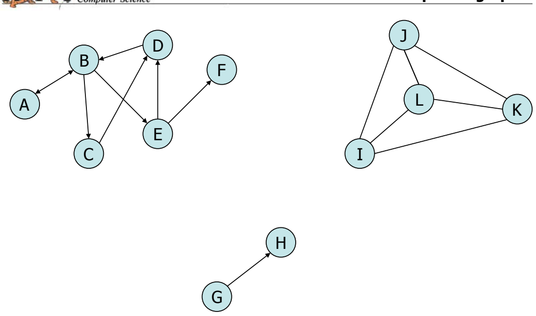
Department of Computer Science

Introduction to Graphs

What's a graph?

- A graph is a collection of two types of data:
  - vertices
    - also sometimes called "points"
    - can represent geographical locations, activities, etc.
  - edges
    - connections between vertices
    - may have a direction associated with them, in which case the graph is called a *directed graph* (*digraph*)
    - if they do not have directions associated with them, the graph is called an *undirected graph*

Examples of graphs



Graphs are usually drawn using points for vertices and lines for Edges, but a graph is defined independent of its representation

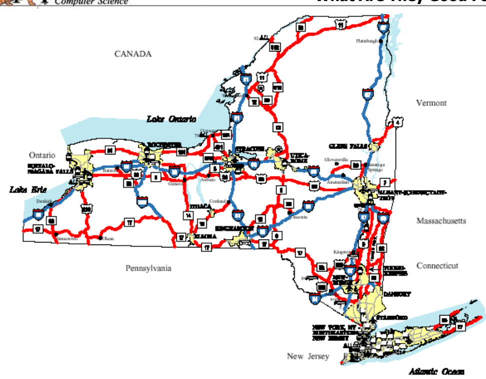
Department of Computer Science

Weighting edges

- In addition to being directed or undirected, edges can be weighted or un-weighted.
  - A weighted edge has a value associated with it
  - The weight often measures the cost of using the edge to go from one node to another
- A vertex may also have data associated with it.
  - This can be the name of a city, or a task to be completed, or whatever.

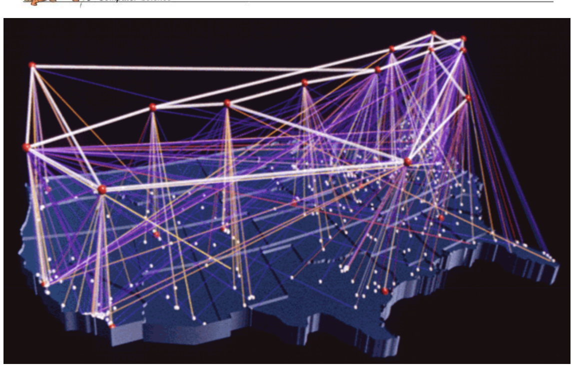
Department of Computer Science

What Are They Good For?



Department of Computer Science

What Are They Good For?



Other examples

- Graphical representation of a UNIX file system tree
- Graph representation of a computer network
- Graph representation of a software system
- Flow graph notation for various constructs
- Relationships between people ("who knows who?")
  - Good for the Kevin Bacon game, etc.

Terminology

- Two different vertices, *x* and *y*, in a graph are said to be *adjacent* if an edge connects *x* to *y*
- A *path* is a sequence of vertices in which each vertex is adjacent to the next one
  - The *length* of a path is the number of edges in the path
  - A *simple path* is a path in which no vertex is repeated
  - A *cycle* is a path of length greater than one that begins and ends at the same vertex
    - A graph with no cycles is called a *tree*
  - A *simple cycle* is a cycle consisting of three or more distinct vertices in which no vertex is visited more than once along the cycle's path

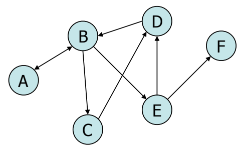
Terminology (ctd.)

- The *degree* of a vertex *x* is the number of edges *e* in which *x* is one of the endpoints of edge *e*
- The *neighbors* of a vertex *v*, are the vertices that are directly connected to *v*

Department of Computer Science

Applying the terms

- What is the length of a simple path from A to F?
- What cycles can you find?
- How many neighbors does B have?



Department of Computer Science

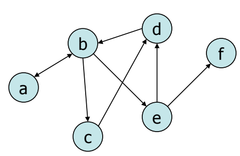
Formal definitions

- A graph $G = (V, E)$, consists of a set of vertices, V, along with a set of edges, E, where the edges in E are formed from distinct pairs of vertices in V.
- In an undirected graph, each edge $e = \{ v_1, v_2 \}$ is an unordered pair of distinct vertices, which connects the two vertices v_1 and v_2, without prescribing a direction from v_1 to v_2 or from v_2 to v_1.
- In a directed graph, each edge $e = \{ v_1, v_2 \}$ is an ordered pair of vertices, which connects the pair of vertices v_1 and v_2, in the direction from v_1 to v_2. In this case we say v_1 is the origin of the edge $e = \{ v_1, v_2 \}$ and v_2 is the end of the edge e.

Department of Computer Science

Being formal

- Write a formal definition for this directed graph.
  - G = { V, E }, where
    - V = { a, b, c, d, e, f }
    - E = { {a, b}, {b, a}, {b, c}, {c, d}, {d, b}, {b, e}, {e, d}, {e, f} }
- Write another definition, assuming that it's an undirected graph.
  - G = { V, E }, where
    - V = { a, b, c, d, e, f }
    - E = { {a, b}, {b, c}, {b, e}, {b, d}, {c, d}, {e, d}, {e, f} }



Connectivity

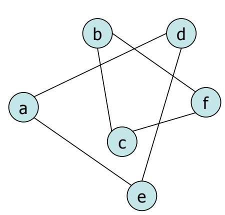
- Two vertices in a graph *G* are said to be *connected* if there is a path from the first to the second in *G*
  - If $x \in V$ and $y \in V$, where $x \neq y$, then *x* and *y* are *connected* if there exists a path, $p = v_1, v_2, \ldots, v_n$, in *G*, such that $x = v_1$ and $y = v_n$
- In the graph *G*, a *connected component* is a subset, *S*, of the vertices *V* that are all connected to one another
  - *S* is a *connected component* of *G* if, for any two distinct vertices, $x \in S$ and $y \in S$, *x* is connected to *y*
- A graph is *connected* if there is a path from every node to every other node in the graph
  - A graph that is not connected is made up of connected components

Connectivity (ctd.)

- A digraph is said to be strongly connected if for every pair of nodes, n_i and n_j, there exists a path from node n_i to node n_j.
- An undirected graph is said to be connected if for every pair of nodes, n_i and n_j, there is a path connecting the two nodes

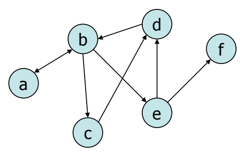
What's connected? (1)

- Given the graph to the right, what vertices are connected?
- Answer:
  - The vertices {a, d, e} and {b, c, f} form connected components in the graph



What's connected? (2)

- Given the graph to the right, what vertices are connected?
- Answer:
  - The vertices {a, b, c, d, e} form a connected component in the graph
  - The vertex f isn't part of the connected component, because you can't get <u>from</u> it to the other vertices



Edges

- If we denote the number of vertices in a graph by *V* and the number of edges by *E*, note that *E* can range anywhere from 0 to $\frac{1}{2}V(V-1)$
  - Graphs with all edges present are called *complete* graphs
  - Graphs with relatively few edges (say less than $V \log V$) are called *sparse* graphs
  - Graphs with relatively few of the possible edges missing are called *dense*

DiGraph.java (old version)

```
import java.util.*;

public interface DiGraph {

    // Methods to build the graph
    public void addVertex( Object key, Object data );
    public void addEdge( Object fromKey, Object toKey, Object data )
                throws NoSuchVertexException;

    // Operations on edges
    public boolean isEdge( Object fromKey, Object toKey ) throws NoSuchVertexException;
    public Object getEdgeData( Object fromKey, Object toKey )
                throws NoSuchVertexException;

    // Operations on vertices
    public boolean isVertex( Object key );
    public Object getVertexData( Object key ) throws NoSuchVertexException;
    public int numVertices();
    public int inDegree( Object key );
    public int outDegree( Object key );
    public Collection neighborData( Object key ) throws NoSuchVertexException;
    public Collection neighborKeys( Object key ) throws NoSuchVertexException;

    // Utility methods
    public Collection vertexData();
    public Collection vertexKeys();
    public Collection edgeData();
    public void clear();

} // DiGraph
```

Department of Computer Science

DiGraph.java (new version)

```java
import java.util.*;

public interface DiGraph<VertexKey, VertexData, EdgeData> {

    // Methods to build the graph
    public void addVertex( VertexKey key, VertexData data );
    public void addEdge( VertexKey fromKey, VertexKey toKey, EdgeData data )
                throws NoSuchVertexException;

    // Operations on edges
    public boolean isEdge( VertexKey fromKey, VertexKey toKey )
                throws NoSuchVertexException;
    public EdgeData getEdgeData(VertexKey fromKey, VertexKey toKey )
                throws NoSuchVertexException;

    // Operations on vertices
    public boolean isVertex( VertexKey key );
    public VertexData getVertexData( VertexKey key ) throws NoSuchVertexException;
    public int numVertices();
    public int inDegree( VertexKey key );
    public int outDegree(VertexKey key );
    public Collection<VertexData> neighborData( VertexKey key )
                throws NoSuchVertexException;
    public Collection<VertexKey> neighborKeys(VertexKey key )
                throws NoSuchVertexException;

    // Utility methods
    public Collection<VertexData> vertexData();
    public Collection<VertexKey> vertexKeys();
    public Collection<EdgeData> edgeData();
    public void clear();

} // DiGraph
```

Department of Computer Science

Graphs for problem solving

- A problem can often be represented as a graph
- The solution to the problem is then obtained by solving a problem on the corresponding graph

Department of Computer Science

The coin game

- 3 coins are placed on a table in a row
- The goal is to get the coin row into a head-tail-head configuration in the shortest number of coin flips possible
- Rules:
  - You may flip the middle coin whenever you want to
  - You may flip one of the end coins only if the other two coins are the same as each other (both heads or both tails)
  - You are not allowed to change coins in any other way (such as shuffling them around)

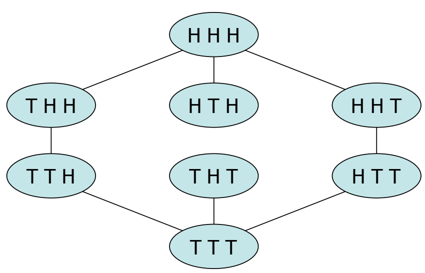
Department of Computer Science

How does that fit in?

- We can used a graph to represent the possible states of the 3 coins
  - Vertices represent a given set of states for the coins
  - Edges represent a transformation from one state to another (i.e., flipping one of the coins)
- By finding a path between two states, we can find a way of getting from one combination to another
  - The shorter the path, the fewer the number of coin flips required

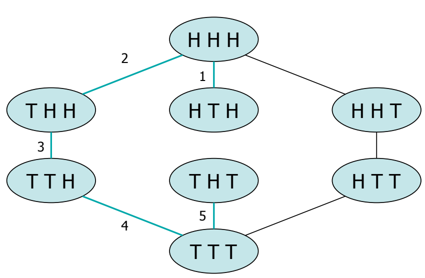
Department of Computer Science

A graph for the coin game



Department of Computer Science

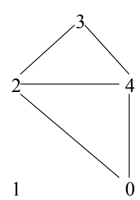
One possible path from "H T H"



Ways of representing graphs

- There are two common ways to represent a graph:
  - With an adjacency matrix
  - With an adjacency list

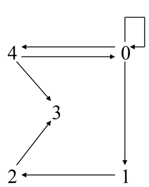
Adjacency Matrix



	0	1	2	3	4
0	F	F	T	F	T
1	F	F	F	F	F
2	T	F	F	T	T
3	F	F	T	F	T
4	T	F	T	T	F

Note: the adjacency matrix representation is often satisfactory only if the graphs to be represented are dense (most of the array will be true)

Adjacency Matrix



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

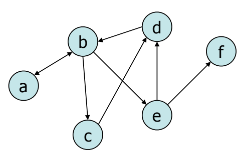
Note: sometimes folks will simply leave spots in the matrix where there is no edge empty, to improve readability....

Adjacency matrices implementation

- Usually implemented with two-dimensional arrays
  - The first (source) number is often used to denote the row, while the second (destination) denotes the column
  - Although this is just a handy convention, it is what I will assume you're using on tests, etc. unless you clearly state otherwise

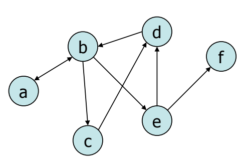
Building adjacency matrices

- What is the adjacency matrix for this graph?



Building adjacency matrices

	a	b	c	d	e	f
a	F	**T**	F	F	F	F
b	**T**	F	**T**	F	**T**	F
c	F	F	F	**T**	F	F
d	F	**T**	F	F	F	F
e	F	F	F	**T**	F	**T**
f	F	F	F	F	F	F



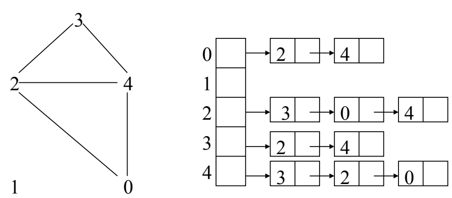
Department of Computer Science

Good and bad about matrices

- Considerations:
  - Adding or removing edges
  - Checking whether or not a particular edge is present.
  - Iterating a loop that executes one time for each edge with a particular source vertex.
- What are the good things about using an adjacency matrix?
- The bad things?

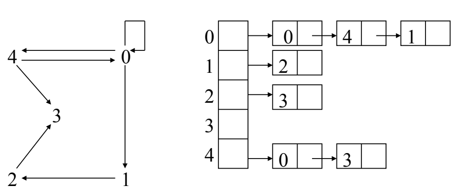
Department of Computer Science

Adjacency List



Department of Computer Science

Adjacency List



Good and bad about lists

- What are the positives to using an adjacency list?
- What are some of the negatives of using an adjacency list?

Searching a graph

- Several questions arise when processing a graph
  - Is the graph connected?
  - If not, what are the connected components?
  - Does the graph have a cycle?
  - …

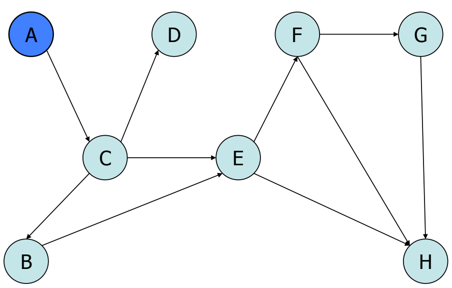
Searching various ADTs

- Trees
  - Breadth-first
  - Depth-first
    - in-order
    - pre-order
    - post-order
- Graphs
  - Breadth-first
  - Depth-first
- What do "breadth-first" and "depth-first" mean for graphs?

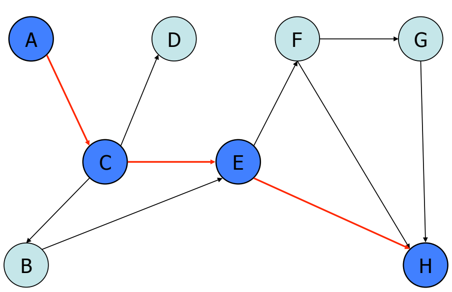
Depth-First Search

- In a depth-first search of a graph:
  - we start at a node
  - we follow whatever path we like as far as we can from that node
  - when we can go no further, we back up and find paths to other (unvisited) nodes

DFS Example

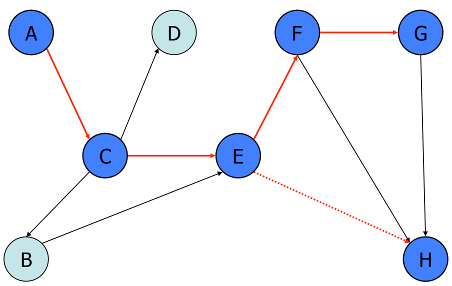


DFS Example



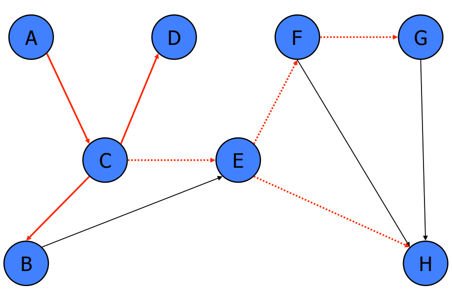
Department of Computer Science

DFS Example



Department of Computer Science

DFS Example



Department of Computer Science

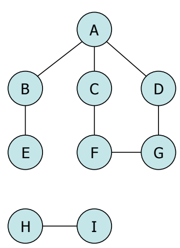
DFS Pseudo-code

- Associate with each vertex an `Integer` value
  - A value of zero indicates that the vertex has not been visited
  - A non-zero value indicates that the node has been visited
- The *pseudo-code*
  - Build the graph and initialize the values associated with each vertex to zero
  - Get a collection that contains the keys of all the vertices in the graph
  - Set an integer variable, named component, to 1
  - Iterate over the keys
    - Get the data associated with the vertex identified by the current key
    - If the *visit* value is 0 → visit(vertex, component)

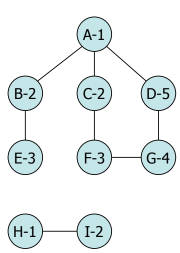
DFS Visit

- Visit(vertex *v*, Integer *component*)
  - Change the "visit value" associated with *v* to *component*
  - Get a collection that contains the keys of the neighbors of *v*
  - Iterate over the collection
    - Get the data associated with the current key
    - If the vertex has not been visited → visit(*vertex*, *component + 1*)

DFS: What values go where?



DFS: Solution (assuming alphabetical ordering)



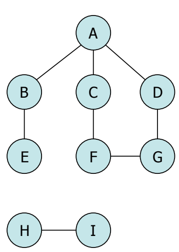
DFS Visit – Rewritten as non-recursive

- Visit(Stack *s*, vertex *v*)
  - Set *v*'s "visit value" to 1
  - Push *v* onto the stack *s*
  - While *s* is not empty
    - Pop the stack and make the vertex the current vertex
    - Get a collection that contains the keys of the neighbors of the current vertex
    - Iterate over the collection
      - If a neighbor's "visit value" is equal to 0 (i.e., it hasn't been visited)
        - » Change its "visit value" to *v.key* + 1
        - » Push the neighboring vertex onto the stack

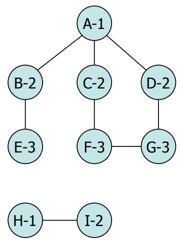
Breadth-First Search (BFS)

- What if we changed the stack in the non-recursive DFS-visit to a queue?
- Visit(Queue *q*, vertex *v*)
  - Set *v*'s "visit value" to 1
  - Enqueue *v* into the queue *q*
  - While the *q* is not empty
    - Dequeue and make the vertex the current vertex
    - Get a collection that contains the keys of the neighbors of the current vertex
    - Iterate over the collection
      - If a neighbor's "visit value" is equal to 0 (i.e., it hasn't been visited)
        - » Change its "visit value" to *v.visitValue*+ 1
        - » Enqueue the current vertex

BFS: What values go where?



BFS: Solution (assuming alphabetical ordering)



Common graph algorithms

- Some common problems/algorithms involving graphs are:
  - Graph coloring
  - Shortest path identification
    - Dijkstra's Algorithm
  - Identifying minimum spanning trees
    - Kruskal's Algorithm

So that's the basics of graphs....

- Any questions?

Department of Computer Science

Graph coloring

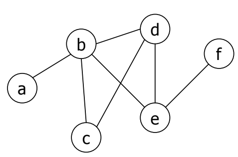
One common problem....

Graph coloring

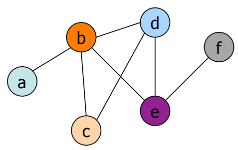
- A very common and challenging problem in graph theory is that of "coloring" a graph.
  - In this context "coloring" means assigning a color (or a number) to each node such that none of its neighbors has that same color (number).
- For a given graph, it's possible that lots of colorings may exist
  - We typically want the one that takes the least # of colors

Coloring a simple graph

- Consider this graph, for instance

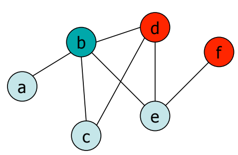


- One possibility would just be to assign a different color to *every* vertex



Number of colors: 6

- A better (optimal) solution might be to re-use colors from some nodes:



Number of colors: 3

- Graph coloring can be used
  - to coloring on a map of nations (or other entities), providing distinctive appearances for each
  - to solve problems involving scheduling and assignments, or other resource allocation issues

Department of Computer Science

Sample problem #1

- Suppose you want to schedule final exams for 7 courses, and you want to avoid having a student do more than one exam a day.
  - We shall call the courses 1,2,3,4,5,6,7.
  - In the table below a star in entry ij means that course i and j have at least one student in common so you can't have them on the same day.
  - What is the least number of days you need to schedule all the exams? Show how you would schedule the exams.

Department of Computer Science

Problem #1 data

	1	2	3	4	5	6	7
1		*	*	*		*	*
2	*		*				*
3	*	*		*			
4	*		*		*	*	
5				*		*	
6	*			*	*		*
7	*	*				*	

Department of Computer Science

Sample problem #2

- Suppose you run a day care for an office building, and you need to assign a locker where each child's parent can put the child's food.
  - There are seven children A,B,C,D,E,F,G.
  - The children come and leave so they are not all there at the same time.
  - You have 1 hour time slots starting 7:00 a.m. to 12:00 noon.
  - A star in the table means a child is present at that time.
  - What is the minimum number of lockers necessary? Show how you would assign the lockers.

Problem #2 data

	A	B	C	D	E	F	G
7:00	*			*	*		
8:00	*	*	*				
9:00	*		*	*		*	
10:00	*		*			*	*
11:00	*					*	*
12:00	*				*		

Complexity

- Graph coloring is hard to do in an efficient fashion
  - Graph coloring is considered to be an "NP-complete" problem
    - Simply *proving* that an answer to a graph coloring problem is correct is $O(n^2)$
    - *Finding* the answer is even tougher (probably a lot tougher)
  - "Greedy" algorithms exist, which try to optimize for performance by providing an approximation ("best guess") of the best coloring, but even these are $O(n^2)$

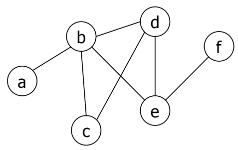
A greedy coloring algorithm

- Local variables: "numColors", "curColor"
- Set the color for all vertices in the graph to 0 ("undefined")
- Set numColors to 0
- For each vertex in the graph:
  - Set curColor to 1
  - While curColor <= numColors and one of the neighbors of the current vertex is colored with "curColor", increment curColor
  - If curColor > numColors, then numColors = curColor
  - Set the color for the current vertex to curColor

Department of Computer Science

Graph coloring example

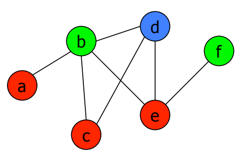
Try coloring the following graph, working in alphabetic order.
Use color ordering: red, green, blue, yellow, purple, gray.



Department of Computer Science

Graph coloring example

Try coloring the following graph, working in alphabetic order.
Use color ordering: red, green, blue, yellow, purple, gray.



R·I·T
Department of Computer Science

Shortest paths in graphs

Another common problem....

Dijkstra's Shortest Path Algorithm

- Each node is labeled with its distance from the source node along the best path
- Initially no paths are known, so the values are infinity
- The algorithm starts at the source node and explores possible paths, one hop at a time
- When a label is marked permanent, its label will not change

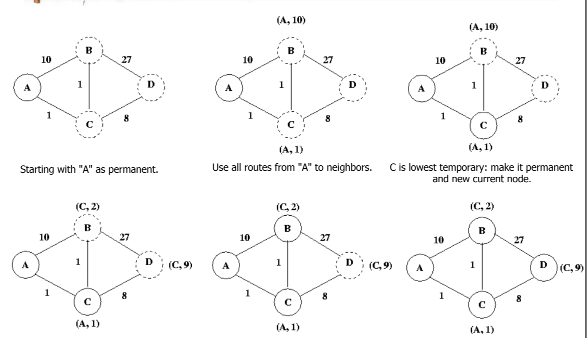
The Algorithm

- Make the source node permanent; the source node is the first working node
- Examine each non-permanent node adjacent to the working node
  - if it is not labeled, label with the distance from the source and the name of the working node
  - if it is labeled, see if the cost computed using the working node is better than the cost in the label; if so change the label to reflect the better path

The Algorithm (ctd)

- Find the non-permanent node with the smallest label, and make it permanent
  - If all the nodes are marked permanent, the algorithm terminates
  - Otherwise, the node just made permanent becomes the working node
- When the algorithm is complete, the path is found (in reverse) by reading the labels from the destination node back to the source

Example: shortest paths from "A" to other nodes



Starting with "A" as permanent.

Use all routes from "A" to neighbors.

C is lowest temporary: make it permanent and new current node.

Update C's non-permanent neighbors.

B is lowest temporary: make it permanent and new current node.

Final state (no changes from "B").

Department of Computer Science

Any questions?

Department of Computer Science

- Some additional algorithms of note:
  - Bellman-Ford Algorithm
    - computes single-source shortest paths in a weighted digraph (where some of the edge weights may be negative, unlike in Dijkstra's algorithm)
  - Floyd-Warshall Algorithm
    - a graph analysis algorithm for finding shortest paths in a weighted, directed graph
  - Prim's Algorithm
    - finds a minimum spanning tree for a connected weighted graph
  - Ford-Fulkerson algorithm
    - for computing the *Maximum Flow* within a Network Graph
  - Edmonds-Karp algorithm
    - an alternative approach for computing maximum flow in a network graph