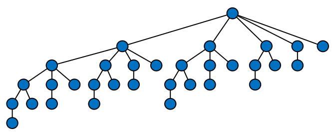
Lecture 21. Graph Algorithms, Connectivity Analysis

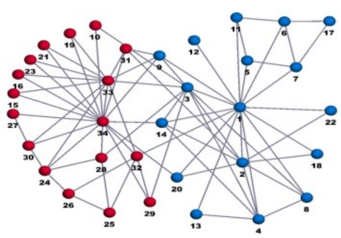
CpSc 8400: Algorithms and Data Structures
Brian C. Dean

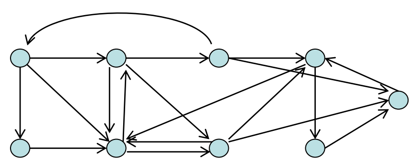


School of Computing
Clemson University
Spring, 2016

Graphs (Networks)

- A **graph** is comprised of nodes (a.k.a. vertices) and edges. Each edge joins a pair of vertices.
- Graphs can be either undirected or directed; by default "graph" == "undirected graph".
- Graphs are extremely common in CS and algorithms, since they can model a very wide range of problems and applications.



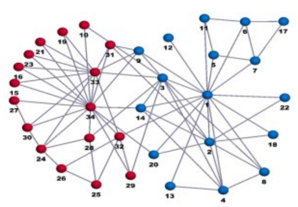


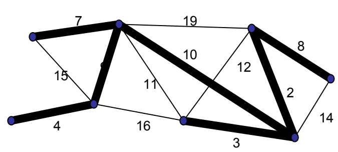
Graph Types

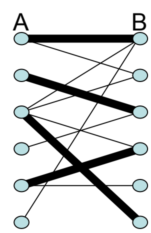
- Graph theorists have studied many different special types of graphs.
- Some simple examples of special classes of graphs we'll tend to see include:
  - The path (P_n)
  - The cycle (C_n)
  - The complete graph (K_n)
  - Trees and forests
  - Bipartite graphs
  - Planar graphs

Some Classical Graph Problems

- [Shortest Paths]: Find the shortest path between two nodes, or from one node to all other nodes.
- [Minimum Spanning Trees]: Find a min-cost subset of edges that connects together all nodes.
- [Matchings]: Pair up as many nodes as possible, or pair up all nodes at minimum total cost.
- [Flow / Routing]: Route a maximum amount of some commodity through a capacitated network, possibly at minimum total cost.

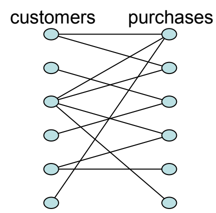


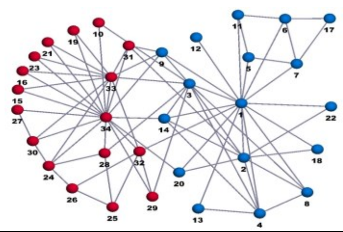


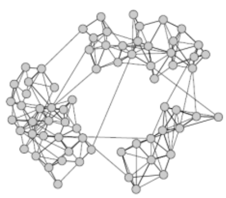


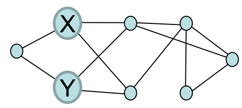
Network Analysis / Data Mining

- [Similarity / Connectivity]: How similar are nodes X and Y if edges connect directly-related elements?
- [Clustering]: Does a graph break naturally into several large “clusters”?
- [Centrality]: Find nodes that are well-connected with all other nodes.









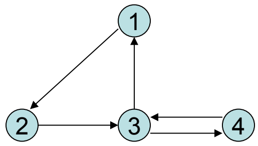
Terminology

- Walk: series of nodes connected by edges.
  - Also fine to think of it as a sequence of edges.
  - In a digraph, we tend to focus on directed walks.
- Path: a “simple” walk (visits no node twice).
- Cycle: a path that starts and ends at the same node.
- Connected components
- Strongly connected components in a digraph
  - Two nodes i and j belong to the same SCC if there’s a directed path from i to j and from j to i.

6

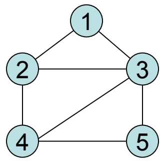
Graph Representation

- Two main ways to represent a graph:
  - An **adjacency matrix** (good for **dense** graphs)



	1	2	3	4
1	0	1	0	0
2	0	0	1	0
3	1	0	0	1
4	0	0	1	0

  - **Adjacency lists** (ideal for **sparse** graphs)



1: 2 3
2: 1 3 4
3: 1 2 4 5
4: 2 3 5
5: 3 4

- Unless otherwise stated, we'll assume our graphs are represented using adjacency lists.

7

Simple Connectivity Questions

- Some of the most fundamental graph algorithms related to issues of connectivity:
  - Are nodes i and j connected by some path?
  - If so, determine such a path.
  - In a digraph, is there a directed path from i to j?
  - Does a (directed) graph have a (directed) cycle?
  - Partition a graph into its connected components.
  - Partition a digraph into its strongly connected components.
- We can answer all of these questions in linear time using a remarkably versatile algorithm known as **depth-first search**.

8

Depth-First Search (DFS)

- ```
  DFS-Visit(i):
  Status[i] = 'visited'
  For all j such that (i,j) is an edge:
     If Status[j] = 'unvisited':
        Pred[j] = i
        DFS-Visit(j)
  ```

- ```
  Full-DFS:
  For all i: Pred[i] = null, Status[i] = 'unvisited'
  For all i: If Status[i] = 'unvisited': DFS-Visit(i)
  ```

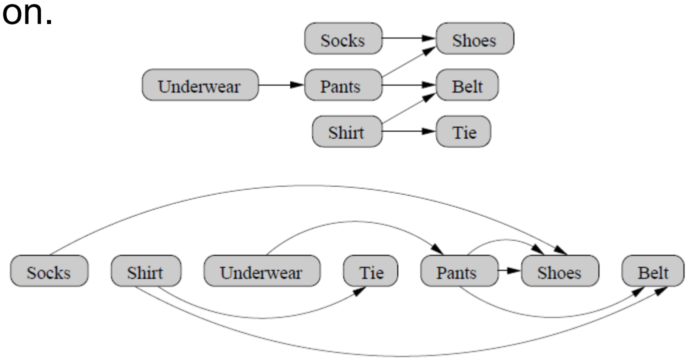
- Works in undirected and directed graphs.
- Full-DFS takes O(m + n) time since it spends O(1) time on each node and edge.

Depth-First Search (DFS)

- Full-DFS gives us an easy way to partition an undirected graph into its connected components.
- The pred[i] pointers define what is called a depth-first search tree.
- To find a path from i to j (if it exists):
  - Intialize pred and status values for all nodes.
  - Call DFS-Visit(i).
  - Then follow pred pointers backward from j to i

Topological Sorting

- Directed Acyclic Graphs (DAGs) are often used to model systems with “precedence” constraints.
- Topological sorting is the process of ordering the nodes of a DAG so all edges point a consistent direction.



Topological Sorting

- There are several ways to topologically sort in O(m + n) time; for example:
  - Find a node with no incoming edges, add it next to the ordering, remove it from our graph, repeat.
  - If we ever find that every node has an incoming edge, then our graph must contain a cycle (so this gives an alternate way to do cycle detection).
- Depth-first search gives us another very simple topological sorting algorithm…

Topological Sorting

- Let's associate with each node i a "discovery time" d[i] and a "finishing time" f[i]:

```
DFS-Visit(i):
Status[i] = 'visited'
d[i] = current_time, Increment current_time
For all j such that (i,j) is an edge:
   If Status[j] = 'unvisited':
      Pred[j] = i
      DFS-Visit(j)
f[i] = current_time, Increment current_time
```

- To topologically sort a DAG, just perform a Full-DFS and then output nodes in reverse order of finishing times.
  - We can output each node when it is "finished", or we can sort the nodes by their finishing times (in linear time with counting sort) as a post-processing step.

Topological Sorting

- **Claim**: In a DAG with an edge from node i to node j, we will have f(i) > f(j).
- **Proof**: Consider two cases:

  (a) Full-DFS visits i first.

  (b) Full-DFS visits j first.

  In both cases, we have f(i) > f(j) as long as our graph contains no directed cycles (which it doesn't, since it's a DAG!)

Cycle Detection

- Suppose we modify DFS-Visit as follows:

```
DFS-Visit(i):
Status[i] = 'pending'
For all j such that (i,j) is an edge:
   If Status[j] = 'pending': cycle detected!
   If Status[j] = 'unvisited':
      Pred[j] = i
      DFS-Visit(j)
Status[i] = 'visited'
```

- Full-DFS will now detect the existence of a cycle in a directed graph in O(m + n) time and in an undirected graph in only O(n) time!

Strongly-Connected Components

- We have already seen how DFS can be used to partition an undirected graph into its connected components.
- Although it's slightly less obvious, we can also use DFS to partition a directed graph into its strongly connected components!
  - Recall that two nodes i and j belong to the same strongly connected component if there is a directed path from both i to j and from j to i.
- Any thoughts?

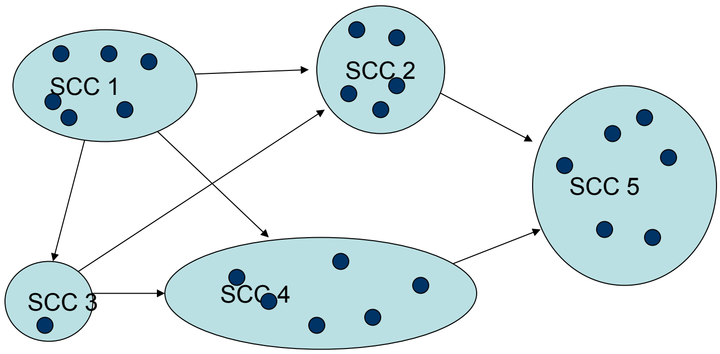
Strongly-Connected Components

- Consider this algorithm:
  - Full-DFS
  - Reverse all the edges in our graph
  - Full-DFS again, processing nodes in decreasing order of finishing time (according to the 1st DFS).
- **Claim**: In the second Full-DFS, each call to DFS-Visit will end up visiting precisely the set of nodes in one of the strongly connected components in our graph!
- So we can partition a graph into its strongly connected components in O(m + n) time.

Strongly-Connected Components

- Consider this algorithm:
  - Full-DFS
  - Reverse all the edges in our graph
  - Full-DFS again, processing nodes in decreasing order of finishing time (according to the 1st DFS).
- Key realization:
  - If we contract each strong component down to a single node, what remains is a DAG.
  - The algorithm above is just topologically sorting this DAG using our original “repeatedly remove a node with no incoming edges” approach!

Strongly-Connected Components



Claim: For any two SCCs A and B with edges from A to B, $\max_{i \in A} f[i] > \max_{j \in B} f[j]$

Finding Bridges

- A **bridge** is an edge whose removal breaks a graph into two pieces.
- An **articulation node** is a node whose removal breaks a graph into two pieces.
- We can modify DFS to find all bridges and articulation nodes in a graph in O(m + n) time.
- Let's focus on bridges for now (the approach for articulation nodes is quite similar).

Finding Bridges

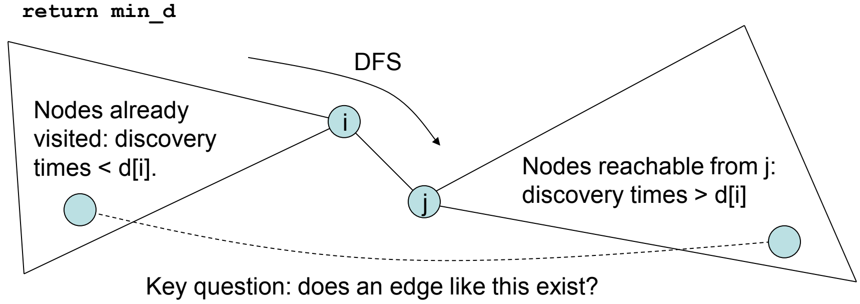
- Modify DFS-Visit(i) so it returns the minimum discovery time of all nodes encounted when visiting i and everything reachable from i:

```
DFS-Visit(i):
   Status[i] = 'visited'
   d[i] = current_time, Increment current_time
   min_d = d[i]
   For all j such that (i,j) is an edge:
      If Status[j] = 'unvisited':
         Pred[j] = i
         DFS-Visit(j)
      min_d = min(d[j], min_d)
   return min_d
```

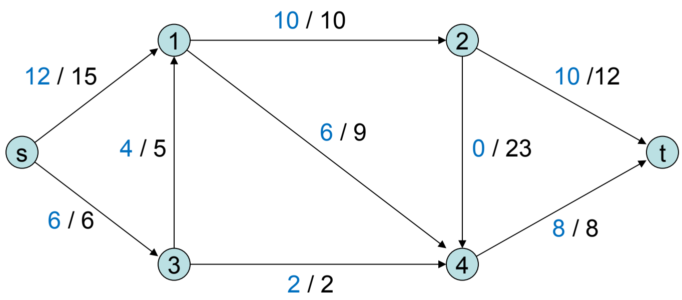
- Now how do we detect bridges?

Finding Bridges

```
DFS-Visit(i):
   Status[i] = 'visited'
   d[i] = current_time, Increment current_time
   min_d = d[i]
   For all j such that (i,j) is an edge:
      If Status[j] = 'unvisited':
         Pred[j] = i
         If DFS-Visit(j) > d[i] then (i,j) is a bridge!
      min_d = min(d[j], min_d)
   return min_d
```

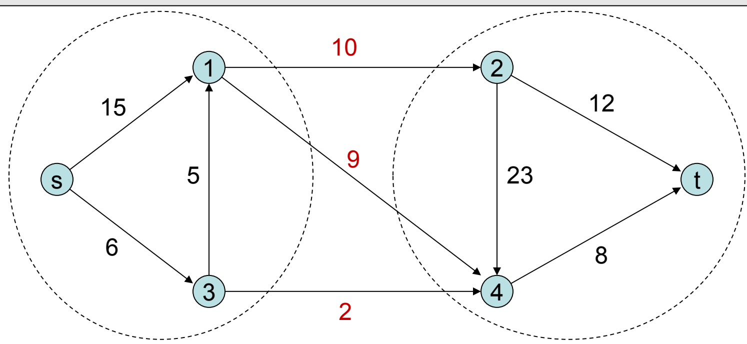


Towards Higher Orders of Connectivity: The Maximum Flow Problem



Given a directed graph with capacities on edges, what is the maximum amount of flow that can be shipped from a source node s to a sink node t? 18 units

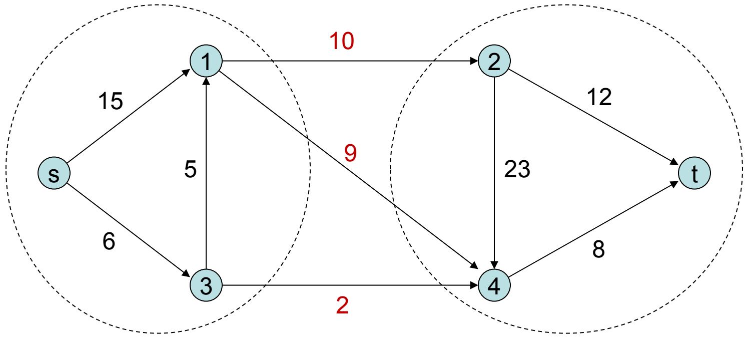
s-t Cuts



- An s-t cut is a partition of the node in our graph into two sets, one containing s and the other containing t.
- The capacity of a cut is the sum of the capacities of the edges crossing the cut in the s → t direction.

24

s-t Cuts

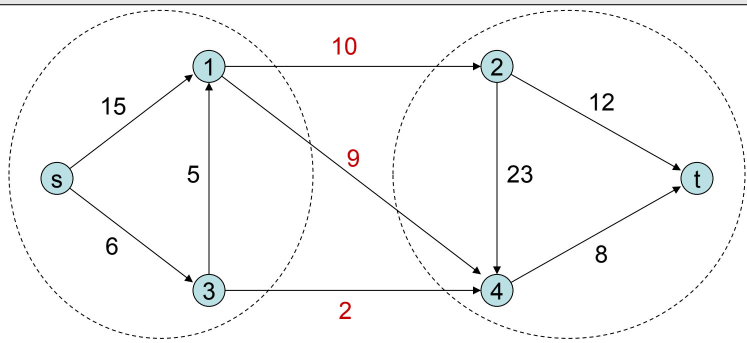


Capacity = 10 + 9 + 2 = 21

- **Easy observation:** The value of the maximum flow is at most the capacity of any cut.
- So: max s-t flow value ≤ min s-t cut capacity.

25

s-t Cuts

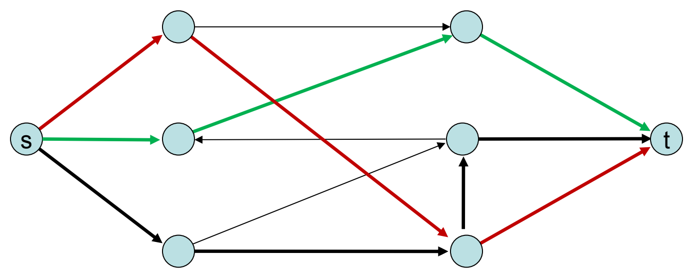


Capacity = 10 + 9 + 2 = 21

- **Easy observation:** The value of the maximum flow is at most the capacity of any cut.
- So: max s-t flow value ≤ min s-t cut capacity.
- **Famous result:** max s-t flow value **=** min s-t cut capacity.

26

Higher Levels of Connectivity

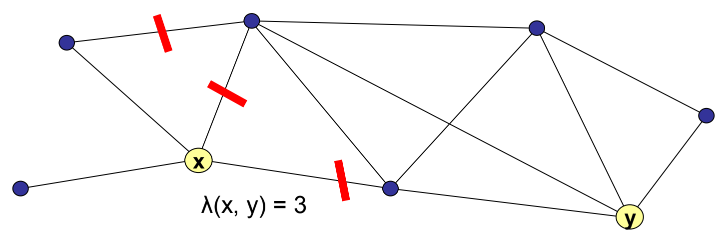


Maximum # of edge-disjoint s-t paths = Minimum # of edges we need to remove to separate s and t.

(path decomposition of a max flow in a graph with unit capacities)

(minimum s-t cut)

Higher Levels of Connectivity



- Let λ(x,y) denote the value of a unit-cap max flow from x to y.
- Equivalently, the minimum # of edge removals to separate x from y.
- This quantity is known as the <u>edge connectivity</u> between x and y.

28

The Hierarchical, Nested Structure of k-Edge-Connected Components