A* search algorithm

From Wikipedia, the free encyclopedia

In computer science, **A*** (pronounced "A star") is a best-first, graph search algorithm that finds the least-cost path from a given initial node to one goal node (out of one or more possible goals).

It uses a distance-plus-cost heuristic function (usually denoted $f(x)$) to determine the order in which the search visits nodes in the tree. The distance-plus-cost heuristic is a sum of two functions: the path-cost function (usually denoted $g(x)$, which may or may not be a heuristic) and an admissible "heuristic estimate" of the distance to the goal (usually denoted $h(x)$). The path-cost function $g(x)$ is the cost from the starting node to the current node.

Since the $h(x)$ part of the $f(x)$ function must be an admissible heuristic, it must not overestimate the distance to the goal. Thus for an application like routing, $h(x)$ might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any two points (or nodes for that matter).

The algorithm was first described in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael[1]. In their paper, it was called algorithm A. Since using this algorithm yields optimal behavior for a given heuristic, it has been called A*.

Graph search algorithms

Search

- **A***
- B*
- Bellman-Ford algorithm
- Best-first search
- Bidirectional search
- Breadth-first search
- D*
- Depth-first search
- Depth-limited search
- Dijkstra's algorithm
- Floyd–Warshall algorithm
- Hill climbing
- Iterative deepening depth-first search
- Johnson's algorithm
- Uniform-cost search

This algorithm has been generalized into a bidirectional heuristic search algorithm; see bidirectional search.

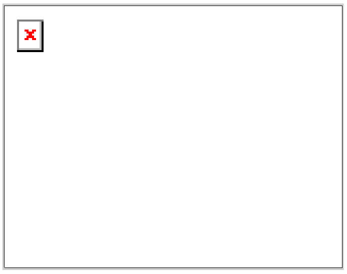
Contents

Algorithm description

Like all informed search algorithms, it first searches the routes that *appear* to be most likely to lead towards the goal. What sets A* apart from a greedy best-first search is that it also takes the distance already traveled into account (the $g(x)$ part of the heuristic is the cost from the start, and not simply the local cost from the previously expanded node).

The algorithm traverses various paths from start to goal. For each node x traversed, it maintains 3 values:

- g(x): the actual shortest distance traveled from initial node to current node
- h(x): the estimated (or "heuristic") distance from current node to goal
- f(x): the sum of g(x) and h(x)


A single-step simulation

Starting with the initial node, it maintains a priority queue of nodes to be traversed, known as the *open set* (not to be confused with open sets in topology). The lower *f*(*x*) for a given node *x*, the higher its priority. At each step of the algorithm, the node with the lowest *f*(*x*) value is removed from the queue, the *f* and *h* 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). (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.) 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. If the actual shortest path is desired, the algorithm may also update each neighbor with its immediate predecessor in the best path found so far; this information can then be used to reconstruct the path by working backwards from the goal node. Additionally, if the heuristic is *monotonic* (see below), a *closed set* of nodes already traversed may be used to make the search more efficient.

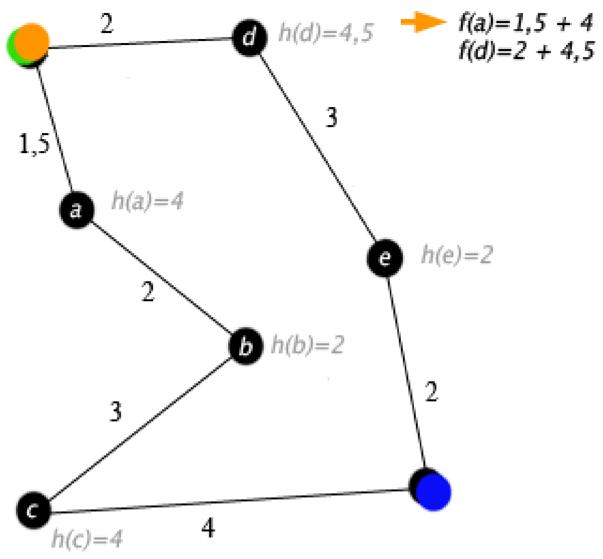
```
function A*(start,goal)
    closedset := the empty set                 % The set of nodes already evaluated.
    openset := set containing the initial node % The set of tentative nodes to be evaluated.
    g_score[start] := 0                        % Distance from start along optimal path.
    h_score[start] := heuristic_estimate_of_distance(start, goal)
    f_score[start] := h_score[start]           % Estimated total distance from start to goal through
    while openset is not empty
        x := the node in openset having the lowest f_score[] value
        if x = goal
            return reconstruct_path(came_from,goal)
        remove x from openset
        add x to closedset
        foreach y in neighbor_nodes(x)
            if y in closedset
                continue
            tentative_g_score := g_score[x] + dist_between(x,y)
            tentative_is_better := false
            if y not in openset
                add y to openset
                h_score[y] := heuristic_estimate_of_distance(y, goal)
                tentative_is_better := true
            elseif tentative_g_score < g_score[y]
                tentative_is_better := true
            if tentative_is_better = true
                came_from[y] := x
                g_score[y] := tentative_g_score
                f_score[y] := g_score[y] + h_score[y]
    return failure

function reconstruct_path(came_from,current_node)
    if came_from[current_node] is set
        p = reconstruct_path(came_from,came_from[current_node])
        return (p + current_node)
    else
        return the empty path
```

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 A star (A*) algorithm in action (nodes are cities connected with roads, h(x) is the straight-line distance to target point).



green - start, blue - target, orange - visited

Properties

Like breadth-first search, A* is *complete* in the sense that it will always find a solution if there is one.

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:

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

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

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

where $L(\cdot)$ 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 $P = (f, v_1, v_2, \ldots, v_n, g)$ be a shortest path from any node *f* to the nearest goal *g*):

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

A* is also optimally efficient for any heuristic *h*, meaning that no 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 nodes.

Special cases

Generally speaking, depth-first search and breadth-first search are two special cases of A* algorithm. Dijkstra's algorithm, as another example of a best-first search algorithm, is the special case of A* where $h(x) = 0$ for all x. For depth-first search, we may consider that there is a global counter C initialized with a very big 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.

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. If ties are broken so the queue behaves in a FIFO manner, A* will behave like Breadth-first search among equal cost paths.

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. When finding a node in a queue to perform this check, many standard implementations of a min-heap require $O(n)$ time. Augmenting the heap with a Hash table can reduce this to constant time.

Why A* is admissible and computationally optimal

A* is both admissible and considers fewer nodes than any other admissible search algorithm with the same heuristic, because A* works from 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.

When A* terminates its search, it has, by definition, 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. Algorithm B cannot rule out the possibility, based on the heuristic information it has, that a path through that node might have 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 that uses a no more accurate heuristic estimate.

This is only true when A* uses a consistent heuristic. Otherwise, A* is not guaranteed to expand fewer nodes than another search algorithm with the same heuristic. See (Generalized best-first search strategies and the optimality of A*, Rina Dechter and Judea Pearl, 1985[2])

Complexity

The time complexity of A* depends on the heuristic. In the worst case, the number of nodes expanded is exponential in the length of the solution (the shortest path), but it is polynomial when the heuristic function *h* meets the following condition:

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

where h^* is the optimal heuristic, i.e. the exact cost to get from *x* to the goal. In other words, the error of *h* should not grow faster than the logarithm of the “perfect heuristic” h^* that returns the true distance from *x* to the goal (Russell and Norvig 2003, p. 101[3]).

More problematic than its time complexity is A*’s memory usage. In the worst case, it must also remember an exponential number of nodes. Several variants of A* have been developed to cope with this, including iterative deepening A* (IDA*), memory-bounded A* (MA*) and simplified memory bounded A* (SMA*) and recursive best-first search (RBFS).

References

- 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.
- Nilsson, N. J. (1980). *Principles of Artificial Intelligence*. Palo Alto, California: Tioga Publishing Company. ISBN 0935382011.
- Pearl, Judea (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley. ISBN 0-201-05594-5.

1. ^ 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* (2): 100–107. doi:10.1109/TSSC.1968.300136 (http://dx.doi.org/10.1109/TSSC.1968.300136).
2. ^ 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 (http://dx.doi.org/10.1145/3828.3830).
3. ^ Russell, S. J.; Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. pp. 97–104. ISBN 0-13-790395-2.

External links

- Tarik Attar's Implementation and visualisation of the A* algorithm (http://www.tarikattar.com/napier/osmastermap/) - Implementation in PHP and visualisation using Google Map API
- Justin Heyes-Jones' A* algorithm tutorial (http://www.geocities.com/jheyesjones/astar.html)
- Herbert Glarner's Interactive Single Step Simulation in VB 6.0 (http://herbert.gandraxa.com/herbert/pfa.asp), implemented as a DLL, including a GUI allowing simulation in user-defined grids.
- Another A* Pathfinding for Beginners (http://www.policyalmanac.org/games/aStarTutorial.htm) (note: incorrectly states that A* always needs a "closed set")
- Amit's Thoughts on Path-Finding and A* (http://theory.stanford.edu/~amitp/GameProgramming/)
- Sven Koenig's Demonstration of Lifelong Planning A* and A* (http://idm-lab.org/applet.html)

- Another Java Applet comparing LPA* and A* Lifelong Planning A* Demonstration (http://homepages.dcc.ufmg.br/~lhrios/applet_lpa/index.html)
- Cuneyt Mertayak's A Generic C++ A* Library (http://www.ceng.metu.edu.tr/~cuneyt/astar.tar.gz)
- Tony Stentz's Papers on D* (Dynamic A*) Path-Finding (http://www.frc.ri.cmu.edu/~axs/)
- Remko Tronçon and Joost Vennekens's JSearch demo (http://el-tramo.be/software/jsearchdemo/): demonstrates various search algorithms, including A*.
- A* search algorithm module (http://lostsouls.org/grimoire_astar) in LPC
- A* search algorithm module (http://www.jamespoag.com/AStarPathfinder.html) in object-oriented C++ with Demo Written for the PopCap Games Framework
- A* search algorithm demo (http://bravobug.com/news/?p=118) in Objective-C/Cocoa for XCode
- Open Source A* (http://sourceforge.net/projects/argorha/) in polygon soup (3D world).
- Variation on A* called Near Optimal Hierarchical Path-Finding (http://www.cs.ualberta.ca/~mmueller/ps/hpastar.pdf) and associated presentation (http://www.cs.ualberta.ca/~bulitko/F06/presentations/2006-09-29-ML.pdf).
- [fr] Implementation and visualisation of the A* algorithm in C# (http://www.csharpfr.com/codes/ALGORITHME-PATHFINDING-STAR_41235.aspx) (with source code)

- A python implementation of the algorithm (http://kks.cabal.fi/A-star)
- A JavaScript implementation of the algorithm (http://www.devpro.it/javascript_id_137.html)

Retrieved from "http://en.wikipedia.org/wiki/A*_search_algorithm"
Categories: Graph algorithms | Routing algorithms | Search algorithms | Combinatorial optimization | Game artificial intelligence | Articles with example code