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
• 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
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:
  
  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.

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

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.

```
0  1  2  3
0  null
1  null
2  39
3  null
```

```
1. Portland
2. Portsmouth
3. Worcester
0. Boston
```
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 (in ~cscie119/examples/graphs/Graph.java)

```java
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:
  ```java
  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

```java
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;
    }
}
```

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

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?

![Diagram of a graph with cities and edges labeled with distances.]

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:

```java
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;
        }
    }
}           (~cscie119/examples/graphs/Graph.java)
```

Example: Breadth-First Traversal from Portland

![Diagram of Breadth-First Traversal from Portland]

Evolution of the queue:

<table>
<thead>
<tr>
<th>remove</th>
<th>insert</th>
<th>queue contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Portland</td>
<td>Portlan</td>
<td>Portland</td>
</tr>
<tr>
<td>Portsmouth</td>
<td>Concord, Boston,</td>
<td>Portland,</td>
</tr>
<tr>
<td>Boston</td>
<td>Providence</td>
<td>Concord,</td>
</tr>
<tr>
<td>Worcester</td>
<td>Albany</td>
<td>Boston,</td>
</tr>
<tr>
<td>Providence</td>
<td>Albany</td>
<td>Worcester,</td>
</tr>
<tr>
<td>Albany</td>
<td>New York</td>
<td>Providence,</td>
</tr>
<tr>
<td>New York</td>
<td>none</td>
<td>Albany, New</td>
</tr>
<tr>
<td></td>
<td></td>
<td>York</td>
</tr>
<tr>
<td></td>
<td></td>
<td>empty</td>
</tr>
</tbody>
</table>
Breadth-First Spanning Tree

breadth-first spanning tree:

depth-first spanning tree:

Another Example:
Breadth-First Traversal from Worcester

Evolution of the queue:
remove  insert  queue contents
**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 $2E = O(E)$
    - $O(V + E) \ll O(V^2)$ 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:*

  - one possible spanning tree
    (total cost = $39 + 83 + 54 = 176$)
  - the minimal-cost spanning tree
    (total cost = $39 + 54 + 44 = 137$)

- 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:
    - 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.

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.

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:

<table>
<thead>
<tr>
<th>edge added</th>
<th>set A</th>
<th>set B</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Con, Wor)</td>
<td>{Con}</td>
<td>{Alb, Bos, NY, Ptl, Pts, Pro, Wor}</td>
</tr>
<tr>
<td>(Wor, Pro)</td>
<td>{Con, Wor, Pro}</td>
<td>{Alb, Bos, NY, Ptl, Pts}</td>
</tr>
<tr>
<td>(Wor, Bos)</td>
<td>{Con, Wor, Pro, Bos}</td>
<td>{Alb, NY, Ptl, Pts}</td>
</tr>
<tr>
<td>(Bos, Pts)</td>
<td>{Con, Wor, Pro, Bos, Pts}</td>
<td>{Alb, NY, Ptl}</td>
</tr>
<tr>
<td>(Pts, Ptl)</td>
<td>{Con, Wor, Pro, Bos, Pts, Ptl}</td>
<td>{Alb, NY}</td>
</tr>
<tr>
<td>(Wor, Alb)</td>
<td>{Con, Wor, Pro, Bos, Pts, Ptl, Alb}</td>
<td>{NY}</td>
</tr>
<tr>
<td>(Pro, NY)</td>
<td>{Con, Wor, Pro, Bos, Pts, Ptl, Alb, NY}</td>
<td>{}</td>
</tr>
</tbody>
</table>

MST May Not Give Shortest Paths

- The MST is the spanning tree with the minimal total edge cost.
- It does not 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

• Vertex \( v \) is in set \( A \) if \( v.\text{done} = \text{true} \), else it’s in set \( B \).

• One possible implementation: scan to find the next edge to add:
  • iterate over the list of vertices in the graph
  • for each vertex in \( A \), iterate over its adjacency list
  • keep track of the minimum-cost edge connecting a vertex in \( A \) to a vertex in \( B \)

• Time analysis:
  • let \( V = \) the number of vertices in the graph
    \( E = \) the number of edges
  • how many scans are needed?
  • how many edges are considered per scan?
  • time complexity of this implementation = ?

Prim’s: A More Efficient Implementation

• Maintain a heap-based priority queue containing the vertices in \( B \).
  • priority = \(-1 \times \) cost of the lowest-cost edge connecting the vertex to a vertex in \( A \)

• On each iteration of the algorithm:
  • remove the highest priority vertex \( v \), and add its lowest-cost edge to the MST
  • update as needed the priority of \( v \)'s neighbors
    • why might their priorities change?

  • what do we need to do if a vertex's priority changes?
Prim's: A More Efficient Implementation (cont.)

- Time analysis:
  - # of steps to create the initial heap?
  - removing vertices from the heap:
    - # of steps to remove one vertex?
    - total # of steps spent removing vertices?
  - updating priorities:
    - # of steps to update the priority of one vertex?
    - # of times we update a priority in the worst case?

- total # of steps spent updating priorities?
- time complexity of this implementation?

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 vertex \( v \) to \( \textit{all other vertices} \) that can be reached from \( v \).

- Basic idea:
  - maintain estimates of the shortest paths from \( v \) to every vertex (along with their associated costs)
  - gradually refine these estimates as we traverse the graph

**Dijkstra’s Algorithm (cont.)**

- We say that a vertex \( w \) is \textit{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:
Simple Example: Shortest Paths from Providence

- Start out with the following estimates:
  - Boston: infinity
  - Worcester: infinity
  - Portsmouth: infinity
  - Providence: 0

- Providence has the smallest unfinalized estimate, so we finalize it.
  Given what we know about the roads from Providence, we update our estimates:
  - Boston: 49 (< infinity)
  - Worcester: 42 (< infinity)
  - 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 \(\rightarrow\) Worc) < (Prov \(\rightarrow\) Bos), we can't do better.

Shortest Paths from Providence (cont.)

- After finalizing Worcester, we use information about the roads from Worcester to update our estimates:
  - Boston: 49 (< 42 + 44)
  - Worcester: 42
  - Portsmouth: 125 (via Worcester)
  - Providence: 0

- Boston has the smallest unfinalized estimate, so we finalize it.
  - any other route from Prov. to Boston. would need to go via Worcester, and (Prov \(\rightarrow\) Bos) < (Prov \(\rightarrow\) Worc \(\rightarrow\) Bos) (Prov \(\rightarrow\) Bos) < (Prov \(\rightarrow\) Worc \(\rightarrow\) Ports)
Shortest Paths from Providence (cont.)

- After finalizing Boston, we update our estimates:
  - Boston 49
  - Worcester 42
  - Portsmouth 103 (via Boston)
  - Providence 0

- Only Portsmouth is left, so we finalize it:
  - Boston 49
  - Worcester 42
  - Portsmouth **103 (via Boston)**
  - Providence 0

---

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
• The Java version is in `~cscie119/examples/graphs/Graph.java`

Example: Shortest Paths from Concord

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

<table>
<thead>
<tr>
<th></th>
<th>inf</th>
<th>inf</th>
<th>197</th>
<th>197</th>
<th>197</th>
<th>197</th>
<th>197</th>
<th>197</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albany</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boston</td>
<td>inf</td>
<td>74</td>
<td>74</td>
<td>74</td>
<td>74</td>
<td>74</td>
<td>74</td>
<td>74</td>
</tr>
<tr>
<td>Concord</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>New York</td>
<td>inf</td>
<td>inf</td>
<td>inf</td>
<td>inf</td>
<td>inf</td>
<td>290</td>
<td>290</td>
<td>290</td>
</tr>
<tr>
<td>Portland</td>
<td>inf</td>
<td>84</td>
<td>84</td>
<td>84</td>
<td>84</td>
<td>84</td>
<td>84</td>
<td>84</td>
</tr>
<tr>
<td>Portsmouth</td>
<td>inf</td>
<td>146</td>
<td>128</td>
<td>123</td>
<td>123</td>
<td>123</td>
<td>123</td>
<td>123</td>
</tr>
<tr>
<td>Providence</td>
<td>inf</td>
<td>105</td>
<td>105</td>
<td>105</td>
<td>105</td>
<td>105</td>
<td>105</td>
<td>105</td>
</tr>
<tr>
<td>Worcester</td>
<td>inf</td>
<td>63</td>
<td>63</td>
<td>63</td>
<td>63</td>
<td>63</td>
<td>63</td>
<td>63</td>
</tr>
</tbody>
</table>

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):

<table>
<thead>
<tr>
<th></th>
<th>39</th>
<th>54</th>
<th>44</th>
<th>83</th>
<th>49</th>
<th>185</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albany</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boston</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concord</td>
<td></td>
<td></td>
<td>63</td>
<td>74</td>
<td></td>
<td></td>
</tr>
<tr>
<td>New York</td>
<td></td>
<td></td>
<td>63</td>
<td>54</td>
<td>42</td>
<td>185</td>
</tr>
<tr>
<td>Portland</td>
<td>84</td>
<td>39</td>
<td>63</td>
<td>54</td>
<td>49</td>
<td>185</td>
</tr>
<tr>
<td>Portsmouth</td>
<td>84</td>
<td>39</td>
<td>63</td>
<td>54</td>
<td>49</td>
<td>185</td>
</tr>
<tr>
<td>Providence</td>
<td>84</td>
<td>39</td>
<td>63</td>
<td>54</td>
<td>49</td>
<td>185</td>
</tr>
<tr>
<td>Worcester</td>
<td>84</td>
<td>39</td>
<td>63</td>
<td>54</td>
<td>49</td>
<td>185</td>
</tr>
</tbody>
</table>

Techniques Illustrated by Dijkstra’s

- **Dynamic programming**: store solutions to subproblems, and reuse them when solving larger problems.
  - example: once we know that the shortest path from Concord to Providence goes through Worcester, we know that the shortest path from Concord to New York via Providence must also go through Worcester

- **Greedy strategy**: at every stage, do what is “locally” best by finalizing the minimal-cost city that has not yet been finalized.
  - in this case, a greedy approach produces the optimal solution
  - for other problems, doing what seems best in the short term does not always produce the optimal solution
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

  - Another application: finding a serial ordering of concurrent transactions in a database system.
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**  
E-124  
E-162  
E-215  
E-104  
E-119  
E-160  
E-10  
E-50b  
E-50a

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

---

### Another Topological Sort Example

Evolution of the stack:

**push**  
B  
C  
D  
E  
G  
H

**stack contents (top to bottom)**  
B  
C  
D  
E  
G  
H

---
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

- 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

Source: http://www.tsp.gatech.edu/world/pictures.html
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 $n = 14$, $((n-1)!)/2 = 3$ 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):

<table>
<thead>
<tr>
<th>name</th>
<th>example expressions</th>
<th>big-O notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant time</td>
<td>1, 7, 10</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>logarithmic time</td>
<td>3\log_{10}n, \log_2n + 5</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>linear time</td>
<td>5n, 10n - 2\log_2n</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>n log n time</td>
<td>4n \log_2n, n \log_2n + n</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>quadratic time</td>
<td>2n^2 + 3n, n^2 - 1</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$n^c$ (c &gt; 2)</td>
<td>$n^3 - 5n$, $2n^3 + 5n^2$</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>exponential time</td>
<td>$2^n$, $5e^n + 2n^2$</td>
<td>$O(c^n)$</td>
</tr>
<tr>
<td>factorial time</td>
<td>$(n-1)!/2$, $3n!$</td>
<td>$O(n!)$</td>
</tr>
</tbody>
</table>

- 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:

<table>
<thead>
<tr>
<th></th>
<th>CPU 1</th>
<th>CPU 2 (1000x faster)</th>
</tr>
</thead>
<tbody>
<tr>
<td>max problem size for $O(n)$ alg:</td>
<td>N</td>
<td>1000N</td>
</tr>
<tr>
<td>$O(n^2)$ alg:</td>
<td>N</td>
<td>31.6 N</td>
</tr>
<tr>
<td>$O(2^n)$ alg:</td>
<td>N</td>
<td>N + 9.97</td>
</tr>
</tbody>
</table>
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 polynomial

• 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

• Think about efficiency when solving a problem.
  • What are the possible data structures and algorithms for this problem?
  • How efficient are they (time and space)?
  • What’s the best data structure/algorithm for the instances of the problem that you expect to see?
    • example: sorting
  • Can you afford to find the optimal solution, or should you use a heuristic technique to find a solution that’s good enough?

• Use the tools in your toolbox:
  • OOP, data abstraction, generic data structures
  • lists/stacks/queues, trees, heaps, …
  • 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