Informed Route Planning Algorithms

Jonas Daniel Blocher
Technische Hochschule Ulm
B.Sc. Computer Science

January 27, 2023

Abstract

An introduction to informed route planning algorithms, giving implementation details and examples on A* and showing a selection of modern approaches to further speed up search queries, including evaluation and efficiency comparisons.

1 Introduction

Calculating the fastest or shortest route between two places is a problem everyone regularly encounters when traveling. But this is not limited to car navigation systems, shortest path problems occur in way more different fields: From routing of network traffic over constrained robot movement to optimizing the layouts of printed circuit boards or solving of more abstract problems and proving mechanical theorems, there is a wide range of applications that involve this general problem. Conceived in the 1950s, Dijkstra's algorithm solves it and is guaranteed to always find the shortest path. Although this traditional approach is still widely adapted, faster solutions were needed. Informed route planning algorithms use additional information about the network or graph to speed up search queries. Among these A* (pronounced A-Star) is probably the most popular, due to its completeness, efficiency, and flexibility. But even though A* is a lot faster than Dijkstra's algorithm, it alone cannot meet the performance requirements of modern navigation systems that have to find results in a fraction of seconds for giant networks. That's were modern techniques combine clever optimization

of the network with precomputed information to further shrink down query times. Recent studies also investigated how A* could be adapted for efficient navigation of service robots or planning of drone flight paths.
This paper should serve as an introduction to informed route planning algorithms. It will explain the fundamentals, give details and examples on how A* works and show off some of the techniques that are used for further speed improvements, including an efficiency comparison.

2 Fundamentals of Route Planning

2.1 Shortest Path Problem

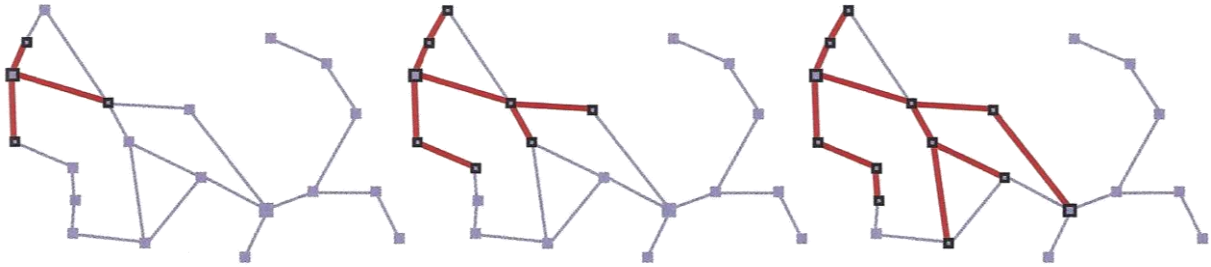
We consider a Graph $G = (V, E)$ where V is a set of nodes and E a set of edges, with each connecting two nodes (u, v). Every edge has a positive weight $w(u, v)$. For each path between a starting node s and a target node t we can calculate the total weight, also called *cost*, $c(s, t)$ of all edges in the path. The shortest path between s and t is the path with the minimum total cost.[1], [2]
This is also referred to as *single-pair shortest path problem* (as it is looking for the shortest path between a pair of points) to distinguish it from variations where all shortest paths for either a given start or end node are desired. Practical applications of the shortest path problem are seen in navigation systems and routing of communication networks, but also in operations research and circuit integration.

2.2 Static Routing

Static routing is a specific case of the shortest path problem where the edge weights do not change, contrary to dynamic routing. This allows routing algorithms to precompute and permanently store shortest paths between pairs of nodes. Reusing such precomputed information allows accelerated processing of queries. This results in a tradeoff between query time, preprocessing time and space usage of preprocessed information. [1]

2.3 Best-first Search

To search for paths from node s to t we superimpose a search tree over the Graph G with s as root node. This tree describes the possible paths from s, reaching towards the goal t. The nodes of the search tree may be expanded, by considering the corresponding node n from G, looking for nodes that are reachable directly from that node but have not yet been added to the search tree. All these are added to the search tree as children of n. Each expansion step may create new leave nodes in the search tree that may be expanded next.[3]



sequence of search trees, where on each step all nodes of the frontier are expanded. Taken from [3]

This divides up G into an *interior* region of already expanded nodes, and an *exterior* region with so far unreached nodes. The reached but unexpanded leave nodes are therefore the *frontier* of the search tree. Depending on the search algorithm used, the criteria which node should be expanded first differ. Best-first search algorithms define an evaluation function $f(n)$ that returns a value for each Node n and always expand the one with the lowest value first.[3]

2.4 Priority Queues

Best-first search algorithms must manage the nodes at the frontier of the search tree in a way that allows fast addition of nodes on expansion, as well as fast access to the unexpanded node with the lowest evaluation function value $f(n)$. Therefore, the frontier nodes are stored in a priority queue. Usually the elements of a priority queue are stored alongside their priorities in a mini-heap to allow $\Theta(log\, n)$ complex insert and extract-min operations. New elements are added to the queue using $insert(x)$

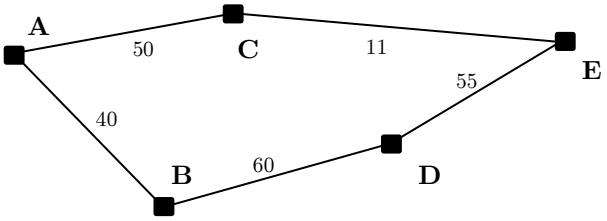
operation, which adds the new element at the end of the heap. To re-establish the minimum heap property the element is swapped with its parent in the heap, till the parent either has a bigger value or the element is the root of the heap.

$extractMin()$ retrieves the root of the heap, using the element at the end of the heap as new root. This element is then swapped with its child with the lowest priority, until it has a smaller priority than both child's or until it has no children.

Alternative heap implementations like a Binomial- or a Fibonacci Heap provide $\Theta(1)$ insertion [2] which is able to speed up the search algorithm, but for routing in large networks "the impact [...] is rather limited since cache faults for accessing the graph are usually the main bottleneck".[1]

2.5 Dijkstra's Algorithm

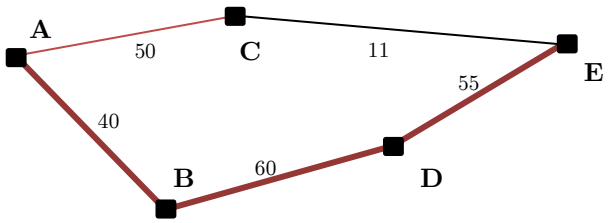
Dijkstra's Algorithm is the classical approach to solving shortest path problems and was published by Edsger Wybe Dijkstra in 1959 [4]. It is a best-first search algorithm that uses the paths total cost $c(s, n)$, from the root s to the current node n as evaluation function $f(n)$. This leads to paths spreading out from s in waves of uniform path cost, till a path to the target node t is found. Therefore, the algorithm is also called *uniform-cost search*. The following example should contribute to a better understanding:



example inspired by [3]

The problem is to get from A to E. A is connected to B and C over edges with a cost of $c(A, C) = 50$ and $c(A, B) = 40$. So, B is the next node with the least cost and therefore expanded next, adding node D with $c(A, D) = 40 + 60 = 100$. So now the frontier of the search tree consists of nodes D and C. As $c(A, C) < c(A, D)$ the next node to expand is C. On expansion, E is added to the frontier with a cost of $c_1(A, E) =$

$50 + 110 = 160$. The algorithm continues, although E is the target, as it only checks if a node is the target on expansion. The current cost $c(A, E) > c(A, D)$, therefore D is the next node to expand, adding a second path to E with cost $c_2(A, E) = 40 + 60 + 55 = 155 < c_1(A, E)$. As this path has a lower cost it replaces the previous path to E, adding it to the frontier. As there are no more nodes on the frontier with lower costs, E is next to expand, the goal is reached and the found path is returned as solution.



search tree with **shortest path**

As this algorithm expands outwards from the starting node uniformly, there is no direction towards the target. This results in it being relatively slow for large (road) networks compared to goal directed search Algorithms. See chapter 0 for more details on time complexity. Dijkstra's Algorithm is *complete* and *cost-optimal*, meaning that is guaranteed to find the fastest path or report a failure if no such path from source to target exists.

3 Goal Directed Search

3.1 A*

First published 1968 by Peter Hart, Nils Nilsson and Bertram Raphael in [5], A* utilizes the intuition that the shortest path to a target commonly leads in the direction of the target. Therefore, the growth of the search tree is directed towards its goal, categorizing it as *goal directed search*. *Greedy* search algorithms use a heuristic to guide themselves on each expansion as close towards the target as possible, reducing time complexity of the algorithm while returning non cost optimal results. A* combines the speed advantage of such a heuristic with the completeness of uniform cost search, by selecting

new nodes based on a combination of the paths previous cost and the heuristic cost towards the target. [6]

In fact, A* is defined as a best-first search algorithm with an evaluation function of:

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

$g(n)$ is the cost of travelling from the start node s, to the current node n. $h(n)$ is a heuristic function for the estimated cost of the shortest path from n to the target node t. So $f(n)$ is the estimated total cost of the best path from s through n to t. [3]

Both time complexity and quality of the result are highly dependent on the used heuristic. A* is only cost-optimal if an *admissible heuristic* is used, meaning that the heuristic never overestimates the cost to reach the goal.[3] Inadmissible Heuristics return bigger values than the actual cost of travelling to the target for some nodes. They reduce the time complexity of the algorithm but risk that the found path is not the shortest one. Its therefore always a trade of between time complexity and quality. If a heuristic function always returns 0, the algorithm is exactly identical to Dijkstra's Algorithm. [6] In routing it is common to use the Euclidean distance as heuristic function:

$$h(n) = \sqrt{(x_n - x_t)^2 + (y_n - y_t)^2} \qquad [7]$$

Among this x_n and y_n are the coordinates of node n and x_t and y_t the coordinates of the target.

Algorithm: A*

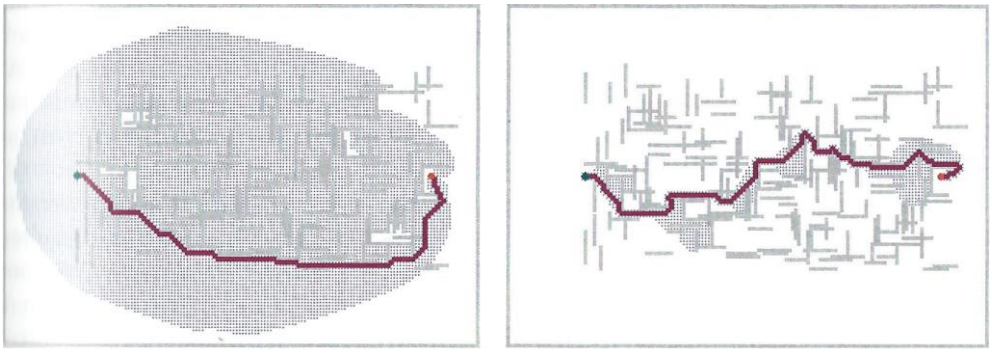
```
Let frontier as priority queue with priority f()
Let reached as empty map

frontier.insert(START)
While Frontier is not empty:
    node = frontier.extractMin()
    if node = GOAL:
        Return reconstruct path from START to GOAL using reached

    For each neighbor of node:
        neighborCost = g(node) + cost(node, neighbor)
        if neighbor not in reached or neighborCost < g(neighbor):
            reached[neighbor] = node
            g(neighbor) = neighborCost
            f(neighbor) = g(neighbor) + h(neighbor)
            frontier.insert(neighbor)

Return failure
```

3.2 Weighted A*

While A* with an admissible heuristic is complete and cost-optimal, it often expands a lot of nodes not needed for the shortest path, leading to way higher time complexity than greedy search algorithms. If we are willing to accept solutions that are not cost-optimal for the sake of time efficiency, an inadmissible heuristic may be used. This can be done by adding some weight factor to the heuristic:

$$f(n) = g(n) + W \times h(n) \quad 1 < W < \infty$$

In this example with $W = 2$ we can see how less nodes need to be expanded, but the found path is not the shortest one:



Comparison of A* and weighted A* with W=2 from [3]

3.3 Landmarks

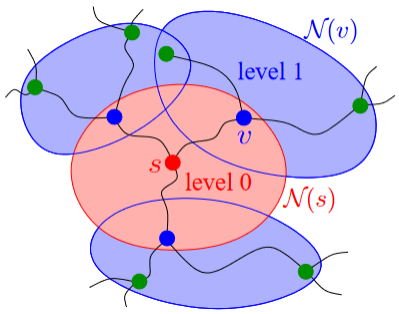
For routing in large road networks even with some optimisations and a carefully chosen heuristic, A* is still a million times slower than modern navigation services. As mentioned in 2.2, static routing allows precomputing costs of some or all paths in a network to speed up search queries. But in a Graph with N nodes and E edges, precomputing shortest paths for all nodes would take $O(|N|^2)$ space and $O(|E|^3)$ time[3]. As worldwide road networks can contain billions of nodes [8], this is obviously not feasible. Instead, a better approach is to select a small number of *landmark points* from the nodes. We can then precompute and store the cost of the shortest path $C^*(n, L)$ for all nodes n and all landmark points L. These precomputed costs are used to create an optimized heuristic such as:

$$h_{DH}(n) = \max_{L \in Landmarks} |C^*(n, L) - C^*(t, L)|$$

t is the target node, so it considers the cost of the entire path from n to L, subtracting the cost from the target node to L, which should give a lower approximation of the paths cost.[3] The closer the target t is to the shortest path from n to L, the more accurate is the heuristic. With landmarks carefully spread around the perimeter of the networks the best results are achieved.

3.4 Highway Hierarchies

Highway Hierarchies utilize the hierarchical way road networks are build (with smaller roads connecting neighbourhoods and highways connecting cities). In this approach a local area around start and target node are considered as neighbourhood node set. Outside of these neighbourhoods only a subset of important nodes is considered, the *highway network*. Edges only belong to the highway network if they appear in the shortest path that connects nodes of different neighbourhoods. The number of nodes in the network can additionally be reduced by contracting nodes that can be bypassed. The highway network is precomputed for the entire graph to speed up search queries. A *highway hierarchy* of a graph consists of several levels $G_0, G_1, \ldots, G_L$. Level 1 is obtained by computing the highway network of level 0, level 2 by computing the highway network of the core of level 1 and so on.



Highway Hierarchy with neighbourhoods & entry points [9]

Highway Hierarchies were the first speedup technique that was able to handle queries in milliseconds for the largest available road networks. For more in-depth information read on [9].

3.5 Distance Tables

As explained above hierarchical routing can greatly reduce the size of the network. Once a network has been shrunk down to a size of $\Theta(\sqrt{n})$ it is affordable to precompute all paths of a highway network and store their cost in a distance table.[1] Finding the cost of a path is then as simple as looking it up from the table.

4 Efficiency Comparison

The main 3 factors when combining the efficiency of search algorithms are space usage, preprocessing time and query time. [1] compared these parameters for all mentioned

algorithms. The measurements are selected for the European road network. All used algorithms are complete and cost optimal. For more clarity the algorithms discussed by this paper are highlighted:

method	first pub.	date mm/yy	data from	size $n/10^6$	space [B/n]	preproc. [min]	speedup
DIJKSTRA	[20]	08/59	-	18	21	0	1
separator multi-level	[67]	04/99	[36]	0.1	?	> 5 400	52
edge flags (basic)	[44]	03/04	[45]	1	35	299	523
landmark A^*	[25]	07/04	[28]	18	89	13	28
edge flags	[43,48]	01/05	[33]	18	30	1 028	3 951
HHs (basic)	[58]	04/05	[58]	18	49	161	2 645
reach + shortc. + A^*	[26]	10/05	[28]	18	100	1 625	1 559
	[28]	08/06	[28]	18	56	141	3 932
HHs + dist. tab.	[59]	04/06	[64]	18	68	13	10 364
HHs + dist. tab. + A^*	[17]	08/06	[64]	18	92	14	12 902
high-perf. multi-level	[51]	06/06	[13]	18	181	1 440	401 109
transit nodes (eco)	[3]	10/06	[64]	18	140	25	574 727
transit nodes (gen)	[3]	10/06	[64]	18	267	75	1 470 231
highway nodes	[65]	01/07	[64]	18	28	15	7 437
approx. planar $\epsilon = 0.01$	[52]	09/07	[52]	1	2 000	150	18 057
SHARC	[6]	09/07	[7]	18	34	107	21 800
bidirectional SHARC	[6]	09/07	[7]	18	41	212	97 261
contr. hier. (aggr)	[23]	01/08	[23]	18	17	32	41 051
contr. hier. (eco)	[23]	01/08	[23]	18	21	10	28 350
CH + edge flags (aggr)	[8]	01/08	[8]	18	32	99	371 882
CH + edge flags (eco)	[8]	01/08	[8]	18	20	32	143 682
transit nodes + edge flags	[8]	01/08	[8]	18	341	229	3 327 372
contr. hier. (mobile)	[62]	04/08	[62]	18	8	31	9 878

A combination of A*, Highway Hierarchies and Distance Tables was able to speed up search queries by 12.902 times compared to Dijkstra's Algorithm while increasing space usage only by 4,4. Even faster alternatives exist, like the combination of transit nodes and edge flags, which speeds up search queries by over 3 million times, but these algorithms go way beyond the scope of this document.

5 Conclusion

We have shown how informed route planning algorithms use well picked, admissible heuristics to optimize node expansions compared to Dijkstra's Algorithm. A* is utilized to find shortest paths in a complete, cost-optimal and time efficient manner. It can be

combined with Highway Hierarchies and Distance Tables to further speed up search queries, reaching time efficiencies over 12.000 times better than Dijkstra's Algorithm.

References

[1] D. Delling, P. Sanders, D. Schultes, and D. Wagner, 'Engineering Route Planning Algorithms'. Universität Karlsruhe, 2009. Accessed: Nov. 06, 2022. [Online]. Available: https://i11www.iti.kit.edu/extra/publications/dssw-erpa-09.pdf

[2] R. K. Ahuja, K. Mehlhorn, J. Orlin, and R. E. Tarjan, 'Faster algorithms for the shortest path problem', *J. ACM*, vol. 37, no. 2, pp. 213–223, Apr. 1990, doi: 10.1145/77600.77615.

[3] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Fourth edition, Global edition. Harlow: Pearson, 2021.

[4] E. W. Dijkstra, 'A note on two problems in connexion with graphs', *Numer. Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959, doi: 10.1007/BF01386390.

[5] P. E. Hart, N. J. Nilsson, and B. Raphael, 'A Formal Basis for the Heuristic Determination of Minimum Cost Paths', *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, Jul. 1968, doi: 10.1109/TSSC.1968.300136.

[6] T. Chen, G. Zhang, X. Hu, and J. Xiao, 'Unmanned aerial vehicle route planning method based on a star algorithm', in *2018 13th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, May 2018, pp. 1510–1514. doi: 10.1109/ICIEA.2018.8397948.

[7] Y. Song and P. Ma, 'Research on Mobile Robot Path Planning Based on Improved A-star Algorithm', in *2021 International Conference on Electronic Information Engineering and Computer Science (EIECS)*, Sep. 2021, pp. 683–687. doi: 10.1109/EIECS53707.2021.9588002.

[8] 'Node – OpenStreetMap Wiki'. https://wiki.openstreetmap.org/wiki/Node (accessed Dec. 05, 2022).

[9] D. Delling, P. Sanders, D. Schultes, and D. Wagner, 'Highway Hierarchies Star*'. Universität Karlsruhe, 2007. doi: 10.1090/dimacs/074.