**FRI-1.416.1-SSS-I-01**

# GRAPH SHORTEST PATH ALGORITHMS AND THEIR APPLICATIONS

**Hristo Hristov – Student**
Department of Informatics and Information Technologies
University of Ruse "Angel Kanchev"
Phone: +359 89 680 1366
E-mail: s236266@stud.uni-ruse.bg

**Assoc. Prof. Galina Atanasova, PhD – Supervisor**
Department of Informatics and Information Technologies
University of Ruse "Angel Kanchev"
Phone: +359 82 888 326
E-mail: gatanasova@uni-ruse.bg

***Abstract:*** *This paper provides a review of shortest-path algorithms across several categories: single source shortest path, all-pairs shortest path, and heuristic algorithms. We review both classical algorithms and recently developed approaches, analyzing and comparing them to one another to determine their practical applications and efficiency in terms of computational complexity and memory usage. This paper aims to provide insights into the trade-offs among these algorithms while summarizing recent developments in the field of graph theory.*

***Keywords:*** *Graph, Shortest Path Algorithms, Trade-Offs, Optimizations, Recent Developments.*

## INTRODUCTION

Graphs are a nonlinear data structure that can be used to model solutions to a large variety of problems. Their history dates back to 1736, when Leonhard Euler solved the "Seven Bridges of Königsberg" problem, which sought to determine whether it is possible to traverse all four areas of the city, connected by seven bridges, crossing each bridge exactly once [1]. Graphs can represent an almost unlimited number of sets, whose elements correspond to real-world objects, provided there is a relation between each pair of elements. This gives graphs exceptional flexibility and makes them an indispensable tool in the development of systems in many areas of human activity.

Graph pathfinding algorithms are a class of algorithms that, for a graph, where  is the set of vertices and  is the set of edges in the graph, search for a path that meets certain criteria. In this context, a path is a sequence of vertices  for which there exists an edge between each pair for. These algorithms have applications in logistics and transportation, computer networks, electronics, databases, video games, and other areas [2]. Advancements in algorithms and graph theory also pave the way for new scientific fields, such as graph Data Science, which has emerged from the effort to use specialized graph databases – structures optimized for the efficient storage and analysis of complex relationships between objects. The process includes graph EDA (Exploratory Data Analysis), where key network characteristics are assessed to select optimal algorithms [3].

This paper reviews and compares several categories of shortest path problems, presenting both classical algorithms, as well as their optimizations and/or modern approaches for solving them. The algorithms are compared based on their performance and memory usage, determined through repeated testing of the algorithms on graphs with small, medium, and large numbers of vertices (N = 100, 1000, 10000) with low density D ∈ [0.001, 0.1].

Since graph pathfinding algorithms are influenced not only by the number of vertices but also by the density of the graphs, their performance will also be evaluated on sparse (D ∈ [0.001,

 ***Copyrights© 2025 ISBN 978-954-712-978-8 (Print); e-ISBN 978-954-712-979-5***

0.1]), medium (D = 0.5), and dense (D = 0.9) graphs, for which N = 1000. The graphs used for comparison are generated automatically using the Erdős-Rényi model [4].

## EXPOSITION

One of the most frequently asked questions when dealing with graphs is how to find the shortest path, where finding a "shortest path" means finding the path between a given start and end vertex that has the minimum sum of edge weights.

### Single-Source Shortest Path Algorithms

For unweighted graphs, the solution is relatively trivial, using a breadth-first search (BFS), which traverses the vertices level by level and ensures that the target vertex is reached for the first time via the shortest path (i.e., with the minimum number of edges). If predecessors are stored in parallel for each visited vertex, the path itself can be reconstructed upon finishing the traversal. This algorithm has linear time complexity of O (E + V). The shortest path in a directed acyclic graph (DAG) can be found with the same time complexity using a topological sort followed by a single pass over the vertices. Interestingly, using the same method in a DAG also makes it possible to find the longest path in linear time, an NP-hard problem for other types of graphs. This is done by simply negating the weights of all edges (i.e., multiplying them by -1), computing the shortest path, and then negating the result. However, finding the shortest path in other types of weighted graphs requires slightly more sophisticated algorithms.

*When All Edge Weights Are Positive*

A commonly used algorithm for finding the shortest path when all edges in a graph have positive weights is Dijkstra's algorithm. It is a greedy algorithm that maintains a subset of vertices for which the shortest paths are already known. Initially, the distance to every vertex in the graph is set to, except for the starting vertex which is set to 0. At every iteration, the algorithm selects the vertex with the shortest distance so far, stores it in an array (or another structure) that contains the shortest distances to every vertex, thus marking it as visited, then for each neighbor of this vertex, a temporary shortest distance  is calculated, where d(u) and d(v) are the current shortest distances to the currently visited vertex and its neighbour, while w(u, v) is the weight of the edge connecting them.

The naive implementation of Dijkstra's algorithm has time complexity of, but it is almost always implemented with heaps, or the so-called priority queues – astructure that always provides the neighbor of the current vertex with the shortest distance without having to iterate over all of them. This speeds up the algorithm significantly to. Of course, if in a given case only the shortest path to a specific vertex is needed, and there is no need to subsequently search for shortest paths from the same starting vertex to others, then the algorithm can be terminated early upon finding the shortest path to the desired vertex.

The priority queue in Dijkstra's algorithm is typically implemented using a binary heap. In this structure, each extraction of the vertex with the minimum tentative distance, as well as each decrease-key operation, takes  on average. In Dijkstra's algorithm, usually there are predominantly decrease-key operations and fewer extractions of the minimum element in the heap. To optimize performance, the binary heap can be replaced with a d-ary heap (or d-heap), a type of heap that allows each node to have d children instead of two. Using a d-heap reduces the height of the tree, which speeds up decrease-key operations to, but in exchange for slower extractions of the minimum element, . Balancing these trade-offs carefully speeds up the algorithm even further, especially for dense graphs, where the number of edges is significantly larger than the number of vertices and setting d to a large enough value, such as E/V, leads to a smaller logarithmic value, and in practice, brings the time complexity of decrease-key operations close to linear.

Another distinct approach to optimizing the algorithm is the use of the so-called Radix Heap. Instead of a tree-based structure, it stores values in a sequence of "buckets" (or segments) with exponentially increasing boundaries. There are two essential conditions for its use: all elements must be non-negative integers, and each successive extraction of the minimum element must return a non-decreasing value. The second condition is always satisfied in Dijkstra's algorithm, since the tentative distances only increase or stay the same over time. The smallest bucket holds the elements with the smallest keys. When it becomes empty, the next non-empty bucket is located, and its elements are redistributed into the lower-indexed buckets based on updated boundary values, which are shifted forward accordingly. In this way, both the extraction of the minimum element and the relocation of elements to lower buckets take amortized constant time. This further speeds up the algorithm, but introduces an additional limitation — the edge weights must be integer values [5].

Table 1. Comparison between the three implementations of Dijkstra's algorithm

| | | Priority queue Dijkstra | | D-Heap Dijkstra | | | Radix Heap Dijkstra | | |
|---|---|---|---|---|---|---|---|---|---|
| N | D | Time | Memory | Time | Improvement | Memory | Time | Improvement | Memory |
| 100 | 0.1 | 0.00025106875 s | 128 Kb | 0.00019119 s | 31.31897589% | 128 Kb | 0.00021358125 s | 17.55186843 % | 128 Kb |
| 1000 | 0.1 | 0.01475176375 s | 1.188 Mb | 0.0132957875 s | 10.95065824 % | 1.188 Mb | 0.01345283875 s | 9.655397081 % | 1.2 Mb |
| 1000 | 0.5 | 0.06124786613 s | 5.568 Mb | 0.05758207638 s | 6.366199312 % | 5.388 Mb | 0.0549920125 s | 11.37593141 % | 5.536 Mb |
| 1000 | 0.9 | 0.1096190564 s | 9.996 Mb | 0.0998114875 s | 9.826092287 % | 10 Mb | 0.1002571988 s | 9.337840815 % | 9.996 Mb |
| 10000 | 0.1 | 1.226313003 s | 107.26 Mb | 1.222641 s | 0.3003336936 % | 106.676 Mb | 1.123332686 s | 9.167392529 % | 107.352 Mb |

*When Negative Edge Weights Are Present*

Although Dijkstra's algorithm may be extremely fast, it has one significant drawback – it cannot handle negative edge weights. Since the algorithm assumes that once a vertex is removed from the priority queue, the shortest path to it has definitely been found. This assumption holds only in graphs with non-negative edge weights, otherwise, a shorter path to that vertex might appear later on, leading the algorithm to produce incorrect results. In such cases, the Bellman-Ford algorithm is typically used, as it can handle negative edge weights, provided there are no negative-weight cycles in the graph. The algorithm works by repeatedly traversing all edges and relaxing each one in every iteration. Upon finishing the algorithm, the array of shortest distances contains the length of the shortest path from the source to each vertex, assuming that the graph contains no negative cycles. If, after the final relaxation, any values continue to decrease, this indicates the presence of negative-weight cycles. The time complexity of the algorithm is, which makes it slower than Dijkstra's algorithm.

A faster alternative to Bellman-Ford's algorithm is the Shortest Path Faster Algorithm (SPFA). It is an improvement over the classical Bellman-Ford algorithm that maintains a queue of actively changing vertices whose outgoing vertices could produce shorter paths. In other words, when an edge with weight is relaxed, the algorithm checks whether and if that is true, it sets and adds to the end of the queue if it hasn't previously been added. This way, the algorithm processes only the edges that have previously produced shorter paths. SPFA often executes significantly faster in practice, since it doesn't iterate over all possible vertices in every iteration, and practically executes in nearly linear time in many graphs, despite also having time complexity of in the worst case. The queue is usually implemented as an ordinary FIFO structure, although different variations such as the Small Label First or Large Label Last, implemented with double-ended queue, or deque, may further speed up the algorithm [8]. The practical tests have concluded that SPFA is typically significantly faster than Bellman-Ford's algorithm in graphs with only positive weights or predominantly positive weights, as well as sparse ones.

 ***Copyrights© 2025 ISBN 978-954-712-978-8 (Print); e-ISBN 978-954-712-979-5***

Table 2. Comparison between Bellman-Ford's algorithm and the two variants of SPFA on graphs with positive weights only

| | | Bellman-Ford | | SPFA | | | Deque SPFA | | |
|---|---|---|---|---|---|---|---|---|---|
| N | D | Time | Memory | Time | Improvement | Memory | Time | Improvement | Memory |
| 100 | 0.1 | 0.000588s | 128Kb | 0.000337s | 74.25760% | 128Kb | 0.000295s | 99.031887% | 128Kb |
| 1000 | 0.1 | 0.03771s | 1.6Mb | 0.019561s | 92.761779% | 1.2Mb | 0.02057s | 83.309172% | 1.2Mb |
| 1000 | 0.5 | 0.203055s | 6.2Mb | 0.09586s | 111.82575% | 5.4Mb | 0.090195s | 125.13052% | 5.4Mb |
| 1000 | 0.9 | 0.36683s | 12.3Mb | 0.16282s | 125.30648% | 10Mb | 0.15476s | 137.03006% | 10Mb |
| 10000 | 0.1 | 3.91288s | 196Mb | 1.9879167s | 96.83337% | 106Mb | 1.827618s | 114.09748% | 106Mb |

In graphs that contain numerous edges with negative weights, it performs worse due to the need to repeatedly re-enqueue and process more vertices, leading to overhead and degraded performance.

Table 3. Comparison between Bellman-Ford's algorithm and the two variants of SPFA on graphs with both positive and negative weights

| | | Bellman-Ford | | SPFA | | | Deque SPFA | | |
|---|---|---|---|---|---|---|---|---|---|
| N | D | Time | Memory | Time | Improvement | Memory | Time | Improvement | Memory |
| 100 | 0.1 | 0.000589s | 128Kb | 0.000303s | 94.377868% | 128Kb | 0.000297s | 98.359331% | 128Kb |
| 1000 | 0.1 | 0.038s | 1.6Mb | 0.05098s | -25.462347% | 1.2Mb | 0.039262s | -3.2142751% | 1.2Mb |
| 1000 | 0.5 | 0.21317s | 6.2Mb | 0.4884s | -56.344352% | 5.4Mb | 0.501168s | -57.4645% | 5.4Mb |
| 1000 | 0.9 | 0.38985s | 12Mb | 0.78137s | -50.10768% | 10Mb | 0.755149s | -48.374992% | 10Mb |
| 10000 | 0.1 | 4.009339s | 198Mb | 21.717s | -81.53823% | 106Mb | 11.6487s | -65.581262% | 106Mb |

The development of shortest path algorithms for graphs with negative weights and cycles is an active area of research. Generally, it is not possible to reliably improve the time complexity boundary set by the Bellman-Ford algorithm, since detecting every possible cycle and checking for negativity, or proving that no shorter hidden path exists, significantly slows down the computation. Hence why, many of these algorithms seek to improve performance via techniques that involve randomization. The randomization helps overcome the problem of checking every possible path by randomly shuffling the order of relaxations or by introducing small random noise, which yields a theoretical near-linear runtime complexity. This paper employs one such Las Vegas algorithm[7]. It uses a recursive scaling method called ScaleDown, which progressively adjusts the edge weights to make them non-negative through elementary graph decompositions. At each step, the graph is partitioned into strongly connected components with low diameters, and paths are corrected using elementary weight adjustment functions. By gradually reducing the number of negative-weight edges, the algorithm ensures that only a small portion of the graph requires heavy processing, thereby achieving, in theory, an expected near-linear runtime of , where W is a lower bound for the weights of the edges in the graph [6]. However, the practical tests concluded that this and many other randomized algorithms hide an enormous overhead caused by the randomization, complex data structures used, and the sophisticated methods utilized in them behind their theoretical time complexity. After thorough testing, it was concluded that, in comparison to this particular algorithm, the Bellman-Ford algorithm is several orders of magnitude faster in practice.

[7] Las Vegas algorithm – A randomized algorithm that guarantees a correct result, in contrast to Monte Carlo algorithms, which have a small probability of producing an incorrect one [9].

Table 4. Comparison between Bellman-Ford's algorithm and the Las Vegas algorithm

| | Bellman-Ford | | Randomized algorithm | |
|---|---|---|---|---|
| N | Time | Memory | Time | Memory |
| 1000 | 0.04201835314 s | 212 Kb | 6.32982132038 s | 12.6 Mb |
| 5000 | 0.98801542751 s | 1.2 Mb | 105.33995120498 s | 65.3 Mb |
| 10000 | 4.372559002 s | 2.3 Mb | 235.60215523319 s | 131.2 Mb |

**All-Pairs Shortest Path Algorithms**

A well-known algorithm used to solve the all-pairs shortest path problem is the Floyd-Warshall algorithm. It handles negative weights but not negative cycles, similarly to the Bellman-Ford algorithm. Nevertheless, unlike Bellman-Ford's algorithm it does not relax edges weights, instead it uses a dynamic-programming approach to find shortest paths. Every pair of vertices U and V in the graph is stored in a distance matrix d[U][V], which is initially initialized to . At every iteration, the algorithm improves the distances using the recurrence relation. It is often implemented using three nested loops, for U, V and K. Thus, its time complexity is $O(V^3)$ [10].

Another algorithm commonly used for this task is Johnson's algorithm. It combines Bellman-Ford and Dijkstra's algorithms to efficiently compute shortest paths between all pairs of vertices in a graph with negative edge weights (but no negative-weight cycles). In the first step, a new artificial vertex S is added and connected to every other edge in the graph with edges of zero weight. Then, Bellman-Ford's algorithm is run from S to calculate a potential for every vertex V, such that for every edge a reweighted cost that satisfies is calculated. Using these potentials, it defines a new weighted graph with non-negative weights. Since the shortest paths in the reweighted graph correspond to the shortest paths in the original graph, Dijkstra's algorithm is then run from each vertex. Finally, to recover the actual distance from U to V, the algorithm subtracts. As a result, the shortest paths between every two pairs of vertices in the graph are found in time [7].

Table 5. Comparison between Floyd-Warshall's algorithm and Johnson's method

| | | Floyd-Warshall | | Johnson's method | | |
|---|---|---|---|---|---|---|
| N | D | Time | Memory | Time | Improvement | Memory |
| 100 | 0.1 | 0.0024232325 s | 256 Kb | 0.004654355 s | -47.93623391% | 256 Kb |
| 1000 | 0.1 | 3.613280634 s | 7.852 Mb | 1.372114772 s | 163.3366179 % | 10.1 Mb |
| 1000 | 0.5 | 5.449615518 s | 7.852 Mb | 3.189079043 s | 70.8836766 % | 17.125 Mb |
| 1000 | 0.9 | 5.989906183 s | 7.853 Mb | 4.670088849 s | 28.26107548 % | 26.076 Mb |
| 10000 | 0.001 | 337.219039 s | 781.840 Mb | 7.445193177 s | 4429.352443 % | 790.964 Mb |

In small and dense graphs, the Floyd-Warshall algorithm tends to come close or even outperform it, despite being slower in theory, due to the cache-efficient nature of its three nested loops and simple data structures, in contrast to the heaps used in Johnson's method.

**Heuristic Algorithms**

Heuristic algorithms are pathfinding methods that rely on a heuristic function to estimate the presumed cost of reaching the goal from a given vertex, thereby guiding the search process. One of the most well-known examples is the A* (A-star) algorithm. In each iteration, it selects the vertex with the lowest value of f(n) = g(n) + h(n), where g(n) represents the accumulated cost from the start to vertex n, and h(n) is the heuristic estimate from n to the goal. The f-values of the

 *Copyrights© 2025 ISBN 978-954-712-978-8 (Print); e-ISBN 978-954-712-979-5*

neighboring vertices are then updated, and the vertex with the next lowest f-value is selected. A critical factor in the successful application of such algorithms is the proper choice of the heuristic function, which must not overestimate the actual cost and should return consistent results. Although heuristic algorithms may, in some cases, sacrifice completeness or optimality in favor of speed, they make many otherwise intractable or slow-to-solve problems practically solvable. This makes them widely applicable in fields such as robotics, video games, and other domains where fast computation of a reasonably optimal solution is essential. [11].

## CONCLUSION

Shortest path algorithms remain an active and important area of research, with a wide range of applications. This work has reviewed fundamental approaches to solving such problems, highlighting their advantages and limitations. The article does not claim to provide an exhaustive overview of all existing shortest path algorithms. Additional approaches may be included in future publications.

## REFERENCES

Taheri-Dehkordi, M. (2024). *Graph Theory; History, Applications and Vision.* Journal of Mathematics and Society. ResearchGate

Roughgarden, T. (2018). *Algorithms Illuminated: Graph algorithms and data structures. Part 2*. Soundlikeyourself Publishing LCC.

Robinson, S. (2023, February 17). *What is Graph Data Science? A complete introduction to critical new ways of analyzing your data.* Graphable AI. https://www.graphable.ai/blog/graph-data-science/

Erdos, P., Rényi, A. (1960). *On the evolution of random graphs.* Publications of the Mathematical Institute of the Hungarian Academy of Sciences, 5(1), 17-60.

Ahuja, R., Mehlhorn, K., Orlin, J. and Tarjan, R. 1990. *Faster algorithms for the shortest path problem.* J. ACM 37, 2 (April 1990), 213–223. https://doi.org/10.1145/77600.77615

Bernstein, A., Nanongkai, D., Wulff-Nilsen, C. (2023). *Negative-weight single-source shortest paths in near-linear time. arXiv preprint arXiv:2203.03456*. https://arxiv.org/abs/2203.03456

Johnson, D. (1977). *Efficient Algorithms for Shortest Paths in Sparse Networks*. J. ACM 24, 1 (Jan. 1977), 1–13. https://doi.org/10.1145/321992.321993

Zou, X. (2013). *An Improved SPFA Algorithm for Single-Source Shortest Path Problem Using Forward Star Data Structure.*

*Las Vegas algorithm*. (n.d.). Wikipedia. Retrieved May 1, 2025, from https://en.wikipedia.org/wiki/Las_Vegas_algorithm

Nakov, P., Dobrinkov, P. (2015). *Programming=++Algorithms; (5th edition)*. Software University

Rubio, F. (2023, June 6). *Pathfinding Algorithms- Top 5 Most Powerful*. Graphable. Retrieved May 1, 2025, from https://www.graphable.ai/blog/pathfinding-algorithms/