Graphs: Topological Sort, Shortest paths problems

CIT594

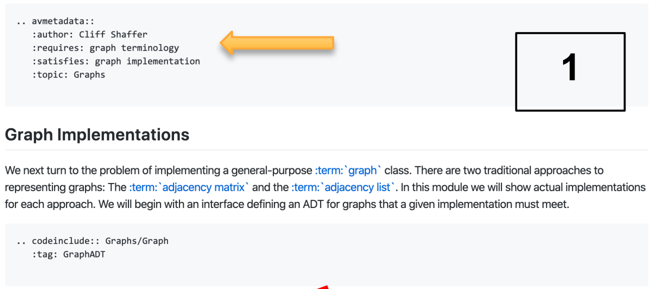
Topological Sort

- 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

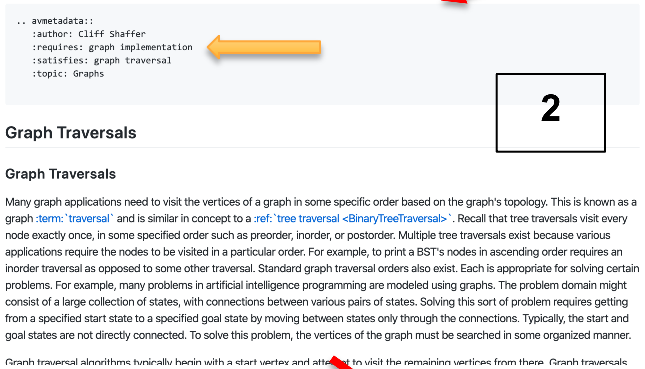


```
.. avmetadata::
   :author: Cliff Shaffer
   :requires: graph terminology
   :satisfies: graph implementation
   :topic: Graphs
```

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
```



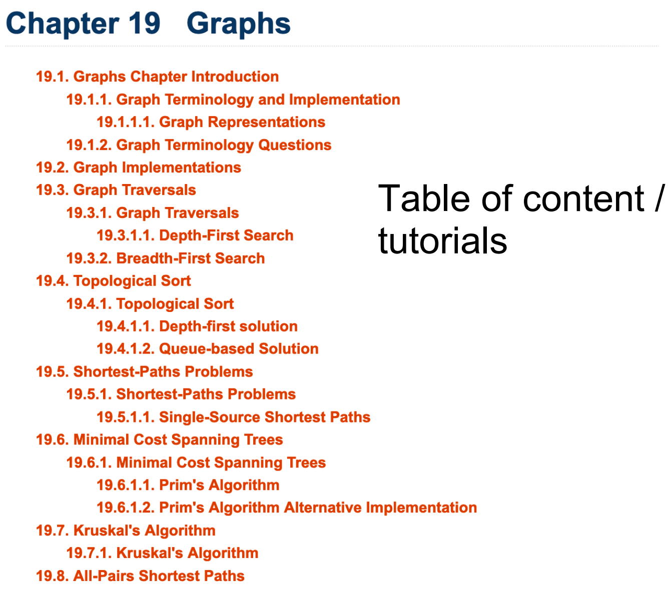
```
.. avmetadata::
   :author: Cliff Shaffer
   :requires: graph implementation
   :satisfies: graph traversal
   :topic: Graphs
```

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.

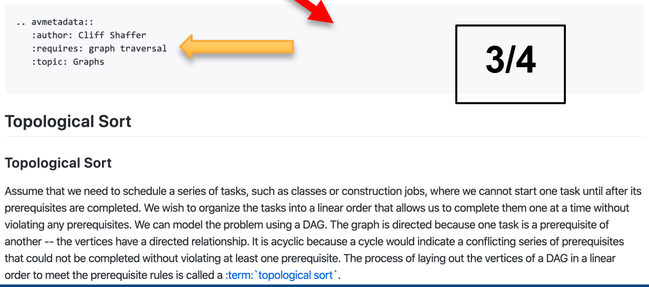
Graph traversal algorithms typically begin with a start vertex and atte[illegible]t to visit the remaining vertices from there. Graph traversals



Chapter 19 Graphs

19.1. Graphs Chapter Introduction
19.1.1. Graph Terminology and Implementation
19.1.1.1. Graph Representations
19.1.2. Graph Terminology Questions
19.2. Graph Implementations
19.3. Graph Traversals
19.3.1. Graph Traversals
19.3.1.1. Depth-First Search
19.3.2. Breadth-First Search
19.4. Topological Sort
19.4.1. Topological Sort
19.4.1.1. Depth-first solution
19.4.1.2. Queue-based Solution
19.5. Shortest-Paths Problems
19.5.1. Shortest-Paths Problems
19.5.1.1. Single-Source Shortest Paths
19.6. Minimal Cost Spanning Trees
19.6.1. Minimal Cost Spanning Trees
19.6.1.1. Prim's Algorithm
19.6.1.2. Prim's Algorithm Alternative Implementation
19.7. Kruskal's Algorithm
19.7.1. Kruskal's Algorithm
19.8. All-Pairs Shortest Paths

Table of content / list o
tutorials

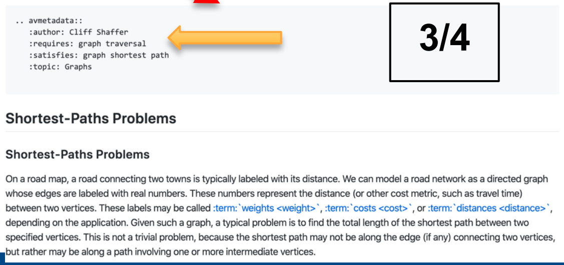


```
.. avmetadata::
   :author: Cliff Shaffer
   :requires: graph traversal
   :topic: Graphs
```

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
```

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:

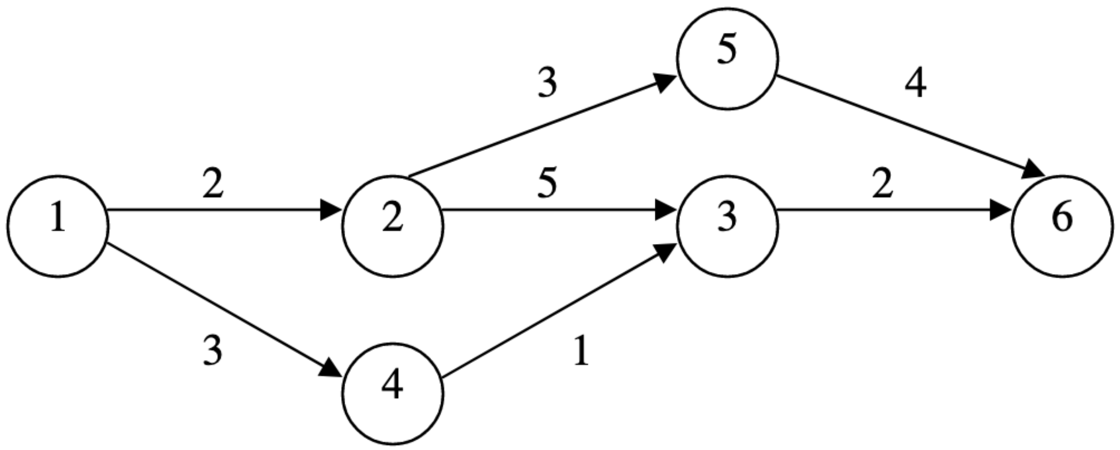
    1. When Vertex *v* is dequeued, it is printed, and all neighbors of *v* (all vertices that have *v* as a prerequisite) have their counts decremented by one

    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

- 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

Here Comes the Java

- Need a way of storing additional info per vertex
- Could implement with some additional tables:
  - HashMap<Vertex, Vertex> predecessor
  - HashMap<Vertex, Double> distance
- Could instead just create a wrapper class
  - PathVertexInfo, with fields:
    - Vertex vertex;
    - double distance;
    - Vertex predecessor;

Here Comes the Java

- In this implementation, Dijkstra’s returns a Map associating the Vertex with its PathVertexInfo
- Need a way to reconstruct the path from the Map!
- Exercise:

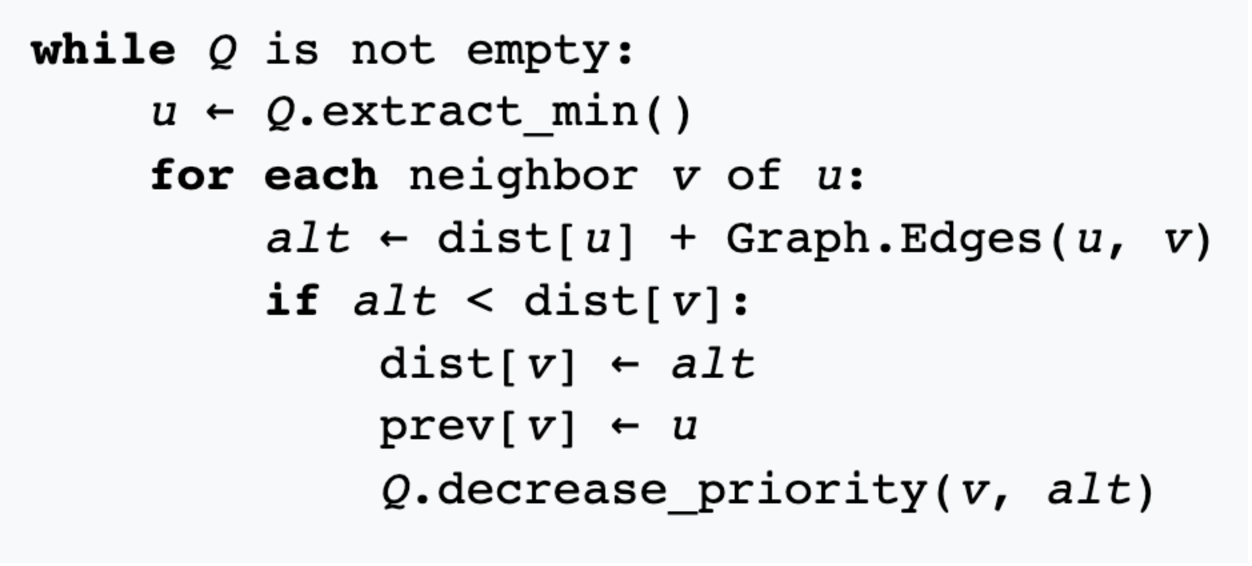
```
String getShortestPath(Vertex startVertex, Vertex endVertex,
                        HashMap<Vertex, PathVertexInfo> infoMap)
```

Dijkstra's algorithm: USEFUL FOR HW 7!!!

```java
// 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);
    }
  }
}
```

Runtime Revisited

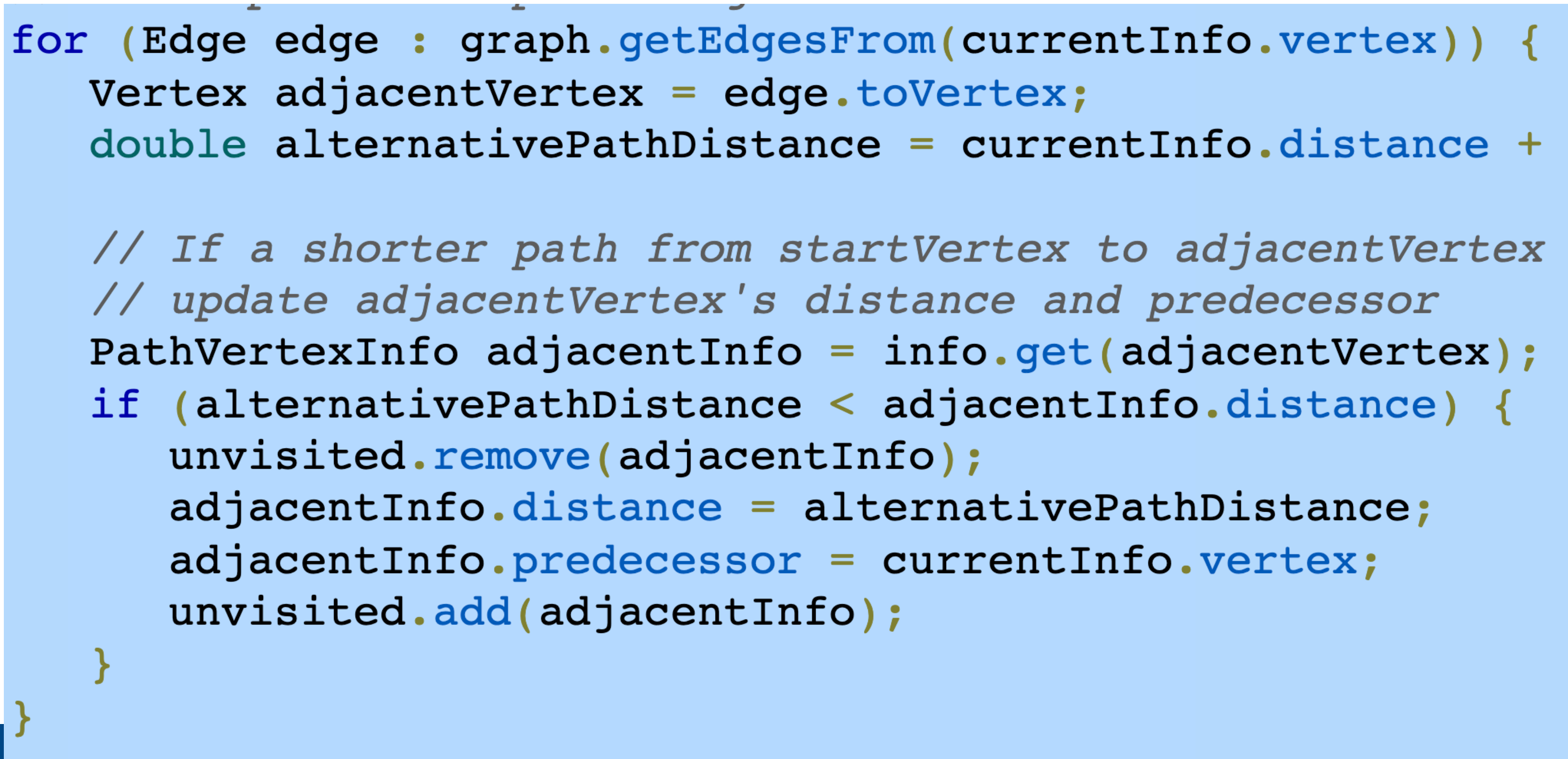
- Runtime analysis:
  - **$O(|V^2|)$** if we use a linear DS to find the minimum distance. Appropriate when the graph is dense
  - **$O((|V|+|E|)\log|E|)$** if we use a priority queue to find the minimum distance. Appropriate when the graph is sparse
    - ??????????????????????????????????



```
while Q is not empty:
    u ← Q.extract_min()
    for each neighbor v of u:
        alt ← dist[u] + Graph.Edges(u, v)
        if alt < dist[v]:
            dist[v] ← alt
            prev[v] ← u
            Q.decrease_priority(v, alt)
```

← **Fantasy**

Reality vvv



```
for (Edge edge : graph.getEdgesFrom(currentInfo.vertex)) {
    Vertex adjacentVertex = edge.toVertex;
    double alternativePathDistance = currentInfo.distance +

    // If a shorter path from startVertex to adjacentVertex
    // update adjacentVertex's distance and predecessor
    PathVertexInfo adjacentInfo = info.get(adjacentVertex);
    if (alternativePathDistance < adjacentInfo.distance) {
        unvisited.remove(adjacentInfo);
        adjacentInfo.distance = alternativePathDistance;
        adjacentInfo.predecessor = currentInfo.vertex;
        unvisited.add(adjacentInfo);
    }
}
```

Runtime Revisited

- Runtime analysis:
  - **O(|V²|)** if we use a linear DS to find the minimum distance. Appropriate when the graph is dense
  - **O((|V|+|E|)log|V|)** if we use a priority queue to find the minimum distance. Appropriate when the graph is sparse **and when the priority queue actually lets you update priority in log(V) time!!**

Graphs: Minimal Cost Spanning Trees

CIT5940

Minimal Cost Spanning Trees

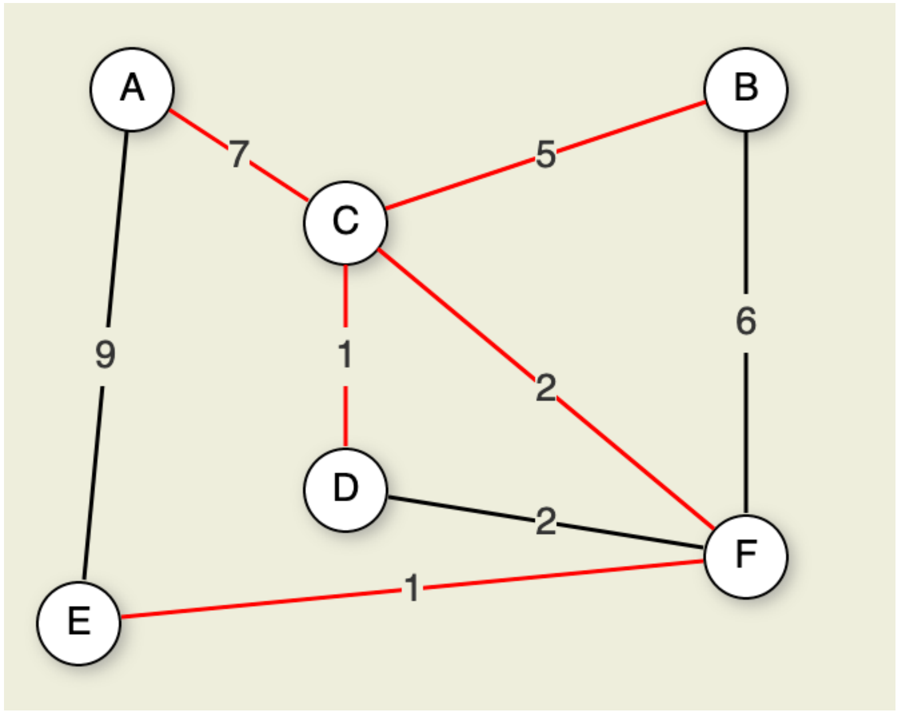
- Goal: Find the subset of the (weighted) graph's edges that maintains the connectivity of the graph while having the lowest total cost

Minimal Cost Spanning Trees (MCST)

- The cost is defined by the sum of the *weights* of the edges in the MCST
- The MCST would never have a *cycle (tree)*
    - An edge can be removed from the cycle and still preserve connectivity
- Algorithms:
    - ***Prim's algorithm* (not covering here, but included in slides)**
    - ***Kruskal's algorithm***

Minimal Cost Spanning Trees (MCST)

- Applications:
  - Connected network design: find the least amount of cables need to connect cities, offices, etc.



MCST: Prim’s Algorithm

1. Start with any Vertex *N* in the graph, setting the MCST to be *N* initially
2. Pick the least-cost edge connected to *N*. This edge connects *N* to another vertex; call this *M*. Add Vertex *M* and Edge (*N*,*M*) to the MCST
3. Pick the least-cost edge coming from either *N* or *M* to any other vertex in the graph. Add this edge and the new vertex it reaches to the MCST
4. Continue the process (2-3), at each step expanding the MCST by selecting the least-cost edge from a vertex currently in the MCST to a vertex not currently in the MCST

MCST: Prim’s Algorithm

- Similar to Dijkstra’s algorithm
- **Theorem:** Prim's algorithm produces a minimum-cost spanning tree

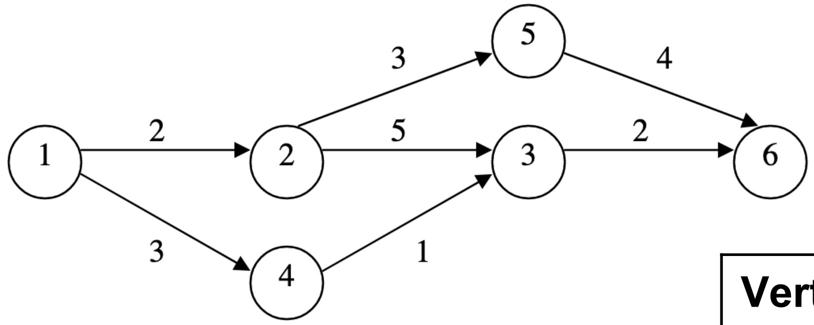
```
// Compute shortest distances to the MCST, store them in D.
// V[i] will hold the index for the vertex that is i's parent in the MCST
void Prim(Graph G, int s, int[] D, int[] V) {
  for (int i=0; i<G.nodeCount(); i++)    // Initialize
    D[i] = INFINITY;
  D[s] = 0;
  for (int i=0; i<G.nodeCount(); i++) {  // Process the vertices
    int v = minVertex(G, D);     // Find next-closest vertex
    G.setValue(v, VISITED);
    if (D[v] == INFINITY) return; // Unreachable
    if (v != s) AddEdgetoMST(V[v], v);
    int[] nList = G.neighbors(v);
    for (int j=0; j<nList.length; j++) {
      int w = nList[j];
      if (D[w] > G.weight(v, w)) {
        D[w] = G.weight(v, w);
        V[w] = v;
      }
    }
  }
}
```

MCST: Prim’s Algorithm

```
// Compute shortest distances to the MCST, store them in D.
// V[i] will hold the index for the vertex that is i's parent in the MCST
void Prim(Graph G, int s, int[] D, int[] V) {
  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
    if (v != s) AddEdgetoMST(V[v], v);
    int[] nList = G.neighbors(v);
    for (int j=0; j<nList.length; j++) {
      int w = nList[j];
      if (D[w] > G.weight(v, w)) {
        D[w] = G.weight(v, w);
        V[w] = v;
      }
    }
  }
}
```

MCST: Prim's Algorithm

- Class Activity: Given the following graph. Build its MCST using Prim's algorithm starting at vertex 1. Indicate the distances and the order in which the vertices are processed.



List of vertices (in processing order)

____

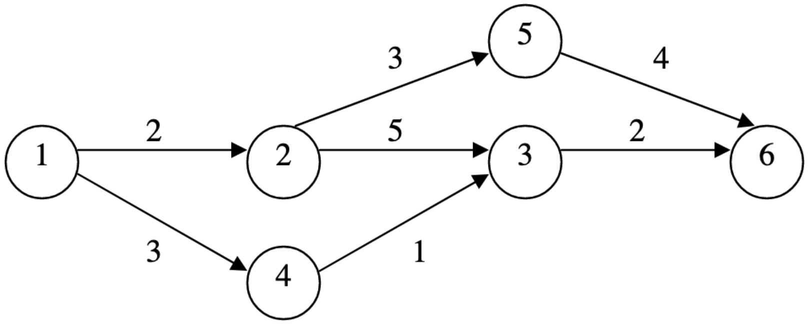
Verte x	Distance	From Vertex
1		
2		
3		
4		
5		
6		

Kruskal's Algorithm

1. Partition the set of vertices into |**V**| *disjoint sets* (each consisting of one vertex)
2. Process the edges in order of weight
    - An edge is added to the MCST, and two disjoint sets are combined, if the edge connects two vertices in different disjoint sets
    - This process is repeated until only one disjoint set remains

Kruskal's Algorithm

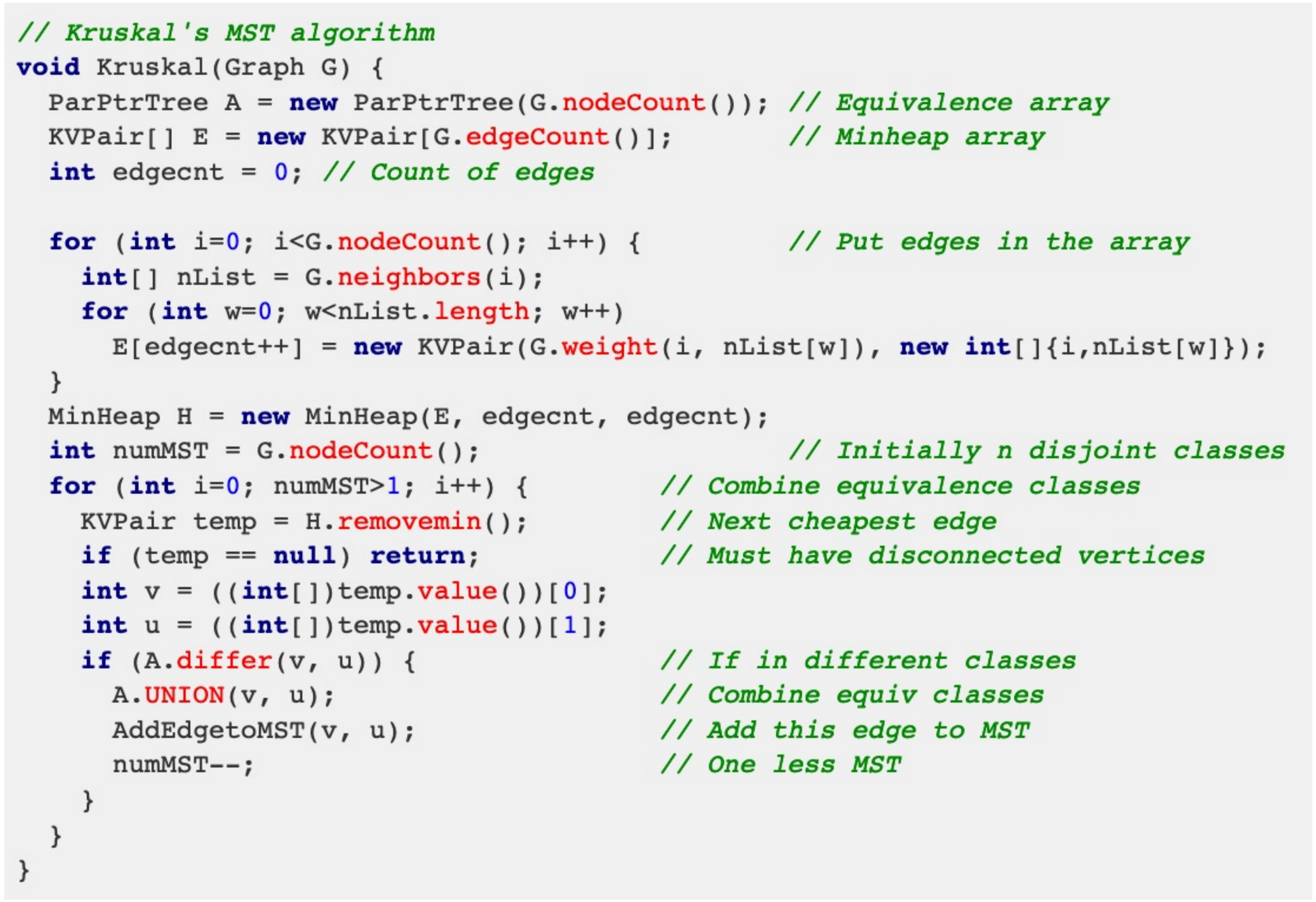
- **Class Activity**: Given the following graph. Build its MCST using Kruskal's algorithm Indicate the parent of each vertex and the order in which the edges are added to the MCST.



List of MCST edges (in order)

Vertex	Parent
1	
2	
3	
4	
5	
6	

Kruskal’s Algorithm (for



```
// Kruskal's MST algorithm
void Kruskal(Graph G) {
  ParPtrTree A = new ParPtrTree(G.nodeCount()); // Equivalence array
  KVPair[] E = new KVPair[G.edgeCount()];       // Minheap array
  int edgecnt = 0; // Count of edges

  for (int i=0; i<G.nodeCount(); i++) {         // Put edges in the array
    int[] nList = G.neighbors(i);
    for (int w=0; w<nList.length; w++)
      E[edgecnt++] = new KVPair(G.weight(i, nList[w]), new int[]{i,nList[w]});
  }
  MinHeap H = new MinHeap(E, edgecnt, edgecnt);
  int numMST = G.nodeCount();                    // Initially n disjoint classes
  for (int i=0; numMST>1; i++) {         // Combine equivalence classes
    KVPair temp = H.removemin();         // Next cheapest edge
    if (temp == null) return;            // Must have disconnected vertices
    int v = ((int[])temp.value())[0];
    int u = ((int[])temp.value())[1];
    if (A.differ(v, u)) {                // If in different classes
      A.UNION(v, u);                     // Combine equiv classes
      AddEdgetoMST(v, u);                // Add this edge to MST
      numMST--;                          // One less MST
    }
  }
}
```

- O(|E|log|E|) in the worst case
- O(|V|log|E|) in the average case

Kruskal’s Algorithm: Union/Find

- A process for maintaining a collection of disjoint sets.
- The ***FIND*** operation determines which disjoint set a given object resides in
- The ***UNION*** operation combines two disjoint sets when it is determined that they are members of the same *equivalence class* under some *equivalence relation*.

Kruskal’s Algorithm: Union/Find

- Equivalence relation: “*is connected to*” (ref)
- A relation *R* is an equivalence relation on set **S** if it is *reflexive*, *symmetric*, and *transitive*
- Each subset/tree represents an equivalence class (connected components)

Kruskal’s Algorithm: Union/Find

- Parent pointer representation (PPR): a node implementation (for a Tree) where each node stores only a pointer to its parent, rather than to its children.
- PPR makes it easy to go up the tree toward the root, but not down the tree toward the leaves
- PPR is often used to maintain a collection of *disjoint sets*.
- PPR implements a parent pointer tree

Kruskal’s Algorithm: Union/Find

```java
// General Tree implementation for UNION/FIND
public class ParPtrTree {
  private int[] array;     // Node array

  ParPtrTree(int size) {
    array = new int[size]; // Create node array
    for (int i=0; i<size; i++)
      array[i] = -1;       // Each node is its own root to start
  }

  // Merge two subtrees if they are different
  public void UNION(int a, int b) {
    int root1 = FIND(a);     // Find root of node a
    int root2 = FIND(b);     // Find root of node b
    if (root1 != root2)          // Merge two trees
      array[root1] = root2;
  }

  // Return the root of curr's tree
  public int FIND(int curr) {
    while (array[curr] != -1)
      curr = array[curr];
    return curr; // Now at root
  }
}
```