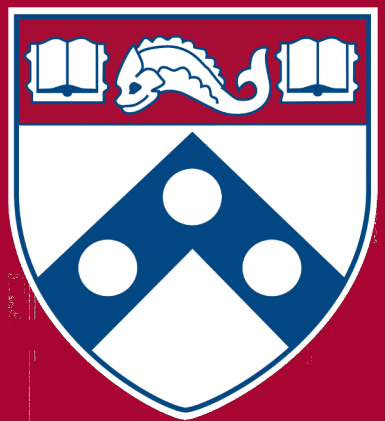
Graphs: Topological Sort, Shortest paths problems

CIT594



Topological Sort

- Goal: Find an acceptable order for processing subtasks

Example

- Problem: We want to write a program that automatically builds an online textbook from a collection of tutorials.
- We need to organize the tutorials so that given two tutorials A and B, if a A is a prerequisite for B, then A should be added and listed in the online textbook before B
- Our program needs to access and list the tutorials in a specific order
- Topological sort allows us to do just that

Example - OpenDSA

```
.. avmetadata::
   :author: Cliff Shaffer
   :requires: graph terminology
   :satisfies: graph implementation
   :topic: Graphs
```

1

Graph Implementations

We next turn to the problem of implementing a general-purpose :term:`graph` class. There are two traditional approaches to representing graphs: The :term:`adjacency matrix` and the :term:`adjacency list`. In this module we will show actual implementations for each approach. We will begin with an interface defining an ADT for graphs that a given implementation must meet.

```
.. codeinclude:: Graphs/Graph
   :tag: GraphADT
```

Chapter 19 Graphs

Table of content / list o
tutorials

```
.. avmetadata::
   :author: Cliff Shaffer
   :requires: graph implementation
   :satisfies: graph traversal
   :topic: Graphs
```

2

Graph Traversals

Graph Traversals

Many graph applications need to visit the vertices of a graph in some specific order based on the graph's topology. This is known as a graph :term:`traversal` and is similar in concept to a :ref:`tree traversal <BinaryTreeTraversal>`. Recall that tree traversals visit every node exactly once, in some specified order such as preorder, inorder, or postorder. Multiple tree traversals exist because various applications require the nodes to be visited in a particular order. For example, to print a BST's nodes in ascending order requires an inorder traversal as opposed to some other traversal. Standard graph traversal orders also exist. Each is appropriate for solving certain problems. For example, many problems in artificial intelligence programming are modeled using graphs. The problem domain might consist of a large collection of states, with connections between various pairs of states. Solving this sort of problem requires getting from a specified start state to a specified goal state by moving between states only through the connections. Typically, the start and goal states are not directly connected. To solve this problem, the vertices of the graph must be searched in some organized manner.

Graph traversal algorithms typically begin with a start vertex and attempt to visit the remaining vertices from there. Graph traversals

```
.. avmetadata::
   :author: Cliff Shaffer
   :requires: graph traversal
   :topic: Graphs
```

3/4

Topological Sort

Topological Sort

Assume that we need to schedule a series of tasks, such as classes or construction jobs, where we cannot start one task until after its prerequisites are completed. We wish to organize the tasks into a linear order that allows us to complete them one at a time without violating any prerequisites. We can model the problem using a DAG. The graph is directed because one task is a prerequisite of another -- the vertices have a directed relationship. It is acyclic because a cycle would indicate a conflicting series of prerequisites that could not be completed without violating at least one prerequisite. The process of laying out the vertices of a DAG in a linear order to meet the prerequisite rules is called a :term:`topological sort`.

```
.. avmetadata::
   :author: Cliff Shaffer
   :requires: graph traversal
   :satisfies: graph shortest path
   :topic: Graphs
```

3/4

Shortest-Paths Problems

Shortest-Paths Problems

On a road map, a road connecting two towns is typically labeled with its distance. We can model a road network as a directed graph whose edges are labeled with real numbers. These numbers represent the distance (or other cost metric, such as travel time) between two vertices. These labels may be called :term:`weights <weight>`, :term:`costs <cost>`, or :term:`distances <distance>`, depending on the application. Given such a graph, a typical problem is to find the total length of the shortest path between two specified vertices. This is not a trivial problem, because the shortest path may not be along the edge (if any) connecting two vertices, but rather may be along a path involving one or more intermediate vertices.

Topological Sort

- The process of laying out the *vertices* of a ***DAG*** in a *linear order* such that no vertex *A* in the order is preceded by a vertex that can be reached by a (directed) *path* from *A*
    - *DAG: directed, acyclic graph*
- The (directed) edges in the graph define a prerequisite system
- Goal: list the vertices in an order such that no prerequisites are violated

Topological Sort

- Depth-first implementation

  1. When a node (n) is visited, do nothing
  2. Recursively call topological sort on all the neighbors of n
  3. When the recursion pops back to n (after processing all its neighbors) add n to the (output) list of nodes

- This method produces a topological sort in reverse order
- It does not matter where the sort starts, but all vertices must be visited

Topological Sort

- Queue-based implementation

1. Count the number of edges that lead to each vertex
2. All vertices with no prerequisites are placed on the queue
3. Process the queue:

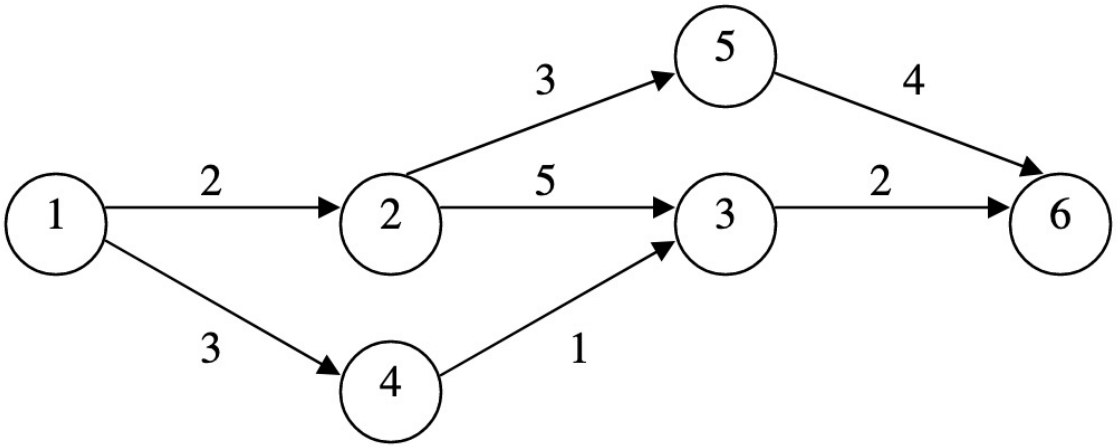
    1. When Vertex *v* is dequeued, it is printed, and all neighbors of *v* (all vertices that have *v* as a prerequisite) have their counts decremented by one

    2. Enqueue any neighbor whose count becomes zero

- If the queue is empty without printing all the vertices, then the graph is not a DAG
- This method produces a topological sort in order

Class Activity

- Given the following graph, return its topological sort using the depth-first implementation.
- Start at the vertex 4. Always select the vertex with the smallest label at each step



Shortest-Paths Problems

- Goal: find the total length of the shortest path between two specified vertices

Shortest-Paths Problems

- We can model a road or a computer network as a directed graph
- Edges are labeled with numbers representing the distance (or other cost metrics, such as travel time) between two vertices

Single-source Shortest-Paths Problems

- Given a *graph* with *weights* or distances on the *edges*, and a designated start *vertex s*, find the shortest path from *s* to every other vertex in the graph
- If the graph is unweighted (or all edges have the same cost) then BFS can be used
- If the graph is weighted, we need another solution: *Dijkstra's algorithm*

Dijkstra's algorithm

- Idea: process the vertices in a fixed order
- We process the vertices in order of distance from the start vertex (*S)*
- Assume that we have processed in order of distance from *S* to the first *i*−1 vertices that are closest to *S*; call this set of vertices **N**, we are now processing the i^{th} closest vertex; call it *X:*
    - The shortest path from *S* to *X* is the minimum overall paths that go from *S* to *U*, then have an edge from *U* to *X*, where *U* is some vertex in **N**

Dijkstra's algorithm

```
// Compute shortest path distances from s, store them in D
static void Dijkstra(Graph G, int s, int[] D) {
  for (int i=0; i<G.nodeCount(); i++)    // Initialize
    D[i] = INFINITY;
  D[s] = 0;
  for (int i=0; i<G.nodeCount(); i++) {  // Process the vertices
    int v = minVertex(G, D);      // Find next-closest vertex
    G.setValue(v, VISITED);
    if (D[v] == INFINITY) return; // Unreachable
    int[] nList = G.neighbors(v);
    for (int j=0; j<nList.length; j++) {
      int w = nList[j];
      if (D[w] > (D[v] + G.weight(v, w)))
        D[w] = D[v] + G.weight(v, w);
    }
  }
}
```

Dijkstra's algorithm

- Runtime analysis:
  - **$O(|V^2|)$** if we use a linear DS to find the minimum distance. Appropriate when the graph is dense
  - **$O((|V|+|E|)\log|E|)$** if we use a priority queue to find the minimum distance. Appropriate when the graph is sparse