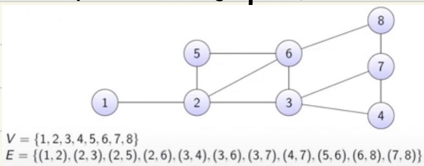
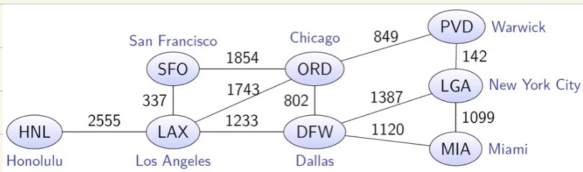
Video 1. Definitions

V-set of nodes, vertices
E-collection of pairs of vertices, edges



undirected graph - edges are unordered pairs
directed graph - edges are ordered pairs



end vertices - the two vertices joined by an edge
-> origin/destination (direc.)
adjacent vertices - edge-conn
edges incident on a vertex - edges' end point include the vertex
→ incoming/outcoming edges (directed)
parallel edges - edges with common end vertices
self-loop - edge with the same end vertices
degree of a vertex [deg(v)] - number of incident edges of a vertex
→ indegree/outdegree (directed)
simple graph — graph without parallel edges and self-loops

path - sequence of alternating vertices and edges
→ begins with a vertex
→ ends with a vertex
→ each edge - incident to its predecessor and successor
simple path - path with all vertices distinct
directed path - path with directed edges, traversed along direction
cycle - path starting with and ending in the same vertex, incl. at least one edge
simple cycle - distinct vertices and edges
directed cycle - directed edges, traversed along their direction

subgraph of a graph G
- → vertices of subgraph are subset of the vertices of G
- → edges of subgraph are subset of the edges of G

spanning subgraph – contains all vertices of the graph
vertex reaching another vertex – exist (directed) path
connected graph – paths between every pair of vertices
strongly connected directed graph – directed paths between V
connected components – the maximal connected subgraphs

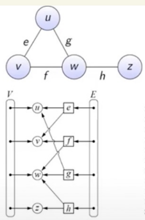
Video 2. Data Structures

```java
interface Vertex<V> {
  public V getElement();
}

interface Edge<E> {
  public E getElement();
}
```

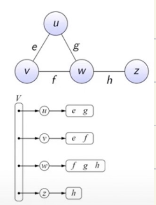
without parallel edges

```java
interface Graph<V,E> {
  public int numVertices();
  public Iterable<Vertex<V>> vertices();

  public int numEdges();
  public Iterable<Edge<E>> edges();

  public Edge<E> getEdge(Vertex<V> u, Vertex<V> v);
  public Vertex<V> opposite(Vertex<V> v, Edge<E> e);

  // for an undirected graph, both return the same
  public Iterable<Edge<E>> outgoingEdges(Vertex<V> v);
  public Iterable<Edge<E>> incomingEdges(Vertex<V> v);

  public Vertex<V> insertVertex(V x);
  public Edge<E> insertEdge(Vertex<V> u, Vertex<V> v, E x);
  public void removeVertex(Vertex<V> v);
  public void removeEdge(Edge<E> e);
}
```

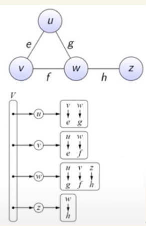
```java
class EdgeList<V,E> implements Graph<V,E> {
  class InnerVertex<V> implements Vertex<V> {
    private V elem;
    private Position <InnerVertex<V>> pos;
  }

  class InnerEdge<V,E> implements Edge<E> {
    private E elem;
    private InnerVertex<V> begin;
    private InnerVertex<V> end;
    private Position <InnerEdge<V,E>> pos;
  }

  private LinkedPositionalList <InnerVertex<V>> vertices;
  private LinkedPositionalList <InnerEdge<V,E>> edges;

  // ...
}
```



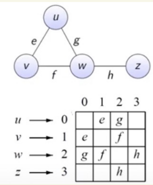

```java
class AdjacencyList<V,E> implements Graph<V,E> {
  class InnerVertex<V> implements Vertex<V> {
    private V elem;
    private Position <InnerVertex<V>> pos;
    private LinkedPositionalList
      <InnerEdge<V,E>> outgoing, incoming;
  }

  class InnerEdge<V,E> implements Edge<E> {
    private E elem;
    private InnerVertex<V> begin;
    private InnerVertex<V> end;
    private Position <InnerEdge<V,E>> pos;
  }

  private LinkedPositionalList <InnerVertex<V>> vertices;
  private LinkedPositionalList <InnerEdge<V,E>> edges;

  // ...
}
```




```java
class AdjacencyMap<V,E> implements Graph<V,E> {
  class InnerVertex<V> implements Vertex<V> {
    private V elem;
    private Position <InnerVertex<V>> pos;
    private Map
      <Vertex<V>,InnerEdge<V,E>> outgoing, incoming;
  }

  class InnerEdge<V,E> implements Edge<E> {
    private E elem;
    private InnerVertex<V> begin;
    private InnerVertex<V> end;
    private Position <InnerEdge<V,E>> pos;
  }

  private LinkedPositionalList <InnerVertex<V>> vertices;
  private LinkedPositionalList <InnerEdge<V,E>> edges;

  // ...
}
```




```java
class AdjacencyMatrix<V,E> implements Graph<V,E> {
  // ...

  private InnerEdge<V,E>[][] matrix;

  // ...
}
```

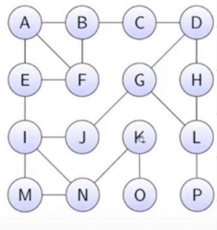


Video 3. Traversals

traversal - exploring graph by examining vertices and edges

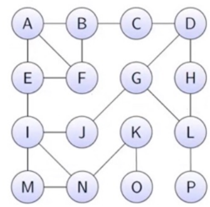
depth-first search

```
public static <V,E> void depthfirst(Graph<V,E> g,
    Vertex<V> u, Set<Vertex<V>> known) {
  visit(u);
  known.add(u); // u has been discovered
  for (Edge<E> e : g.outgoingEdges(u)) {
    Vertex<V> v = g.opposite(u, e);
    // check whether v has already been discovered
    if (!known.contains(v)) {
      // recursively explore from v
      depthfirst(g, v, known);
    }
  }
}
```



breadth-first search

```
public static <V,E> void breadthfirst(Graph<V,E> g,
    Vertex<V> s) {
  // vertices that we have already visited
  Set<Vertex<V>> known = new Set();
  // breadth first queue
  Queue<Position<E>> q = new Queue();
  q.enqueue(s);
  known.add(s);

  while (!q.isEmpty()) {
    Vertex<V> u = q.dequeue();
    visit(u);
    for (Edge<E> e : g.outgoingEdges(u)) {
      Vertex<V> v = g.opposite(u, e);
      if (!known.contains(v))
        q.enqueue(v);
        known.add(v);
    }
  }
}
```



Assume we are given a graph $G = (V, E)$ with n vertices and m edges: DFS

- n calls to add
- n calls to outgoingEdges
- $\sum_{v \in V} \deg(v)$ calls to opposite
- $\sum_{v \in V} \deg(v)$ calls to contains

```
public static <V,E> void depthfirst(Graph<V,E> g,
    Vertex<V> u, Set<Vertex<V>> known) {
  known.add(u);
  for (Edge<E> e : g.outgoingEdges(u)) {
    Vertex<V> v = g.opposite(u, e);
    if (!known.contains(v))
      depthfirst(g, v, known);
  }
}
```

outgoingEdges -> $O(\deg(v))$

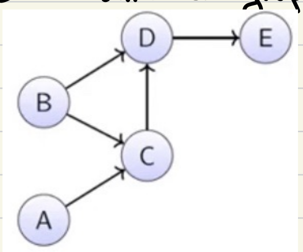
opposite -> $O(1)$

$\sum_{v \in V} \deg(v) = 2m$

=> total time: $O(n+m)$

Video 4. DAG - Directed Acyclic Graphs

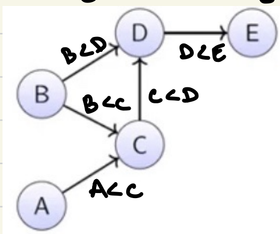
DAG - directed graph without directed cycles



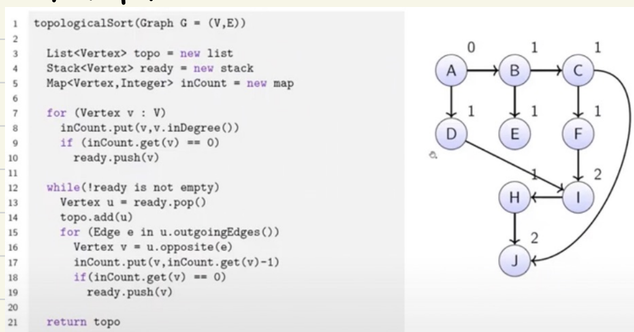
Video 5. Topological sorting

$G=(V,E)$ $v_1 \dots v_n \in V$

topological ordering: $\forall (v_i, v_j) \in E \rightarrow i<j$



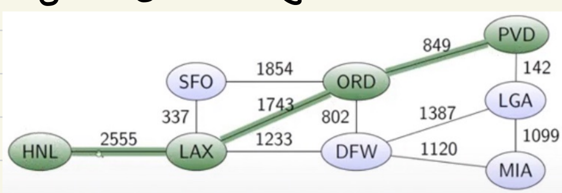
possible topological orderings:
- → A, B, C, D, E
- → B, A, C, D, E



```
topologicalSort(Graph G = (V,E))

  List<Vertex> topo = new list
  Stack<Vertex> ready = new stack
  Map<Vertex,Integer> inCount = new map

  for (Vertex v : V)
    inCount.put(v,v.inDegree())
    if (inCount.get(v) == 0)
      ready.push(v)

  while(!ready is not empty)
    Vertex u = ready.pop()
    topo.add(u)
    for (Edge e in u.outgoingEdges())
      Vertex v = u.opposite(e)
      inCount.put(v,inCount.get(v)-1)
      if(inCount.get(v) == 0)
        ready.push(v)

  return topo
```


Video 6. Shortest paths

weighted graph – edges have numerical value (weight)



shortest path – path between two vertices with minimal weight

HNL → PVD = 5147

Video 7. Dijkstra's algorithm

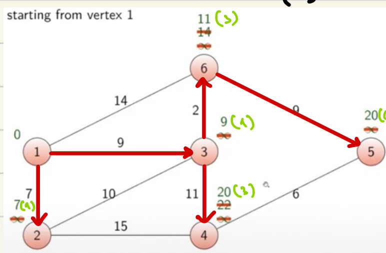
idea: $\forall v \in V \rightarrow$ boolean visited(v), label $D(v) \rightarrow$ weight of best path

initially: $D[s]=0$, $D[v]=\infty$ for $v \neq s$

repeat: mark unvisited vertex with at least D[u]

D[v] for $\forall (u,v) \in E$ with weight(u,v):

$$D[v] = \begin{cases} D[u] + w(u,v) & \text{if } D[u] + w(u,v) < D[v] \\ D[v] & \text{otherwise} \end{cases}$$



BFS → queue with closest vertex in terms of #edges

DA → adaptable priority queue with closest vertex in terms of weight to source

```java
/* The edges are integers that represent the weight. For simplicity, we assume that the vertices are integers
'0, ..., n-1', so that we can use them as indexes into an array. A version with hashmaps is in the book. */
public static int[] dijkstra(Graph<Integer,Integer> g, Vertex<Integer> src) {
  int[] res = new int[g.numVertices()];                 // the weight of the best path so far
  boolean[] visited = new boolean[g.numVertices()];     // Keep track of whether we have visited a vertex

  AdaptablePriorityQueue<Integer, Vertex<Integer>> pq =
    new HeapAdaptablePriorityQueue<>();                 // a PQ with weights as keys, and vertices as values
  Entry<Integer,Vertex<Integer>>[] pqTokens =
    new Entry<Integer,Vertex<Integer>>[g.numVertices()]; // the locator of each vertex in the PQ

  for (Vertex<Integer> v : g.vertices()) {
    res[v.getElement()] = v == src ? 0 : Integer.MAX VALUE;
    pqTokens[v] = pq.insert(res[v.getElement()], v);    // save entry for edge relaxation
  }

  while (!pq.isEmpty()) {
    Vertex<Integer> u = pq.removeMin().getValue();
    visited[u] = true;
    for (Edge<E> e : g.outgoingEdges(u)) {
      Vertex<Integer> v = g.opposite(u, e);
      if (!visited[v] && res[u] + e.getElement() < res[v]) {
        res[v] = res[u] + e.getElement();
        pq.replaceKey(pqTokens[v], res[v]);
      }
    }
  }
  return res;
}
```

$n \times$ removeMin $O(\log n)$

$n \times$ outgoingEdges $O(\deg(v))$

$\sum_{v \in V} \deg(v) \times$ opposite $O(1)$

$\sum_{v \in V} \deg(v) \times$ replaceKey $O(\log n)$

$O((n+m)\log n)$

Video 8. Minimum spanning trees

MST: $w(T) \triangleq \sum_{(u,v) \in T} w(u,v) \leftarrow$ minimize

Video 9. Kruskal's algorithm

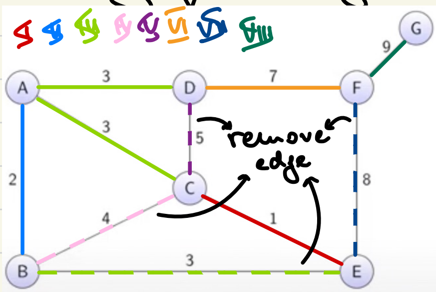
idea: partition of the vertices into clusters

initially: $\forall v \in V$ → single cluster

cluster = minimum spanning tree

pick edge with smallest weight (not in MST) → merge clusters; remove edge

Priority queue: Keys = weights Elements = edges




```
interface Partition<E> {
  // Makes a new cluster containing element e and returns its position
  public Position<E> makeCluster(E e);

  // Finds the cluster containing the element at position p and returns
  // the position of the cluster's leader.
  public Position<E> find(Position<E> p);

  // Merges the clusters containing the elements at positions p and q
  // (if both elements are in the same cluster, nothing happens)
  public void union(Position<E> p, Position<E> q);
}
```