

Graphs

CIS*2520 Summer 2006

Overview

- One of the most important models for specifying more complex relationships
 - structures
 - patterns of connectivity between cities
 - network routing
 - processes
 - flow of control through a program
 - communication patterns
 - relationships
 - dependencies between expressions in a mathematical expression
- Convenient graphical representation
 - points (or circles) representing individual components of interest
 - line segments between the points representing relationships or flow of control

CIS*2520 (506)

Terminology

▪ **Formal Definition:**

- a graph $G=V,E$ is a set of vertices (or “nodes”) V together with a set of edges (or “arcs”) E which are pairs of distinct vertices from V
 - $E = \{ (a, b) \mid a, b \in V \}$

▪ **Undirected Graph**

- $(u,v) \in E \Leftrightarrow (v,u) \in E$
- i.e. edge relationships are symmetric

▪ **Directed Graph (“digraph”)**

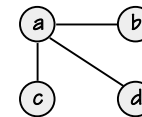
- $\exists (u,v) \in E$ s.t. $(v,u) \notin E$
- i.e. (u,v) is ordered
- Directed Acyclic Graphs (DAGs) are an important class of graph

CIS*2520 (506)

Formal and Visual Graphs

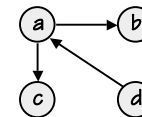
▪ Undirected

- $V = \{ a, b, c, d \}$
- $E = \{ (a,b), (b,a), (a,c), (c, a), (a,d), (d,a) \}$



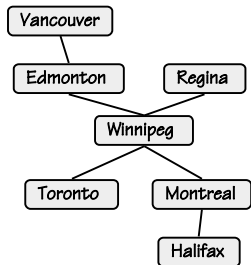
▪ Directed

- $V = \{ a, b, c, d \}$
- $E = \{ (a,b), (a, c), (d,a) \}$

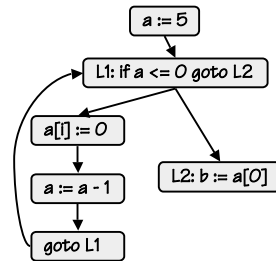


CIS*2520 (506)

Undirected and Directed Graphs



Air routes between cities
(undirected)



Flow of control in a program
(directed)

CIS*2520 (506)

Graphs in Practice

- Collaboration graph
 - vertices correspond to researchers
 - undirected edges represent co-authorship on papers
 - note: edges modeling relationships, not connections or paths
- Inheritance graph
 - vertices are classes
 - directed edges indicate one class extending another
- City graph
 - vertices are intersections (or dead-ends)
 - undirected/directed edges represent streets (two and one-way) connecting them

CIS*2520 (506)

Graphs in Practice

- Electrical wiring as a graph
 - vertices are connection points
 - edges represent pathways between connection points
- Transportation network graph
 - vertices are airports (or other points of arrival departure)
 - edges indicate where it is possible to travel from one point to another
 - may wish to consider edges with weights to model cost, distance, etc.
- The Internet
 - vertices are computers
 - edges represent connections between computers

CIS*2520 (506)

Terminology (cont.)

- Adjacent Vertices
 - $u, v \in V$ s.t. $(u, v) \in E$
- Path
 - a sequence of distinct vertices, each one adjacent to the one before and after it in sequence
- Cycle
 - a path of at least 3 vertices in which the last vertex is adjacent to the first

CIS*2520 (506)

Terminology (cont.)

Connected Graph

- a graph is connected if there is a path from any vertex to any other vertex
- *recall: a Tree is a connected graph with no cycles*
 - trees as we have discussed previously are referred to as **rooted trees**
 - distinguish from **free trees** which are the more general graph-based definition

Endpoints of an Edge

- **origin**: the first endpoint of a directed edge
- **insertion**: the second endpoint of a directed edge (aka. *destination*)

Incident Edge

- an edge is incident on a vertex if the vertex is one of the edge's endpoints

CIS*2520 (506)

Terminology (cont.)

Outgoing Edges

- directed edges for which a given vertex is the origin

Incoming Edges

- directed edges for which a given vertex is the insertion

Degree

- number of incident edges of a vertex v ; $deg(v)$
- **in-degree**
 - number of incoming edges of a vertex v ; $indeg(v)$
- **out-degree**
 - number of outgoing edges of a vertex v ; $outdeg(v)$

CIS*2520 (506)

Terminology (cont.)

Subgraph

- given a graph $G = (V, E)$, a graph $H = (V', E')$ where $V' \subseteq V$, $E' \subseteq E$ is termed a subgraph of G
- a **spanning subgraph** of G is a subgraph of G for which $V' = V$ (i.e. contains all the vertices of G)

Weighted Graph

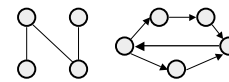
- a graph for which there is a numerical weight associated with edge
- commonly used for modeling costs, time or probabilities

Simple Graph

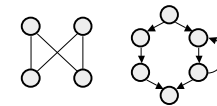
- $\forall v \in V, (v, v) \notin E$
- no edges from a vertex to itself

CIS*2520 (506)

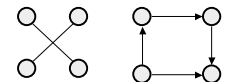
Graph Examples



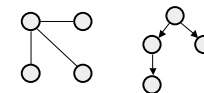
connected graphs



cyclic graphs



disconnected graphs



trees

CIS*2520 (506)

Graph ADT Considerations

- Definition
 - ...
- Operations
 - general methods
 - *numVertices, numEdges*
 - *degree, adjacent, incident, areAdjacent*
 - directed edges
 - *origin, insertion, isDirected*
 - *indegree, outdegree*
 - updating
 - *insertEdge, insertDirectedEdge, insertVertex, removeEdge, setEdge*
 - iterators
 - many, many possibilities (vertices, edges, adjacentvertices, etc.)
- these are by no means definitive or comprehensive
 - see chapter 10 text for additional discussion

CIS*2520 (506)

Graph Implementation

- There are two major ways to represent a graph in implementation, each with its own pros and cons
 - node lists
 - edge list
 - list of vertices, list of edges
 - references from edge objects to vertices which are incident
 - adjacency list
 - edge list with adjacency information associated with vertices
 - adjacency matrix
 - vertices are indexed (0..n-1)
 - adjacency relationships stored in a $n \times n$ matrix where $adj[i][j] = 1$ if there exists an edge (i,j) in the graph
- Note:
 - it is common and convenient to consider all vertices as being indexed 0 to n-1 (or 1 to n) regardless of other labeling

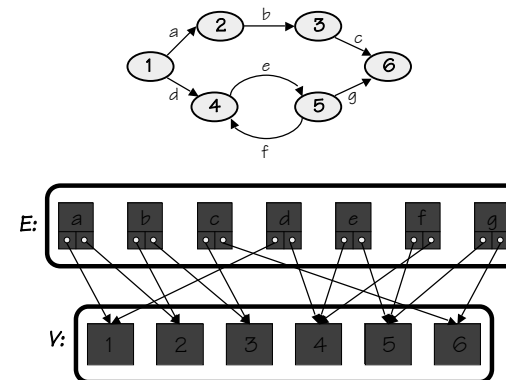
CIS*2520 (506)

Edge List Structure

- Given a graph $G = (V, E)$
 - each $v \in V$ is represented by a Vertex ADT
 - vertex is indexed/labeled and can store a data reference if desired
 - each $e \in E$ is represented by an Edge ADT
 - edge is indexed/labeled and can store a data reference if desired (e.g. weight, input, etc.)
 - also contains references to its origin and insertion vertex
 - Vertex and Edge ADTs each stored in some suitable container (list, table, etc.)
- Advantages:
 - simple
- Disadvantages:
 - inefficient
 - incidence is explicit (easy), computing adjacency is more involved

CIS*2520 (506)

Edge List Example



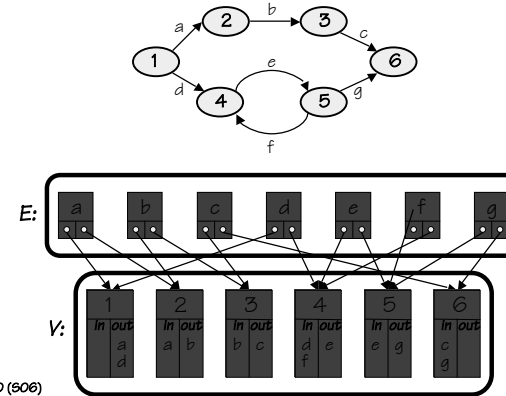
CIS*2520 (506)

Adjacency List Structure

- Given a graph $G = (V, E)$
 - similar to edge list, we have a container of Vertex and Edge ADTs
 - each vertex additionally contains references to the incoming/outgoing edges
 - there are potential issues and/or simplifications depending on whether the graph is directed, undirected or mixed
- Advantages:
 - ready access to both incidence and adjacency information
 - relatively efficient for all operations
- Disadvantages:
 - somewhat more complex to realize
 - lot of reference juggling

CIS*2520 (506)

Adjacency List Example



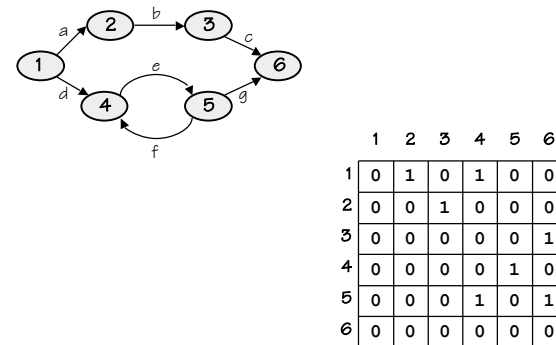
CIS*2520 (506)

Adjacency Matrix Structure

- Given a graph $G = (V, E)$
 - define a $|V| \times |V|$ matrix such that entry (i, j) is defined iff $(i, j) \in E$
 - may still use basic edge list structure as well
 - depending on what information is required it is possible to represent a graph completely with just the matrix in many cases
- Advantages:
 - constant time adjacency information
 - mathematically convenient for many algorithms
- Disadvantages:
 - many operations made less efficient (requiring row traversal)
 - significant space requirements

CIS*2520 (506)

Adjacency Matrix Example



CIS*2520 (506)

Efficiency of Graph Structures

size, isEmpty, replace, swap	$O(1)$	$O(1)$	$O(1)$
numVertices, numEdges	$O(1)$	$O(1)$	$O(1)$
insertEdge, removeEdge	$O(1)$	$O(1)$	$O(1)$
insertVertex	$O(1)$	$O(1)$	$O(n^2)$
removeVertex	$O(m)$	$O(deg(v))$	$O(n^2)$
areAdjacent	$O(m)$	$O(\min(deg(u), deg(v)))$	$O(1)$
endVertices, origin, insertion, isDirected, degree	$O(1)$	$O(1)$	$O(1)$
vertices*	$O(n)$	$O(n)$	$O(n)$
edges*	$O(m)$	$O(m)$	$O(m)$
adjacentVertices*, adjacentEdges*	$O(m)$	$O(deg(v))$	$O(n)$

CIS*2520 (506)

* implemented as an iteration (iterator)

Graph Structure Notes

- So what implementation do we choose?
 - adjacency matrix was developed earlier
 - adjacency list appears superior in most situations
- Highly problem dependent as it turns out
 - adjacency matrix
 - favours high edge count
 - for static graphs this is fairly easy to manipulate
 - if graph is to be dynamic this is very costly
 - consumes more space
 - adjacency list
 - favours lower edge count
 - much easier to handle dynamically
 - if the most significant information represented by the graph is adjacency and that is referred to frequently, this will be less time efficient
- As with all things it requires some thought.

CIS*2520 (506)

Graph Traversal

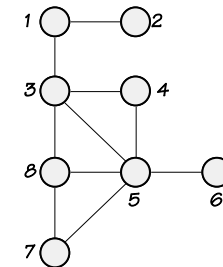
- Visit all the vertices in a graph in some systematic way (recall: tree traversal)
 - many orders we could visit the nodes
 - two are of particular significance
- Depth First Traversal
 - similar to preorder traversal of a tree
 - recurses as far as it can before exploring other options bottom-up
 - highly adventurous traversal
- Breadth First Traversal
 - similar to a level-by-level traversal of a tree
 - considers all adjacent nodes before considering adjacency of each of these nodes in turn
 - more conservative

CIS*2520 (506)

Depth First Traversal

```

depthFirst(u)
{
    visit(u)
    for all unvisited v s.t.
        (u,v) in E
        depthFirst(v)
}
    
```



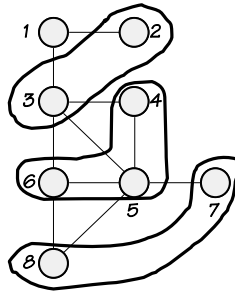
- Note:
 - need a mechanism for tracking visited vertices
 - iterative implementation is relatively straightforward using a stack to store unvisited vertices
- Example
 - assume edges are ordered clockwise from the right

CIS*2520 (506)

Breadth First Traversal

```

breadthFirst(G=(V,E))
{
  add u to empty queue
  while queue not empty
  {
    add all unvisited
      v s.t. (u,v) in E
      to queue
    get next vertex from queue
    visit(u)
  }
}
    
```



- You can picture this as visiting all nodes 1 step from starting point, then all nodes 2 steps from starting point, ...

CIS*2520 (506)

Traversal Notes

- As outlined, these traversals do not handle disconnected graphs
 - need an enclosing loop
 - for all unvisited v in V ...
 - action depends on which traversal is being done
 - DFS: do DFS from v
 - BFS: add v to queue before starting loop while queue not empty
- What about directed graphs?
 - if you implement the above appropriately it's the same thing
 - minor variation otherwise (see text)
- See text 10.4 for additional discussion

CIS*2520 (506)

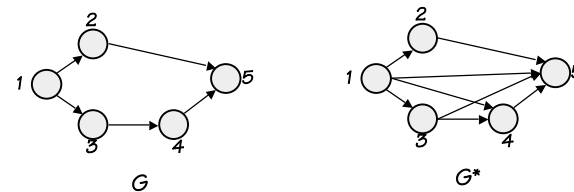
Graph Algorithms

- Transitive closure
 - a graph representing path relationships
- Shortest paths
 - minimum cost between two vertices along any path
- Minimum spanning trees
 - a subgraph with minimum cost that links all vertices
- Topological sort
 - order $v \in V$ s.t. for every edge $(v_i, v_j) \in E, i < j$
 - see text 10.5
- Graph Colouring, Hamiltonian circuit, isomorphism, etc., etc., etc., ...

CIS*2520 (506)

Transitive Closure

- A graph representing reachability in directed graphs
- Given a graph $G=(V,E)$, the **transitive closure** $G^*=(V,E')$ is defined to be the directed graph:
 - with the same vertices as G
 - $(u,v) \in E'$ if there is a directed path from u to v in G
- e.g.



CIS*2520 (506)

Implementing Transitive Closure

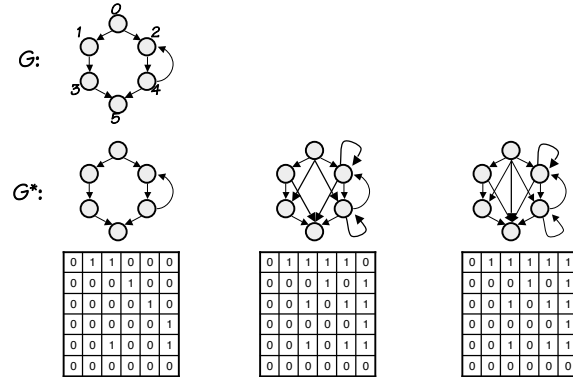
- Floyd Warshall Algorithm
- The general idea is to incrementally construct G^*
 - start with a copy of G
 - adding edges for paths of length 2 iteratively
 - net path length of closure grows by 1 each iteration, so iterate n times
 - traverse matrix adding edge (i,j) to G^* if (i,k) and (k,j) are in G^*
- Easy with an adjacency matrix (do it with logical operations)
 - the process is conceptually identical with adjacency list

```

transitiveClosure(G)
{
  G := G*
  for k := 1 to n
    for i := 1 to n
      for j := 1 to n
        if (i,k) AND (k,j) in G*
          G* := G* + (i,j)
}
  
```

CIS*2520 (506)

Transitive Closure Example



CIS*2520 (506)

Shortest Path

- Shortest/minimum cost path between two vertices in a graph (directed/undirected)
 - If edges are not weighted
 - we can simply use BFS (justify this)
 - If edges are weighted
 - the weight of a path is the sum of weights on the edges traversed on that path
 - minimum weight path not necessarily related to number of edges (long path may have lowest weight)

CIS*2520 (506)

Implementing Shortest Path

- Dijkstra's Algorithm
- Given a graph $G=(V,E)$ and a starting vertex v
 - label every vertex $u \in V$ with $D[u]$, the length of the shortest known path from v to u
 - initially $D[v] = 0$, $D[u] = +\infty$
 - iteratively expand known path(s) by making a greedy choice of a new u and re-computing D for nodes adjacent to u

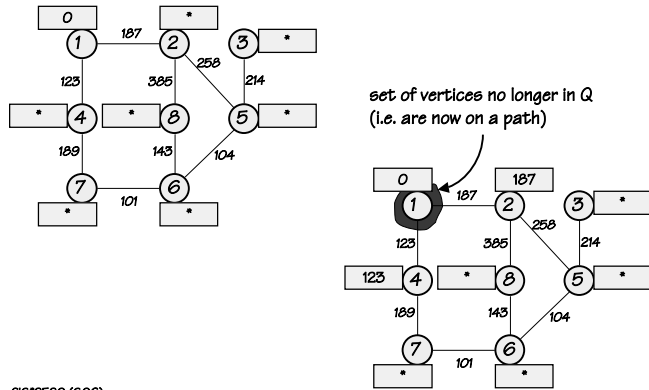
```

shortestPath(G=(V,E), v)
{
  D[v] := 0
  for all u in V s.t u != v
    D[u] := MAX
  Q := V

  while Q not empty
    u := remove q in Q, min D[q]
    for all z adjacent to u
      if (D[u] + w(u,z) < D[z])
        D[z] := D[u] + w(u,z)
}
  
```

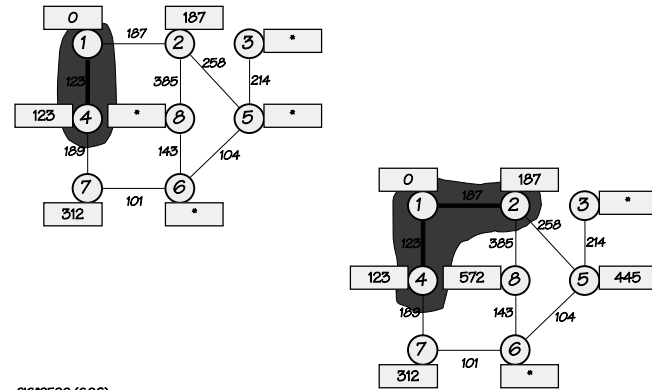
CIS*2520 (506)

Shortest Path Example



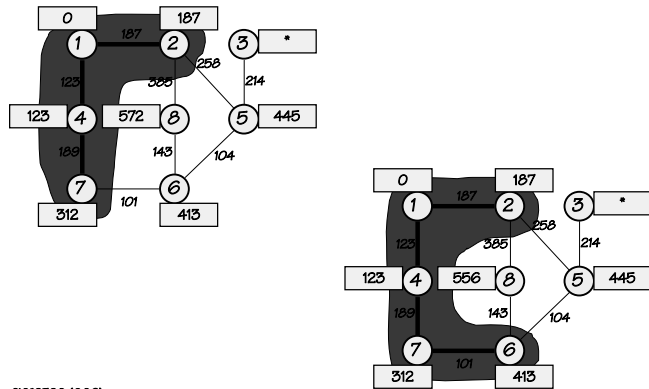
CIS*2520 (506)

Shortest Path Example (cont.)



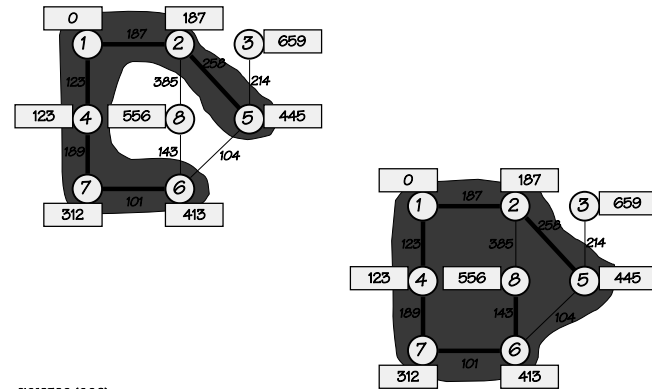
CIS*2520 (506)

Shortest Path Example (cont.)



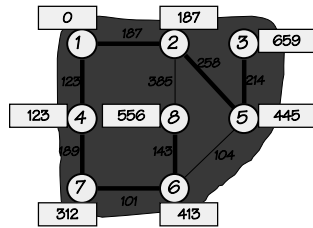
CIS*2520 (506)

Shortest Path Example (cont.)

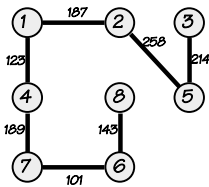


CIS*2520 (506)

Shortest Path Example (cont.)



done:
shortest path from v_1 to
all other nodes in G



CIS*2520 (506)

Notes on Shortest Path

- This is an example of a **greedy algorithm**
 - solve a problem by making the best choice of those available at each step (i.e. make a locally greedy choice)
 - may problems can be solved in this manner
 - depending on nature of problem, may not produce an optimum result (in this case it does --- see text)
- How to find vertex label $D[v]$ with minimum cost?
 1. ordered List (text uses a PriorityQueue --- same idea)
 - take first entry for next choice of vertex
 2. search for minimum in unordered collection
 - in either case, must update $D[v]$ values of vertices still in list when necessary
 - what effect does your choice make on overall efficiency?

CIS*2520 (506)

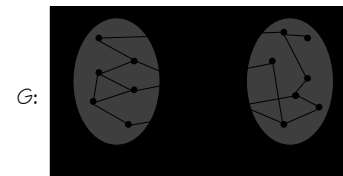
Notes on Shortest Path (cont.)

- Length of shortest path vs. actual path
 - if all we track are the $D[v]$ values, we only get distance of shortest path
 - if we want the actual path, we need to also track edges
 - note that an edge is only a "candidate" until the adjacent vertices are both in the set of shortest paths
- Implementation with adjacency matrix vs. adjacency list
 - a lot of set-like notation and manipulation of containers
 - more direct translation to implement this with adjacency lists, however it is possible to adapt it for either representation
- How would you modify this to find shortest path between two specific vertices?
 - or would you?

CIS*2520 (506)

Minimum Spanning Tree

- Minimum distance/cost connected subgraph containing all vertices
 - superficially similar to shortest path
 - subtle difference: we want a minimum *total* distance between all vertices
- Solution relies on a surprisingly simple result:
 - given a graph $G=(V,E)$ and any non-empty partitioning of $V = V_1, V_2$ the minimum weight edge connecting some node in V_1 to some node in V_2 is in the minimum spanning tree of G (*proof left as exercise*)



CIS*2520 (506)

Implementing Minimum Spanning Tree

- Kruskal's Algorithm

- Given a graph $G=(V,E)$

- start with all vertices in their own cluster
- iteratively merge two clusters with minimum weight edge between them (that edge is in minimum spanning tree)
- when we are left with one cluster containing all vertices we have a minimum spanning tree

```

minimumSpanningTree(G)
{
  for all v in V
    C(v) := v
  initialize T (edge set) empty

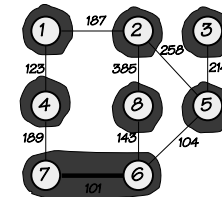
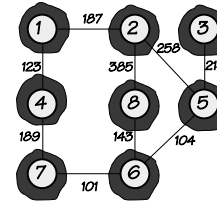
  while |T| < (n-1)
    choose minimum (u,v) remain
    if (C(v) != C(u))
      add (u,v) to T
      merge C(v) and C(u)
}
    
```

- Prim-Jarnik Algorithm

- variation on shortest path

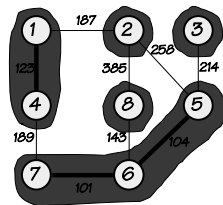
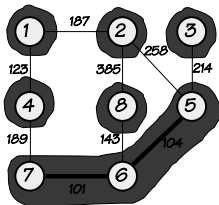
CIS*2520 (506)

Minimum Spanning Tree Example



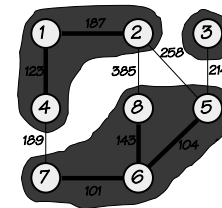
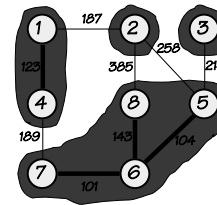
CIS*2520 (506)

Minimum Spanning Tree Example (cont.)



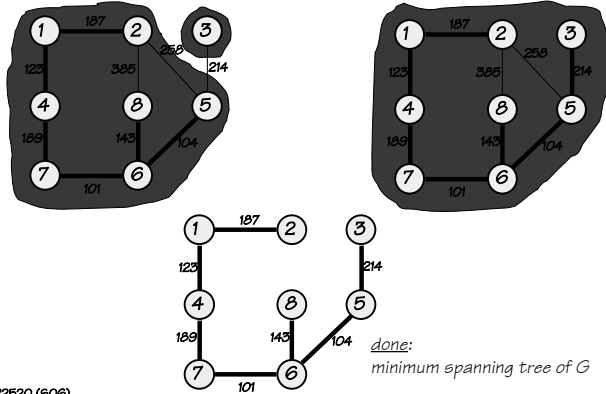
CIS*2520 (506)

Minimum Spanning Tree Example (cont.)



CIS*2520 (506)

Minimum Spanning Tree Example (cont.)



Notes on Minimum Spanning Tree

- Another example of a greedy algorithm
- Choosing minimum weight edges
 - ordered List (or PriorityQueue) vs. search
 - in either case we need to initialize a container with all edges prior to iterating
 - by removing one edge per iteration we are sure to consider each edge at most once
- Representation
 - more direct translation of algorithm using adjacency lists; however can be adapted to list or matrix representation
- Another approach: Prim-Jarnik Algorithm (see text)
 - grow MST from a seed vertex (minor modification of shortest path)

CIS*2520 (506)

Graph Algorithms

- Effect of representation on ease of implementation
 - adjacency list vs. adjacency matrix
- Efficiency issues
 - computational complexity
 - how does representation affect this?
- Implications for directed vs. undirected graph
 - transitive closure defined in terms of DAGs
 - trivial to apply to undirected graphs (why?)
 - shortest path/minimum spanning tree assumed undirected
 - what changes would have to be made in order to apply these algorithms to directed graphs?

CIS*2520 (506)