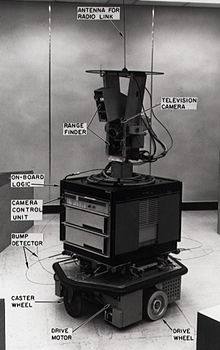
# wikipedia

# A\* search algorithm

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **A\*** is a [computer algorithm](https://en.wikipedia.org/wiki/Computer_algorithm) that is widely used in [pathfinding](https://en.wikipedia.org/wiki/Pathfinding) and [graph traversal](https://en.wikipedia.org/wiki/Graph_traversal), the process of plotting an efficiently directed path between multiple points, called nodes. It enjoys widespread use due to its [performance](https://en.wikipedia.org/wiki/Computer_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]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-1) although other work has found A\* to be superior to other approaches.[[2]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-Zeng-2)

[Peter Hart](https://en.wikipedia.org/wiki/Peter_E._Hart), [Nils Nilsson](https://en.wikipedia.org/wiki/Nils_Nilsson_%28researcher%29) and [Bertram Raphael](https://en.wikipedia.org/wiki/Bertram_Raphael) of Stanford Research Institute (now [SRI International](https://en.wikipedia.org/wiki/SRI_International)) first described the algorithm in 1968.[[3]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-nilsson-3) It is an extension of [Edsger Dijkstra's](https://en.wikipedia.org/wiki/Edsger_Dijkstra) [1959 algorithm](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm). A\* achieves better performance by using [heuristics](https://en.wikipedia.org/wiki/Heuristic_%28computer_science%29) to guide its search.

## History

[](https://en.wikipedia.org/wiki/File:SRI_Shakey_with_callouts.jpg)

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

In 1968, [AI](https://en.wikipedia.org/wiki/Artificial_intelligence) researcher Nils Nilsson was trying to improve the path planning done by [Shakey the Robot](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Dijkstra%27s_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.

## Description

A\* is an [informed search algorithm](https://en.wikipedia.org/wiki/Informed_search_algorithm), or a [best-first search](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Weighted_graph): starting from a specific [node](https://en.wikipedia.org/wiki/Node_%28graph_theory%29) of a graph, it constructs a [tree](https://en.wikipedia.org/wiki/Tree_%28data_structure%29) 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](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Admissible_heuristic), meaning that it never overestimates the actual cost to get to the nearest goal node.

Typical implementations of A\* use a [priority queue](https://en.wikipedia.org/wiki/Priority_queue) to perform the repeated selection of minimum (estimated) cost nodes to expand. This priority queue is known as the [*open set*](https://en.wikipedia.org/w/index.php?title=Open_set_%28Computer_science%29&action=edit&redlink=1) or [*fringe*](https://en.wikipedia.org/w/index.php?title=Fringe_%28Computer_science%29&action=edit&redlink=1). 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]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-4) 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](https://en.wikipedia.org/wiki/Euclidean_distance) to the goal, since that is physically the smallest possible distance between any two points.

If the [heuristic](https://en.wikipedia.org/wiki/Heuristic) *h* satisfies the additional condition *h*(*x*) ≤ *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](https://en.wikipedia.org/wiki/Consistent_heuristic). 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](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm) with the [reduced cost](https://en.wikipedia.org/wiki/Reduced_cost) *d'*(*x*, *y*) = *d*(*x*, *y*) + *h*(*y*) − *h*(*x*).

### Pseudocode

The following [pseudocode](https://en.wikipedia.org/wiki/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 := the 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. The distance from start to a

//neighbor

tentative\_gScore := gScore[current] + dist\_between(current,

neighbor)

if neighbor not in openSet // Discover a new node

openSet.Add(neighbor)

else 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]

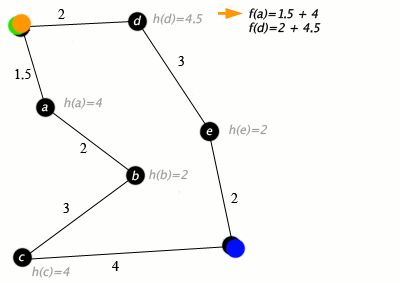
total\_path.append(current)

return total\_path

**Remark:** the above pseudocode assumes that the heuristic function is *monotonic* (or [consistent](https://en.wikipedia.org/wiki/Consistent_heuristic), 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:

[](https://en.wikipedia.org/wiki/File:AstarExampleEn.gif)

**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.

## Properties

Like [breadth-first search](https://en.wikipedia.org/wiki/Breadth-first_search), A\* is *complete* and will always find a solution if one exists provided d ( x , y ) > ε > 0 {\displaystyle d(x,y)>\varepsilon >0} d(x,y)> ε >0 for fixed ε {\displaystyle \varepsilon }ε. If the heuristic function *h* is [admissible](https://en.wikipedia.org/wiki/Admissible_heuristic), 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](https://en.wikipedia.org/wiki/Consistent_heuristic)) for A\* to be optimal. This means that for any pair of adjacent nodes *x* and *y*, where d(x,y) d ( x , y ) {\displaystyle d(x,y)} d(ddenotes the length of the edge between them, we must have:

h(x)≤d(x,y)+h(y)h ( x ) ≤ d ( x , y ) + h ( y ) {\displaystyle 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) ≤ L(X) + d(x,y) + h(y) = L(y) + h(y)h ( x ) ≤ d ( x , y ) + h ( y ) {\displaystyle h(x)\leq d(x,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](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm).) Monotonicity implies admissibility when the heuristic estimate at any goal node itself is zero, since (letting *P* = (*f*,*v*1,*v*2,...,*vn*,*g*) be a shortest path from any node *f* to the nearest goal *g*):

h(f) ≤ d(f,v1) + h(v1) ≤ d(f, v1) + d(v1,v2) + h(v2) ≤ ... ≤ 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 nodes.

### Special cases

[Dijkstra's algorithm](https://en.wikipedia.org/wiki/Dijkstra%27s_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]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-geospatial-5)[[5]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-pythalgs-6) General [depth-first search](https://en.wikipedia.org/wiki/Depth-first_search) can be implemented using the 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. It should be noted, however, that 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](https://en.wikipedia.org/wiki/LIFO_%28computing%29) manner, A\* will behave like [depth-first search](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Fibonacci_heap) can perform the same decrease-priority operations in constant [amortized time](https://en.wikipedia.org/wiki/Amortized_time).

## Admissibility and optimality

A\* is [admissible](https://en.wikipedia.org/wiki/Admissible_heuristic) 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](https://en.wikipedia.org/wiki/Admissible_heuristic). Otherwise, A\* is not guaranteed to expand fewer nodes than another search algorithm with the same heuristic.[[6]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-7)
* 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](https://en.wikipedia.org/wiki/Incremental_heuristic_search) algorithms.[[7]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-8)

### 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. 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 path. This new guarantee is referred to as *ε*-admissible.

There are a number of *ε*-admissible algorithms:

* Weighted A\*/Static Weighting.[[8]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-9) If *ha*(*n*) is an admissible heuristic function, in the weighted version of the A\* search one uses *hw*(*n*) = *ε ha*(*n*), *ε > 1* as the heuristic function, and perform the A\* search as usual (which eventually happens faster than using *ha* 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]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-pearl84-10)
* Dynamic Weighting[[10]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-11) uses the cost function f(n) = g(n) +(1 +εw(n))h(n), where,

and where d(n) is the depth of the search and *N* is the anticipated length of the solution path.

* Sampled Dynamic Weighting[[11]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-12) uses sampling of nodes to better estimate and debias the heuristic error.
* A ε ∗ {\displaystyle A\_{\varepsilon }^{\*}}A\*ε.[[12]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-13) uses two heuristic functions. The first is the FOCAL list, which is used to select candidate nodes, and the second *hF* is used to select the most promising node from the FOCAL list.
* *Aε*[[13]](https://en.wikipedia.org/wiki/A*_search_algorithm" \l "cite_note-14) selects nodes with the function A Af(n) + BhF(n), where *A* and *B* are constants. If no nodes can be selected, the algorithm will backtrack with the function Cf(n) + DHF(n), where *C* and *D* are constants.
* AlphA\*[[14]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-15) attempts to promote depth-first exploitation by preferring recently expanded nodes. AlphA\* uses the cost function f α ( n ) = ( 1 + w α ( n ) ) f ( n ) {\displaystyle f\_{\alpha }(n)=(1+w\_{\alpha }(n))f(n)} fα(n) = (1 + wα(n))f(n), where

where *λ* and *Λ* are constants with *λ≤Λ*, *π*(*n*) is the parent of n, and *ñ* is the most recently expanded node.

## Complexity

The [time complexity](https://en.wikipedia.org/wiki/Computational_complexity_theory) of A\* depends on the heuristic. In the worst case of an unbounded search space, the number of nodes expanded is [exponential](https://en.wikipedia.org/wiki/Exponential_time) in the depth of the solution (the shortest path) *d*: *O*(*bd*), where *b* is the [branching factor](https://en.wikipedia.org/wiki/Branching_factor) (the average number of successors per state).[[15]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-aima-16) 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 *bd* 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]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-aima3-17)

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

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

The time complexity is [polynomial](https://en.wikipedia.org/wiki/Polynomial_time) 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](https://en.wikipedia.org/wiki/Logarithm) of the "perfect heuristic" *h*\* that returns the true distance from *x* to the goal.[[9]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-pearl84-10)[[15]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-aima-16)

## 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]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-nilsson-3) It finds applications to diverse problems, including the problem of [parsing](https://en.wikipedia.org/wiki/Parsing) using [stochastic grammars](https://en.wikipedia.org/wiki/Stochastic_context-free_grammar) in [NLP](https://en.wikipedia.org/wiki/Natural_language_processing).[[17]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-18) Other cases include an Informational search with online learning [[18]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-WPCleanerAuto1-19)

## Relations to other algorithms

What sets A\* apart from a [greedy](https://en.wikipedia.org/wiki/Greedy_algorithm) best-first search algorithm is that it takes the cost/distance already traveled, *g*(*n*), into account.

Some common variants of [Dijkstra's algorithm](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm) can be viewed as a special case of A\* where the heuristic h(n) = 0 h(h for all nodes;[[4]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-geospatial-5)[[5]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-pythalgs-6) in turn, both Dijkstra and A\* are special cases of [dynamic programming](https://en.wikipedia.org/wiki/Dynamic_programming).[[19]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-20) A\* itself is a special case of a generalization of [branch and bound](https://en.wikipedia.org/wiki/Branch_and_bound)[[20]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-21) and can be derived from the primal-dual algorithm for [linear programming](https://en.wikipedia.org/wiki/Linear_programming).[[21]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-22)

### Variants of A\*

* Anytime Repairing A\* (ARA\*)[[22]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-23)
* [Block A\*](https://en.wikipedia.org/wiki/Block_A*)
* [D\*](https://en.wikipedia.org/wiki/D*)
* [Field D\*](https://en.wikipedia.org/wiki/Any-angle_path_planning)
* [Fringe](https://en.wikipedia.org/wiki/Fringe_search)
* [Fringe Saving A\* (FSA\*)](https://en.wikipedia.org/wiki/Incremental_heuristic_search)
* [Generalized Adaptive A\* (GAA\*)](https://en.wikipedia.org/wiki/Incremental_heuristic_search)
* [IDA\*](https://en.wikipedia.org/wiki/IDA*)
* Informational search [[18]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-WPCleanerAuto1-19)
* [Jump point search](https://en.wikipedia.org/wiki/Jump_point_search)
* [Lifelong Planning A\* (LPA\*)](https://en.wikipedia.org/wiki/Incremental_heuristic_search)
* [Simplified Memory bounded A\* (SMA\*)](https://en.wikipedia.org/wiki/SMA*)
* [Theta\*](https://en.wikipedia.org/wiki/Theta*)
* [Anytime A\*](https://en.wikipedia.org/wiki/Anytime_A*) [[23]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-24)
* Realtime A\* [[24]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-25)
* [Anytime Dynamic A\*](https://en.wikipedia.org/w/index.php?title=Anytime_Dynamic_A*&action=edit&redlink=1)

A\* can also be adapted to a [bidirectional search](https://en.wikipedia.org/wiki/Bidirectional_search) algorithm. Special care needs to be taken for the stopping criterion.[[25]](https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-26)

Alte site-uri

http://mnemstudio.org/path-finding-a-star.htm

http://www.redblobgames.com/pathfinding/a-star/introduction.html

Observații generale despre algoritmul A\*

Are la bază estimarea euristică h(x) a costului de la nodul curent x până la nodul la care se dorește să se ajungă. Această estimare trebuie să fie admisibilă = h(x) să fie mai mică decât costul efectiv până acolo (de obicei distanța euclidiană sau city block - în cazul unei grile discrete)

Se pornește de la nodul de start: la fiecare pas costul curent pentru nodul curent x = costul de la start pana x. Apoi se studiază vecinii și se pun într-o coadă de priorități, unde prioritatea unui vecin v = costul nodului x + costul muchie(x,v) + h(v).

La pasul următor se extrage din coada de priorități nodul cu prioritate minimă.