Graphs-III

Presented by Yasin Ceran

August 5, 2024

Understanding Union-Find

Purpose:

Union-Find is primarily used to manage a partition of a set into disjoint subsets and is highly efficient in checking and updating the connections between components.

Applications:

- Efficiently determines whether two elements are in the same subset.
- Can quickly unite two subsets into a single subset.

Disjoint Sets:

- Disjoint sets are those that have no common elements.
- Examples: $S1 = \{1, 2, 3\}$ and $S2 = \{4, 5, 6\}$ are disjoint.
- Sets $S3 = \{1, 2, 5\}$ and $S4 = \{5, 6, 7\}$ are not disjoint since they share an element.

Functionality:

- **Union:** Combines two subsets into a single subset.
- **Find:** Determines which subset a particular element is in.
- Used in scenarios where the graph is dynamic, making it preferable over static graph analysis methods like DFS when edges are added over time.

Detailed Operations in Union-Find

Union-Find Operations:

- **Find Operation:** Identifies the root parent of a vertex, incorporating path compression for efficiency.
- **Union Operation:** Merges two distinct sets, maintaining tree balance using rank optimization.
- **Aim:** Ensure no cycle formation by preventing a vertex from reconnecting within its set.

Cycle Prevention:

- Critical in maintaining acyclic properties in graph structures, especially important in algorithms like Kruskal's for MST.

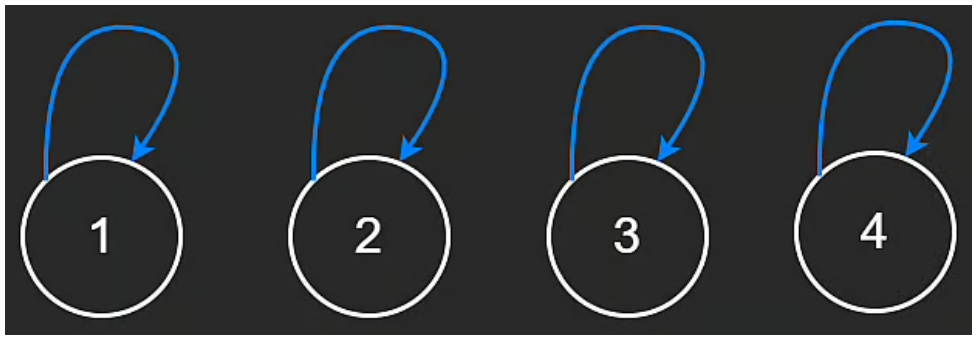
Understanding Union-Find for Graph Cycle Detection

Example Introduction:

- Given edges: $[1, 2]$, $[4, 1]$, $[2, 4]$.
- Task: Detect cycles in a graph using Union-Find.

Initial Setup:

- Each vertex initially points to itself, forming individual sets.
- Union-Find is visualized as a forest of trees, with each vertex as a standalone tree.



Connecting Components:

- For the connection $[2, 1]$, choose 2 as child of 1.
- Parent-child roles are initially arbitrary and adjust based on tree height (rank) during further operations.

Note: This frame sets up the scenario and initial conditions for using Union-Find in graph processing.

Implementation of Union-Find

UnionFind Class:

- **Data Structures:**
  - **Parent Map:** Tracks the parent of each node.
  - **Rank Map:** Tracks the "rank" of each tree in the forest, used to optimize union operations.
- **Initialization:**
  - Each node is its own parent initially.
  - Each node has an initial rank of 0.
- **Choice of Data Structure:**
  - Using hashmaps for flexibility and ease of use.
  - Arrays can also be used, especially when vertex identifiers are compact and sequential.

```
class UnionFind:
def __init__(self, n):
    self.par = {}
    self.rank = {}

    for i in range(1, n + 1):
        self.par[i] = i
        self.rank[i] = 0
```

Methods:

- **Find:** Path compression to keep trees flat.
- **Union:** Union by rank to ensure that the smaller tree is always attached under the larger tree, preventing deep trees.

Implementation of 'Find' in Union-Find

Purpose

Locate the root parent of a given vertex, employing path compression to optimize future lookups.

Implementation:

- Input: A vertex n.
- Output: The root parent of vertex n.

```
def find(self, n):
    p = self.parent[n]
    while p != self.parent[p]:
        self.parent[p] = self.parent[self.parent[p]]   # Path Compression
        p = self.parent[p]
    return p
```

Path Compression:

- Reduces the height of the tree by making nodes directly point to their grandparent.
- Improves the efficiency of future 'find' operations.

Union Operation in Union-Find

Purpose

To connect two vertices if they are in different components, updating their parents and ranks accordingly.

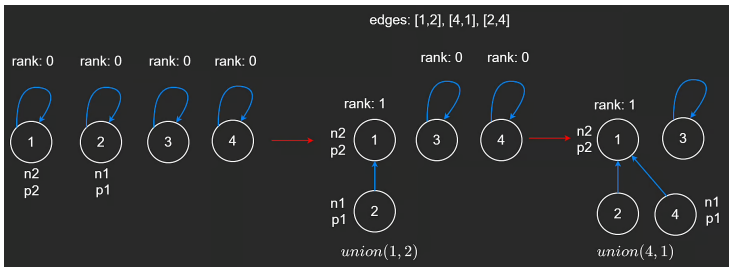
Implementation:

To connect two vertices if they are in different components, updating their parents and ranks accordingly.

```
def union(self, n1, n2):
    p1, p2 = self.find(n1), self.find(n2)
    if p1 == p2:
        return False  # Already in the same set

    if self.rank[p1] > self.rank[p2]:
        self.par[p2] = p1  # p1 becomes the parent
    elif self.rank[p1] < self.rank[p2]:
        self.par[p1] = p2  # p2 becomes the parent
    else:
        self.par[p1] = p2  # Tie, p2 becomes the parent
        self.rank[p2] += 1
    return True
```

The below visual demonstrates the find and the union function given edges = [1,2], [4,1], [2,4]. Notice that we connect 2 to 1, then we connect 4 to 1 because 1 has a higher rank. But, when we reach [2,4], 2's parent is 1 and 4's parent is also 1, meaning they belong to the same connected component, i.e. there is a cycle.



Time and Space Complexity of Union-Find

Complexity Analysis

- **Naive Case:**
  - The `find` function can potentially traverse every node in the worst case (forming a chain).
  - Complexity: $O(n)$.
- **Path Compression:**
  - Updates the parent to be the grandparent, flattening the structure significantly.
  - Reduces the time complexity to $O(\log n)$.
- **Overall Union-Find:**
  - With union by rank and path compression, the complexity becomes nearly constant, denoted by Inverse Ackermann function $\alpha(n)$.
  - For most practical input sizes, $\alpha(n)$ is assumed to be $O(1)$.
  - Overall complexity with m operations: $O(m\alpha(n))$, simplified to $O(m)$.

Remark:

Union-Find is extremely efficient for problems involving connectivity checking and component tracking in dynamic graphs, making it essential for competitive programming and network analysis.

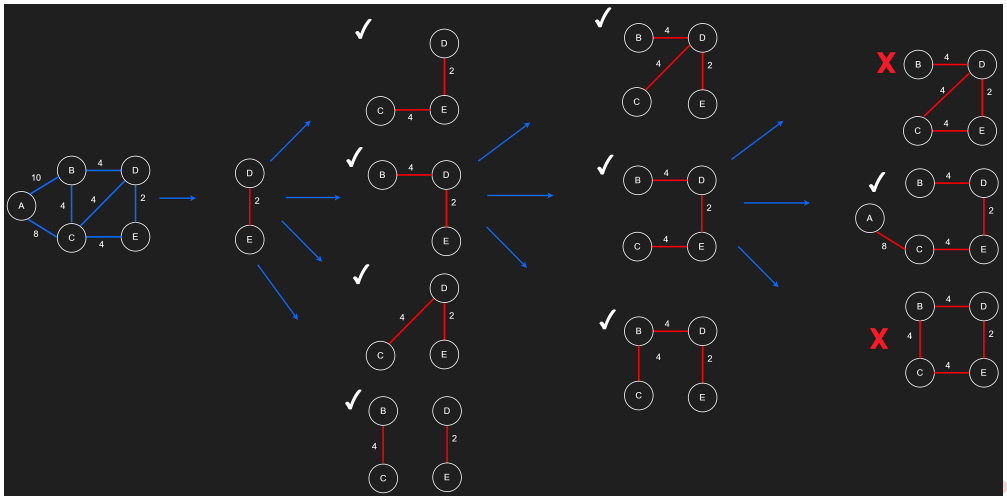
Kruskal's Algorithm Overview

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.

```
    # 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.

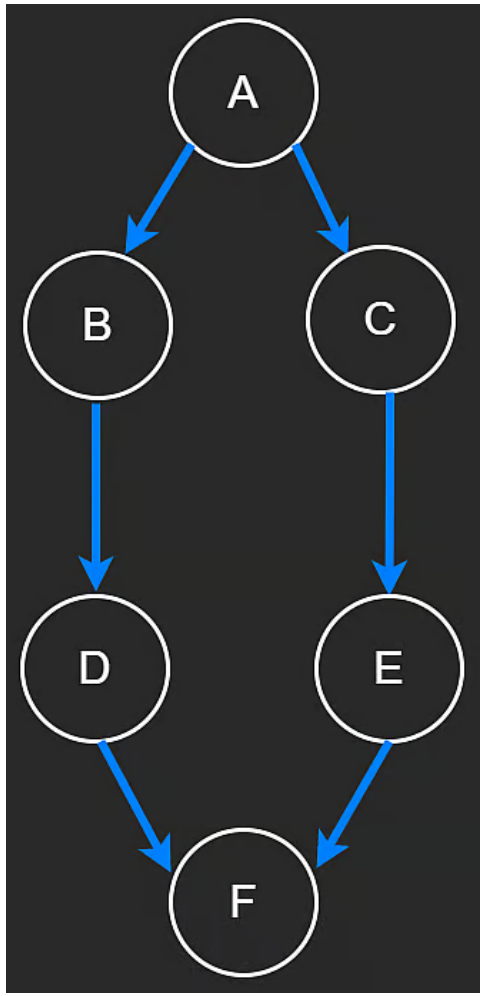
Key Points in Topological Sorting

- 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.

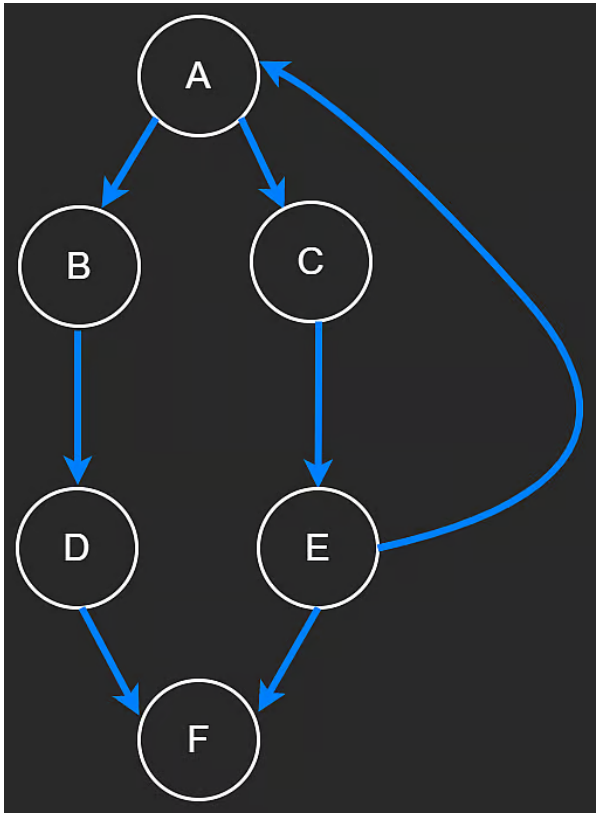
Example I

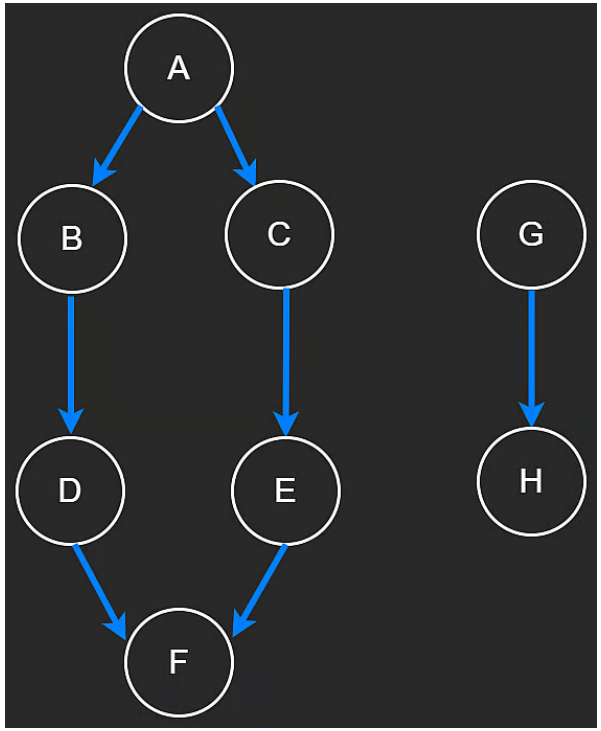
Illustration: University Courses

- 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.

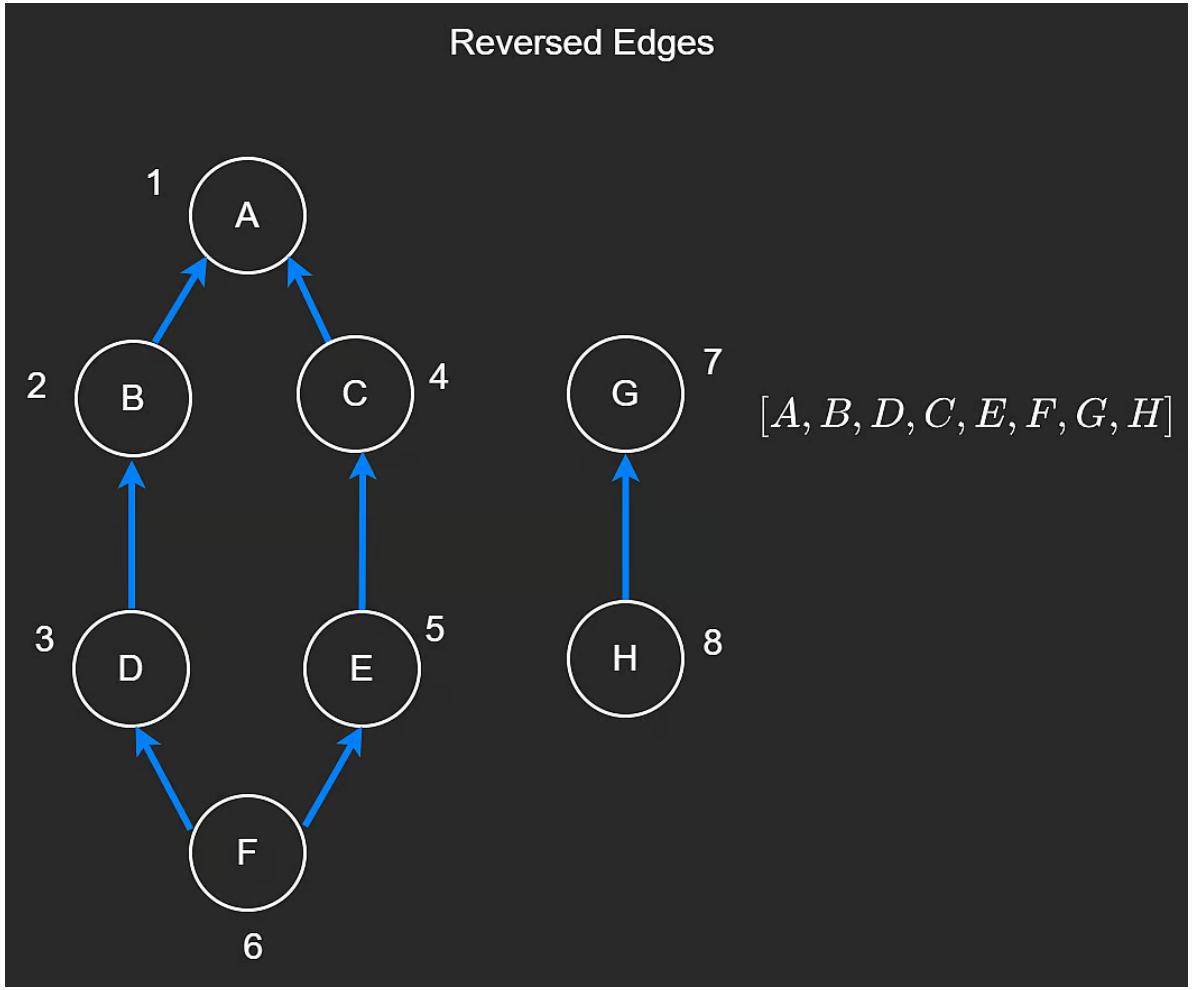


Example-II





Example-III



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.

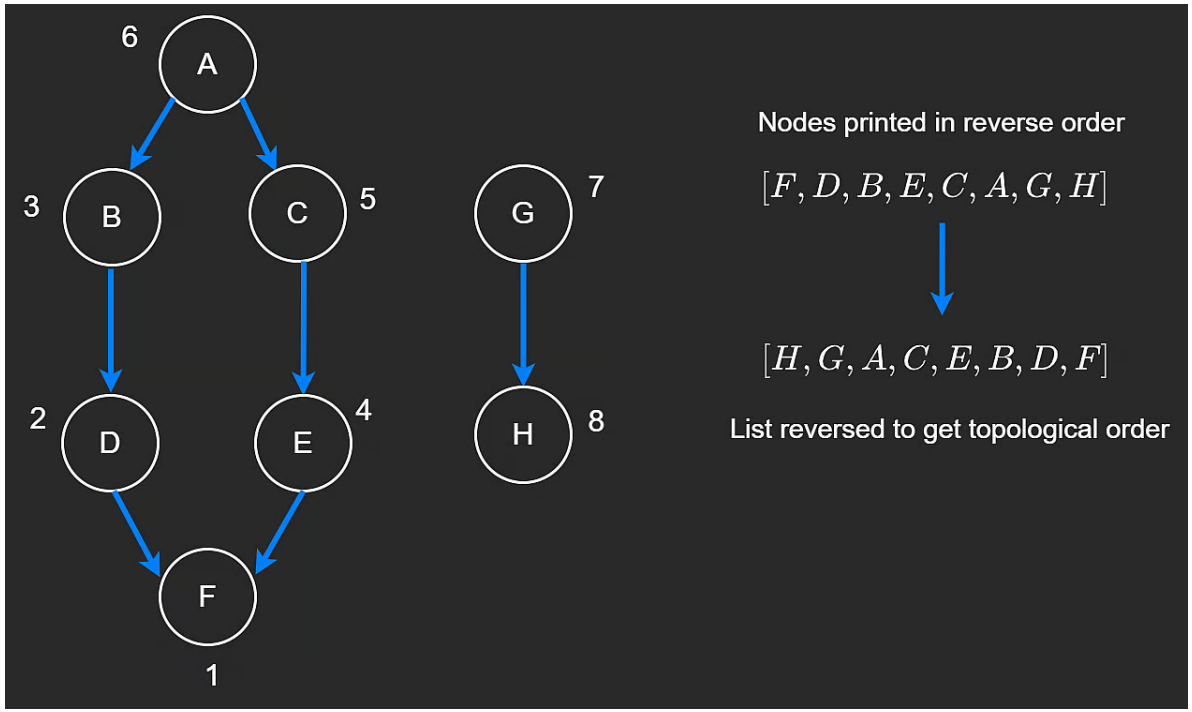
Addressing the Order Issue

- 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

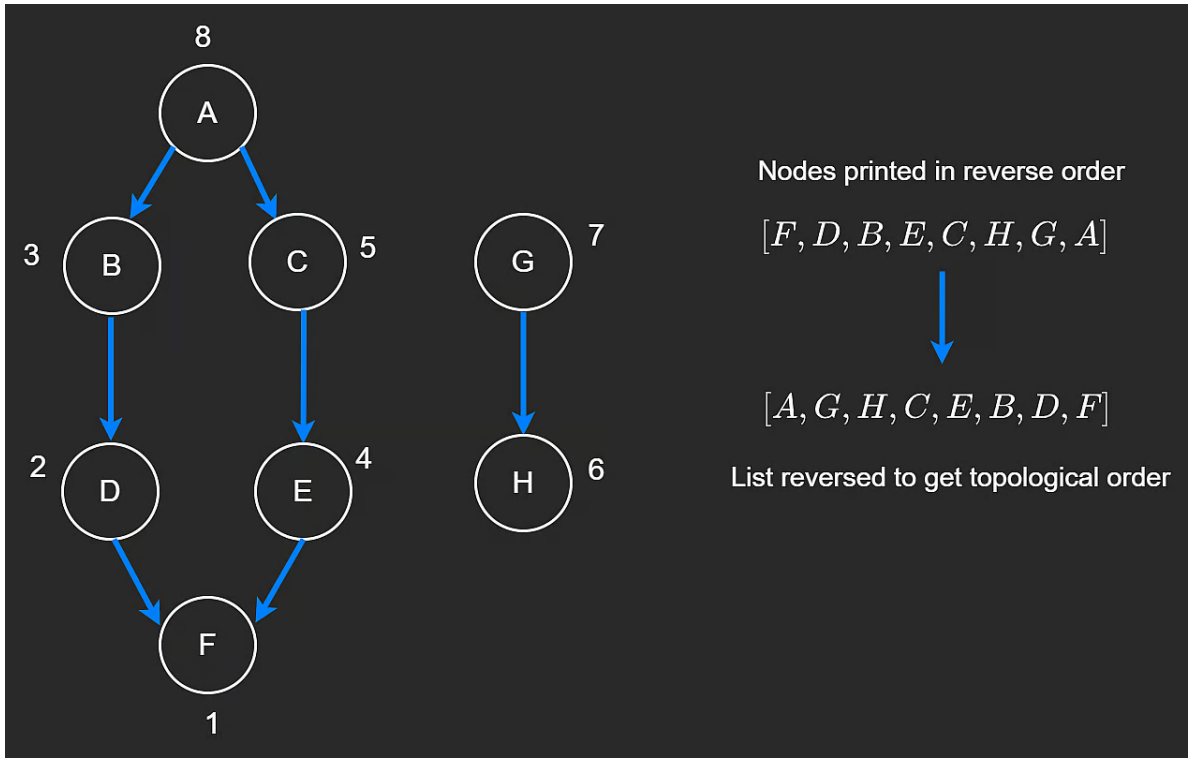
- Reverse the edges of the graph and run a post-order traversal. Recall that post-order traversal is: left, right, root. This will give us the correct topological order, which is shown below.
- The numbers next to the vertices represent the order in which they are visited.



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:

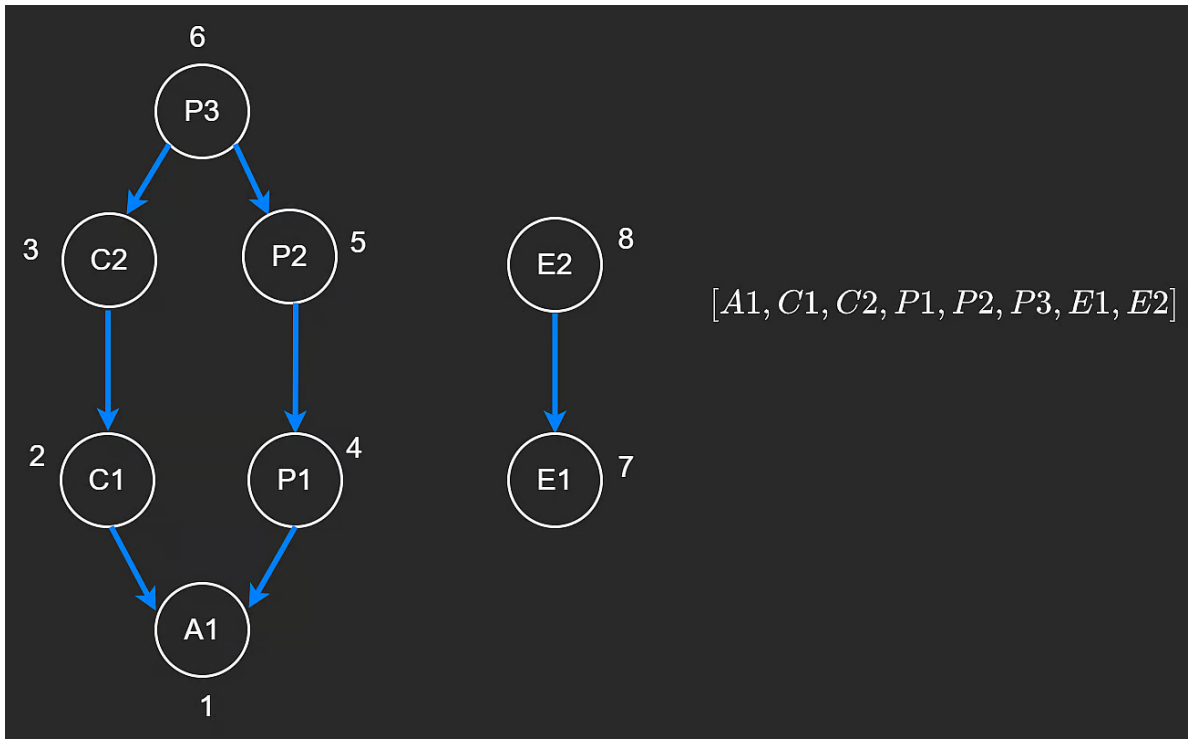
- Instead of initiating a Depth-First Search (DFS) from a known "head" or source vertex, the strategy involves executing DFS on every vertex within the graph.
- This comprehensive approach ensures that each component of the graph is explored in the sequence they are encountered or listed, regardless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

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: $O(V + E)$.