WIKIPEDIA

A* search algorithm

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

A* (pronounced "A-star") is a graph traversal and path search algorithm, which is often used in many fields of computer science due to its completeness, optimality, and optimal efficiency.[1] One major practical drawback is its $O(b^d)$ space complexity, as it stores all generated nodes in memory. Thus, in practical travel-routing systems, it is generally outperformed by algorithms which can pre-process the graph to attain better performance,[2] as well as memory-bounded approaches; however, A* is still the best solution in many cases.[3]

Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute (now SRI International) first published the algorithm in 1968.[4] It can be seen as an extension of Dijkstra's algorithm. A* achieves better performance by using heuristics to guide its search.

Contents

History
Description
- Pseudocode
- Example
- Implementation details
- Special cases

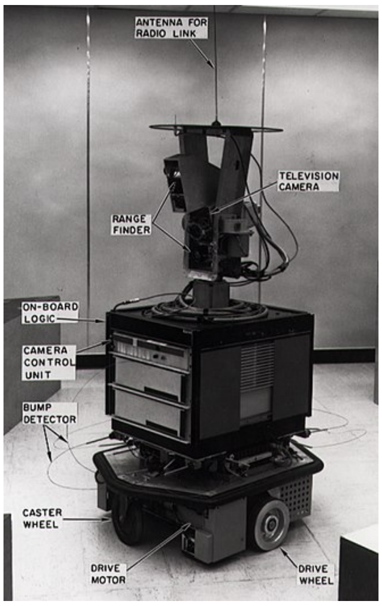
Properties
- Termination and Completeness
- Admissibility
- Optimality and Consistency

Bounded relaxation
Complexity
Applications
Relations to other algorithms
Variants
See also
Notes
References
Further reading
External links

History

A* was created as part of the Shakey project, which had the aim of building a mobile robot that could plan its own actions. Nils Nilsson originally proposed using the Graph Traverser algorithm[5] for Shakey's path planning.[6] Graph Traverser is guided by a heuristic function $h(n)$, the estimated distance from node n to the goal node: it entirely ignores $g(n)$, the distance from the start node to n. Bertram Raphael suggested using the sum, $g(n) + h(n)$.[6] Peter Hart invented the concepts we now call admissibility and consistency of heuristic functions. A* was originally designed for finding least-cost paths when the cost of a path is the sum of its costs, but it has been shown that A* can be used to find optimal paths for any problem satisfying the conditions of a cost algebra.[7]

The original 1968 A* paper[4] contained a theorem stating that no A*-like algorithm[a] could expand fewer nodes than A* if the heuristic function is consistent and A*'s tie-breaking rule is suitably chosen. A "correction" was published a few years later[8] claiming that consistency was not required, but this was shown to be false in Dechter and Pearl's definitive study of A*'s optimality (now called optimal efficiency), which gave an example of A* with a heuristic that was admissible but not consistent expanding arbitrarily more nodes than an alternative A*-like algorithm.[9]



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 is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes

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

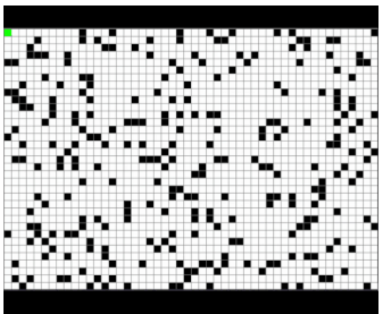
where n is the next node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal. A* terminates when the path it chooses to extend is a path from start to goal or if there are no paths eligible to be extended. The heuristic function is problem-specific. If the heuristic function is admissible – meaning that it never overestimates the actual cost to get to the goal –, A* is guaranteed to return a least-cost path from start to goal.

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 removed node (thus the node with the lowest f value out of all fringe nodes) is a goal node.[b] The f value of that goal is then also the cost 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. For a grid map from a video game, using the Manhattan distance or the octile distance becomes better depending on the set of movements available (4-way or 8-way).

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. With a consistent heuristic, A* is guaranteed to find an optimal path without processing any node more than once and A* is equivalent to running Dijkstra's algorithm with the reduced cost $d'(x, y) = d(x, y) + h(y) - h(x)$.


A* pathfinding algorithm navigating around a randomly-generated maze

Pseudocode

The following pseudocode describes the algorithm:

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

// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimates the cost to reach goal from node n.
function A_Star(start, goal, h)
    // The set of discovered nodes that may need to be (re-)expanded.
    // Initially, only the start node is known.
    // This is usually implemented as a min-heap or priority queue rather than a hash-
set.
    openSet := {start}

    // For node n, cameFrom[n] is the node immediately preceding it on the cheapest path
from start
    // to n currently known.
    cameFrom := an empty map

    // For node n, gScore[n] is the cost of the cheapest path from start to n currently
known.
    gScore := map with default value of Infinity
    gScore[start] := 0

    // For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our current best
guess as to
    // how short a path from start to finish can be if it goes through n.
    fScore := map with default value of Infinity
    fScore[start] := h(start)

    while openSet is not empty
        // This operation can occur in O(Log(N)) time if openSet is a min-heap or a
priority queue
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        for each neighbor of current
            // d(current,neighbor) is the weight of the edge from current to neighbor
            // tentative_gScore is the distance from start to the neighbor through
current
            tentative_gScore := gScore[current] + d(current, neighbor)
            if tentative_gScore < gScore[neighbor]
                // This path to neighbor is better than any previous one. Record it!
                cameFrom[neighbor] := current
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := tentative_gScore + h(neighbor)
                if neighbor not in openSet
                    openSet.add(neighbor)

    // Open set is empty but goal was never reached
    return failure
```

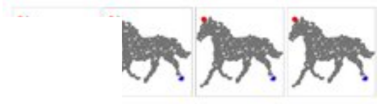
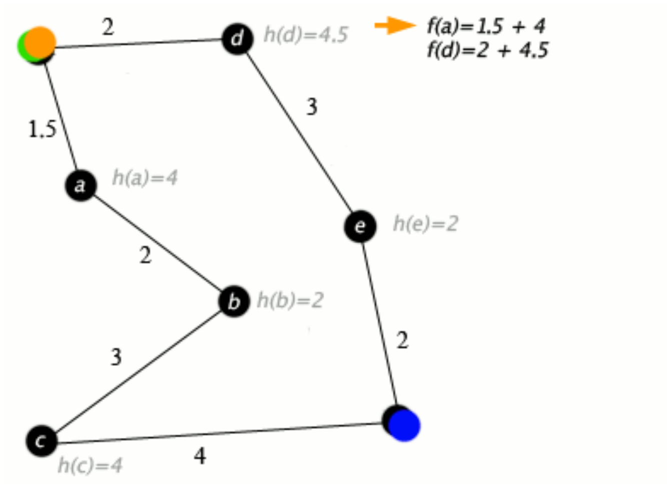


Illustration of A* search for finding a path between two points on a graph.

Remark: In this pseudocode, if a node is reached by one path, removed from openSet, and subsequently reached by a cheaper path, it will be added to openSet again. This is essential to guarantee that the path returned is optimal if the heuristic function is admissible but not consistent. If the heuristic is consistent, when a node is removed from openSet the path to it is guaranteed to be optimal so the test 'tentative_gScore < gScore[neighbor]' will always fail if the node is reached again.

Example

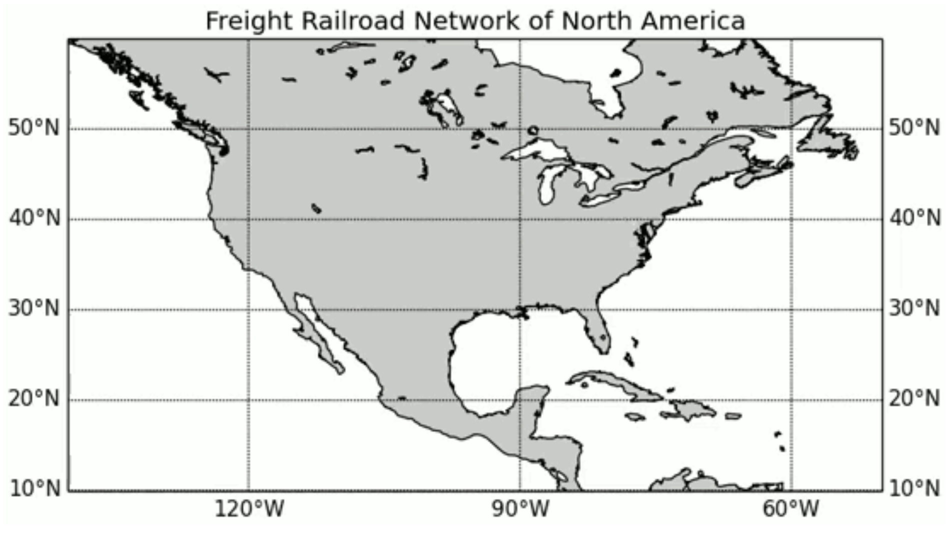
An example of an A* algorithm in action where nodes are cities connected with roads and h(x) is the straight-line distance to target point:



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

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 goal: the greener, the closer. 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.

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.



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.

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.[10][11] 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 an $h(x)$ value at each node.

Properties

Termination and Completeness

On finite graphs with non-negative edge weights A* is guaranteed to terminate and is *complete*, i.e. it will always find a solution (a path from start to goal) if one exists. On infinite graphs with a finite branching factor and edge costs that are bounded away from zero ($d(x, y) > \varepsilon > 0$ for some fixed ε), A* is guaranteed to terminate only if there exists a solution.[1]

Admissibility

A search algorithm is said to be *admissible* if it is guaranteed to return an optimal solution. If the heuristic function used by A* is admissible, then A* is admissible. An intuitive "proof" of this is as follows:

When A* terminates its search, it has found a path from start to goal whose actual cost is lower than the estimated cost of any path from start to goal through any open node (the node's f value). When the heuristic is admissible, those estimates are optimistic (not quite—see the next paragraph), so A* can safely ignore those nodes because they cannot possibly lead to a cheaper solution than the one it already has. In other words, A* will never overlook the possibility of a lower-cost path from start to goal and so it will continue to search until no such possibilities exist.

The actual proof is a bit more involved because the f values of open nodes are not guaranteed to be optimistic even if the heuristic is admissible. This is because the g values of open nodes are not guaranteed to be optimal, so the sum $g + h$ is not guaranteed to be optimistic.

Optimality and Consistency

Algorithm A is optimally efficient with respect to a set of alternative algorithms **Alts** on a set of problems **P** if for every problem P in **P** and every algorithm A′ in **Alts**, the set of nodes expanded by A in solving P is a subset (possibly equal) of the set of nodes expanded by A′ in solving P. The definitive study of the optimal efficiency of A* is due to Rina Dechter and Judea Pearl.[9] They considered a variety of definitions of **Alts** and **P** in combination with A*'s heuristic being merely admissible or being both consistent and admissible. The most interesting positive result they proved is that A*, with a consistent heuristic, is optimally efficient with respect to all admissible A*-like search algorithms on all "non-pathological" search problems. Roughly speaking, their notion of non-pathological problem is what we now mean by "up to tie-breaking". This result does not hold if A*'s heuristic is admissible but not consistent. In that case, Dechter and Pearl showed there exist admissible A*-like algorithms that can expand arbitrarily fewer nodes than A* on some non-pathological problems.

Optimal efficiency is about the *set* of nodes expanded, not the *number* of node expansions (the number of iterations of A*'s main loop). When the heuristic being used is admissible but not consistent, it is possible for a node to be expanded by A* many times, an exponential number of times in the worst case.[12] In such circumstances Dijkstra's algorithm could outperform A* by a large margin. However, more recent research found that this pathological case only occurs in certain contrived situations where the edge weight of the search graph is exponential in the size of the graph, and that certain inconsistent (but admissible) heuristics can lead to a reduced number of node expansions in A* searches.[13][14]

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 + ε) times the optimal solution