INFS 519 – Fall 2015

Program Design and Data Structures

Lecture 12

Instructor: James Pope

Email: jpope8@gmu.edu

Today

- Review Last Class
  - Graphs / Motivation
  - Data Structure Representations
  - Basic graph searching algorithms: BFS and DFS
- Schedule
  - Directed/Edge weights Graphs
  - Shortest Path Algorithms
  - Minimum Spanning Tree
- Note: Planning to add “Algorithms” Sedgewick/Wayne as supplementary textbook.

Honor Code Policy Reminder

- Posting assignments to a website seeking assistance is a clear violation of the Honor Code.
  - Just because many websites cater to students seeking to solve problems does not make it OK
- Applies subsequently to this course
  - If you upload an assignment and/or solution to a website (including social media, e.g. Facebook), you violate the Honor Code. Even if you do this next year, it is still a violation.
- If a search with an assignment/solution is traced to you, I WILL forward you to the Honor Council.
  - Graduate students are held to higher standard!

Graphs

- Graph consists of the set of vertices and a set of edges that connect those vertices.
  - Denoted G(V,E)
  - Edges a.k.a. arcs, Vertices a.k.a. nodes
  - Cardinality is the number of elements in a set
    - |V| number of vertices
    - |E| number of edges
- Each edge is a pair v, w element of V
  - Can be weighted (“edge cost”)
  - Can have a direction

Graph Applications Overview

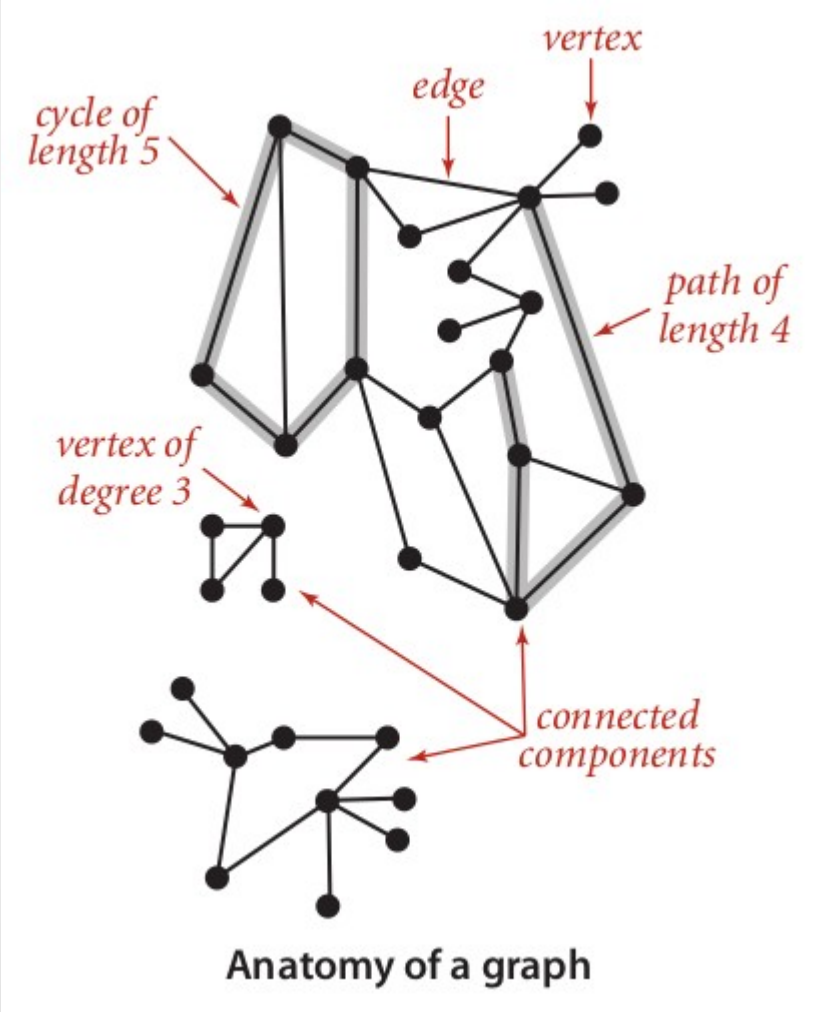
Sedgwick 4.1

Graph	Vertex	Edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	street intersection, airport	highway, airway route
internet	class C network	connection
game	board position	legal move
social	relationship person, actor	friendship, movie cast
neural network	neuron	synapse
protein	network protein	protein-protein interaction
chemical compound	molecule	bond

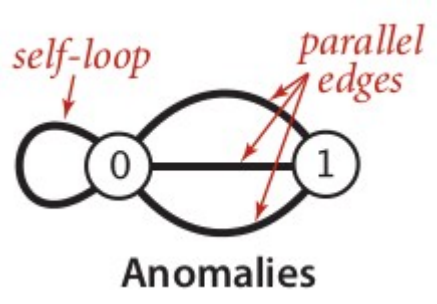
Graphs – Terminology 1

- Directed Graph (Digraph) – consists of edges with specified direction.
  - Ordered (from,to)
- Undirected Graph – consists of edges with no specific direction.
  - May be modeled as a directed graph with edge in both directions for each edge
- Path is a sequence of vertices connected by edges.
  - Path length number of edges on path
  - Weighted path length sum of edge weights

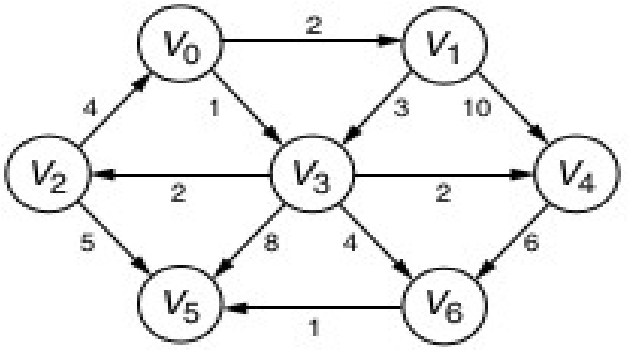
Graphs Examples



Anatomy of a graph

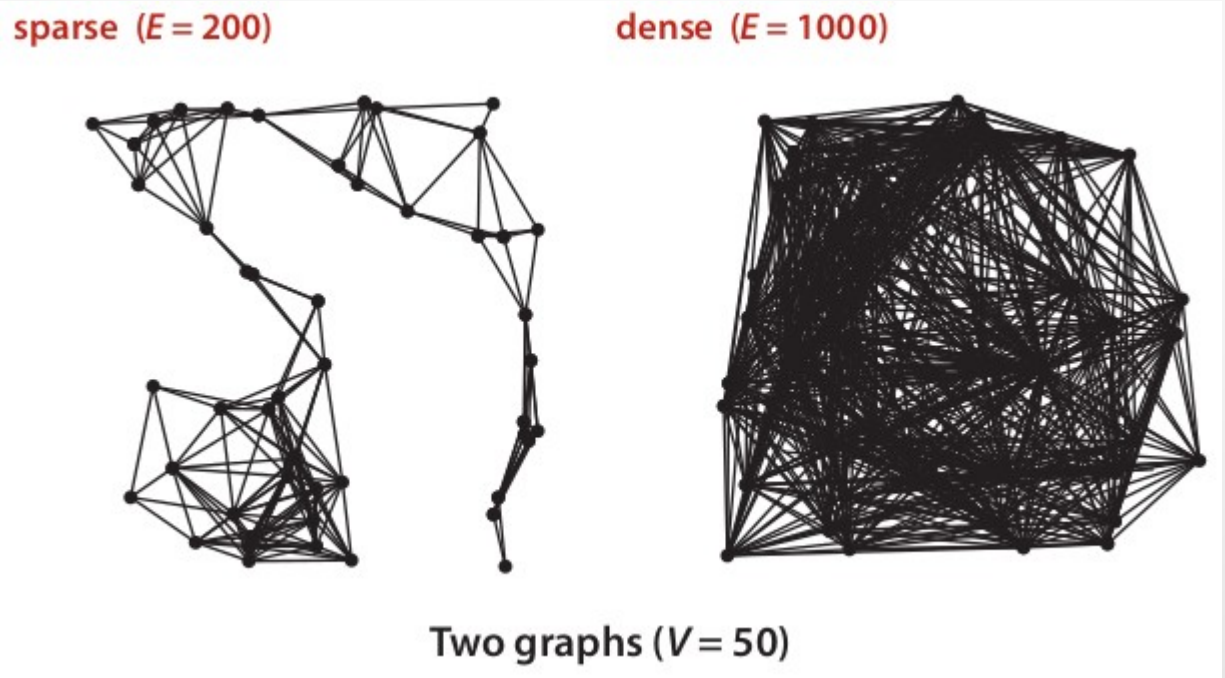


Anomalies



Graphs Density

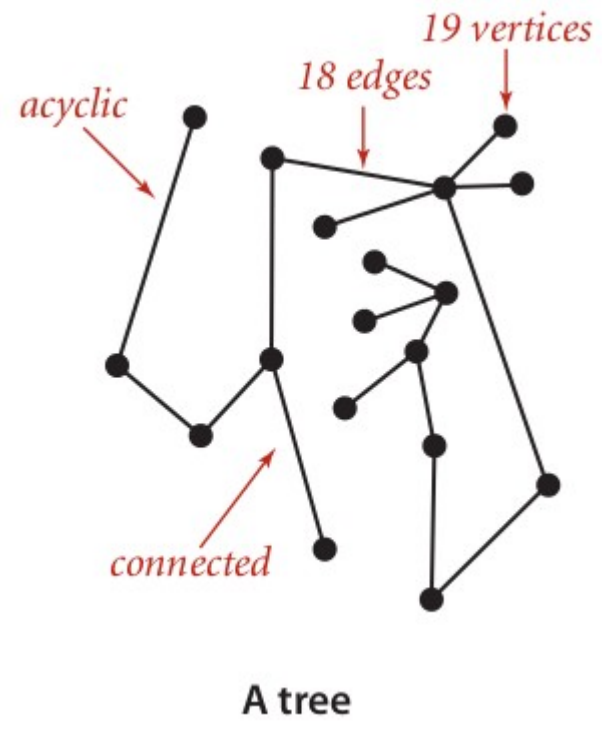
- Graph is dense if it has a large (i.e. |V| x |V|, specifically (V (V-1)) / 2) number of edges.
  - Dense roughly means E ~ V^2
  - Typical applications have sparse graphs



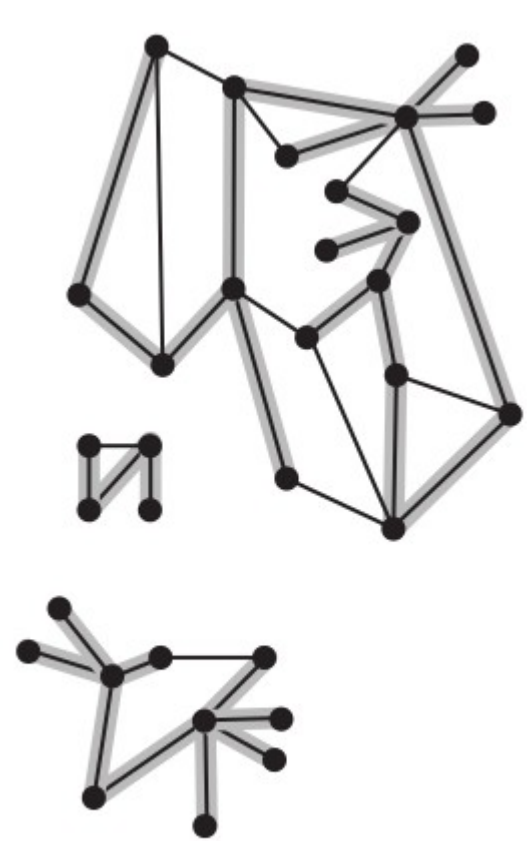
Trees

- Graph is connected is there is a path from every vertex to every other vertex.
- Graph that is not connected consists of a set of connected components
- Tree is an acyclic, connected graph
- Tree iff has the following properties
  - G has |V| – 1 edges and no cycles
  - G has |V| – 1 edges and is connected
  - G is connected and removing any edge disconnects
  - G is acyclic and adding any edge creates a cycle
  - Exactly one simple path connected each pair vertices

Tree Examples



A tree



A spanning forest

Graph Data Structure Representations 1/2

Sedgewick 4.1

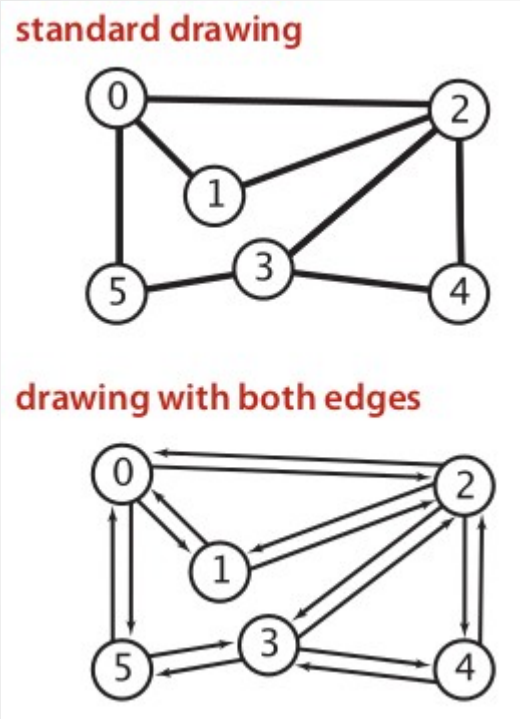
- Three approaches (there are others)
  - Edge List
  - Adjacency Matrix
  - Adjacency List
- There are pros and cons related to amount of memory for and performance of various algorithms for a given data representation. Common issues:
  - How much space is used?
  - Performance of adding an edge between v and w?
  - Performance checking whether v is adjacent to w?
  - Performance of iterating adjacent vertices to v?

Graph Data Structure Representations 1/2

Sedgewick 4.1

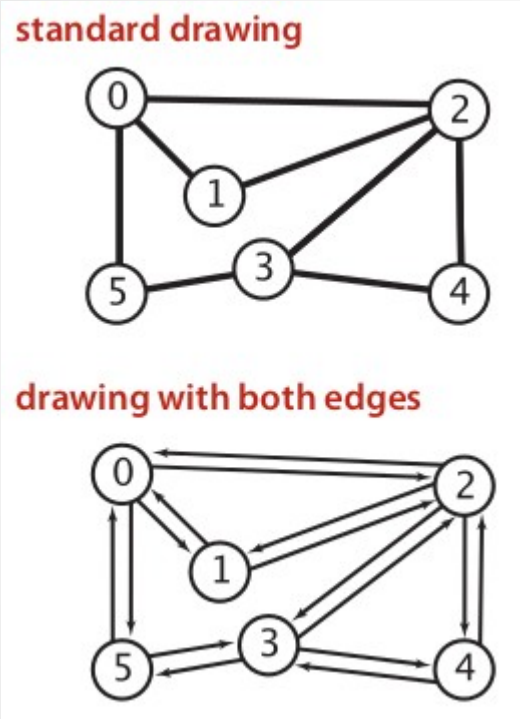
- Edge List is list of edges.
- Adjacency Matrix is a two-dimensional array in which the value at the intersection of two vertices record the edge weight (1 if unweighted), e.g. m[i][j] = 23
- Adjacency List is an array of linked list where the index represents the vertex and the list represents the adjacent vertices. The list node has the neighbor vertex id and edge weight.
- Note: Will not need/support deleting edges or adding/deleting vertices

Edge List Example



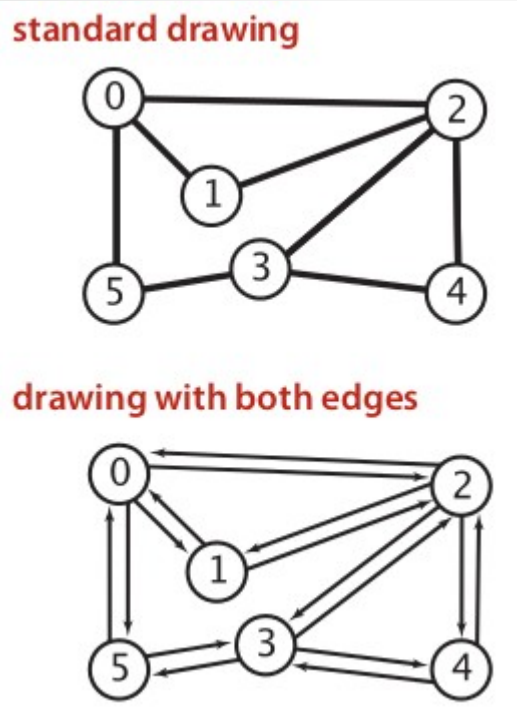
0 1	1 0
0 2	2 0
0 5	5 0
1 2	2 1
2 3	3 2
2 4	4 2
3 4	4 3
5 3	3 5

Adjacency Matrix Example



	0	1	2	3	4	5
0	0	1	1	0	0	1
1	1	0	1	0	0	0
2	1	1	0	1	1	0
3	0	0	1	0	1	1
4	0	0	1	1	0	0
5	1	0	0	1	0	0

Adjacency List Example



0->1,2,5

1->0,2

2->0,1,3,4

3->2,4,5

4->2,3

5->0,3

Graph Data Structure Order of growth

Sedgwick 4.1

- Rule of thumb: When dealing with a dense graph, use an adjacency matrix. Use adjacency list when dealing with sparse graphs. When you are not sure, use an adjacency list (most applications have sparse graphs).

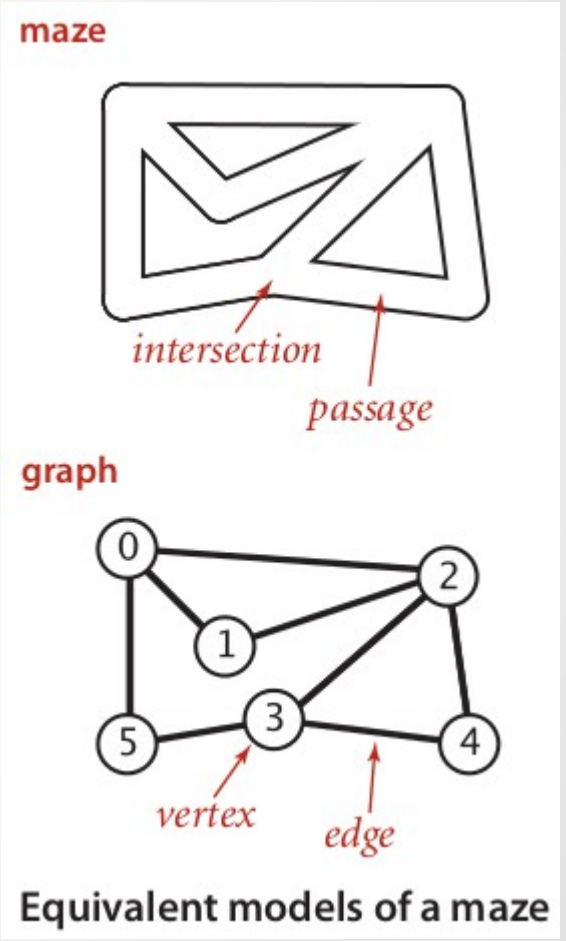
Data Structure Implementation	memory	add edge	v adjacent to w	iterate v's neighbors
Edge List	E	1	E	E
Adjacency Matrix	V^2	1	1	V
Adjacency List	V + E	1	degree(v)	degree(v)

Graph Problems

- Given a graph.
  - Connectivity problem: Are two given vertices v and w connected?
  - Connected Components problem: How many connected components for a given graph?
- Given a graph and source vertex s
  - Single-source paths problem: Is there a path(s) between from s to some v in the graph? If so, find one.
  - Single-source shortest paths problem: What is the shortest path from s to some v in the graph?

Depth First Search

- Can be used to solve:
  - Connectivity problem
  - Connected Components problem
  - Single-source paths problem
- Derived from maze searching algorithm
  - Theseus and the Minotaur
  - Explore maze without getting lost
- Depth first search intuition
  - Visit a vertex, mark as visited
  - Visit neighbors not yet marked



Equivalent models of a maze

Depth First Search Algorithm

```
// Iterative version of DFS psuedocode
// Implicitly receives Graph and source
// Returns the marked and paths arrays
private void search( )
{
    Create the stack
    Push the source onto the stack
    Set source path to itself
    While the stack is not empty
        Pop v from the stack
        If v is not yet marked
            Mark v
            For each v's neighbors
               If neighbor is not marked
                   Push onto the stack
                   Update paths for neighbor to v
}
```

Depth First Search Example

Graph: V=6 E=8

[0] neighbors=5, 2, 1

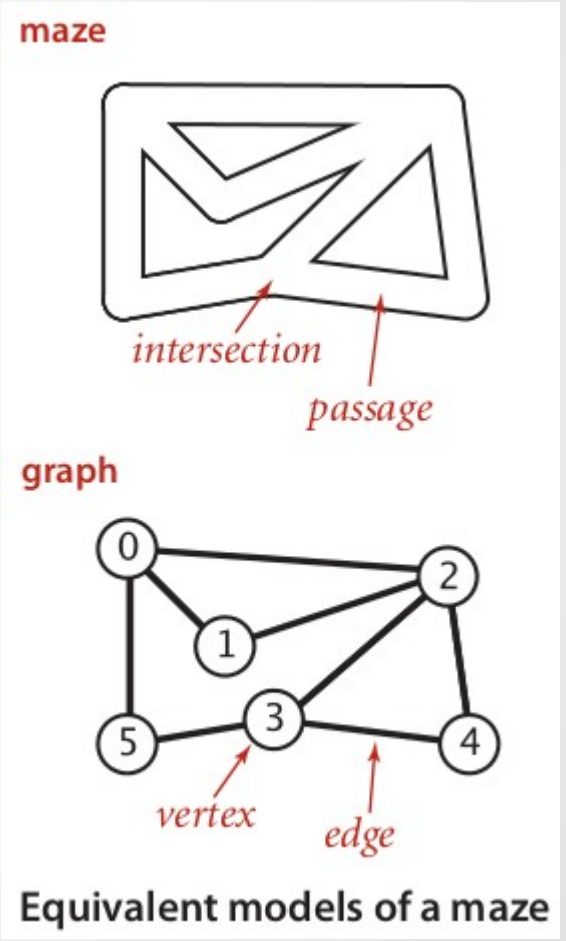
[1] neighbors=2, 0

[2] neighbors=4, 3, 1, 0

[3] neighbors=4, 2, 5

[4] neighbors=2, 3

[5] neighbors=3, 0



Equivalent models of a maze

Iterating DFS Path

- Easily determine if path exists using marked array
- Use a stack to push the vertices in order from v until we encounter the source vertex. The resulting stack will iterate the path in reverse order (i.e. from source to v)

```
// Returns path from the source to the given vertex v,in that order.
// Given: paths[], for example [0,0,1,2,3,3]
public Iterable<Integer> pathFromSource( int v )
{
    If path does not exists from source to v, return null
    Create a stack
    While v is not the source
        Push v onto the stack
        Set v equal to paths at v
    Return stack
}
// pathFromSource(5): ->0->1->2->3->5
// Note: Stack reverses the order
```

Depth First Search Performance

- DFS marks all vertices connected to a given source in time proportional to the sum of their degrees, which is 2E
  - Have to also initialize marked arrays and path array both of size V
  - Note: Incorrect analysis is VE, can't just count loops
  - O(V + E)
- May add vertex to stack twice for each edge, looking at the vertex from both directions
- Keep track of path back to source in an array. Can use simple integer array (“paths”) because a tree rooted at the source is created

Breadth First Search

- Can be used to solve:
  - Single-source shortest paths problem
  - Weiss calls this unweighted shortest path problem
    - Special case of weighted shortest path problem where cost for all edges is 1
- Breadth first search intuition
  - Processes in layers
  - Those vertices closest to source marked first
  - Fans out from the source

Breadth First Search Algorithm

```
// BFS psuedocode
// Implicitly receives Graph and source
// Returns the marked and paths arrays
private void search( )
{
    Create the queue
    Mark the source
    Set source path to itself
    Enqueue the source
    While the queue is not empty
        Dequeue v
        For each v's neighbors
            If the neighbor is not yet marked
                Mark neighbor
                Update paths for neighbor to v
                Enqueue neighbor
}
```

Breadth First Search Example

Graph: V=6 E=8

[0] neighbors=5, 2, 1

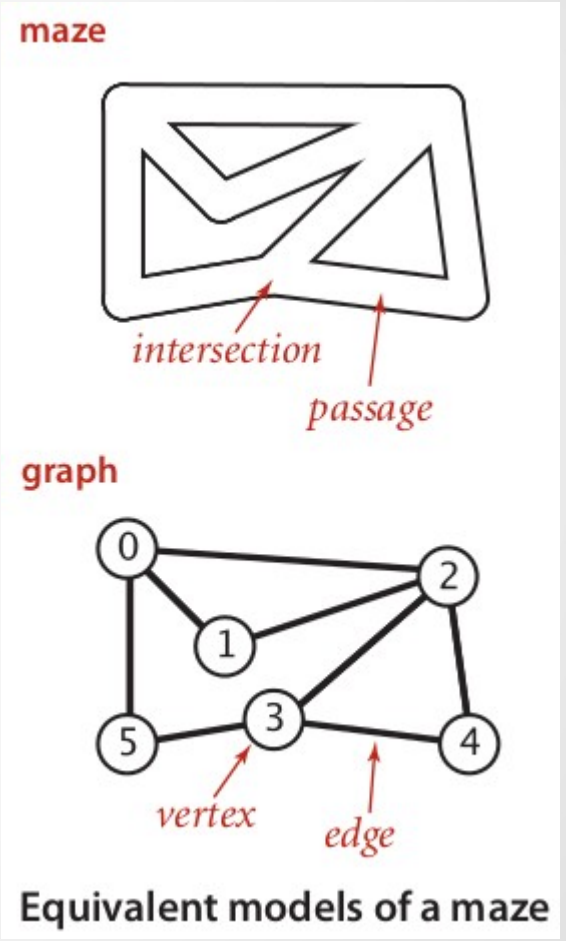
[1] neighbors=2, 0

[2] neighbors=4, 3, 1, 0

[3] neighbors=4, 2, 5

[4] neighbors=2, 3

[5] neighbors=3, 0



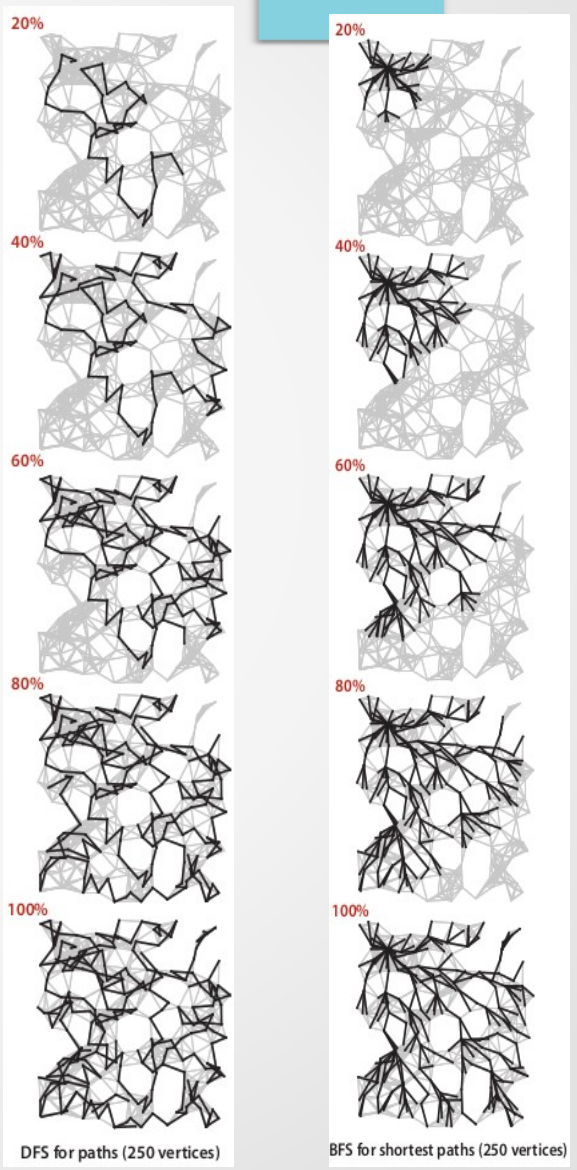
Equivalent models of a maze

Breadth First Search Performance

- BFS takes time proportional to the sum of their degrees, which is 2E
  - As with DFS, time is proportional to sum of the vertex degrees. If the graph is connected, this is 2E.
  - Creating/maintaining the marked and paths arrays takes time proportional to V
  - O(V + E)

Breadth First vs Depth First Search

- Depth first is LIFO
  - Moves far away from source before returning
- Breadth first is FIFO
  - Fans out one layer at a time from the source
- Algorithms
  - DFS adds unmarked to stack, marks when pop'ed
  - BFS adds marked vertices to queue



Marking Algorithms

- Generally proceed as follows:
  - Take next unmarked vertex v from data structure and mark it
  - Add all unmarked neighbors of v to data structure
- Difference is when to take the next vertex
  - DFS most recently added
  - BFS least recently added
- All vertices and edges are examined for both

Connected Components

- Given a graph.
  - Connected Components problem: How many connected components for a given graph?
- Can we use tools we already have to solve this problem?
  - Yes. Use DFS repeatedly until all vertices are marked. Each vertex is tagged with a component identifier. Each time we perform a new DFS, update the component identifier.
  - Takes space and time proportional to V + E to support constant time connectivity operation.

Connected Components Algorithm

```
// CC constructor
public CC( Graph graph )
{
    ...
    this.componentId = new int[graph.V()];
    for (int v = 0; v < this.graph.V(); v++)
    {
        if( this.marked[v] == false )
        {
            this.search( v );
            this.count++;
        }
    }
}

// DFS psuedocode
private void search( )
{
    ...
        If v is not yet marked
            Mark v
            Update componentId to count
            ...
}
```

Cycle Detection

- Given a graph.
  - Cycle Detection problem: Is the graph acyclic?
- Can we use tools we already have to solve this problem?
  - Yes. Use DFS with modification for each search to keep track of additional parameter representing the current vertex.
  - If the current vertex is seen as a neighbor in the subsequent searches, then there must exist a path from the current vertex back to itself – i.e. a cycle exists.

Undirected Graph Problems Summary

From Sedgewick/Wayne 4.1

- BFS and DFS are fundamental graph processing algorithms and subsequent algorithms directly use or are some variant of these approaches
  - Both work with unweighted, directed graphs (Digraphs)

Problem	BFS	DFS	Time
Single-source Paths	X	X	E + V
Single-source Shortest Paths	X		E + V
Connected Components	X	X	E + V
Cycle Detection	X	X	E + V

Questions?

Graph Representations

- Edges can be weighted/non-weighted and/or directed/un-directed. The edge representation determines the type of graph.
- Undirected Graph
  - Unweighted (Graph.java)
  - Weighted (WeightedGraph.java)
- Directed Graph
  - Unweighted (Digraph.java)
  - Weighted (WeightedDigraph.java)
- Problems/solutions may only apply to a given representation.

Edge-weighted, Undirected Graphs

- Edges consist of two vertices and a weight
  - Requires explicit ADT, should be Comparable
- Spanning tree weight is sum of the tree's edge weights
- A minimum spanning tree (MST) is the spanning tree with the smallest spanning tree weight
  - If edge weights are not unique, may be many MSTs
- Given a connected undirected graph with edge weights.
  - Minimum spanning tree problem: Find an MST for a given graph?

Weighted Edge ADT

```
public class WeightedEdge implements Comparable<WeightedEdge> {
    private final int v;
    private final int w;
    private final double weight;
    ...
    public int either() {
        return this.v;
    }

    public int other( int v ) {
        return ( this.v == v ) ? this.w : this.v;
    }

    public double weight() {
        return this.weight;
    }

    public int compareTo(WeightedEdge that) {
        if(      this.weight < that.weight ) return -1;
        else if( this.weight > that.weight ) return +1;
        return 0;
    }
}
```

Edge-weighted, Undirected Graph ADT

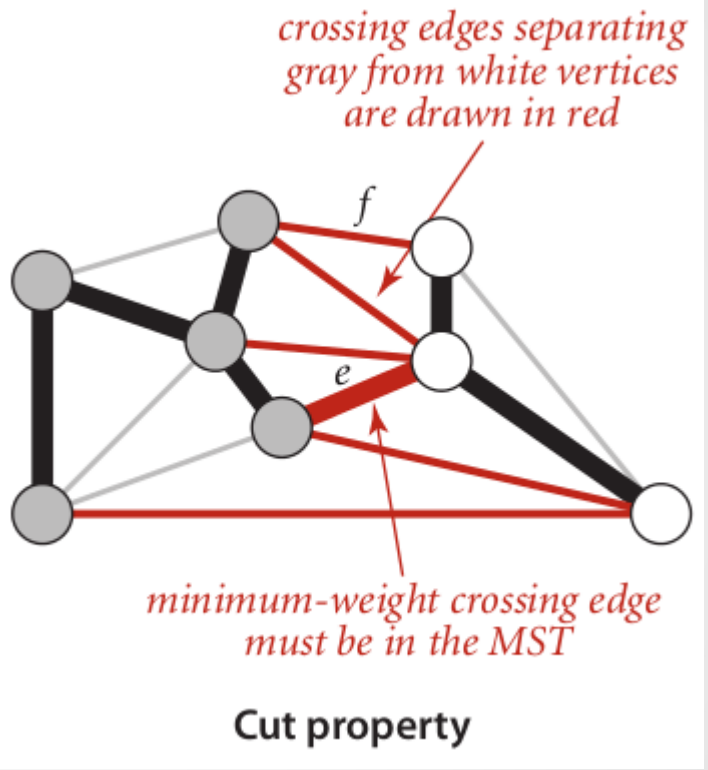
- Again, use Edge in both directions for undirected link.
  - Use same Edge object pointed to from different Bags

```
public class WeightedGraph {
    private final Bag<WeightedEdge>[] vertices;
    private int E;
    ...
    public Iterable<WeightedEdge> neighbors( int vertexId )...
    public Iterable<WeightedEdge> edges() ...

    public void addEdge( int v, int w, double weight ) {
        this.addEdge( new WeightedEdge(v,w,weight) );
    }

    public void addEdge( WeightedEdge e ) {
        int v = e.either();
        int w = e.other(v);
        this.vertices[v].add(e);
        this.vertices[w].add(e);
        this.E++;
    }
}
```

Cuts

Sedgewick 4.3

- Cut of a graph is a partition into two non-empty, disjoint set of vertices. A crossing edge is an edge where one vertex is in one set and the other vertex is in the other set.



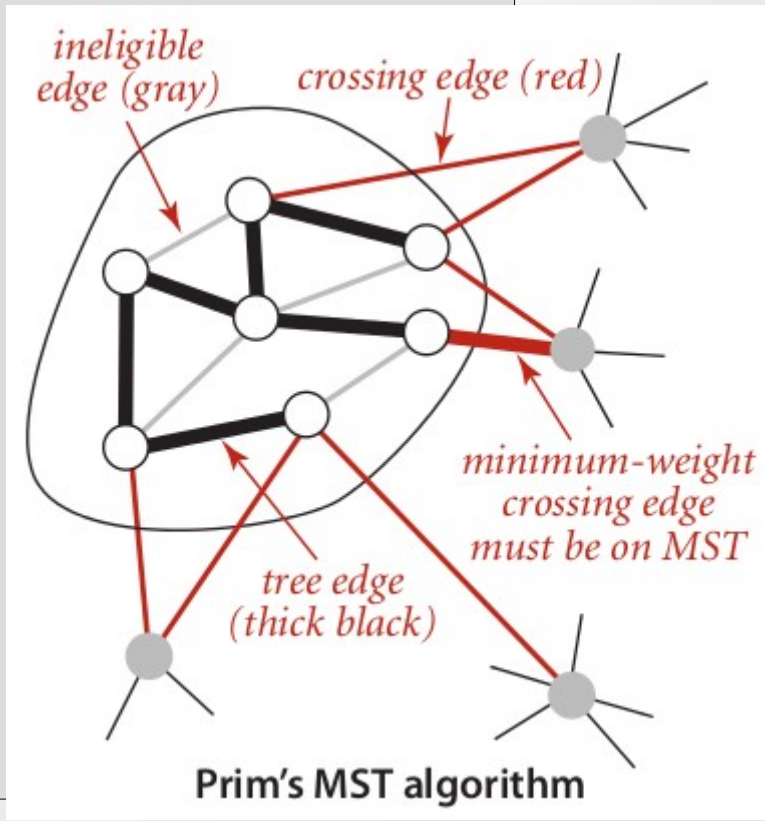
- Cut property: Given any cut in an edge weighted graph, the crossing edge of minimum weight is in the MST.

Prim's MST Algorithm

- Given a connected undirected graph with edge weights.
  - Minimum spanning tree problem: What is a MST for the given graph? Use the cut property.
- Assume set of vertices already selected (i.e. marked) and part of the current MST.
  - There is a cut (set of edges) that connects the current MST to the remaining vertices (set of unmarked vertices). Which edge to add?
- Greedy algorithm: at each step make choice that is locally optimal with hope being close or equal to global optimal.
  - Select crossing edge of min weight. Repeat.

Prim's MST Algorithm

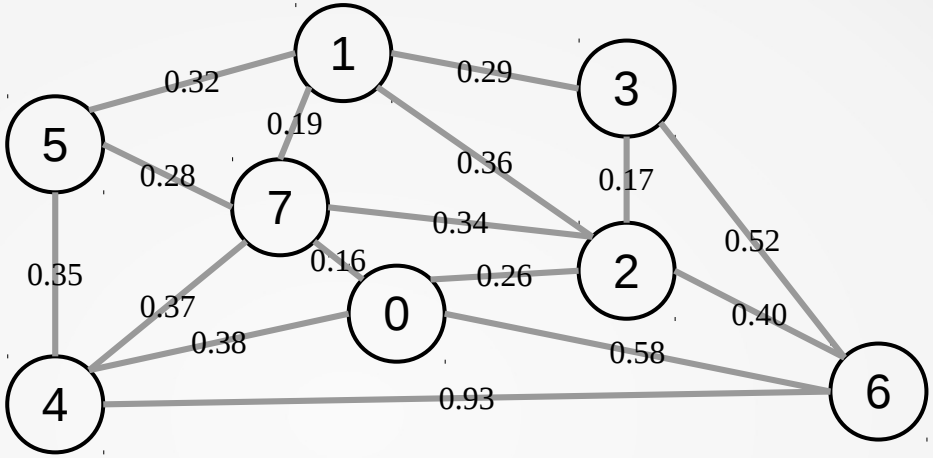
```
private final MinHeap<WeightedEdge> minPQ;
private final boolean[] intree;

private void mst( WeightedGraph graph, int sourceVertex ) {
    // Visit marks vertex, adds unmarked neighbors to minPQ
    this.visit(graph, sourceVertex);

    while( this.minPQ.isEmpty() == false ) {
        WeightedEdge minEdge = minPQ.delMin();
        // One marked, one not marked
        // means crossing edge
        int v = minEdge.either();
        int w = minEdge.other(v);
        if( !intree[v] || !intree[w] ) {
            // Exactly one of them is
            // in tree, the other is not
            w = ( intree[v] ) ? w : v;
            this.spanningTree.add(minEdge);
            this.visit(graph, w);
        }
    }
}
```



Prim Example MST

Edge List

0 7 0.16
2 3 0.17
1 7 0.19
0 2 0.26
5 7 0.28
1 3 0.29
1 5 0.32
2 7 0.34
4 5 0.35
1 2 0.36
4 7 0.37
0 4 0.38
6 2 0.40
3 6 0.52
6 0 0.58
6 4 0.93



Adjacency List

[0] neighbors=4: (6,0=0.58), (0,2=0.26), (0,4=0.38), (0,7=0.16)
[1] neighbors=4: (1,3=0.29), (1,2=0.36), (1,7=0.19), (1,5=0.32)
[2] neighbors=5: (6,2=0.40), (2,7=0.34), (1,2=0.36), (0,2=0.26), (2,3=0.17)
[3] neighbors=3: (3,6=0.52), (1,3=0.29), (2,3=0.17)
[4] neighbors=4: (6,4=0.93), (0,4=0.38), (4,7=0.37), (4,5=0.35)
[5] neighbors=3: (1,5=0.32), (5,7=0.28), (4,5=0.35)
[6] neighbors=4: (6,4=0.93), (6,0=0.58), (3,6=0.52), (6,2=0.40)
[7] neighbors=5: (2,7=0.34), (1,7=0.19), (0,7=0.16), (5,7=0.28), (4,7=0.37)

Prim's MST Algorithm Analysis

- Even though Prim's MST algorithm is greedy (making only locally optimal decisions) it turns out to be globally optimal also.
  - This is exceptional, greedy algorithms are rarely globally optimal and can result in very poor solutions.
- Relies on priority queue implementation.
  - Add up to E edges to MinPQ, E lg(E)
  - Delete up to E edges from MinPQ, E lg(E)
- Described approach is Lazy Prim as it leaves edges in MinPQ that we could remove when it's unmarked vertex is marked from another edge (keeps MinPQ ~ V). Eager Prim approach does this, need IndexedMinPQ

Kruskal's MST Algorithm

- Given a connected undirected graph with edge weights.
  - Minimum spanning tree problem: What is a MST for the given graph?
- Idea: Enumerate all the edges in the graph in order smallest to largest weight. If the edge does not create a cycle, then add it to the MST. Stop when V-1 edges.
  - Can use connectivity test to determine if two vertices are in the same tree (tree is a graph).
  - Use UnionFind data structure to determine and update connectivity. Good for dynamic situations like this, our DFS approach was static.

Kruskal's MST Algorithm

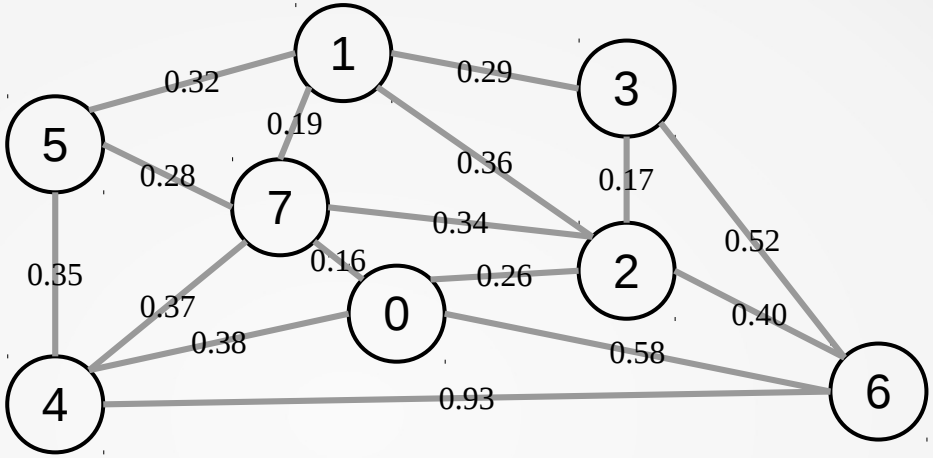
From Sedgewick/Wayne 4.3

```
public KruskalMST(WeightedGraph G)
{
    spanningTree = new Queue<WeightedEdge>();
    MinPQ<WeightedEdge> pq = new MinPQ<WeightedEdge>(G.edges());
    UF uf = new UF(G.V());
    while (!pq.isEmpty() &&
           spanningTree.size() < G.V()-1)
    {
        // Get min weight edge on
        // pq and its vertices
        WeightedEdge e = pq.delMin();
        int v = e.either(), w = e.other(v);
        // Ignore ineligible edges.
        if (uf.connected(v, w)) continue;
        // Merge components.
        uf.union(v, w);
        // Add edge to MST
        spanningTree.enqueue(e);
    }
}
```

Kruskal Example MST

Edge List

0 7 0.16
2 3 0.17
1 7 0.19
0 2 0.26
5 7 0.28
1 3 0.29
1 5 0.32
2 7 0.34
4 5 0.35
1 2 0.36
4 7 0.37
0 4 0.38
6 2 0.40
3 6 0.52
6 0 0.58
6 4 0.93



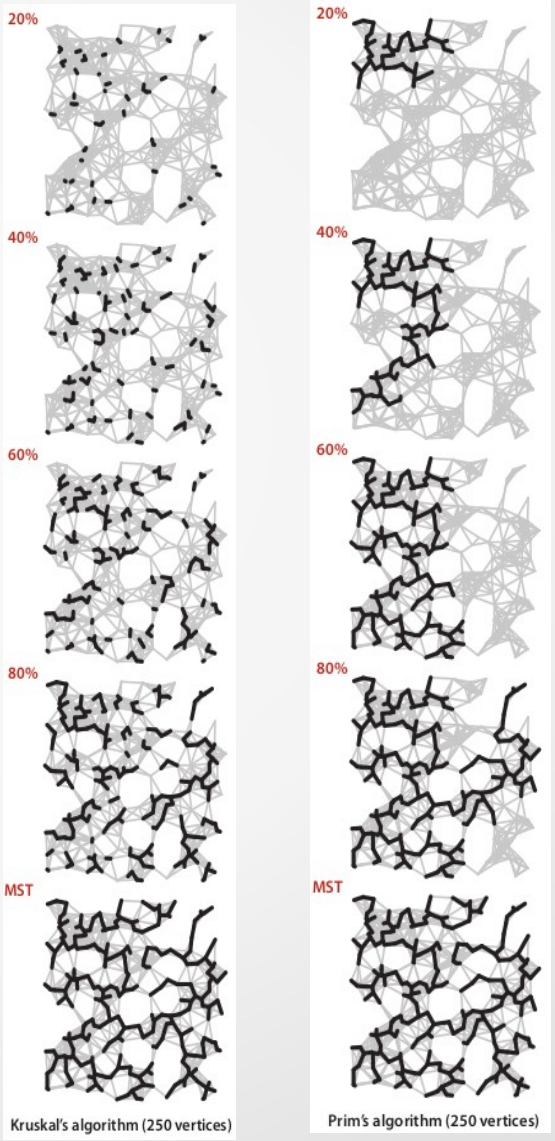
Adjacency List

[0] neighbors=4: (6,0=0.58), (0,2=0.26), (0,4=0.38), (0,7=0.16)
[1] neighbors=4: (1,3=0.29), (1,2=0.36), (1,7=0.19), (1,5=0.32)
[2] neighbors=5: (6,2=0.40), (2,7=0.34), (1,2=0.36), (0,2=0.26), (2,3=0.17)
[3] neighbors=3: (3,6=0.52), (1,3=0.29), (2,3=0.17)
[4] neighbors=4: (6,4=0.93), (0,4=0.38), (4,7=0.37), (4,5=0.35)
[5] neighbors=3: (1,5=0.32), (5,7=0.28), (4,5=0.35)
[6] neighbors=4: (6,4=0.93), (6,0=0.58), (3,6=0.52), (6,2=0.40)
[7] neighbors=5: (2,7=0.34), (1,7=0.19), (0,7=0.16), (5,7=0.28), (4,7=0.37)

Prim versus Kruskal

From Sedgewick/Wayne 4.3

- Kruskal starts by creating clusters that gradually get larger and evolve into larger trees until MST
- Prim starts from the source and expands out to include more and more of the graph.
- Assuming unique edge weights, both approaches result in the same MST.
- In practice Prim performs a little better than Kruskal.



MST Summary

From Sedgewick/Wayne 4.3

- MST algorithms primarily limited by priority queue
  - There are better performing MinPQ implementations but complicated to implement
- Prim/Kruskal cannot be applied to directed graphs
  - MST for directed graph known as the *minimum cost arborescence* problem

MST Algorithm	memory	time
Lazy Prim	E	E lg E
Eager Prim	V	E lg V
Kruskal	E	E lg E

Unknown, but linear time unlikely.

Questions?

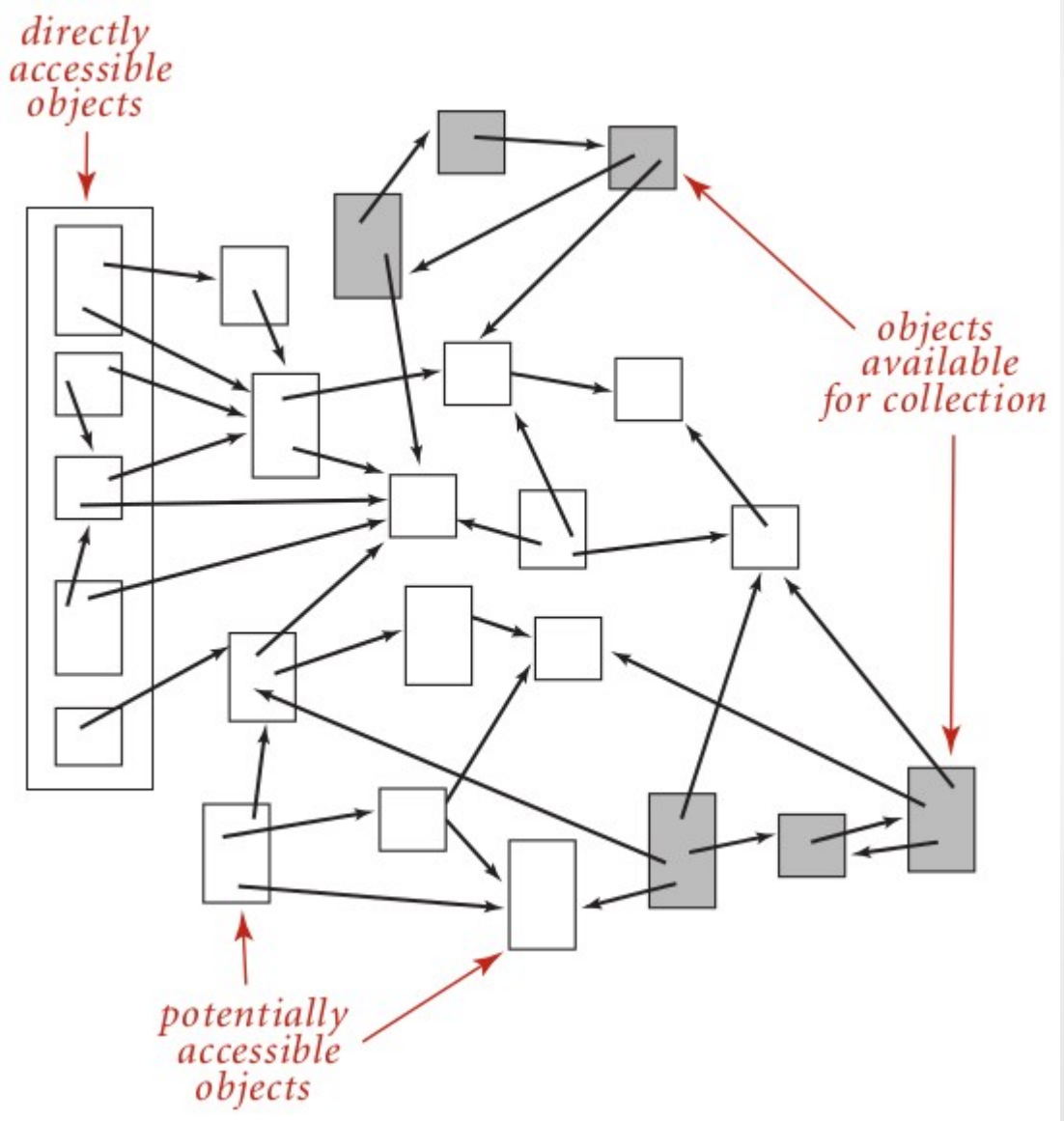
Directed Graphs

OPTIONAL Sedgewick/Wayne 4.2

- Edges consist of two ordered vertices
  - Vertex from and to
  - Directed edge abbreviated Diedge
- Digraph consists of directed edges
  - Adding an edge only adds in one from-to direction
- Example Problems:
  - Single-source directed paths (BFS/DFS?)
  - Single-source shortest directed paths (BFS?)
  - Single-source reachability (existence of path)
  - Multiple-source reachability (existence of path)
    - What can this be used for?

Directed Graph Mark-and-Sweep Application

OPTIONAL Sedgewick/Wayne 4.2



Edge-weighted, Directed Graphs

- Edges consist of two ordered vertices and a weight
  - Vertex from and to
- Digraph may only have an edge in one direction unlike the previous undirected and weighted graphs
  - Problems are generally harder
- Given a connected directed graph with edge weights.
  - Single source shortest paths problem: Find shortest path from source to every other vertex?

Directed Graph Applications Overview

Sedgwick 4.2

Graph	Vertex	Edge
food chain web	species	predator–prey
internet content	page	URL
computer program	module	reference
cell phone	phone	call
research papers	paper	citation
financial	stock	transaction
computer network	machine	connection

Weighted, Directed Edge ADT

```
public class WeightedDiedge implements Comparable<WeightedDiedge> {
    private final int from;
    private final int to;
    private final double weight;
    ...
    public int from() {
        return this.from;
    }

    public int to() {
        return this.to;
    }

    public double weight() {
        return this.weight;
    }

    public int compareTo(WeightedEdge that) {
        if(      this.weight < that.weight ) return -1;
        else if( this.weight > that.weight ) return +1;
        return 0;
    }
}
```

Edge-weighted, Directed Graph ADT

- Combination of previous weighted/directed graphs

```
public class WeightedDigraph {
    private final Bag<WeightedDiedge>[] vertices;
    private int E;
    ...
    public Iterable<WeightedDiedge> neighbors( int vertexId )...
    public Iterable<WeightedDiedge> edges() ...

    public void addEdge( int from, int to, double weight ) {
        this.addEdge( new WeightedDiedge(from,to,weight) );
    }

    public void addEdge( WeightedDiedge e ) {
        this.vertices[e.from()].add(e);
        this.E++;
    }
}
```

Relaxing a Directed Edge

- To relax an edge from v->w means to test whether the best known way from s to w is to go from s to v. If it is, then update the data structure to take the edge from v->w.
  - edgeTo[V] edge connects v to parent, all initially null
  - distTo[V] current distance to v, initially INF, distTo[s] = 0

```
private void relax(WeightedDiedge edge) {
    int v = edge.from(); int w = edge.to();
    // Is better to go through v to get to w
    if ( distTo[w] > distTo[v] + edge.weight() ) {
        // Update distance to w and remember edge
        distTo[w] = distTo[v] + edge.weight();
        edgeTo[w] = edge;
    }
}
```

Relaxing a Vertex

- To relax an vertex means to relax each of the edges.

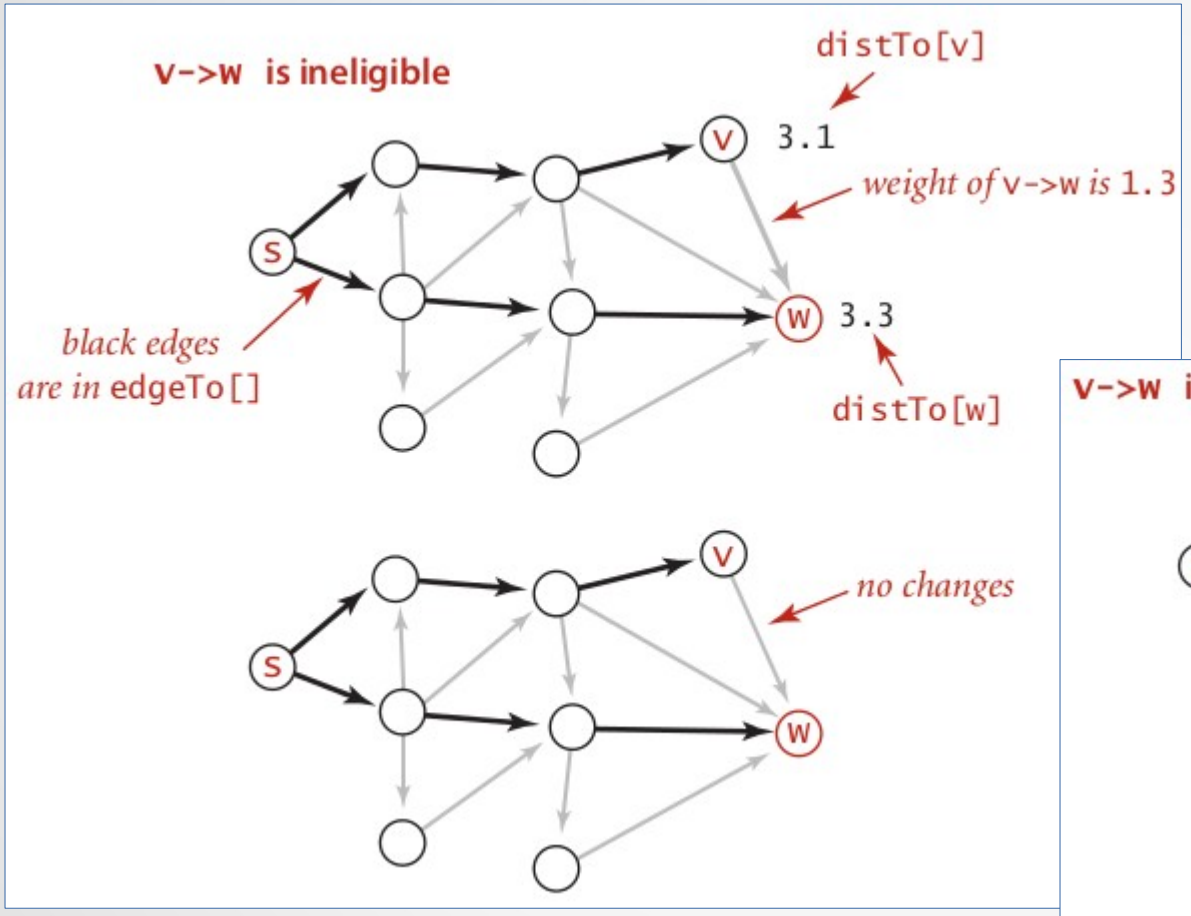
```
private void relax(WeightedDigraph graph, int v) {
    for( WeightedDigraph edge : graph.neighbors(v) ) {
        int w = edge.to();
        // Is better to go through v to get to w
        if ( distTo[w] > distTo[v] + edge.weight() ) {
            // Update distance to w and remember edge
            distTo[w] = distTo[v] + edge.weight();
            edgeTo[w] = edge;
        }
    }
}
```

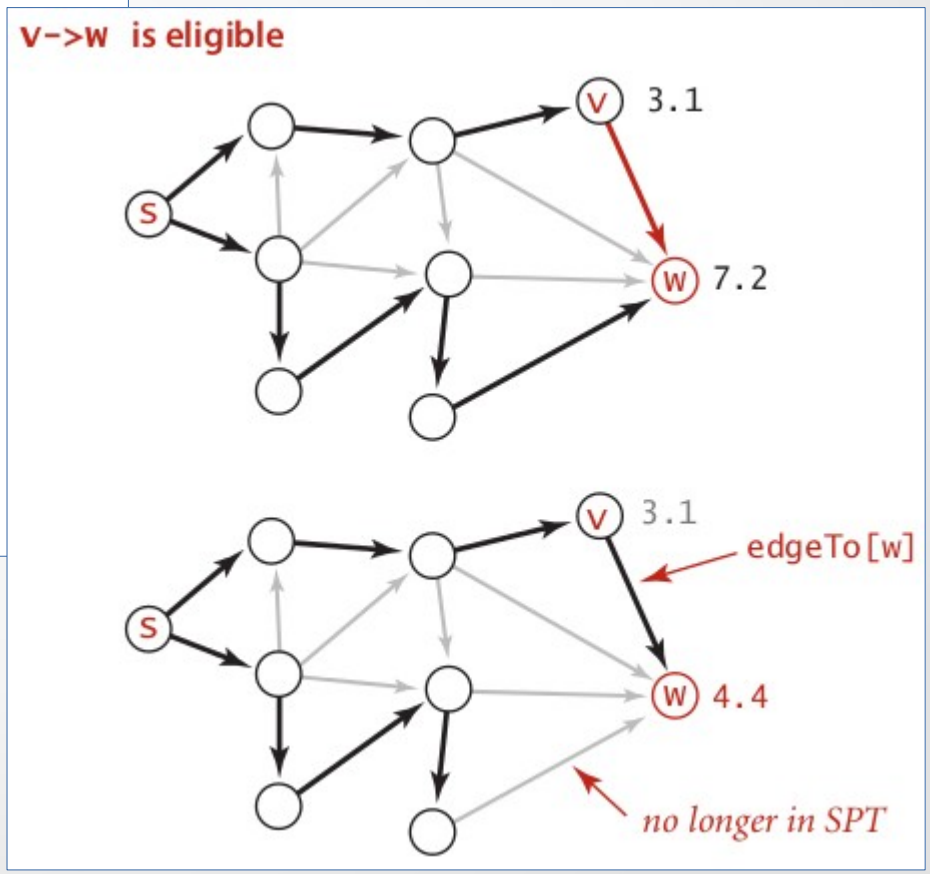
Generic Shortest Path Tree (SP) Algorithm

- An edge is eligible if it can be successfully relaxed, i.e. distTo[w] > distTo[v] + edge.weight()
- Generic algorithm: relax any edge in the graph until no more edges are eligible.
  - Many SP algorithms use this fact for correctness. Differ in how/order the edges are selected.
- Idea: Use a Prim-like approach where we start with a shortest path tree and add to it one at a time. Add vertex with the minimum edge to the tree next and relax each of its edges.
  - When complete (proof omitted), all eligible edges will have been relaxed.

Relaxing Examples

Sedgewick/Wayne 4.4





Dijkstra's SP Algorithm

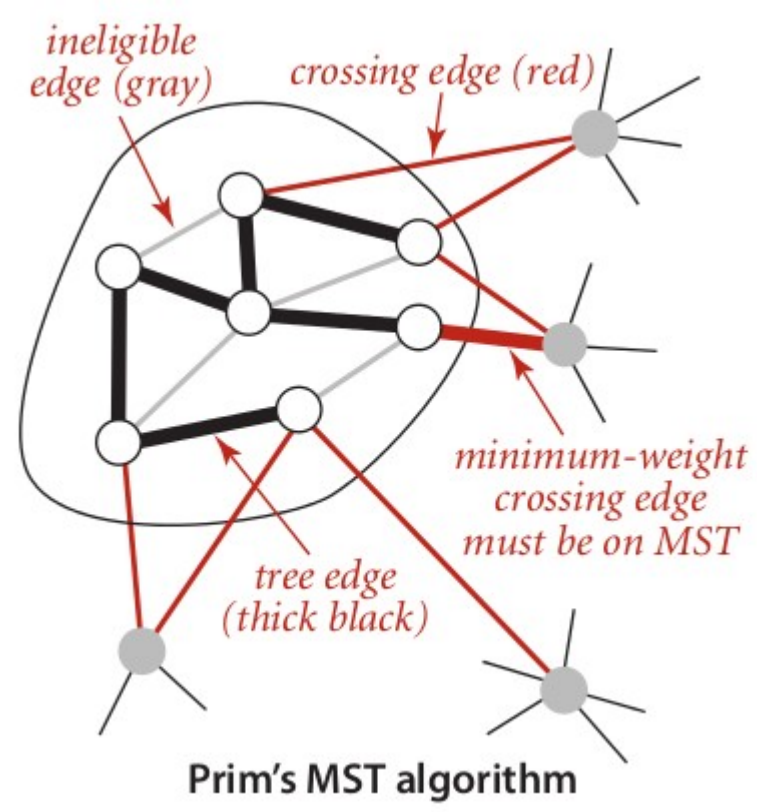
- Use a Prim-like idea. Works only with non-negative edge weights.

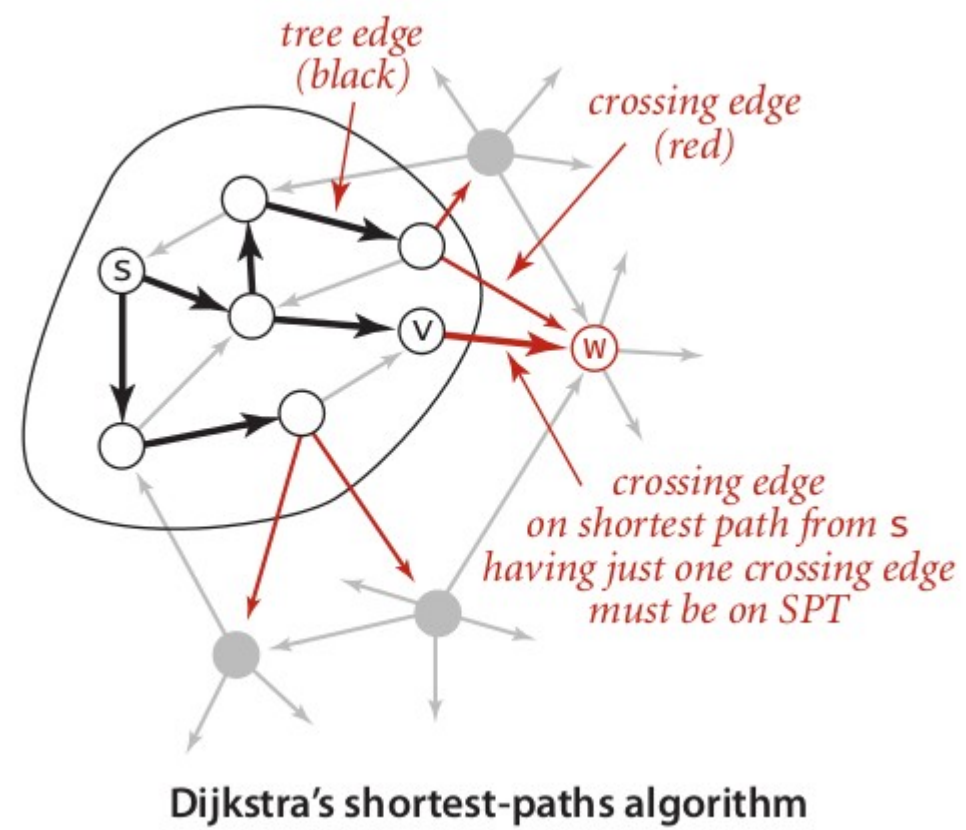
```
Given: weighted directed graph, source vertex s

Make sure all edges of graph are non-negative

Initialize distTo[V] to Double.MAX_VALUE, distTo[s] = 0.0
Initialize edgeTo[V] to null
Initialize minimum priority queue minPQ

Add source vertex's neighbors to the minPQ

While minPQ is not empty
    Remove next vertex v from minPQ
    If v has not yet been processed
        Add v to SPT
        Relax v
```

Prim versus Dijkstra SPT Algorithm

Sedgewick/Wayne 4.4

- Prim adds next non-tree vertex closest to the tree
- Dijkstra adds next non-tree vertex closest to source





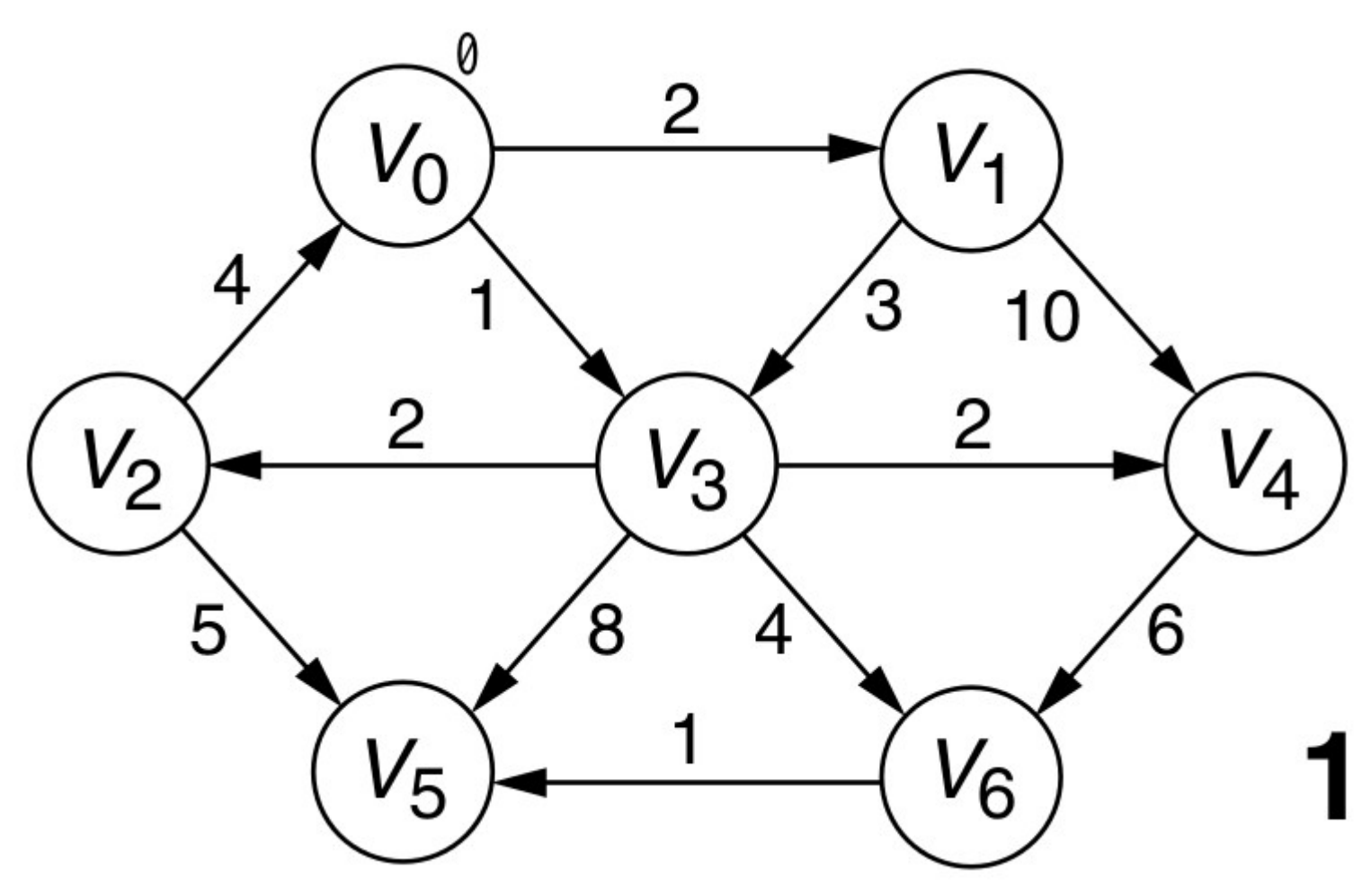
Dijkstra's SP Algorithm Analysis

Weiss 14.3.1

- Need to know how large the priority queue minPQ
  - Can add up to |E| objects, one for each time we enumerate the edges of a vertex during relax.
- Can use simple priority queue
  - Discards unnecessary objects in the minPQ instead of trying to avoid adding them.
  - O(E lg (E))
- Better implementations only keep |V| objects in minPQ and either update existing path go a vertex or add the path to the minPQ
  - Requires *pairing heap* with decreaseKey operation
  - O(E lg (V))

Dijkstra Example

Weiss Figure 14.25



Bellman-Ford Algorithm

Weiss 14.4

- Dijkstra's SP Algorithm efficiently solves single-source shortest paths problems for a weighted, directed graph only if all edge weights are non-negative.
  - Reasonable assumption? Certainly not always.
- Bellman-Ford Algorithm
  - Handles negative edge weights
  - Can solve only if no negative cycles
  - Time proportional EV
- Why do negative cycles cause a problem?

All-pairs Shortest Paths Weighted Digraph

OPTIONAL. From https://en.wikipedia.org/wiki/Floyd_Warshall_algorithm

- Shortest path tree (SPT) is same for all sources in an undirected graph. Directed graphs potentially have different SPT for each source vertex.
  - All-pairs shortest paths problem
- Floyd-Warshall Algorithm. Handles negative weights, but not negative cycles. $O(V^3)$

```
for k from 1 to |V| // standard Floyd-Warshall implementation
    for i from 1 to |V|
        for j from 1 to |V|
            if dist[i][k] + dist[k][j] < dist[i][j] then
               dist[i][j] ← dist[i][k] + dist[k][j]
               next[i][j] ← next[i][k]
```

- If non-negative edge weights and sparse graph
  - Better to use Repeated Dijkstra SP Algorithm
  - O(VE lg (V))

Questions?

Graph/Problems Summary

Problem	Graph	Weighted Graph	Digraph	Weighted Digraph
Single-source Paths	BFS,DFS	BFS,DFS	BFS,DFS	BFS,DFS
Single-source Shortest Paths	BFS	Dijkstra*, Bellman-Ford**	BFS	Dijkstra*, Bellman-Ford**
All-pairs Shortest Paths		Dijkstra*, Floyd-Warshall**		Dijkstra*, Floyd-Warshall**
Connected Components (CC)	BFS,DFS	BFS,DFS		
Minimum Spanning Tree		Prim Kruskal		

* Applicable when edge weights are non-negative.

** Applicable when no negative cycles.

Questions?

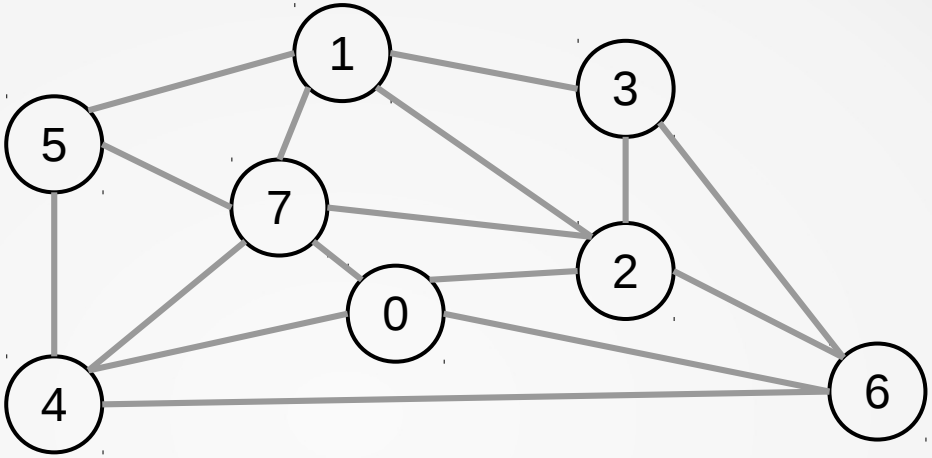
PA11

- Implement Dijkstra's Shortest Path Algorithm. Refer to Weiss 14.3.2 implementation O(E lg (E)).

Extra Credit – Graph Coloring Problem

- Given an non-weighted, un-directed graph, find some minimum number of colors necessary to color the graph so that no two adjacent vertices have the same color.
  - You will be given a graph with 1000 vertices
  - You must find a color assignment that results in below 300 distinct colors to receive full credit

Adjacent colors must be different



Coloring

0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7

GC(G)=7

Free Question Time!