to subdividing the search space with a hierarchical representation. *Hierarchical pathfinding* is the concept that the search space can be subdivided into at least two levels: a high-level zone-to-zone representation and a low-level step-by-step representation [Rabin 00]. In this scheme, a path is first found in the high-level representation from the starting zone to the goal zone (think of rooms in a castle, starting in the foyer and finding a room-to-room path to the balcony). Then, to begin moving, the low-level path is found from within the starting zone to the next zone on the path (for example, from the standing place in the foyer to the next room on the path). Once the second zone is entered, a step-by-step path is then found from the second zone to the third zone, and so on.

Two concrete examples of hierarchical pathfinding in shipped games include *Dragon Age: Origins* [Sturtevant 08] and *Company of Heroes* [Jurney et al. 07]. The final architecture in *Dragon Age: Origins* used two levels of grid-based abstraction above the low-level grid. In *Company of Heroes*, the high-level search space representation was a hex-grid and the low-level representation was a regular square grid. If there isn't enough memory to store the Roy–Floyd–Warshall solution in memory for the low-level state space, there is usually enough memory to store this information in an abstract state space.

17.5 Optimization #3: Preallocate All Necessary Memory

Once you have optimized the search space, the next step is to ensure that absolutely no memory is allocated during the search. Although this knowledge should be ingrained among all game programmers, it can't be stressed enough. Memory allocation during a search can increase search times by at least an order of magnitude.

How do you avoid memory allocations? Simply preallocate a pool of memory at start time and reuse it. Since nodes are all the same memory size, they can easily be pulled out of a pre-allocated buffer without any fragmentation issues.

The memory needed for A* can also be part of the map representation. This avoids the need to explicitly store a separate closed list, as each node in the map can have a flag

indicating whether it is on closed or not. If an id is used instead of a Boolean flag, the search can avoid having to reset the nodes in the closed list between most searches.

17.6 Optimization #4: Overestimating the Heuristic

In order for A* to guarantee an optimal path, the heuristic must be admissible, meaning that the heuristic guess of the cost from the current node to the goal node must never overestimate the true cost. However, by using an overestimating heuristic, you can get a tremendous speed-up at the possible expense of a slightly nonoptimal path. While this sounds like a terrible trade-off initially, it turns out that a small amount of overestimating has large benefits with very little noticeable nonoptimality. In the world of search algorithms this once might have been seen as heresy, but in the video game industry it's a shrewd and worthwhile optimization.

In order to understand how to overestimate the heuristic, let's first look at Equation 17.1, which is the classic A* cost formula. As you can see, the final cost, $f(x)$, is the sum of the given cost, $g(x)$, and the heuristic cost, $h(x)$. Each node added to the Open List gets this final cost assigned to it and this is how the Open List is sorted.

$$f(x) = g(x)+h(x) \tag{17.1}$$

Equation 17.2 shows the addition of a weight on the heuristic portion of the formula.

$$f(x) = g(x) + \left(h(x) \times weight\right) \tag{17.2}$$

By altering the weight, we can tune how A* behaves. If the weight is zero, then the formula reduces down to just $g(x)$, which is identical to the Dijkstra search algorithm. This approach is guaranteed to find an optimal path, but is not a "smart" search because it explores uniformly outward in all directions. If the weight is 1.0, then the equation is the classic A* formula, guaranteed to expand the minimal number of nodes needed to find an optimal path given the current heuristic estimate, modulo tie-breaking. If the weight is larger than 1.0, then we are tilting the algorithm toward the behavior of Greedy Best-First search, which is not optimal but focuses the search on finding the goal as quickly as possible.

Thus, we can tune A* with the weight to lean it toward Dijkstra or Greedy Best-First. By using weights in the neighborhood of 1.1 to 1.5 or higher, we can progressively force the search to more aggressively push toward the goal node, at the increasing expense of a possible sub-optimal path. When the terrain is filled with random obstacles that resemble columns or trees, then a larger weight makes a lot of sense and the path is not noticeably suboptimal. However, if significant backtracking away from the goal is required for the final path, then a lower weight is advisable.

The correct weight for your game or parts of your game must be discovered experimentally. There is also the possibility of adaptively discovering the ideal weight for an area given a particular error tolerance on how suboptimal the path is allowed to be. See [Thayer and Ruml 08] for one such algorithm. At any rate, overestimating the heuristic is a tried and true optimization for games that you will want to explore.

17.7 Optimization #5: Better Heuristics

There are two ways to use better heuristics. First, some heuristics are more suited to solve certain problems, and so selecting the correct heuristic can significantly reduce the work required to solve a problem. The second approach is to build and store improved heuristics, of which the Roy–Floyd–Warshall algorithm is just one example. Building a new heuristic is most useful when the map topology is relatively static, as changing the world can invalidate a heuristic.

As for heuristic selection, consider pathfinding on a grid, with 8-directional movement. Three possible heuristics are straight-line (Euclidean) distance, octile distance, which assumes that only 45° and 90° angles are allowed, and Manhattan (city-block) distance. Manhattan distance is a poor heuristic, because it will overestimate distances, not taking diagonal movement into account. A straight-line heuristic is also poor, because it will underestimate distances, assuming that paths can take any angle. The octile heuristic, which corresponds exactly to movement in the world, is the most accurate and best heuristic to use on a grid. The octile heuristic between two points can be computed as $max(\Delta x, \Delta y) + 0.41 \cdot min(\Delta x, \Delta y)$, assuming that diagonal movement has cost 1.41.

17.8 Optimization #6: Open List Sorting

A textbook description of A* states that the node with the lowest *f*-cost should be expanded at each step. A more efficient implementation will break ties between nodes with equal *f*-costs towards those with largest *g*-cost, as these nodes are expected to be closer to the goal. This results in better tie-breaking, which can be quite significant in some maps. The sorting for the Open list can be done in a priority queue structure such as a heap, although researchers have spent significant effort improving these data structures, so if full sorting is required, a weak-heap [Edelkamp et al., 12] is just one option for speeding up A*. It is very often the case that the last node inserted into priority queue is immediately removed again for the next expansion. Caching an inserted node before inserting it into the Open List can reduce this overhead.

An even more efficient approach is to avoid explicitly sorting states. In state spaces where the number of unique *f*-costs will be small, a list can be maintained for each unique *f*-cost; finding the best node on the Open List is as simple as taking the first node off the list with the smallest *f*-cost. It would seem that this would preclude tie-breaking by higher *g*-costs, but treating each *f*-cost list as a LIFO stack will produce similar tie-breaking.

In some searches, the number of nodes considered is limited, such that the Open List typically has under 10 or so nodes with a max of 20 or 30. In such a case, an unordered Open List represented as an array might actually be the best choice, although this should always be verified experimentally. Inserting new nodes is essentially free, O(1), and finding the cheapest node is a matter of simply walking the short list. Without the overhead of even a trivial scheme, an unordered Open List can be extremely fast since it simply doesn't execute that many instructions. To offer a little more detail, the unordered Open List is held as an array, with inserted nodes placed at the end of the array. When a node must be removed, it is replaced with the node at the end, in order to stay packed. With this data structure, the overhead in maintaining the data structure is almost nonexistent and its

minimalist size (one pointer per node) means that it is very cache-friendly, thus resulting in even more speed.

17.9 Optimization #7: Don't Backtrack during the Search

It might seem obvious, but no path ever backtracks along the same nodes it's already visited, so similarly an A* search should also not consider nodes that backtrack. In practice, this is as simple as not considering a neighboring node if it is the same as the parent node. This simple optimization will speed up a search by roughly one over the branching factor. For a grid search space this is 1/8, but for a navmesh search space it's about 1/3.

In grids there are many short cycles, meaning that there is overhead from redundantly looking up a given state from many different neighbors. An inexpensive scan through the state space can allow the search to skip many intermediate nodes to avoid these redundant lookups. The full details of this approach are part of the Jump-Point Search algorithm [Harabor and Grastien 11].

17.10 Optimization #8: Caching Successors

One of the most common operations in an A* search is to look up the neighbors of a node. Thus, it follows that this operation should be as cheap as possible. Storing the neighbors of each node explicitly, rather than traversing more expensive data structures, can result in significant improvements in speed, at the cost of additional memory.

17.11 Bad Ideas for Pathfinding

The following are a list of bad ideas that usually result in slower pathfinding searches.

17.11.1 Bad Idea #1: Simultaneous Searches

When many pathfinding search requests are required all at once, an architectural decision must be made as to how many simultaneous search requests should be processed at the same time. For example, if 10 requests are all needed, it's a tempting thought to time-slice between all of the requests so that one very slow search doesn't hold the others up.

Unfortunately, supporting many simultaneous searches at the same time is fraught with disaster. The primary problem is that you'll need to support separate Open Lists for each request. The implications are severe as to the amount of memory required, and the subsequent thrashing in the cache can be devastating.

But what is to be done about a single search that holds up all other searches? On one hand, this might be a false concern because your pathfinding engine should be blindingly fast for all searches. If it isn't, then that's an indication that you chose the wrong search space representation or should be using hierarchical pathfinding.

However, if we concede that a single search might take a very long time to calculate, then one solution is to learn from supermarkets. The way supermarkets deal with this problem is to create two types of check-out lanes. One is for customers with very few items (10 items or less) and one for the customers with their cart overflowing with groceries. We can do a similar thing with pathfinding by allowing up to two searches at a time.

One queue is for requests deemed to be relatively fast (based on distance between the start and goal) and one queue for requests deemed to take a long time (again based on distance between the start and goal).

17.11.2 Bad Idea #2: Bidirectional Pathfinding

One innovative approach for a search algorithm is to search the path from both directions and complete the path when the searches meet each other. This bidirectional pathfinding reduces the amount of nodes visited with breadth-first and depth-first searches [Pohl 71], but what about A*?

One brilliant reason to consider this approach is the continent and island problem. Consider a search that begins on the continent and the goal is on an island, but we are unable to cross water. With a traditional A* search starting on the continent, the entire continent must be explored before the search concludes that no path exists, which is very, very time consuming. With bidirectional pathfinding, the search starts at both ends, the island side quickly runs out of nodes, and the search concludes that there is no path (with a minimal amount of work).

The continent-island argument however is better solved using a hierarchical approach. At a minimum, the continent would be considered one zone and the island another zone. For a hierarchical architecture, the top-level search would almost instantly discover that no path connects the two zones and the search would fail quickly.

But even without considering the continent-island argument, the truth is that bidirectional pathfinding for A* often requires twice the amount of work. This can be seen when the two searches are separated by a barrier and both searches back up behind the barrier until one spills over and connects. Since we care more about the worst case than best case, this is an important case to avoid. For these reasons, bidirectional pathfinding for A* is usually a poor choice.

17.11.2 Bad Idea #3: Cache Successful or Failed Paths

While caching results for expensive operations is generally good optimization advice, it is not a good idea for paths. The reason is that there are simply too many unique paths. The memory requirements would be very large (similar to Roy–Floyd–Warshall), and the chance that you'll request exactly the same path again is very small.

17.12 Conclusion

This article has covered the techniques for improving the speed of your A* implementation. Using these ideas can ensure that your code is on par with the best possible implementations, and several orders of magnitude faster than more naïve implementations.

References

[Abraham et al. 10] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. F. Werneck. "Highway Dimension, Shortest Paths, and Provably Efficient Algorithms." *ACM-SIAM Symposium on Discrete Algorithms*, pp. 782–793, 2010. Available online (http://research.microsoft.com/pubs/115272/soda10.pdf).

[Bast et al. 07] H. Bast, S. Funke, P. Sanders, and D. Schultes. "In Transit to Constant Time Shortest-Path Queries in Road Networks." Workshop on Algorithm Engineering and Experiments, 2007. Available online (http://www.siam.org/proceedings/alenex/2007/alx07_transit.pdf).

[Botea 11] A. Botea. "Ultra-fast optimal pathfinding without runtime search." *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pp. 122–127, 2011. Available online (http://www.aaai.org/ocs/index.php/AIIDE/AIIDE11/paper/view/4050).

[Edelkamp et al., 12] S. Edelkamp, A. Elmasry, and J. Katajainen. "The weak-heap family of priority queues in theory and praxis." *Proceedings of the 18th Computing: The Australasian Theory Symposium, Conferences in Research and Practice in Information Technology*, pp. 103–112, 2012. Available online (http://www.cphstl.dk/Paper/CATS12/cats12.pdf).

[Goldberg and Harrelson 05] A. V. Goldberg and C. Harrelson. "Computing the shortest path: A search meets graph theory." *ACM-SIAM Symposium on Discrete Algorithms*, pp. 156–165, 2005.

[Goldenberg et al. 11] M. Goldenberg, N. R. Sturtevant, A. Felner, and J. Schaeffer. "The compressed differential heuristic." *AAAI Conference on Artificial Intelligence*, pp. 24-29, 2011. Available online (http://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/view/3723).

[Harabor and Grastien 11] D. Harabor and A. Grastein. Online Graph Pruning for Pathfinding On Grid Maps, *Proceedings of the AAAI Conference on Artificial Intelligence (2011)*, pp. 1114–1119. Available online (http://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/view/3761).

[Jurney et al. 07] C. Jurney and S. Hubick. "Dealing with destruction: AI from the trenches of company of heroes." *Game Developers Conference*, 2007. Available online (https://store.cmpgame.com/product.php?cat=24&id=2089).

[Millington 09] I. Millington. "Constant Time Game Pathfinding with the Roy–Floyd–WarshallAlgorithm." http://idm.me.uk/ai/wfi.pdf, 2009.

[Ng and Zhang 01] T. S. Eugene Ng and H. Zhang. "Predicting Internet network distance with coordinates-based approaches." *IEEE International Conference on Computer Communications (INFOCOM)*, pp. 170–179, 2001.

[Pohl 71] I. Pohl. "Bi-directional search." In *Machine Intelligence 6*, edited by Meltzer and D. Michie. American Elsevier, 1971, pp. 127–140.

[Rabin 00] S. Rabin. "A* Speed optimizations." In *Game Programming Gems*, edited by Mark DeLoura. Hingham, MA: Charles River Media, 2000, pp. 272–287.

[Rayner et al. 11] C. Rayner, M. Bowling, and N. Sturtevant. "Euclidean Heuristic Optimization." *AAAI Conference on Artificial Intelligence*, pp. 81–86, 2011. Available online (http://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/view/3594).

[Sturtevant 08] N. Sturtevant. "Memory-efficient pathfinding abstractions." In *AI Game Programming Wisdom 4*, edited by Steve Rabin. Hingham, MA: Charles River Media, 2008, pp. 203–217.

[Sturtevant 13] N. Sturtevant. "Choosing a search space representation." In *Game AI Pro*, edited by Steve Rabin. Boca Raton, FL: CRC Press, 2013.

[Thayer and Ruml 08] J. T. Thayer and W. Ruml. "Faster Than Weighted A*: An Optimistic Approach to Bounded Suboptimal Search," *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS-08)*, pp. 355–362, 2008. Available online (http://www.cs.unh.edu/~ruml/papers/optimistic-icaps-08.pdf).

[van der Sterren 04] W. van der Sterren. "Path look-up tables—small is beautiful." In *AI Game Programming Wisdom 2*, edited by Steve Rabin. Hingham, MA: Charles River Media, 2004, pp. 115–129.

[Waveren 01] J. P. van Waveren. "The Quake III Arena Bot," pp. 40–45, 2001. Available online (http://www.kbs.twi.tudelft.nl/docs/MSc/2001/Waveren_Jean-Paul_van/thesis.pdf).