10-GRAPHS

Topics

- Introduction to Graphs
- Representing a Graph
- Searching a Graph
- Topological Sorting with Directed Graphs
- Connectivity in Directed Graphs
- Shortest Path in Weighted Directed Graphs

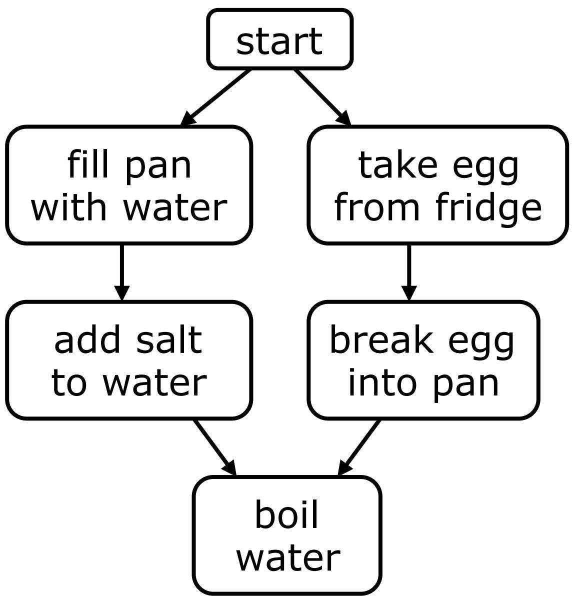
Introduction to Graphs

Graph applications

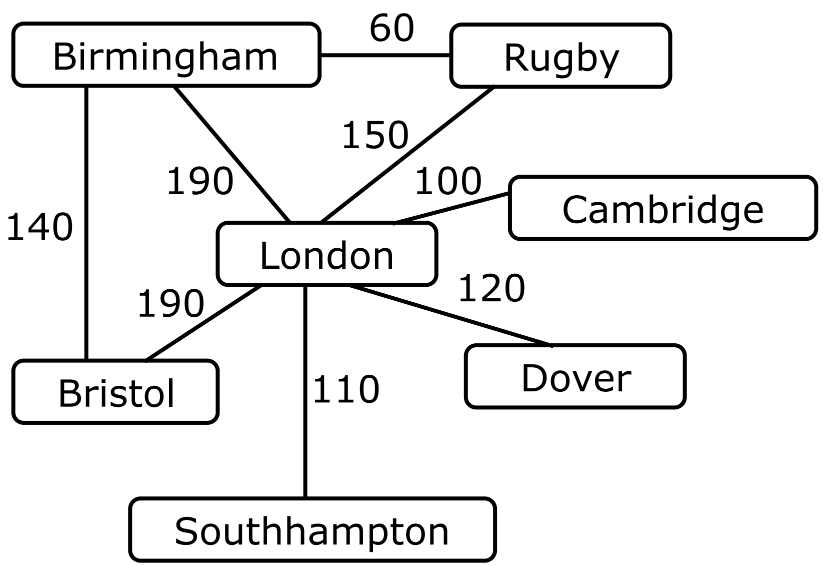
- Graphs can be used for:
  - Finding a route to drive from one city to another
  - Finding connecting flights from one city to another
  - Determining least-cost highway connections
  - Designing optimal connections on a computer chip
  - Implementing automata
  - Implementing compilers
  - Doing garbage collection
  - Representing family histories
  - Pert charts
  - Playing games

Graph definitions

- There are two kinds of graphs: directed graphs (sometimes called digraphs) and undirected graphs



A directed graph



An undirected graph

Graph terminology I

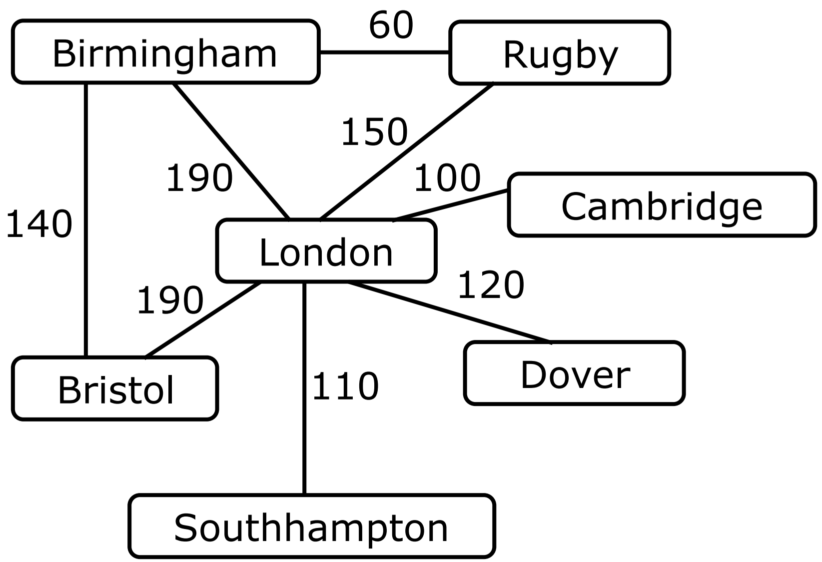
- A graph is a collection of nodes (or vertices, singular is vertex) and edges (or arcs)
  - Each node contains an element
  - Each edge connects two nodes together (or possibly the same node to itself) and may contain an edge attribute
- A directed graph is one in which the edges have a direction
- An undirected graph is one in which the edges do not have a direction
  - Note: Whether a graph is directed or undirected is a *logical* distinction—it describes how we think about the graph
  - Depending on the *implementation,* we may or may not be able to follow a directed edge in the "backwards" direction

Graph terminology II

- The size of a graph is the number of *nodes* in it
- The empty graph has size zero (no nodes)
- If two nodes are connected by an edge, they are neighbors (and the nodes are adjacent to each other)
- The degree of a node is the number of edges it has
- For directed graphs,
  - If a directed edge goes from node S to node D, we call S the source and D the destination of the edge
    - The edge is an out-edge of S and an in-edge of D
    - S is a predecessor of D, and D is a successor of S
  - The in-degree of a node is the number of in-edges it has
  - The out-degree of a node is the number of out-edges it has

Graph terminology III

- A path is a list of edges such that each node (but the last) is the predecessor of the next node in the list
  - We may have more than one path between two nodes.
- A cycle is a path whose first and last nodes are the same

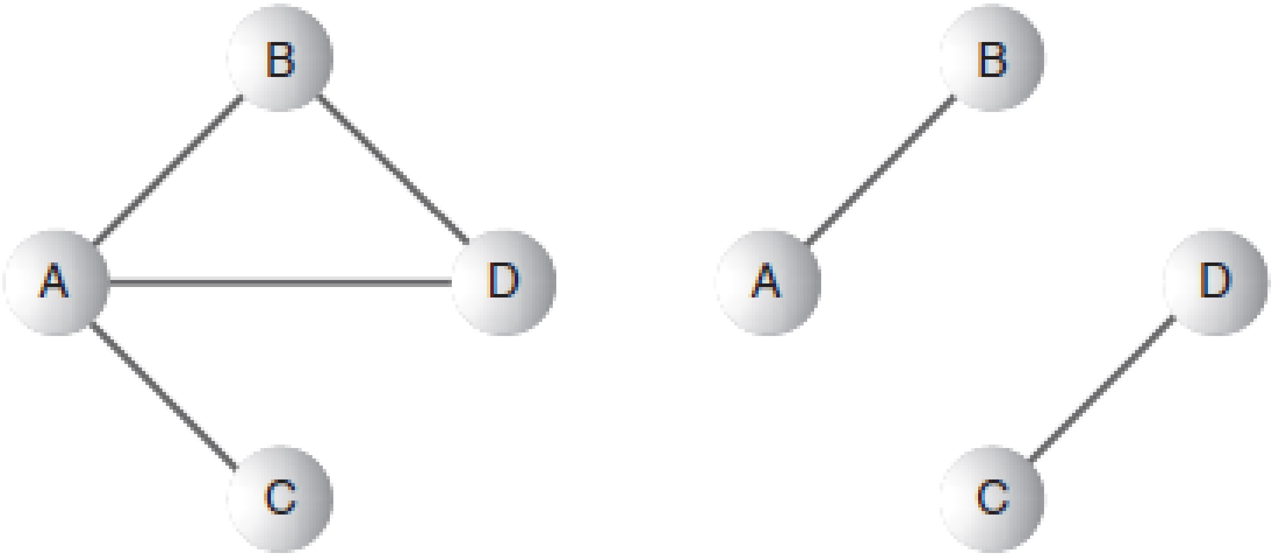


- Example: (London, Bristol, Birmingham, London, Dover) is a path
- Example: (London, Bristol, Birmingham, London) is a cycle
- A cyclic graph contains at least one cycle
- An acyclic graph does not contain any cycles

Graph terminology IV

- An undirected graph is connected if there is a path from every node to every other node
- A directed graph is strongly connected if there is a path from every node to every other node
- A directed graph is weakly connected if the underlying undirected graph is connected
- Node X is reachable from node Y if there is a path from Y to X
- A subset of the nodes of the graph is a connected component (or just a component) if there is a path from every node in the subset to every other node in the subset

Graph terminology V



a) Connected Graph b) Non-connected Graph

Connected and non-connected graphs.

Representing a Graph in a Program

Representing a Graph in a Program

- Vertices:
  - A vertex represents some real-world object, such as a city in an airline route simulation.

```
class Vertex
{
    public char label; // label (e.g. 'A')
    public boolean wasVisited;
    public Vertex(char lab) // constructor
    {
        label = lab;   wasVisited = false;
    }
} //
```

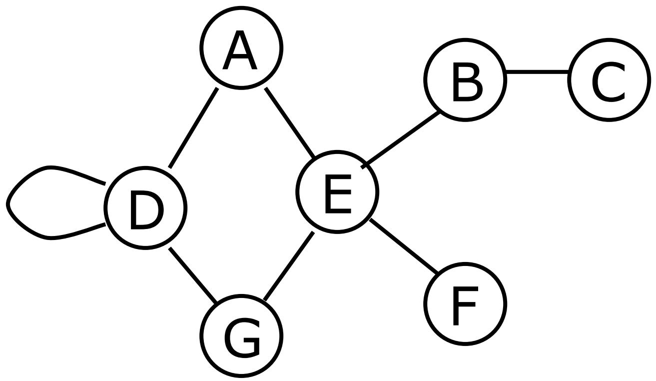
- Vertex objects can be placed in an array called vertexList.

Representing a Graph in a Program

- Edges:
  - In a binary tree, each node has a maximum of two children, but in a graph each vertex may be connected to an arbitrary number of other vertices.
  - To model this sort of free-form organization, three methods are commonly used for graphs:

1. the *adjacency matrix*
2. the *edge set*
3. the *adjacency set*

Adjacency-matrix representation I



	A	B	C	D	E	F	G
A				●	●		
B			●		●		
C		●					
D	●			●			●
E	●	●				●	●
F					●		
G				●	●		

- One simple way of representing a graph is the adjacency matrix
- A 2-D array has a mark at [i][j] if there is an edge between node i and node j
- The adjacency matrix is symmetric about the main diagonal

- This representation is only suitable for *small* graphs! (Why?)

Adding Vertices and Edges to a Graph

- Creation of a vertex:

```
vertexList[nVerts++] = new Vertex('F');
```

- Add an edge to a graph using an adjacency matrix and want to add an edge between vertices 1 and 3.

```
adjMat[1][3] = 1;
adjMat[3][1] = 1;
```

The Graph Class - 1

```
class Graph
{
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // array of vertices
   private int adjMat[][]; // adjacency matrix
   private int nVerts; // current number of vertices
// -------------------------------------------------------------
```

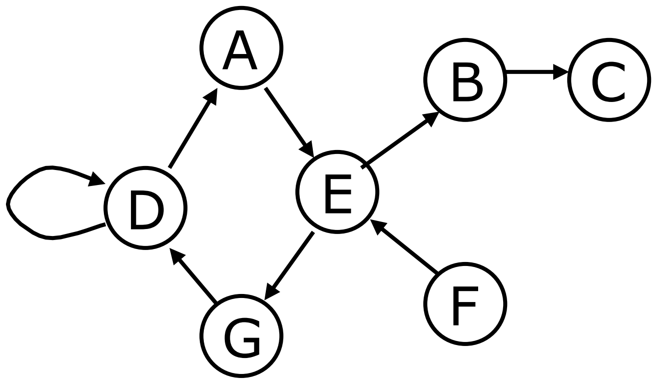
The Graph Class - 2

```
public Graph() // constructor
{
   vertexList = new Vertex[MAX_VERTS];
   // adjacency matrix
   adjMat = new int[MAX_VERTS][MAX_VERTS];
   nVerts = 0;
   for(int j=0; j<MAX_VERTS; j++) // set adjacency
      for(int k=0; k<MAX_VERTS; k++) // matrix to 0
         adjMat[j][k] = 0;
} // end constructor
// -------------------------------------------------------------
```

The Graph Class - 3

```
public void addVertex(char lab) // argument is label
{ vertexList[nVerts++] = new Vertex(lab); }
// ---------------------------------------------------------
public void addEdge(int start, int end)
{  adjMat[start][end] = 1;    adjMat[end][start] = 1;   }
// ---------------------------------------------------------
public void displayVertex(int v)
{  System.out.print(vertexList[v].label); }
// ---------------------------------------------------------
} // end class Graph
```

Adjacency-matrix representation II



Destination

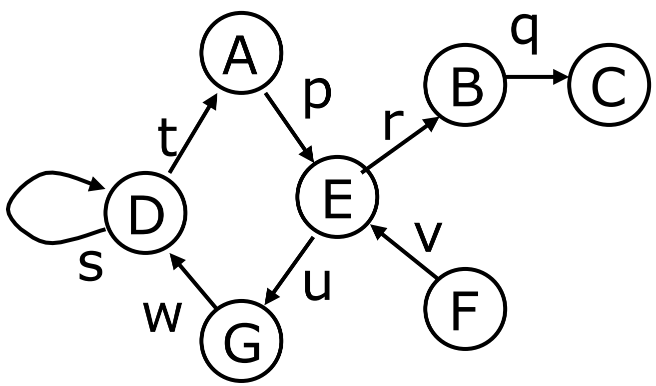
Source	A	B	C	D	E	F	G
A					●		
B			●				
C							
D	●			●			
E		●					●
F					●		
G				●			

- An adjacency matrix can equally well be used for digraphs (directed graphs)
- A 2-D array has a mark at [i][j] if there is an edge from node i to node j
- Again, this is only suitable for *small* graphs!

Edge-set representation I

- An edge-set representation uses a *set* of nodes and a *set* of edges
  - The sets might be represented by, say, linked lists
  - The set links are stored in the nodes and edges themselves
- The only other information in a node is its element (that is, its value)—it does not hold information about its edges
- The only other information in an edge is its source and destination (and attribute, if any)
  - If the graph is undirected, we keep links to both nodes, but don't distinguish between source and destination
- This representation makes it easy to find nodes from an edge, but you must search to find an edge from a node
- This is seldom a good representation

Edge-set representation II



- Here we have a set of nodes, and each node contains only its element (not shown)

nodeSet = {A, B, C, D, E, F, G}

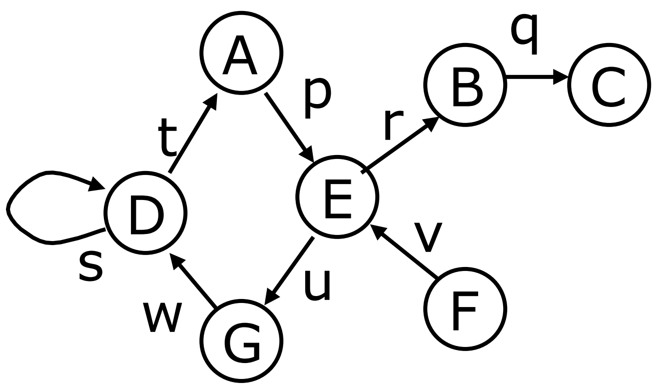
edgeSet = { p: (A, E),
q: (B, C), r: (E, B),
s: (D, D), t: (D, A),
u: (E, G), v: (F, E),
w: (G, D) }

- Each edge contains references to its source and its destination (and its attribute, if any)

Adjacency-set representation I

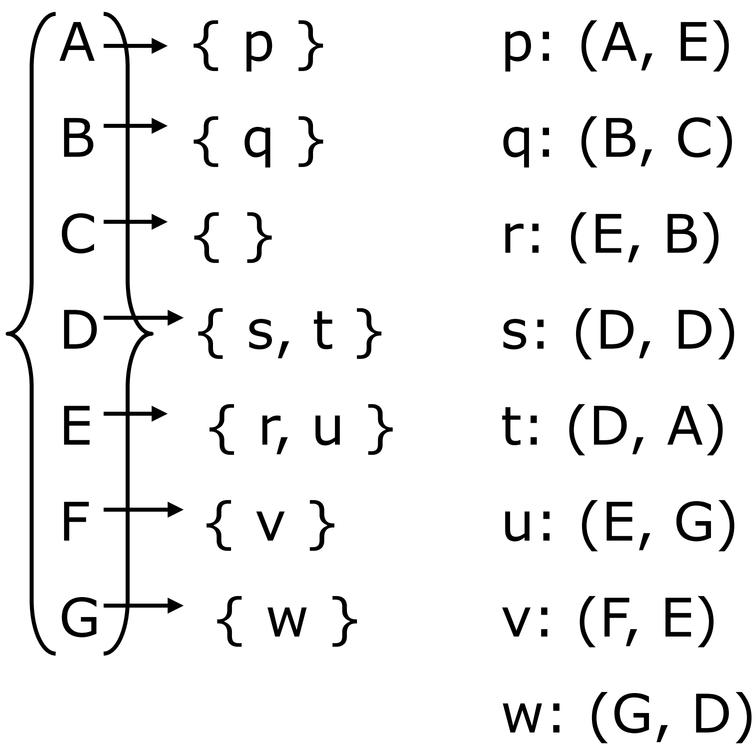
- An adjacency-set representation uses a *set* of nodes
  - Each node contains a reference to the set of *its* edges
  - For a directed graph, a node might only know about (have references to) its out-edges
- Thus, there is not one single edge set, but rather a separate edge set for each node
  - Each edge would contain its attribute (if any) and its destination (and possibly its source)

Adjacency-set representation II



- Here we have a set of nodes, and each node refers to a set of edges

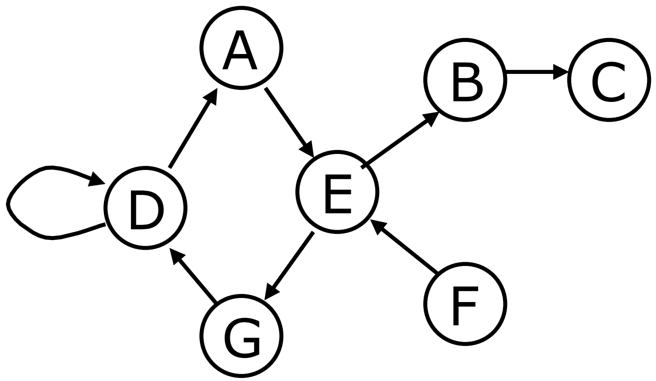
- Each edge contains references to its source and its destination (and its attribute, if any)



Adjacency-set representation III

- If the edges have no associated attribute, there is no need for a separate Edge class
  - Instead, each node can refer to a set of its *neighbors*
  - In this representation, the edges would be implicit in the connections between nodes, not a separate data structure
- For an undirected graph, the node would have references to all the nodes adjacent to it
- For a directed graph, the node might have:
  - references to all the nodes adjacent to it, or
  - references to only those adjacent nodes connected by an out-edge from this node

Adjacency-set representation IV



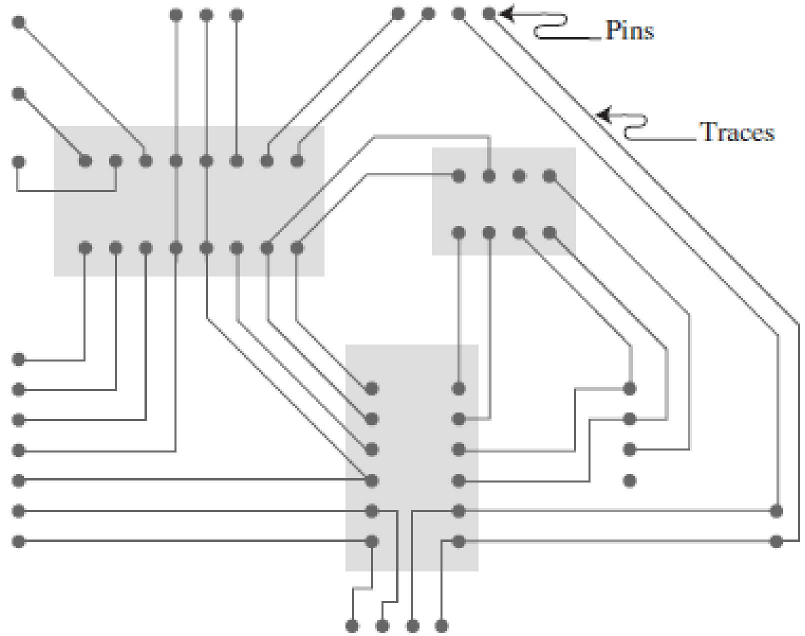
$$\left\{\begin{array}{l} A \to \{ E \} \\ B \to \{ C \} \\ C \to \{ \} \\ D \to \{ D, A \} \\ E \to \{ B, G \} \\ F \to \{ E \} \\ G \to \{ D \} \end{array}\right\}$$

- Here we have a set of nodes, and each node refers to a set of other (pointed to) nodes
- The edges are *implicit*

Searching a Graph

Searching a graph

- Algorithms to find which vertices can be reached from a specified vertex.
- For example:
  - Find which cities are reachable from a specified city X
  - Find, on a circuit board, which pins are connected to the same electrical circuit.



Pins and traces on a circuit board.

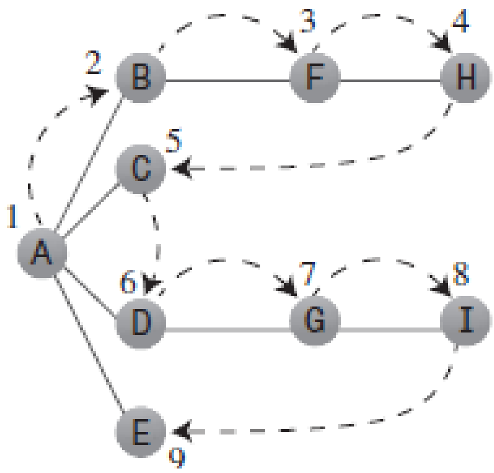
Searching a graph

- There are two common approaches to searching a graph:
  - *depth-first search (DFS)*
  - *breadth-first search (BFS).*
- Both will eventually reach all connected vertices.
- The depth-first search is implemented with a stack, whereas the breadth-first search is implemented with a queue.
- These mechanisms result, as we'll see, in the graph being searched in different ways.
- (See Workshop Applet GraphN)

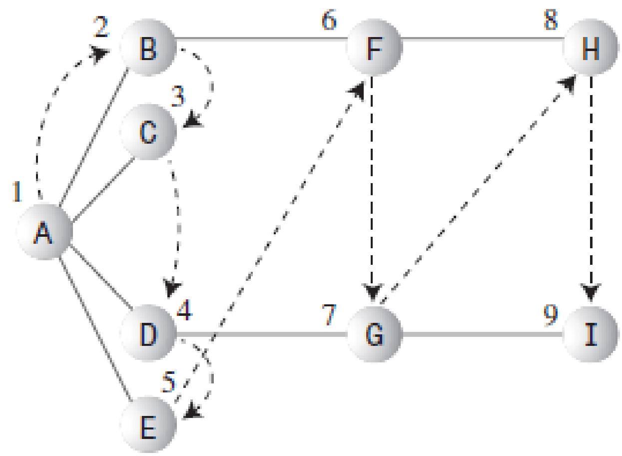
Depth-first search vs. Breadth-first search

- In depth-first search, the algorithm acts as though it wants to get as far away from the starting point as quickly as possible.
- In breadth-first search, on the other hand, the algorithm likes to stay as close as possible to the starting point.

Order of visiting
the nodes in DFS
ABFHCDGIE



Order of visiting
the nodes in BFS
ABCDEFGHI



Searching a graph

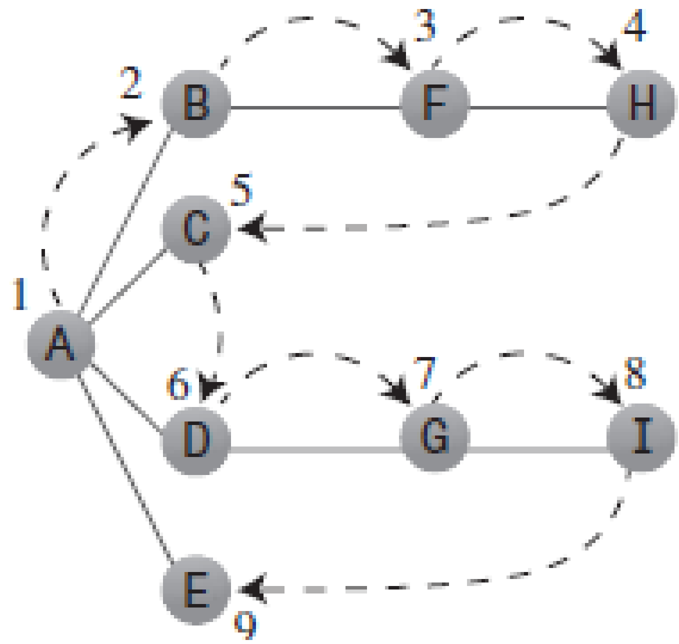
- A graph may have cycles
  - We don't want to search around and around in a cycle
- To avoid getting caught in a cycle, we must keep track of which vertices (nodes) we have already explored
- There are two basic techniques for this:
  - Keep a set of already explored vertices, or
  - Mark the vertex itself as having been visited

Example: Depth-first search

- To carry out the depth-first search, you pick a starting point —in this case, vertex A.
- You then do three things:

1. visit this vertex,
2. push it onto a stack so you can remember it,
3. and mark it so you won't visit it again.

- Next, you go to any vertex adjacent to A that hasn't yet been visited, and repeat 3 steps.



Order of visiting the nodes in DFS ABFHCDGIE

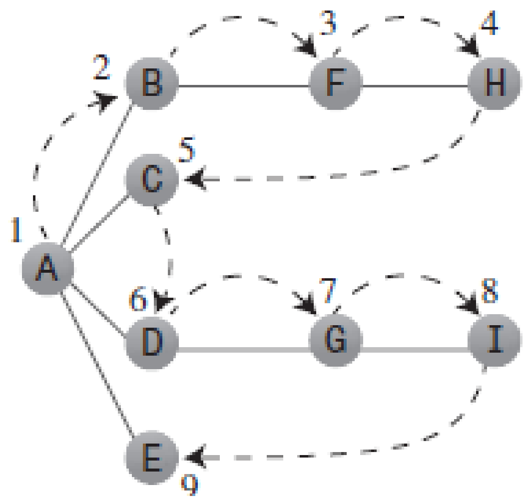
Example: Depth-first search

- Rules of DFS:
- Pick a starting vertex, say "A"; visit it, process it and push it on stack, then follow these rules:
  - Rule 1: If possible, visit an adjacent unvisited vertex, mark it, and push it on the stack.
  - Rule 2: If you can't follow Rule 1, then, if possible, pop a vertex off the stack.
  - Rule 3: If you can't follow Rule 1 or Rule 2, you're done.
- **Here is how to do DFS on a graph:**

```
Push the starting vertex to the top of the stack, process it, and mark it as
visited;
while (stack is not empty) {
   if (there is an unvisited adjacent vertex "V" …
                              … to the vertex "T" at the top of stack)
      push vertex "V" on stack, process it, and mark it as visited;
   else
      pop vertex "T" from stack;
}
```

TABLE 13.3 Stack Contents During Depth-First Search

Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC
Pop C	A
Visit D	AD
Visit G	ADG
Visit I	ADGI
Pop I	ADG
Pop G	AD
Pop D	A
Visit E	AE
Pop E	A
Pop A	
Done	




```
Push the starting vertex to the top of
the stack, process it, and mark it as
visited;
while (stack is not empty) {
    if (there is an unvisited adjacent
    vertex "V" to the vertex "T" at
    the top of stack)
       push vertex "V" on stack,
       process it,
       and mark it as visited;
    else
       pop vertex "T" from stack;
}
```

Java Code – Depth First Search - 1

```
public void dfs() // depth-first search
{ // begin at vertex 0
   vertexList[0].wasVisited = true; // mark it
   displayVertex(0); // display it
   theStack.push(0); // push it
```

Java Code – Depth First Search - 2

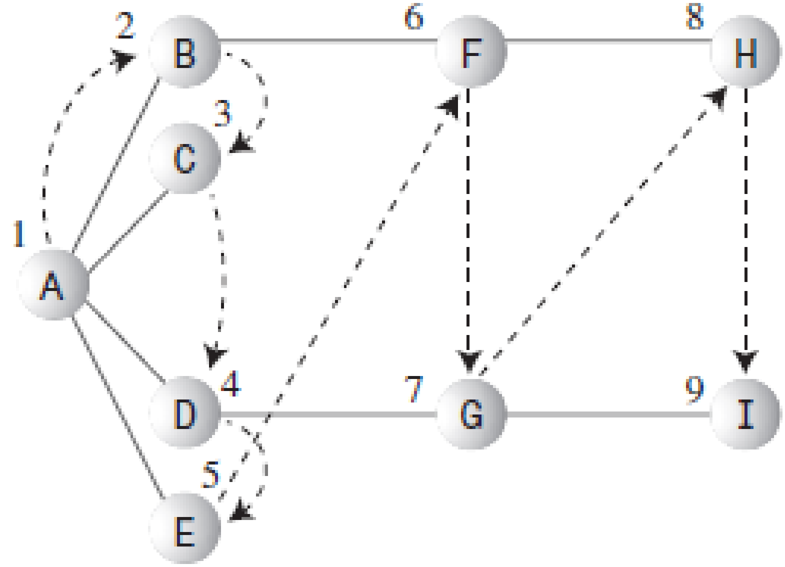
```
while( !theStack.isEmpty() ) // until stack empty,
{
  // get an unvisited vertex adjacent to stack top
  int v = getAdjUnvisitedVertex( theStack.peek() );
  if(v == -1) // if no such vertex,
    theStack.pop(); // pop a new one
  else // if it exists,
  {
    vertexList[v].wasVisited = true; // mark it
    displayVertex(v); // display it
    theStack.push(v); // push it
  }
} // end while
// stack is empty, so we're done
} // end dfs
```

Java Code – Get Adjacent Unvisited Vertex

```
// returns an unvisited vertex adjacent to v
public int getAdjUnvisitedVertex(int v)
{
   for(int j=0; j<nVerts; j++)
   if(adjMat[v][j]==1 &&
                    vertexList[j].wasVisited==false)
      return j; // return first such vertex
   return -1; // no such vertices
} // end getAdjUnvisitedVertex()
```

Example: Breadth-first search

- It visits all the vertices adjacent to the starting vertex, and only then goes further afield.
  - This kind of search is implemented using a queue instead of a stack.



Order of visiting
the nodes in BFS
ABCDEFGHI

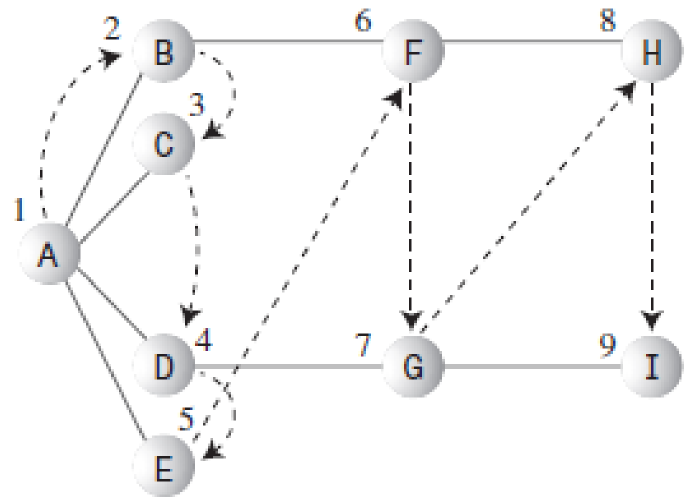
Example: Breadth-first search

- Rules of BFS:
- "A" is the starting vertex, so you visit it and make it the current vertex. Then you follow these rules:
  - Rule 1: Visit the next unvisited vertex (if there is one) that's adjacent to the current vertex, mark it, and insert it into the queue.
  - Rule 2: If you can't carry out Rule 1 because there are no more unvisited vertices, remove a vertex from the queue (if possible) and make it the current vertex.
  - Rule 3: If you can't carry out Rule 2 because the queue is empty, you're done.
- **Here is how to do BFS on a graph:**

```
Insert starting vertex in the queue, process it, and mark it as visited;
while (queue is not empty) {
    Remove the element at the head of the queue …
                              … and make it the current vertex;
    while (current vertex has another unvisited adjacent vertex "V" )
      insert vertex "V" at the tail of queue, process it, …
                              … and mark it as visited;
}
```

TABLE 13.4 Queue Contents During Breadth-First Search

Event	Queue (Front to Rear)
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Remove B	CDE
Visit F	CDEF
Remove C	DEF
Remove D	EF
Visit G	EFG
Remove E	FG
Remove F	G
Visit H	GH
Remove G	H
Visit I	HI
Remove H	I
Remove I	
Done	




```
Insert starting vertex in the queue, process
it, and mark it as visited;
while (queue is not empty) {
    Remove the element at the head of the
    queue and make it the current vertex;

    while (current vertex has another
    unvisited adjacent vertex "V" )
      insert vertex "V" at the tail of queue,
      process it, and mark it as visited;
 }
```

Java Code – Breadth First Search - 1

```
public void bfs() // breadth-first search
{ // begin at vertex 0
   vertexList[0].wasVisited = true; // mark it
   displayVertex(0); // display it
   theQueue.insert(0); // insert at tail
   int v2;
```

Java Code – Breadth First Search - 2

```
while( !theQueue.isEmpty() ) // until queue empty,
{
  int v1 = theQueue.remove(); // remove vertex at head
  // until it has no unvisited neighbors
  while( (v2=getAdjUnvisitedVertex(v1)) != -1 )
  { // get one,
    vertexList[v2].wasVisited = true; // mark it
    displayVertex(v2); // display it
    theQueue.insert(v2); // insert it
  } // end while(unvisited neighbors)
} // end while(queue not empty)
// queue is empty, so we're done
} // end bfs()
```

Example: Breadth-first search

- The breadth-first search has an interesting property:
  - It first finds all the vertices that are one edge away from the starting point, then all the vertices that are two edges away, and so on.
  - This is useful if you're trying to find the shortest path from the starting vertex to a given vertex.
  - You start a BFS, and when you find the specified vertex, you know the path you've traced so far is the shortest path to the node.

Finding connected components

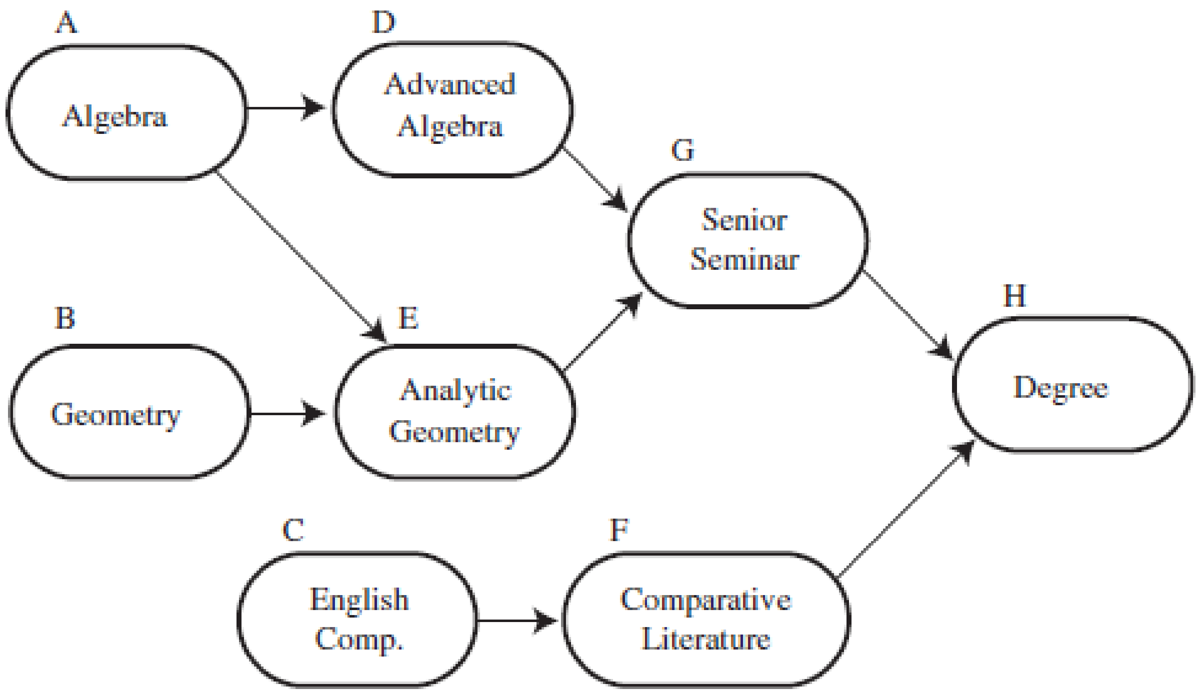
- A depth-first search can be used to find connected components of a graph
  - A connected component is a set of nodes; therefore,
  - A set of connected components is a set of sets of nodes
- To find the connected components of a graph:

```
while there is a node not assigned to a component {
    put that node in a new component
    do a DFS from the node, and put every node
        reached into the same component
}
```

Topological Sorting with Directed Graphs

Topological Sorting of Directed Graphs - 1

- An Example: Course Prerequisites
  - Some courses have prerequisites—other courses that must be taken first



Topological Sorting of Directed Graphs - 2

- Imagine that you make a list of all the courses necessary for your degree.
- You then arrange the courses in the order you need to take them.
- Obtaining your degree is the last item on the list, which might look like this:
  - BAEDGCFH
- Arranged this way, the graph is said to be *topologically sorted*. Any course you must take before some other course occurs before it in the list.
- Actually, many possible orderings would satisfy the course prerequisites. You could take the English courses C and F first, for example:
  - CFBAEDGH
- (See Workshop Applet GraphD)

Topological Sorting of Directed Graphs - 3

- The idea behind the topological sorting algorithm is unusual but simple. Two steps:
  - Step 1: Find a vertex that has no successors.
  - Step 2: Delete this vertex from the graph, and insert its label at the beginning of a list.
- These two steps are repeated until all the vertices are gone.
- The topological sort should be carried out on a graph without Cycles.
  - Such a graph is called Directed Acyclic Graph (DAG).

Connectivity in Directed Graphs

Connectivity in Directed Graphs

- To find all the connected vertices in a directed graph, you can't just start from a randomly selected vertex and expect to reach all the other connected vertices.
  - For example: If you start on A, you can get to C but not to any of the other vertices.

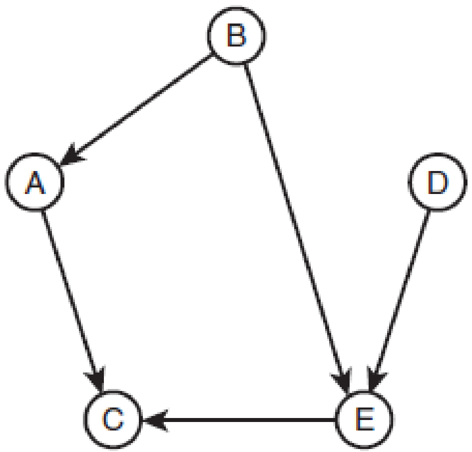


TABLE 13.6 Adjacency Matrix

	A	B	C	D	E
A	0	0	1	0	0
B	1	0	0	0	1
C	0	0	0	0	0
D	0	0	0	0	1
E	0	0	1	0	0

Connectivity in Directed Graphs

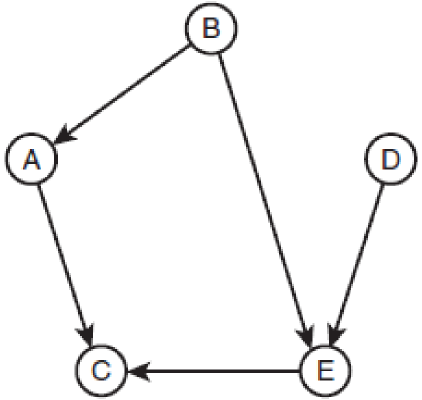
- **Warshall's Algorithm:**
- It's based on a simple idea (transitive closure):
  - If you can get from vertex L to vertex M, and you can get from M to N, then you can get from L to N.
- Consider the rows of the adjacency matrix, one row at a time and try to deduce paths between vertices.
- For example, in row1, we have a path from A to C, so if there is a path from X to A, then we can deduce that there is a path from X to C

TABLE 13.6 Adjacency Matrix

	A	B	C	D	E
A	0	0	1	0	0
B	1	0	0	0	1
C	0	0	0	0	0
D	0	0	0	0	1
E	0	0	1	0	0

Connectivity in Directed Graphs

- **Warshall's Algorithm:**



Steps in Warshall's algorithm.

TABLE 13.6 Adjacency Matrix

	A	B	C	D	E
A	0	0	1	0	0
B	1	0	0	0	1
C	0	0	0	0	0
D	0	0	0	0	1
E	0	0	1	0	0

a) y = 0

	A	B	C	D	E
A			1		
B	1		1		1
C					
D					1
E			1		

A to C and B to A
so B to C

b) y = 4

	A	B	C	D	E
A			1		
B	1		1		1
C					
D			1		1
E			1		

E to C and D to E
so D to C

Shortest Path in Weighted Directed Graph

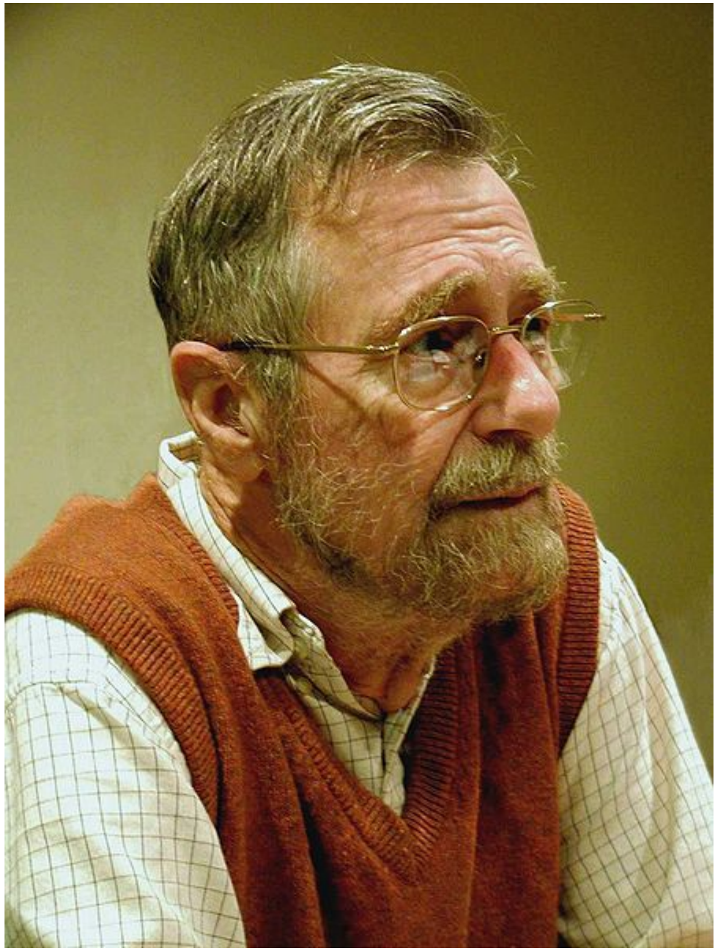
Shortest-path

- Suppose we want to find the shortest path from node X to node Y
- It turns out that, in order to do this, we need to find the shortest path from X to *all* other nodes
  - Why?
  - If we don't know the shortest path from X to Z, we might overlook a shorter path from X to Y that contains Z
- Dijkstra's Algorithm finds the shortest path from a given node to *all* other reachable nodes in a Directed Graph

Edsger Wybe Dijkstra

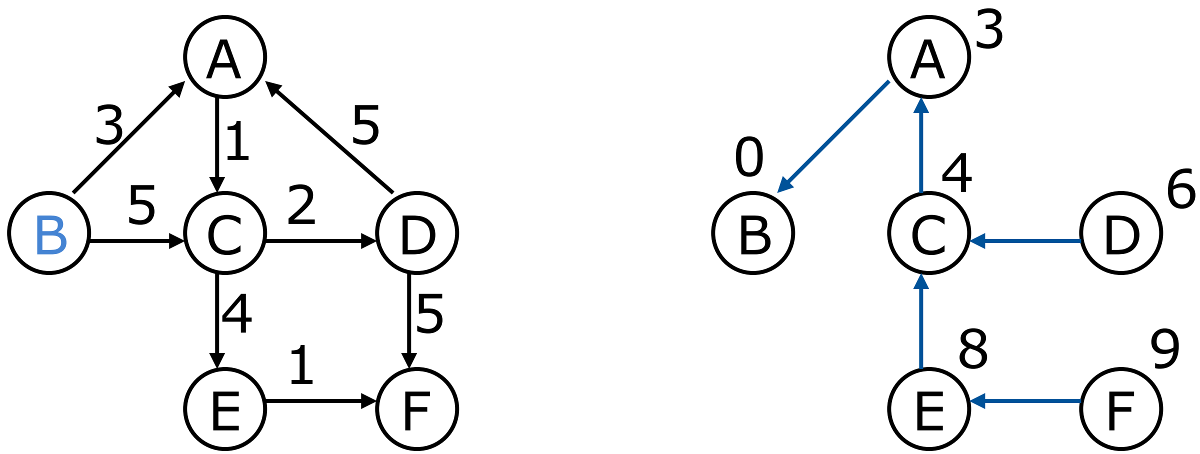
Edsger Wybe Dijkstra
Computer Scientist

(1930 – 2002)



Dijkstra's algorithm I

- Dijkstra's algorithm builds up a *tree:* there is a path from each node back to the starting node
- For example, in the following graph, we want to find shortest paths from node B



- Edge values in the graph are weights
- Node values in the tree are *total* weights
- The arrows point in the *right direction* for what we need (why?)

Dijkstra's algorithm II

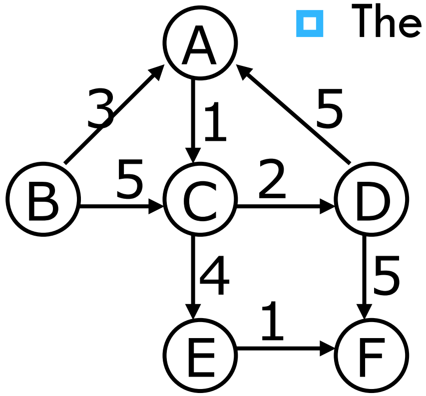
- For each vertex v, Dijkstra's algorithm keeps track of three pieces of information:
  - A boolean telling whether we *know* the shortest path to that node (initially true only for the starting node)
  - The length of the shortest path to that node known so far (0 for the starting node)
  - The predecessor of that node along the shortest known path (unknown for all nodes)
- Dijkstra's algorithm proceeds in phases—at each step:
  - From the vertices for which we don't know the shortest path, pick a vertex v with the smallest distance known so far
  - Set v's "known" field to true
  - For each vertex w adjacent to v, test whether its distance so far is greater than v's distance plus the distance from v to w; if so, set w's distance to the new distance and w's predecessor to v

Dijkstra's algorithm III

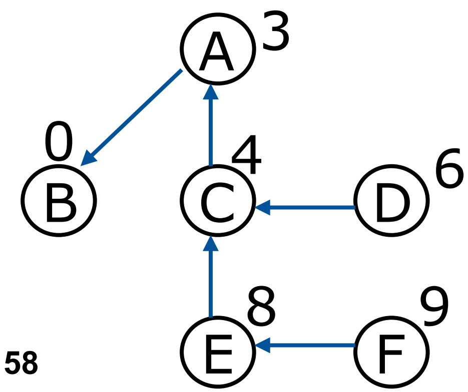
```
function Dijkstra(Graph, source):
    for each vertex v in Graph:            // Initializations
        dist[v] := infinity ;               // Unknown distance function from source to v
        previous[v] := undefined ;          // Previous node in optimal path from source
    end for ;
    dist[source] := 0 ;                     // Distance from source to source
    Q := the set of all nodes in Graph ;
    // All nodes in the graph are unoptimized - thus are in Q
    while Q is not empty:                  // The main loop
        u := vertex in Q with smallest dist[] ;
        if dist[u] = infinity:
            break ;                         // all remaining vertices are inaccessible from source
        end if ;
        remove u from Q ;
        for each neighbor v of u:          // where v has not yet been removed from Q.
            alt := dist[u] + dist_between(u, v) ;
            if alt < dist[v]:              // Relax (u,v,a)
                dist[v] := alt ;
                previous[v] := u ;
            end if ;
        end for ;
    end while ;
    return dist[] ;
end Dijkstra.
```

Dijkstra's algorithm III

- Three pieces of information for each node (e.g. +3B):
  - \+ if the minimum distance is known *for sure*, blank otherwise
  - The best distance so far (3 in the example)
  - The node's predecessor (B in the example, - for the starting node)



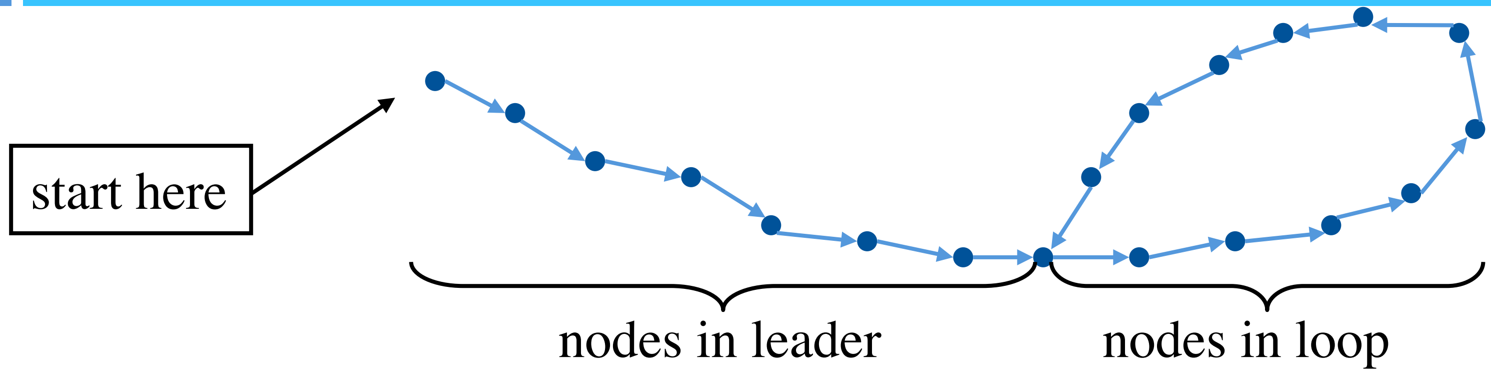
node	init'ly	1	2	3	4	5	6
A	inf	3B	+3B	+3B	+3B	+3B	+3B
B	0-	+0-	+0-	+0-	+0-	+0-	+0-
C	inf	5B	4A	+4A	+4A	+4A	+4A
D	inf	inf	inf	6C	+6C	+6C	+6C
E	inf	inf	inf	8C	8C	+8C	+8C
F	inf	inf	inf	inf	11D	9E	+9E



Summary

- A graph may be directed or undirected
- The edges (=arcs) may have weights or contain other data, or they may be just connectors
- Similarly, the nodes (=vertices) may or may not contain data
- There are various ways to represent graphs
  - The "best" representation depends on the problem to be solved
  - You need to consider what kind of access needs to be quick or easy
- Many tree algorithms can be modified for graphs
  - Basically, this means some way to recognize cycles

A graph puzzle



- Suppose you have a directed graph with the above shape
  - You don't know how many nodes are in the leader
  - You don't know how many nodes are in the loop
  - You don't know how many nodes there are total
  - You aren't allowed to mark nodes
- Devise an O(n) algorithm (n being the total number of nodes) to decide when you *must already be* in the loop
  - This is *not* asking to find the *first* node in the loop
  - You can only use a *fixed* (constant) amount of extra memory

The End