Synopsis of Dijkstra and A-star

Single-Source Shortest Path Algorithms

Luca Labruna

Computer Science Department, Florida State University, Panama City, Florida, USA, ldl17c@my.fsu.edu

Wendy Slattery

Computer Science Department, Florida State University, Panama City, Florida, USA, wls12@my.fsu.edu

Kadin Unger

Computer Science Department, Florida State University, Panama City, Florida, USA, knu18@my.fsu.edu

ABSTRACT

Path Finding (single-source shortest path, SSSP) and graph traversal techniques elicit popular interest due to their broad and significant influence within vital aspects of daily life from network routing capabilities, transportation, economics, and even to entertainment industries. The Dijkstra and A star (A\*) algorithms are widely considered modern favorites for their efficiency at addressing the shortest path dilemma. Offered is a brief background and evolution of these algorithms, an analysis of their time and space complexities, as well as summary of their reach and impact on society garnering their fame as members of the top ten most influential algorithms.

CCS CONCEPTS

• Theory of computation ~ Design and analysis of algorithms ~ Graph algorithms analysis • Applied computing ~ Life and medical sciences • Networks ~Network algorithms ~ Control path algorithms

KEYWORDS

Single-Source Shortest Path (SSSP), Graph Traversal, Weighted Graphs, Breadth-First Search, Single-Pair Search Path, Best-First Search, Dijkstra, A-Star (A\*), Bellman-Ford, Floyd-Warshall, Relaxation, Heuristics, Efficient Time Complexity, Optimization, Greedy Algorithms, Dynamic Programming, Smart Algorithms, AI, Distant-Vector Routing Protocol, Navigation Systems, Critical Path Method (CMP), Epidemiology

1. Introduction:

The underlying premise of the shortest path problem is simply to find the path of minimum sums of adjacent weighted edges between two different vertices in a graph. What is the shortest calculated distance between point A and point Z? Such a question has broad implications in traffic and navigation systems, where it is used in directing drivers between locations, or in economic considerations within development of real-world large-scale construction projects, which seek to optimize time and reduce waste. Similarly, it is a focus in networking or telecommunication systems for seeking the shortest widest path for signal transmission and in epidemiological studies where it helps in modelling the spread of infectious diseases. Likewise, fields such as artificial intelligence, robotics, as well as the entertainment and gaming industries all benefit from efficiently designed shortest search path algorithms. Its applications are broad, numerous, and an important and lucrative problem to explore.

1. The Shortest Path Problem

There are several variations of the shortest path problem which may be defined on graphs whether they are directed, undirected, or mixed. Below are examples of the various types of shortest path searches. In order to grasp the distinctions within the Dijkstra and A\* algorithms, it is important to understand the context of the larger picture between the various schemes.

The Single-Source Shortest Path (SSSP) problem defines such algorithms as Breadth-First Search, Dijkstra, and Bellman-Ford; as well as, calculating the distance from one vertex to all vertices within the graph, eventually eliminating all paths except the most minimum. These algorithms may be easily modified to behave like single-pair shortest path algorithms by allowing the algorithm to stop as soon as the destination vertex is reached and processed. Single-Pair Shortest Path (SPSP) algorithms explore the minimal path from one vertex to another vertex. Best-First Search and the A\* algorithm both fall within this category. Single-Destination Shortest Path algorithms function as SSSPs in reverse and calculate the shortest path from all vertices to one specific vertex instead. Lastly, All-Pairs Shortest Path algorithms include those such as Johnson, Floyd-Warshall, and Seidel’s algorithms which explore shortest distance paths between every pair of vertices in the graph [23].

* 1. Breadth-First Search

Breadth First Search (BFS) is the simplest case of the shortest path problem. Given a graph where all edges have a weight of 1, the length of the shortest path between the start vertex and any other vertex is just the number of edges between the two vertices. This concept can be generalized for any case in which the weights of all the edges are equivalent, since the shortest past would correspond to the number of edges between the two vertices scaled by the weight of the edges. Breadth First Search traverses a graph by layers, starting at layer 0 where the starting vertex is located, and ending at the layer of the destination vertex. This layer number corresponds to the number of edges or the shortest path between the starting and destination vertices.

The BFS algorithm is implemented by utilizing a queue. It visits/enqueues the start vertex and then proceeds to enqueue all the adjacent (unvisited) vertices onward. It then dequeues the start vertex and similarly processes all the adjacent unvisited vertices of the vertex now at the front of the queue which represents start’s adjacent neighbor. The algorithm continues in this manner until the queue is empty, meaning that every vertex has been visited or alternatively until the destination vertex has been visited. This can be seen as when the destination vertex has reached the font of the queue. The path from the start vertex to the destination vertex can be reconstructed by saving the parent of each vertex visited. BFS algorithm can be considered a special case of the Dijkstra algorithm where all the edges of a graph have equal weights. To account for these unequal weights, Dijkstra employs a Min-Heap Priority Queue instead and consequently becomes slower than BFS.

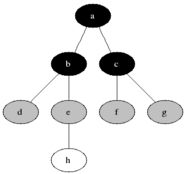


Figure 1: An in-progress Breadth First Search on a binary tree [6].

* 1. Dijkstra

Though also an SSSP, the Dijkstra algorithm operates on positively weighted graphs and employs a greedy algorithm approach, selecting the nearest adjacent unvisited vertex which implies shortest local optimal choice from one vertex to next [18]. Edge relaxation is an additional embedded process which takes the shortest calculated distance sum from the source to the current node location and considers the neighboring assigned edge weights in order to accurately determine which nodes are members of the shortest overall path. Thus, it is considered a reliable guarantee to find the shortest path. Dijkstra’s Algorithm utilizes a minimum heap priority queue to hold the unvisited vertices sorted by their current known distance from the start vertex. Though the essential structure of BFS can be found inside the Dijkstra algorithm, BFS does not utilize a priority queue nor does it perform edge relaxations. One drawback of Dijkstra’s over BFS is that it may perform unnecessary work by visiting current optimal paths that will not be part of the final global optimal solution.

* 1. Best-First Search

Best-First Search improves the run time on Dijkstra’s unnecessary computations by incorporating a different metric to drive its algorithm. This metric, called heuristics, is simplified as an educated guess of the remaining distance as it extends to the destination vertex unlike Dijkstra which keeps tallies on distances from the start to current vertex [1]. In this type of context, the algorithm is taking advantage of predictive distance information which leads it toward those nodes which are guaranteed a shortest path as long as those predictions underestimate or exactly calculate those values. Best-First Search runs quicker than Dijkstra but since it doesn’t account for the current length of the path from the start vertex it is not guaranteed to find a shortest path. The heuristic gives the algorithm a direction towards the destination vertex, allowing it to save computation time by avoiding those vertices branching away from the destination.

* 1. A-star (A\*)

Dijkstra provides accuracy and Best-First Search offers speed, whereas the A\* pathfinding algorithm exploits the strengths of both. It is likewise driven by heuristics, and depending on how those heuristics are defined, the trade-off between speed and accuracy may be calibrated. An exact calculation of distance from a node to its destination for every node in a graph can be quite time consuming; yet, in instances where a generalized distance may be assumed and still maintains good results, then the algorithm acquires the benefit of speed that might otherwise be consumed by precision. The caveat is that for the algorithm to maintain accuracy, the heuristics need to be admissible. This means they must either be exact or an underestimate of the true distance values in order to ensure a shortest path outcome. When the heuristics are overestimated, the algorithm will behave more like a greedy Best-First Search, offering speed, but not guaranteeing to find a shortest path. Ideally, the heuristic estimate will be as close as possible to the actual remaining distance from the current vertex to the destination vertex, but not greater. If the heuristics are non-informative, meaning equal to zero or a constant value, then the A\* algorithm has essentially been reverted to that of Dijkstra [1]. Heuristics are what make A\* so valuable. It is an informed variation of Dijkstra. Additionally, it’s the heuristics that allow A\* to be morphed into a variety of pathfinding algorithms, thus making it quite versatile.

The A\* algorithm is often considered a uniquely intelligent algorithm and is one of the best and popular modern techniques for pathfinding and graph traversals. Not only does it tally the sum distance from the start node forward, but the added heuristics incorporate degrees of prediction which allow this algorithm to not only follow the shortest path up to the current location within the search, but further, to greedily choose the minimal path forward toward the terminal node. In such a manner, A\* branches toward the destination at each node, thereby shedding the unproductive node searches and adding toward its efficiency.

|  |  |
| --- | --- |
|  |  |

Figure 2: Result of Dijkstra’s Algorithm and Algorithm A\* on a graph. Note how Algorithm A\* is more direct after passing the barrier [7, 5].

* 1. Bellman-Ford

In the presence of negative weight edges, the Dijkstra and A\* greedy approach algorithms don’t guarantee to find the shortest path since they rely on the fact that when all weights are nonnegative, adding an edge to a path can never make it shorter and potential paths might be calculated inaccurately. The Bellman-Ford Algorithm instead, works similar to Dijkstra but allows negative weights, thus making it available to use on a wider variety of applications, albeit running slower [10].

1. Analysis: Overview/Comparison

The Dijkstra and A\* algorithm share similarities and, in fact, A\* could be considered a modified Dijkstra. When heuristics are non-informative, zero, or a constant value, then A\* will essentially function exactly as the Dijkstra algorithm. The added heuristics are what make A\* an intelligent algorithm and afford it a large measure of performance flexibility. Because the heuristics may be measured exactly or afforded degrees of prediction, the algorithm may be tweaked to find the balance between accuracy and speed according to the needs of the program or task. They may also be used in artificial intelligence and machine learning by measuring the patterns of efficiency.

Both algorithms are greedy in moving toward the direction of the shortest path whether that be the next closest node or the minimal edge weight. Dynamic Programming is also a means of enhancing the efficiency by keeping a forward moving tally of shortest path cumulative distances. Each is a powerful tool depending on tasks and needs. Dijkstra is consistently reliable at finding the shortest path because it actually visits all the nodes in the graph and calculates the shortest path, whereby A\* branches early, shedding unnecessary node visits, yet its success is entirely dependent on the quality of the heuristics. Thus, successful use of A\* depends upon sufficient understanding of admissible heuristics. Furthermore, searches that have multiple target nodes or nodes that are closer together might benefit more from Dijkstra rather than A\* due to its comprehensive search. Sometimes, the time added to account for accuracy offers a better long-term efficiency.

* 1. Dijkstra

Dijkstra is mostly likely to be the algorithm implemented in vehicle GPS systems as it is considered a reliable algorithm for finding the shortest path distance, not just an optimal one. It is implemented broadly over all the vertices in the graph and is a good choice for those instances where information about the target node is unknown. The Dijkstra algorithm works because of two mathematical principles:

**Lemma 1.** The subpath of any shortest path is itself a shortest path.

**Lemma 2.** Triangle Inequality: For all u, v, x ∈ V, if δ(u,v) is the shortest path length between u and v, then δ(u,v) ≤ δ(u,x) + δ(x,v)

* + 1. The Algorithm Stepwise

A Dijkstra graph is initialized with a start vertex distance equal to zero. It is the start and has not traveled yet, so a value of zero is appropriate. Every other vertex is thus far unprocessed and assigned a distance value of infinity. Two containers are created which are often referred to as the open and closed sets. The open set, S, represents the list of visited vertices and initially starts off empty. Q is the closed list and represents the min heap priority queue and initially contains all the vertices which at the start are unvisited. While Q is not empty, the node with the minimum distance, u, will be extracted from the Q and added to the set S. For each adjacent neighbor of u, the process of relaxation will be performed in order to determine the next shortest distance. Relaxation is defined by taking the distance sum of the previous vertex and adding it to the cost of the weighted edge between the previous and current vertices. If that value is lower than the current vertex’s distance, the new calculation replaces the current distance value at the vertex. This process is more specifically defined in the pseudocode below.

**Relaxtion**  🢧 **distanceSum[VD] = distanceSum[VB] + edgeWeight[VB,VD]**

if distanceSum calculated < currentSum[VD]

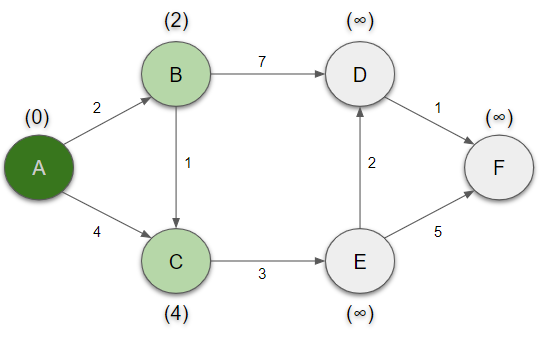


Figure 3: Dijkstra’s Algorithm proceeding through a weighted graph.

DIJKSTRA: Iterative Greedy SSSP Algorithm [19]

**distance[source] ←0**

**for all v ∈ V–{source}**

**do distance[v] ← ∞**

**S←∅** *(S, set of visited vertices is initially empty)*

**Q←V** *(Q, the queue initially contains all vertices)*

**while Q ≠ ∅**

**do u ← extractMin (Q, distance)** *(select node in Q with min distance)*

**S←S ∪ {u}** *(add u to list of visited vertices)*

**for all v ∈ neighbors[u]**

**do if distance[v] > distance[u] + cost (u, v)** *(if new shortest path found)*

**then distance[v] ←distance[u] + cost (u, v)** *(set new value of shortest path)*

**return distance**

* + 1. Run-Time Complexity: O(E log V)

Dijkstra has a few variations, but is widely accepted as bound by a time-complexity of **O(E log V)**. The most inefficient implementation is to employ two arrays as containers for the open and closed sets, resulting in a matrix representation of O(|V|2) time-complexity. Use of a min heap priority queue is considered the standard approach and is represented in the pseudocode above. At first glance of the nested loops, it might also appear that to be a O(|V|2) boundary, but this is not actually the case. A min heap is dominated by three functions, insert(), decreaseKey(), and extractMin(). The outer while loop consists of the extractMin() function for which all complexity variations are dependent. The insert() function resting outside of the inner loop, costs O(log V) time for each V. Yet, we observe that the inner loop is traversed V+E times in a similar manner as BFS. The decreaseKey() function is located inside the inner loop and serves to rebalance the heap at a cost of O(log V). Note that V-1 edges implies that V is within E. Thus, the run time is calculated as:

O(V+E) \* O(log V) 🢧 O((V+E)\*log V) 🢧 **O(E log V)**

The Fibonacci heap was designed in 1984 [8] specifically with the intent to add efficiency to Dijkstra’s algorithm. Though, it does accomplish this, it is only a slight gain with a worst case run time of **O(E+V logV)**

* + 1. Space-Complexity: O(V)

The space-complexity for Dijkstra depends to some extent on the structures implemented but is generally considered to be O(V). Utilizing a min heap priority queue containing two sets, all nodes generated are stored and thus, bound by the space constraints of O(V) + O(V) = O(2V) = **O(V)**.

* 1. A-star (A\*)

A\* is frequently utilized as the pathfinding solution in video games but is also used in situations like parsing using stochastic grammars as in NLP [15], new hierarchical algorithms for robot point-to-point path planning tasks, and new algorithms for power-aware routing of messages in large communication networks [9].

* + 1. The Algorithm Stepwise

Quite similarly to Dijkstra’s algorithm, A\* will initialize two sets, an open set of vertices unvisited and a closed set of vertices processed and essentially represent the path found. Initially, the open set contains the start node with an f value of 0. A\* will also keep a tally of the sum minimal distance, g(n), as calculated from the start to local node. However, unlike Dijkstra, A\* will expand the nodes and calculate the f(n) values which consist of the sum minimal distance thus far plus the heuristic weight appropriated to the edge between said node and its successor. Thus, f(n) = g(n) + h(n) is the generalized formula for expansion.

Once the sets are initialized and while the open set of unvisited nodes is not empty, the next minimum node will be removed from the open set and as long as it is not the target node, it will then be added to the closed set and successor (neighboring) nodes will be added to the open set and expanded. The expansion process will determine the f(n) and if that successor is either already in the closed set or if the temporary f(n) calculated is not better then the algorithm will do nothing and continue to the next successor. Otherwise, the node is added to the open set or updated with the new f(n) if it is already there.

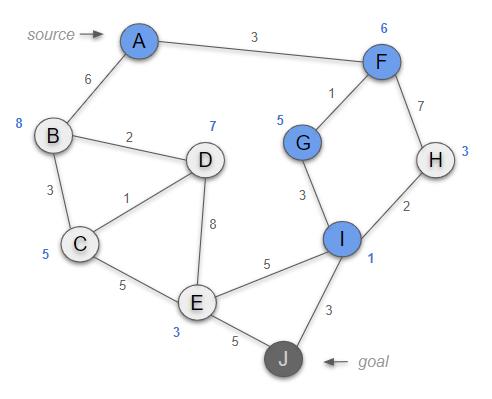


Figure 4: The result of Algorithm A\* on a graph gave Shortest Path = {A,F,G,I,J}.

A\* ALGORITHM: Dynamic Programming and Greedy SPSP [20]

**declare openSet as Priority Queue with Nodes**

**declare closedSet as Set with Nodes**

**function a-star ()**

**openSet.enqueue(startNode, 0)** *(initialize open set, closed set still empty)*

**while openSet.isNotEmpty()** *(repeat until shortest path found or no path exists)*

**currentNode = openSet.removeMin()**

**if currentNode == endNode**

**return PathFound**

**closedSet.add(currentNode)** *(currentNode fully evaluated, analysis done*

**expandNode(currentNode)** *(add all successors of currentNode to open set)*

**return NoPathFound** *(open set is empty, no path exists)*

**function expandNode (currentNode)**

**foreach successor of currentNode**

**if closedSet.contains(successor)** *(if already in closed set, continue to next successor)*

**continue**

**tentative\_g = g(currentNode) + costFromTo(currentNode, successor)**

**if openSet.contains(successor) and tentative\_g >= g(successor)**

**continue** *(if new path is not better, do nothing)*

**successor.predecessor = currentNode** (*set pointer of successor and set its g value)*

**g(successor) = tentative\_g**

**f = tentative\_g + h(successor)** *(add successor to open set with new f value*

*or update f value if already in set)*

**if openSet.contains(successor)**

**openSet.decreaseKey(successor, f)**

**else**

**openSet.enqueue(successor, f)**

* + 1. Heuristics: h(n) [2]

The topic of heuristics is a study in itself, but it’s worth highlighting some principles for the sake of grabbing their context in pathfinding algorithms such as Best-First Search or A\*. As mentioned previously, heuristics are a type of educated guess that may help provide shortcuts in time and can be calculated in several ways. Distances may, in fact, be exactly calculated and thus an exact heuristic may be implemented but this is costly and may severely reduce the efficiency of the algorithm, not to mention it somewhat defeats the point of the heuristics. Depending on the nature of the graph, directional capabilities, and the task being accomplished, heuristics might implement various means to calculate a distance. Three varieties are shown below best utilized in 4, 8 and infinitely directional grid maps, respectively.

* Manhattan Distance = D \* ( abs | node.x – goal.x | + abs | node.y – goal.y | )
* Diagonal Distance = D \* (dx + dy) + (D2 – 2\*D) \* min(dx, dy)
* Euclidian Distance = D \* sqrt( (node.x – goal.x)2 + (node.y – goal.y)2 )

Where D: cost of movement; D2: cost of moving diagonally

Methods such as these might be used to provide a realiable educated guess as to distances for nodes within a graph. If the heuristics are underestimasted of the exact values, they are considered admissible and thus, are guaranteed to find the shortest path distance. However, if the hueristics are over-estimated, there are no guarantees that the optimal shortest distance can be found. It’s actually possible to incorporate a variety of heuristics into one setting and thereby provide a trade-off between accuracy and speed as needed in order to maximize efficiency. That’s the beauty of hueristics.

* + 1. Run-Time Complexity: O(E) or O(bd)

Heuristics have the strongest impact on the A\* time complexity. In an unbounded graph search, the path solution, if one exists and is reachable, is defined exponentially by the depth, *d*, of the expanding vertices that terminate at the target node. The expansion is defined by the average number of successors at each state, or the branching factor, *b*. Thus, in a worst case scenario, A\* is most often described as a measure of this representative subset, O(bd). It may also be represented as O(E) representing the number of edges explored.

A good heuristic will offer a branching factor that can effectively prune off the excess bd nodes that an uninformed search, like Dijkstra, would otherwise expand. This quality lends speed and efficiency to A\*. A good heuristic will have a low branching factor which makes sense as the smaller the average number of successors branching reflects a narrow, more precise path toward the goal. Thus, b\* = 1 is the optimal branching factor

The time complexity is polynomial when the search state is a tree, there is only one target node, and the heuristics meet the condition |h(x) – h\*(x)| = O(log h\*(x)) where h\* is an optimal heuristic [5].

* + 1. Space-Complexity

The space complexity of A\* is O(|V|). One of the biggest drawbacks with A\* and is generally true of pathfinding algorithms, including Dijkstra, is the O(|V|) space-complexity because the algorithm stores into memory every node it generates. This limitation has promoted a great deal of interest in memory bounded heuristics, such as iterative deepening A\*, memory bounded A\*, and the like [5].

1. Variations/ Spin-Offs

Both Dijkstra and A\* are widely considered members of the top ten algorithms designed. Though these algorithms are considered some of the most efficient of the shortest path solutions, there are vast efforts to find further enhancements. Just as one of the major advantages to A\* is the pruning and reduction of nodes traversed, many of the techniques designed to make A\* more efficient focus on reducing the number of nodes. In some cases, this might be a bi-directional search, starting at source and end nodes working inward. It might mean a jump point search which skips further ahead to nodes visible from the current one. Beam search attempts to reduce the size of the open set, as it grows, by removing the nodes with the worst chance of offering a good path. Iterative deepening is often used in AI which starts with an approximate answer, looks ahead to see if values will be changing. If not then there is a good chance it won’t improve any further. This search sets a cutoff for the depth of f-values of which if too large, the node won’t even be considered. Dynamic A\* (D\*) is intended for use when you don’t have complete information. It compensates for inevitable errors without losing too much time. Lifelong Planning A\* (LPA) is used when the costs are changing and reuses previous A\* computations to produce a new path. Theta\* helps to incorporate path smoothing, straightening out a path, during the A\* process instead of after construction. Dynamic weighting whereby approximate weights start off larger and become smaller as they fine tune toward the goal [3]. It is evident there are a vast number of variations to these algorithms. However, most often the gain of speed is usually at the cost of space. It’s difficult to find advantage within both.

1. Impact

When considering the impact which the Dijkstra and A\* algorithms have made and what has promoted their status to that of the top ten most influential algorithms, it’s important to consider their broad influence on society through technology, telecommunications, transportation, medicine, sciences, economics, and even in within the field of entertainment. It would be difficult to find some aspect of our modern life that hasn’t somehow been influenced directly or indirectly by these algorithms and, in fact, most of modern technology is based upon these algorithms.

* 1. Technology & Telecommunications

The success of the modern internet and it’s ability to manage the monumental amount of traffic that daily transmits is largely due to early decisions to incorporate Dijkstra’s algorithm. The Distance-Vector Routing Protocol Algorithm (DVRPA) and the Link-State Routing Protocol Algorithm (LSRPA) are the two most essential algorithms we use every day as they efficiently route data traffic between the billions of connected networks that make up the Internet. The A\* algorithm is likewise used in wireless sensor networks (WSNs) because of its efficiency “in practical systems, where it can preprocess the graph to attain better performance.” [22]. A proposed change to this method is working in conjunction with a clustering algorithm and fuzzy inference, where results have theorized to show improvements over the current A\* algorithms in WSNs. Both algorithms are used extensively for signal routing through telecommunications and that alone is an enormous impact.

Yet, there is also a great deal of use of the A\* algorithm with artificial intelligence systems in robot point-to-point path planning tasks. Motion planning, moving the robot from one source to another and the ability to execute a task, is an essential goal within robotics. Robots need to move in the real world, be able to avoid collisions, and reach their location as fast as possible. Algorithms such as A\* are transforming that research and was originally developed specifically toward the study of robotics.

* 1. Transportation

The Dijkstra algorithm is the ‘routing pioneer’ upon which vehicle navigation systems and the GPS in cell phones is built. These algorithms have completely transformed the way we travel, affording timely and more energy efficient routes proving to be an asset to the environment as well as man. Yet, it’s useful for multiple needs and situations, not just for commuting to work or travelling to a new city. They are used in private as well as public transportation sectors, and in time-sensitive logistics like shipping packages or medicine across the country. An adaptation of Dijkstra’s algorithm was used in shortest route finding for the Spanish railway network, which gave estimates for costs and emissions based on already implemented track maps, as well as potential replacements with improved efficiency. Another impactful use of Dijkstra’s algorithm is “strategic level routing of hazardous material through a given route network” [21] in the context of military operations. These operations are done in planning and the algorithm is used given the assumption that there will be real time complications and the path will need to be changed on the fly. The interesting part of this example is that Dijkstra’s algorithm is used very dynamically and, on a time-space network, considering very large, implicitly defined graphs.

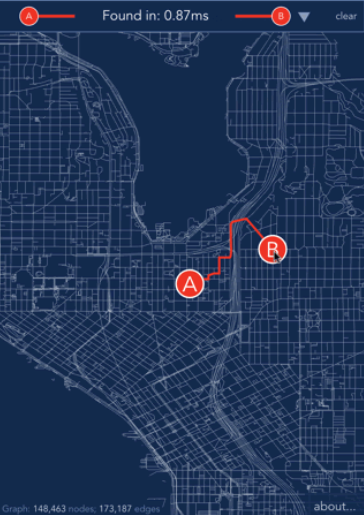


Figure 5: A real world example of how Algorithm A\* is used in transportation [4].

* 1. Economics

Dijkstra’s algorithm is involved in the Critical Path Method, which is used to reduce project length time in construction, networking, and many other fields that involve large scale projects. This plays right into the algorithm’s name, as the single source shortest path (SSSP) is used to find the optimal path in the planning stages so development time can be reduced in the long run. This is economically beneficial because it can help speed up processes in those fields mentioned earlier, and overall increase efficiency in the respective fields.

Dijkstra’s algorithm paved the way to A\* Algorithm, where A\* has attained better performance in graphing systems. Both algorithms have had countless uses in routing and problem-solving applications where they have been shown to improve efficiency and costs.

* 1. Medicine & Science

Dijkstra’s Algorithm has seen use in epidemiology as well to model the spread of infection disease. These models have a big impact on how medicine is handled and distributed and can help find the most affected areas so that resources can be allocated more precisely. Figure 6 is a model of a virus spreading, the vertices on the graph represent the individuals and the edges represent their possible contacts. This layout is useful to show how an individual manages to interact with others and how disease can spread on a person to person basis.

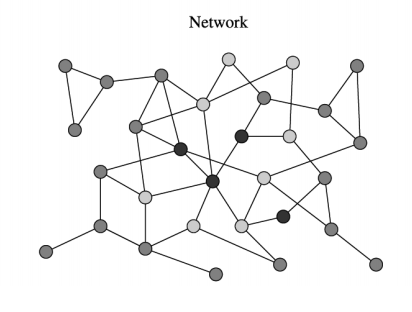


Figure 6: Model of Virus Spread [19].

* 1. Entertainment

One of the most common and frustrating artificial intelligence problems within the gaming industry is that of pathfinding. It has long been studied and various algorithms, including varieties of Dijkstra, Breadth-First Search, and Depth-First Search were implemented to fulfill this goal. The advent of A\* proved to be a game changer, literally. It has succesfully attracted the attention of researchers as the most optimal solution to the pathfinding problem. Currently, a great deal of exploration is being invested in A\* varieties and techniques which seek to further optimize this popular algorithm [25].

1. Conclusion

Since the 1950s mathematicians and researchers have been working to improve upon the question of shortest search paths. Algorithms such as Breadth-First Search, Depth-First Search, and Bellman-Ford were the precursors upon which Dijkstra and A\* were formulated. These algorithms have proven to be enormous impacts on society and the way we all live our lives. Many spin-offs and variations of these algorithms are being explored daily in an effort to find more efficiency, yet they are still considered the most trustworthy and efficient of their kind. They have without a doubt claimed their rightful title as one of the top ten influential algorithms in history.

REFERENCES

1. Abigail See. SAILORS Tutorial: Graph Search Algorithms (2017). Retrieved April 15, 2020 from https://cs.stanford.edu/people/abisee/tutorial/
2. Amit Patel. 1994. Heuristics. (1994). Retrieved April 15, 2020 from http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html
3. Amit Patel. 1994. Variants of A\*. (1994). Retrieved April 15, 2020 from <http://theory.stanford.edu/~amitp/GameProgramming/Variations.html>
4. Andrei Kashcha. 2019. A graph showing the algorithm A\* in action, GitHub, Inc.
5. Anon. 2020. A\* search algorithm. (April 2020). Retrieved April 15, 2020 from [https://en.wikipedia.org/wiki/A\*\_search\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm)
6. Anon. 2020. Breadth-first search. (March 2020). Retrieved April 16, 2020 from <https://en.wikipedia.org/wiki/Breadth-first_search>
7. Anon. 2020. Dijkstra's algorithm. (April 2020). Retrieved April 16, 2020 from https://en.wikipedia.org/wiki/Dijkstra's\_algorithm
8. Anon. 2020. Fibonacci heap. (March 2020). Retrieved April 15, 2020 from <https://en.wikipedia.org/wiki/Fibonacci_heap>
9. Antti Autere. 2005. Extensions and Applications of the A\* Algorithm. (2005). DOI:http://dx.doi.org/951-22-7947-9
10. Anupam Gupta. CMU Advanced Algorithms (2015). Retrieved April 15, 2020 from http://www.cs.cmu.edu/~anupamg/advalgos15/
11. Albert Aliu, Arbnor Halili, and Kujtim Gashi. 2013. Socio and economic impact of dijkstra algorithm implementation for mobile device platforms as a way to foster a green environment in urban area. (November 2013).
12. Alberto Almech, Eugenio Roanes-Lozano, Carmen Solano-Macías, and Antonio Hernando. 2019. A New Approach to Shortest Route Finding in a Railway Network with Two Track Gauges and Gauge Changeovers. *Mathematical Problems in Engineering* 2019 (May 2019), 1–16. DOI:<http://dx.doi.org/10.1155/2019/8146150>
13. Cisco. 2005. An Introduction to IGRP - Cisco, Cisco.
14. [Cormen, Thomas H.](https://en.wikipedia.org/wiki/Thomas_H._Cormen); [Leiserson, Charles E.](https://en.wikipedia.org/wiki/Charles_E._Leiserson); [Rivest, Ronald L.](https://en.wikipedia.org/wiki/Ronald_L._Rivest); [Stein, Clifford](https://en.wikipedia.org/wiki/Clifford_Stein) (2009). "Section 24.3: Dijkstra's algorithm". [*Introduction to Algorithms*](https://en.wikipedia.org/wiki/Introduction_to_Algorithms) *Third ed.*). [MIT Press](https://en.wikipedia.org/wiki/MIT_Press) and [McGraw–Hill](https://en.wikipedia.org/wiki/McGraw%E2%80%93Hill). pp. 658–663. [ISBN](https://en.wikipedia.org/wiki/International_Standard_Book_Number) 978-0-262-03384-8.
15. Dan Klein and Christopher D. Manning. 2003. A parsing. Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - NAACL 03 (2003). DOI:http://dx.doi.org/10.3115/1073445.1073461
16. E.C. Rosen, J.M. Mcquillan, J.G. Herman, and I. Richer. 1979. ARPANET Routing Algorithm Improvements. (January 1979). DOI:<http://dx.doi.org/10.21236/ada086340>
17. [Fredman, Michael Lawrence](https://en.wikipedia.org/wiki/Michael_Fredman); [Tarjan, Robert E.](https://en.wikipedia.org/wiki/Robert_Tarjan) (1987). "Fibonacci heaps and their uses in improved network optimization algorithms". *Journal of the Association for Computing Machinery*. 34 (3): 596–615. [doi](https://en.wikipedia.org/wiki/Digital_object_identifier):[10.1145/28869.28874](https://doi.org/10.1145%2F28869.28874)
18. Kenneth H. Rosen. 2019. *Discrete mathematics and its applications*, New York, NY: McGraw-Hill. pp. 744–749. [ISBN](https://en.wikipedia.org/wiki/International_Standard_Book_Number) 978-1-259-67651-2
19. Laksman Verravagu and Luis Barrera. PDF.
20. Maik Dreibholz. A\*-Algorithm. Retrieved April 15, 2020 from http://www.algostructure.com/search/a-star.php
21. N.Ravi Shankar and V. Sireesha. 2010. Using modified Dijkstra's algorithm for critical path method in a project network. *International Journal of Computational and Applied Mathematics* 5, 2 (March 2010).
22. saurabhsharma56. Routing Information Protocol (RIP), GeeksforGeeks.
23. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to algorithms* 3rd ed., Cambridge, England: Mit Press. pp. 643–645. [ISBN](https://en.wikipedia.org/wiki/International_Standard_Book_Number) 978-0-262-03384-8.
24. P. Chardaire, G.P. Mckeown, S.A. Verity-Harrison, and S.B. Richardson. 2005. Solving a Time-Space Network Formulation for the Convoy Movement Problem. *Operations Research* 53, 2 (2005), 219–230. DOI:<http://dx.doi.org/10.1287/opre.1040.0183>
25. Xiao Cui and Hao Shi. 2011. A\*-Based Pathfinding in Modern Computer Games. *IJCSNS International Journal of Computer Science and Network Security* 11, 1 (January 2011).
26. Yali Yuan, Caihong Li, Yi Yang, Xiangliang Zhang, and Lian Li. 2014. CAF: Cluster Algorithm and A-Star with Fuzzy Approach for Lifetime Enhancement in Wireless Sensor Networks. *Abstract and Applied Analysis* 2014 (2014), 1–17. DOI:http://dx.doi.org/10.1155/2014/936376
27. Zhan, F. Benjamin; Noon, Charles E. (February 1998). ["Shortest Path Algorithms: An Evaluation Using Real Road Networks"](https://semanticscholar.org/paper/c71301816cfe1e0c7ed1a04fddd7740ceb2e8197). [*Transportation Science*](https://en.wikipedia.org/wiki/Transportation_Science). 32 (1): 65–73. [doi](https://en.wikipedia.org/wiki/Digital_object_identifier):[10.1287/trsc.32.1.65](https://doi.org/10.1287%2Ftrsc.32.1.65)