More Graph Algorithms
Info 2950 - Fall 2008
Spanning Trees
Minimum Spanning Trees

• Suppose edges are weighted, and we want a spanning tree of *minimum cost*

• where the cost is the sum of edge weights
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it.
A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it.
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it
A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it.
A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it
3 Greedy Algorithms

A. Find a max weight edge – if it is on a cycle, throw it out, otherwise keep it
3 Greedy Algorithms

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm
3 Greedy Algorithms

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm
3 Greedy Algorithms

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm
3 Greedy Algorithms

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm
3 Greedy Algorithms

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm
B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm
3 Greedy Algorithms

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm
B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm
3 Greedy Algorithms

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm
3 Greedy Algorithms

B. Find a min weight edge – if it forms a cycle with edges already taken, throw it out, otherwise keep it

Kruskal's algorithm
3 Greedy Algorithms

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
3 Greedy Algorithms

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
3 Greedy Algorithms

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
3 Greedy Algorithms

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
3 Greedy Algorithms

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prim's algorithm
3 Greedy Algorithms

C. Start with any vertex, add min weight edge extending that connected component that does not form a cycle

Prime's algorithm
3 Greedy Algorithms

All 3 greedy algorithms give the same minimum spanning tree (assuming distinct edge weights)
Applications of MSTs

• In designing a network of computers (or phones, power distribution, etc.)
  ▪ Vertices are computers (or homes, cities)
  ▪ Edges are possible connections
  ▪ Weights represent the $ cost of installing each link
  ▪ MST is the lowest-cost way of connecting the nodes

• In approximation algorithms

• In image analysis
Traveling Salesperson

- Find a path of minimum distance that visits every city
Traveling Salesperson

- Following the MST, a salesperson could visit every city (but possibly multiple times), traversing every edge exactly twice.
- Thus the path produced by the MST is at most twice as costly as the optimal solution (which we cannot determine efficiently).
Another example: Image segmentation

- Goal: reduce an image to a small number of homogeneous regions ("segments")
Segmentation as a graph problem

- Represent an image as a graph
  - Vertices represent image pixels
  - Edges between adjacent pixels
  - Edge weights give difference in color between pixels

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>15</th>
<th>10</th>
<th>15</th>
<th>10</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>15</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>100</td>
<td>60</td>
<td>60</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>100</td>
<td>60</td>
<td>50</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>50</td>
<td>60</td>
<td>50</td>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>
Segmentation as a graph problem

- **Goal:** Find a small number of homogeneous regions
  - Or: Find a set of connected components in the graph, such that the sum of edge weights in each component is low
  - We can do this by finding a minimum spanning tree, and then removing a few high-weight edges
Segmentation as a graph problem

- **Goal:** Find a small number of homogeneous regions
  - Or: Find a set of connected components in the graph, such that the sum of edge weights in each component is low
  - We can do this by finding a minimum spanning tree, and then removing a few high-weight edges
Segmentation as a graph problem

• **Goal:** Find a small number of homogeneous regions
  - Or: Find a set of connected components in the graph, such that the sum of edge weights in each component is low
  - We can do this by finding a minimum spanning tree, and then removing a few high-weight edges
Segmentation as a graph problem

- **Goal:** Find a small number of homogeneous regions
  - Or: Find a set of connected components in the graph, such that the sum of edge weights in each component is low
  - We can do this by finding a minimum spanning tree, and then removing a few high-weight edges
Some results
More Graph Algorithms

• Search
  – depth-first search
  – breadth-first search

• Shortest paths
  – Dijkstra's algorithm

• Topological sort
Depth-First Search

• DFS: An algorithm for visiting every node of a graph, in a particular order (depth-first ordering)

• Choose a starting vertex, $v_1$

• Choose an edge that leads out of $v_1$ to a vertex $v_2$ that we haven’t visited yet

  • Repeat this step recursively: i.e. choose an edge that leads out of $v_2$ to a vertex $v_3$, then choose an edge out of $v_3$...

  • If there are no such edges, then backtrack to the node we came from, and try again
Depth-First Search

Order of node traversal: 1
Depth-First Search

Order of node traversal: 1, 2
Depth-First Search

Order of node traversal: 1, 2, 6
Depth-First Search

Order of node traversal: 1, 2, 6, 5
Depth-First Search

Order of node traversal: 1, 2, 6, 5
Depth-First Search

Order of node traversal: 1, 2, 6, 5, 7
Depth-First Search

Order of node traversal: 1, 2, 6, 5, 7, 3
Depth-First Search

Order of node traversal: 1, 2, 6, 5, 7, 3
Depth-First Search

Order of node traversal: 1, 2, 6, 5, 7, 3
Depth-First Search

Order of node traversal: 1, 2, 6, 5, 7, 3
Depth-First Search

Order of node traversal: 1, 2, 6, 5, 7, 3, 4
Depth-First Search

Order of node traversal: 1, 2, 6, 5, 7, 3, 4
Depth-First Search

Order of node traversal: 1, 2, 6, 5, 7, 3, 4
Depth-First Search

Order of node traversal: 1, 2, 6, 5, 7, 3, 4
Depth-First Search

Order of node traversal: 1, 2, 6, 5, 7, 3, 4
Application of depth-first search

• e.g. Navigating through a maze
  ▪ Represent each intersection, corner, and dead-end as a vertex, and each corridor as an edge
Application of depth-first search

- e.g. Navigating through a maze
  - Represent each intersection, corner, and dead-end as a vertex, and each corridor as an edge
Application of depth-first search

- e.g. Navigating through a maze
  - Represent each intersection, corner, and dead-end as a vertex, and each corridor as an edge
Application of depth-first search

- e.g. Navigating through a maze
  - Represent each intersection, corner, and dead-end as a vertex, and each corridor as an edge
Application of depth-first search

- e.g. Navigating through a maze
  - Represent each intersection, corner, and dead-end as a vertex, and each corridor as an edge
Application of depth-first search

• e.g. Navigating through a maze
  ▪ Represent each intersection, corner, and dead-end as a vertex, and each corridor as an edge
Application of depth-first search

• e.g. Navigating through a maze
  ▪ Represent each intersection, corner, and dead-end as a vertex, and each corridor as an edge
Application of depth-first search

• e.g. Navigating through a maze
  ▪ Represent each intersection, corner, and dead-end as a vertex, and each corridor as an edge
Application of depth-first search

• e.g. Navigating through a maze
  ▪ Represent each intersection, corner, and dead-end as a vertex, and each corridor as an edge
Application of depth-first search

- e.g. Navigating through a maze
  - Represent each intersection, corner, and dead-end as a vertex, and each corridor as an edge
Application of depth-first search

• e.g. Navigating through a maze
  ▪ Represent each intersection, corner, and dead-end as a vertex, and each corridor as an edge
Application of depth-first search

- e.g. Navigating through a maze
  - Represent each intersection, corner, and dead-end as a vertex, and each corridor as an edge
Application of depth-first search

- e.g. Navigating through a maze
  - Represent each intersection, corner, and dead-end as a vertex, and each corridor as an edge
Application of depth-first search

- e.g. Navigating through a maze
  - Represent each intersection, corner, and dead-end as a vertex, and each corridor as an edge
Application of depth-first search

- e.g. Navigating through a maze
  - Represent each intersection, corner, and dead-end as a vertex, and each corridor as an edge
Breadth-First Search

• Also visits every node of a graph, but in a different order

• Maintain a list $L = v_1, v_2, \ldots, v_n$ of the nodes visited so far

• Choose a starting vertex, $v_1$.
  • Initially, $L = v_1$, and $j = 1$.

• Visit nodes adjacent to $v_j$ (that have not yet been visited) and add them to the end of $L$. Then increment $j$.
  • When $j = n+1$, we’re done.
Breadth-First Search

Order of node traversal: 1
Breadth-First Search

Order of node traversal: 1, 2
Breadth-First Search

Order of node traversal: 1, 2, 6
Breadth-First Search

Order of node traversal: 1, 2, 6, 3
Breadth-First Search

Order of node traversal: 1, 2, 6, 3
Breadth-First Search

Order of node traversal: 1, 2, 6, 3, 4
Breadth-First Search

Order of node traversal: 1, 2, 6, 3, 4, 5
Breadth-First Search

Order of node traversal: 1, 2, 6, 3, 4, 5
Breadth-First Search

Order of node traversal: 1, 2, 6, 3, 4, 5, 7
Breadth-First Search

Order of node traversal: 1, 2, 6, 3, 4, 5, 7
Breadth-First Search

Order of node traversal: 1, 2, 6, 3, 4, 5, 7
Breadth-First Search

Order of node traversal: 1, 2, 6, 3, 4, 5, 7
Breadth-First Search

Order of node traversal: 1, 2, 6, 3, 4, 5, 7
Applications of Breadth First Search

• Breadth-first search can be used to find the path with the fewest edges between two nodes
  ▪ e.g. How to fly from ITH to SFO with the fewest layovers?
Applications of Breadth First Search

- Breadth-first search can be used to find the path with the fewest edges between two nodes
  - e.g. How to fly from ITH to SFO with the fewest layovers?
Applications of Breadth First Search

• Breadth-first search can be used to find the path with the fewest edges between two nodes
  ▪ e.g. How to fly from ITH to SFO with the fewest layovers?

L = ITH, DTW, LGA
Applications of Breadth First Search

• Breadth-first search can be used to find the path with the fewest edges between two nodes
  ▪ e.g. How to fly from ITH to SFO with the fewest layovers?

L = ITH, DTW, LGA, PHL, MIA

\[L = \text{ITH, DTW, LGA, PHL, MIA}\]
Applications of Breadth First Search

- Breadth-first search can be used to find the path with the fewest edges between two nodes
  - e.g. How to fly from ITH to SFO with the fewest layovers?

\[ L = \text{ITH, DTW, LGA, PHL, MIA} \]
Applications of Breadth First Search

- Breadth-first search can be used to find the path with the fewest edges between two nodes
  - e.g. How to fly from ITH to SFO with the fewest layovers?

$L = ITH, DTW, LGA, PHL, MIA, SFO$
Applications of Breadth First Search

• Breadth-first search can be used to find the path with the fewest edges between two nodes
  - e.g. How to fly from ITH to SFO with the fewest layovers?

L = ITH, DTW, LGA, PHL, MIA, SFO
Weighted Shortest Paths

- What if the edges are weighted, and we want to find the lowest-cost path between two nodes in a graph?
  - Can be solved efficiently using Dijkstra’s algorithm

- The basic idea: weights are additive
  - If we know the best path from LGA to SFO, and the best path from DTW to SFO, we can find the best path from ITH to SFO
Dijkstra’s Algorithm

• Dijkstra’s algorithm maintains two data structures,
  ▪ A table of the best path known so far from every vertex to dest
  ▪ A set $X$ of nodes, for which we know the actual best path to dest

• Set $X=\{\text{dest}\}$. Then,
  ▪ Update the table for any nodes with edges into $X$
  ▪ Find an edge leading into $X$ with the smallest weight. Add the source of the edge to $X$. 

```
src = 1
     2.4
      2
1.5  0.1
     4
     3.1
     dest = 3
```
Shortest Paths

Graph with nodes labeled SFO, MIA, DTW, ITH, LGA, and PHL, connected with edges and distances as follows:
- SFO to DTW: 150
- DTW to ITH: 1000
- ITH to LGA: 300
- LGA to PHL: 50
- PHL to MIA: 150
- MIA to SFO: 100
- SFO to MIA: 500
- MIA to DTW: 50
- DTW to SFO: 150

Distances in miles or kilometers, depending on context.
More Graph Terminology

- **A path** is a sequence $v_0, v_1, v_2, \ldots, v_p$ of vertices such that $(v_i, v_{i+1}) \in E$, $0 \leq i \leq p - 1$
- The **length of a path** is its number of edges
  - In this example, the length is 5
- **A path** is **simple** if it does not repeat any vertices
- **A cycle** is a path $v_0, v_1, v_2, \ldots, v_p$ such that $v_0 = v_p$
- **A cycle** is **simple** if it does not repeat any vertices except the first and last
- **A graph** is **acyclic** if it has no cycles
- A directed acyclic graph is called a **dag**
Is This a Dag?

- Intuition:
  - If it’s a dag, there must be a vertex with indegree zero – why?
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears
Is this a dag?

- **Intuition:**
  - If it’s a dag, there must be a vertex with indegree zero – why?
- **This idea leads to an algorithm**
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears
Is this a dag?

- Intuition:
  - If it’s a dag, there must be a vertex with indegree zero – why?
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears
Is this a dag?

- Intuition:
  - If it’s a dag, there must be a vertex with indegree zero – why?
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears
Is this a dag?

- Intuition:
  - If it’s a dag, there must be a vertex with indegree zero – why?
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears
Is this a dag?

- Intuition:
  - If it’s a dag, there must be a vertex with indegree zero – why?
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears
Is this a dag?

- Intuition:
  - If it’s a dag, there must be a vertex with indegree zero – why?
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears
Is this a dag?

- Intuition:
  - If it’s a dag, there must be a vertex with indegree zero – why?

- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears
Is this a dag?

• Intuition:
  ▪ If it’s a dag, there must be a vertex with indegree zero – why?
• This idea leads to an algorithm
  ▪ A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears
Topological Sort

• We just computed a topological sort of the dag
  ▪ This is a numbering of the vertices such that all edges go from lower- to higher-numbered vertices

Useful in scheduling with constraints on precedence and prerequisites