

A* Search Algorithm

Student ID: 720052156

Abstract

The A* Search algorithm is the undisputed best general traversal algorithm for connected graphs. This report covers some of its strengths, weaknesses, and the ways it can be specialised and strengthened.

I certify that all material in this report which is not my own has been identified.

Word Count: 1,187

The A* search algorithm is a pathfinding algorithm which finds paths using known distance travelled and some heuristic value associated with each node (typically estimated proximities to a destination). Using these two combined values, it is able to both ensure certainty of optima (completeness) and cut down on run time compared to its respective parent algorithms. One such parent algorithm, Dijkstra’s algorithm, does not use a heuristic value, meaning it can be time consuming and visit most if not all nodes that aren’t included in the shortest path. Another, best-first search, only finds paths based on heuristic data, leading to fast, but usually suboptimal paths. In this way, the A* algorithm can be seen as the direct combination of both these concepts, taking the positives from each with very minor and situational drawbacks for doing so. (Cox 2024) Lending to its efficiency and efficacy, A* is something of a standard for most lowest-cost graph traversal needs, and when it isn’t used, a derivative of it often is. Graph theory being as broadly applicable as it is(S 2024), this means that A* has a long list of use cases. If it can be modelled as a connected, weighted graph (even if uniformly weighted), A*, or some variant thereupon, is almost always applicable. To name a few use cases:

- Machine learning(Meng et al. 2024)
 - The relevance weighting of words in a natural language model can be ascertained using documents and passages with heuristics representing relevance
- Video games(Barnouti, Al-Dabbagh, and Naser 2016)
 - Some non-player-controlled entity may need to find the shortest path from one place to another in a 2D or 3D space.
- Logistics planning(Zeng et al. 2024)
 - Timings can be calculated easier by pathfinding using heuristics which factor in not only distances but path reliability
- Route planning(Singh 2023)
 - In a very obvious sense, it can be used to pathfind in real world scenarios.
- Real-time Network Optimisation(Cai and Ji 2018)
 - Routing packets in the most efficient way across both local and global networks is vital to maintain acceptable latency

For some of these purposes, A* is often adapted into one of its variants, all of which tend to make small tweaks either to adapt it to a particular input type, (such as in the case of Fringe search(Björnsson et al. 2005)), turn a discrete path expressed as a sequential list of individual points as a more efficient continuous path (such as in the case of Block A*(Yap et al. 2011)) or mitigate some of the memory costs (such as in the case of Simplified Memory Bounded A*(Projcheski 2020)). These are situational improvements, but broadly speaking, A* is considered the single best generic shortest path algorithm for connected graphs.

Despite the numerous variations upon A*, it could be argued that the greatest come from within, as the method for estimating proximity of a node to a destination can be one of a few heuristics(Patel, n.d.). The best heuristics to use depend on the value of accuracy in terms of computational cost, and mode of traversal. If an exact heuristic is needed, the shortest path between every possible pair of nodes can be stored, but this is often more costly than is considered worthwhile. One reason for this is that in the situations in which every node requires a shortest distance to one another to be calculated, Dijkstra’s algorithm is typically the more useful option as it goes about mapping out every node by default. If an approximate heuristic is needed, often Manhattan distance or Diagonal distance are often the most efficient (which is to say computationally cheapest for reasonably high return), but for a greater level of specificity, Euclidean distance can also be used, though this runs the risk of diminishing returns. When in or completely free traversal spaces wherein coordinates can be any real number, Euclidean distance will always find a linear, direct path, though these spaces

often create new issues, since typical A^* requires nodes with discrete positions at time of discovery (Patel, n.d.). This can, however, be circumvented using D^* - which exists to handle the freespace assumption and dynamic weighting (Stentz 1994) - or by imposing an arbitrary Cartesian grid to simplify it. All that said, heuristics don't strictly have to correspond to proximity to the end node, in fact, heuristics can correspond to anything that can be reduced to a numerical value (if it is not one already). Due to the diversity of potential heuristics, the potential effects of A^* are impossible to comprehensibly describe. Far from the drawback this sounds like, however, this allows for a greater complexity of graph representation that can be used for. Additional heuristics are never a prerequisite for A^* , but they can lead to variations that are better suited to more particular needs. In terms of complexity, A^* and Dijkstra are both difficult to define. This is because a large part of their efficiency hinges on the data structures they use, and the manner in which they're implemented. For example, the priority queue structure which gives the best worst-case time complexity, the fibonacci heap, is both expensive in terms of memory, requiring each node to have several descriptive variables associated with it, leading to poor scaling in practice. However, caveats aside, the worst-case time complexity for Dijkstra's algorithm with a priority queue is $O(E \log V)$ where E is the number of edges and V is the number of nodes (vertices) (Baeldung 2024). The theoretical worst-case for A^* , is also $O(E \log V)$, which is interesting since it is considered a direct upgrade. The confounding factor, with these observations is that the worst-case is very niche, and, in the case of A^* , the time complexity is so tremendously affected by choice and implementation of heuristic that observing a useful expression of time complexity requires a more in-depth observation of the traversal space in question. Issues arise from this, however, when spaces are infinite, or the region within the space being traversed is not a representative average for the terrain of the entire space, leading to the efficacy of A^* being, in some situations, very difficult to ascertain. Beyond noting that it will likely outperform Dijkstra's algorithm, there are too many factors to give a universally useful expression.

As for space complexity, A^* is very typical for graph traversal algorithms, as each node must be stored in memory along with its attributes. This leads to a space complexity of $O(V)$. This, however, is not as innocent as it seems, as the amount of data stored about each node varies drastically based on the type of graph and heuristic complexity.

Figure 1 is a pseudocode representation of the A^* algorithm I used for this report. I saved on time by representing my priority queues as python dictionaries, a type of hash table. This allows for the pop function to have $O(1)$ complexity, the update function (simplified in line 11 as a direct dictionary swap) has a complexity of $O(\log E)$. This is a generalised version of the algorithm I wrote. Additionally, the heuristics and distance data are precomputed, meaning that referencing them can also be done in $O(1)$ time. This leads to very fast loading times. This is also supported by the graph in question (a maze of variable size generated using Kruskal's Maze algorithm) being a discrete Cartesian grid with orthogonal movement.

```

// this function assumes a graph data type representing an orthogonal cartesian grid with a maze
// inside
// the graph data type contains:
// heuristic calculations
// edge and node data
// a method to get neighbours for a given node
// a method to get cost between 2 given neighbours
// these are not included because there are several representations one could use to get these values.

def AStarSearch(graph):
    // all coords,
    // their initialised distances (an arbitrarily high number, to avoid needed a "visited" list)
    // their preprogrammed heuristic values
    nodes ← {source: [graph.d(node), graph.h(node)] for node in graph}
    // parents of nodes such that path can be traced
    parents ← {node: None for node in graph}
    // a list of costs mapped to nodes, such that
    // with just the smallest cost, a node can be popped in the same way as from a priority queue
    unvisited ← {sum(source): source }
    // uniform cost (for all intents and purposes, may as well always be 1)
    UNIFORM_COST ← 1

    // Standard Dijkstra loop
    while unvisited is not empty:
        // this pop function finds the shortest distance, and picks a random node from the set of
        // nodes which have that cost
        current_node ← unvisited.pop(min(unvisited.keys()))
        // terminate if the destination has been reached
        if current_node = destination:
            break

        for each neighbour in graph.neighbours(current_node):
            // since heuristic data is static, precomputed, and used in the pop function in line 6
            // only nodes' distances from source need be compared
            cost ← graph.get_cost(current_node, neighbour)
            if nodes[current_node][0] + cost < nodes[neighbour][0]:
                // update the neighbour's distance to be
                // that of its shortest path neighbour + the uniform distance (in this case 1)
                cost ← nodes[current_node][0] + cost
                nodes[neighbour][0] ← cost
                // adds neighbour to the unvisited pseudo-priority queue
                // In practice, some error handling regarding whether a set exists to be added to or
                // needs to be created must be included, but this is abstracted.
                unvisited[nodes[cost].add(neighbour)
                // update the neighbour's parentage for path tracing
                parents[neighbour] ← current_node

    // a path tracing loop to find the resulting path, working backwards from the destination
    path ← []
    while destination ≠ source:
        path.append(destination)
        destination ← parents[destination]
    return reverse(path)

```

Figure 1: An A* algorithm in Pseudocode

1 Appendix: GenAI Report

1.1 Main Report

1.1.1 GenAI and Prompt

GenAI used: ChatGPT 4o mini

Prompt 1: "You are a computer science final year student. Write the main principles about the A* search algorithm."

1.1.2 Result

The A* (A-star) search algorithm is a popular and efficient pathfinding and graph traversal algorithm widely used in artificial intelligence, particularly in robotics, game development, and navigation systems. It is used to find the shortest path from a start node to a goal node in a weighted graph, combining the advantages of Dijkstra's algorithm (which guarantees the shortest path) and greedy best-first search (which attempts to find a solution quickly). Here are the key principles behind the A* algorithm:

1. Heuristic Function ($h(n)$)

- The A* algorithm uses a heuristic function to estimate the cost from the current node to the goal. This heuristic, denoted as $h(n)$, provides an estimate of the remaining distance or cost to reach the goal node from node n .
- The heuristic must be admissible, meaning it should never overestimate the actual cost to reach the goal (it should be optimistic). If it satisfies this condition, A* is guaranteed to find the optimal path.
- Common heuristics for grid-based pathfinding include Manhattan distance (for grids with horizontal and vertical movement) and Euclidean distance (for diagonal movement).

2. Cost Function ($g(n)$)

- The cost function $g(n)$ represents the exact cost from the start node to the current node n . This cost is accumulated as the algorithm traverses from the start to the goal.
- The $g(n)$ value is updated as the algorithm explores different paths in the search space.

3. Evaluation Function ($f(n)$)

- The evaluation function $f(n)$ is a combined score used by A* to determine which node to expand next. It is computed as:

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

Where:

- $g(n)$ is the cost to reach node n from the start node.
- $h(n)$ is the estimated cost from node n to the goal (heuristic).
- Nodes with the smallest $f(n)$ values are prioritized for expansion.

4. Open and Closed Sets

- Open Set: This is a priority queue (or min-heap) that holds nodes that have been discovered but not yet fully explored. Nodes are added to the open set when they are encountered, and they are removed once expanded. The algorithm selects nodes from the open set based on the lowest $f(n)$ value.

- Closed Set: This contains nodes that have already been explored and expanded. Once a node is moved to the closed set, it will not be reconsidered.
- A* avoids revisiting nodes by checking whether they are in the closed set, which helps to avoid cycles and redundant calculations.

5. Optimality and Completeness

- Optimality: If the heuristic $h(n)$ is admissible (does not overestimate the true cost to the goal), A* is guaranteed to find the optimal path, i.e., the shortest path in terms of cost.
- Completeness: A* is complete, meaning that it will always find a solution if one exists, assuming the search space is finite and the cost function does not cause infinite loops.

6. A in Practice*

- A* algorithm works iteratively by expanding the node with the lowest $f(n)$ value from the open set. After expanding a node, A* generates its neighbors and updates their $g(n)$, $h(n)$, and $f(n)$ values.
- If a neighbor node has not been visited or a better path to it is found (i.e., the new $g(n)$ is lower than the previous one), it is added to the open set and its data is updated.

7. Time and Space Complexity

- The time complexity of A* depends on the number of nodes in the search space and the efficiency of the data structure used to store the open set. The worst-case time complexity is $O(b^d)$, where b is the branching factor and d is the depth of the solution.
- The space complexity is also $O(b^d)$ because the open set can grow to include all nodes at a certain depth in the worst case.

8. Variants of A Algorithm*

- Memory-bounded A (MA)**: A variant that uses less memory by bounding the amount of memory available for storing the open set.
- Anytime A (AA)**: An A* variant that can provide a solution at any time with a guaranteed improvement over time as more resources are available.
- A with Weighted Heuristic*: Sometimes, the heuristic is weighted to make the search faster at the cost of optimality (less accurate but faster solutions).

9. Applications of A Search Algorithm*

- Pathfinding in games: A* is widely used in video games for character movement, navigation, and obstacle avoidance.
- Robotics: In autonomous robots, A* is used for navigation, helping robots find the shortest route in dynamic environments.
- AI for route planning: In navigation systems, A* helps calculate the best route for vehicles, drones, or other forms of transportation.

By combining an actual cost ($g(n)$) and a heuristic estimate ($h(n)$), A* strikes a balance between exploration and exploitation, ensuring that it explores promising paths while guaranteeing an optimal solution when possible.

```

function A*(start, goal):
    // Initialize the open and closed sets
    openSet := {start}
    closedSet := {}

    // Initialize the cost and heuristic functions
    g(start) := 0 // Cost from start to start is zero
    h(start) := heuristic(start, goal) // Heuristic estimate from start to goal
    f(start) := g(start) + h(start) // f(n) = g(n) + h(n)

    // Initialize parent map to reconstruct the path later
    cameFrom := {}

    // Loop until the open set is empty
    while openSet is not empty:
        // Get the node in the open set with the lowest f(n)
        current := node in openSet with the lowest f(n)

        // If current node is the goal, reconstruct the path and return it
        if current == goal:
            return reconstruct_path(cameFrom, current)

        // Remove current from the open set and add it to the closed set
        openSet.remove(current)
        closedSet.add(current)

        // Loop through neighbors of the current node
        for each neighbor of current:
            // If the neighbor is in the closed set, ignore it
            if neighbor in closedSet:
                continue

            // Calculate tentative g(n) score
            tentative_g := g(current) + distance(current, neighbor)

            // If neighbor is not in the open set or tentative g(n) is better, update the neighbor
            if neighbor not in openSet or tentative_g < g(neighbor):
                // Set parent of neighbor to current
                cameFrom[neighbor] := current

                // Update g, h, and f for neighbor
                g(neighbor) := tentative_g
                h(neighbor) := heuristic(neighbor, goal)
                f(neighbor) := g(neighbor) + h(neighbor)

            // If neighbor is not in openSet, add it
            if neighbor not in openSet:
                openSet.add(neighbor)

    // If we exit the loop, no path was found
    return failure

function reconstruct_path(cameFrom, current):
    totalPath := [current]
    while current in cameFrom:
        current := cameFrom[current]
        totalPath.append(current)
    return reverse(totalPath)

```

Figure 2: An A* algorithm in Pseudocode generated by GPT4o mini

1.2 Pseudocode

1.2.1 GenAI and Prompt

GenAI used: ChatGPT 4o mini

Prompt 2: "write the pseudocode describing the A* search algorithm"

1.2.2 Result

Key Points: 1. Initialization:

- The openSet starts with the start node, and the closedSet is empty.
- The cost ($g(n)$) for the start node is 0, and the heuristic ($h(n)$) is calculated based on the start node's estimated cost to reach the goal.

2. Main Loop:

- The algorithm repeatedly selects the node with the lowest $f(n)$ (where $f(n) = g(n) + h(n)$) from the open set to explore.
- If the goal node is reached, the algorithm reconstructs and returns the path from the start to the goal using the `cameFrom` map.

3. Node Expansion:

- For each neighbor of the current node, the algorithm calculates the tentative `g` (the cost to reach that neighbor through the current node).
- If the neighbor has not been visited or a better path (lower $g(n)$) is found, the neighbor's values (`g`, `h`, and `f`) are updated and the neighbor is added to the open set.

4. Termination:

- The algorithm terminates when the goal node is found, and the path is reconstructed.
- If the open set is empty and the goal has not been reached, it means no path exists, and the algorithm returns failure.

Functions:

- `heuristic(n, goal)`: A function that estimates the cost to reach the goal from node `n`.
- `distance(n1, n2)`: A function that computes the actual distance (or cost) between two nodes, typically based on the problem's specific cost model (e.g., Euclidean distance, Manhattan distance).
- `reconstruct_path(cameFrom, current)`: This function traces the path from the goal node back to the start node using the `cameFrom` map.

This pseudocode captures the essential behavior of the A* search algorithm, including the use of the open and closed sets, the cost and heuristic functions, and the process of path reconstruction.

1.3 Report Comparison and GenAI appraisal

AI generated report goes into very explicit, instructional detail as to how the most general and standardised A* search algorithm may be constructed. This is, naturally, very beneficial, as, for most purposes, the most standard, boilerplate version will do, but it fails to give any form of appraisal, or in cases even acknowledgement for the level of choice one has when implementing an A* algorithm, which I consider to be an oversight that one's understanding of A* is incomplete without, which is a respect in which I think my report, although necessarily shorter, succeeded it. The pseudocode was, naturally, flawless, although it was incredibly derivative of the one shown on Wikipedia, to an extent that it doesn't give particularly novel insight. Generative AI, unsurprisingly at this point in time, should only be used to corroborate basic factoids to prompt more guided research, as its perceived reliability only creates graver consequences for a non-rigorous cross-referencing process. I think that in an ideal world it occupies only this space in the educational process, but I also think at a time where the value of rote memorization has long since been called into question, more consideration ought to be paid into the value of certain types of academic assessments in all respects, as GenAI seems to be here to stay. It's useful for researching algorithms, undeniably, but, in my opinion, less reliably useful than simply putting an hour or so into reading up on literature, as the doctrine of algorithms necessarily attempts to be comprehensive in its descriptions and does not always serve as a replacement of this process, particularly for more niche or under-documented algorithms.

References

- Baeldung. 2024. “Understanding Time Complexity Calculation for Dijkstra Algorithm.” *Baeldung CS*, <https://www.baeldung.com/cs/dijkstra-time-complexity>.
- Barnouti, Nawaf Hazim, Sinan Sameer Mahmood Al-Dabbagh, and Mustafa Abdul Sahib Naser. 2016. “Pathfinding in Strategy Games and Maze Solving Using A* Search Algorithm.” *Journal of Computer and Communications* 4:14–25. <https://doi.org/10.4236/jcc.2016.411002>.
- Björnsson, Yngvi, Markus Enzenberger, Robert Holte, and Jonathan Schaeffer. 2005. “Fringe Search: Beating A* at Pathfinding on Game Maps.” January.
- Cai, Yuan, and Xiang Ji. 2018. “ASA-routing: A-Star Adaptive Routing Algorithm for Network-on-Chips.” In *Algorithms and Architectures for Parallel Processing*, edited by Jaideep Vaidya and Jin Li, 187–198. Cham: Springer International Publishing. ISBN: 978-3-030-05054-2.
- Cox, Graham. 2024. “<https://www.baeldung.com/cs/a-star-algorithm>.” *Baeldung CS*.
- Meng, Silin, Yiwei Wang, Cheng-Fu Yang, Nanyun Peng, and Kai-Wei Chang. 2024. *LLM-A*: Large Language Model Enhanced Incremental Heuristic Search on Path Planning*. Technical report. University of California.
- Patel, Amit. n.d. “Heuristics.” *Amit’s Thoughts on Pathfinding*, <https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#a-starx27s-use-of-the-heuristic>.
- Projcheski, Filip. 2020. “How SMA* (Simplified Memory Bounded A*) Algorithm Works.” *Laconic Machine Learning*, <https://laconicml.com/simplified-memory-bounded-a-star/>.
- S, Ravikiran A. 2024. “A* Search Algorithm Explained: Applications Uses.” *Simplilearn*, https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm#applications_of_a_algorithm.
- Singh, Sachin. 2023. “The Algorithms Behind The Working Of Google Maps.” *Medium*, <https://medium.com/@sachin28/the-algorithms-behind-the-working-of-google-maps-73c379bcc9b9>.
- Stentz, A. 1994. “Optimal and efficient path planning for partially-known environments.” In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, 3310–3317 vol.4. <https://doi.org/10.1109/ROBOT.1994.351061>.
- Yap, Peter, Neil Burch, Rob Holte, and Jonathan Schaeffer. 2011. “Block A*: Database-Driven Search with Applications in Any-angle Path-Planning,” <https://webdocs.cs.ualberta.ca/~holte/Publications/aaai11PeterYapFinal.pdf>.
- Zeng, Xianyang, Jiawang Zhang, Wenhui Yin, Hongli Yang, Hao Yu, Yuansheng Liang, and Jinwu Tong. 2024. “Path planning strategies for logistics robots: Integrating enhanced A-star algorithm and DWA.” *Electronics Letters* 60 (November). <https://doi.org/10.1049/ell2.70090>.