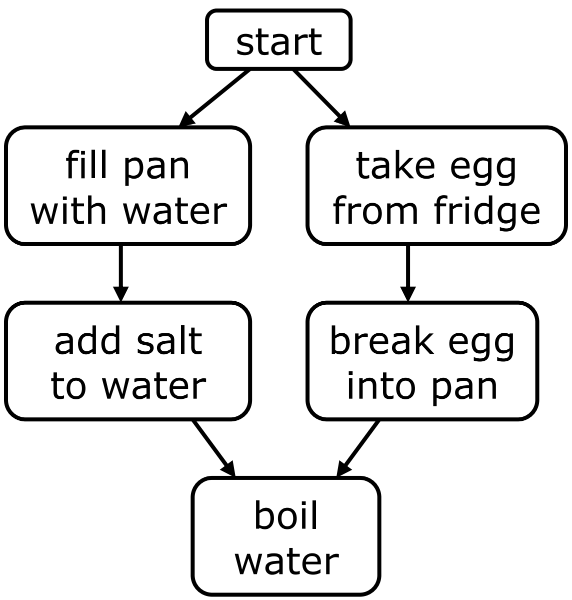
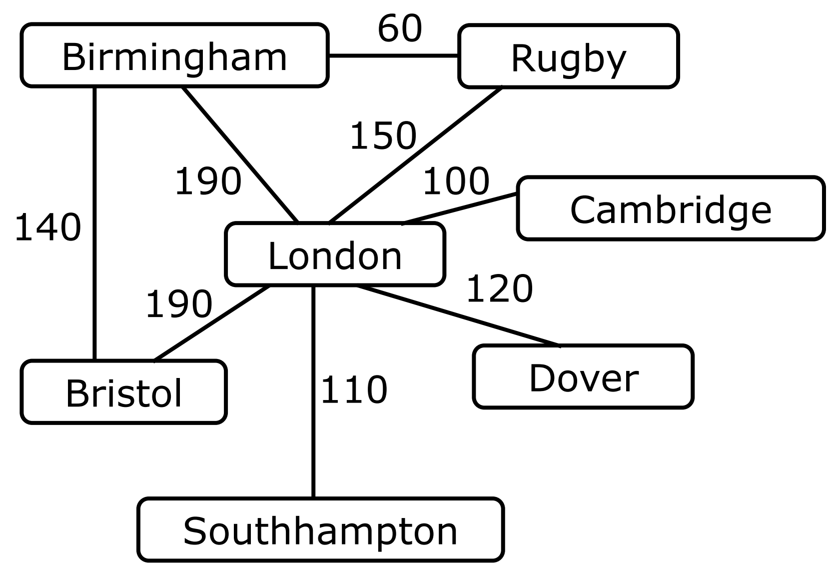
# Graphs

# 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

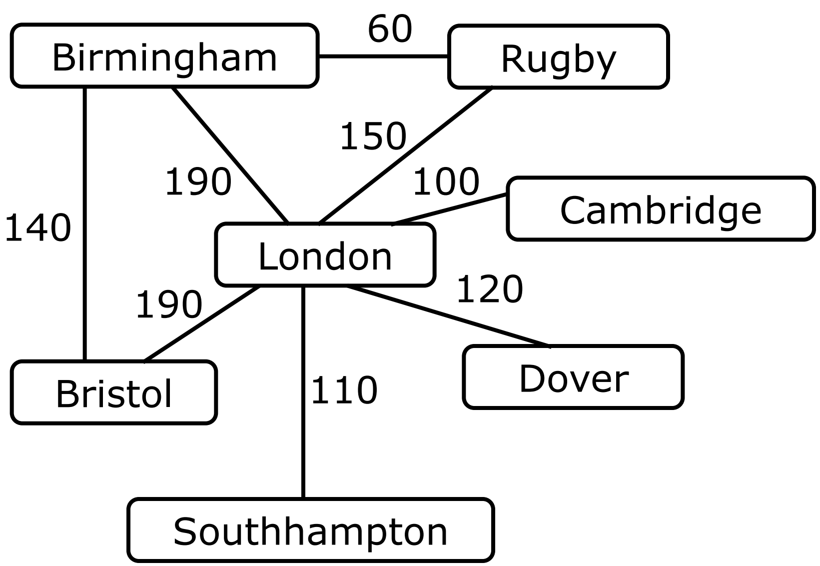
- A graph is a collection of nodes (or vertices, singular is vertex) and edges (or arcs)
  - Each node contains an element
  - Each edge connects two nodes together (or possibly the same node to itself) and may contain an edge attribute
- A directed graph is one in which the edges have a direction
- 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
- A cycle is a path whose first and last nodes are the same

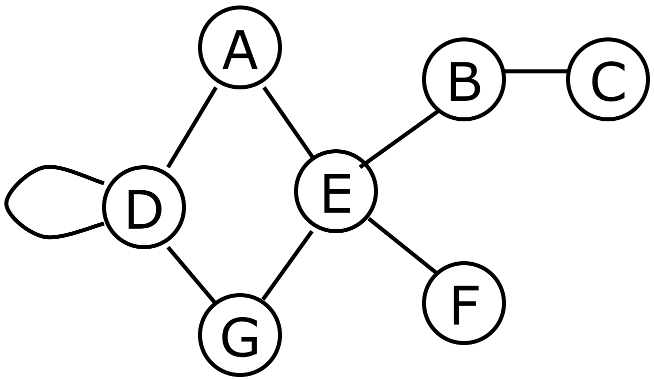


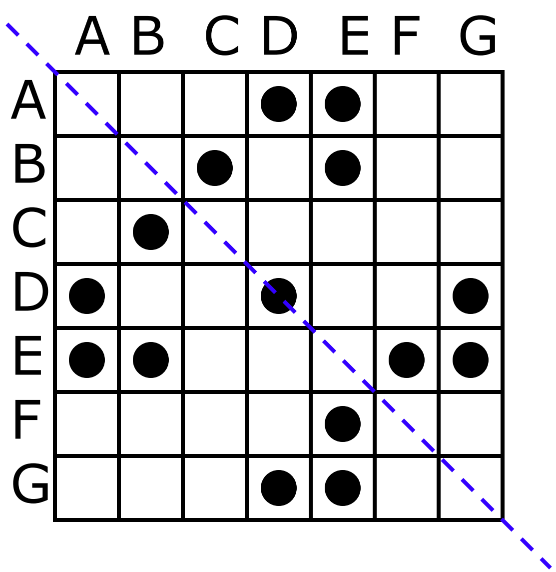

- Example: (London, Bristol, Birmingham, London, Dover) is a path
- Example: (London, Bristol, Birmingham, London) is a cycle
- A cyclic graph contains at least one cycle
- An acyclic graph does not contain any cycles

# 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

# Adjacency-matrix representation I

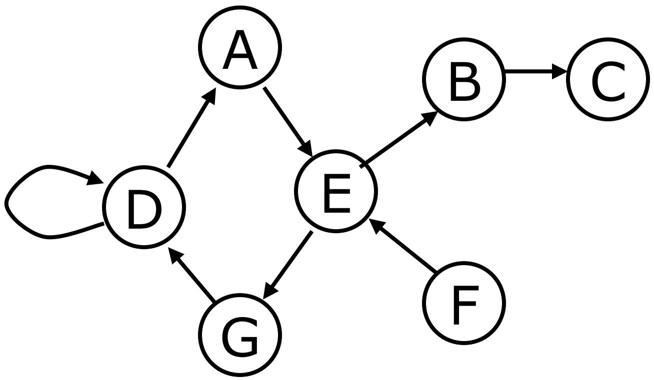



- One simple way of representing a graph is the adjacency matrix
- A 2-D array has a mark at `[i][j]` if there is an edge from node `i` to node `j`
- The adjacency matrix is symmetric about the main diagonal

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

# Adjacency-matrix representation II



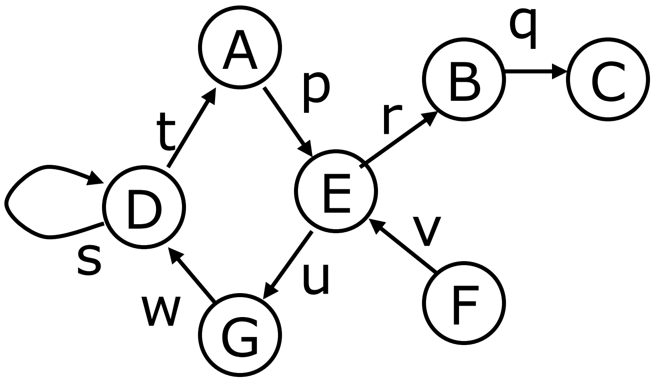

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| A | | | | | ● | | |
| B | | | ● | | | | |
| C | | | | | | | |
| D | ● | | | ● | | | |
| E | | ● | | | | | ● |
| F | | | | | ● | | |
| G | | | | ● | | | |

- 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




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) }

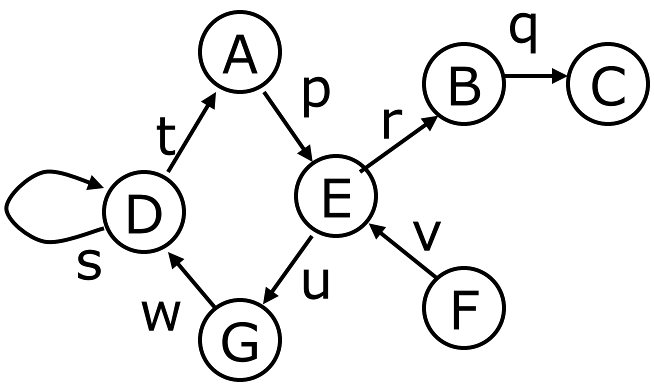
- Here we have a set of nodes, and each node contains only its element (not shown)

- 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




A → { p }
B → { q }
C → { }
D → { s, t }
E → { r, u }
F → { v }
G → { w }

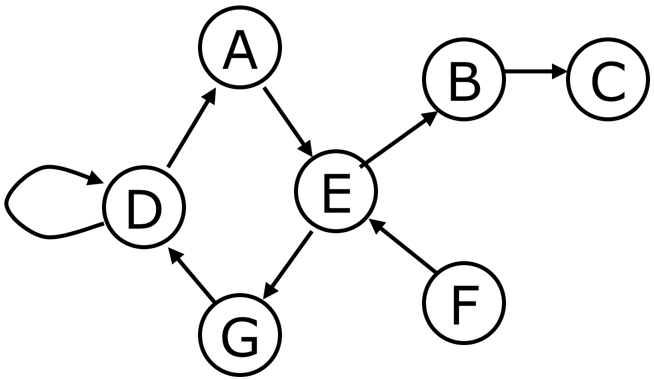
p: (A, E)
q: (B, C)
r: (E, B)
s: (D, D)
t: (D, A)
u: (E, G)
v: (F, E)
w: (G, D)

- 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




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

- 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

- With certain modifications, any tree search technique can be applied to a graph
  - This includes depth-first, breadth-first, depth-first iterative deepening, and other types of searches
- The difference is that 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 nodes we have already explored
- There are two basic techniques for this:
  - Keep a set of already explored nodes, or
  - Mark the node itself as having been explored
    - Marking nodes is not always possible (may not be allowed)

# Example: Depth-first search

- **Here is how to do DFS on a tree:**

```
Put the root node on a stack;
while (stack is not empty) {
    remove a node from the stack;
    if (node is a goal node) return success;
    put all children of the node onto the stack;
}
return failure;
```

- **Here is how to do DFS on a graph:**

```
Put the starting node on a stack;
while (stack is not empty) {
    remove a node from the stack;
    if (node has already been visited) continue;
    if (node is a goal node) return success;
    put all adjacent nodes of the node onto the stack;
}
return failure;
```

# 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
}
```

# Graph applications

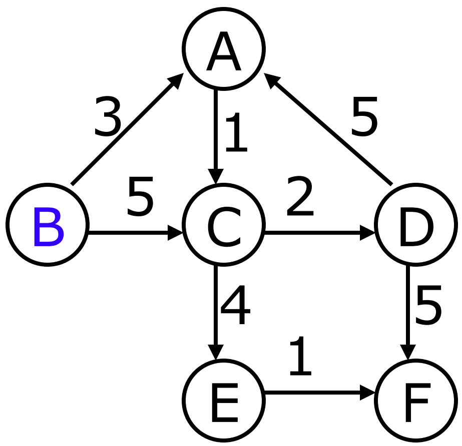
- Graphs can be used for:
  - Finding a route to drive from one city to another
  - Finding connecting flights from one city to another
  - Determining least-cost highway connections
  - Designing optimal connections on a computer chip
  - Implementing automata
  - Implementing compilers
  - Doing garbage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

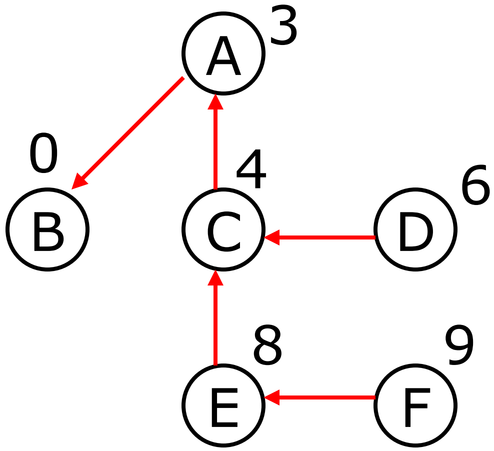
# Shortest-path

- 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

# 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



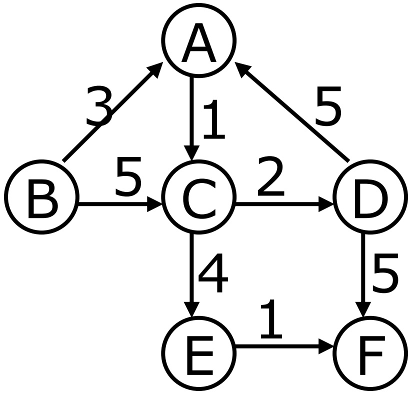

- Edge values in the graph are weights
- Node values in the tree are *total* weights
- The arrows point in the *right direction* for what we need (why?)

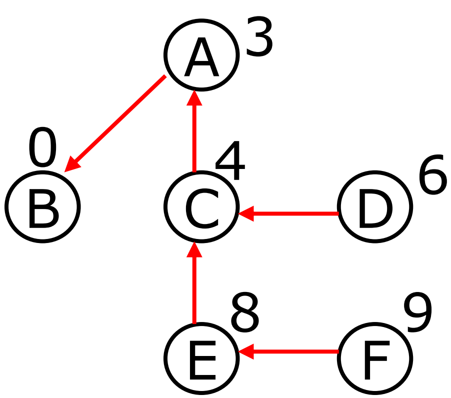
# Dijkstra's algorithm II

- 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

- 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)



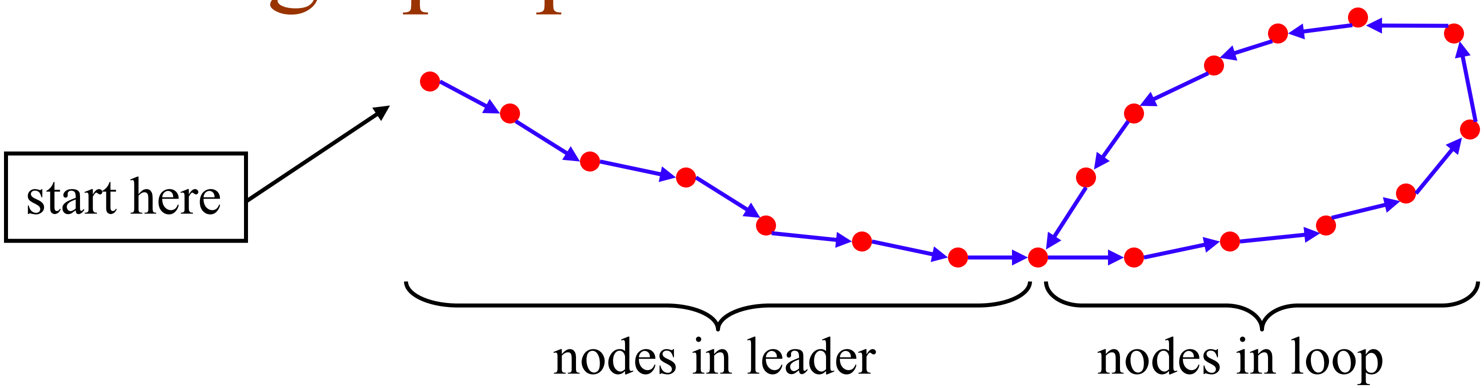




| node | init'ly | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| A | inf | 3B | 3B | +3B | +3B | +3B | +3B |
| B | 0- | +0- | +0- | +0- | +0- | +0- | +0- |
| C | inf | 5B | 4A | +4A | +4A | +4A | +4A |
| D | inf | inf | inf | 6C | +6C | +6C | +6C |
| E | inf | inf | inf | 8C | +8C | +8C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