WIKIPEDIA

A* search algorithm

In computer science, **A*** (pronounced as "A star") is a computer algorithm that is widely used in pathfinding and graph traversal, the process of plotting an efficiently directed path between multiple points, called "nodes". It enjoys widespread use due to its performance and accuracy. However, in practical travel-routing systems, it is generally outperformed by algorithms which can pre-process the graph to attain better performance,[1] although other work has found A* to be superior to other approaches.[2]

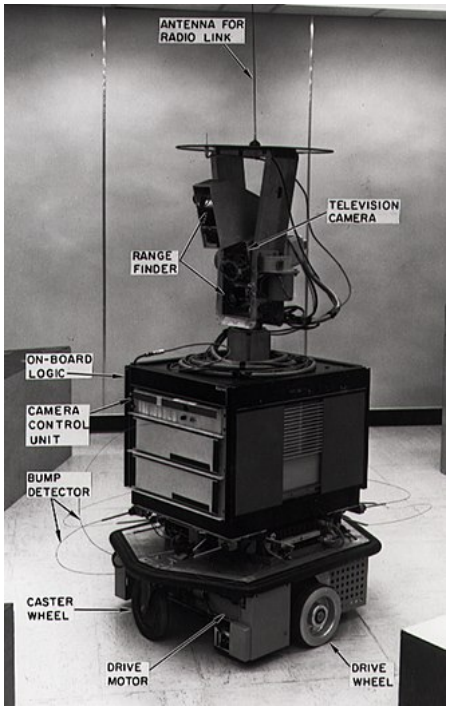
Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute (now SRI International) first described the algorithm in 1968.[3] It is an extension of Edsger Dijkstra's 1959 algorithm. A* achieves better performance by using heuristics to guide its search.

Class	Search algorithm
Data structure	Graph
Worst-case performance	$O(\|E\|) = O(b^d)$
Worst-case space complexity	$O(\|V\|) = O(b^d)$

Contents

History

In 1968, AI researcher Nils Nilsson was trying to improve the path planning done by Shakey the Robot, a prototype robot that could navigate through a room containing obstacles. This path-finding algorithm, which Nilsson called A1, was a faster version of the then best known method, Dijkstra's algorithm, for finding shortest paths in graphs. Bertram Raphael suggested some significant improvements upon this algorithm, calling the revised version A2. Then Peter E. Hart introduced an argument that established A2, with only minor changes, to be the best possible algorithm for finding shortest paths. Hart, Nilsson and Raphael then jointly developed a proof that the revised A2 algorithm was *optimal* for finding shortest paths under certain well-defined conditions.



A* was invented by researchers working on Shakey the Robot's path planning.

Description

A* is an informed search algorithm, or a best-first search, meaning that it solves problems by searching among all possible paths to the solution (goal) for the one that incurs the smallest cost (least distance travelled, shortest time, etc.), and among these paths it first considers the ones that *appear* to lead most quickly to the solution. It is formulated in terms of weighted graphs: starting from a specific node of a graph, it constructs a tree of paths starting from that node, expanding paths one step at a time, until one of its paths ends at the predetermined goal node.

At each iteration of its main loop, A* needs to determine which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the cost (total weight) still to go to the goal node. Specifically, A* selects the path that minimizes

$$f(n) = g(n) + h(n)$$

where n is the last node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic that estimates the cost of the cheapest path from n to the goal. The heuristic is problem-specific. For the algorithm to find the actual shortest path, the heuristic function must be admissible, meaning that it never overestimates the actual cost to get to the nearest goal node.

Typical implementations of A* use a priority queue to perform the repeated selection of minimum (estimated) cost nodes to expand. This priority queue is known as the *open set* or *fringe*. At each step of the algorithm, the node with the lowest $f(x)$ value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty).[a] The f value of the goal is then the length of the shortest path, since h at the goal is zero in an admissible heuristic.

The algorithm described so far gives us only the length of the shortest path. To find the actual sequence of steps, the algorithm can be easily revised so that each node on the path keeps track of its predecessor. After this algorithm is run, the ending node will point to its predecessor, and so on, until some node's predecessor is the start node.

As an example, when searching for the shortest route on a map, $h(x)$ might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any two points.

If the heuristic h satisfies the additional condition $h(x) \leq d(x, y) + h(y)$ for every edge (x, y) of the graph (where d denotes the length of that edge), then h is called monotone, or consistent. In such a case, A* can be implemented more efficiently—roughly speaking, no node needs to be processed more than once (see *closed set* below)—and A* is equivalent to running Dijkstra's algorithm with the reduced cost $d'(x, y) = d(x, y) + h(y) - h(x)$.

Pseudocode

The following pseudocode describes the algorithm:

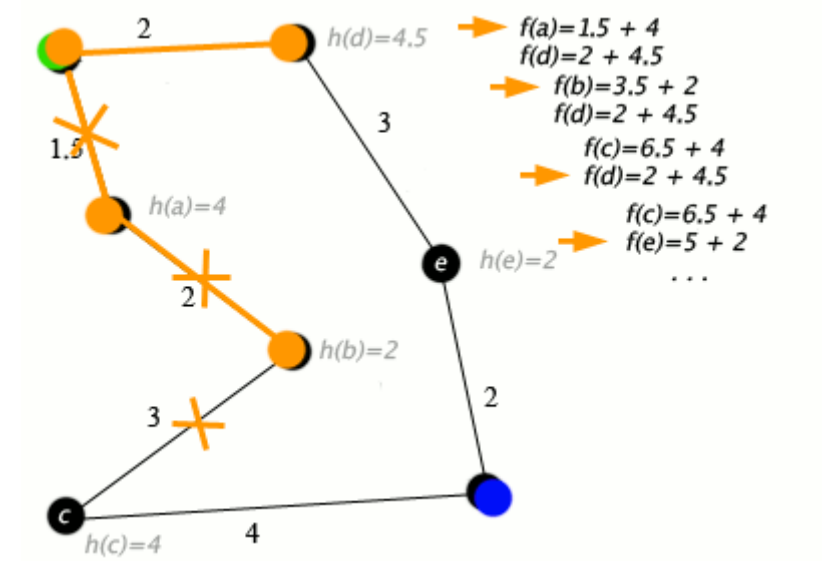
```
function A*(start, goal)
    // The set of nodes already evaluated
    closedSet := {}

    // The set of currently discovered nodes that are not evaluated yet.
    // Initially, only the start node is known.
    openSet := {start}

    // For each node, which node it can most efficiently be reached from.
    // If a node can be reached from many nodes, cameFrom will eventually contain the
    // most efficient previous step.
    cameFrom := an empty map

    // For each node, the cost of getting from the start node to that node.
    gScore := map with default value of Infinity

    // The cost of going from start to start is zero.
    gScore[start] := 0

    // For each node, the total cost of getting from the start node to the goal
    // by passing by that node. That value is partly known, partly heuristic.
    fScore := map with default value of Infinity

    // For the first node, that value is completely heuristic.
    fScore[start] := heuristic_cost_estimate(start, goal)

    while openSet is not empty
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        closedSet.Add(current)

        for each neighbor of current
            if neighbor in closedSet
                continue        // Ignore the neighbor which is already evaluated.
```

```
            if neighbor not in openSet  // Discover a new node
                openSet.Add(neighbor)

            // The distance from start to a neighbor
            //the "dist_between" function may vary as per the solution requirements.
            tentative_gScore := gScore[current] + dist_between(current, neighbor)
            if tentative_gScore >= gScore[neighbor]
                continue        // This is not a better path.

            // This path is the best until now. Record it!
            cameFrom[neighbor] := current
            gScore[neighbor] := tentative_gScore
            fScore[neighbor] := gScore[neighbor] + heuristic_cost_estimate(neighbor, goal)

    return failure

function reconstruct_path(cameFrom, current)
    total_path := [current]
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.append(current)
    return total_path
```

Remark: the above pseudocode assumes that the heuristic function is *monotonic* (or consistent, see below), which is a frequent case in many practical problems, such as the Shortest Distance Path in road networks. However, if the assumption is not true, nodes in the **closed** set may be rediscovered and their cost improved. In other words, the closed set can be omitted (yielding a tree search algorithm) if a solution is guaranteed to exist, or if the algorithm is adapted so that new nodes are added to the open set only if they have a lower f value than at any previous iteration.

Example

An example of an A* algorithm in action where nodes are cities connected with roads and h(x) is the straight-line distance to target point:



Key: green: start; blue: goal; orange: visited

The A* algorithm also has real-world applications. In this example, edges are railroads and h(x) is the great-circle distance (the shortest possible distance on a sphere) to the target. The algorithm is searching for a path between Washington, D.C. and Los Angeles.

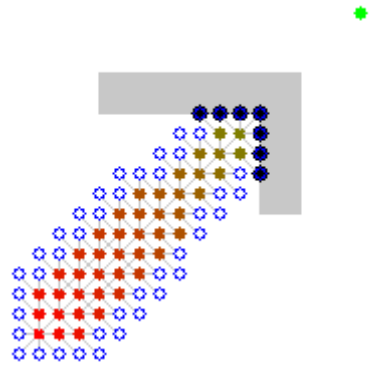
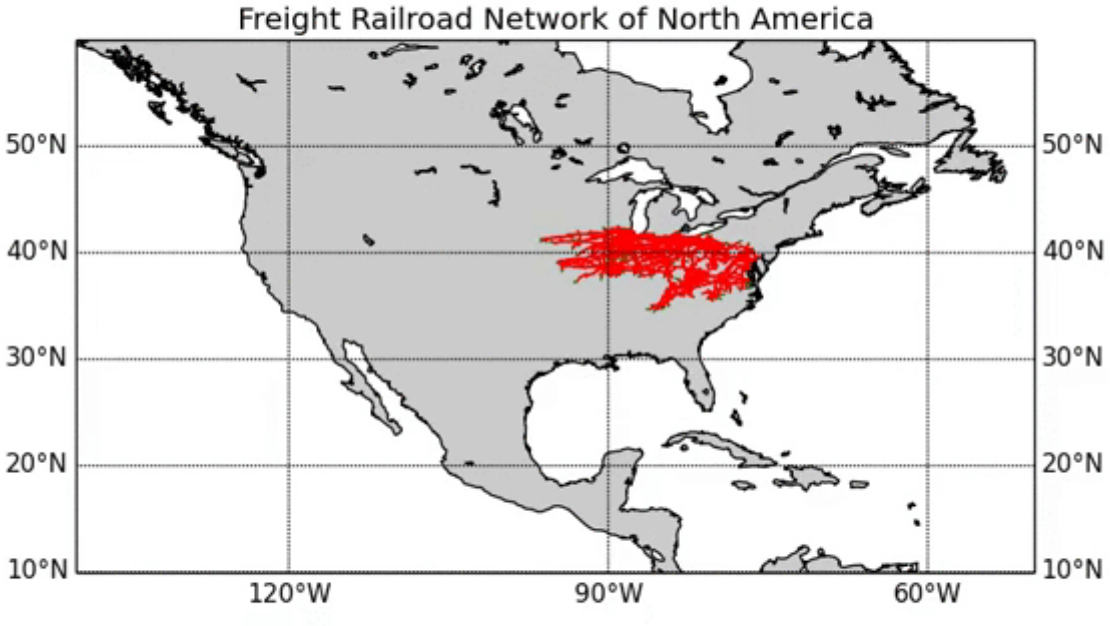


Illustration of A* search for finding path from a start node to a goal node in a robot motion planning problem. The empty circles represent the nodes in the *open set*, i.e., those that remain to be explored, and the filled ones are in the closed set. Color on each closed node indicates the distance from the start: the greener, the farther. One can first see the A* moving in a straight line in the direction of the goal, then when hitting the obstacle, it explores alternative routes through the nodes from the open set.



Properties

Like breadth-first search, A* is *complete* and will always find a solution if one exists provided $d(x, y) > \varepsilon > 0$ for fixed ε.

If the heuristic function *h* is admissible, meaning that it never overestimates the actual minimal cost of reaching the goal, then A* is itself admissible (or *optimal*) if we do not use a closed set. If a closed set is used, then *h* must also be *monotonic* (or consistent) for A* to be optimal. This means that for any pair of adjacent nodes *x* and *y*, where $d(x, y)$ denotes the length of the edge between them, we must have:

$$h(x) \leq d(x, y) + h(y)$$

This ensures that for any path *X* from the initial node to *x*:

$$L(X) + h(x) \leq L(X) + d(x, y) + h(y) = L(Y) + h(y)$$

where L is a function that denotes the length of a path, and Y is the path X extended to include y. In other words, it is impossible to decrease (total distance so far + estimated remaining distance) by extending a path to include a neighboring node. (This is analogous to the restriction to nonnegative edge weights in Dijkstra's algorithm.) Monotonicity implies admissibility when the heuristic estimate at any goal node itself is zero, since (letting $P = (f, v_1, v_2, ..., v_n, g)$ be a shortest path from any node f to the nearest goal g):

$$h(f) \leq d(f, v_1) + h(v_1) \leq d(f, v_1) + d(v_1, v_2) + h(v_2) \leq \ldots \leq L(P) + h(g) = L(P)$$

A* is also optimally efficient for any heuristic h, meaning that no optimal algorithm employing the same heuristic will expand fewer nodes than A*, except when there are multiple partial solutions where h exactly predicts the cost of the optimal path. Even in this case, for each graph there exists some order of breaking ties in the priority queue such that A* examines the fewest possible node

Special cases

Dijkstra's algorithm, as another example of a uniform-cost search algorithm, can be viewed as a special case of A* where $h(x) = 0$ for all x.[4][5] General depth-first search can be implemented using A* by considering that there is a global counter C initialized with a very large value. Every time we process a node we assign C to all of its newly discovered neighbors. After each single assignment, we decrease the counter C by one. Thus the earlier a node is discovered, the higher its $h(x)$ value. Both Dijkstra's algorithm and depth-first search can be implemented more efficiently without including a $h(x)$ value at each node.

Implementation details

There are a number of simple optimizations or implementation details that can significantly affect the performance of an A* implementation. The first detail to note is that the way the priority queue handles ties can have a significant effect on performance in some situations. If ties are broken so the queue behaves in a LIFO manner, A* will behave like depth-first search among equal cost paths (avoiding exploring more than one equally optimal solution).

When a path is required at the end of the search, it is common to keep with each node a reference to that node's parent. At the end of the search these references can be used to recover the optimal path. If these references are being kept then it can be important that the same node doesn't appear in the priority queue more than once (each entry corresponding to a different path to the node, and each with a different cost). A standard approach here is to check if a node about to be added already appears in the priority queue. If it does, then the priority and parent pointers are changed to correspond to the lower cost path. A standard binary heap based priority queue does not directly support the operation of searching for one of its elements, but it can be augmented with a hash table that maps elements to their position in the heap, allowing this decrease-priority operation to be performed in logarithmic time. Alternatively, a Fibonacci heap can perform the same decrease-priority operations in constant amortized time.

Admissibility and optimality

A* is admissible and considers fewer nodes than any other admissible search algorithm with the same heuristic. This is because A* uses an "optimistic" estimate of the cost of a path through every node that it considers—optimistic in that the true cost of a path through that node to the goal will be at least as great as the estimate. But, critically, as far as A* "knows", that optimistic estimate might be achievable.

To prove the admissibility of A*, the solution path returned by the algorithm is used as follows:

When A* terminates its search, it has found a path whose actual cost is lower than the estimated cost of any path through any open node. But since those estimates are optimistic, A* can safely ignore those nodes. In other words, A* will never overlook the possibility of a lower-cost path and so is admissible.

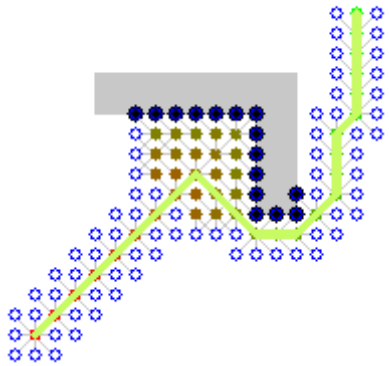
Suppose now that some other search algorithm B terminates its search with a path whose actual cost is *not* less than the estimated cost of a path through some open node. Based on the heuristic information it has, Algorithm B cannot rule out the possibility that a path through that node has a lower cost. So while B might consider fewer nodes than A*, it cannot be admissible. Accordingly, A* considers the fewest nodes of any admissible search algorithm.

This is only true if both:

- A* uses an admissible heuristic. Otherwise, A* is not guaranteed to expand fewer nodes than another search algorithm with the same heuristic.[6]
- A* solves only one search problem rather than a series of similar search problems. Otherwise, A* is not guaranteed to expand fewer nodes than incremental heuristic search algorithms.[7]

Bounded relaxation

While the admissibility criterion guarantees an optimal solution path, it also means that A* must examine all equally meritorious paths to find the optimal path. To compute approximate shortest paths, it is possible to speed up the search at the expense of optimality by relaxing the admissibility criterion. Oftentimes we want to bound this relaxation, so that we can guarantee that the solution path is no worse than $(1 + \varepsilon)$ times the optimal solution path. This new guarantee is referred to as ε-admissible.



A* search that uses a heuristic that is 5.0(=ε) times a consistent heuristic, and obtains a suboptimal path.

There are a number of ε-admissible algorithms:

- Weighted A*/Static Weighting.[8] If $h_a(n)$ is an admissible heuristic function, in the weighted version of the A* search one uses $h_w(n) = \varepsilon\, h_a(n)$, $\varepsilon > 1$ as the heuristic function, and perform the A* search as usual (which eventually happens faster than using h_a since fewer nodes are expanded). The path hence found by the search algorithm can have a cost of at most ε times that of the least cost path in the graph.[9]
- Dynamic Weighting[10] uses the cost function $f(n) = g(n) + (1 + \varepsilon w(n))h(n)$, where $w(n) = \begin{cases} 1 - \frac{d(n)}{N} & d(n) \leq N \\ 0 & \text{otherwise} \end{cases}$, and where $d(n)$ is the depth of the search and N is the anticipated length of the solution path.

- Sampled Dynamic Weighting[11] uses sampling of nodes to better estimate and debias the heuristic error
- A^*_ε.[12] uses two heuristic functions. The first is the FOCAL list, which is used to select candidate nodes, and the second is used to select the most promising node from the FOCAL list.
- A_ε[13] selects nodes with the function $\boldsymbol{Af(n) + Bh_F(n)}$, where *A* and *B* are constants. If no nodes can be selected, the algorithm will backtrack with the function $\boldsymbol{Cf(n) + Dh_F(n)}$, where *C* and *D* are constants.
- AlphA*[14] attempts to promote depth-first exploitation by preferring recently expanded nodes. AlphA* uses the cost function $\boldsymbol{f_\alpha(n) = (1 + w_\alpha(n))f(n)}$, where $\boldsymbol{w_\alpha(n) = \begin{cases} \lambda & g(\pi(n)) \leq g(\tilde{n}) \\ \Lambda & \text{otherwise} \end{cases}}$, where λ and Λ are constants with $\boldsymbol{\lambda \leq \Lambda}$, $\pi(n)$ is the parent of *n*, and $\tilde{n}$ is the most recently expanded node.

Complexity

The time complexity of A* depends on the heuristic. In the worst case of an unbounded search space, the number of nodes expanded is exponential in the depth of the solution (the shortest path) d: $O(b^d)$, where b is the branching factor (the average number of successors per state).[15] This assumes that a goal state exists at all, and is reachable from the start state; if it is not, and the state space is infinite, the algorithm will not terminate.

The heuristic function has a major effect on the practical performance of A* search, since a good heuristic allows A* to prune away many of the b^d nodes that an uninformed search would expand. Its quality can be expressed in terms of the *effective* branching factor b^*, which can be determined empirically for a problem instance by measuring the number of nodes expanded, N, and the depth of the solution, then solving[16]

$$N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d.$$

Good heuristics are those with low effective branching factor (the optimal being $b^* = 1$).

The time complexity is polynomial when the search space is a tree, there is a single goal state, and the heuristic function h meets the following condition:

$$|h(x) - h^*(x)| = O(\log h^*(x))$$

where h^* is the optimal heuristic, the exact cost to get from x to the goal. In other words, the error of h will not grow faster than the logarithm of the "perfect heuristic" h^* that returns the true distance from x to the goal.[9][15]

Applications

A* is commonly used for the common pathfinding problem in applications such as games, but was originally designed as a general graph traversal algorithm.[3] It finds applications to diverse problems, including the problem of parsing using stochastic grammars in NLP.[17] Other cases include an Informational search with online learning.[18]

Relations to other algorithms

What sets A* apart from a greedy best-first search algorithm is that it takes the cost/distance already traveled, $g(n)$, into account.

Some common variants of Dijkstra's algorithm can be viewed as a special case of A* where the heuristic $h(n) = 0$ for all nodes;[4][5] in turn, both Dijkstra and A* are special cases of dynamic programming.[19] A* itself is a special case of a generalization of branch and bound[20] and can be derived from the primal-dual algorithm for linear programming.[21]

Variants

- Anytime Repairing A* (ARA*)[22]
- Block A*
- D*
- Field D*
- Fringe
- Fringe Saving A* (FSA*)
- Generalized Adaptive A* (GAA*)
- IDA*
- Informational search[18]
- Jump point search
- Lifelong Planning A* (LPA*)
- Simplified Memory bounded A* (SMA*)
- Theta*
- Anytime A*[23]
- Realtime A*[24]
- Anytime Dynamic A*
- Time-Bounded A* (TBA*)[25]

A* can also be adapted to a bidirectional search algorithm. Special care needs to be taken for the stopping criterion.[26]

See also

- Pathfinding
- Breadth-first search
- Depth-first search
- Any-angle path planning search for paths that are not limited to move along graph edges but rather can take on any angle

Notes

a. Goal nodes may be passed over multiple times if there remain other nodes with lower f values, as they may lead to a shorter path to a goal.

.

References

1. Delling, D.; Sanders, P.; Schultes, D.; Wagner, D. (2009). *Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation*. Springer. pp. 117–139. doi:10.1007/978-3-642-02094-0_7 (https://doi.org/10.1007%2F978-3-642-02094-0_7).
2. Zeng, W.; Church, R. L. (2009). "Finding shortest paths on real road networks: the case for A*" (https://zenodo.org/record/979689). *International Journal of Geographical Information Science* **23** (4): 531–543. doi:10.1080/13658810801949850 (https://doi.org/10.1080%2F13658810801949850).
3. Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *IEEE Transactions on Systems Science and Cybernetics SSC4* **4** (2): 100–107. doi:10.1109/TSSC.1968.300136 (https://doi.org/10.1109%2FTSSC.1968.300136).
4. De Smith, Michael John; Goodchild, Michael F.; Longley, Paul (2007), *Geospatial Analysis: A Comprehensive Guide to Principles, Techniques and Software Tools* (https://books.google.com/books?id=SULMdT8qPwEC&pg=PA344), Troubadour Publishing Ltd, p. 344, ISBN 9781905886609.
5. Hetland, Magnus Lie (2010), *Python Algorithms: Mastering Basic Algorithms in the Python Language* (https://books.google.com/books?id=9_AXCmGDiz8C&pg=PA214), Apress, p. 214, ISBN 9781430232377.
6. Dechter, Rina; Judea Pearl (1985). "Generalized best-first search strategies and the optimality of A*" (http://portal.acm.org/citation.cfm?id=3830&coll=portal&dl=ACM). *Journal of the ACM* **32** (3): 505–536. doi:10.1145/3828.3830 (https://doi.org/10.1145%2F3828.3830).
7. Koenig, Sven; Maxim Likhachev; Yaxin Liu; David Furcy (2004). "Incremental heuristic search in AI" (http://portal.acm.org/citation.cfm?id=1017140). *AI Magazine*. **25** (2): 99–112.
8. Pohl, Ira (1970). "First results on the effect of error in heuristic search". *Machine Intelligence*. **5**: 219–236.
9. Pearl, Judea (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley. ISBN 0-201-05594-5.
10. Pohl, Ira (August 1973). "The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving" (https://www.cs.auckland.ac.nz/courses/compsci709s2c/resources/Mike.d/Pohl1973WeightedAStar.pdf) (PDF). *Proceedings of the Third International Joint Conference on Artificial Intelligence (IJCAI-73)*. **3**. California, USA. pp. 11–17.
11. Köll, Andreas; Hermann Kaindl (August 1992). "A new approach to dynamic weighting". *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI-92)*. Vienna, Austria. pp. 16–17.
12. Pearl, Judea; Jin H. Kim (1982). "Studies in semi-admissible heuristics". *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*. **4** (4): 392–399.
13. Ghallab, Malik; Dennis Allard (August 1983). "A_ε – an efficient near admissible heuristic search algorithm" (https://web.archive.org/web/20140806200328/http://ijcai.org/Past%20Proceedings/IJCAI-83-VOL-2/PDF/048.pdf) (PDF). *Proceedings of the Eighth International Joint Conference on Artificial Intelligence (IJCAI-83)*. **2**. Karlsruhe, Germany. pp. 789–791. Archived from the original (http://ijcai.org/Past%20Proceedings/IJCAI-83-VOL-2/PDF/048.pdf) (PDF) on 2014-08-06.
14. Reese, Bjørn (1999). "AlphA*: An ε-admissible heuristic search algorithm" (http://home1.stofanet.dk/breese/astaralpha-submitted.pdf.gz)
15. Russell, Stuart; Norvig, Peter (2003) [1995]. *Artificial Intelligence: A Modern Approach* (2nd ed.). Prentice Hall. pp. 97–104. ISBN 978-0137903955.
16. Russell, Stuart; Norvig, Peter (2009) [1995]. *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall. p. 103. ISBN 978-0-13-604259-4.
17. Klein, Dan; Manning, Christopher D. (2003) *A* parsing: fast exact Viterbi parse selection*. Proc. NAACL-HLT.

18. Kagan E. and Ben-Gal I. (2014). "A Group-Testing Algorithm with Online Informational Learning" (http://www.eng.tau.ac.il/~bengal/GTA.pdf) (PDF). IIE Transactions, 46:2, 164-184,.
19. Ferguson, Dave; Likhachev, Maxim; Stentz, Anthony (2005). *A Guide to Heuristic-based Path Planning* (https://www.cs.cmu.edu/afs/cs.cmu.edu/Web/People/maxim/files/hsplanguide_icaps05ws.pdf) (PDF). Proc. ICAPS Workshop on Planning under Uncertainty for Autonomous Systems.
20. Nau, Dana S.; Kumar, Vipin; Kanal, Laveen (1984). "General branch and bound, and its relation to A* and AO*" (https://www.cs.umd.edu/~nau/papers/nau1984general.pdf) (PDF). *Artificial Intelligence*. **23** (1): 29–58. doi:10.1016/0004-3702(84)90004-3 (https://doi.org/10.1016%2F0004-3702%2884%2990004-3)
21. Ye, Xugang; Han, Shih-Ping; Lin, Anhua (2010). "A Note on the Connection Between the Primal-Dual and the A* Algorithm" *Int'l J. Operations Research and Information Systems*. **1** (1): 73–85.
22. Likhachev, Maxim; Gordon, Geof; Thrun, Sebastian. "ARA*: Anytime A* search with provable bounds on sub-optimality (http://robots.stanford.edu/papers/Likhachev03b.pdf)". In S. Thrun, L. Saul, and B. Schölkopf, editors *Proceedings of Conference on Neural Information Processing Systems (NIPS)* Cambridge, MA, 2003. MIT Press.
23. Hansen, Eric A., and Rong Zhou. "Anytime Heuristic Search. (http://www.jair.org/media/2096/live-2096-3136-jair.pdf?q=anytime)" J. Artif. Intell. Res.(JAIR) 28 (2007): 267-297.
24. Korf, Richard E. "Real-time heuristic search. (https://pdfs.semanticscholar.org/2fda/10f6079156c4621fefc8b7cad72c1829ee94.pdf) Artificial intelligence 42.2-3 (1990): 189-211.
25. Björnsson, Yngvi; Bulitko, Vadim; Sturtevant, Nathan (July 11–17, 2009). *TBA*: time-bounded A** (http://web.cs.du.edu/~sturtevant/papers/TBA.pdf) (PDF). IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence. Pasadena, California, USA: Morgan Kaufmann Publishers Inc. pp. 431–436.
26. "Efficient Point-to-Point Shortest Path Algorithms" (http://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP%20shortest%20path%20algorithms.pdf) (PDF). from Princeton University

Further reading

- Hart, P. E.; Nilsson, N. J.; Raphael, B. (1972). "Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths"" *SIGART Newsletter*. **37**: 28–29. doi:10.1145/1056777.1056779 (https://doi.org/10.1145%2F1056777.1056779)
- Nilsson, N. J. (1980). *Principles of Artificial Intelligence* Palo Alto, California: Tioga Publishing Company. ISBN 0-935382-01-1.

External links

- Clear visual A* explanation, with advice and thoughts on path-finding (http://theory.stanford.edu/~amitp/GameProgramming/)
- Variation on A* called Hierarchical Path-Finding A* (HPA*) (http://www.cs.ualberta.ca/~mmueller/ps/hpastar.pdf)

Retrieved from 'https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=825630950'