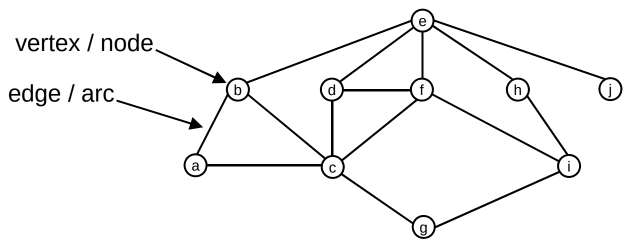
Graphs

Computer Science E-22
Harvard Extension School

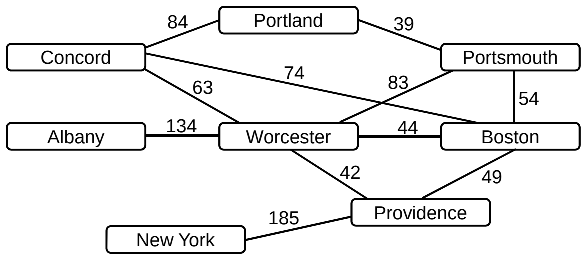
David G. Sullivan, Ph.D.

What is a Graph?



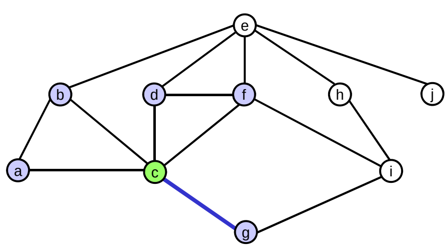
- A graph consists of:
  - a set of *vertices* (also known as *nodes*)
  - a set of *edges* (also known as *arcs*), each of which connects a pair of vertices

Example: A Highway Graph



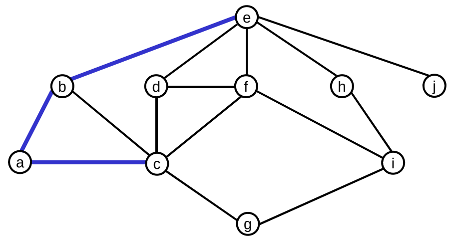
- Vertices represent cities.
- Edges represent highways.
- This is a *weighted* graph, because it has a *cost* associated with each edge.
  - for this example, the costs denote mileage
- We'll use graph algorithms to answer questions like "What is the shortest route from Portland to Providence?"

Relationships Among Vertices

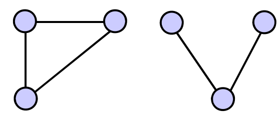


- Two vertices are *adjacent* if they are connected by a single edge.
  - *ex:* c and g are adjacent, but c and i are not
- The collection of vertices that are adjacent to a vertex v are referred to as v's *neighbors.*
  - *ex:* c's neighbors are a, b, d, f, and g

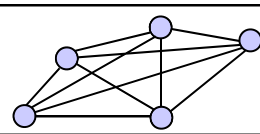
Paths in a Graph



- A *path* is a sequence of edges that connects two vertices.
  - *ex:* the path highlighted above connects c and e
- A graph is *connected* if there is a path between any two vertices.
  - *ex:* the six vertices at right are part of a graph that is *not* connected

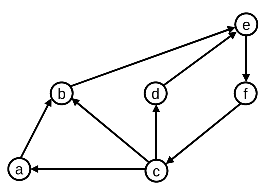


- A graph is *complete* if there is an edge between every pair of vertices.
  - *ex:* the graph at right is complete



Directed Graphs

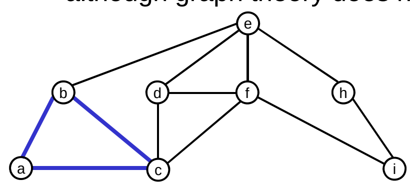
- A *directed* graph has a direction associated with each edge, which is depicted using an arrow:



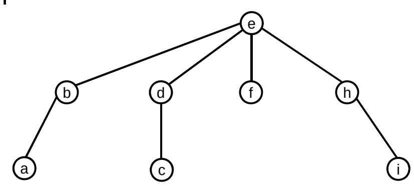
- Edges in a directed graph are often represented as ordered pairs of the form (start vertex, end vertex).
  - *ex:* (a, b) is an edge in the graph above, but (b, a) is not.
- A path in a directed graph is a sequence of edges in which the end vertex of edge i must be the same as the start vertex of edge i + 1.
  - *ex:* { (a, **b**), (**b**, **e**), (**e**, f) } is a valid path.
    { (a, **b**), (**c**, **b**), (**c**, a) } is not.

Trees vs. Graphs

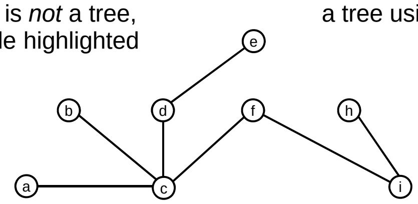
- A tree is a special type of graph.
  - it is connected and undirected
  - it is *acyclic:* there is no path containing distinct edges that starts and ends at the same vertex
  - we usually single out one of the vertices to be the root of the tree, although graph theory does not require this



a graph that is *not* a tree,
with one cycle highlighted



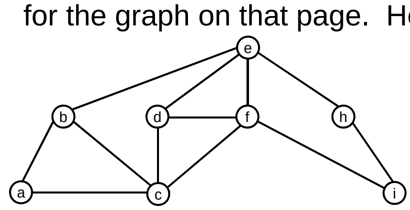
a tree using the same nodes



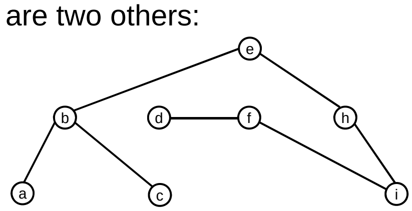
another tree using the same nodes

Spanning Trees

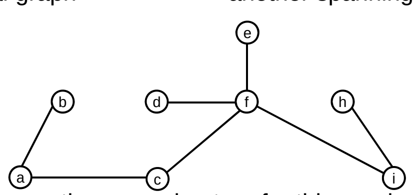
- A spanning tree is a subset of a connected graph that contains:
  - all of the vertices
  - a subset of the edges that form a tree
- The trees on the previous page were examples of spanning trees for the graph on that page. Here are two others:



the original graph



another spanning tree for this graph

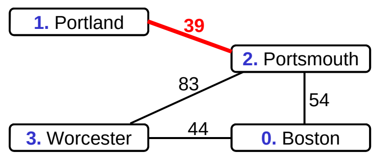


another spanning tree for this graph

Representing a Graph Using an Adjacency Matrix

- Adjacency matrix = a two-dimensional array that is used to represent the edges and any associated costs
  - edge[r][c] = the cost of going from vertex r to vertex c
- Example:

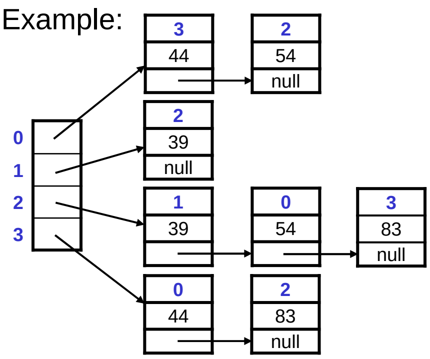
	0	1	2	3
0			54	44
1			39	
2	54	39		83
3	44		83	

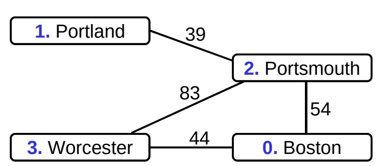


- Use a special value to indicate that you can't go from r to c.
  - either there's no edge between r and c, or it's a directed edge that goes from c to r
  - this value is shown as a shaded cell in the matrix above
  - we can't use 0, because we may have actual costs of 0
- This representation is good if a graph is *dense* – if it has many edges per vertex – but wastes memory if the graph is *sparse* – if it has few edges per vertex.

Representing a Graph Using an Adjacency List

- Adjacency list = a list (either an array or linked list) of linked lists that is used to represent the edges and any associated costs
- Example:

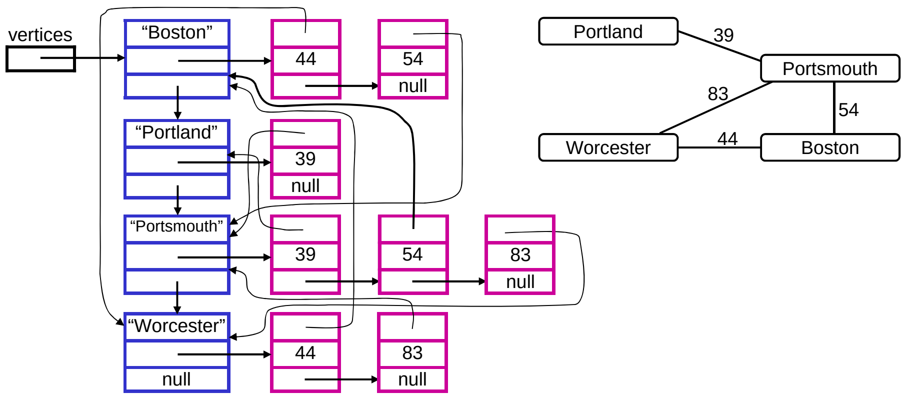




- No memory is allocated for non-existent edges, but the references in the linked lists use extra memory.
- This representation is good if a graph is sparse, but wastes memory if the graph is dense.

Our Graph Representation

- Use a linked list of linked lists for the adjacency list.
- Example:



`vertices` is a reference to a linked list of `Vertex` objects.
Each `Vertex` holds a reference to a linked list of `Edge` objects.
Each `Edge` holds a reference to the `Vertex` that is the end vertex.

Graph Class

```
public class Graph {
    private class Vertex {
        private String id;
        private Edge edges;              // adjacency list
        private Vertex next;
        private boolean encountered;
        private boolean done;
        private Vertex parent;
        private double cost;
        …
    }

    private class Edge {
        private Vertex start;
        private Vertex end;
        private double cost;
        private Edge next;
        …
    }

    private Vertex vertices;
    …
}
```

The highlighted fields are shown in the diagram on the previous page.

Traversing a Graph

- Traversing a graph involves starting at some vertex and visiting all of the vertices that can be reached from that vertex.
  - visiting a vertex = processing its data in some way
    - example: print the data
  - if the graph is connected, all of the vertices will be visited
- We will consider two types of traversals:
  - **depth-first**: proceed as far as possible along a given path before backing up
  - **breadth-first**: visit a vertex
    visit all of its neighbors
    visit all unvisited vertices 2 edges away
    visit all unvisited vertices 3 edges away, etc.
- Applications:
  - determining the vertices that can be reached from some vertex
  - state-space search
  - web crawler (vertices = pages, edges = links)

Depth-First Traversal

- Visit a vertex, then make recursive calls on all of its yet-to-be-visited neighbors:

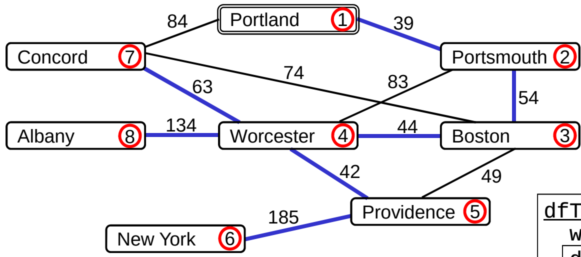
```
dfTrav(v, parent)
    visit v and mark it as visited
    v.parent = parent
    for each vertex w in v's neighbors
        if (w has not been visited)
            dfTrav(w, v)
```

- Java method:

```
private static void dfTrav(Vertex v, Vertex parent) {
    System.out.println(v.id);    // visit v
    v.done = true;
    v.parent = parent;

    Edge e = v.edges;
    while (e != null) {
        Vertex w = e.end;
        if (!w.done)
            dfTrav(w, v);
        e = e.next;
    }
}
```

Example: Depth-First Traversal from Portland

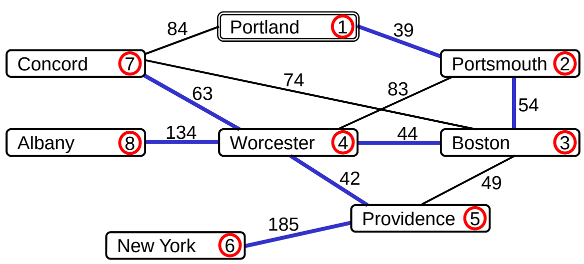


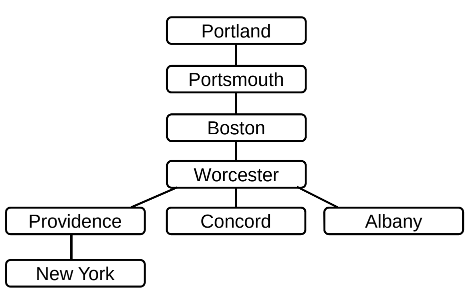
For the examples, we'll assume that the edges in each vertex's adjacency list are sorted by increasing edge cost.

```
void dfTrav(Vertex v, Vertex parent) {
    System.out.println(v.id);
    v.done = true;
    v.parent = parent;
    Edge e = v.edges;
    while (e != null) {
        Vertex w = e.end;
        if (!w.done)
            dfTrav(w, v);
        e = e.next;
    }
}
```

```
dfTrav(Ptl, null)
  w = Pts
  dfTrav(Pts, Ptl)
    w = Ptl, Bos
    dfTrav(Bos, Pts)
      w = Wor
      dfTrav(Wor, Bos)
        w = Pro
        dfTrav(Pro, Wor)
          w = Wor, Bos, NY
          dfTrav(NY, Pro)
            w = Pro
            return
          no more neighbors
          return
        w = Bos, Con
        dfTrav(Con, Wor)
          …
```

Depth-First Spanning Tree



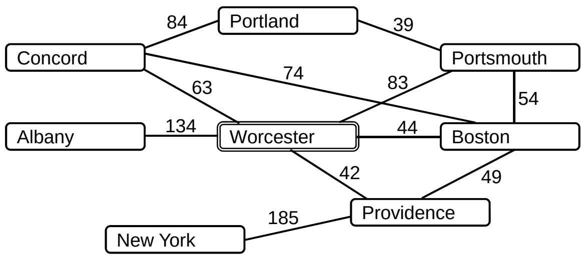


The edges obtained by following the `parent` references form a spanning tree with the origin of the traversal as its root.

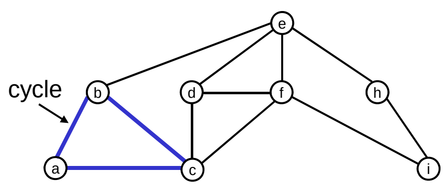
From any city, we can get to the origin by following the roads in the spanning tree.

Another Example: Depth-First Traversal from Worcester

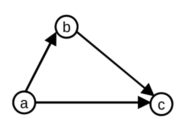
- In what order will the cities be visited?
- Which edges will be in the resulting spanning tree?



Checking for Cycles in an Undirected Graph



- To discover a cycle in an undirected graph, we can:
  - perform a depth-first traversal, marking the vertices as visited
  - when considering neighbors of a visited vertex, if we discover one already marked as visited, there must be a cycle
- If no cycles found during the traversal, the graph is acyclic.
- This doesn't work for directed graphs:
  - c is a neighbor of both a and b
  - there is no cycle



Breadth-First Traversal

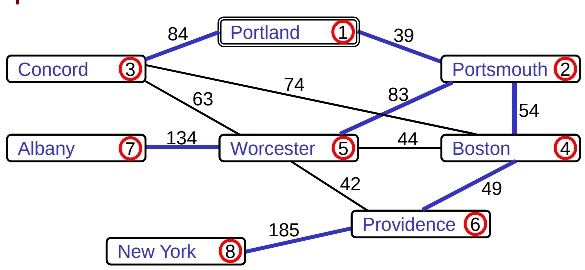
- Use a queue, as we did for BFS and level-order tree traversal:

```
private static void bfTrav(Vertex origin) {
    origin.encountered = true;
    origin.parent = null;
    Queue<Vertex> q = new LLQueue<Vertex>();
    q.insert(origin);

    while (!q.isEmpty()) {
        Vertex v = q.remove();
        System.out.println(v.id);        // Visit v.

        // Add v's unencountered neighbors to the queue.
        Edge e = v.edges;
        while (e != null) {
            Vertex w = e.end;
            if (!w.encountered) {
                w.encountered = true;
                w.parent = v;
                q.insert(w);
            }
            e = e.next;
        }
    }
}
```

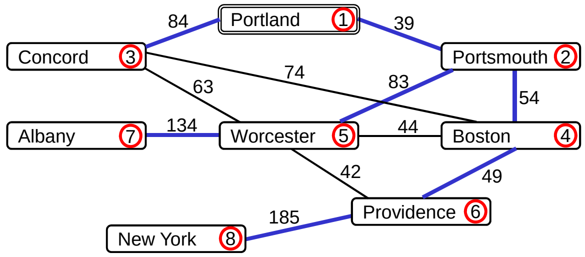
Example: Breadth-First Traversal from Portland



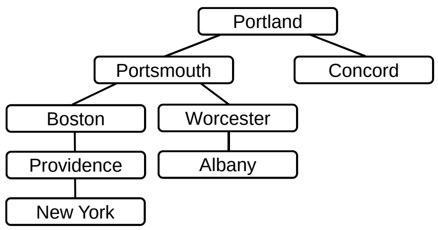
Evolution of the queue:

remove	insert	queue contents
	Portland	Portland
Portland	Portsmouth, Concord	Portsmouth, Concord
Portsmouth	Boston, Worcester	Concord, Boston, Worcester
Concord	*none*	Boston, Worcester
Boston	Providence	Worcester, Providence
Worcester	Albany	Providence, Albany
Providence	New York	Albany, New York
Albany	*none*	New York
New York	*none*	*empty*

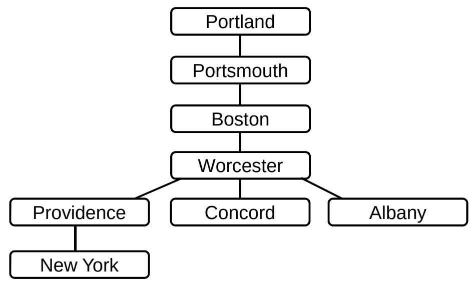
Breadth-First Spanning Tree



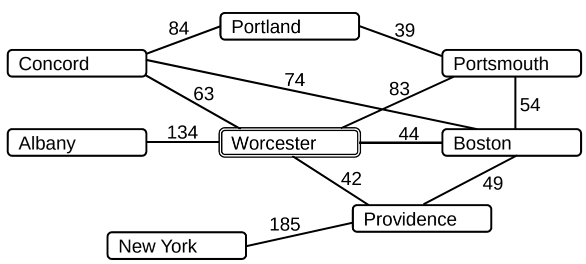
breadth-first spanning tree:



depth-first spanning tree:



Another Example: Breadth-First Traversal from Worcester



Evolution of the queue:

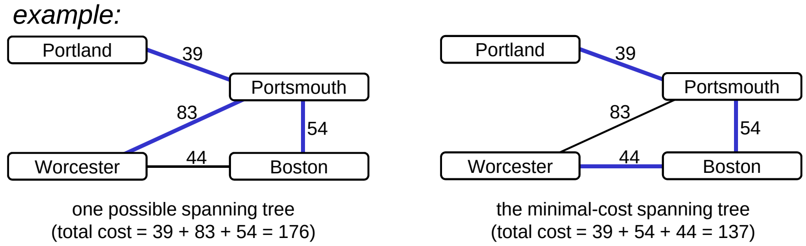
<u>remove</u> <u>insert</u> <u>queue contents</u>

Time Complexity of Graph Traversals

- let V = number of vertices in the graph
  E = number of edges
- If we use an adjacency matrix, a traversal requires $O(V^2)$ steps.
  - why?
- If we use an adjacency list, a traversal requires $O(V + E)$ steps.
  - visit each vertex once
  - traverse each vertex's adjacency list at most once
    - the total length of the adjacency lists is at most $2E = O(E)$
  - $O(V + E) << O(V^2)$ for a sparse graph
  - for a dense graph, $E = O(V^2)$, so both representations are $O(V^2)$
- In our implementations of the remaining algorithms, we'll assume an adjacency-list implementation.

Minimum Spanning Tree

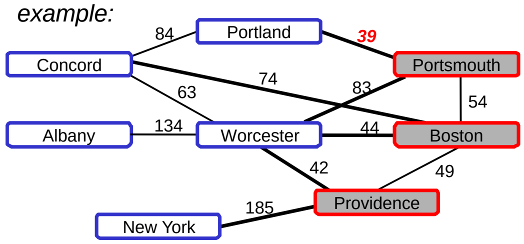
- A minimum spanning tree (MST) has the smallest total cost among all possible spanning trees.
  - *example:*



- If no two edges have the same cost, there is a unique MST. If two or more edges have the same cost, there may be more than one MST.
- Finding an MST could be used to:
  - determine the shortest highway system for a set of cities
  - calculate the smallest length of cable needed to connect a network of computers

Building a Minimum Spanning Tree

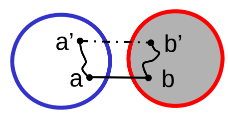
- Key insight: if you divide the vertices into two disjoint subsets A and B, then the lowest-cost edge joining a vertex in A to a vertex in B – call it (a, b) – must be part of the MST.
  - *example:*



The 6 bold edges each join an unshaded vertex to a shaded vertex.

The one with the lowest cost (Portland to Portsmouth) must be in the MST.

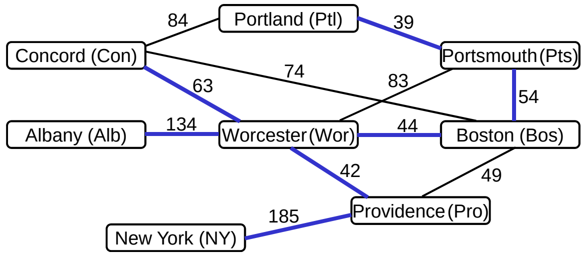
- Proof by contradiction:
  - assume there is an MST (call it T) that doesn't include (a, b)
  - T must include a path from a to b, so it must include one of the other edges (a', b') that spans subsets A and B, such that (a', b') is part of the path from a to b
  - adding (a, b) to T introduces a cycle
  - removing (a', b') gives a spanning tree with lower cost, which contradicts the original assumption.



Prim's MST Algorithm

- Begin with the following subsets:
  - A = any one of the vertices
  - B = all of the other vertices
- Repeatedly select the lowest-cost edge (a, b) connecting a vertex in A to a vertex in B and do the following:
  - add (a, b) to the spanning tree
  - update the two sets: A = A U {b}
    B = B – {b}
- Continue until A contains all of the vertices.

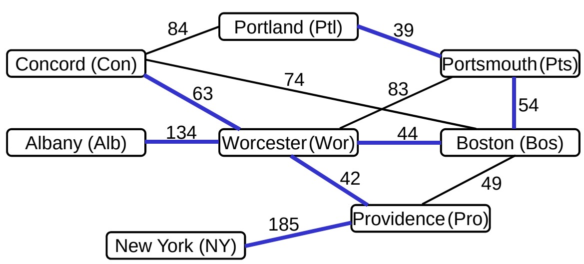
Example: Prim's Starting from Concord



- Tracing the algorithm:

edge added	set A	set B
	{Con}	{Alb, Bos, NY, Ptl, Pts, Pro, Wor}
(Con, Wor)	{Con, Wor}	{Alb, Bos, NY, Ptl, Pts, Pro}
(Wor, Pro)	{Con, Wor, Pro}	{Alb, Bos, NY, Ptl, Pts}
(Wor, Bos)	{Con, Wor, Pro, Bos}	{Alb, NY, Ptl, Pts}
(Bos, Pts)	{Con, Wor, Pro, Bos, Pts}	{Alb, NY, Ptl}
(Pts, Ptl)	{Con, Wor, Pro, Bos, Pts, Ptl}	{Alb, NY}
(Wor, Alb)	{Con, Wor, Pro, Bos, Pts, Ptl, Alb}	{NY}
(Pro, NY)	{Con, Wor, Pro, Bos, Pts, Ptl, Alb, NY}	{}

MST May Not Give Shortest Paths



- The MST is the spanning tree with the minimal *total* edge cost.
- It does <u>not</u> necessarily include the minimal cost path between a pair of vertices.
- Example: shortest path from Boston to Providence is along the single edge connecting them
  - that edge is not in the MST

Implementing Prim's Algorithm in our `Graph` class

- Use the `done` field to keep track of the sets.
  - if `v.done == true`, v is in set A
  - if `v.done == false`, v is in set B
- Repeatedly scan through the lists of vertices and edges to find the next edge to add.

  ➔ *O*(EV)
- We can do better!
  - use a heap-based priority queue to store the vertices in set B
  - priority of a vertex x = –1 `*` cost of the lowest-cost edge connecting x to a vertex in set A
    - why multiply by –1?
  - somewhat tricky: need to update the priorities over time

  ➔ *O*(E log V)

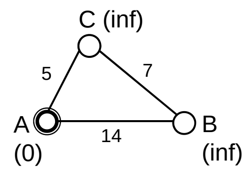
The Shortest-Path Problem

- It's often useful to know the shortest path from one vertex to another – i.e., the one with the minimal total cost
  - example application: routing traffic in the Internet
- For an *unweighted* graph, we can simply do the following:
  - start a breadth-first traversal from the origin, v
  - stop the traversal when you reach the other vertex, w
  - the path from v to w in the resulting (possibly partial) spanning tree is a shortest path
- A breadth-first traversal works for an unweighted graph because:
  - the shortest path is simply one with the fewest edges
  - a breadth-first traversal visits cities in order according to the number of edges they are from the origin.
- Why might this approach fail to work for a *weighted* graph?

Dijkstra's Algorithm

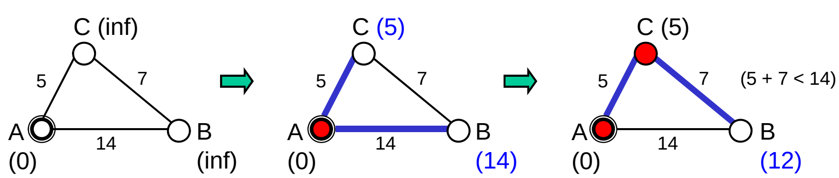
- One algorithm for solving the shortest-path problem for weighted graphs was developed by E.W. Dijkstra.
- It allows us to find the shortest path from a vertex v (the origin) to *all other vertices* that can be reached from v.
- Basic idea:
  - maintain estimates of the shortest paths from the origin to every vertex (along with their costs)
  - gradually refine these estimates as we traverse the graph
- Initial estimates:

	path	cost
the origin itself:	stay put!	0
all other vertices:	unknown	infinity



Dijkstra's Algorithm (cont.)

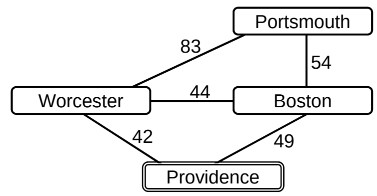
- We say that a vertex w is *finalized* if we have found the shortest path from v to w.
- We repeatedly do the following:
  - find the unfinalized vertex w with the lowest cost estimate
  - mark w as finalized (shown as a filled circle below)
  - examine each unfinalized neighbor x of w to see if there is a shorter path to x that passes through w
    - if there is, update the shortest-path estimate for x
- Example:



Another Example: Shortest Paths from Providence

- Initial estimates:

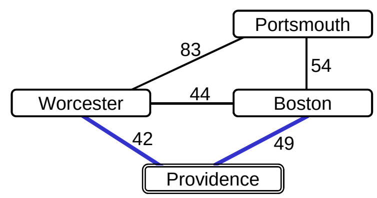
Boston	infinity
Worcester	infinity
Portsmouth	infinity
Providence	***0***



- Providence has the smallest unfinalized estimate, so we finalize it.

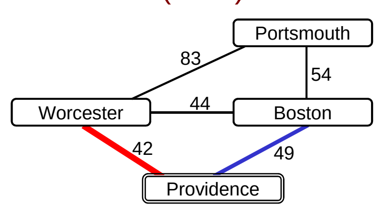
- We update our estimates for its neighbors:

Boston	*49* (< infinity)
Worcester	*42* (< infinity)
Portsmouth	infinity
Providence	0



Shortest Paths from Providence (cont.)

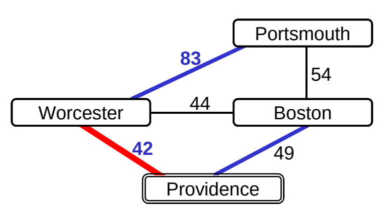
Boston	49
Worcester	***42***
Portsmouth	infinity
Providence	0



- Worcester has the smallest unfinalized estimate, so we finalize it.
  - any other route from Prov. to Worc. would need to go via Boston, and since (Prov → Worc) < (Prov → Bos), we can't do better.

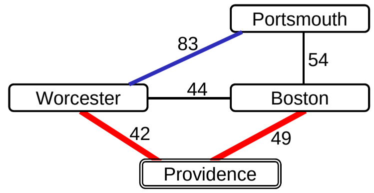
- We update our estimates for Worcester's unfinalized neighbors:

Boston	49 (no change)
Worcester	42
Portsmouth	*125* (42 + 83 < infinity)
Providence	0



Shortest Paths from Providence (cont.)

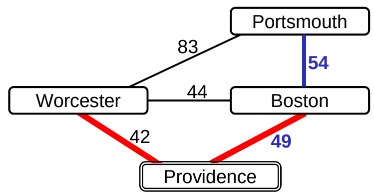
Boston	***49***
Worcester	42
Portsmouth	125
Providence	0



- Boston has the smallest unfinalized estimate, so we finalize it.
  - we'll see later why we can safely do this!

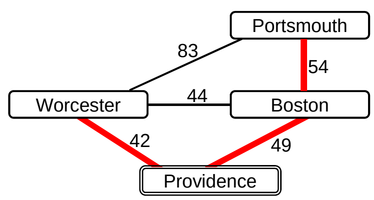
- We update our estimates for Boston's unfinalized neighbors:

Boston	49
Worcester	42
Portsmouth	*103* (49 + 54 < 125)
Providence	0



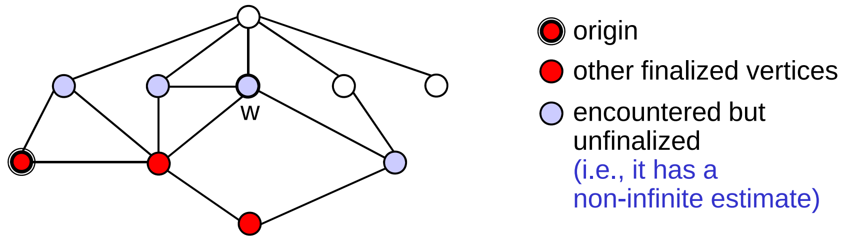
Shortest Paths from Providence (cont.)

Boston	49
Worcester	42
Portsmouth	***103***
Providence	0



- Only Portsmouth is left, so we finalize it.

Finalizing a Vertex



- Let w be the unfinalized vertex with the smallest cost estimate. Why can we finalize w, before seeing the rest of the graph?
- We know that w's current estimate is for the shortest path to w that passes through only *finalized* vertices.
- Any shorter path to w would have to pass through one of the other encountered-but-unfinalized vertices, but we know that they're all further away from the origin than w is.
  - their cost estimates may decrease in subsequent stages of the algorithm, but they can't drop below w's current estimate!

Pseudocode for Dijkstra's Algorithm

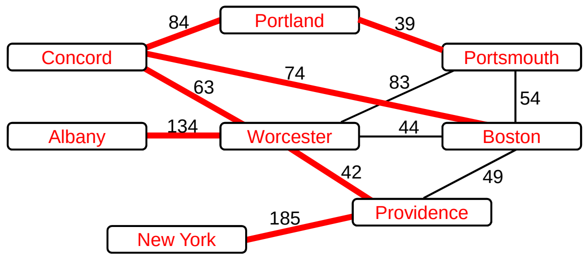
```
dijkstra(origin)
    origin.cost = 0
    for each other vertex v
        v.cost = infinity;

    while there are still unfinalized vertices with cost < infinity
        find the unfinalized vertex w with the minimal cost
        mark w as finalized

        for each unfinalized vertex x adjacent to w
            cost_via_w = w.cost + edge_cost(w, x)
            if (cost_via_w < x.cost)
                x.cost = cost_via_w
                x.parent = w
```

- At the conclusion of the algorithm, for each vertex v:
  - v.cost is the cost of the shortest path from the origin to v; if v.cost is infinity, there is no path from the origin to v
  - starting at v and following the parent references yields the shortest path

Example: Shortest Paths from Concord

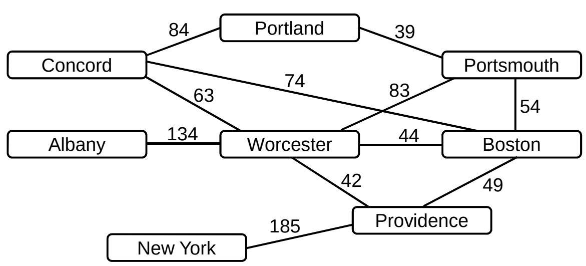


Evolution of the cost estimates (costs in bold have been finalized):

Albany	inf	inf	197	197	197	197	***197***	
Boston	inf	74	***74***					
Concord	***0***							
New York	inf	inf	inf	inf	inf	290	290	***290***
Portland	inf	84	84	***84***				
Portsmouth	inf	inf	146	128	123	***123***		
Providence	inf	inf	105	105	***105***			
Worcester	inf	***63***						

Note that the Portsmouth estimate was improved three times!

Another Example: Shortest Paths from Worcester



Evolution of the cost estimates (costs in bold have been finalized):

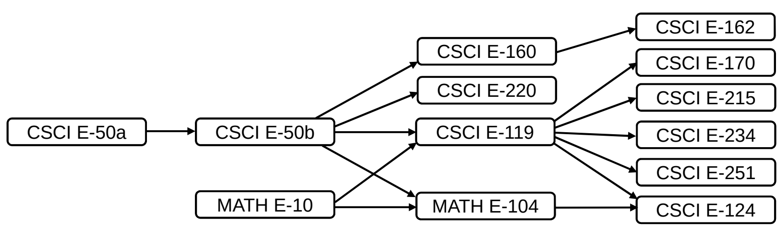
Albany								
Boston								
Concord								
New York								
Portland								
Portsmouth								
Providence								
Worcester								

Implementing Dijkstra's Algorithm

- Similar to the implementation of Prim's algorithm.
- Use a heap-based priority queue to store the unfinalized vertices.
  - priority = ?
- Need to update a vertex's priority whenever we update its shortest-path estimate.
- Time complexity = $O(E\log V)$

Topological Sort

- Used to order the vertices in a directed acyclic graph (a DAG).
- Topological order: an ordering of the vertices such that, if there is directed edge from a to b, a comes before b.
- Example application: ordering courses according to prerequisites



  - a directed edge from a to b indicates that a is a prereq of b
- There may be more than one topological ordering.

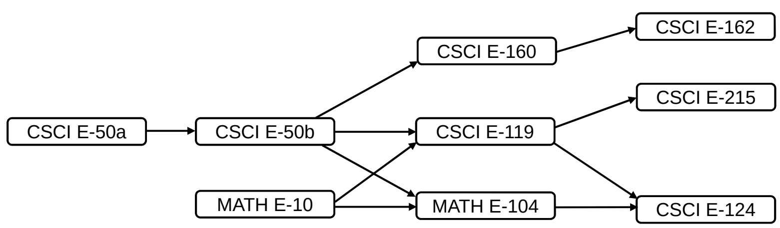
Topological Sort Algorithm

- A *successor* of a vertex v in a directed graph = a vertex w such that (v, w) is an edge in the graph (v → w)
- Basic idea: find vertices that have no successors and work backward from them.
  - there must be at least one such vertex. why?
- Pseudocode for one possible approach:

```
topolSort
    S = a stack to hold the vertices as they are visited
    while there are still unvisited vertices
        find a vertex v with no unvisited successors
        mark v as visited
        S.push(v)
    return S
```

- Popping the vertices off the resulting stack gives one possible topological ordering.

Topological Sort Example

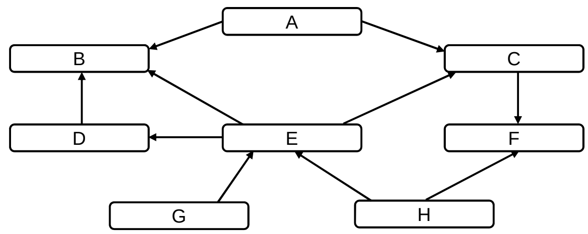


Evolution of the stack:

push	stack contents (top to bottom)
E-124	E-124
E-162	E-162, E-124
E-215	E-215, E-162, E-124
E-104	E-104, E-215, E-162, E-124
E-119	E-119, E-104, E-215, E-162, E-124
E-160	E-160, E-119, E-104, E-215, E-162, E-124
E-10	E-10, E-160, E-119, E-104, E-215, E-162, E-124
E-50b	E-50b, E-10, E-160, E-119, E-104, E-215, E-162, E-124
E-50a	**E-50a, E-50b, E-10, E-160, E-119, E-104, E-215, E-162, E-124**

one possible topological ordering

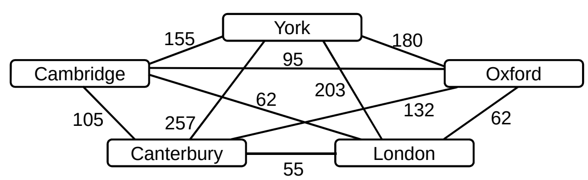
Another Topological Sort Example



Evolution of the stack:

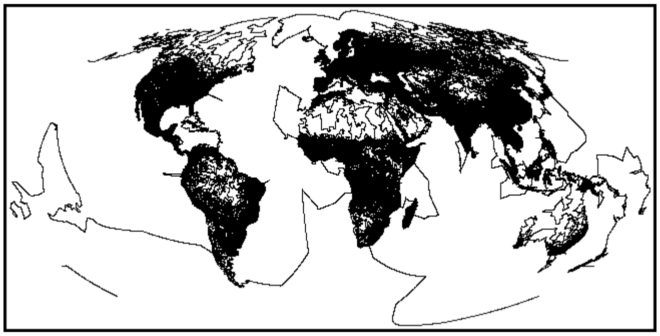
<u>push</u> <u>stack contents (top to bottom)</u>

Traveling Salesperson Problem (TSP)



- A salesperson needs to travel to a number of cities to visit clients, and wants to do so as efficiently as possible.
- As in our earlier problems, we use a weighted graph.
- A *tour* is a path that begins at some starting vertex, passes through every other vertex *once and only once*, and returns to the starting vertex. (The actual starting vertex doesn't matter.)
- TSP: find the tour with the lowest total cost
- TSP algorithms assume the graph is complete, but we can assign infinite costs if there isn't a direct route between two cities.

TSP for Santa Claus



source: http://www.tsp.gatech.edu/world/pictures.html

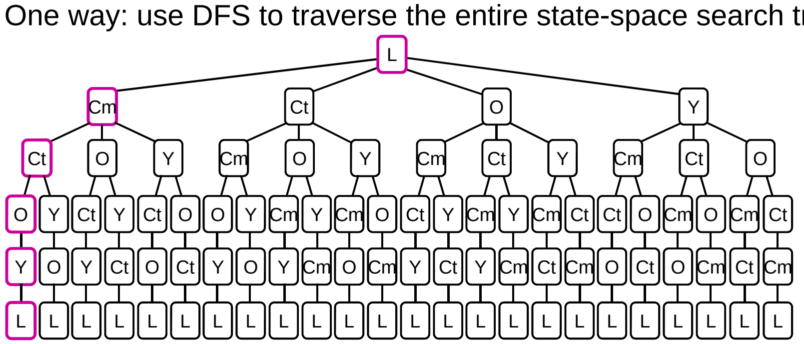
A "world TSP" with 1,904,711 cities.

The figure at right shows a tour with a total cost of 7,516,353,779 meters – which is at most 0.068% longer than the optimal tour.

- Other applications:
  - coin collection from phone booths
  - routes for school buses or garbage trucks
  - minimizing the movements of machines in automated manufacturing processes
  - many others

Solving a TSP: Brute-Force Approach

- Perform an exhaustive search of all possible tours.
- One way: use DFS to traverse the entire state-space search tree.



- The leaf nodes correspond to possible solutions.
  - for n cities, there are (n – 1)! leaf nodes in the tree.
  - half are redundant (e.g., L-Cm-Ct-O-Y-L = L-Y-O-Ct-Cm-L)
- Problem: exhaustive search is intractable for all but small n.
  - example: when n = 14, ((n – 1)!) / 2 = over 3 billion

Solving a TSP: Informed State-Space Search

- Use A* with an appropriate heuristic function for estimating the cost of the remaining edges in the tour.
- This is much better than brute force, but it still uses exponential space and time.

Algorithm Analysis Revisited

- Recall that we can group algorithms into classes (n = problem size):

name	example expressions	big-O notation
constant time	1, 7, 10	$O(1)$
logarithmic time	$3\log_{10}n$, $\log_2 n + 5$	$O(\log n)$
linear time	$5n$, $10n - 2\log_2 n$	$O(n)$
n log n time	$4n \log_2 n$, $n \log_2 n + n$	$O(n \log n)$
quadratic time	$2n^2 + 3n$, $n^2 - 1$	$O(n^2)$
n^c $(c > 2)$	$n^3 - 5n$, $2n^5 + 5n^2$	$O(n^c)$
exponential time	2^n, $5e^n + 2n^2$	$O(c^n)$
factorial time	$(n - 1)!/2$, $3n!$	$O(n!)$

- Algorithms that fall into one of the classes above the dotted line are referred to as *polynomial-time* algorithms.
- The term *exponential-time algorithm* is sometimes used to include *all* algorithms that fall below the dotted line.
  - algorithms whose running time grows as fast or faster than c^n

Classifying Problems

- Problems that can be solved using a polynomial-time algorithm are considered “easy” problems.
  - we can solve large problem instances in a reasonable amount of time
- Problems that don’t have a polynomial-time solution algorithm are considered “hard” or "intractable" problems.
  - they can only be solved exactly for small values of n
- Increasing the CPU speed doesn't help much for intractable problems:

	CPU 1	CPU 2 (1000x faster)
max problem size for $O(n)$ alg:	N	1000N
$O(n^2)$ alg:	N	31.6 N
$O(2^n)$ alg:	N	**N + 9.97**

Classifying Problems (cont.)

- The class of problems that can be solved using a polynomial-time algorithm is known as the class P.
- Many problems that don’t have a polynomial-time solution algorithm belong to a class known as NP
  - for *non-deterministic polymonial*
- If a problem is in NP, it’s possible to guess a solution and verify if the guess is correct in polynomial time.
  - example: a variant of the TSP in which we attempt to determine if there is a tour with total cost <= some bound b
  - given a tour, it takes polynomial time to add up the costs of the edges and compare the result to b

Classifying Problems (cont.)

- If a problem is *NP-complete*, then finding a polynomial-time solution for it would allow you to solve a large number of other hard problems.
  - thus, it’s extremely unlikely such a solution exists!
- The TSP variant described on the previous slide is NP-complete.
- Finding the optimal tour is at least as hard.
- For more info. about problem classes, there is a good video of a lecture by Prof. Michael Sipser of MIT available here:

  http://claymath.msri.org/sipser2006.mov

Dealing With Intractable Problems

- When faced with an intractable problem, we resort to techniques that quickly find solutions that are "good enough".
- Such techniques are often referred to as *heuristic* techniques.
  - heuristic = rule of thumb
  - there's no guarantee these techniques will produce the optimal solution, but they typically work well

Iterative Improvement Algorithms

- One type of heuristic technique is what is known as an *iterative improvement algorithm.*
  - start with a randomly chosen solution
  - gradually make small changes to the solution in an attempt to improve it
    - e.g., change the position of one label
  - stop after some number of iterations
- There are several variations of this type of algorithm.

Hill Climbing

- Hill climbing is one type of iterative improvement algorithm.
  - start with a randomly chosen solution
  - repeatedly consider possible small changes to the solution
  - if a change would improve the solution, make it
  - if a change would make the solution worse, don't make it
  - stop after some number of iterations
- It's called hill climbing because it repeatedly takes small steps that improve the quality of the solution.
  - "climbs" towards the optimal solution

Simulated Annealing

- *Simulated annealing* is another iterative improvement algorithm.
  - start with a randomly chosen solution
  - repeatedly consider possible small changes to the solution
  - if a change would improve the solution, make it
  - if a change would make the solution worse,
    make it some of the time (according to some probability)
  - the probability of doing so reduces over time

Take-Home Lessons

- Object-oriented programming allows us to capture the abstractions in the programs that we write.
  - creates reusable building blocks
  - key concepts: encapsulation, inheritance, polymorphism
- Abstract data types allow us to organize and manipulate collections of data.
  - a given ADT can be implemented in different ways
  - fundamental building blocks: arrays, linked nodes
- Efficiency matters when dealing with large collections of data.
  - some solutions can be *much* faster or more space efficient than others!
  - what's the best data structure/algorithm for the specific instances of the problem that you expect to see?
    - example: sorting an almost sorted collection

Take-Home Lessons (cont.)

- Use the tools in your toolbox!
  - generic data structures
  - lists/stacks/queues
  - trees
  - heaps
  - hash tables
  - recursion
  - recursive backtracking
  - divide-and-conquer
  - state-space search
  - ...

From the Introductory Lecture…

- We will study fundamental *data structures.*
  - ways of imposing order on a collection of information
  - sequences: lists, stacks, and queues
  - trees
  - hash tables
  - graphs
- We will also:
  - study *algorithms* related to these data structures
  - learn how to *compare* data structures & algorithms
- Goals:
  - learn to think more intelligently about programming problems
  - acquire a set of useful tools and techniques