International Journal of Scientific Research and Engineering Development-– Volume 9 Issue 1, Jan-Feb 2026
Available at www.ijsred.com

RESEARCH ARTICLE OPEN ACCESS

A Study About Graph Algorithms and Applications of Graph Theory in Real Life

Deepa T[1], Anuradha A[2], Rathi K[3] and Padmavathi R[4]
[1, 2, 3, 4]Assistant Professors, PG & Research Department of Mathematics,
Sri Ramakrishna College of Arts & Science, Nava India, Coimbatore - 641 006, Tamil Nadu, India
Email: deepa.t@srcas.ac.in, padmavathi@srcas.ac.in, anuradha@srcas.ac.in

Abstract

Graph algorithms form a cornerstone of computer science, enabling efficient solutions to problems modeled through networks of vertices and edges. These algorithms address fundamental tasks such as shortest path computation, maximum flow determination, graph traversal, and connectivity analysis. Their theoretical foundations provide insights into complexity, optimization, and data structures, while their practical relevance spans diverse domains.

Applications of graph algorithms are pervasive: in communication networks for routing and bandwidth optimization, in social network analysis for community detection and influence modeling, in bioinformatics for protein interaction mapping, and in transportation systems for traffic flow and route planning. Emerging areas such as machine learning, recommendation systems, and cybersecurity increasingly leverage graph-based methods to capture relational data and uncover hidden patterns. Thus, graph algorithms not only advance computational theory but also serve as indispensable tools in solving real-world problems across science, engineering, and society.

1.Introduction

Graph theory relies on a wide range of algorithms that help solve problems like shortest paths, network flows, connectivity, and optimization. The most widely used algorithms include BFS, DFS, Dijkstra's, Bellman-Ford, Floyd-Warshall, Prim's, Kruskal's, and Ford-Fulkerson.

2.Graph Algorithms

2.1 Traversal Algorithms

A graph is a data structure made up of nodes (also called vertices) and edges that connect pairs of nodes. Graphs can be directed or undirected, weighted or unweighted, and they can represent a wide variety of relationships, from friendships in a social network to links between web pages or routes on a map. This flexibility makes graphs a powerful way to explore connections within your data.

Graph traversal is the process of visiting nodes in a graph by following the edges that connect them. But simply moving from node to node without a clear strategy can quickly become inefficient, especially in large or densely connected networks.

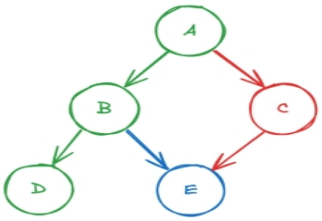
Depth-First Search (DFS)

DFS explores a graph by going as far as possible along one path before backtracking. From the starting node, it visits an unvisited neighbor, following unvisited neighbors down that path until it reaches a node with no more unexplored connections. At that point, it backtracks to the previous node and repeats the process.

DFS evaluates the cost solely based on the node's depth in the search tree, and uses the following evaluation function:

$$g(n) = depth(n), h(n) = 0$$

By choosing the path with the highest estimated cost through n, we prioritize expanding the **deepest unvisited nodes**. This behavior allows us to use a **stack** to keep track of the path, where we expand the most recently added node. This is either done explicitly with a data structure, or implicitly through the call stack in a recursive implementation. This leads to two variations of DFS: Iterative DFS and recursive DFS.



DFS traversal starting from node A. Nodes are visited in the order A → B → D → E → C.

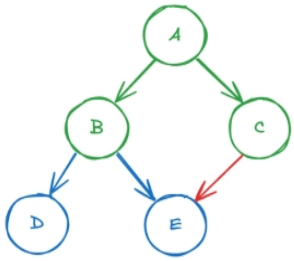
Breadth-First Search (BFS): Explores level by level using a queue.

BFS explores a graph by visiting all nodes at the current depth before moving deeper. Starting from the initial node, it visits all unvisited neighbors, then processes each of those neighbors' unvisited neighbors in turn, gradually expanding outward from the source.

BFS uses the same evaluation function as DFS:

$$g(n) = depth(n), h(n) = 0$$

Unlike DFS, which follows a single path as far as it can before backtracking, BFS systematically explores all nearby nodes before moving further out. It chooses the path with the lowest estimated cost through n, prioritizing the expansion of the nearest unvisited nodes. To manage this, BFS uses a queue to track nodes to visit next, always processing the earliest added node first. BFS is typically implemented iteratively, since it relies on a queue for **FIFO** (First In, First Out) ordering, whereas recursion uses a stack, which follows **LIFO** (Last In, First Out) behavior.



BFS traversal starting from node A. Nodes are visited in the order A → B → C → D → E.

2.2 Shortest Path Algorithms

The shortest path algorithms are the ones that focuses on calculating the minimum travelling cost from **source node** to **destination node** of a graph in optimal time and space complexities.

Dijkstra's Algorithm: Finds shortest paths from a source to all vertices in weighted graphs with non-negative edges.

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a weighted graph, which may represent, for example, a road network.

Dijkstra's algorithm finds the shortest path from a given source node to every other node. It can be used to find the shortest path to a specific destination node, by terminating the algorithm after determining the shortest path to that node. For example, if the nodes of the graph represent cities, and the costs of edges represent the distances between pairs of cities connected by a direct road, then Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. A common application of shortest path algorithms is network routing protocols, most notably IS-IS (Intermediate System to Intermediate System) and OSPF (Open Shortest Path First). It is also employed as a subroutine in algorithms such as Johnson's algorithm.

The algorithm requires a starting node, and computes the shortest distance from that starting node to each other node. Dijkstra's algorithm starts with infinite distances and tries to improve them step by step:

1. Create a set of all unvisited nodes: the unvisited set.
2. Assign to every node a distance from start value: for the starting node, it is zero, and for all other nodes, it is infinity, since initially no path is known to these nodes. During execution, the distance of a node *N* is the length of the shortest path discovered so far between the starting node and *N*.
3. From the unvisited set, select the current node to be the one with the smallest (finite) distance; initially, this is the starting node (distance zero). If the unvisited set is empty, or contains only nodes with infinite distance (which are unreachable), then the algorithm terminates by skipping to step 6. If the only concern is the path to a target node, the algorithm terminates once the current node is the target node. Otherwise, the algorithm continues.
4. For the current node, consider all of its unvisited neighbors and update their distances through the current node; compare the newly calculated distance to the one currently assigned to the neighbor and assign the smaller one to it. For example, if the current node *A* is marked with a distance of 6, and the edge connecting it with its neighbor *B* has length 2, then the distance to *B* through *A* is 6 + 2 = 8. If B was previously marked with a distance greater than 8, then update it to 8 (the path to B through A is shorter). Otherwise, keep its current distance (the path to B through A is not the shortest).
5. After considering all of the current node's unvisited neighbors, the current node is removed from the unvisited set. Thus a visited node is never rechecked, which is correct because the distance recorded on the current node is minimal (as ensured in step 3), and thus final. Repeat from step 3.
6. Once the loop exits (steps 3–5), every visited node contains its shortest distance from the starting node.

Bellman-Ford Algorithm: Handles graphs with negative weights; detects negative cycles.

The **Bellman–Ford algorithm** is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph. It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers.[

Like Dijkstra's algorithm, Bellman–Ford proceeds by relaxation, in which approximations to the correct distance are replaced by better ones until they eventually reach the solution. In both algorithms, the approximate distance to each vertex is always an overestimate of the true distance, and is replaced by the minimum of its old value and the length of a newly found path.

However, Dijkstra's algorithm uses a priority queue to greedily select the closest vertex that has not yet been processed, and performs this relaxation process on all of its outgoing edges; by contrast, the Bellman–Ford algorithm simply relaxes all the edges, and does this $|V| - 1$ times, where $|V|$ is the number of vertices in the graph.

2.3 Floyd-Warshall Algorithm: Computes shortest paths between all pairs of vertices.

The **Floyd–Warshall algorithm** (also known as **Floyd's algorithm**, the **Roy–Warshall algorithm**, the **Roy–Floyd algorithm**, or the **WFI algorithm**) is an algorithm for finding shortest paths in a directed weighted graph with positive or negative edge weights (but with no negative cycles). A single execution of the algorithm will find the lengths (summed weights) of shortest paths between all pairs of vertices. Although it does not return details of the paths themselves, it is possible to reconstruct the paths with simple modifications to the algorithm. Versions of the algorithm can also be used for finding the transitive closure of a relation, or (in connection with the Schulze voting system) widest paths between all pairs of vertices in a weighted graph.

3. Minimum Spanning Tree (MST) Algorithms

3.1 Prim's Algorithm: Builds MST by growing from a starting vertex.

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

The algorithm was developed in 1930 by Czech mathematician Vojtěch Jarník and later rediscovered and republished by computer scientists Robert C. Prim in 1957 and Edsger W. Dijkstra in 1959.

The most basic form of Prim's algorithm only finds minimum spanning trees in connected graphs. However, running Prim's algorithm separately for each connected component of the graph, it can also be used to find the minimum spanning forest. In terms of their asymptotic time complexity, these three algorithms are equally fast for sparse graphs, but slower than other more sophisticated algorithms. However, for graphs that are sufficiently dense, Prim's algorithm can be made to run in linear time, meeting or improving the time bounds for other algorithms.

The algorithm may informally be described as performing the following steps:

1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
2. Grow the tree by one edge: Of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
3. Repeat step 2 (until all vertices are in the tree).

3.2 Kruskal's Algorithm: Builds MST by adding edges in increasing weight order.

Kruskal's algorithm finds a minimum spanning forest of an undirected edge-weighted graph. If the graph is connected, it finds a minimum spanning tree. It is a greedy algorithm that in each step adds to the forest the lowest-weight edge that will not form a cycle. The key steps of the algorithm are sorting and the use of a disjoint-set data structure to detect cycles. Its running time is dominated by the time to sort all of the graph edges by their weight.

A minimum spanning tree of a connected weighted graph is a connected subgraph, without cycles, for which the sum of the weights of all the edges in the subgraph is minimal. For a disconnected graph, a minimum spanning forest is composed of a minimum spanning tree for each connected component.

This algorithm was first published by Joseph Kruskal in 1956, and was rediscovered soon afterward by Loberman & Weinberger (1957).

The algorithm performs the following steps:

- Create a forest (a set of trees) initially consisting of a separate single-vertex tree for each vertex in the input graph.
- Sort the graph edges by weight.
- Loop through the edges of the graph, in ascending sorted order by their weight. For each edge:
- Test whether adding the edge to the current forest would create a cycle.
- If not, add the edge to the forest, combining two trees into a single tree.

3.3 Borůvka's Algorithm: Another MST approach, useful for parallel computation.

Borůvka's algorithm is a greedy algorithm for finding a minimum spanning tree in a graph, or a minimum spanning forest in the case of a graph that is not connected.

The algorithm begins by finding the minimum-weight edge incident to each vertex of the graph, and adding all of those edges to the forest. Then, it repeats a similar process of finding the minimum-weight edge from each tree constructed so far to a different tree, and adding all of those edges to the forest. Each repetition of this process reduces the number of trees, within each connected component of the graph, to at most half of this former value, so after logarithmically many repetitions the process finishes. When it does, the set of edges it has added forms the minimum spanning forest.

3.4 Network Flow Algorithms

Ford-Fulkerson Method: Finds maximum flow in a network.
Edmonds-Karp Algorithm: Implementation of Ford-Fulkerson using BFS for efficiency.
Push-Relabel Algorithm: Efficient for large-scale flow problems.

3.5 Graph Coloring & Matching

Graph Coloring Algorithms: Assign colors to vertices such that no two adjacent vertices share the same color (used in scheduling, register allocation).
Maximum Matching Algorithms: Hopcroft-Karp algorithm for bipartite graphs.

3.6 Topological Sorting

Orders vertices in a Directed Acyclic Graph (DAG) so that each directed edge goes from earlier to later in the order.

Use cases: Task scheduling, dependency resolution.

4 Applications of Graph Theory in Real Life

4.1 Computer Science & Technology

Network Design: Graphs model computer networks where nodes represent devices and edges represent connections.

Data Routing: Algorithms like Dijkstra's shortest path help in efficient data transfer across the internet.

Database Management: Graph databases (e.g., Neo4j) store and query highly connected data such as social media relationships.

4.2 Transportation & Logistics

Route Optimization: Cities as nodes and roads as edges allow planners to find the shortest or fastest routes (used in Google Maps, GPS systems).

Traffic Flow Analysis: Graphs help manage congestion by modeling intersections and traffic lights.

Airline Scheduling: Airports and flights are modeled as graphs to minimize delays and optimize connections.

4.3 Social Networks

Relationship Mapping: Facebook, LinkedIn, and Twitter use graph theory to represent users (nodes) and friendships/follows (edges).

Recommendation Systems: Algorithms suggest friends, groups, or content by analyzing graph connectivity.

Influence Tracking: Identifying key influencers in a network relies on graph centrality measures.

4.4 Biology & Medicine

Disease Spread Modeling: Graphs simulate how infections move through populations, aiding epidemiology.

Protein Interaction Networks: Nodes represent proteins, edges represent interactions, helping researchers understand biological processes.

Neuroscience: Brain connectivity is studied using graph theory to map neural pathways.

4.5 Communication

Telecommunication Networks: Graphs optimize signal routing and bandwidth allocation.

Internet Infrastructure: Graph theory ensures resilience by modeling redundancy in server connections.

4.6 Operations Research & Business

Supply Chain Management: Graphs model suppliers, warehouses, and retailers to optimize distribution.

Project Scheduling: Critical Path Method (CPM) uses graph theory to identify task dependencies and minimize project duration.

Conclusion

Graph theory algorithms are the backbone of modern computing, logistics, and optimization. From BFS and DFS for exploration to Dijkstra's and Kruskal's for optimization, these algorithms enable efficient solutions to real-world problems in networking, transportation, and scheduling. Graph theory is not just abstract mathematics it is the backbone of modern systems like Google Maps, social media, epidemiology, and logistics. Its ability to simplify complex networks makes it indispensable in both academic research and everyday life.

References

1. Mehlhorn, Kurt; Sanders, Peter (2008). "Chapter 10. Shortest Paths" (PDF). *Algorithms and Data Structures: The Basic Toolbox*. Springer. doi:10.1007/978-3-540-77978-0. ISBN 978-3-540-77977-3.

2. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2022) [1990]. *Introduction to Algorithms* (4th ed.). MIT Press and McGraw-Hill. ISBN 0-262-04630-X.
3. Cormen, Thomas H.*;* Leiserson, Charles E.*;* Rivest, Ronald L. *(1990). Introduction to Algorithms (1st ed.). MIT Press and McGraw-Hill.* ISBN 0-262-03141-8.
4. Kenneth H. Rosen (2003). *Discrete Mathematics and Its Applications, 5th Edition*. Addison Wesley. ISBN 978-0-07-119881-3.
5. *Kepner, Jeremy; Gilbert, John (2011), Graph Algorithms in the Language of Linear Algebra, Software, Environments, and Tools, vol. 22,* Society for Industrial and Applied Mathematics*, p. 55,* ISBN 9780898719901.
6. Loberman, H.; Weinberger, A. (October 1957). "Formal Procedures for connecting terminals with a minimum total wire length". *Journal of the ACM*. **4** (4): 428–437. doi:10.1145/320893.320896. S2CID 7320964.

