**Shortest Path Algorithm - Based on Reverse tracking from destination, using Dynamic Programming**

Srihari M Dr. Bhagyashree Ambore Dr. Suresh L

Dept. of ISE Associate Professor Head of the Department

RNSIT Dept. of ISE, RNSIT Dept. of ISE, RNSIT

[sriharisai6230@gmail.com](mailto:sriharisai6230@gmail.com) [ambore.bhagyashree@gmail.com](mailto:ambore.bhagyashree@gmail.com) [hod.ise@rnsit.ac.in](mailto:hod.ise@rnsit.ac.in)

Dr. Sunitha K

Assistant Professor

Dept. of ISE, RNSIT

[sunithakrisnamurthy@gmail.com](mailto:sunithakrisnamurthy@gmail.com)

**Abstract:**

This research is conducted for acquiring out a new algorithm for finding out the classic “Shortest Path” between every pair of vertices using an atypical approach. The approach includes the principles of Dynamic Programming, and is slightly different from the traditional “Dijkstra’s and Floyd-Warshall” algorithms, wherein, we start from the “destination” node and backtrack to all the previous vertices.

The core of the algorithm involves initializing data structures to track distances and paths, and then iteratively updating these values based on the cost matrix of the graph. By iteratively selecting the node with the minimum distance and considering its neighbours, the algorithm systematically constructs the shortest path. The result is a versatile and efficient tool for solving various shortest path problems.

The algorithm's adaptability and effectiveness make it a valuable addition to the toolkit of graph-related tasks, offering a unique perspective on finding the shortest path while accommodating different graph structures and edge weights. Its flexibility and ease of implementation make it a useful tool for a wide range of applications in fields such as network routing, transportation, and logistics.

***Key Terms: Graph Traversal, NumPy, GPU, Dijkstra’s Algorithm, Floyd-Warshall’s Algorithm***

**Introduction:**

The problem of finding the shortest path between two nodes in a graph is a fundamental challenge in computer science and network analysis, with wide-ranging applications in various domains, from transportation to computer networks and beyond. Traditional approaches, such as Dijkstra's and Floyd-Warshall's algorithms, typically rely on forward traversal from the source to the destination node. In this study, we introduce a novel and innovative algorithm, the **Reverse Tracking Shortest Path Algorithm**, which takes a distinct approach by working in reverse, starting from the destination node and backtracking to the source.

This algorithm stands out as a departure from conventional methods, leveraging dynamic programming principles to provide a unique perspective on the shortest path problem. Unlike many existing techniques, our algorithm is not limited to specific graph types; it seamlessly handles unidirectional, bidirectional, multistage, and even graphs with negative weighted edges, making it a versatile tool for researchers and practitioners across diverse fields.

At its core, the Reverse Tracking Shortest Path Algorithm is designed to efficiently compute the shortest path by initializing and updating data structures that keep track of distances and paths. By iteratively selecting nodes with the minimum distance and exploring their neighbours, the algorithm systematically constructs the shortest path from the destination node to the source.

One of the algorithm's distinguishing features is its ability to adapt to a multitude of real-world scenarios. Its flexibility and ease of implementation render it invaluable in applications such as network routing, logistics optimization, and transportation planning. Whether it's finding the quickest route for goods delivery or optimizing data packet routing in a network, this algorithm offers a fresh and efficient perspective on solving the shortest path problem.

In the sections that follow, we delve into the intricacies of the Reverse Tracking Shortest Path Algorithm. We examine its theoretical foundations, provide insights into its computational efficiency, and showcase its practical applications. By introducing this algorithm, we aim to broaden the horizons of shortest path problem-solving, offering a new tool for researchers and practitioners seeking innovative solutions in the world of graph theory and network analysis.

**Core of the Reverse-based Algorithm:**

The Reverse-based algorithm we deploy here is based on the fact that we know the paths to all the nodes which are present in the graph.

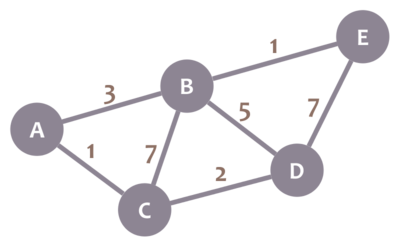
We deploy the algorithm such that, we mark the destination node as “visited”, and iteratively backtrack to the previous vertices.

On traversal, it finds out the paths from the destination vertex to all the other vertices, and uses this memory to find out paths to other nodes.

The exploring of neighbours is done similar to the “Breadth-First Search” method of graph-traversal. The algorithm finds out minimum of every other node level by level and employs the data for recursive traversal.

**Related works for Underlying principles:**

[1] Dijkstra’s Algorithm: Dijkstra's algorithm (named after it’s discover, E.W. Dijkstra) solves the problem of finding the shortest path from a point in a graph (the source) to a destination. It turns out that one can find the shortest paths from a given source to all points in a graph in the same time, hence this problem is sometimes called the single-source shortest paths problem.



*A completely connected weighted graph*

[2] Dynamic Programming: Dynamic programming is a method for solving a complex problem by breaking down the given problem into a number of sub problems and solving these sub problems once and storing the solution to these sub problems in a table. Generally, dynamic programming is applied to optimization problems. Dynamic programming is applied when there is an overlapping between sub problems of the same problem.

**The Proposed Algorithm:**

In the initial steps of the Reverse Tracking Shortest Path Algorithm, several critical data structures are established to facilitate the efficient calculation of the shortest path between nodes in a graph.

These data structures include “num\_nodes”, which represents the total number of nodes in the given cost matrix, and visited, a boolean array of size “num\_nodes” initialized with all elements set to False. Additionally, “shortest\_distance” is initialized as an array of size “num\_nodes”, with each element set to an initially high value. To keep track of the path itself, “shortest\_path” is initialized as an array of empty lists, also of size “num\_nodes”.

Furthermore, the algorithm sets “shortest\_distance[destination]” to 0 and “shortest\_path[destination]” to a list containing just the destination node. These preparatory steps lay the foundation for the subsequent computations in the algorithm.

The heart of the **Reverse Tracking Shortest Path Algorithm** unfolds in a systematic iteration process. Beginning with a loop that ranges from “1 to num\_nodes – 1”, the algorithm continuously seeks the minimum distance (min\_distance) and the corresponding node (current\_node) that have not yet been visited. This is done by examining each node in the range of “0 to num\_nodes – 1”. The criteria for selecting the next node include its unvisited status (visited[node] is False) and having a shorter distance than the “current min\_distance”. Once the loop identifies a suitable “current\_node”, it is marked as visited (visited[current\_node] = True). The algorithm then proceeds to examine the edges from the newly visited node to its predecessors (prev\_node). Importantly, the part of the algorithm that deals with negative graphs is outlined, and the computation of the new distance (new\_distance) between nodes is described. If the newly calculated distance is shorter than the previously recorded “shortest\_distance[prev\_node]”, the algorithm updates both “shortest\_distance[prev\_node]” and “shortest\_path[prev\_node]”. This process continues until all nodes are visited, and the shortest path and its cost are determined.

**Data Structures and Processing:**

The Reverse-based algorithm deployed, utilises “NumPy” type of array data structures. Typically, while working on a smaller cost-matrix (The matrix which contains all the respective weights of the graphs), we might tend to utilise the typical “List” data-structure in Python. But, owing to the current trends in technology and the need for the computation of vastly available raw data to be processed in a very efficient manner, we utilise the “NumPy” module provided for us.

[3] NumPy Arrays: A NumPy array is a multidimensional, uniform collection of elements. An array is characterized by the type of elements it contains and by its shape. For example, a matrix may be represented as an array of shape (M × N) that contains numbers, e.g., floating point or complex numbers. Unlike matrices, NumPy arrays can have any dimensionality.

We prefer NumPy arrays owing to their vast utilization of vectorized computation and parallel processing which includes not only the CPU, but the GPU as well.

We also have modified the code for utilizing all the threads of the processor, by setting a system variable named “OMP\_NUM\_THREADS” to the number of threads available in the system. This data is collected by the “os” module present in the Python Standard library.

**The Correction for Negative Weighted graphs:**

First, we need to understand what a “Negative weighted graph” means. It is nothing but a graph which consists of negative weights. Here, we need to be very careful as to what these “negative weights” refer to. In most of the user scenarios, they typically refer to one of the two things stated below:

* An even shorter path which provides us with “lesser cost”.
* A path which indicates “loss”

The algorithm can be manipulated for the first user scenario, wherein we can remove the condition where we check if the “cost between the current and the previous node” is > 0, i.e., a path exists between those two nodes. But this condition does not make sense when we have negative weighted graphs.

**Test Cases:**

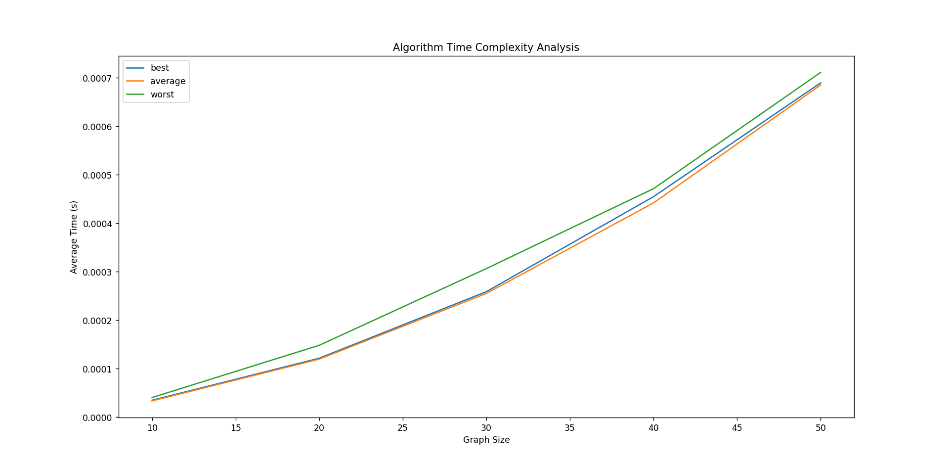
The algorithm was run for the following types of graphs.

* Simple graph with positive edge weights.
* Graph with unreachable destination.
* Negative edge weighted graph.
* Empty graph.
* Single-node graph.
* Negative cycled graph.
* Disconnected graph.
* Large graph.
* Graph with Loops.

All the test cases were verified with the above given algorithm and thoroughly tested in an extensive manner.

**Results - Inclusive of Time Complexities:**

This algorithm, unlike Dijkstra’s Algorithm and Floyd’s algorithm, provides a 𝑉² time complexity, where V is the number of vertices. The time complexity seems linear for the first set of cost matrices, we found out up-to 50x50 this holds true, and the curve steadily increases.



*Time complexity graph of the algorithm*

The algorithm utilises 2 loops, each not nested to one another. Owing to the NumPy utilisations, the complexity reduces real-time.

This algorithm was run for a set of 90000 iterations, wherein each iteration, we run for n set of sizes, the sets containing 0-n, in multiples of 5 for simplicity, and for 3 cases namely “best”, “worst” and “average” for labelling purposes.

**The need for a newer solution (Comparing our solution with already present algorithms) :**

The need for this newer solution for a problem, which already has so many algorithms for solving in various time complexities, is a very intriguing question which might arise within. The solution which we provision, starts from the destination node, and figures out all the possible vertices from the given node. Since Dijkstra’s algorithm is greedy in nature, we do not utilise the previously figured out solutions and start over each time for computing the smallest distance. The newly implemented algorithm utilises this saved memory to speed up computation time. Although we might get the same time complexity, we still have the memory of the paths which eases computation, thus making the problem “Dynamically” solved.

One of the main points to be acknowledged here is the “Ease of Implementation”. We solve the problem using basic iterations and apply basic arithmetic and relational operations to figure out the shortest distances. This might prove to be very helpful to anyone who needs to find out shortest paths just by using basic operations and not recursions.

Here are some other points which suggests the various advantages of this algorithm over traditional path finding algorithms.

* Handling Negative Edge Weights: Unlike Dijkstra's algorithm, which doesn't work with negative edge weights, the Reverse Tracking Algorithm can be adapted to handle graphs with negative weights. It removes the condition that checks if the cost between the current and previous nodes is greater than zero, making it suitable for scenarios where negative weights represent shorter paths or losses.
* Versatility with Graph Types: This algorithm is highly versatile and can handle various types of graphs, including unidirectional, bidirectional, multistage, and those with complex structures. Traditional algorithms like Dijkstra's have limitations based on graph type, whereas this algorithm seamlessly adapts to different scenarios.
* Memory Efficiency: The algorithm efficiently utilizes memory by using NumPy arrays and optimized data structures. It offers faster access to data due to contiguous memory allocation and the utilization of vectorized computation. This memory efficiency is especially crucial when dealing with large graphs.
* Ease of Implementation: The Reverse Tracking Algorithm is relatively easy to implement compared to some traditional algorithms that involve complex data structures and intricate logic. Its reliance on basic arithmetic and relational operations makes it accessible to a wider range of users.
* Parallel Processing: The algorithm's use of NumPy allows for parallel processing, taking advantage of modern hardware, including multi-core processors and GPUs. This can significantly speed up the computation of shortest paths, especially for large graphs.
* Adaptability to Real-World Scenarios: Its ability to adapt to real-world scenarios, such as network routing, logistics optimization, and transportation planning, makes it a valuable tool. It can handle complex situations where traditional algorithms might fall short.
* Unique Perspective: The Reverse Tracking Algorithm provides a unique perspective on solving the shortest path problem. Starting from the destination and working backward offers a fresh approach that can lead to innovative solutions in various domains.
* Efficiency for Certain Cases: In some scenarios, the Reverse Tracking Algorithm may exhibit better performance in terms of time complexity, especially for smaller cost matrices. For specific problem instances, it can provide efficient solutions.

**Memory utilization:**

We utilise NumPy array structure here, which is implemented from “C data structure” [3], which consists of 2 components: the data buffer (the raw data) and the information on the pointers to the raw data. The NumPy arrays are stored in contiguous memory blocks, which helps in faster access to data. The vectorization implementations also speed up the computation process. Owing to contiguous memory allocation, it is well defined that the memory blocks are stored in “chunks”, i.e., the whole data is stored in a single block inside the memory. We utilise associated deallocation functions (pre-implemented in the NumPy modules) to free up memory as and when needed.

**Conclusion:**

In conclusion, the Reverse Tracking Shortest Path Algorithm presents a novel and innovative approach to solving the classic "Shortest Path" problem in graph theory. Unlike traditional algorithms like Dijkstra's and Floyd-Warshall, this algorithm starts from the destination node and backtracks to the source, leveraging the principles of Dynamic Programming.

This algorithm's key strengths lie in its versatility and efficiency. It can handle various types of graphs, including unidirectional, bidirectional, multistage, and those with negative weighted edges. The algorithm's adaptability makes it a valuable tool in diverse fields, such as network routing, transportation planning, logistics optimization, and beyond.

The use of NumPy arrays and optimized memory allocation ensures efficient computation, even for large datasets. Additionally, the algorithm's ease of implementation, relying on basic arithmetic and relational operations rather than complex recursions, makes it accessible to a wide range of users.

Overall, the Reverse Tracking Shortest Path Algorithm offers a fresh perspective on solving the shortest path problem, opening up new possibilities for researchers and practitioners in the realms of graph theory and network analysis. Its ability to find optimal routes in various scenarios and handle complex graph structures makes it a valuable addition to the toolkit of shortest path problem-solving techniques.

**References:**

[1] Javaid, Adeel. (2013). Understanding Dijkstra Algorithm. SSRN Electronic Journal. 10.2139/ssrn.2340905.

[2] Neissery Abdulla, Sajil. (2018). Dynamic programming.

[3] van der Walt, Stéfan & Colbert, S. & Varoquaux, Gael. (2011). The NumPy Array: A Structure for Efficient Numerical Computation. Computing in Science & Engineering. 13. 22 - 30. 10.1109/MCSE.2011.37.

Wang, Kai & Fussell, Don & Lin, Calvin. (2021). A fast work-efficient SSSP algorithm for GPUs. 133-146. 10.1145/3437801.3441605

Ramadiani, Ramadiani & Bukhori, D & Azainil, Azainil & Dengen, N. (2018). Floyd-warshall algorithm to determine the shortest path based on android. IOP Conference Series: Earth and Environmental Science. 144. 012019. 10.1088/1755-1315/144/1/012019.