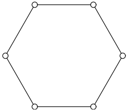
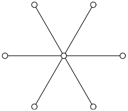
Graphs/Graph Algorithms

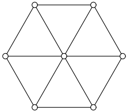
A graph is a set of objects (called vertices, nodes, or points) and a set of pairs of objects (called edges or lines)



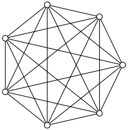
Ring



Star



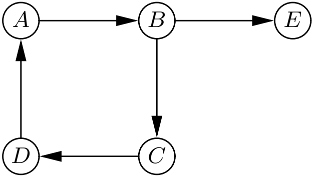
Wheel

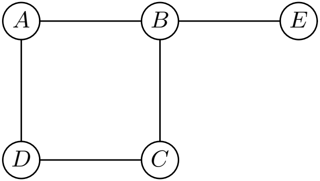


Complete Graph

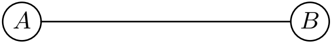
Directed vs. Undirected graphs

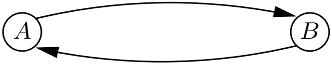
- Graphs can be undirected or directed.
- In a directed graph, or digraph, the edges have directions.





- It is possible to define a mixed graph with some edges undirected, others directed.
- For some purposes, we can replace an edge in an undirected graph by a pair of antiparallel directed edges



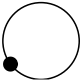


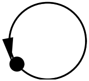
Examples of graphs (from [GT, Section 13.1])

- Visualizing binary relations.
  - Symmetric relations (e.g., "coauthored a paper with")
  - Asymmetric relations (e.g., "inherits from")
- City map
- Wiring/plumbing networks in buildings
- Flight network
- The Internet

Loops and Multiedges

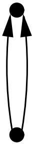
- A loop (sometimes called a self-loop) is an edge with both endpoints the same



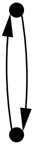


- Multiedges (also called multiple edges, or parallel edges)
  - A multiedge in an undirected graph is a pair of edges with the same endpoints.
  - A multiedge in a directed graph is a pair of edges with the same endpoints and the same direction.

Multiedge



Multiedge



Not a Multiedge

Simple graphs

- A simple graph is a graph that has no loops and no multiedges.
- Unless we state otherwise, we will assume that all graphs are simple.

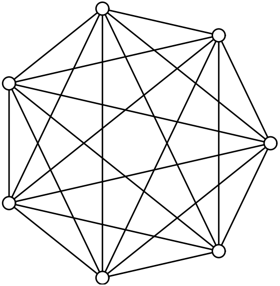
Notation

We write

- $G = (V, E)$ for the graph with vertex set V and edge collection E.
- n for the number of vertices of a graph
- m for the number of edges

Complete Graphs

- A complete graph is an undirected graph with an edge between every pair of vertices.
- A complete graph on n vertices has $m = \binom{n}{2}$ edges

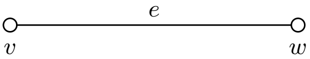


Complete Graph on 7 vertices
$(n = 7, m = 21)$

Some graph terminology

If $e = vw$ is an edge of a graph, we say:

- v (or w) is incident on e.
- e is incident on v (or on w).
- v (or w) an endpoint of e.
- v and w are adjacent (or neighbors)

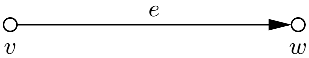


- The degree of a vertex v is the number of edges incident on v.

Some additional terminology for digraphs

If $e = vw$ is an edge of a digraph, we say:

- v is the origin, or tail, of e.
- w is the destination, or head, of e.
- e is an outgoing edge of v.
- e is an incoming edge of w.



- The indegree of a vertex is the number of incoming edges.
- The outdegree of a vertex is the number of outgoing edges.

Some useful formulae:

1. In any graph
$$\sum_{v \in V(G)} \texttt{degree}(v) = 2m$$

2. In any directed graph
$$\sum_{v \in V(G)} \texttt{indegree}(v) = m$$

3. In any directed graph
$$\sum_{v \in V(G)} \texttt{outdegree}(v) = m$$

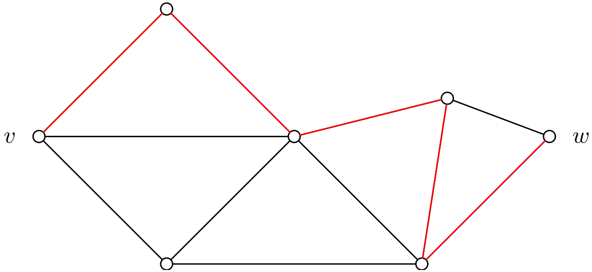
Paths

- A path from vertex v to vertex w is a sequence of edges

  $$v_0v_1,\ v_1v_2,\ \ldots,\ v_{k-1}v_k$$

  1. $v_0 = v$
  2. $v_k = w$

- The path is a simple path if
  3. The vertices $v_0, v_1, \ldots, v_k$ are all distinct



Cycles

- In an undirected graph, a cycle is a set of edges

  $$v_0v_1,\ v_1v_2, \ldots, v_{k-2}v_{k-1},\ v_{k-1}v_k$$

  such that:

  1. $v_0 = v_k$
  2. $k \geq 3$

- The cycle is simple if

  3. $v_0,\ v_1,\ \ldots,\ v_{k-1}$ are all distinct.

- k is called the length of the cycle.
- A graph is acyclic if it has no simple cycles.
- In a directed graph, the definition is similar, but we can omit the requirement that $k \geq 3$, and we can omit the word "simple" in the definition of acyclic.

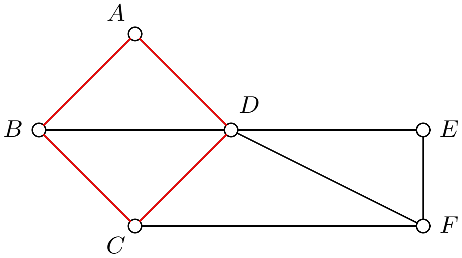
Cycles, continued

Usually when we write a cycle we just list the vertices, since the edges are implicit:

$$v_0, v_1, \ldots, v_{k-1}, v_k$$

Sometimes we omit the last vertex:

$$v_0, v_1, \ldots, v_{k-1}$$

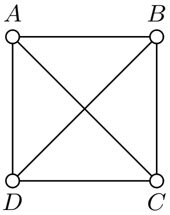


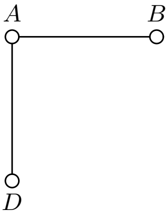
So we can write: *AB,BC,CD,DA*
or *ABCDA*
or just *ABCD*

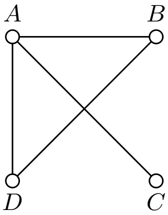
Subgraphs

- A subgraph of $G = (V, E)$ is a graph $H = (V', E')$ such that

  $V' \subseteq V$ and $E' \subseteq E$



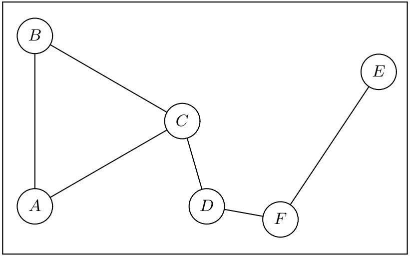




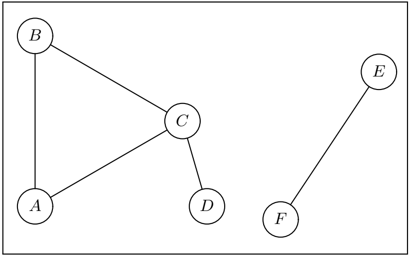
- H must be a valid graph. So every endpoint of an edge in E' must belong to V'.
- A spanning subgraph of G is a subgraph that contains all vertices of G.

Connected Graphs

- An undirected graph is connected if there is a path between any pair of vertices.



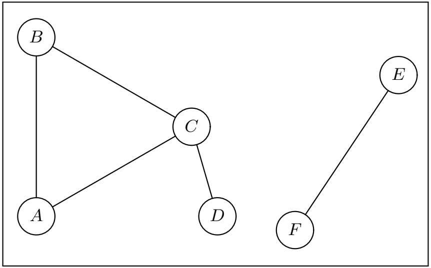
Connected



Not connected

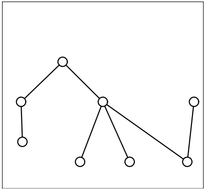
Connected Components

- If a graph G is not connected, a connected component of G is a maximal connected subgraph of G.
- Connected components can also be defined in terms of an equivalence relation, reachability, on the vertices of G.
  - Vertex w is reachable from v if there is a path from v to w.
  - This relation is an equivalence relation (reflexive, symmetric, transitive)
  - The connected components are the equivalence classes of vertices, together with the edges connecting vertices in the equivalence class.

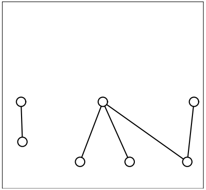


Trees

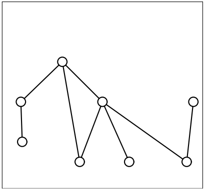
- A graph is acyclic if it has no simple cycles.
- A tree is a connected, acyclic graph.



A tree



Acyclic, but not connected (a forest)

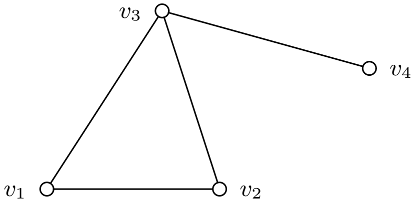


Connected, but not acyclic

Representations of graphs

1. Graph drawing
2. Edge List
3. Adjacency matrix
4. Adjacency list

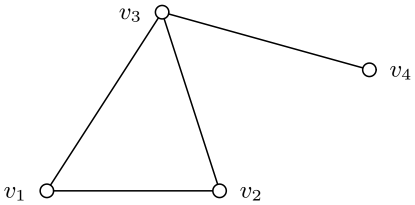
1. Graph drawing



- Good for reasoning on paper, or for a GUI.

2. Edge List

- List of vertices, edges.
- Simple description for input, output.



$$V = (v_1, v_2, v_3, v_4)$$

$$E = (v_1 v_2, v_1 v_3, v_2 v_3, v_3 v_4)$$

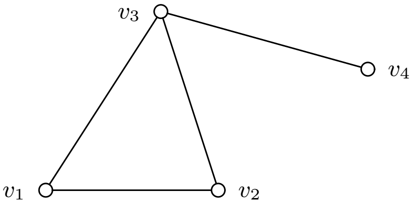
3. Adjacency matrix

- Represent G with a matrix (2D array):

$$a_{i,j} = \begin{cases} 1 & \text{if } v_i v_j \text{ is an edge of } G \\ 0 & \text{otherwise} \end{cases}$$

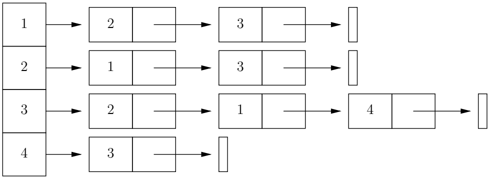
- Space requirement $= \Theta(n^2)$
- Can modify to handle:
  - Directed graphs
  - Weighted graphs

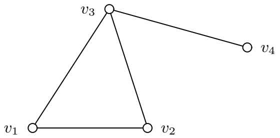
	1	2	3	4
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1
4	0	0	1	0



4. Adjacency list

- Vertices are stored in an array
- For each vertex, there is a pointer to a linked list describing its neighbors
- Space requirement $= \Theta(n + m)$
- Can modify to handle:
  - Directed graphs
  - Weighted graphs





Systematic traversal of graphs and digraphs

Two basic approaches:

- Depth-first search
- Breadth-first search

Examples of applications of depth-first-search:

- Testing whether a graph is connected
- Computing a spanning forest of a graph G (i.e., a subgraph that is a forest and contains every vertex of G)
- Computing the connected components of G
- Computing a path between two vertices v and w in a graph G (or reporting that no such path exists)
- Computing a cycle in an undirected graph G (or reporting that G is acyclic)

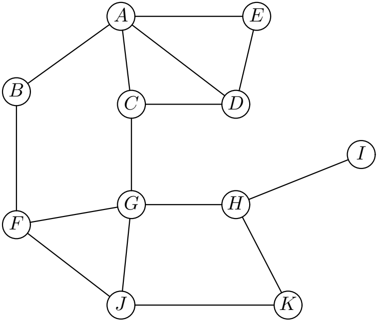
©M. B. Dillencourt—University of California, Irvine

Depth-first search, Breadth-first search

- String/paint analogy
- Depth-first search: Follow path as far as possible, back up when dead end is reached.

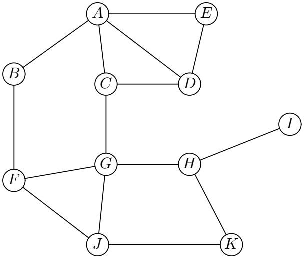
  e.g., *ABFGCDE HI KJ*
- Breadth-first search: Visit all neighbors of start vertex, then their neighbors, then neighbors of neighbors, etc.

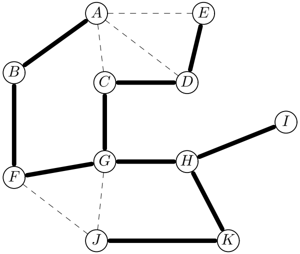
  e.g., *A BCDE FG JH KI*



Depth First Search (DFS) in an undirected graph

- Builds a DFS-forest (sometimes called a DFS-tree)
- Two kinds of edges:
  - tree edge, or discovery edge
  - back edge
- DFS-forest depends on choices of vertex ordering, ordering of neighbors about each vertex.
- Algorithms based on DFS return correct result for every result of these choices





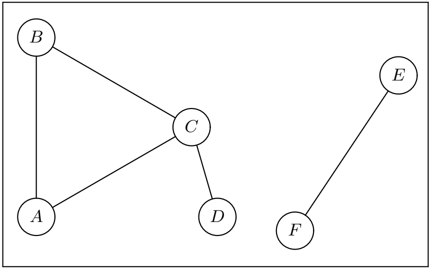
Pseudocode for DFS in an undirected graph

- Initially, each edge and each vertex is `unexplored`

```
def DFS(G,v):
  label v as explored
  for all edges e incident on v:
    if edge e is unexplored:
      w ← opposite(v,e) // opposite endpoint of e from v
      if vertex w is unexplored:
        label e as a discovery edge
        recursively call DFS(G,w)
      else:
        label e as a back edge
```

Analysis of the DFS algorithm

- `DFS(G,v)` is called once per vertex
- Each edge is examined twice (once in each direction)
- Hence, total running time is $O(n+m)$.
- We may have to restart DFS multiple times to visit the entire graph.
- Running time will still be $O(n+m)$.



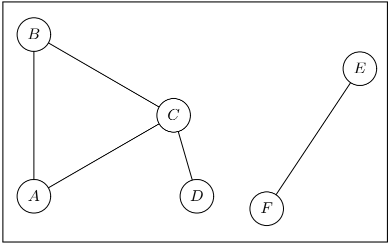
Connected-component labeling

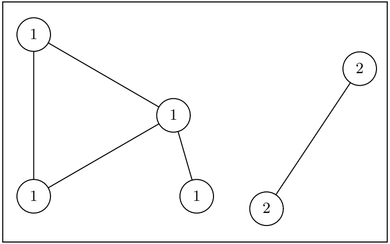
Simple application of depth-first search in undirected graphs

Input: A graph G

Output: Each vertex $v \in V(G)$ is assigned a label, $L(v)$, such that two vertices are in the same connected component if and only if they have the same label.

Example:





Pseudocode for connected component labeling

- Initially, each vertex label is `null`, and each edge is `unexplored`.

Top level:

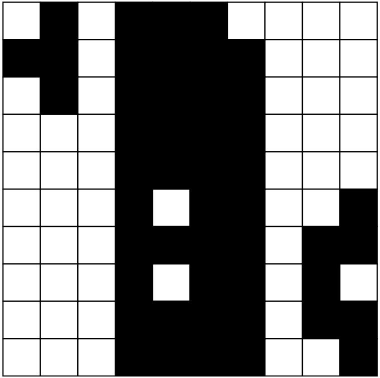
```
k = 0
for all vertices v in V(G):
  if v.label = null:
    k = k + 1
    DFSLabel(G,v,k)
```

Recursive Function to traverse and label the component:

```
def DFSLabel(G,v,k):
  v.label = k
  for all edges e incident on v:
    if edge e is unexplored:
      w ← opposite(v,e)
      mark e as explored
      if (w.label = null):  DFSLabel(G,w,k)
```

Analysis: Runs in $O(m + n)$ time.

One application of connected-component labeling: Image processing



Binary image (black pixels are "on", white pixels are "off")

	1		2	2	2				
1	1		2	2	2	2			
	1		2	2	2	2			
			2	2	2	2			
			2	2	2	2			
			2		2	2			3
			2	2	2	2		3	3
			2		2	2		3	
			2	2	2	2		3	3
			2	2	2	2			3

Connected components of pixels in image that are "on"

Biconnected components,separation edges, separation vertices

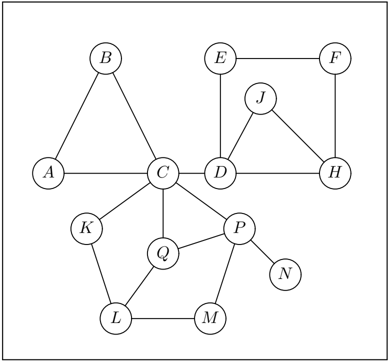
(Material from [GT] Section 13.5. Read it!)

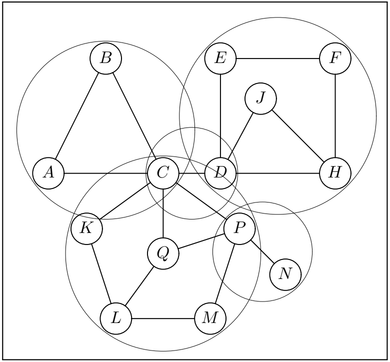
Let G be a connected graph.

- A separation edge is an edge whose removal causes G to become disconnected.
- A separation vertex is a vertex whose removal causes G to become disconnected.
- G is biconnected (or 2-connected) if for any two vertices $u, v \in V(G)$, there are at least two disjoint paths between u and v (i.e., two different paths that have no edges or vertices in common except for u and v).

Biconnected components

- Let G be a connected graph. A biconnected component (or bicomponent of G is a subgraph G' such that either:
  - G' is biconnected, and adding any additional edges or vertices of G would force it to stop being biconnected; or
  - G' consists of a single separation edge and its two endpoints

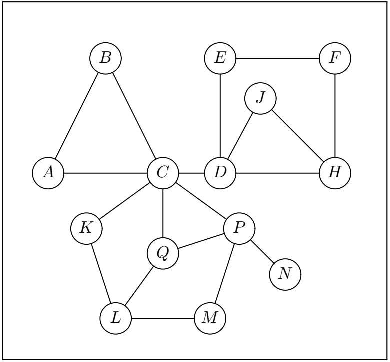


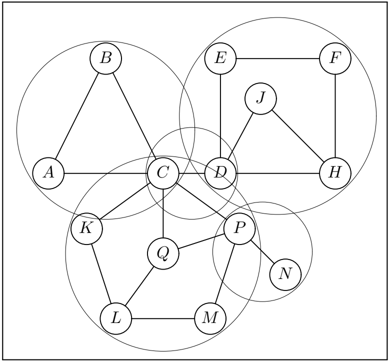


Characterization of Biconnectivity

Lemma: Let G be a connected graph. The following are equivalent:

1. G is biconnected.
2. For any two vertices of G, there is a simple cycle containing them.
3. G does not have any separating vertices or separating edges.





Equivalence relations

Let $R(x, y)$ be a binary relation on a set of objects C. R is an equivalence relation if it satisfies the following three properties:

1. Reflexive Property: $R(x, x)$ is true for each x in C.
2. Symmetric Property: $R(x, y) = R(y, x)$ for each pair x and y in C.
3. Transitive Property: If $R(x, y)$ is true and $R(y, z)$ is true, then $R(x, z)$ is true, for every x, y, and z in C.

The equivalence class of an object x is the set of all objects y such that $R(x, y)$ is true.

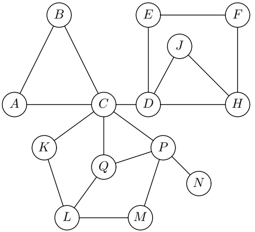
Every element of C is in exactly one equivalence class.

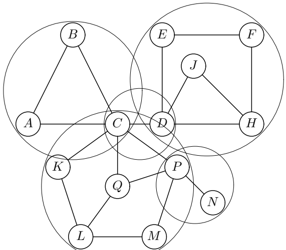
Define a link relation on the edges of a graph G

Two edges e and f in $E(G)$ are linked if $e = f$ or if G has a simple cycle containing e and f.

Lemma: Let G be a connected graph. Then

1. The link relation forms an equivalence relation on the edges of G.
2. A bicomponent is the subgraph induced by the edges of an equivalence class of linked edges.
3. Edge e is a separation edge if and only if it is in a single-element equivalence class
4. Vertex v is a separation vertex if and only if it has incident edges in two different equivalence classes.



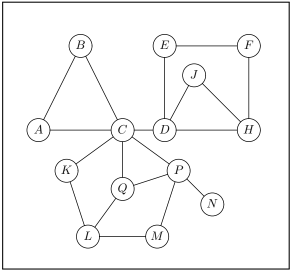


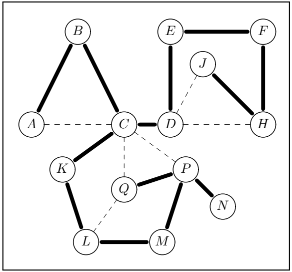
Preliminary biconnected components algorithm

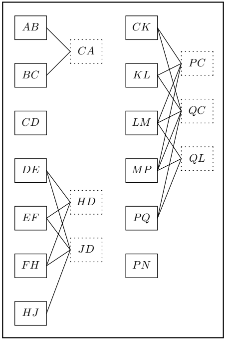
Run DFS on G. Define an auxiliary graph F as follows:

- The vertices of F are the edges of G
- For every back edge e in G, let $f_1, \ldots, f_k$ be the discovery edges of G that form a cycle with e. F contains the edges $(e, f_1), \ldots, (e, f_k)$.

Output the connected components of F. Each of these is a equivalence class of the link relation, and hence corresponds to a bicomponent of G.

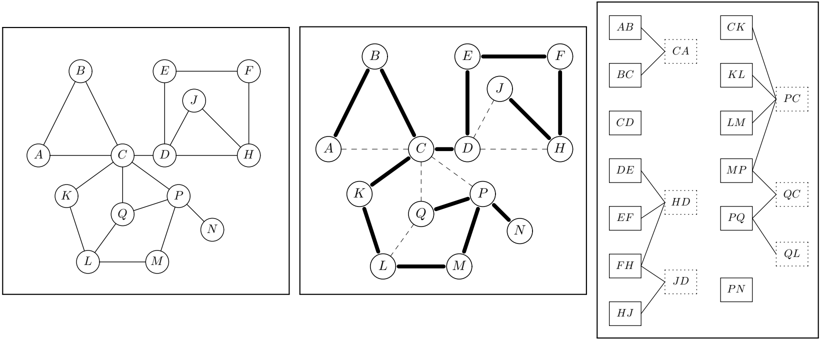






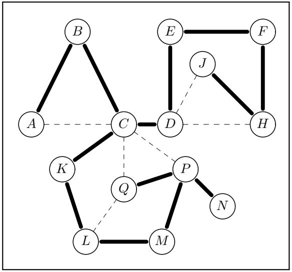
Improved bicomponent algorithm

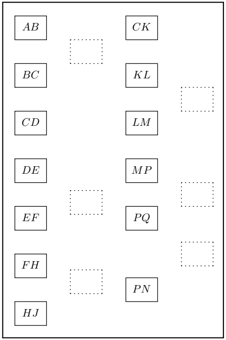
- The previous algorithm can require $\Omega(m \cdot n)$ time. We can improve this to $O(m+n)$.
- Idea behind improved algorithm: We don't need to compute F. We only need the connected components of F. So we build a spanning tree for each connected component of F (a spanning forest of F).

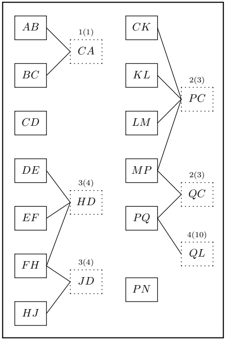


Improved bicomponent algorithm

1. Run DFS on G. Rank nodes of G according to order visited.
2. Add discovery edges of G to F. Mark discovery edges unlinked
3. Process back edges $e = (u, v)$ in order of rank of v. For each such e, let $f_1, \ldots, f_k$ be the discovery edges of G that form a cycle with e. Add edges (e, f_k), $(e, f_{k-1}), \ldots, (e, f_j)$ to F, stopping at first linked edge. Mark edges f_k, $f_{k-1}, \ldots, f_{j+1}$ linked







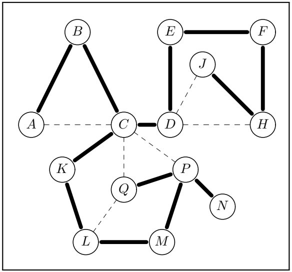
Pseudocode for Biconnected Component Algorithm

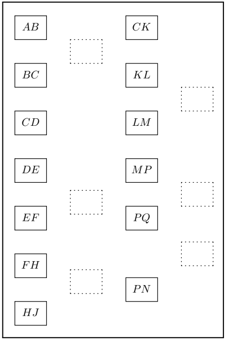
```
F ← an initially empty auxiliary graph
perform DFS traversal of G, starting at some vertex s
add each discovery edge f as a vertex in F, mark f "unlinked"
for each vertex v, in increasing rank order as visited in the
      DFS traversal
  for each back edge e=(u,v) with destination v
    add e as a vertex of the graph
    while u != v do
      Let f be the vertex in F corresponding to the
          discovery edge (parent(u),u)
      add the edge (e,f) to F
      if f is marked "unlinked" then
        mark f as "linked"
        u ← parent(u)
      else
        u ← v //exit while loop
compute the connected components of F
```

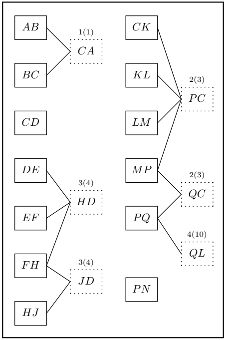
Note on the pseudocode

Note the sequencing:

1. Build DFS tree
2. Add discovery edges to F
3. Add back edges to F. Back edges are processed in the order in which their destination nodes were visited in DFS traversal.







Analysis of the Biconnected Component Algorithm

- DFS: $O(m+n)$ time.
- Building the auxiliary graph F: $O(m)$ time. Because...
  - Each iteration of the `while` loop causes an edge to be added to F.
  - When edge (e, f) is added to F:
    - Charge f if f is marked unlinked
    - Charge e if f otherwise
  - Each edge of G gets charged at most once
- Connected component analysis of F: $O(m)$ time.

So biconnected component analysis can be done in the same asymptotic time as connected component analysis.

Final note on Biconnected Component/Graph Vulnerability Analysis

As we have just seen, bicomponent analysis tells us:

- Whether G is 2-connected (i.e., whether any two vertices are joined by 2 disjoint paths).
- Whether G has a separating edge.
- Whether G has a separating vertex.

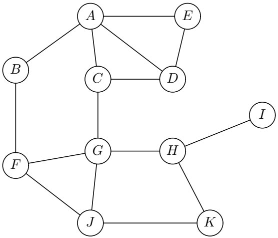
These questions can be generalized to any integer k:

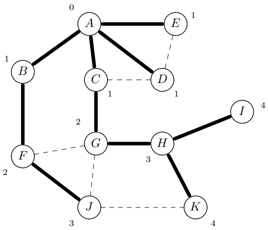
- Are any two vertices joined by k mutually disjoint paths?
- Does G have a set of $k-1$ edges whose removal makes G disconneted?
- Does G have a set of $k-1$ vertices whose removal makes G disconneted?

These questions can be answered using network flow techniques.

Breadth First Search (BFS) in an undirected graph

- Start vertex is "level 0"
- "level $i+1$" nodes are unexplored nodes that are neighbors of "level i" nodeS
- Process all nodes at one level before moving on to next level (special case: FIFO order)



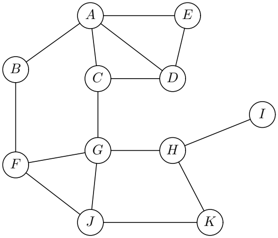


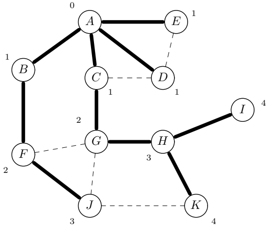
Pseudocode for BFS in an undirected graph

```
def BFS(G,s):
  Set all vertices and edges to unexplored
  create an empty Queue Q[0]
  insert s into Q[0]
  i ← 0
  while Q[i] is not empty:
    create an empty Queue Q[i+1]
    for each vertex v in Q[i]:
      for all edges e incident on v:
        if edge e is unexplored:
          w ← opposite(v,e)
          if vertex w is unexplored:
            label e as a discovery edge
            insert w into Q[i+1]
          else:
            label e as a cross edge
    i ← i+1
```

A useful property of BFS

- The level number of vertex v in BFS-tree rooted at s is the smallest number of edges in a path from s to v.

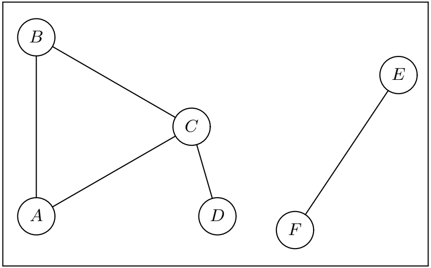




Analysis of the BFS algorithm

- Body of the outer `for` is performed n times
- Body of the inner `for` is performed $2m$ times
- Hence, total running time is $O(n+m)$.

As with DFS, we may have to restart BFS multiple times to visit the entire graph. Running time will still be $O(n+m)$.

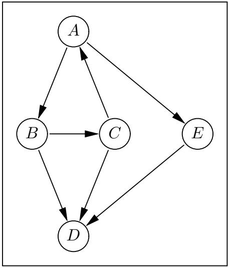


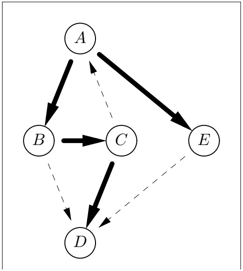
Directed graphs

- Traversing a digraph (Brief mention of why DFS is more complicated in a directed graph than an undirected graph)
- Strong connectivity
- Directed acyclic graphs/topological orderings

DFS in a directed graph

- There are four kinds of edges (as opposed to only two in an undirected graph)
  - tree edge, or discovery edge
  - back edge: connects a vertex to an ancestor in the DFS tree
  - forward edge: connects a vertex to a descendant in the DFS tree
  - cross edge: connects a vertex to another vertex that is neither an ancestor nor a descendant in the DFS tree



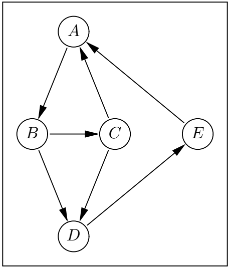


Reachability

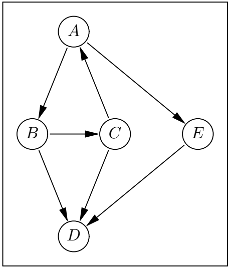
- Vertex v is reachable from vertex w in G if there is a path from w to v in G.
- In a digraph, the reachability relation is reflexive and transitive but not necessarily symmetric

Strong connectivity in digraphs

- A directed graph G is strongly connected if there is a path from every vertex in G to every other vertex in G.



Strongly Connected



Not Strongly Connected

Algorithm for testing strong connectivity in a digraph

1. Pick a start vertex s in G.
2. Run DFS in G, starting from s. If some vertex is not reachable from s, report NO and stop.
3. Let G^R be G with the direction of all edges reversed.
4. Run DFS in G^R, starting from s. If all vertices are reachable from s in G^R, report YES. Otherwise, report NO.

Analysis: Runs in $O(m+n)$ time.

Correctness:

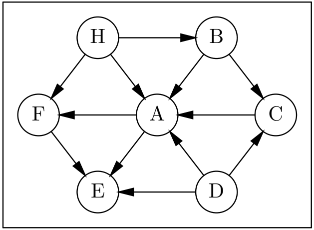
- If either step 2 or step 4 says NO, G is not strongly connected.
- If both step 2 and step 4 say YES, G is strongly connected.
  Proof: Pick any two vertices x and y in G. There is a path from x to y because . . .
  - Since step 4 said YES, there is a path from x to s
  - Since step 2 said YES, there is a path from s to y
  - Since reachability is transitive, there is a path from s to y

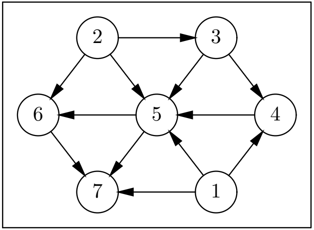
Directed Acyclic Digraphs (DAGs)

- A Directed Acylic Graph, or DAG, is a directed graph with no cycles.
- Examples:
  - Courses in a degree program, with edges representing prerequisites.
  - Classes in C++ or Java, with edges representing inheritance
  - Tasks, with edges representing scheduling constraints

Topological ordering

- A topological ordering of a digraph G is a numbering of the vertices such that every edge is directed from a lower-numbered vertex to a higher-numbered vertex.





Topological ordering

Theorem: A directed graph has a topological ordering if and only if it is a DAG

Proof: see [GT], Section 13.4.4.

Proof Sketch:

- $\Rightarrow$: easy
- $\Leftarrow$: Assume G is a DAG.
  - G must have a vertex with indegree 0.
    - Let vertex v_1 be such a vertex.
  - Remove v_1 (and all incident edges) from G.
  - Result is a smaller DAG.

Topological Sorting

- The processess of finding a topological ordering of a graph is called topological sorting
- Algorithm either
  - Finds a topological ordering, or
  - Reports that graph is not a DAG
- Algorithm repeatedly applies the following steps:
  1. Find a vertex v with indegree 0
  2. Assign v the next available label
  3. Delete v and all its outgoing edges
- Runs in $O(n+m)$ time using $O(n)$ additional space

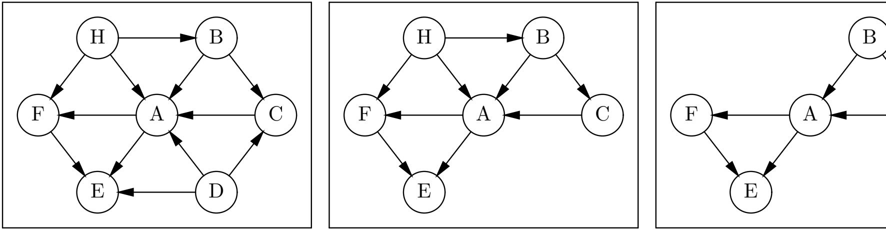
Topological Sorting Pseudocode

```
def TopologicalSort(G):
   L ← initially empty list of vertices
   for each vertex v of G:
      incounter(v) = indegree(v)
      if incounter(v) = 0:  add v to L

   i ← 0
   while L is not empty:
      choose a vertex v in L and remove it from L.
      i ← i+1
      v.number ← i
      for each edge e in v.outEdges:
         w = opposite(v,e);
         incounter(w) = incounter(w)-1;
         if incounter(w) = 0:  add w to L

   if i == n:  print the vertices and their numbers
   else:  print("G is not a DAG!!")
```

Topological Sorting Example



vertex	indegree
A	432100
B	100
C	21100
D	0
E	322100
F	21100
H	0

$L = [D, H] L = [H] L = [B] L =$

$[C] L = [A] L = [F] L = [E] L = [\,]$ Done!

Final Remark on Topological Sorting

Topological sorting is non-deterministic.

- At any step, there could be multiple vertices with indegree 0
- Any valid choice will lead to a valid topological ordering
- Note that this implies that a digraph may have many different topological orderings