INFS 519 – Fall 2015

Program Design and Data Structures

Lecture 13

Instructor: James Pope

Guest: Maryam Bandari

Email: jpope8@gmu.edu

Today

- Interview questions and problems
  - Maryam Bandari
- Review Last Class
  - Directed/Edge weights Graphs
  - Shortest Path Algorithms
  - Minimum Spanning Tree
- Schedule
  - Union-find, weighted-union, path-compression
  - Course Evaluation
  - Bloom Filters (next week)

Interview Questions And Problems

Maryam Bandari
Ph.D. Candidate
Computer Science

Course Evaluation

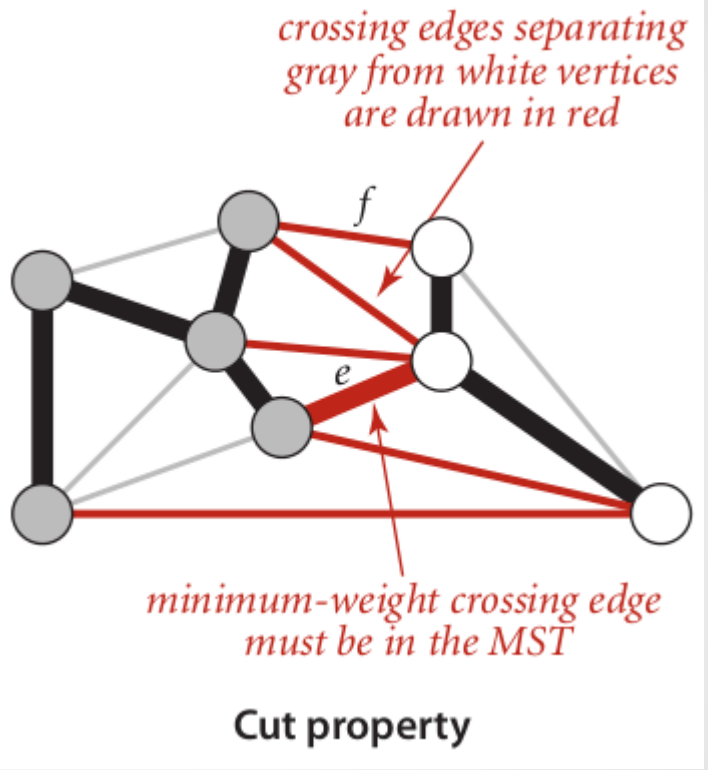
Graph Representations

- 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.

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.



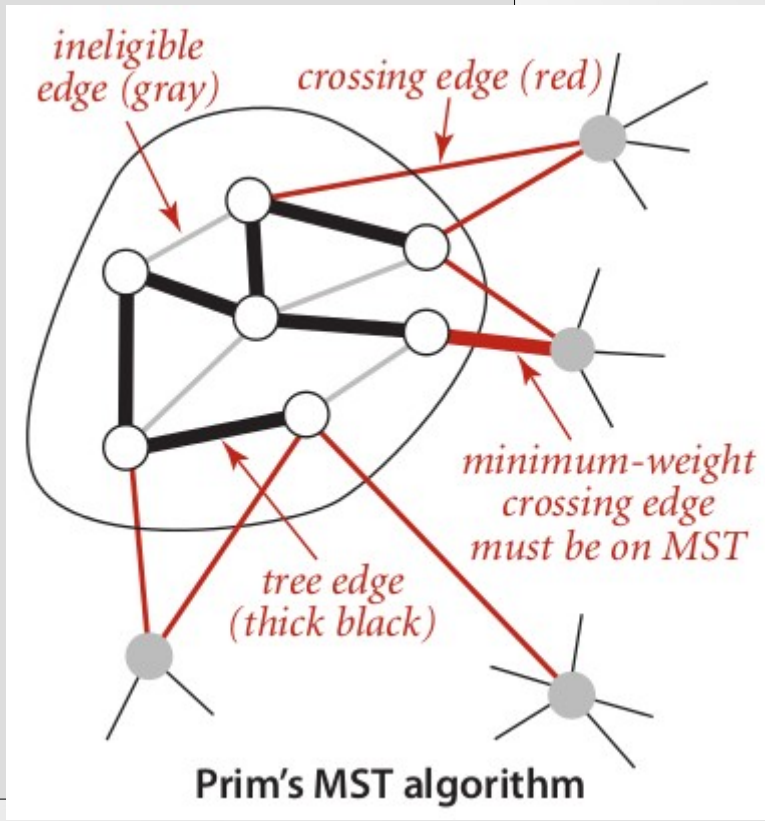
- Cut property: Given any cut in an edge weighted graph, the crossing edge of minimum weight is in the MST.

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

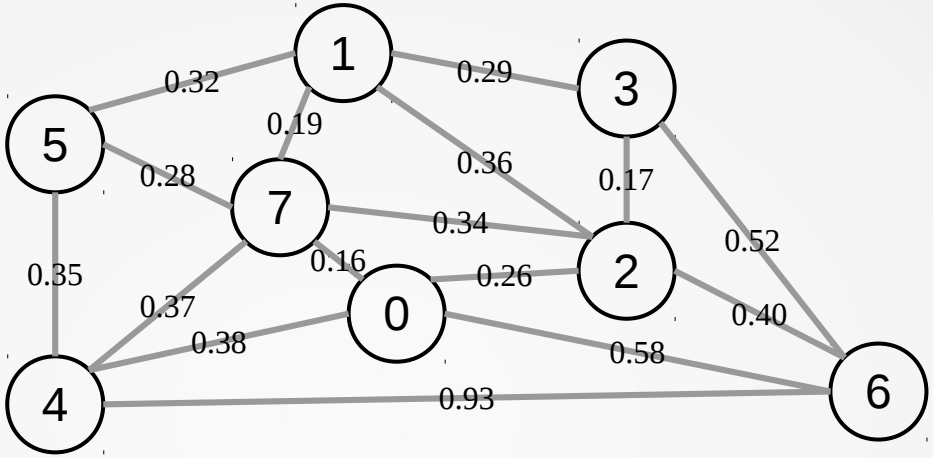
```
private final MinHeap<WeightedEdge> minPQ;
private final boolean[] intree;

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 (*online algorithm*) like this, our DFS approach was static (*offline algorithm*).

Kruskal's MST Algorithm

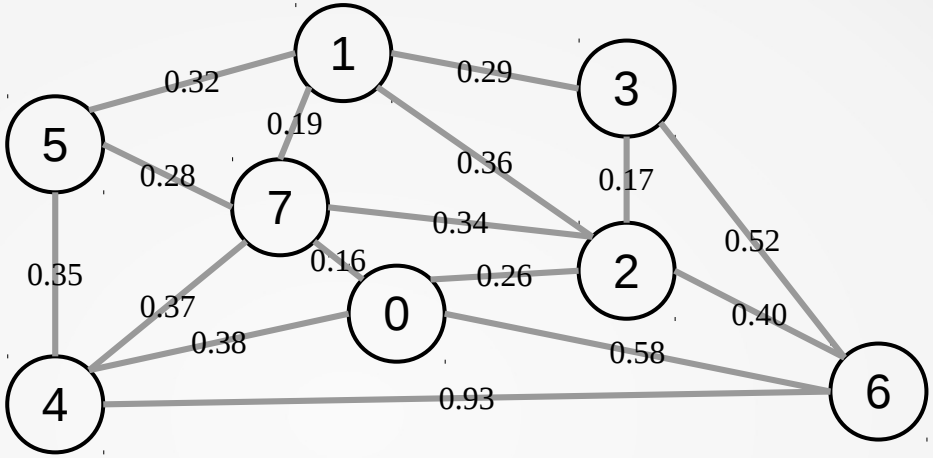
From Sedgewick/Wayne 4.3

```
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



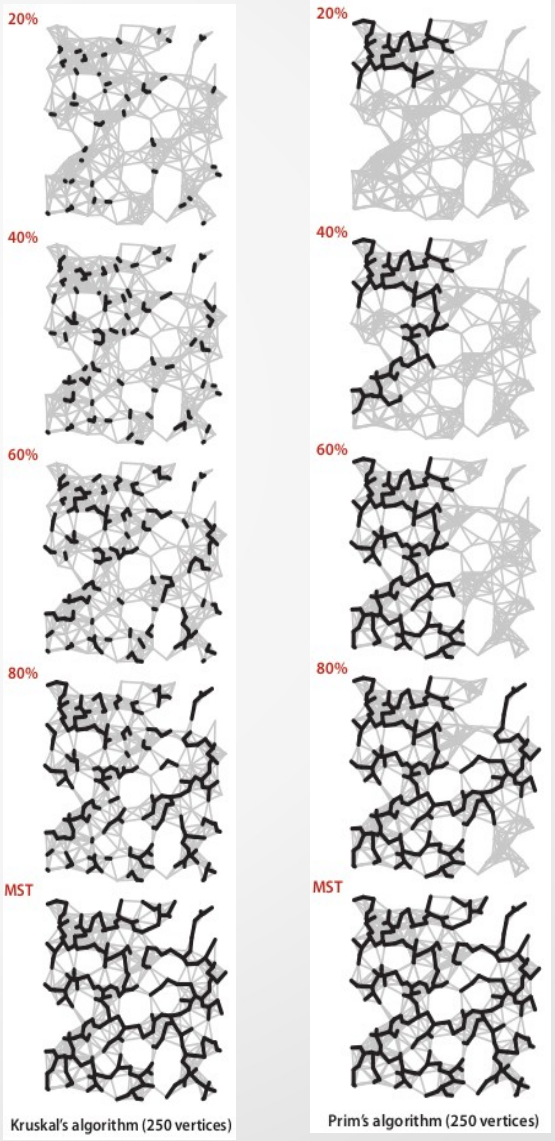
Adjacency List

[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.

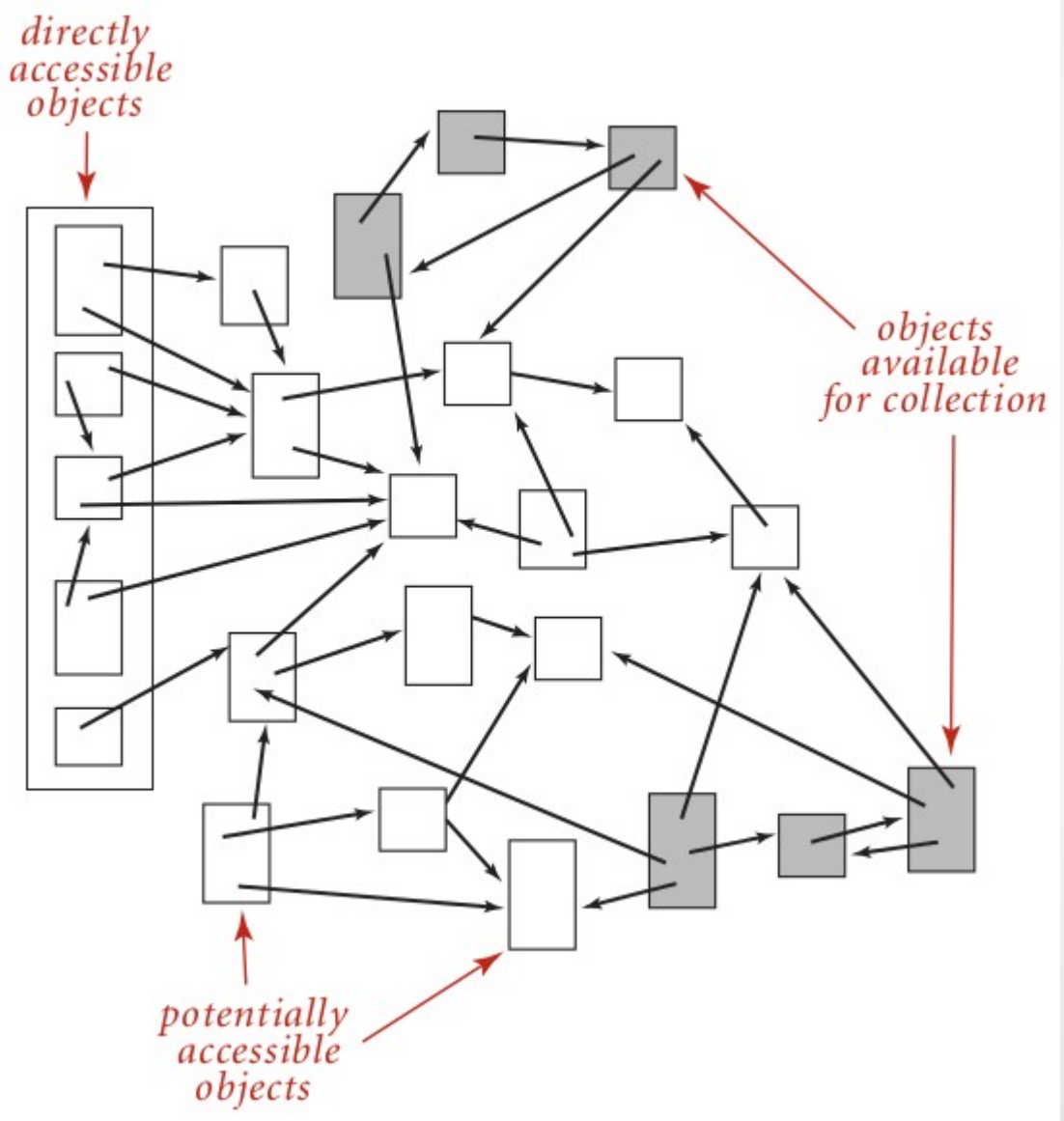
Directed Graphs

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



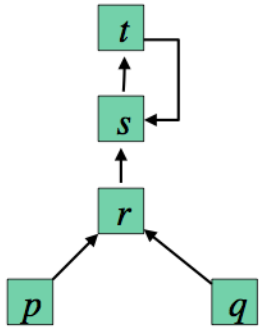
Game Description Language Example

OPTIONAL Coursera: General Game Playing by Michael Genesereth

Dependency Graph

The *dependency graph* for a set of rules is a directed graph in which (1) the nodes are the relations mentioned in the head and bodies of the rules and (2) there is an arc from a node *p* to a node *q* whenever *p* occurs with the body of a rule in which *q* is in the head.

```
r(X,Y) :- p(X,Y) & q(X,Y)
s(X,Y) :- r(X,Y)
s(X,Z) :- r(X,Y) & t(Y,Z)
t(X,Z) :- s(X,Y) & s(Y,X)
```



A set of rules is *recursive* if it contains a cycle. Otherwise, it is *non-recursive*.

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?

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.

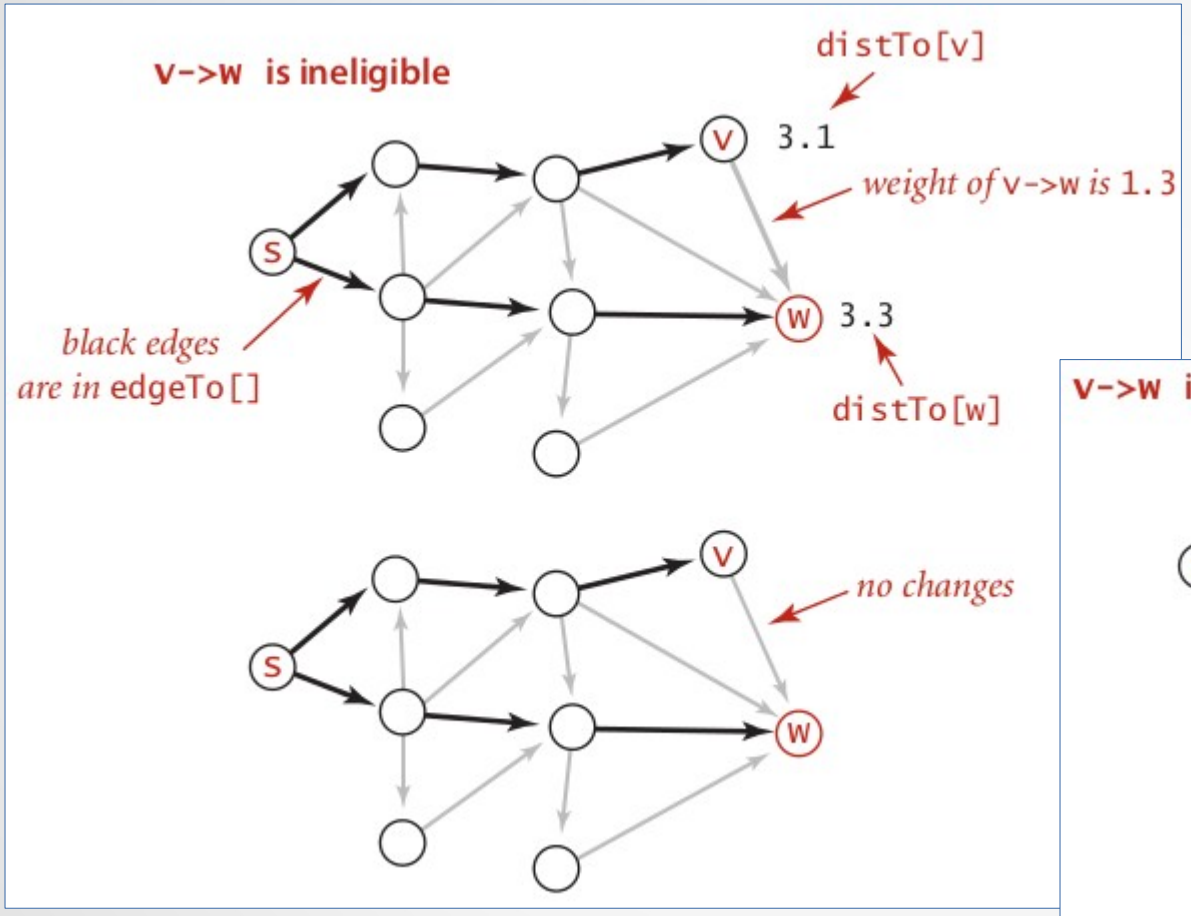
```
private void relax(WeightedDigraph graph, int v) {
    for( WeightedDiedge edge : graph.neighbors(v) ) {
        int w = edge.to();
        // Is better to go through v to get to w
        if ( distTo[w] > distTo[v] + edge.weight() ) {
            // Update distance to w and remember edge
      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

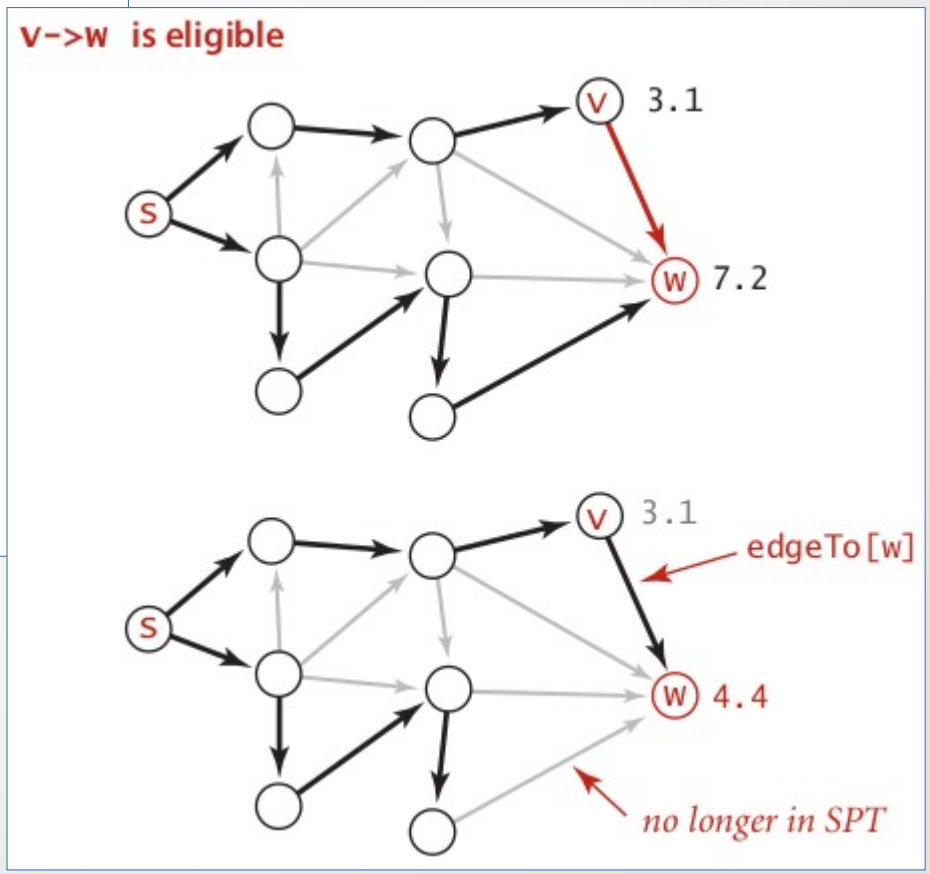
Generic Shortest Path Tree (SP) Algorithm

- 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

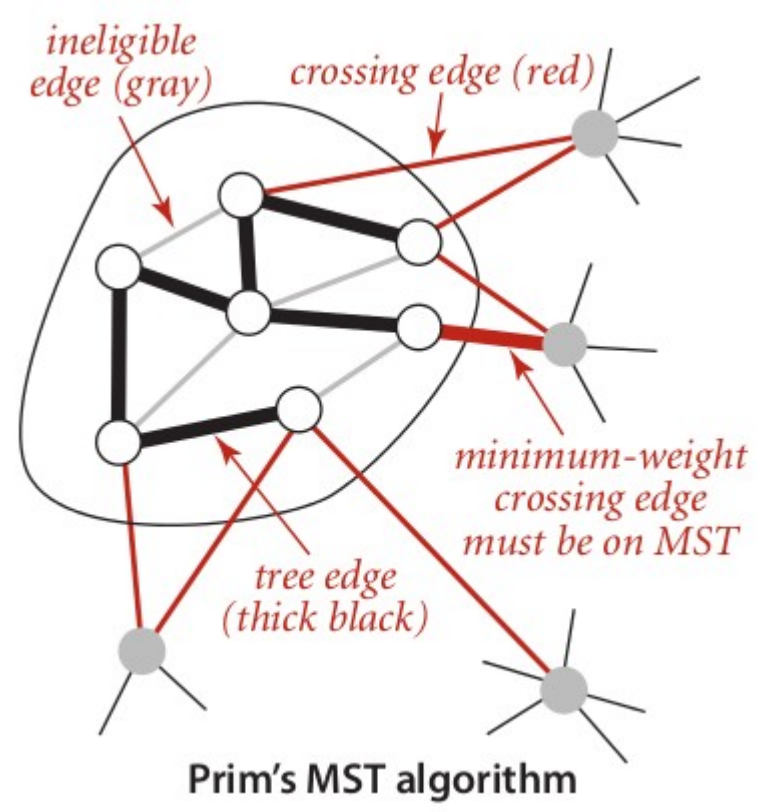
- Use a Prim-like idea. Works only with non-negative edge weights.

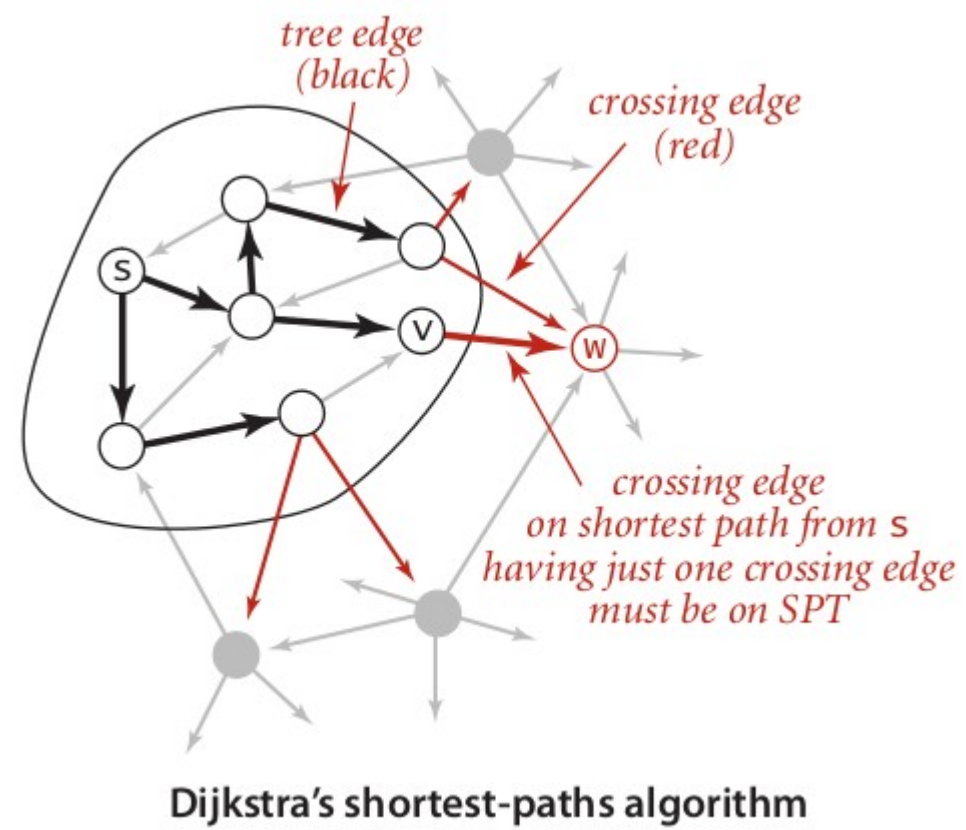
```
Given: weighted directed graph, source vertex s

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





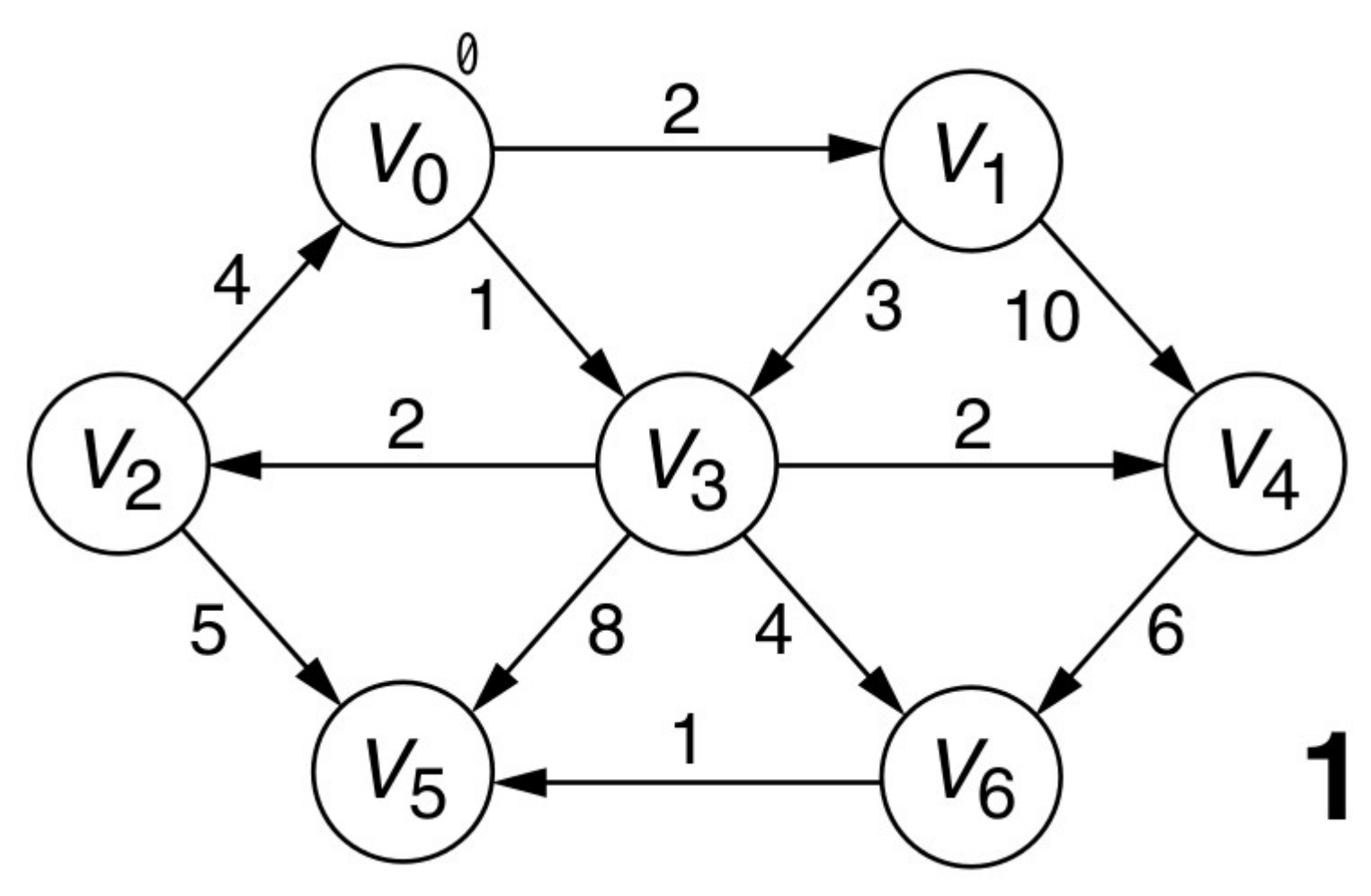
Dijkstra's SP Algorithm Analysis

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.

Equivalence Relations and Disjoint Sets

Weiss 24.1

- A relation is defined on a set if every pair of elements either is related or is not. An equivalence relation is reflexive, symmetric, and transitive. Given relation R:
  - *Reflexive*: a R a
  - *Symmetric*: a R b, b R a
  - *Transitive*: if a R b and b R c, then a R c
- The *equivalence class* of an element x in set S is the subset of S that contains all the elements related to x. The equivalence classes form disjoint sets with two fundamental operations:
  - Find: Returns the name of the set (equivalent class)
  - Union: Combines two sets together into one set

Union-Find (UF) Data Structure

Weiss 24.2

- The union-find data structure is used to carry out these two (union and find) disjoint set operations.
- Union-find algorithms use the UF data structure to provide answers to each query dynamically.
- Union-find is an online algorithm - an answer must be provided for each operation before the next operation can be performed.
  - Answer to current find/union must be completed before next find/union can be performed.
  - An offline algorithm is provided the entire sequence of operations (union/find), can provide answers after seeing all operations.

Union-Find (UF) Applications

Weiss 24.1

- There are many applications, including:
  - Generating mazes
  - Minimum spanning trees (will primarily cover)
  - Nearest common ancestor
- How can UF help with Kruskal's MST? Need to dynamically test connectivity (“is connected to”). Two vertices in the same tree are connected (i.e. same connected component).
  - Need to do this dynamically, trees will be constantly updated throughout Kruskal's MST Algorithm
  - Insight: dynamic connectivity is an equivalence relationship!

Dynamic Connectivity

Sedgewick/Wayne 1.5

- The “is connected to” is an *equivalence* relationship.
  - *Reflexive*: a is connected to a
  - *Symmetric*: a is connected to b, b is connected to a
  - *Transitive*: if a is connected to b and b is connected to c, then a is connected to c
- Within the context of our weighted graph
  - Equivalence class is a connected component
  - Union two vertices combines their components
  - Find the component identifier of a vertex. Two vertices that are connected will have the same component identifier.

Union-Find API

```
public interface UF
{
    // Constructor initializes N components
    // public UF(int N);

    // Add connection between a and b.  Merges components if the
    // two vertices are in different components.  Each successful
    // union decreases the number of components by one.
    public void union( int a, int b );

    // Finds and returns the component identifier for a [0,N-1]
    public int find( int a );

    // Determines if a and b are connected (i.e. in same component)
    // i.e. true if find(a) == find(b), false otherwise
    public boolean connected( int p, int q );

    // Returns the number of components.
    public int size();
}
```

UF Default Code

```
public class UF {
    private int[] ids;
    private int size;
    public UF(int N) {
        this.ids  = new int[N];
        for (int i = 0; i < ids.length; i++) {
            ids[i] = i; // Weiss sets this to -1
        }
        this.size = N;
    }

    public void union( int a, int b ) { ... }
    public int find( int a )          { ... }

    public int connected( int a, int b ) {
        return find(a) == find(b);
    }

    public int size( ) {
        return this.size;
    }
}
```

UF Quick-Find Idea

Sedgewick/Wayne 1.5

- The data structure needs to keep track of components
  - Each component has an integer identifier. Can be any vertex identifier that is in the component.
- Efficiently implementing the operations primarily requires that the union and find operations are efficiently implemented.
  - Connected simply uses find operation
  - Size can be accomplished with size variable
- Quick Find idea: Keep int[] array (denoted *ids*) where each vertex identifier is an index in ids and the value stored is the component identifier for that vertex.

UF Quick-Find Code

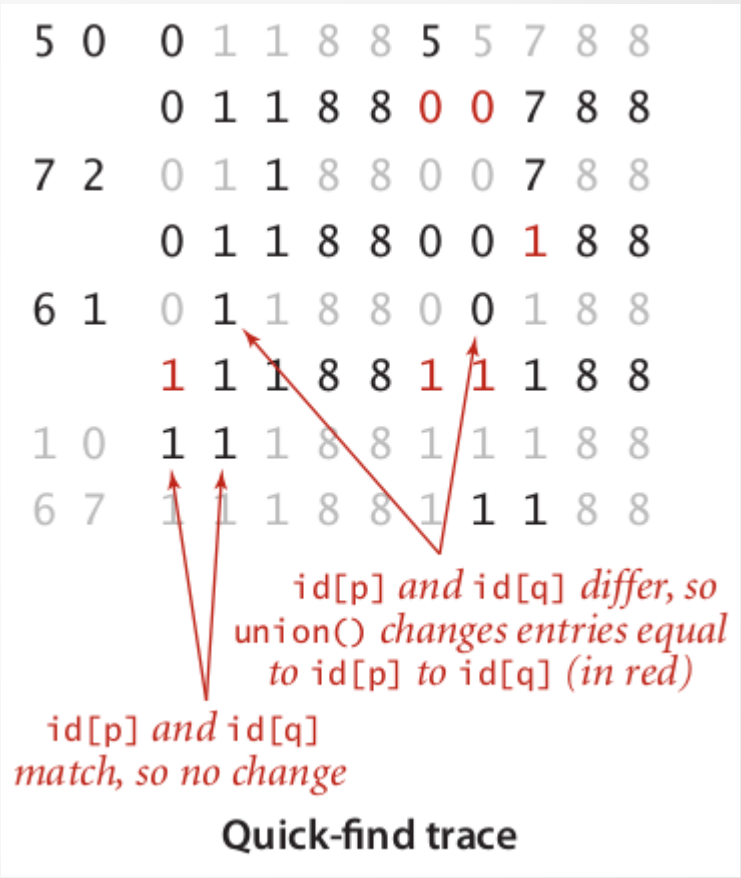
Sedgewick/Wayne 1.5

```
public class UFQuickFind {
    ...
    public void union( int a, int b ) {
        int aCid = this.find(a);
        int bCid = this.find(b);

        // do nothing, already in same component
        if( aCid == bCid ) return;

        // Arbitrarily use b's component identifier to add a's items to
        for (int i = 0; i < ids.length; i++) {
            if( this.ids[i] == aCid ) this.ids[i] = bCid;
        }

        this.size--; // Union combines two into one
    }

    public int find( int a ) {
        return ids[a];
    }
    ...
}
```

UF Quick-Find Example

Sedgewick/Wayner 1.5

		id[]									
p	q	0	1	2	3	4	5	6	7	8	9
4	3	0	1	2	3	4	5	6	7	8	9
		0	1	2	3	3	5	6	7	8	9
3	8	0	1	2	3	3	5	6	7	8	9
		0	1	2	8	8	5	6	7	8	9
6	5	0	1	2	8	8	5	6	7	8	9
		0	1	2	8	8	5	5	7	8	9
9	4	0	1	2	8	8	5	5	7	8	9
		0	1	2	8	8	5	5	7	8	8
2	1	0	1	2	8	8	5	5	7	8	8
		0	1	1	8	8	5	5	7	8	8
8	9	0	1	1	8	8	5	5	7	8	8



UF Quick-Find Analysis

- Find operation is constant, just need to access array once (why this is called quick-find).
- Union operation is not constant
  - What did we have to do in order to union? How much did this cost us? O(N)
- Given some sequence of M operations (where M is approximately N), split evenly
  - 1/2 N x find() + 1/2 N x union()
  - 1/2 N x O(1) + 1/2 N x O(N)
  - UF with Quick-Find is quadratic $O(N^2)$
  - Cannot have more than N-1 union operations, Why?

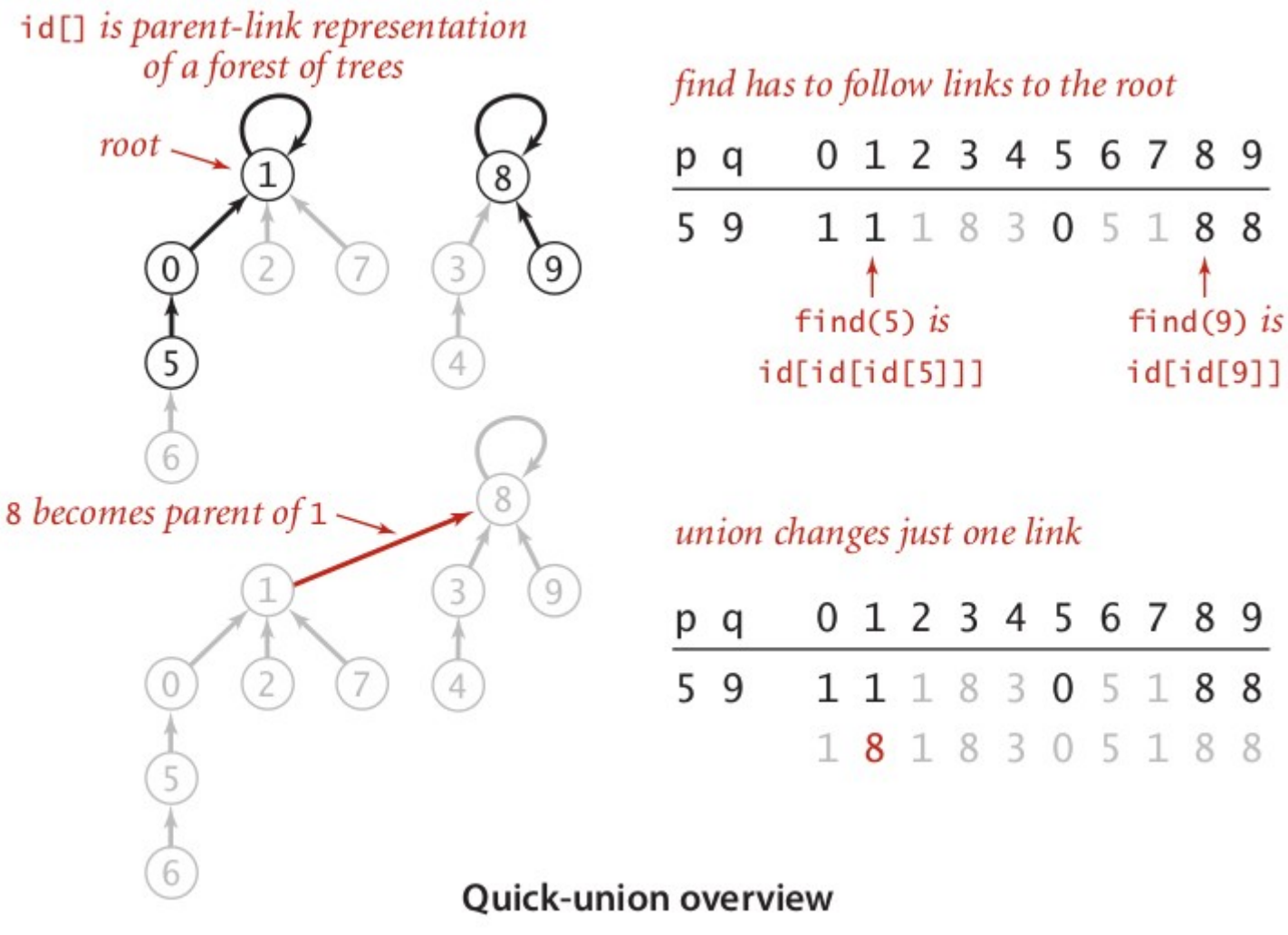
UF Quick-Union Idea

Sedgewick/Wayne 1.5

- Quick Union idea: The int[] ids can be used to represent a tree where each value is the parent. To union two elements, need to simply point the root of one tree to be a child of the other element's root tree.
  - Union simply changes a pointer
  - Find now needs to traverse the tree upwards to find the root of the tree (i.e. the component identifier)

UF Quick-Union Overview

Sedgewick/Wayne 1.5



Quick-union overview

UF Quick-Union Code

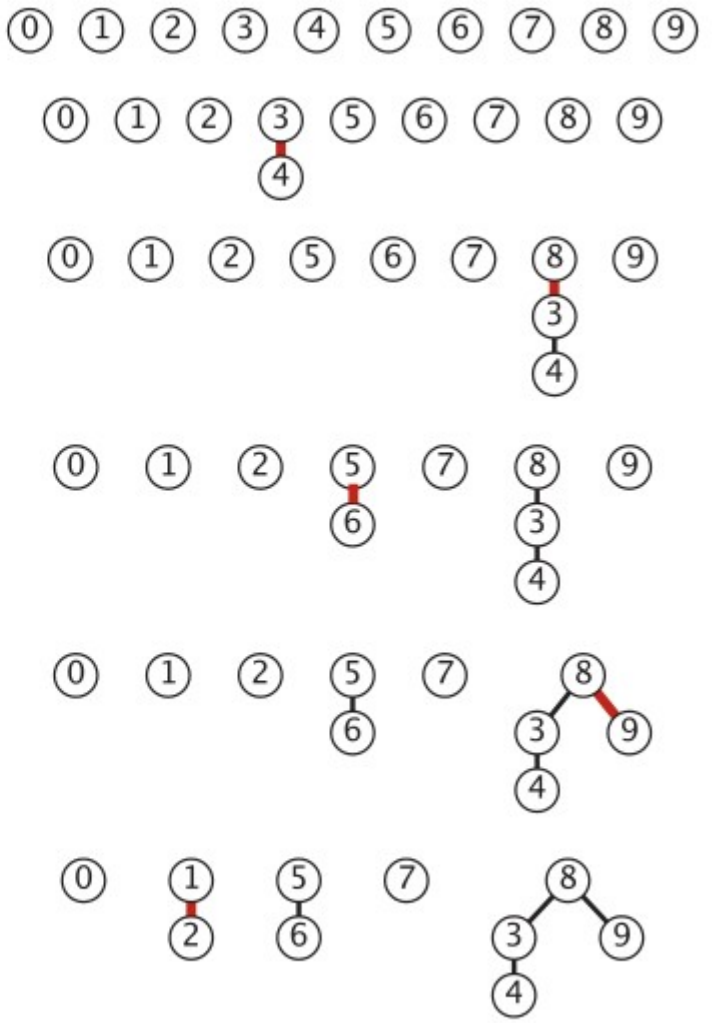
Sedgewick/Wayne 1.5

```
public class UFQuickUnion {
    ...
    public void union( int a, int b ) {
        int aRoot = this.find(a);
        int bRoot = this.find(b);

        if( aRoot == bRoot ) return; // already in same component

        // Arbitrarily add a's root component to b's root component
        this.ids[aRoot] = bRoot;

        this.size--; // Union combines two into one
    }

    public int find( int a ) {
        while( a != this.ids[a] )
        {
            a = this.ids[a];
        }
        return a;
    }
    ...
}
```

UF Quick-Union Example 1/2

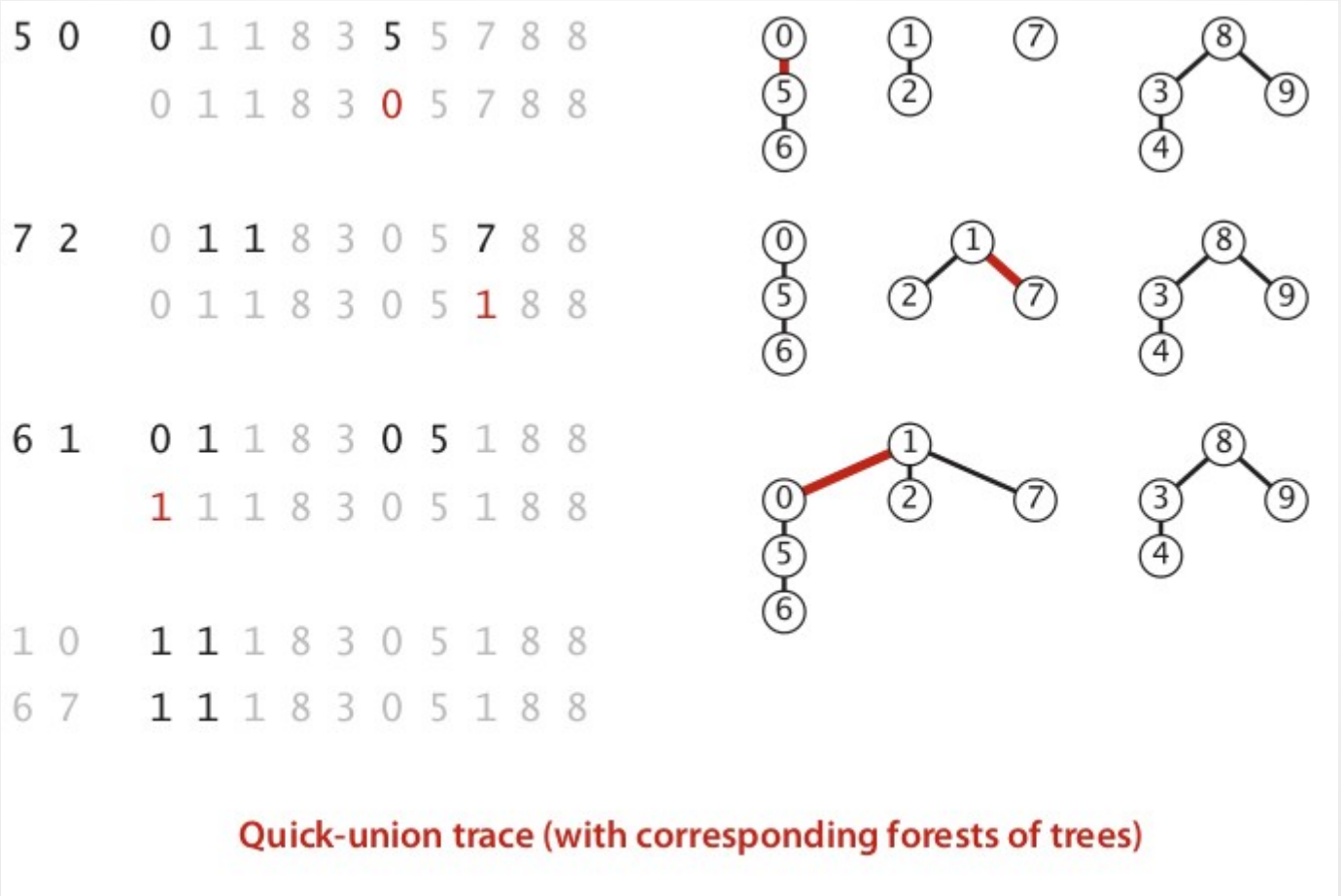
Sedgewick/Wayner 1.5

		id[]									
p	q	0	1	2	3	4	5	6	7	8	9
4	3	0	1	2	3	4	5	6	7	8	9
		0	1	2	3	3	5	6	7	8	9
3	8	0	1	2	3	3	5	6	7	8	9
		0	1	2	8	3	5	6	7	8	9
6	5	0	1	2	8	3	5	6	7	8	9
		0	1	2	8	3	5	5	7	8	9
9	4	0	1	2	8	3	5	5	7	8	9
		0	1	2	8	3	5	5	7	8	8
2	1	0	1	2	8	3	5	5	7	8	8
		0	1	1	8	3	5	5	7	8	8
8	9	0	1	1	8	3	5	5	7	8	8



UF Quick-Union Example 2/2

Sedgewick/Wayner 1.5



UF Quick-Union Analysis

- Find operation is worse case O(N). May get tree that is a linked list. How could this happen?
- Union operation calls find twice but otherwise is constant.
- Given some sequence of M operations (where M is approximately N), split evenly
  - 1/2 N x find() + 1/2 N x union()
  - 1/2 N x O(N) + 1/2 N x O(2N)
  - UF with Quick-Union is quadratic O(N^2)
- Not really any better than Quick-find
  - However, new idea based on quick-union

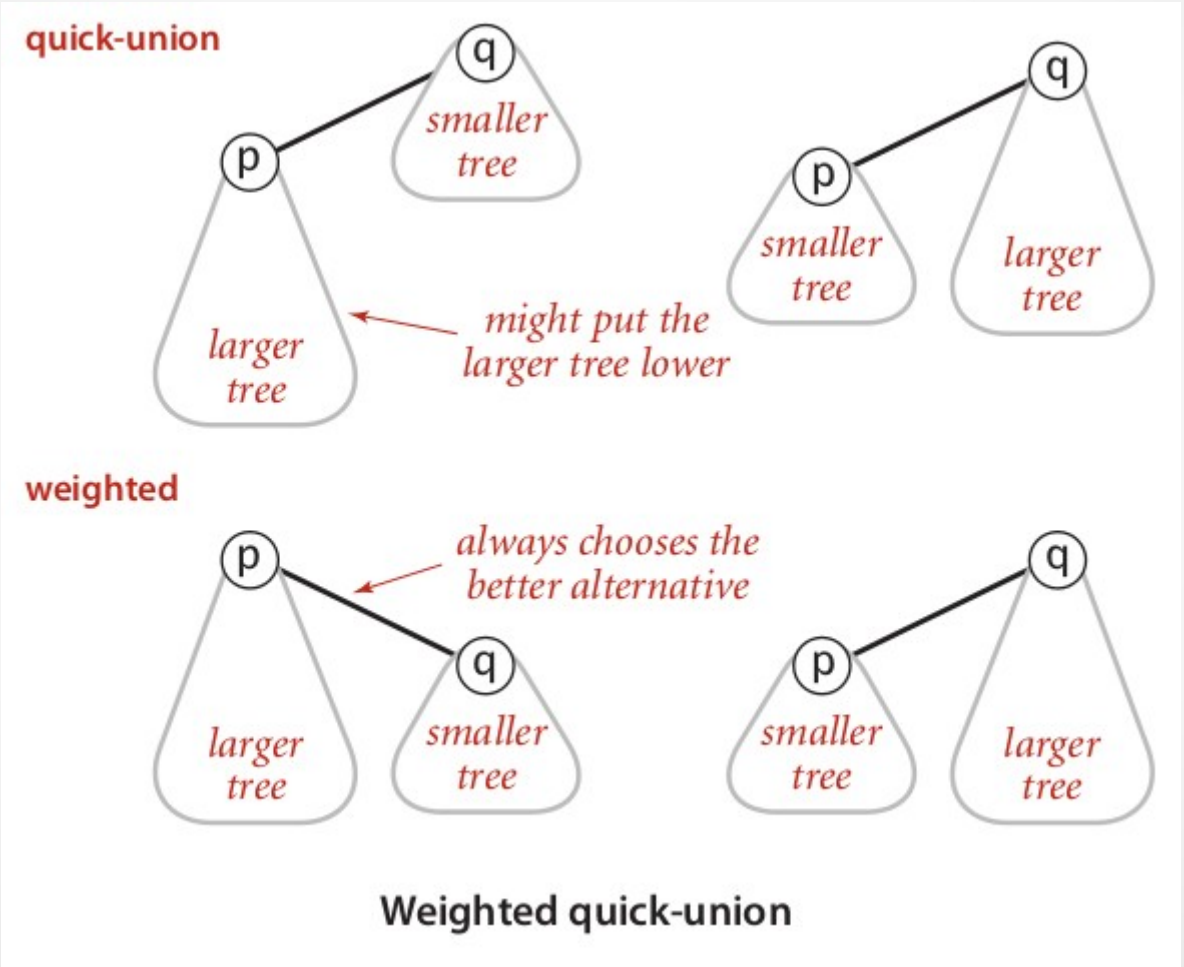
UF Weighted Quick-Union Idea

Sedgewick/Wayne 1.5

- Weighted Heuristic: Same as quick-union but make sure we avoid the linked list scenario.
  - When combining trees, do not just arbitrarily pick one to add to the other – add the smaller tree (with fewer items) to the larger tree.
  - The only way to increase the height of a tree is when the two trees are the same size, then can pick one to the child arbitrarily, and a new tree will be created that is one more than the previous height.
  - Need a new array to maintain the size of the trees.

UF Weighted Quick-Union Overview

Sedgewick/Wayne 1.5



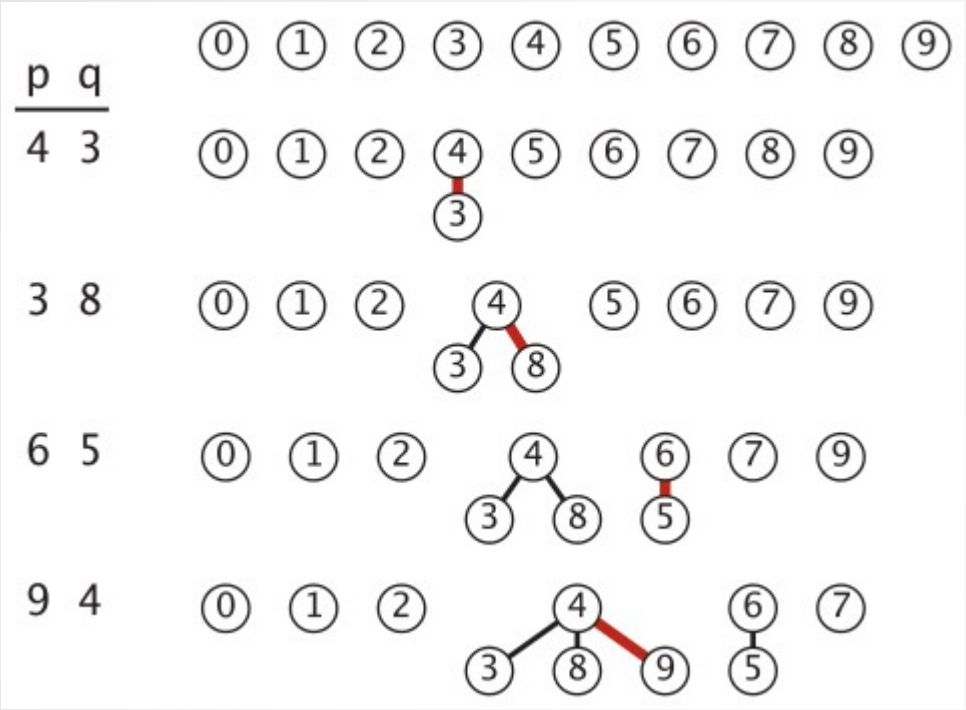
UF Weighted Quick-Union Code

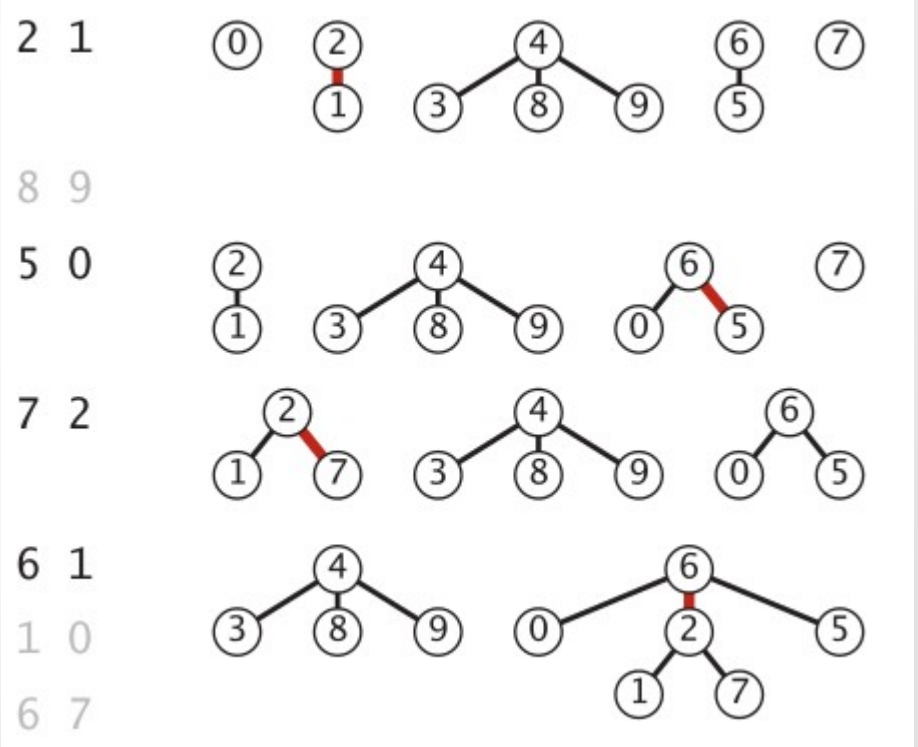
Sedgewick/Wayne 1.5

```
public class UFWeightedQuickUnion {
    ...
    public void union( int a, int b ) {
        int aRoot = this.find(a);
        int bRoot = this.find(b);

        if( aRoot == bRoot ) return; // already in same component

        // Make smaller root point to larger one.
        if(sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
        else              { id[j] = i; sz[i] += sz[j]; }

        this.size--; // Union combines two into one
    }

    public int find( int a ) {
        while( a != this.ids[a] )
        {
            a = this.ids[a];
        }
        return a;
    }
    ...
}
```

UF Weighted Quick-Union Example 1/2

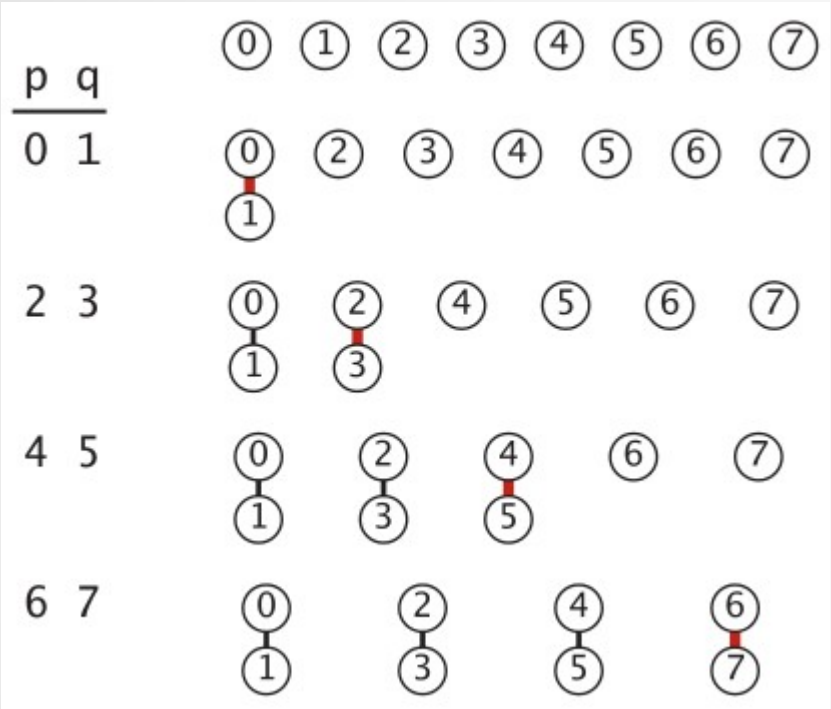
Sedgewick/Wayner 1.5

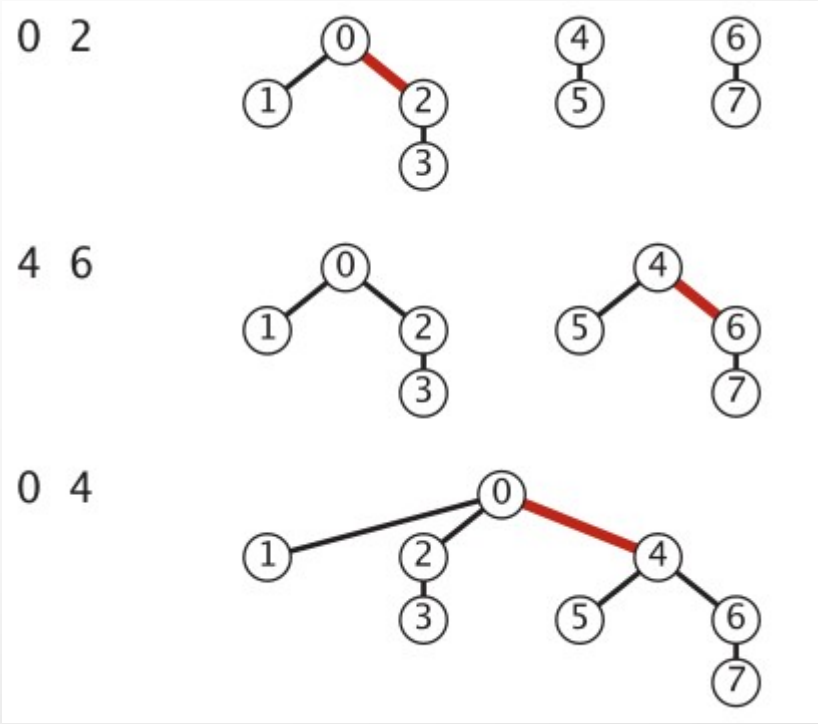




UF Weighted Quick-Union Example 2/2

Sedgewick/Wayner 1.5





UF Weighted Quick-Union Analysis

- Find and union are determined by the maximum height of any tree. Due to the weighted heuristic:
  - Height of a tree with 2^N elements is n
  - Union produces new tree 2^{N+1} items (maximum) with new height of n+1
  - Thus, find and union expected logarithmic
- Given some sequence of M operations (where M is approximately N), split evenly
  - 1/2 N x find() + 1/2 N x union()
  - 1/2 N x O(lg(N)) + 1/2 N x O(2lg(N))
  - UF with Weighted Quick-Union is quadratic O(N lg(N))

UF Weighted QU w/Path Compression

- Path-compression Heuristic: When performing find that traverses the tree, modify the tree to make it shorter.
  - Does not cost any extra and takes constant time, only one or two extra lines of code, still correct operations
  - Analysis is beyond scope but takes *iterated logarithmic* time (similar growth to *inverse Ackermann Function*)
- Iterated logarithm (denoted lg*(N)): given positive number N, the *number of times* necessary to take the logarithm so that the number is reduced <= 1.
  - $lg^*(2^4) = lg(lg(lg(2^4))) = 3$
  - $lg^*(2^{16}) = 4$
  - $lg^*(2^{65536}) = 5$, practically never more than 5, constant

Path Compression Variant Implementations

- Two-pass implementation: add second loop to find() to set the ids[] of each examined node to the root.
- Simpler one-pass variant: Make every other node in path point to its grandparent (thereby halving path length).

```
public class WQUPC {
    ...
    public int find( int a ) {
        while( a != this.ids[a] )
        {
            this.ids[a] = this.id[ id[a] ]; // PC: One pass variant
            a = this.ids[a];
        }
        return a;
    }
    ...
}
```

Union-Find Summary

Sedgwick 1.5

- Typical to implement weighted union and path compression. Both heuristics are easy to implement.
  - Weighted union-find with path compression is nearly optimal as a constant time algorithm has been proven not to exist.

Algorithm	Constructor	union	find
Quick-Find	N	N	1
Quick-Union	N	Tree height	Tree height
Weighted QU	N	lg(N)	lg(N)
Weighted QU w/PC	N	~1*	~1*
Impossible	N	1	1

* Very, very , very nearly one, amortized, close to best that can be done for problem

Kruskal's MST Algorithm Revisited

- Based on weighted union-find w/ path compression, union, find, and connected operations are constant O(1)
  - Makes Kruskal linearithmic O(E lg (E))

```
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);
    }
}
```

Questions?

PA11

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

Free Question Time!