Graphs-II

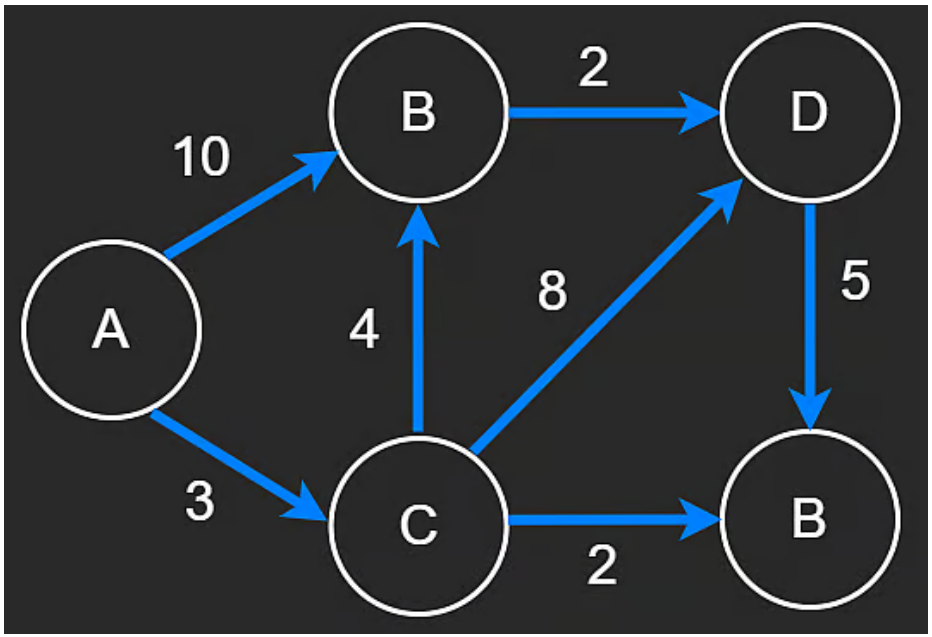
Presented by Yasin Ceran

March 28, 2024

Dijkstra's Algorithm

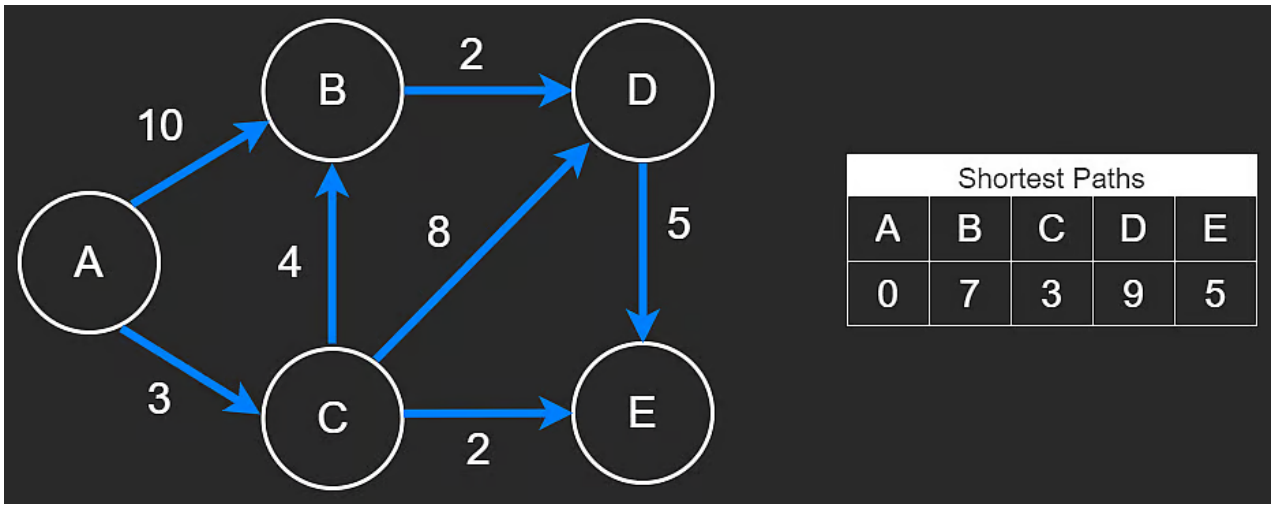
- Unlike BFS, which is optimal for unweighted graphs, Dijkstra's algorithm excels in weighted graph scenarios, providing the shortest path by considering edge weights.
- Dijkstra's algorithm can revisit nodes if it discovers a path with a lower cumulative weight, ensuring that the shortest path based on weight is always identified.

- Example scenario: In an unweighted graph, BFS could easily determine the shortest path from a source node A to a destination node D by the number of vertices traversed. But in weighted graphs, where edge weights vary, BFS's approach doesn't yield the shortest path based on total weight.



The Setup for Dijkstra's Algorithm

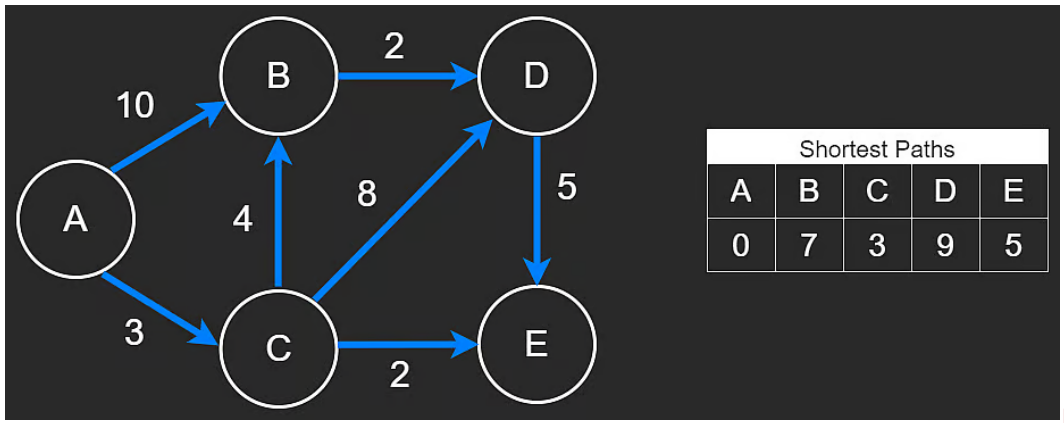
- Problem Statement: Starting from node A, determine the shortest path length to all other nodes in a weighted graph.
- Objective: Identify paths with the smallest total weight (cumulative edge weights), which we refer to as the "lightest" or shortest paths.
- Example Graph: Consider a graph with vertices A, B, C, D, E, and weighted edges such as [A,B,10], [A,C,3], [B,D,2], [C,B,4], [C,D,8], [C,E,2], [D,E,5].



Shortest Paths				
A	B	C	D	E
0	7	3	9	5

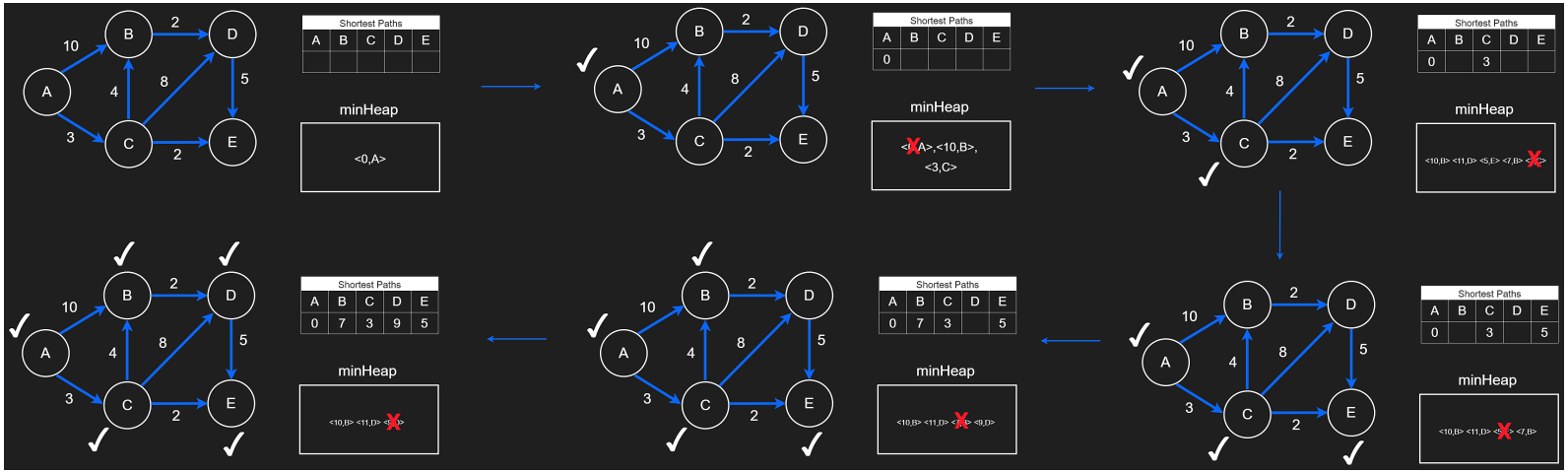
Walk-through of Dijkstra's Algorithm

- Initial Setup: Starting from vertex A, the distance from A to A is 0. Potential paths from A include A -> B with a weight of 10, and A -> C with a weight of 3.
- Path Exploration:
  - For vertex B, the shortest path is updated to 7 (A -> C -> B) since it is more efficient than directly from A to B.
  - For vertex C, the most direct and only path is A -> C, making the shortest path weight 3.
  - For vertex D, the shortest path is found through A -> C -> B -> D with a cumulative weight of 9.
  - For vertex E, the shortest path is A -> C -> E with a total weight of 5. It is noted that other paths exist but are not as efficient.



Shortest Paths				
A	B	C	D	E
0	7	3	9	5

Implementation of Dijkstra's Algorithm



Min-Heap Utilization

The algorithm utilizes a min-heap to prioritize paths with the minimum weight. Each entry in the min-heap contains the node identifier and its associated cost, ensuring paths are sorted by their cost.

Data Structures

- A min-heap to manage the exploration of paths based on ascending costs.
- A hashmap to track the shortest paths from the source to every vertex.

Dijkstra's Algorithm-Python Code

```
import heapq

# Given a connected graph represented by a list of edges, where
# edge[0] = src, edge[1] = dst, and edge[2] = weight,
# find the shortest path from src to every other node in the
# graph. There are n nodes in the graph.
# O(E * logV), O(E * logE) is also correct.
def shortestPath(edges, n, src):
    adj = {}
    for i in range(1, n + 1):
        adj[i] = []

    # s = src, d = dst, w = weight
    for s, d, w in edges:
        adj[s].append([d, w])

    shortest = {}
    minHeap = [[0, src]]
    while minHeap:
        w1, n1 = heapq.heappop(minHeap)
        if n1 in shortest:
            continue
        shortest[n1] = w1

        for n2, w2 in adj[n1]:
            if n2 not in shortest:
                heapq.heappush(minHeap, [w1 + w2, n2])
    return shortest
```

Time Complexity of Dijkstra's Algorithm

- **Overview**: Dijkstra's algorithm's time complexity is primarily influenced by the operations on the min-heap and the structure of the graph.
- **Complexity Analysis**:
  - The time complexity is denoted as $O(E \log E)$, where E represents the number of edges in the graph.
  - In a dense graph scenario, where each vertex is connected to every other vertex, the maximum number of edges approaches V^2, making the graph fully connected.
  - The algorithm's reliance on the min-heap for edge selection means that both insertion and removal operations are bound by $O(\log n)$ complexity, where n is the number of elements in the heap.
  - Given that, in the worst-case scenario, the heap may contain all edges of the graph, the overall time complexity sums up to $O(E \log E)$, accounting for the heap operations across all edges.
- **Implications**: This complexity indicates that while Dijkstra's algorithm is efficient for finding the shortest path in weighted graphs, its performance may vary significantly with the density of the graph and the distribution of edge weights.

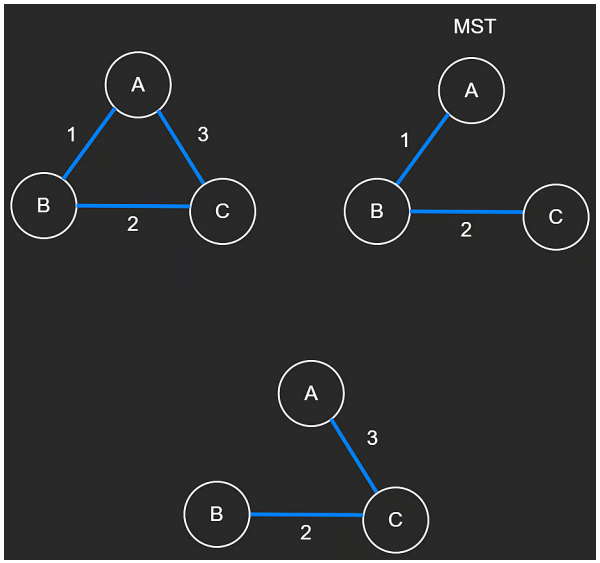
Prim's Minimum Spanning Tree Algorithm

- Prim's algorithm is a greedy algorithm utilized for finding the minimum spanning tree (MST) of a weighted undirected graph.
- **Spanning Tree**: A subset of G's edges forming a tree that includes all the vertices with minimized total weight.
- **Key Characteristics**:
  - A tree cannot have cycles, implying an MST for a graph G with n nodes contains exactly $n - 1$ edges.
  - The algorithm begins with an empty tree and iteratively adds the least weight edge from the graph that connects a vertex in the tree to a vertex outside.

Prim's Process

Process

1. Starting with a single vertex, edges connecting this vertex to other vertices are evaluated.
2. At each step, the edge with the minimum weight connecting the tree to a new vertex is added.
3. This process repeats until the tree spans all n vertices, resulting in $n - 1$ edges.



Example:

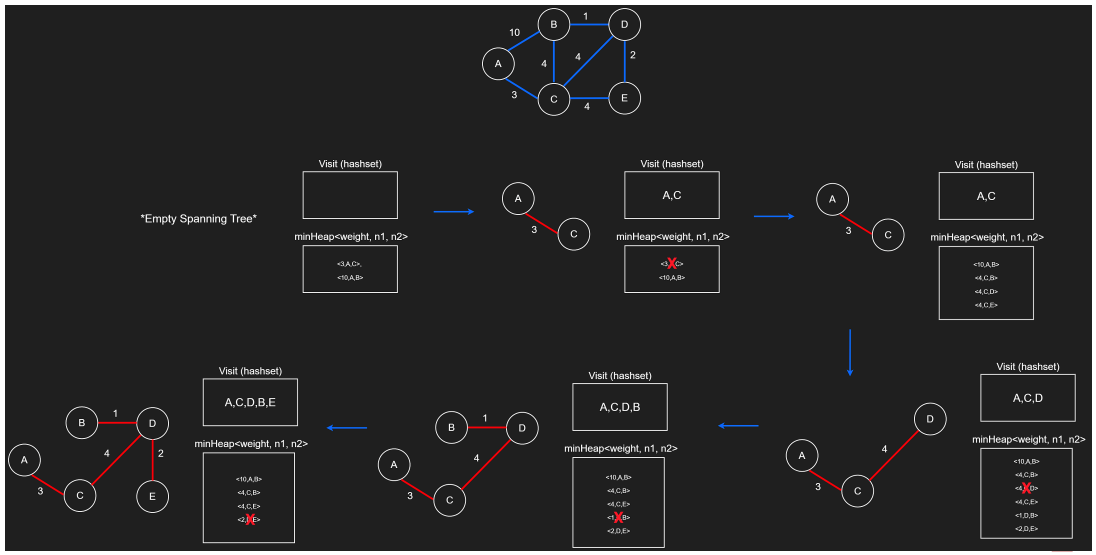
- Beginning at vertex A, the algorithm first adds edge A->B (cost 1), followed by B->C (cost 2), totaling a minimum cost of 3 for the MST.
- Outcome: Prim's algorithm efficiently finds an MST, ensuring connectivity between all vertices with minimal total edge weight, without forming any cycles.

One application of the minimum spanning tree is a road network among cities. Using the example above, if we wanted to connect city A, B and C together, it can be done so using the concept of a MST.

Prim's Algorithm: The Algorithmic Essence

Prim's algorithm, akin to Dijkstra's, adeptly finds the minimum spanning tree (MST) for a graph, utilizing a min-heap for efficient selection of the lowest-cost connections between vertices.

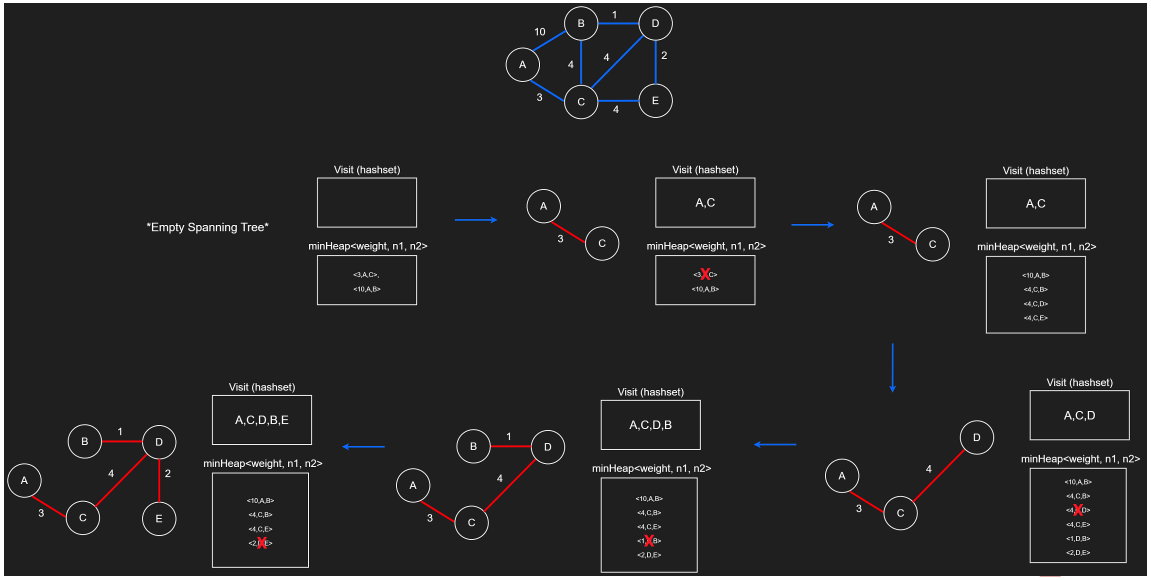
- Each node in the min-heap is characterized by $\langle$weight, n1, n2$\rangle$, indicating the cost to move from vertex *n1* to *n2*.
- A `visit` hashset tracks visited nodes to prevent cyclic paths, ensuring a growing tree structure.
- The MST is incrementally constructed by selecting the smallest available edge not forming a cycle.



Algorithm Termination Conditions

Prim's algorithm employs strategic checks to determine the completion of the MST construction:

1. The algorithm proceeds until the min-heap is exhausted, indicating all possible edges have been considered.
2. It halts once every vertex has been visited (`visit.size() == n`), signifying full coverage of the graph.
3. A termination signal is also the attainment of $n - 1$ edges in the MST, aligning with the definition of a spanning tree.



Code Implementation for Prim's Algorithm

To transform an edge list into a usable structure for Prim's Algorithm, we construct an adjacency list that accounts for the undirected nature of the graph.

Building the Adjacency List

We initialize `adj`, a hashmap, to represent each vertex's connections:

- Key: Vertex
- Value: List of neighboring vertices and their connecting edge weights

This setup ensures that each edge's bidirectional nature is accurately represented, setting the stage for the MST construction.

```
import heapq
# Given a list of edges of a connected undirected graph,
# with nodes numbered from 1 to n,
# return a list edges making up the minimum spanning tree.
def minimumSpanningTree(edges, n):
    adj = {}
    for i in range(1, n + 1):
        adj[i] = []
    for n1, n2, weight in edges:
        adj[n1].append([n2, weight])
        adj[n2].append([n1, weight])
```

Prim's Algorithm: Heap and MST Construction

In Prim's Algorithm, the selection of edges for the Minimum Spanning Tree (MST) relies on a priority queue, implemented as a min-heap, to ensure we're always processing the smallest weighted edge available.

Heap Initialization and MST Construction

- Initialize the minHeap with the source node's (node 1) neighbors.
- Use `mst` list to track edges of the MST.
- Use `visit` set to track visited nodes.

```
# Initialize the heap by choosing a single node
# (in this case 1) and pushing all its neighbors.
minHeap = []
for neighbor, weight in adj[1]:
    heapq.heappush(minHeap, [weight, 1, neighbor])

mst = []
visit = set()
visit.add(1)
while len(visit) < n:
    weight, n1, n2 = heapq.heappop(minHeap)
    if n2 in visit:
        continue

    mst.append([n1, n2])
    visit.add(n2)
    for neighbor, weight in adj[n2]:
        if neighbor not in visit:
            heapq.heappush(minHeap, [weight, n2, neighbor])
return mst
```

Time Complexity of Prim's Algorithm

Complexity Analysis

The algorithm's efficiency is closely related to Dijkstra's, leading to similar time complexity:

- **Time Complexity**: $O(E \log V)$, where E is the number of edges and V is the number of vertices in the graph. This is due to the use of a min-heap for edge selection.
- **Space Complexity**: $O(E)$, attributable to the space required for maintaining the heap and the set of visited nodes, which scales with the number of edges.

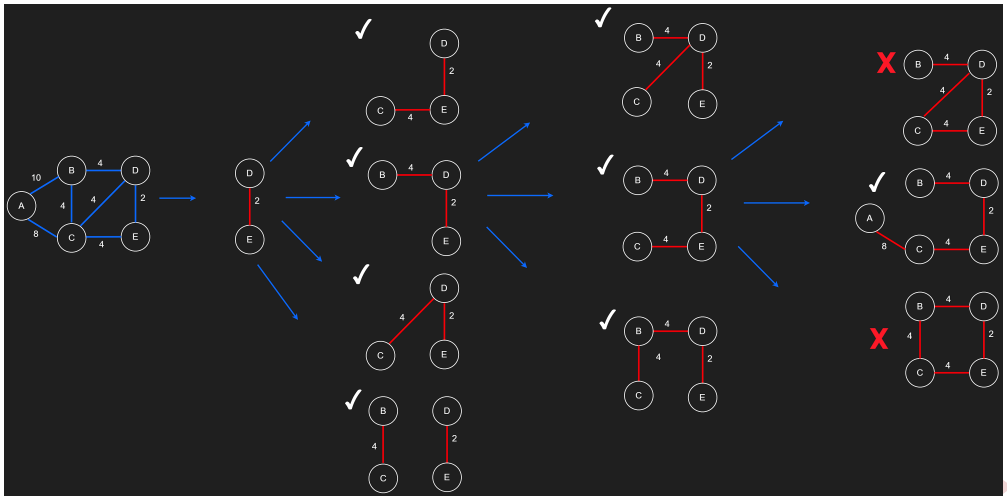
Kruskal's Algorithm Overview

Introduction

Kruskal's algorithm finds the minimum spanning tree (MST) for an undirected weighted graph. It's a greedy algorithm effective on sparse graphs, as opposed to Prim's algorithm which is more suited for dense graphs.

Algorithm's Essence

- Sort edges by ascending weight.
- Begin with an empty MST.
- Sequentially consider each edge:
  - If adding the edge does not form a cycle, include it in the MST.
  - Otherwise, discard the edge.



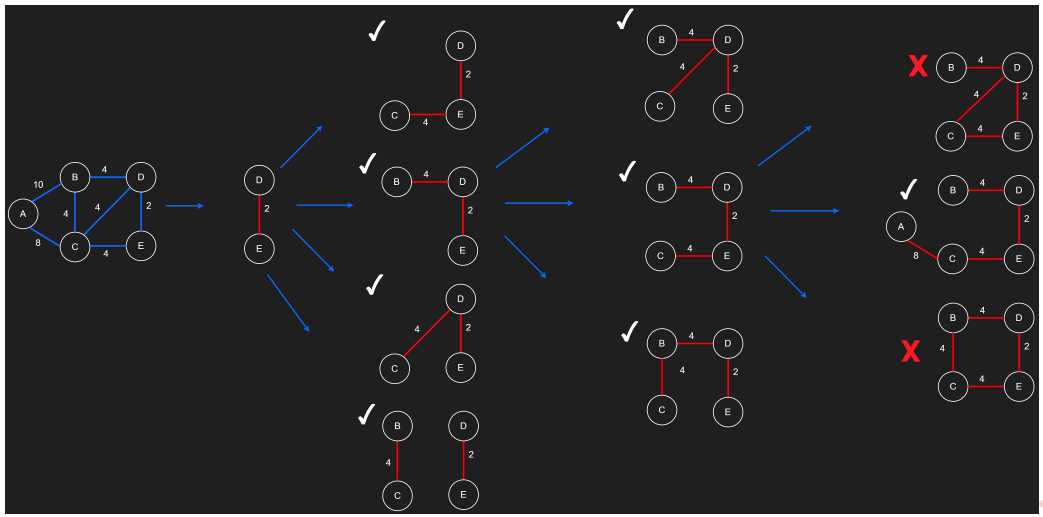
Cycle Detection in Kruskal's Algorithm

Utilizing Union-Find Data Structure

The Union-Find data structure is instrumental in cycle detection for Kruskal's algorithm. It helps manage disjoint sets to ensure a cycle is not formed by adding a new edge.

Key Points

- Maintains the parent for each vertex.
- Vertices in the process of union that share the same parent indicate a cycle, preventing their addition.
- Multiple spanning trees with the same weight are viable, but the algorithm ensures minimal total cost.



Kruskal's Algorithm Implementation

Implementation Strategy

- Utilize a minHeap to sort all edges by their weight to ensure we're always considering the edge with the minimum weight next.
- Directly work with edges, bypassing the need for an adjacency list, focusing on global minimum weights.

Union-Find Data Structure

- Initialize the UnionFind structure with each vertex pointing to itself.
- For each edge popped from the minHeap:
  - Attempt to union the vertices of the edge.
  - If union is successful (no cycle created), include the edge in the MST.
  - If a cycle would be formed, discard the edge.
- Continue until the MST contains $n - 1$ edges.

```python
    # Given an list of edges of a connected undirected graph,
# with nodes numbered from 1 to n,
# return a list edges making up the minimum spanning tree.
def minimumSpanningTree(edges, n):
    minHeap = []
    for n1, n2, weight in edges:
        heapq.heappush(minHeap, [weight, n1, n2])

    unionFind = UnionFind(n)
    mst = []
    while len(mst) < n - 1:
        weight, n1, n2 = heapq.heappop(minHeap)
        if not unionFind.union(n1, n2):
            continue
        mst.append([n1, n2])
    return mst
```

Key Implementation Note

The UnionFind class, used here, facilitates efficient cycle detection and set management.

Kruskal's Algorithm: Time and Space Complexity

Time Complexity

- Utilizing a min-heap for edge management and the Union-Find data structure for cycle detection impacts time complexity.
- The min-heap operations (add/remove) have a time complexity of $O(\log V)$, applying the log power rule.
- Total time complexity for Kruskal's algorithm is $O(E \log V)$, with V being the number of vertices.

Space Complexity

- The space complexity is primarily dictated by the storage of edges.
- With E representing the number of edges, the memory complexity stands at $O(E)$.

Comparative Analysis

While similar to Prim's algorithm in finding the MST, Kruskal's distinct use of Union-Find for cycle detection and edge-focused processing differentiates its approach.

Topological Sort: The Idea

The Idea:

- Topological sort is a linear ordering of vertices in a directed acyclic graph (DAG), where for each directed edge uv, vertex u comes before v in the ordering.
- This sorting is particularly useful in scenarios where you need to understand the sequence of tasks or events, like prerequisites in university courses.
- In a DAG, if there's a path from vertex u to vertex v, u must appear before v in the order.
- It provides a way to execute tasks while respecting all precedence relationships.

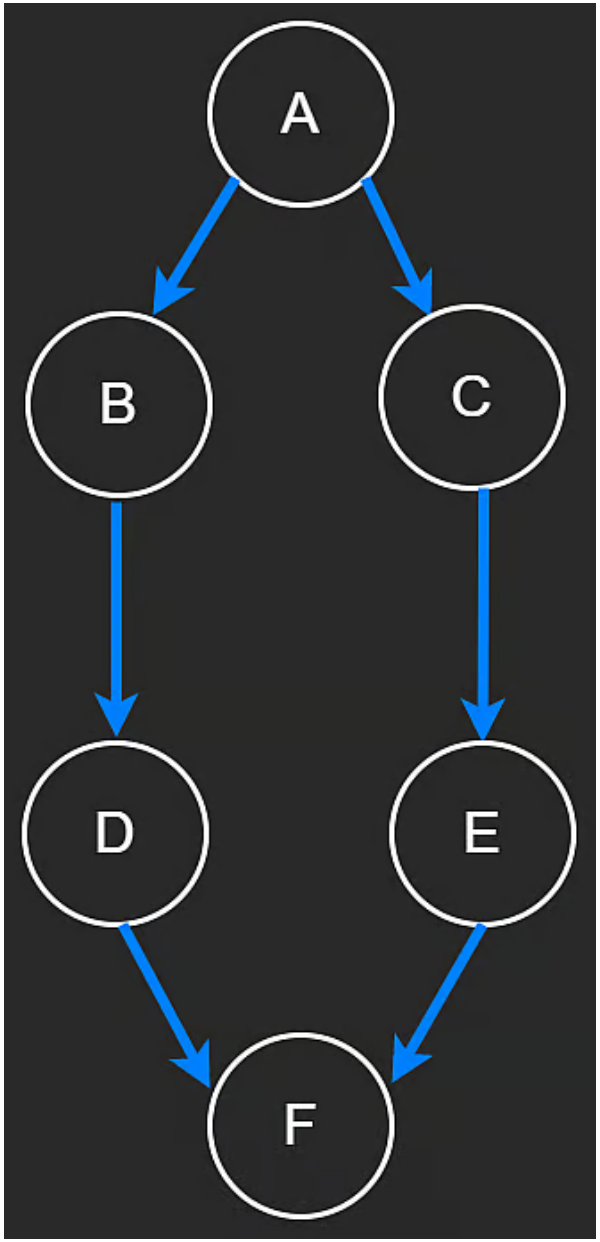
Illustration: University Courses

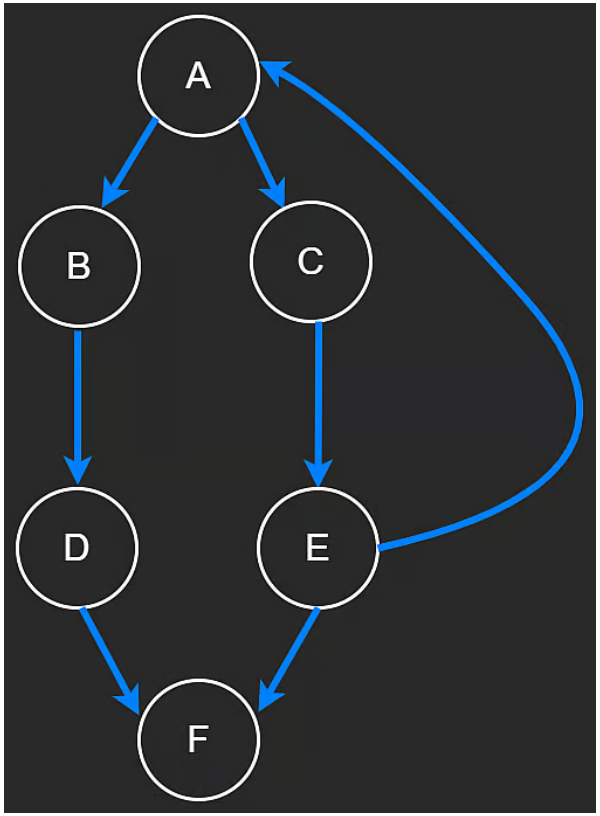
- Consider university courses as nodes in a DAG, where edges represent prerequisite requirements.
- Some courses (nodes) can be taken without prerequisites, while others depend on the completion of prior courses.
- A topological sort of this graph would provide an order in which to take the courses, ensuring all prerequisites are met.
- Example: If Course C depends on Courses A and B, then A and B will appear before C in the topological ordering.

Key Points in Topological Sorting

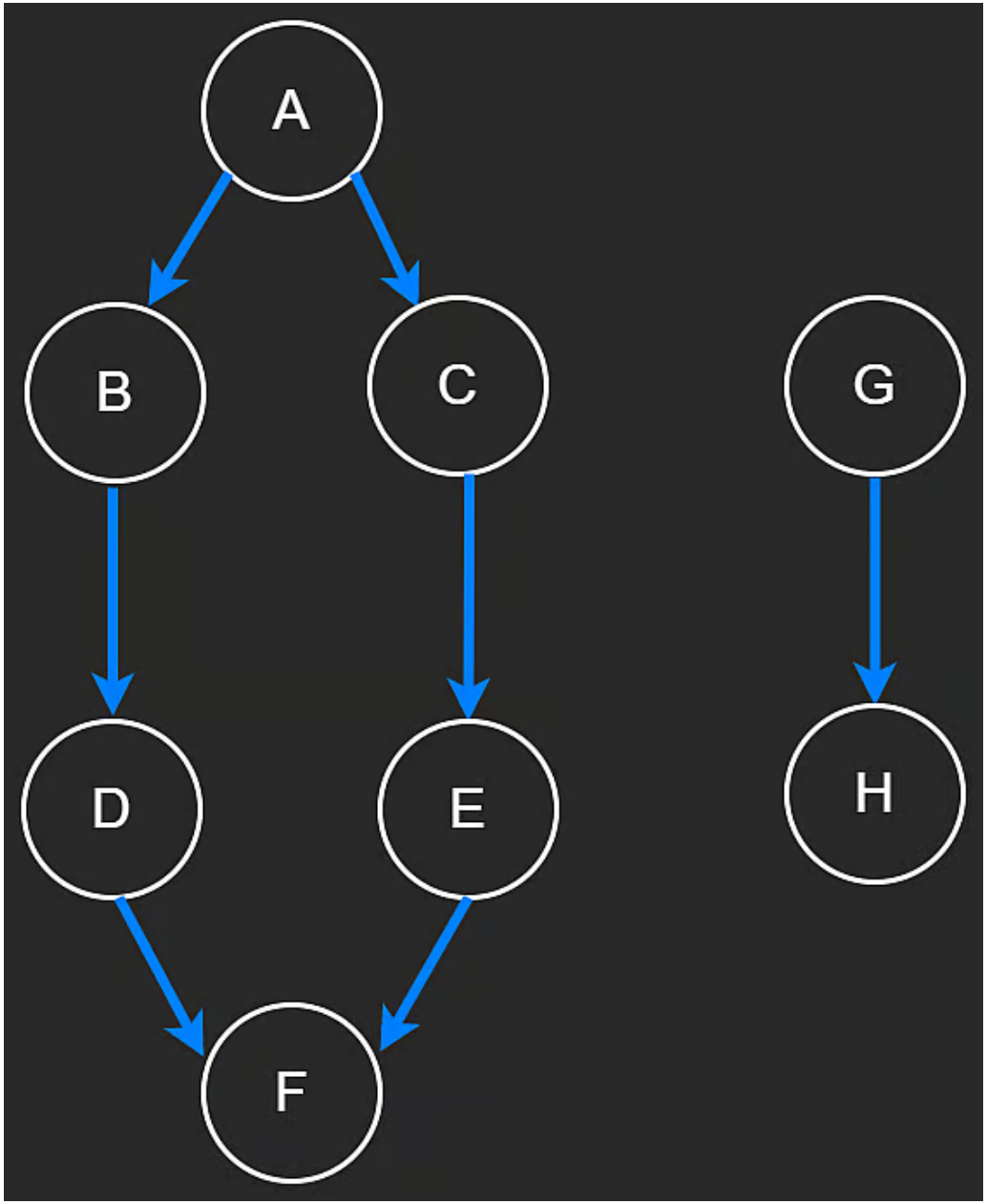
- It's applicable only to DAGs. A cyclic graph does not have a valid topological ordering.
- The order is not necessarily unique. There can be multiple valid topological sorts for a given graph.
- Topological sorts are used in scheduling tasks, ordering compiler tasks, resolving symbol dependencies in linkers, etc.

Example-I





Example-II



The Algorithm for Topological Sort-I

- Topological sort can be performed using either Breadth-First Search (BFS) or Depth-First Search (DFS).
- In our discussion, we focus on a recursive DFS approach to traverse the graph and determine the topological order.

Base Case and Challenges

- The base case in our DFS approach seems to be when we reach the final node, for example, F in a given graph sequence.
- However, directly targeting F leads to revisiting it due to multiple paths (e.g., D->F and E->F), causing an incorrect order.
- To prevent revisiting nodes, a hashset named *visit* is used to track visited nodes during the traversal.

The Algorithm for Topological Sort-II

Encountering an Incorrect Order

- A naive DFS traversal results in an incorrect topological order, with dependencies not properly respected (e.g., F appears before its dependents C and E).
- This necessitates a strategy to ensure all dependencies of a node are processed before the node itself.

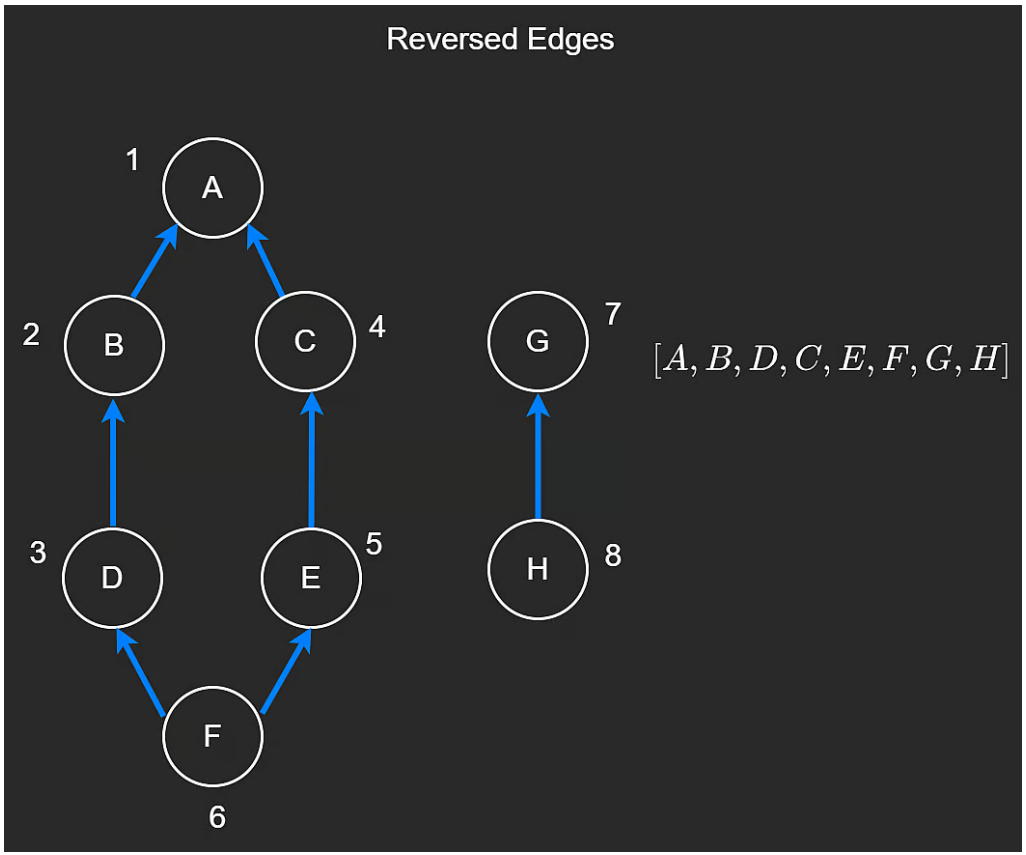
Addressing the Order Issue

- To correct the order and ensure dependencies are met, two techniques can be utilized in the DFS traversal:
  1. Adjusting the traversal strategy to account for dependencies explicitly.
  2. Implementing post-visit actions that ensure a node is added to the topological order only after all its dependents have been visited.

The Algorithm for Topological Sort-III

Reverse the Edges

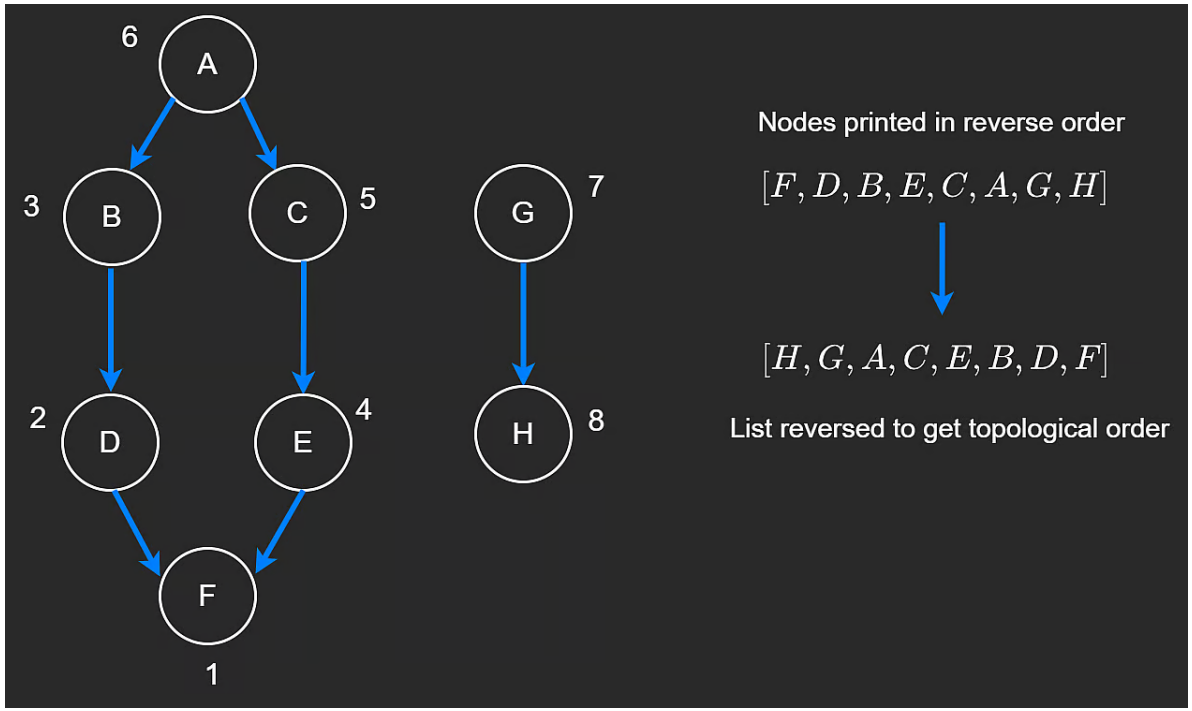
- Reverse the edges of the graph and run a post-order traversal. Recall that post-order traversal is: left, right, root. This will give us the correct topological order, which is shown below.
- The numbers next to the vertices represent the order in which they are visited.



The Algorithm for Topological Sort-IV

Post Order Traversal

- Instead of reversing the graph, perform a post-order traversal and reverse the resultant array instead. This saves us from reversing the graph and still gives us the correct topological order.



Topological Sort without a Known Start Point

Challenge: Graph problems often do not specify a clear starting point for traversal or processing, particularly in scenarios involving dependencies or hierarchical structures.

Solution Approach:

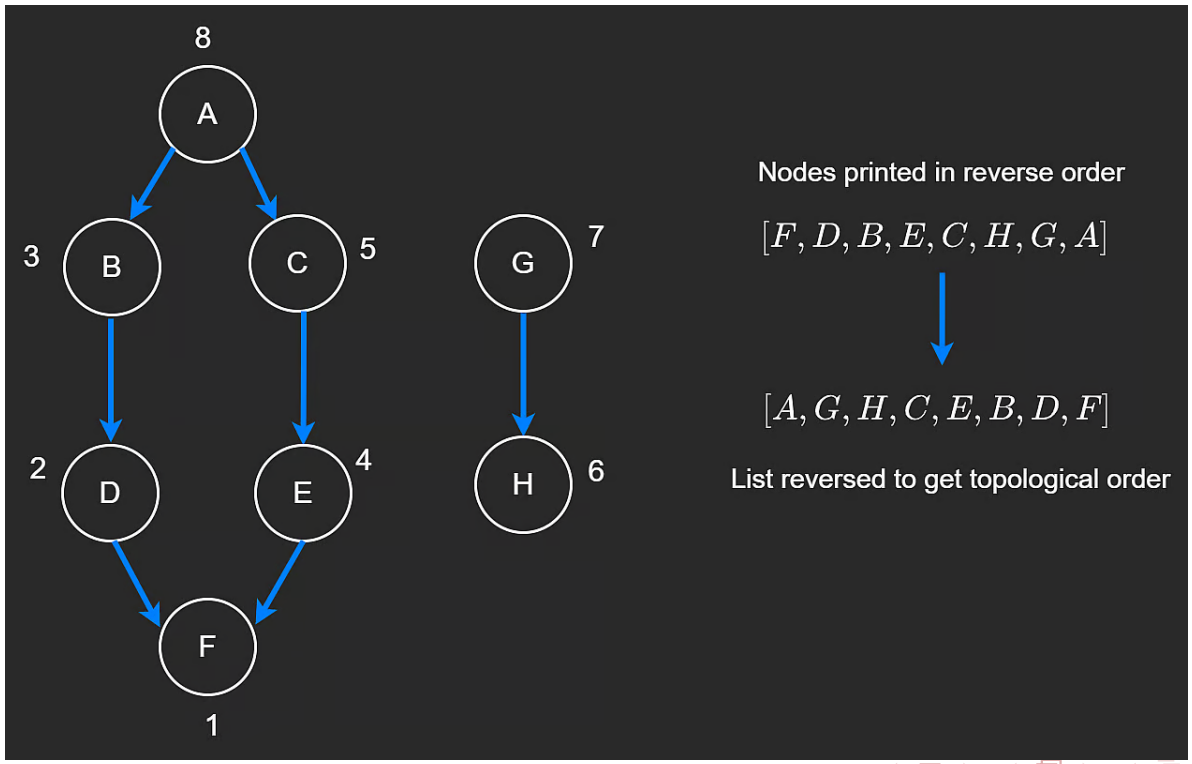
- Instead of initiating a Depth-First Search (DFS) from a known "head" or source vertex, the strategy involves executing DFS on every vertex within the graph.
- This comprehensive approach ensures that each component of the graph is explored in the sequence they are encountered or listed, regardless of the initial absence of a defined starting point.

Application:

- This method is particularly useful in scenarios where dependencies among tasks, courses, or modules are represented in a Directed Acyclic Graph (DAG) without an explicit sequence of execution.
- It enables the identification and ordering of vertices (tasks, courses, etc.) based on their interdependencies, effectively addressing the challenge posed by the lack of a known start point.

The Algorithm for Topological Sort-V

Suppose the first four vertices show up in the order B,C,H,A. In this case, we can just perform post-order DFS one vertex at a time. We can visit neighbors of B, then neighbors of C and finally neighbors of H and A. By the time we visit the other nodes, we will have already visited them or their neighbors, so we can simply return. This would look like the following.



Implementation of Topological Sort

Objective: To determine the topological ordering of vertices in a given Directed Acyclic Graph (DAG).

Steps for Implementation:

1. **Build the Adjacency List:** Utilize the provided array of edges to construct an adjacency list representing the graph.
2. **Initialize Data Structures:** Create a list, *topSort*, to store the topological ordering, and a hashset, *visit*, to keep track of visited vertices and prevent revisits.
3. **Perform DFS:** Execute Depth-First Search (DFS) on each vertex not yet visited, ensuring the exploration of all vertices and their dependencies.
4. **Update Topological Order:** Append vertices to the *topSort* list in the reverse order of their completion in DFS, effectively establishing the topological order.
5. **Return the Result:** Finalize and return the *topSort* list, which now contains the vertices sorted topologically.

Outcome: The returned *topSort* list provides a sequential ordering of the graph's vertices, respecting their interdependencies and ensuring that for every directed edge uv, vertex u precedes vertex v in the ordering.

Implementation of Topological Sort-Code

```
    # Given a directed acyclical graph, return a valid
# topological ordering of the graph.
def topologicalSort(edges, n):
    adj = {}
    for i in range(1, n + 1):
        adj[i] = []
    for src, dst in edges:
        adj[src].append(dst)

    topSort = []
    visit = set()
    for i in range(1, n + 1):
        dfs(i, adj, visit, topSort)
    topSort.reverse()
    return topSort

def dfs(src, adj, visit, topSort):
    if src in visit:
        return
    visit.add(src)

    for neighbor in adj[src]:
        dfs(neighbor, adj, visit, topSort)
    topSort.append(src)
```

Dijkstra's Algorithm

Purpose: Finds the shortest path from a single source to all vertices in a weighted graph. **Key Points:**

- Operates on both directed and undirected graphs.
- Utilizes a priority queue to select the minimum distance vertex.
- Cannot handle negative weight edges.
- Time Complexity: $O(E + V \log V)$.

Prim's Algorithm

Purpose: Generates a minimum spanning tree for a connected weighted undirected graph. **Key Points:**

- Grows the spanning tree from a starting vertex.
- Adds edges with the lowest weight that do not form a cycle.
- Uses a priority queue to keep track of the edges.
- Time Complexity: $O(E + V \log V)$.

Kruskal's Algorithm

Purpose: Finds the minimum spanning tree for a graph by adding edges in order of increasing length. **Key Points:**

- Works well for sparse graphs.
- Sorts all edges before processing.
- Uses Union-Find data structure to detect cycles.
- Time Complexity: $O(E \log V)$.

Topological Sort

Purpose: Orders vertices in a directed acyclic graph such that for every directed edge uv, vertex u comes before v. **Key Points:**

- Can be performed using DFS or BFS (Kahn's Algorithm).
- Useful for scheduling tasks, ordering events, etc.
- Provides linear ordering of vertices.
- Time Complexity: $O(V + E)$.