Graphs

- Graphs are one of the unifying themes of computer science.

Graphs

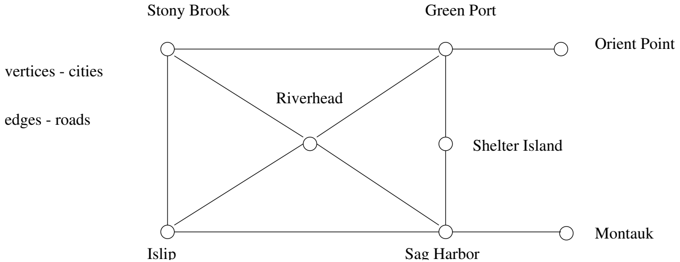
- Graphs are one of the unifying themes of computer science.
- That so many different structures can be modeled using a single formalism is a source of great power to the educated programmer.

Graphs

- Graphs are one of the unifying themes of computer science.
- That so many different structures can be modeled using a single formalism is a source of great power to the educated programmer.
- A graph $G = (V, E)$ is defined by a set of *vertices* V, and a set of *edges* E consisting of ordered or unordered pairs of vertices from V.

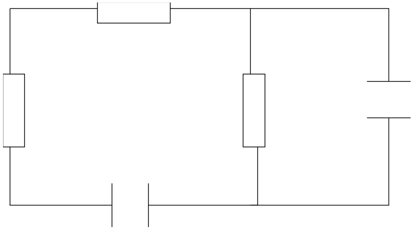
Road Networks

In modeling a road network, the vertices may represent the cities or junctions, certain pairs of which are connected by roads/edges.



Electronic Circuits

In an electronic circuit, with junctions as vertices & components as edges.



vertices: junctions

edges: components

Flavors of Graphs

- The first step in any graph problem is determining which flavor of graph you are dealing with.

Flavors of Graphs

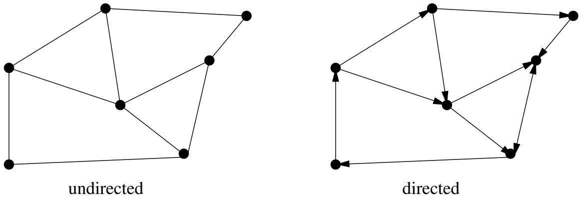
- The first step in any graph problem is determining which flavor of graph you are dealing with.
- Learning to talk the talk is an important part of walking the walk.

Flavors of Graphs

- The first step in any graph problem is determining which flavor of graph you are dealing with.
- Learning to talk the talk is an important part of walking the walk.
- The flavor of graph has a big impact on which algorithms are appropriate and efficient.

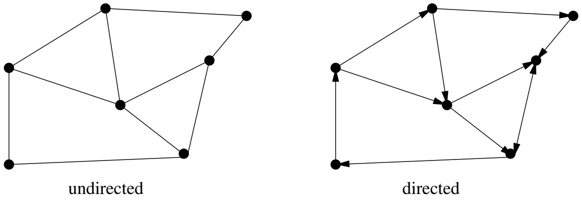
Directed vs. Undirected Graphs

A graph $G = (V, E)$ is *undirected* if edge $(x, y) \in E$ implies that (y, x) is also in E.



Directed vs. Undirected Graphs

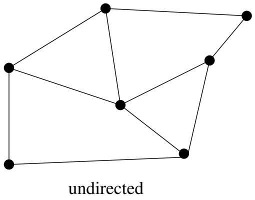
A graph $G = (V, E)$ is *undirected* if edge $(x, y) \in E$ implies that (y, x) is also in E.



Road networks *between* cities are undirected.

Directed vs. Undirected Graphs

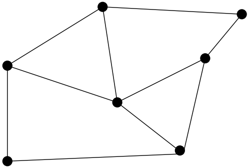
A graph $G = (V, E)$ is *undirected* if edge $(x, y) \in E$ implies that (y, x) is also in E.



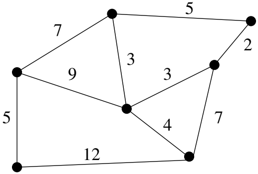
Road networks *between* cities are undirected. Street networks *within* cities may be directed because of one-way streets.

Weighted vs. Unweighted Graphs

In *weighted* graphs, each edge (or vertex) of G is assigned a numerical value, or weight.



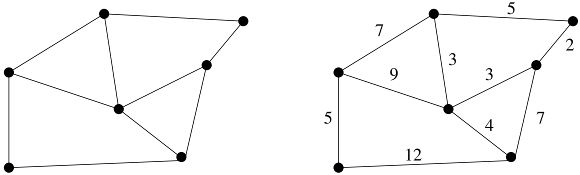
unweighted



weighted

Weighted vs. Unweighted Graphs

In *weighted* graphs, each edge (or vertex) of G is assigned a numerical value, or weight.



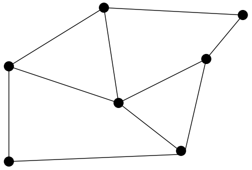
unweighted weighted

The edges of a road network graph might be weighted with their length, drive-time or speed limit.

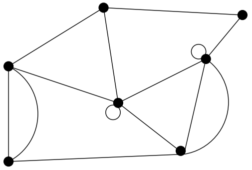
Simple vs. Non-simple Graphs

Certain types of edges complicate the task of working with graphs.

1. A *self-loop* is an edge (x, x).
2. An edge (x, y) is a *multi-edge* if it occurs more than once in the graph.



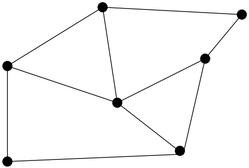
simple



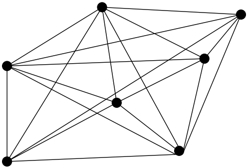
non–simple

Sparse vs. Dense Graphs

Graphs are *sparse* when a small fraction of vertex pairs actually have edges defined between them.



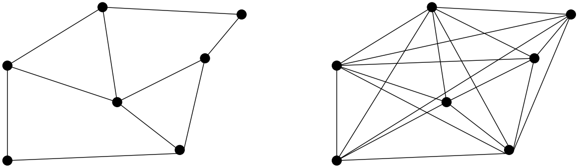
sparse



dense

Sparse vs. Dense Graphs

Graphs are *sparse* when a small fraction of vertex pairs actually have edges defined between them.

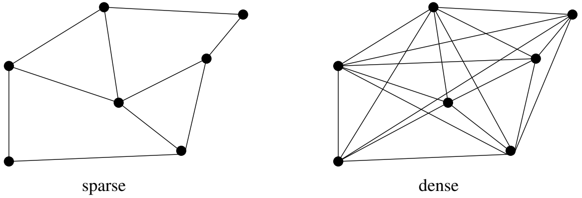


sparse dense

Road networks are sparse because of road junctions.

Sparse vs. Dense Graphs

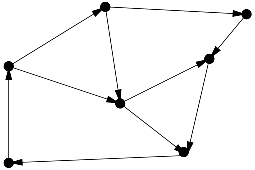
Graphs are *sparse* when a small fraction of vertex pairs actually have edges defined between them.



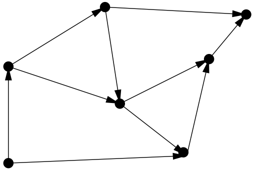
Road networks are sparse because of road junctions. Dense graphs have a quadratic number of edges while sparse graphs are linear in size.

Cyclic vs. Acyclic Graphs

An *acyclic* graph does not contain any cycles. *Trees* are connected acyclic *undirected* graphs.



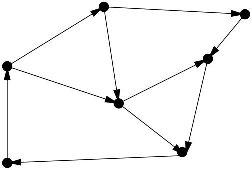
cyclic



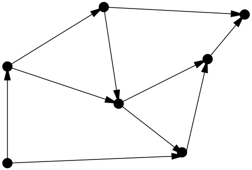
acyclic

Cyclic vs. Acyclic Graphs

An *acyclic* graph does not contain any cycles. *Trees* are connected acyclic *undirected* graphs.



cyclic

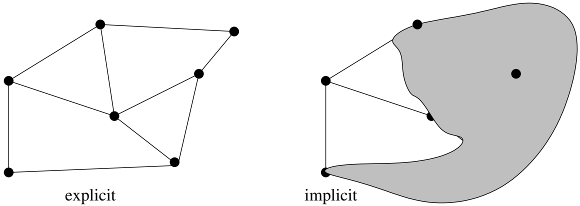


acyclic

Directed acyclic graphs are called *DAGs*. They arise naturally in scheduling problems, where a directed edge (x, y) indicates that x must occur before y.

Implicit vs. Explicit Graphs

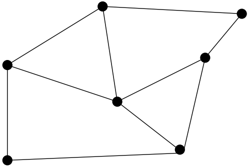
Many graphs are not explicitly constructed and then traversed, but built as we use them.



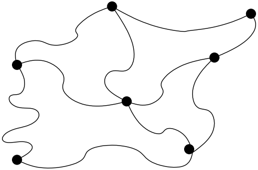
A good example arises in backtrack search.

Embedded vs. Topological Graphs

A graph is *embedded* if the vertices and edges have been assigned geometric positions.



embedded

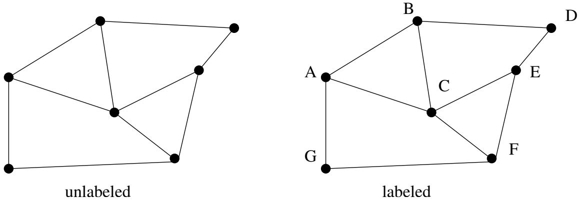


topological

Example: TSP or Shortest path on points in the plane.

Labeled vs. Unlabeled Graphs

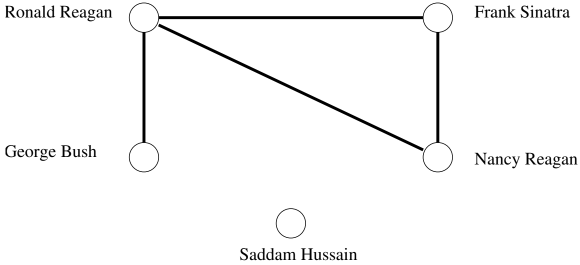
In *labeled* graphs, each vertex is assigned a unique identifier to distinguish it from all other vertices.



An important graph problem is *isomorphism testing*, determining whether the topological structure of two graphs are in fact identical if we ignore any labels.

The Friendship Graph

Consider a graph where the vertices are people, and there is an edge between two people if and only if they are friends.



If I am your friend, does that mean you are my friend?

A graph is *undirected* if (x, y) implies (y, x).
Otherwise the graph is directed.
The “heard-of” graph is directed since countless famous people have never heard of me!

Am I linked by some chain of friends to the President?

A *path* is a sequence of edges connecting two vertices.

How close is my link to the President?

If I were trying to impress you with how tight I am with the President, I would point you to the length of the *shortest path* between me and the President.

Is there a path of friends between any two people?

- An undirected graph is connected if there is a path between any two vertices.
- A directed graph is strongly connected if there is a directed path between any two vertices.

Who has the most friends?

The *degree* of a vertex is the number of edges adjacent to it.

What is the largest clique?

- A social clique is a group of mutual friends who all hang around together.

What is the largest clique?

- A social clique is a group of mutual friends who all hang around together.
- A graph theoretic *clique* is a subset of vertices where each vertex pair has an edge between them.

What is the largest clique?

- A social clique is a group of mutual friends who all hang around together.
- A graph theoretic *clique* is a subset of vertices where each vertex pair has an edge between them.
- Within the friendship graph, we would expect that large cliques correspond to workplaces, neighborhoods, religious organizations, schools, and the like.

Social Network Example

How long will it take for my gossip to get back to me?

- A *cycle* is a path where the last vertex is adjacent to the first.
- A cycle in which no vertex repeats (such as 1-2-3-1 versus 1-2-3-2-1) is said to be *simple*.

Data Structures for Graphs

There are two main data structures used to represent graphs: adjacency matrices and adjacency lists.

We assume the graph $G = (V, E)$ contains n vertices and m edges.

Adjacency Matrices

We can represent G using an $n \times n$ matrix M, where element $M[i,j]$ is 1, if (i,j) is an edge of G, and 0 if it isn't.

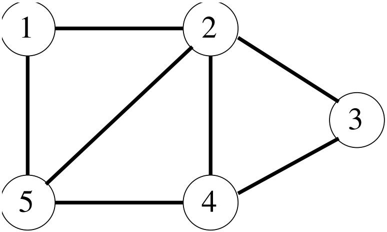
Adjacency Matrices

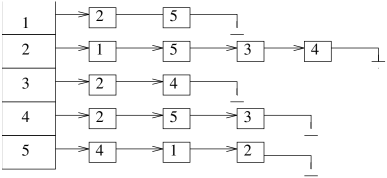
We can represent G using an $n \times n$ matrix M, where element $M[i, j]$ is 1, if (i, j) is an edge of G, and 0 if it isn't.

It uses excessive space for graphs with many vertices and relatively few edges.
Can we save space if
(1) the graph is undirected?
(2) if the graph is sparse?

Adjacency Lists

An *adjacency list* consists of an array of n pointers, where the ith element points to a linked list of the edges incident on vertex i.





Adjacency Lists (2)

To test if edge (i, j) is in the graph, we search the ith list for j, which takes $O(d_i)$, where d_i is the degree of the ith vertex.

Adjacency Lists (2)

To test if edge (i, j) is in the graph, we search the ith list for j, which takes $O(d_i)$, where d_i is the degree of the ith vertex.

d_i is much less than n when the graph is sparse.

Comparison

Comparison	Winner
Faster to test if (x, y) exists?	matrices
Faster to find vertex degree?	lists
Less memory on sparse graphs?	lists $(m + n)$ vs. (n^2)
Less memory on dense graphs?	matrices (small win)
Edge insertion or deletion?	matrices $O(1)$
Faster to traverse the graph?	lists $m + n$ vs. n^2
Better for most problems?	lists

Traversing a Graph

- One of the most fundamental graph problems is to traverse every edge and vertex in a graph.

Traversing a Graph

- One of the most fundamental graph problems is to traverse every edge and vertex in a graph.
- For efficiency, we must make sure we visit each edge at most twice.

Traversing a Graph

- One of the most fundamental graph problems is to traverse every edge and vertex in a graph.
- For efficiency, we must make sure we visit each edge at most twice.
- For correctness, we must do the traversal in a systematic way so that we don't miss anything.

Traversing a Graph

- One of the most fundamental graph problems is to traverse every edge and vertex in a graph.
- For efficiency, we must make sure we visit each edge at most twice.
- For correctness, we must do the traversal in a systematic way so that we don't miss anything.
- Since a maze is just a graph, such an algorithm must be powerful enough to enable us to get out of an arbitrary maze.

Mazes and Graphs

Marking Vertices

The key idea is that we must mark each vertex when we first visit it, and keep track of what have not yet completely explored.

Three States of a Vertex

- [Undiscovered] the vertex in its initial state.
- [Discovered] the vertex after we have encountered it, but before we have checked out all its incident edges.
- [Processed] the vertex after we have visited all its incident edges.

Three States of a Vertex

- [Undiscovered] the vertex in its initial state.
- [Discovered] the vertex after we have encountered it, but before we have checked out all its incident edges.
- [Processed] the vertex after we have visited all its incident edges.

Obviously, a vertex cannot be processed before we discover it, so the state of each vertex progresses from undiscovered to discovered to processed.

To Do List

- Need to maintain a structure containing all the vertices we have discovered but not yet completely explored.

To Do List

- Need to maintain a structure containing all the vertices we have discovered but not yet completely explored.
- Initially, only a single start vertex is considered to be discovered.

To Do List

- Need to maintain a structure containing all the vertices we have discovered but not yet completely explored.
- Initially, only a single start vertex is considered to be discovered.
- To completely explore a vertex, look at each edge going out of it. For each edge to an undiscovered vertex, mark it discovered and add it to the structure.

To Do List

- Need to maintain a structure containing all the vertices we have discovered but not yet completely explored.
- Initially, only a single start vertex is considered to be discovered.
- To completely explore a vertex, look at each edge going out of it. For each edge to an undiscovered vertex, mark it discovered and add it to the structure.
- Each edge is considered exactly twice, when each of its endpoints are explored.

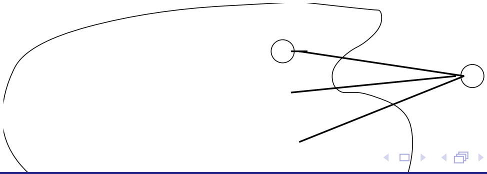
Correctness of Graph Traversal

Every edge and vertex in the connected component is eventually visited.

Correctness of Graph Traversal

Every edge and vertex in the connected component is eventually visited.

Suppose not, ie. there exists a vertex v which was unvisited whose neighbor u *was* visited. This neighbor (u) will eventually be explored so we *would* visit v:



Breadth-First Traversal

- There are two primary traversal algorithms: *breadth-first search* (BFS) and *depth-first search* (DFS).

Breadth-First Traversal

- There are two primary traversal algorithms: *breadth-first search* (BFS) and *depth-first search* (DFS).
- For certain problems, it makes absolutely no difference which one you use, but in other cases the distinction is crucial.

Breadth-First Traversal

- There are two primary traversal algorithms: *breadth-first search* (BFS) and *depth-first search* (DFS).
- For certain problems, it makes absolutely no difference which one you use, but in other cases the distinction is crucial.
- Breadth-first search is appropriate if we are interested in shortest paths on unweighted graphs.

By-Products of BFS

1. Breadth First Tree
2. Shortest path from start vertex s to each vertex x in G.

Info associated with each node u

- color$[u]$:
  WHITE $\Rightarrow$ u is undiscovered.
  GRAY $\Rightarrow$ u is discovered.
  BLACK $\Rightarrow$ u has been explored.
- $d[u]$: distance from s to u.
- $parent[u]$: u's parent in BF tree.

BFS Algorithm: Initialization

Initially, for all nodes:

- *color* is WHITE (GRAY for s)
- d is ∞ (0 for s)
- *parent* is nil.

Use an (initially empty) FIFO queue Q to store discovered vertices.

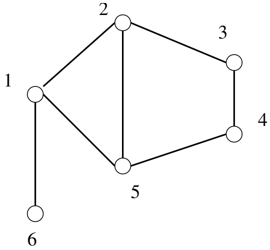
BFS Algorithm

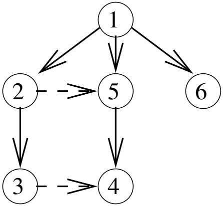
```
Enqueue(Q, s)
while (Q is not empty) do
    u = first element in Q
    for each v adjacent to u
        if (color[v] == WHITE) then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            Enqueue(Q, v)
    Dequeue(Q)
    color[u] = BLACK
```

Notes

1. d records length of shortest path from s to u.
2. Follow *parent* ptrs back to s to actually retrieve the shortest path.
3. Obtain Breadth First Tree by only considering edges of the form $(u, parent[u])$.

BFS Example





Connected Components

- The connected components of an undirected graph are the separate "pieces" of the graph such that there is no connection between the pieces.

Connected Components

- The connected components of an undirected graph are the separate "pieces" of the graph such that there is no connection between the pieces.
- Many seemingly complicated problems reduce to finding or counting connected components.

Connected Components

- The connected components of an undirected graph are the separate “pieces” of the graph such that there is no connection between the pieces.
- Many seemingly complicated problems reduce to finding or counting connected components.
- For example, testing whether a puzzle such as Rubik’s cube or the 15-puzzle can be solved from any position is really asking whether the graph of legal configurations is connected.

Finding Connected Components

- Anything we discover during a BFS must be part of the same connected component.

Finding Connected Components

- Anything we discover during a BFS must be part of the same connected component.
- We then repeat the search from any undiscovered vertex (if one exists) to define the next component, until all vertices have been found:

15-Puzzle

Two-Coloring Graphs

- The vertex coloring problem seeks to assign a label (or color) to each vertex of a graph such that no edge links any two vertices of the same color.

Two-Coloring Graphs

- The vertex coloring problem seeks to assign a label (or color) to each vertex of a graph such that no edge links any two vertices of the same color.
- A graph is bipartite if it can be colored without conflicts while using only two colors.

Two-Coloring Graphs

- The vertex coloring problem seeks to assign a label (or color) to each vertex of a graph such that no edge links any two vertices of the same color.
- A graph is bipartite if it can be colored without conflicts while using only two colors.
- Bipartite graphs are important because they arise naturally in many applications.

Finding a Two-Coloring

- We can assign the first vertex in any connected component to be whichever color we wish.

Finding a Two-Coloring

- We can assign the first vertex in any connected component to be whichever color we wish.
- We can augment breadth-first search so that whenever we discover a new vertex, we color it the opposite of its parent.

Depth-First Search

- DFS exhaustively searches all possibilities by advancing if it is possible, and backing up if it's not possible.

Depth-First Search

- DFS exhaustively searches all possibilities by advancing if it is possible, and backing up if it's not possible.
- Best understood as a recursive algorithm.

Depth-First Search

- DFS exhaustively searches all possibilities by advancing if it is possible, and backing up if it's not possible.
- Best understood as a recursive algorithm.
- Depth-first search can be thought of as breadth-first search with a stack instead of a queue.

Depth-First Search

- DFS exhaustively searches all possibilities by advancing if it is possible, and backing up if it's not possible.
- Best understood as a recursive algorithm.
- Depth-first search can be thought of as breadth-first search with a stack instead of a queue.
- The beauty of implementing DFS recursively is that recursion eliminates the need to keep an explicit stack.

DFS Algorithm

DFS(G)

for each vertex $u \in V[G]$ do

$\quad\quad color[u] = WHITE$

$\quad\quad parent[u] = nil$

$time = 0$

for each vertex $u \in V[G]$ do

$\quad\quad$ if $color[u] = WHITE$ then DFS-VISIT[u]

Visit Each Vertex

DFS-VISIT[u]

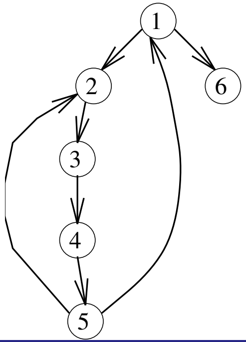
$color[u] = GREY$ //u had been white/undiscovered
$d[u] = time = time + 1$
for each $v \in Adj[u]$ do
    if $color[v] = WHITE$ then
        $parent[v] = u$
        DFS-VISIT(v)
$color[u] = BLACK$ // now finished with u
$f[u] = time = time + 1$

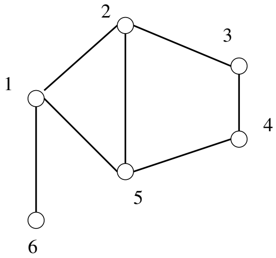
DFS Example on Directed Graph

Do on board!

DFS Example on Undirected Graph

In a DFS of an undirected graph, we assign a direction to each edge from the vertex which discovers it.





Parenthesis Theorem

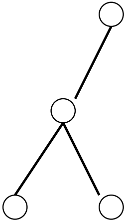
Define vertex u's range to be $[d[u], f[u]]$.

For any pair of vertices u and v, exactly one of the following holds:

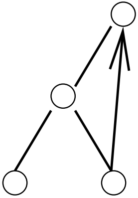
1. u's range and v's range are disjoint.
2. u's range is contained in v's range (u is a descendant of v in DFT).
3. v's range is contained in u's range (v is a descendant of u in DFT).

Edge Classification for DFS (a)

Every edge is either:

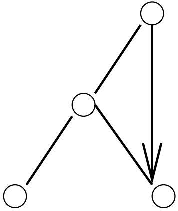


1. A Tree Edge

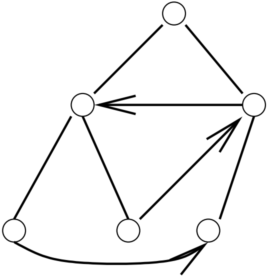


2. A Back Edge
to an ancestor

Edge Classification for DFS (b)



3. A Forward Edge
to a descendant



4. A Cross Edge
to a different node

On any DFS or BFS of a directed or undirected graph, each edge gets classified as one of four.

Edge Classification Implementation

Modify DFS to classify edges: edge (u, v) can be classified by the color of v that is reached by exploring the edge.

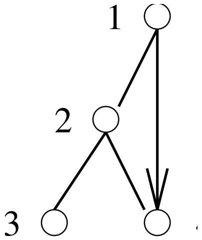
- WHITE $\Rightarrow$ tree (or just check v's parent ptr)
- GRAY $\Rightarrow$ back
- BLACK $\Rightarrow$ forward or cross.

DFS: Tree Edges and Back Edges Only

In a DFS of an UNDIRECTED graph, every edge is either a tree edge or a back edge.

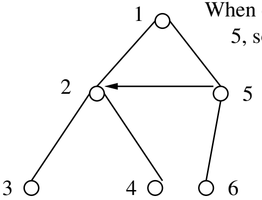
No Forward Edges in DFS

Suppose we have a forward edge. We would have encountered $(4, 1)$ when expanding 4, so this would be classified a back edge.

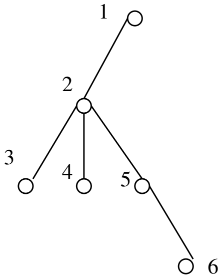


No Cross Edges in DFS

Suppose we have a cross-edge



When expanding 2, we would discover 5, so the tree would look like:

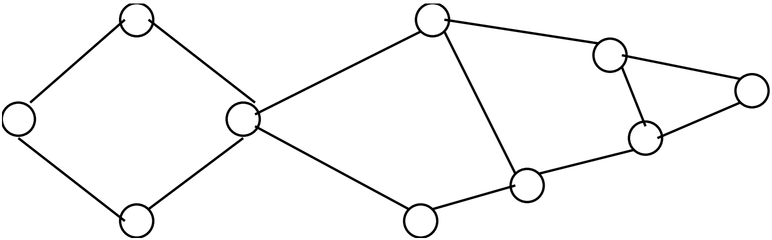


DFS Application: Finding Cycles

Back edges are the key to finding a cycle in a graph. Any back edge going from x to an ancestor y creates a cycle with the path in the tree from y to x.

Another DFS Application

Suppose you are in charge of network security. Which station do you think a terrorist would blow up to disrupt operations?



Articulation Vertices

- An *articulation vertex* is a vertex of a connected graph whose deletion disconnects the graph.

Articulation Vertices

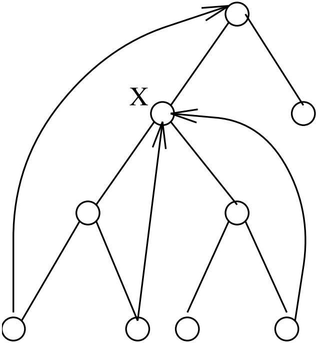
- An *articulation vertex* is a vertex of a connected graph whose deletion disconnects the graph.
- Clearly connectivity is an important concern in the design of any network.

Articulation Vertices

- An *articulation vertex* is a vertex of a connected graph whose deletion disconnects the graph.
- Clearly connectivity is an important concern in the design of any network.
- Articulation vertices can be found in $O(n(m+n))$ – just delete each vertex and do a DFS/BFS on the remaining graph to see if it is connected.

A Faster $O(n + m)$ DFS Algorithm

Run DFS once and work with resulting DFS tree:

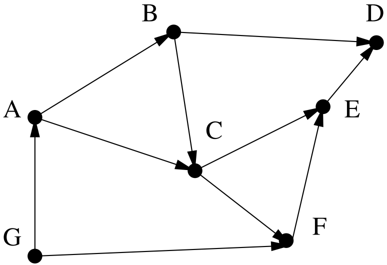


Leaves cannot be articulation vertices

The root is a special case since it has no ancestors.

X is an articulation vertex since the right subtree does not have a back edge to a proper ancestor.

Topological Sorting on DAGs



A topological sort of a graph is an ordering on the vertices so that all edges go from left to right (e.g. G, A, B, C, F, E, D).

Applications of Topological Sorting

Topological sorting is often useful in scheduling jobs in their proper sequence. In general, we can use it to order things given precedence constraints. Example: Courses in curriculum.

Algorithm

A directed graph is a DAG if and only if no back edges are encountered during a depth-first search.

Algorithm

A directed graph is a DAG if and only if no back edges are encountered during a depth-first search.
Theorem: Arranging vertices in decreasing order of DFS finish times gives a topological sort of a DAG.

Algorithm

A directed graph is a DAG if and only if no back edges are encountered during a depth-first search.
Theorem: Arranging vertices in decreasing order of DFS finish times gives a topological sort of a DAG.

Thus, topological sorting takes $O(n+m)$ time.

Proof of Theorem

Consider any directed edge u, v, when we encounter it during the exploration of vertex u:

- If v is white - we start (and finish) a DFS of v before we continue with u.
- If v is grey - then u, v is a back edge, which cannot happen in a DAG.
- If v is black - we have already finished with v, so $f[v] < f[u]$.