INFS 519 – Fall 2015

Program Design and Data Structures

Lecture 11

Instructor: James Pope

Email: jpope8@gmu.edu

Today

- Review Last Class
  - Hashing
  - Linear Probing/Quadratic Probing
  - Separate Chaining
  - Wrap up Symbol Tables
- Schedule
  - Graphs / Motivation
  - Data Structure Representations
  - Basic graph searching algorithms: BFS and DFS

Collision Resolution

- Several ways to address, most common are:
  - Separate Chaining
  - Open Addressing
    - Linear Probing
    - Quadratic Probing
- Commonly refer to index positions as cells. Number of keys in table is N, number of cells (i.e. array.length) is M.
- Separate Chaining
  - Linked list attached to each cell in hash table
- Open addressing
  - Use empty cells to resolve collision

Equals and HashCode Methods

- Requirement for equals and hashCode to be consistent:

  If two objects are equal

  Then must generate same hash code

- Reverse is not true, two objects with same hashCode may not be equal (i.e. they will collide when put into a hash table)
- Always override hashCode when you override the equals method (otherwise they are most certainly not consistent).

Linear Probing Put Operation

Weiss 20.3

- When a collision occurs, sequentially search from collision index to next index to find empty cell
  - If at end of array, wrap around to beginning
  - Care taken to not fill array full, otherwise infinite loop
- Naive analysis
  - Worse case insert would be N
  - Average case would be 1/2 N
- Fortunately, in practice
  - Performs much better

Linear Probing

Figure 20.5

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	*After insert 18*	*After insert 49*	*After insert 58*	*After insert 9*
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Linear Probing Get Operation

Weiss 20.3

- Get (find) follows similar approach
  - If collision, search sequentially until item found
  - Naive analysis similar to put
- Results in cluster of entries in sequential/contiguous order
- Direct implementation of delete not possible
  - Need the contiguous cluster, otherwise break implicit list and cannot access later items
- Delete operation workaround
  - Deleted items are marked (denoted “tombstone”), perhaps later removed during a rehash

Linear Probing Put Non-clustering Analysis

Weiss 20.3.2

- Ignoring clustering, number of probes before finding an empty cell

  $1 / (1 – \lambda)$, e.g. $\lambda = 0.5$, then 2 probes expected

- Unfortunately, this is still incorrect
  - Primary clustering where large blocks of occupied cells are formed.
  - Subsequent collisions add to size of cluster.
  - Put operations are not independent of each other.

Linear Probing Put Clustering Analysis

Weiss 20.3.2

- Clustering analysis very difficult, fortunately Knuth solved

  $1/2 (1 + 1 / (1 – \lambda)^2)$

- Keeping the table's load factor around 0.5 is good
- As the load factor increases, expected number of probes increases dramatically. High load factor results in O(N) for both put and get operations.

Linear Probing Put Analysis Comparison

Weiss 20.3.2

- Cluster and non-cluster analysis roughly agree when λ is below 0.5

λ	Cluster #probes	Non-cluster #probes
0.10	1.12	1.11
0.20	1.28	1.25
0.30	1.52	1.43
0.40	1.89	1.67
0.50	2.50	2.00
0.60	3.63	2.50
0.70	6.06	3.33
0.80	13.00	5.00
0.90	50.50	10.00
0.95	200.50	20.00

Linear Probing Rehashing

- For linear probing, low load factor is desirable. Load factor increases when new entries are inserted.
- When the hash table is created created, we generally do not know how many entries might be inserted.
  - Guess at M? Make arbitrarily large?
- How can we solve this?
  - Rehashing: Dynamically expand the hash table increasing M thereby reducing the load factor λ
  - Have to use put in new hash table because the hash function has changed
  - Classic trade off of using more memory to get better performance

Quadratic Probing

Weiss 20.4

- Quadratic probing avoids primary clustering by inserting at points away from the initial hashed index.
  - Increases quadratically, say we hash to index H, then if a collision occurs, we examine $H + i^2$, for increasing i
  - The first few cells examines, other than the initial cell, would be 1, 4, 9, 16, 25, 36, 49, etc.
- Is it possible to probe a cell twice? Is it possible to never insert even when cells are empty?
  - Yes and yes. Consider M=4 with items at 0,1. Insert an item that hashes to 0
  - Can solve by making M prime

Quadratic Probing Guarantee

Weiss Theorem 20.4

- Theorem 20.4: Using quadratic probing, if table is at most half full **and** the M is prime, then a new element can always be inserted.
  - No cell is probed twice
- To keep these guarantees, we need to dynamically expand the table when λ exceeds 0.5.
  - We cannot simply double the size, won't be prime
  - Can use function to calculate the next prime number from the current prime number efficiently (Figure 20.8)

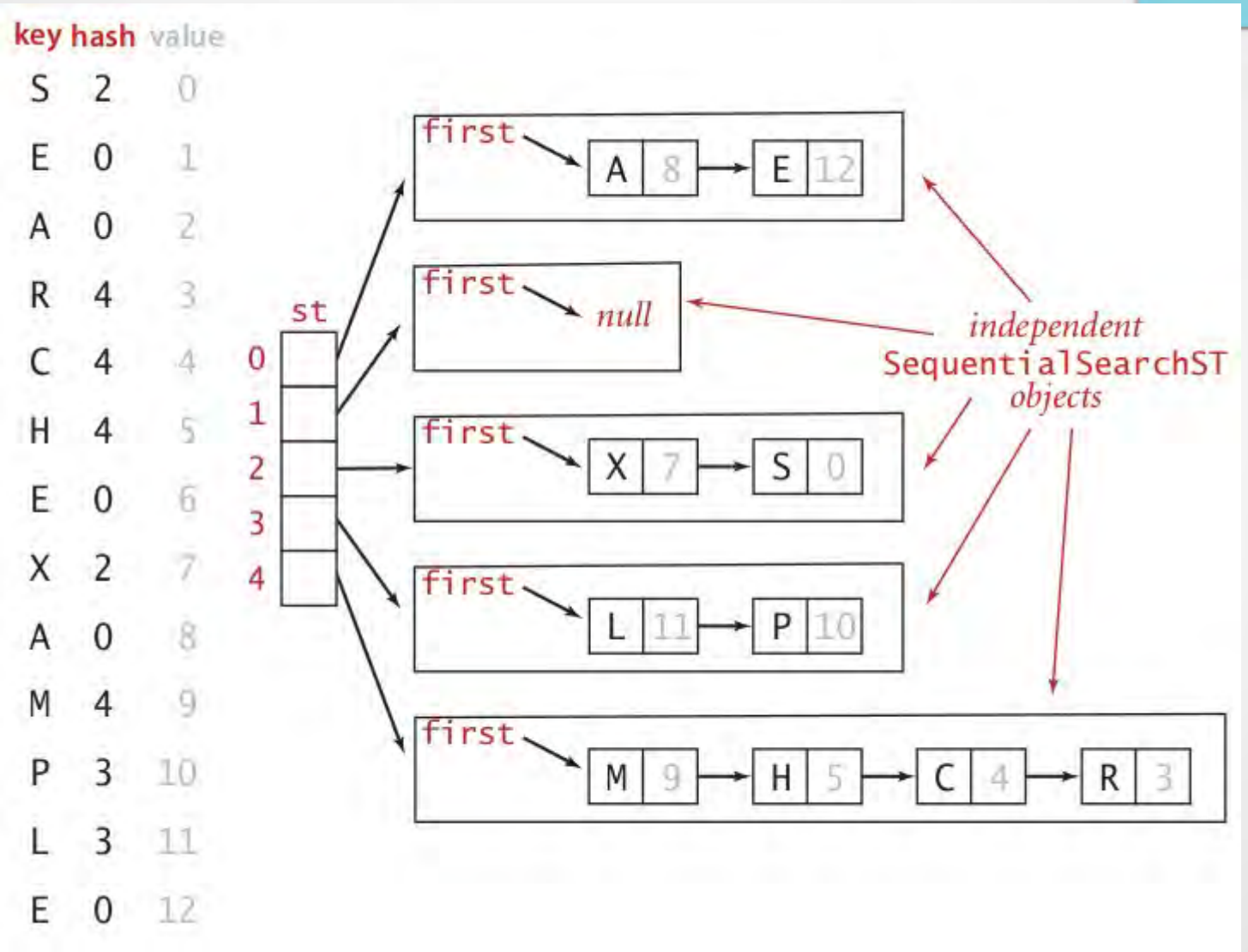
Separate Chaining

Weiss 20.5

- Space efficient alternative to quadratic probing.
- Each cell maintains linked list
  - Less sensitive to high load factors
  - Load factor λ allowed to increase beyond 1.0
- Now searching a linked list
  - This was bad? O(N)
  - Yes. However, can control length of the linked list
- Assuming uniform hashing function
  - Each list expected to have N / M entries, i.e. λ

Separate Chaining Example

Sedgewick 3.4



Separate Chaining Worse Case Analysis

Sedgewick 3.4

- In the worse case, the length of a list could be N
  - Put and get are N
- What about average case?
- Uniform hashing assumption
  - Uniformly and independently distribute keys between 0 and M-1 cells
- Given this, we expect each list to have N / M entries
  - How far from this expected list length?

Separate Chaining Table Size M

- What table size should we use? When to rehash?
  - Low load factor does not necessarily increase performance
  - High load factor acceptable (even about 1.0) can can save memory
- Java collections uses load factor of 0.75 and will rehash
  - “Offers a good trade off between time and space costs” (HashMap)

Hash Table Summary

- Open Addressing
  - Delete not easily supported
  - Linear Probing analysis well understood, primary clustering a problem.
  - Quadratic probing solves primary clustering. Harder to implement, requires M prime.
- Separate Chaining
  - Generally more memory efficient
  - Easy to implement, including delete
- In all cases, can adjust hash table size (rehashing) to get average case O(1) for put and get operations.
  - For separate chaining, make M close to N

Symbol Table Summary

- Rule of thumb: Generally use hash table unless guaranteed performance or need ordered operations

Implementation	Worse-Case		Average-Case		Order Ops	remarks
	Search	Insert	Search	Insert		
Unordered List	N	N	N	N	No	
Ordered Array	lg N	N	lg N	N	Yes	
BST	N	N	lg N	lg N	Yes	Easy
AVL	lg N	lg N	lg N	lg N	Yes	Easy
Red-Black	lg N	lg N	lg N	lg N	Yes	Often Used*
HT Chaining	**N**	**N**	**N / M**	**N / M**	**No**	Often Used*
HT Probing	**N**	**N**	**1**	**1**	**No**	

* Good constants and relatively easy to implement, used in many libraries

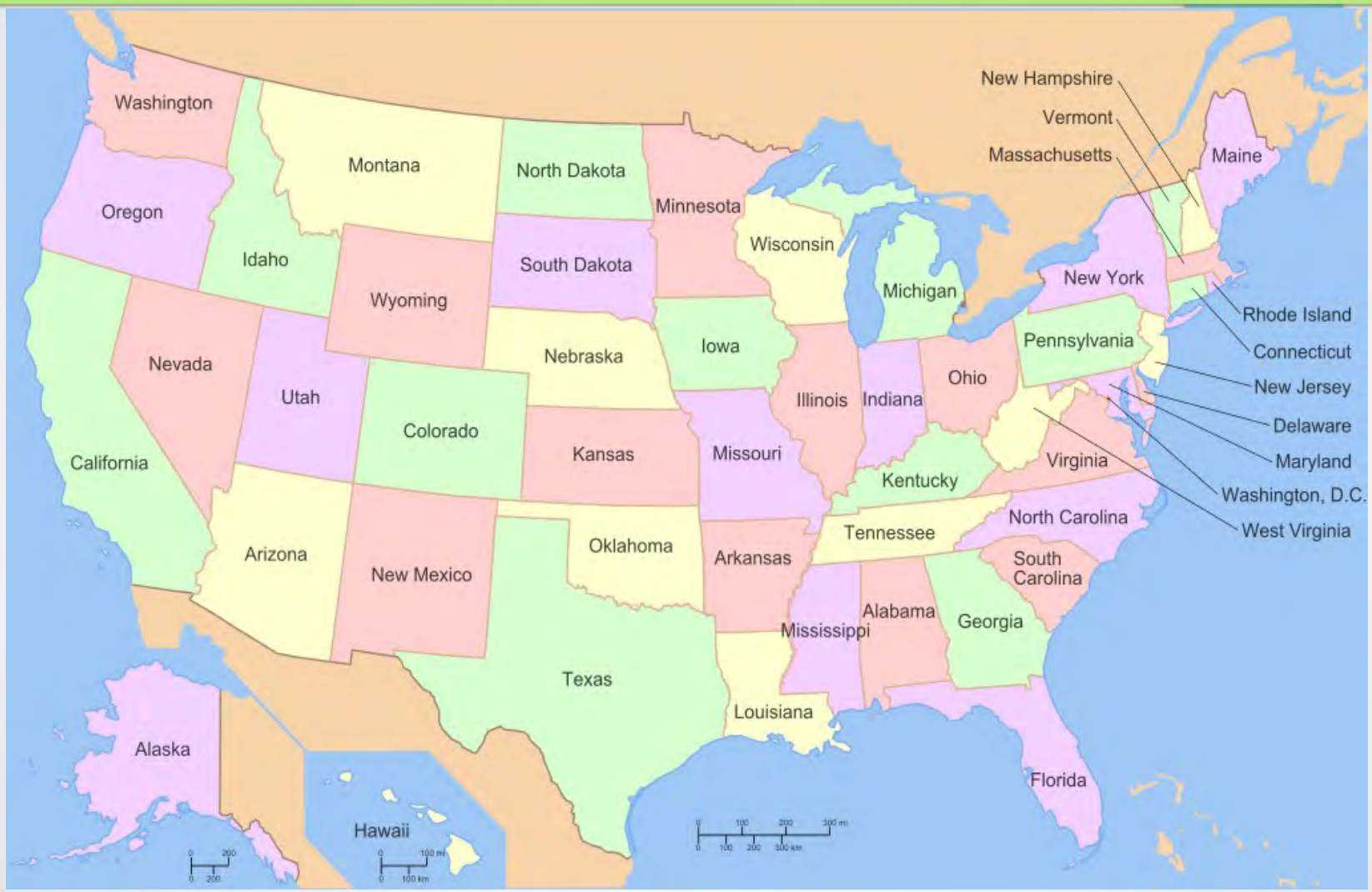
Graphs

- 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 element 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 1

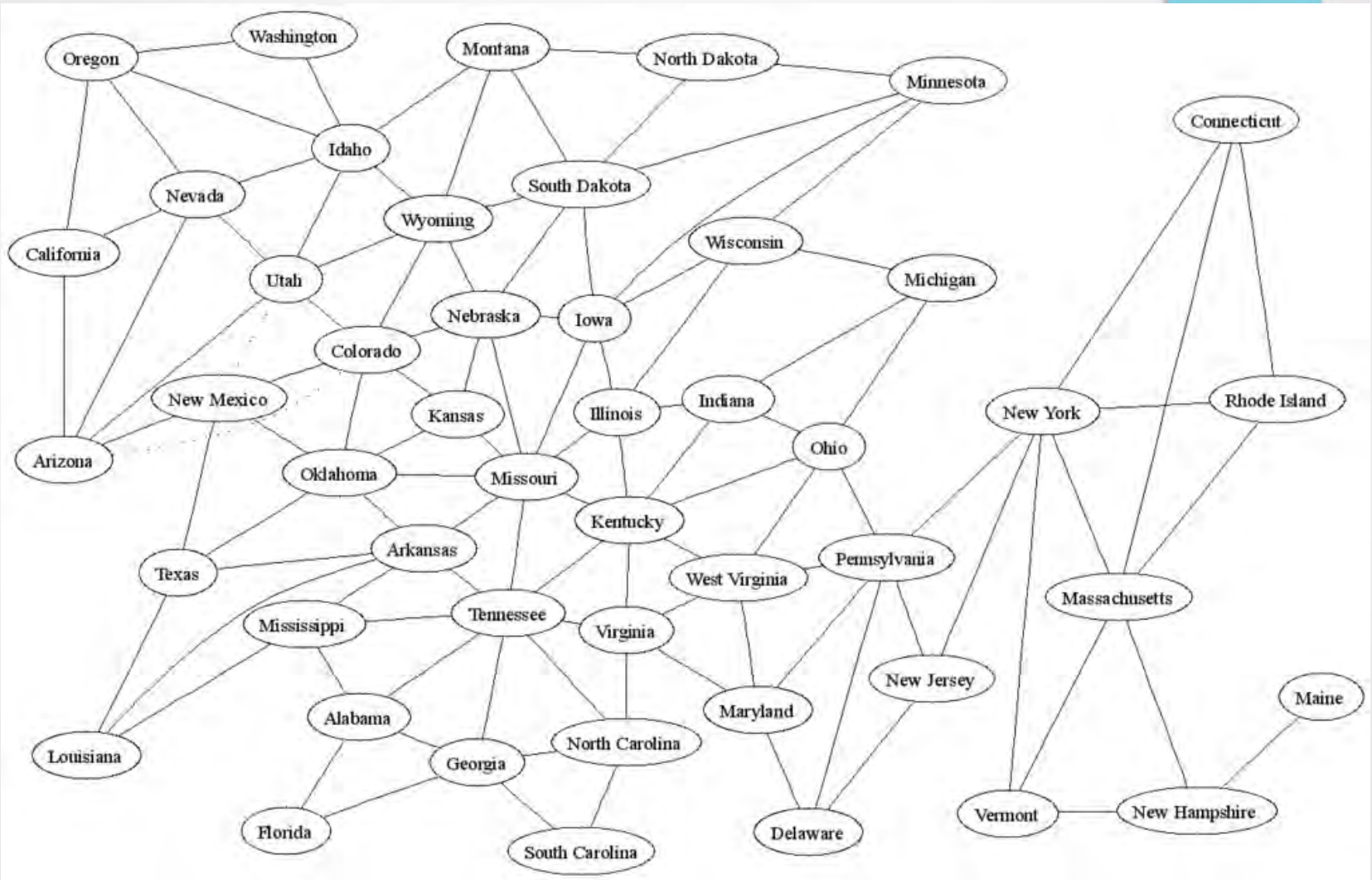
http://www.facebook.com/notes/facebook-engineering/visualizing-friendships/469716398919

Graph Applications 2A



Source: http://en.wikipedia.org/wiki/File:Map_of_USA_with_state_names_2.svg

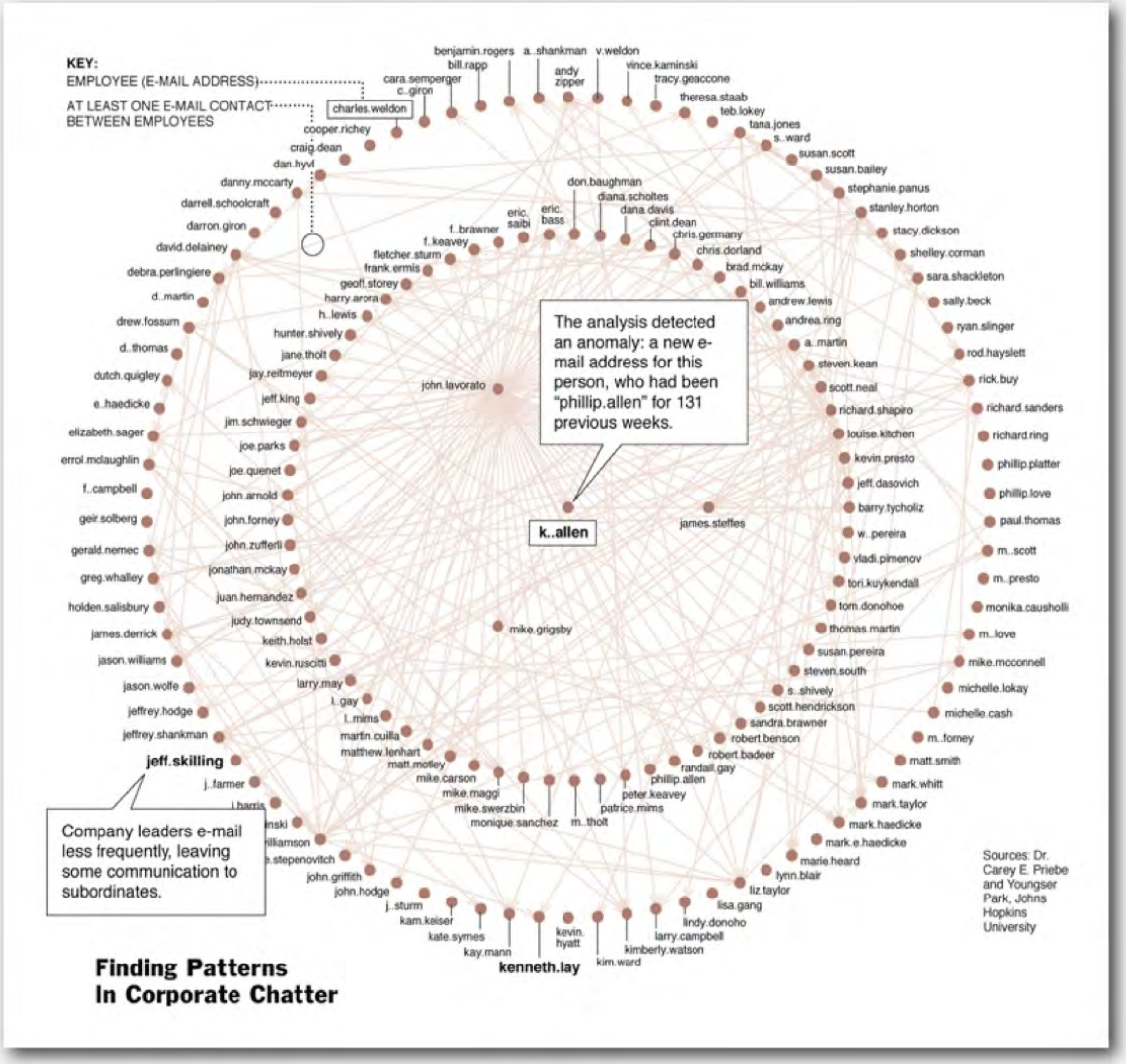
Graph Applications 2B



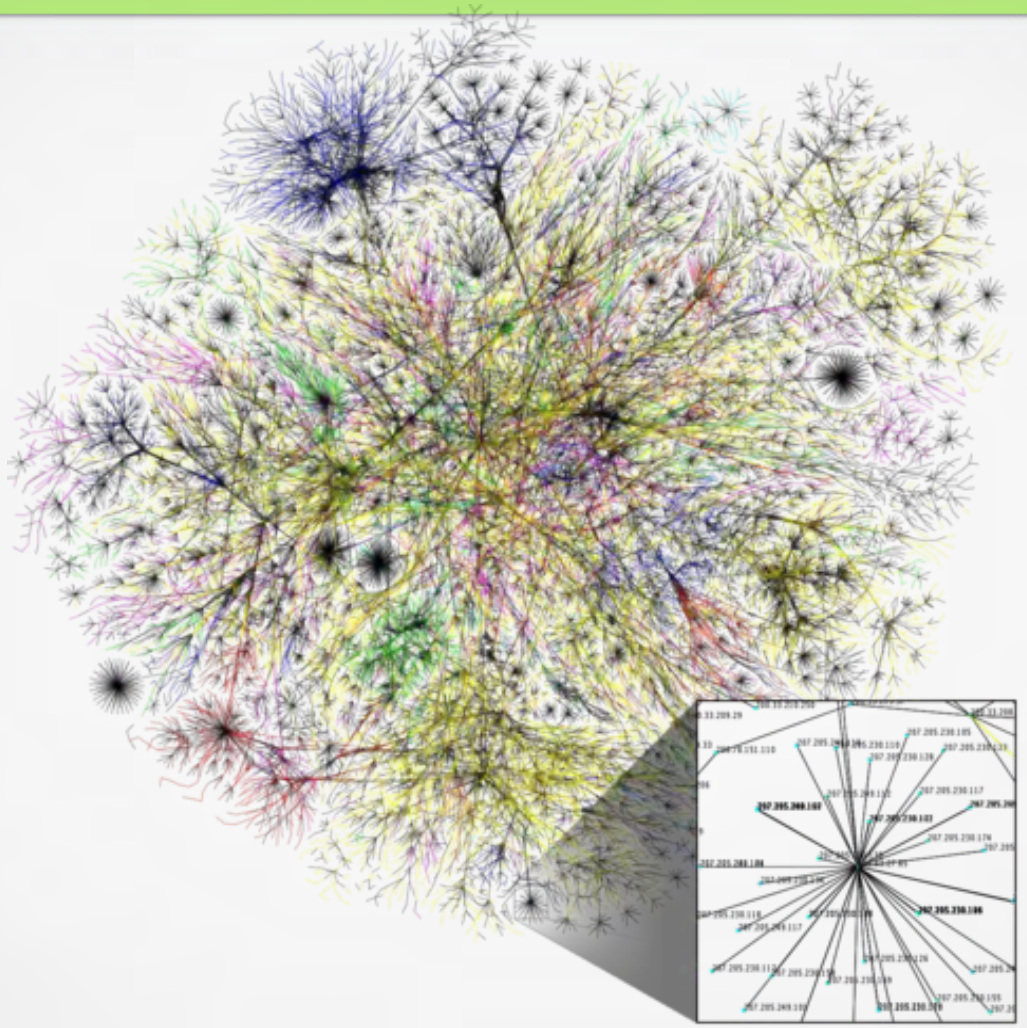
Source: http://en.wikipedia.org/wiki/File:UnitedStatesGraphViz.png

Graph Applications 3

One week of Enron emails



Graph Applications 4



"Internet map 1024 - transparent, inverted" by The Opte Project - Originally from the English Wikipedia
From: https://en.wikipedia.org/wiki/Internet

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

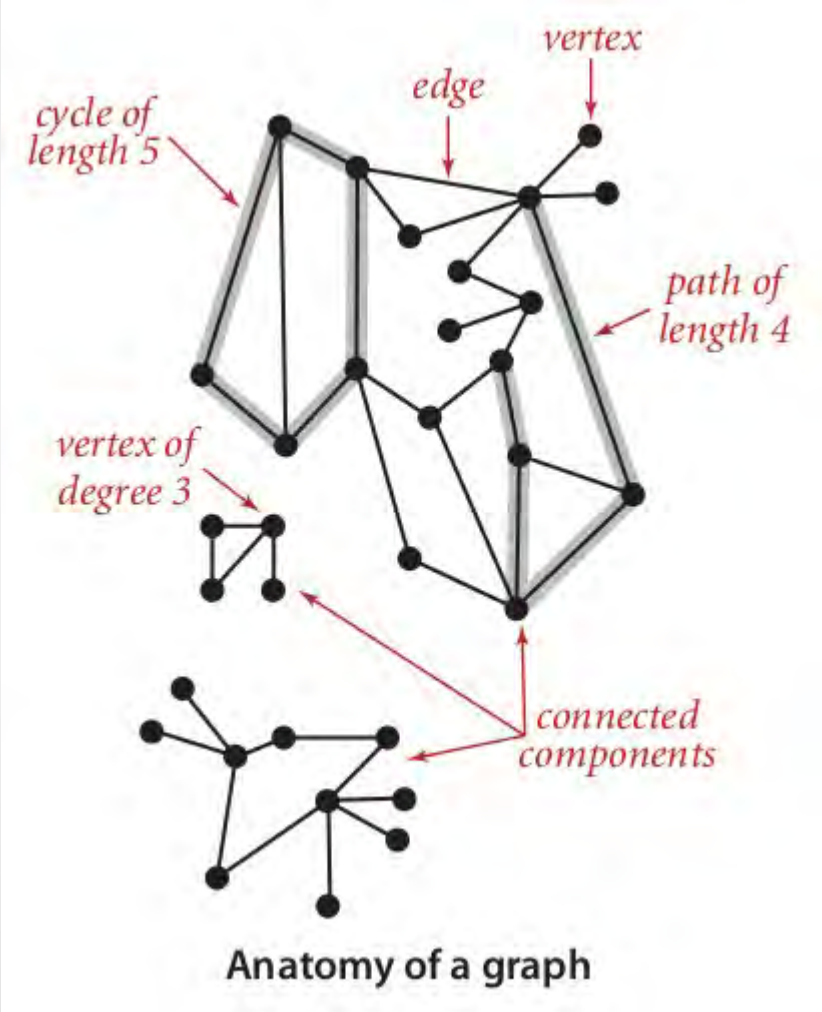
Graphs – Terminology 1

- 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

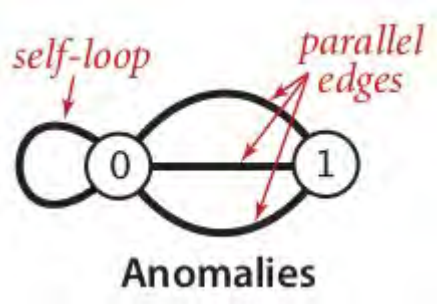
Graphs – Terminology 2

- Simple path where all vertices are distinct except the first and last can be the same
- Cycle in a directed graph is path that begins and ends at the same vertex and contains at least one edge
- Directed Acyclic Graph (DAG) digraph with no cycles
- Two vertices are adjacent is there is an edge connecting them together, the edge is incident to both vertices
- The degree of a vertex is number of edges incident to it
- Anomalies will allow but not use
  - Parallel edges are two edges connected same vertices
  - Self-loop is an edge connecting vertex to itself

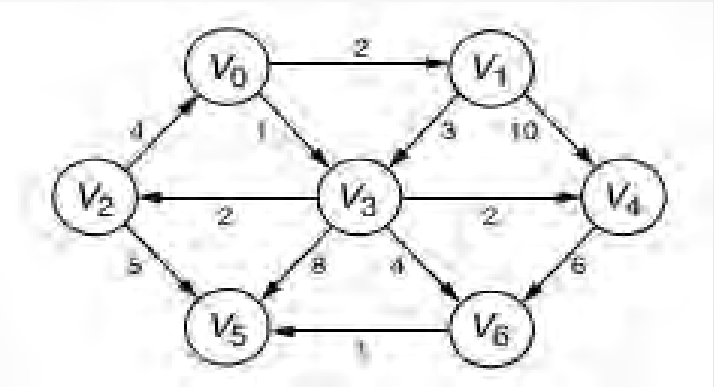
Graphs Examples



Anatomy of a graph

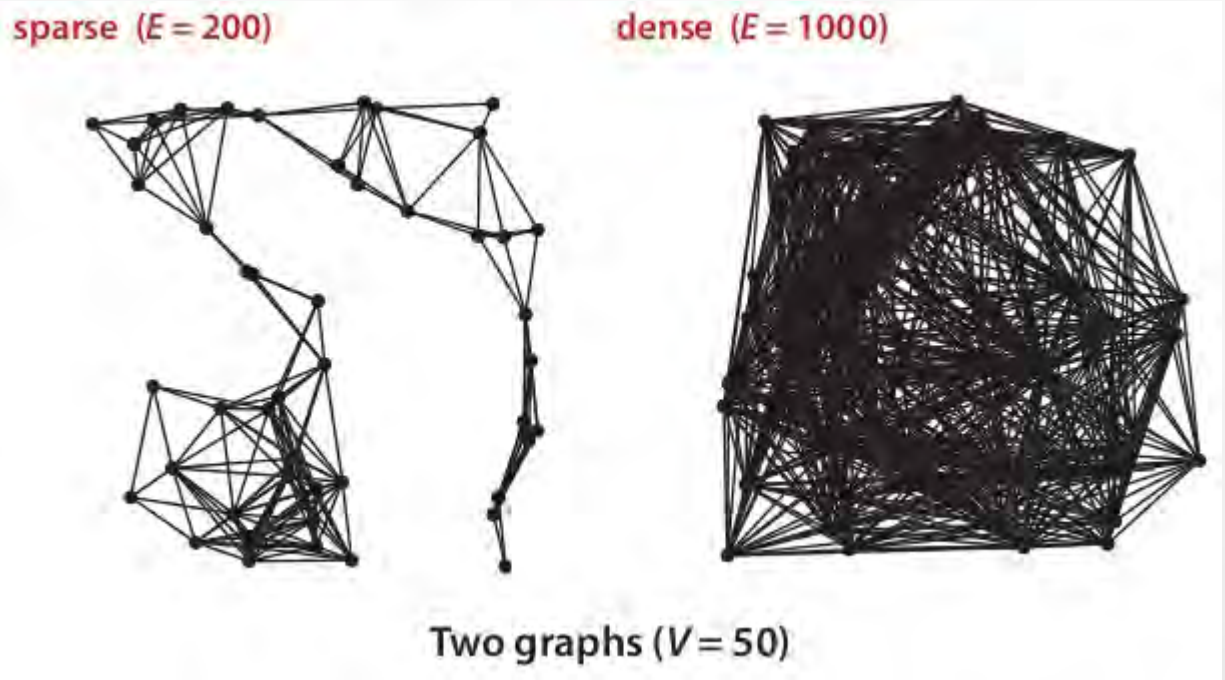


Anomalies



Graphs Density

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



Trees

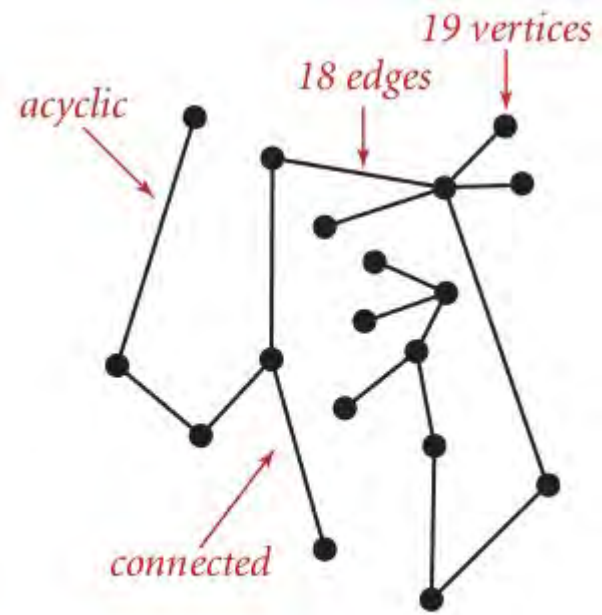
- Graph is connected is there is a path from every vertex to every other vertex.
- Graph that is not connected consists of a set of connected components
- Tree is an acyclic, connected graph
- Tree iff has the following properties
  - G has |V| – 1 edges and no cycles
  - G has |V| – 1 edges and is connected
  - 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

Spanning Trees

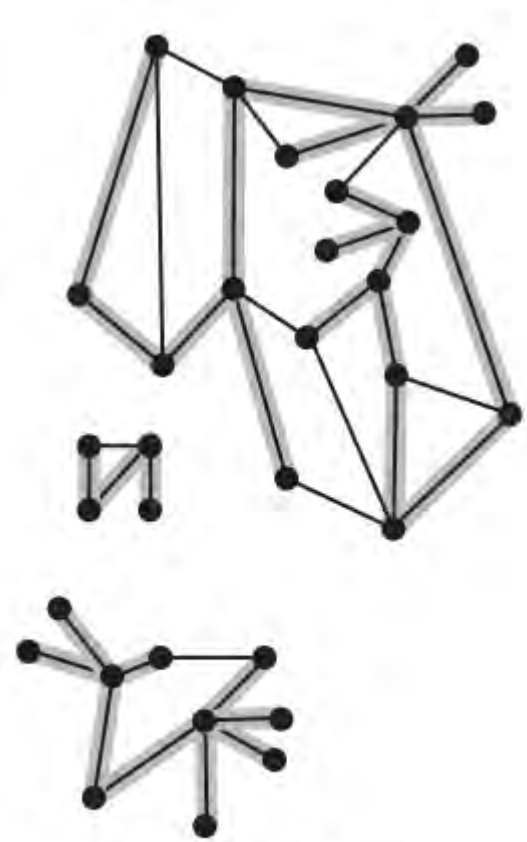
- Forest is a disjoint set of trees
- Spanning tree is a subgraph of a connected graph that contains all vertices and is a single tree
- Spanning forest is a union of spanning trees of connected components of a graph

- Graphs theory is rich with nomenclatures and applications. We will only cover a few.

Tree Examples



A tree



A spanning forest

Graph Data Structure Representations 1/2

Sedgewick 4.1

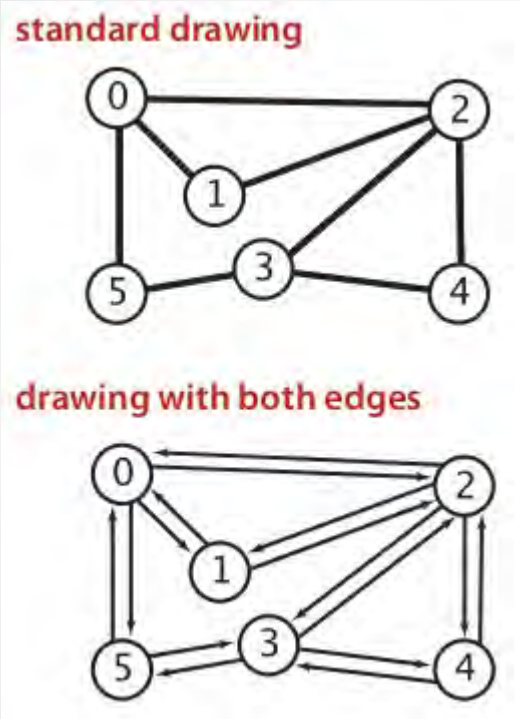
- Three approaches (there are others)
  - Edge List
  - Adjacency Matrix
  - Adjacency List
- There are pros and cons related to amount of memory for and performance of various algorithms for a given data representation. Common issues:
  - How much space is used?
  - Performance of adding an edge between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

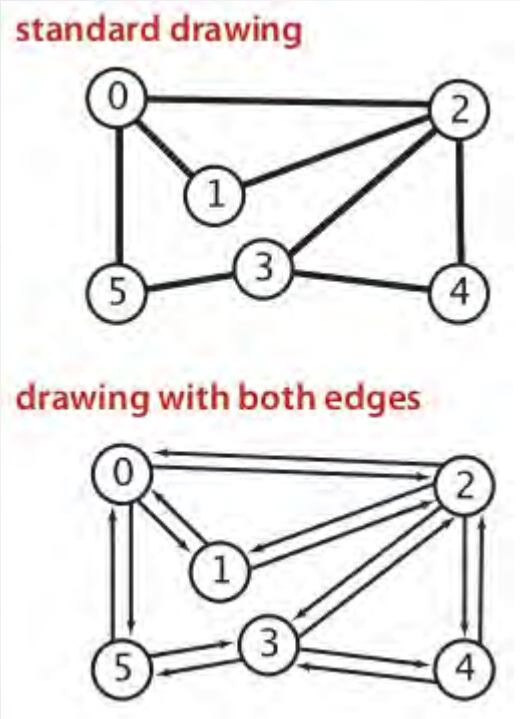
- Edge List is list of edges.
- Adjacency Matrix is a two-dimensional array in which the value at the intersection of two vertices record the edge weight (1 if unweighted), e.g. m[i][j] = 23
- Adjacency List is an array of linked list where the index represents the vertex and the list represents the 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



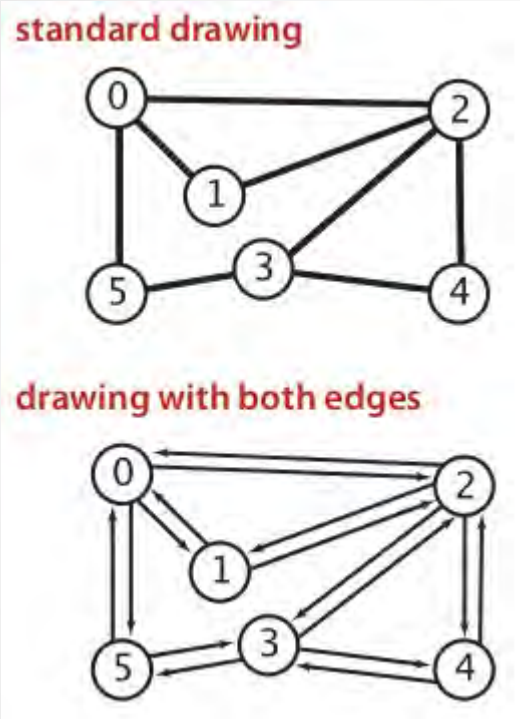
0 1	1 0
0 2	2 0
0 5	5 0
1 2	2 1
2 3	3 2
2 4	4 2
3 4	4 3
5 3	3 5

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 Data Structure Issues

- Approaches assume the graph vertex id is an integer in a contiguous range
  - Almost never the case, how to resolve?
  - Use SymbolTable to map
- Current graph data representations allow parallel edges and self-loops. We will not prevent this but will not use for examples in this course.

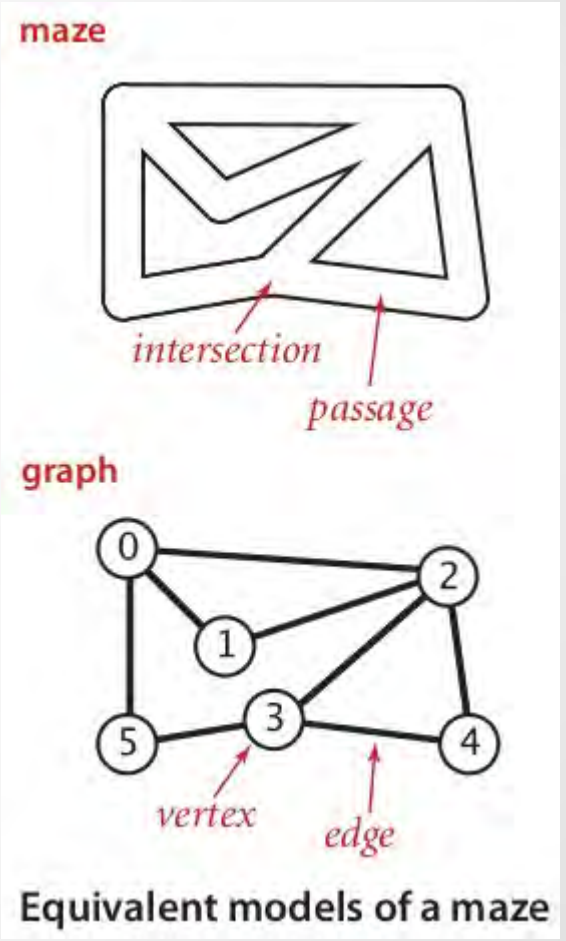
So what kind of problems, and associated applications, might we need to solve using our graph data structure implementation?

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

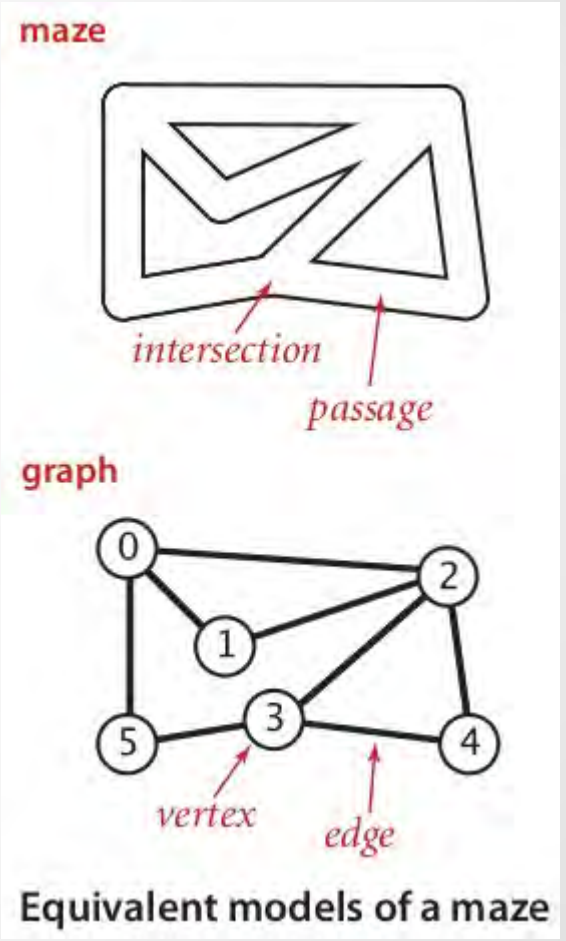
[1] neighbors=2, 0

[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
- 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

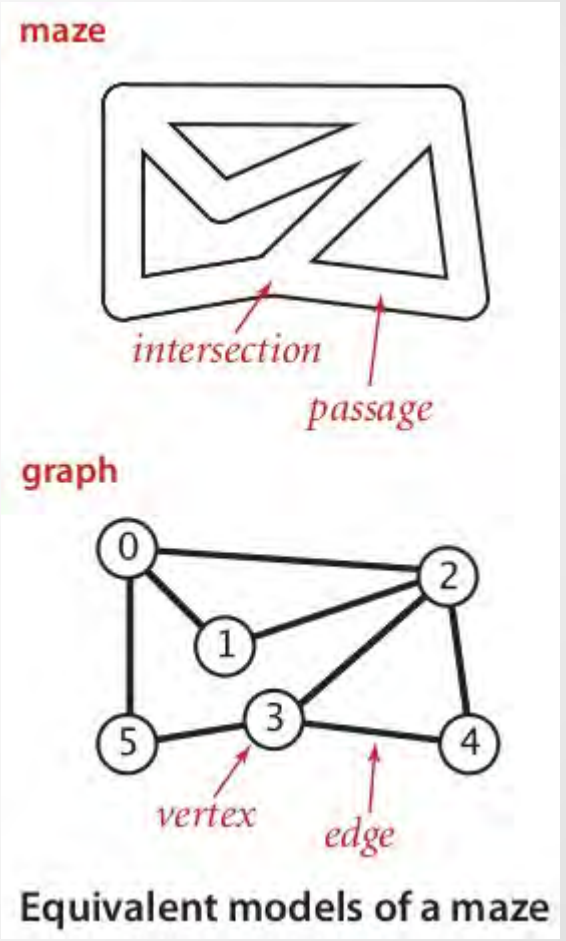
[1] neighbors=2, 0

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

[3] neighbors=4, 2, 5

[4] neighbors=2, 3

[5] neighbors=3, 0



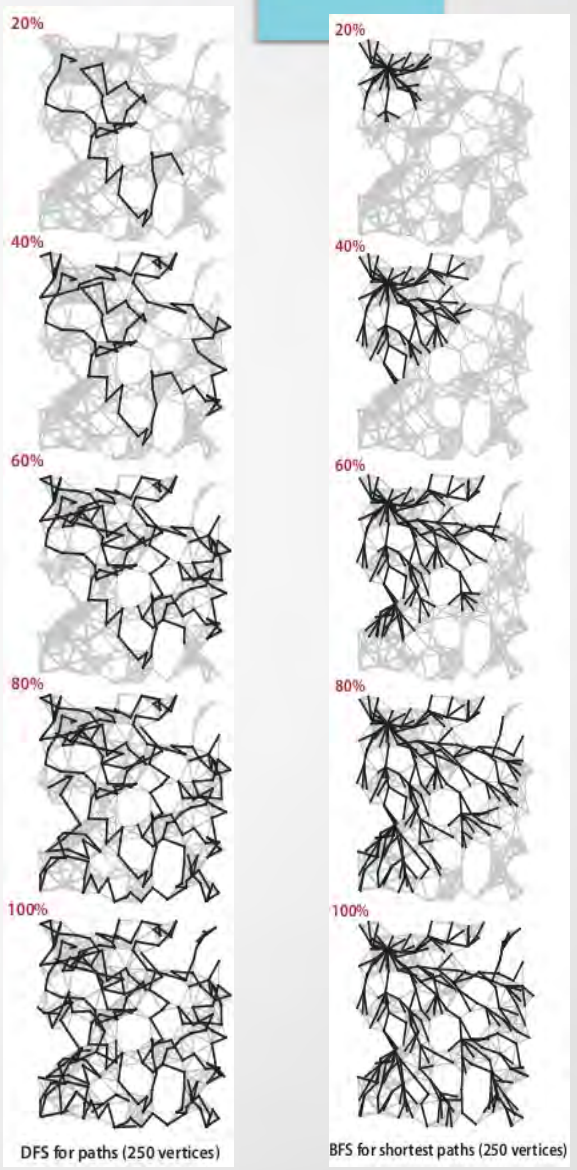
Equivalent models of a maze

Breadth First Search Performance

- BFS takes time proportional to O(V + E) in the worse case
  - 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

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
            ...
}
```

Questions?

PA10

- Create an adjacency list data structure and use it to implement BFS and DFS

Free Question Time!