Graphs

Computer Science S-111
Harvard University
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
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?”

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

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

![Graphs vs. Trees](image)

- 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:

![Spanning Trees](image)
Representing a Graph Using an Adjacency Matrix

- Adjacency matrix = a two-dimensional array that is used to represent the edges and any associated costs
  - \( \text{edge}[r][c] = \text{the cost of going from vertex } r \text{ to vertex } c \)
- 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 \textit{dense} – if it has many edges per vertex – but wastes memory if the graph is \textit{sparse} – if it has few edges per vertex.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>54</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>39</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>54</td>
<td>39</td>
<td>83</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>44</td>
<td>83</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Matrix Diagram](image1)

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
  - 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:
  
  \[
  \text{dfTrav}(v, \text{parent})
  \begin{align*}
  & \quad \text{visit } v \text{ and mark it as visited} \\
  & \quad v.\text{parent} = \text{parent} \\
  & \quad \text{for each vertex } w \text{ in } v\text{'s neighbors} \\
  & \quad \quad \text{if } (w \text{ has not been visited}) \\
  & \quad \quad \quad \text{dfTrav}(w, v)
  \end{align*}
  \]

- 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

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.

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 = Pro
                    w = 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?

![Graph diagram showing cities and edges with weights]

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

![Directed graph diagram showing no cycle]
Breadth-First Traversal

- Use a queue, as we do for 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;
          }
      }
  }
  ```

Example: Breadth-First Traversal from Portland

Evolution of the queue:

- remove: Portland
- insert: Portsmouth, Concord, Boston, Worcester
- queue contents:
  - Portland
  - Portsmouth, Concord
  - Concord, Boston, Worcester
  - Boston, Worcester
  - Providence, Albany
  - Albany, New York
  - New York
  - empty
Breadth-First Spanning Tree

**breadth-first spanning tree:**

```
+--- Portland +---
|     |      |     |
|     |      |     |
|     |      |     |
| 84  |  39   | 63  |
|     |      |     |
+--- Concord +---
|     |      |     |
|     |      |     |
|     |      |     |
| 134 |  74   | 83  |
|     |      |     |
+--- Albany +---
|     |      |     |
|     |      |     |
|     |      |     |
| 44  |  49   |  42 |
|     |      |     |
+--- Worcester +---
|     |      |     |
|     |      |     |
|     |      |     |
| 49  |  54   |  54 |
|     |      |     |
+--- Boston +---
|     |      |     |
|     |      |     |
|     |      |     |
| 185 |       |
|     |      |     |
+--- New York +---
```

**depth-first spanning tree:**

```
+--- Portland +---
|     |      |     |
|     |      |     |
|     |      |     |
| 84  |  39   |
|     |      |
+--- Portsmouth +---
|     |      |     |
|     |      |     |
|     |      |     |
| 63  |
|      |
|      |
|      |
|      |
+--- Concord +---
|     |      |     |
|     |      |     |
|     |      |     |
| 74  |
|      |
|      |
|      |
+--- Worcester +---
|     |      |     |
|     |      |     |
|     |      |     |
| 44  |
|      |
|      |
|      |
+--- Boston +---
|     |      |     |
|     |      |     |
|     |      |     |
| 185 |
|      |
|      |
|      |
+--- New York +---
```

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 \( 2E = O(E) \)
    - \( O(V + E) << O(V^2) \) 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:

  ![Graph](image)

  - 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 \((v_a, v_b)\) – must be part of the MST.

  \[\text{example:}\]
  
  - Proof by contradiction:
    - assume we can create an MST (call it T) that doesn’t include edge \((v_a, v_b)\)
    - T must include a path from \(v_a\) to \(v_b\), so it must include one of the other edges \((v'_a, v'_b)\) that spans subsets A and B, such that \((v'_a, v'_b)\) is part of the path from \(v_a\) to \(v_b\)
    - adding \((v_a, v_b)\) to T introduces a cycle
    - removing \((v'_a, v'_b)\) 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 do the following:
  - select the lowest-cost edge \((v_a, v_b)\) connecting a vertex in A to a vertex in B
  - add \((v_a, v_b)\) to the spanning tree
  - move vertex \(v_b\) from set B to set A

- Continue until set 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, Bos, Pts}</td>
<td>{Alb, NY, Ptl}</td>
</tr>
</tbody>
</table>
| (Pts, Ptl) | {Con, Wor, 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 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 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:

<table>
<thead>
<tr>
<th>path</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>the origin itself: stay put!</td>
<td>0</td>
</tr>
<tr>
<td>all other vertices: unknown</td>
<td>infinity</td>
</tr>
</tbody>
</table>

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.)

• 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

• 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
• The Java version is in Graph.java
Example: Shortest Paths from Concord

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

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

- 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:
  ```plaintext
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

<table>
<thead>
<tr>
<th>push</th>
<th>stack contents (top to bottom)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E-124</td>
<td>E-124</td>
</tr>
<tr>
<td>E-162</td>
<td>E-162, E-124</td>
</tr>
<tr>
<td>E-215</td>
<td>E-215, E-162, E-124</td>
</tr>
<tr>
<td>E-104</td>
<td>E-104, E-215, E-162, E-124</td>
</tr>
<tr>
<td>E-119</td>
<td>E-119, E-104, E-215, E-162, E-124</td>
</tr>
<tr>
<td>E-10</td>
<td>E-10, E-160, E-119, E-104, E-215, E-162, E-124</td>
</tr>
<tr>
<td>E-50a</td>
<td><strong>E-50a, E-50b, E-10, E-160, E-119, E-104, E-215, E-162, E-124</strong></td>
</tr>
</tbody>
</table>

Evolution of the stack:

- one possible topological ordering
Another Topological Sort Example

Evolution of the stack:

push stack contents (top to bottom)

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**

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

Source: [www.tsp.gatech.edu/world/pictures.html](http://www.tsp.gatech.edu/world/pictures.html)

---

**Solving a TSP: Brute-Force Approach**

- Perform an exhaustive search of all possible tours.
- We can represent the set of all possible tours as a 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 Search

- Focus on the most promising paths through the tree of possible tours.
  - use a domain-specific function that estimates how good a given path 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 \( n = \text{problem size} \):

<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_3n )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>( n \log n ) time</td>
<td>( 4n \log_2n, n \log_3n + 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^5 + 5n^2 )</td>
<td>( O(n^c) )</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, 3! )</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>Problem Size</th>
<th>CPU 1</th>
<th>CPU 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>

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
Take-Home Lessons

• Computer science is the science of solving problems using computers.

• Java is one programming language that we can use when solving problems computationally.

• The key concepts transcend Java:
  • flow of control
  • variables, data types, and expressions
  • conditional execution
  • procedural decomposition
  • definite and indefinite loops
  • recursion
  • console and file I/O
  • memory management (stack, heap, references)

Take-Home Lessons (cont.)

• 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