

Graph Algorithms

- Many problems in CS can be modeled as **graph problems**.
- Algorithms for solving graph problems are fundamental to the field of algorithm design.

Definition

- A graph $G = (V, E)$ consists of a **vertex set** V and an **edge set** E . $|V| = n$ and $|E| = m$.
- Each edge $e = (x, y) \in E$ is an **unordered pair of vertices**.
- If $(u, v) \in E$, we say v is a **neighbor of u** .
- The **degree** $\deg(u)$ of a vertex u is the number of edges **incident to u** .

Fact

$$\sum_{v \in V} \deg(v) = 2m$$

This is because, for each $e = (u, v)$, e is counted twice in the sum, once for $\deg(v)$ and once for $\deg(u)$.

Directed Graphs

Definition

- If the two end vertices of e are ordered, the edge is directed, and we write $e = x \rightarrow y$.
- If all edges are directed, then G is a directed graph.
- The in-degree $\deg_{in}(u)$ of a vertex u is the number of edges that are directed into u .
- The out-degree $\deg_{out}(u)$ of a vertex u is the number of edges that are directed from u .

Fact

$$\sum_{v \in V} \deg_{in}(v) = \sum_{v \in V} \deg_{out}(v) = m$$

This is because, for each $e = (u \rightarrow v)$, e is counted once ($\deg_{in}(v)$) in the sum of in-degrees, and once ($\deg_{out}(u)$) in the sum of out-degrees.

Graph Algorithms

- The numbers n ($= |V|$) and m ($= |E|$) are two important parameters to describe the size of a graph.
- It is easy to see $0 \leq m \leq n^2$.
- It is also easy to show: if G is a tree (namely undirected, connected graph with no cycles), then $m = n - 1$.
- If m is close to n , we say G is **sparse**. If m is close to n^2 , we say G is **dense**.
- Because n and m are rather independent to each other, we usually use both parameters to describe the runtime of a graph algorithm. Such as $O(n + m)$ or $O(n^{1/2}m)$.

Graph Representations

We mainly use two **graph representations**.

Adjacency Matrix Representation

We use a 2D array $A[1..n, 1..n]$ to represent $G = (V, E)$:

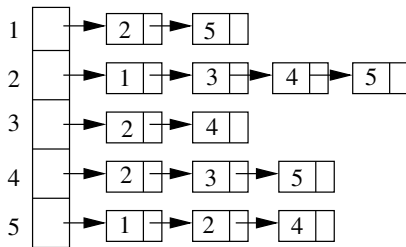
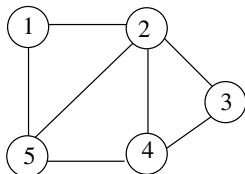
$$A[i, j] = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{if } (v_i, v_j) \notin E \end{cases}$$

- Sometimes, there are other information associated with the edges. For example, each edge $e = (v_i, v_j)$ may have a **weight** $w(e) = w(v_i, v_j)$ (for example, MST). In this case, we set $A[i, j] = w(v_i, v_j)$.
- For undirected graph, A is always **symmetric**.
- The Adjacency Matrix Representation for **directed graph** is similar.
 $A[i, j] = 1$ (or $w(v_i, v_j)$ if G has edge weights) iff $v_i \rightarrow v_j \in E$.
- For directed graphs, $A[*, *]$ is not necessarily symmetric.

Adjacency List Representation

- For each vertex $v \in V$, there's a **linked list** $Adj[v]$. Each entry of $Adj[v]$ is a vertex w such that $(v, w) \in E$.
- If there are other information associated with the edges (such as edge weight), they can be stored in the entries of the adjacency list.
- For undirected graphs, each edge $e = (u, v)$ has **two entries in this representation, one in $Adj[u]$ and one in $Adj[v]$** .
- The Adjacency List Representation for **directed graphs** is similar. For each edge $e = u \rightarrow v$, there is an entry in $Adj[u]$.
- **For directed graphs, each edge has only one entry in the representation.**

Example



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Comparisons of Representations

Graph algorithms often need the representation to support two operations.

Neighbor Testing

Given two vertices u and v , is $(u, v) \in E$?

Neighbor Listing

Given a vertex u , list all neighbors of u .

When deciding which representation to use, we need to consider:

- The space needed for the representation.
- How well the representation supports the two basic operations.
- How easy to implement.

Comparisons of Representations

Adjacency List

- Space:
 - Each entry in the list needs $O(1)$ space.
 - Each edge has two entries in the representation. So there are totally $2m$ entries in the representation.
 - We also need $O(n)$ space for the headers of the lists.
 - Total Space: $\Theta(m + n)$.
- Neighbor Testing: $O(deg(v))$ time. (We need to go through $Adj(v)$ to see if another vertex u is in there.)
- Neighbor Listing: $O(deg(v))$. (We need to go through $Adj(v)$ to list all neighbors of v .)
- More complex.

Comparisons of Representations

Adjacency Matrix

- Space: $\Theta(n^2)$, independent from the number of edges.
 - Neighbor Testing: $O(1)$ time. (Just look at $A[i,j]$.)
 - Neighbor Listing: $\Theta(n)$. (We have to look the entire row i in A to list the neighbors of the vertex i .)
 - Easy to implement.
-
- If an algorithm needs neighbor testing more often than the neighbor listing, we should use Adj Matrix.
 - If an algorithm needs neighbor testing less often than the neighbor listing, we should use Adj List.
 - If we use Adj Matrix, the algorithm takes at least $\Omega(n^2)$ time since even set up the representation data structure requires this much time.
 - If we use Adj List, it is possible the algorithm can run in linear $\Theta(m + n)$ time.

Breadth First Search (BFS)

BFS is a simple algorithm that **travels the vertices of a given graph in a systematic way**. Roughly speaking, it works like this:

- It starts at a given **starting vertex s** .
- From s , we visits **all neighbors of s** .
- These neighbors are placed in a **queue Q** .
- Then the first vertex u in Q is considered. All neighbors of u **that have not been visited yet** are visited, and are placed in Q ...
- When finished, it builds a **spanning tree (called BFS tree)**.

Before describing details, we need to pick a graph representation. Because we need to **visit all neighbors of a vertex**, it seems we need the **neighbor listing operation**. So we use **Adj list** representation.

Input: An undirected graph $G = (V, E)$ given by Adj List.
 s : the starting vertex.

Basic Data Structures: For each vertex $u \in V$, we have

- $\text{Adj}[u]$: the Adj list for u .
- $\text{color}[u]$: It can be one of the following;
 - white, (u has not been visited yet.)
 - grey, (u has been visited, but some neighbors of u have not been visited yet.)
 - black, (u and all neighbors of u have been visited.)
- $\pi[u]$: the parent of u in the BFS tree.
- $d[u]$: the distance from u to the starting vertex s .

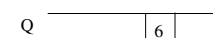
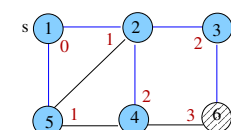
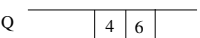
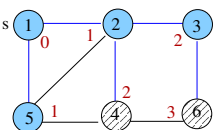
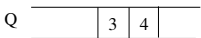
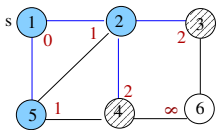
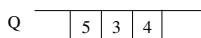
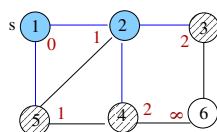
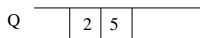
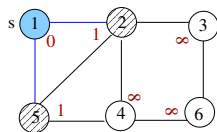
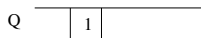
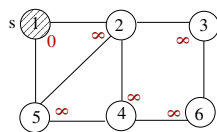
In addition, we also use a queue Q as mentioned earlier.

BFS: Algorithm

BFS(G, s)

- 1 $Q \leftarrow \emptyset$
- 2 **for each** $u \in V - \{s\}$ **do**
- 3 $\pi[u] = \text{NIL}; \quad d[u] = \infty; \quad \text{color}[u] = \text{white}$
- 4 $d[s] = 0; \quad \text{color}[s] = \text{grey}; \quad \pi[s] = \text{NIL}$
- 5 **Enqueue**(Q, s)
- 6 **while** $Q \neq \emptyset$ **do**
- 7 $u \leftarrow \text{Dequeue}(Q)$
- 8 **for each** $v \in \text{Adj}[u]$ **do**
- 9 **if** $\text{color}[v] = \text{white}$
- 10 **then** $\text{color}[v] = \text{grey}; \quad d[v] \leftarrow d[u] + 1; \quad \pi[v] \leftarrow u; \quad \text{Enqueue}(Q, v)$
- 11 $\text{color}[u] = \text{black}$

BFS: Example



2:
d[u] value

—
edges in BFS tree

○ white

◐ grey

● black

BFS: Algorithm

- BFS is not unique.
- The execution depends on the order in which the neighbors of a vertex i appear in $\text{Adj}(i)$.
- In the example above, the neighbors of i appear in $\text{Adj}(i)$ in increasing order.
- If the order is different, then the progress of the BFS algorithm would be different. And the BFS tree T constructed by the algorithm would be different.
- However, regardless of which order we use, the properties of the BFS algorithm and BFS tree are always true.

BFS: Analysis

- Lines 1 and 5: The queue operations take $O(1)$ time.
- Line 2-3: Loop takes $\Theta(n)$ time.
- Lines 4: $O(1)$ time.
- Lines 6-11:
 - Each vertex is **enqueued and dequeued exactly once**.
 - Since each queue operation takes $O(1)$ time, **the total time needed for all queue operations is $\Theta(n)$** .
 - Lines 8-10: **Each item in $Adj[u]$ is processed once**.
 - **When an item is processed, $O(1)$ operations are needed**.
 - So the total time needed is $\Theta(m) \cdot \Theta(1) = \Theta(m)$.
- Add everything together:

BFS algorithm takes $\Theta(n + m)$ time.

BFS: Main Property

Theorem

Let $G = (V, E)$ be a graph. Let $d[u]$ be the value computed by BFS algorithm. Then for any $(u, v) \in E$, $|d[u] - d[v]| \leq 1$.

Proof: First, we make the following observations:

- Each vertex $v \in V$ is **enqueued** and **dequeued** exactly once.
- Initially $\text{color}[v] = \text{white}$. When it is **enqueued**, $\text{color}[v]$ becomes grey. When it is **dequeued**, $\text{color}[v]$ becomes black. The color remains black until the end.
- The $d[v]$ value is set when v is **enqueued**. It is never changed again.
- At any moment during the execution, the vertices in Q consist of two parts, Q_1 followed by Q_2 (either of them can be empty).
 - For all $w \in Q_1$, $d[w] = k$ for some k .
 - For all $x \in Q_2$, $d[x] = k + 1$.

BFS: Main Property

Without loss of generality, suppose that u is visited by the algorithm before v . Consider the while loop in BFS algorithm, when u is at the front of Q .

There are two cases.

Case 1: $\text{color}[v] = \text{white}$ at that moment.

- Since $v \in \text{Adj}[u]$, the algorithm set $d[v] = d[u] + 1$, and $\text{color}[v] = \text{grey}$.
- $d[v]$ is never changed again. Thus $d[v] - d[u] = 1$.

Case 2: $\text{color}[v] = \text{grey}$ at that moment.

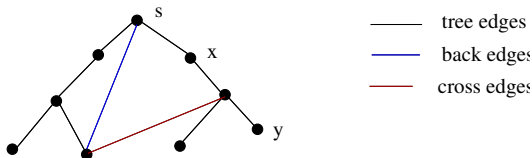
- Then v is in Q at that moment.
- By the previous observation, $d[u] = k$ for some k , and $d[v] = k$ or $k + 1$. Thus $d[v] - d[u] \leq 1$.

BFS: Main Property

Definition

Let $G = (V, E)$ be a graph and T a spanning tree of G rooted at the vertex s . Let x and y be two vertices. Let (u, v) be an edge of G .

- If x is on the path from y to s , we say x is an ancestor of y , and y is a descendent of x .
- If $(u, v) \in T$, we say (u, v) is a tree edge.
- If $(u, v) \notin T$ and u is an ancestor of v , we say (u, v) is a back edge.
- If neither u is an ancestor of v , nor v is an ancestor of u , we say (u, v) is a cross edge.



BFS: Main Property

Theorem

Let T be the BFS tree constructed by the BFS algorithm. Then there are no **back edges** for T .

Proof: Suppose there is an back edge (u, v) for T . Then $|d[u] - d[v]| \geq 2$. This is impossible.

Shortest Path Problem

Let $G = (V, E)$ be a graph and s a vertex of G . For each $u \in V$, let $\delta(s, u)$ be the length of the shortest path between s and u .

Problem: For all $u \in V$, find $\delta(s, u)$ and the shortest path between s and u .

Theorem

Let $d[u]$ be the value computed by BFS algorithm and T the BFS tree constructed by BFS algorithm. Then for each vertex $u \in V$,

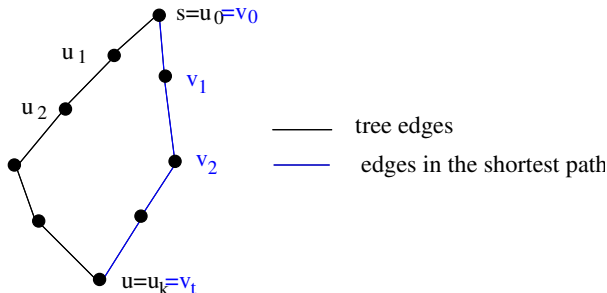
- $d[u] = \delta(s, u)$.
- The tree path in T from u to s is the shortest path.

Proof: Let $P = \{s = u_0, u_1, \dots, u_k = u\}$ be the path from s to u in the BFS tree T . Then: $d[u] = d[u_k] = k$, $d[u_{k-1}] = k - 1$, $d[u_{k-2}] = k - 2 \dots$

Suppose $P' = \{s = v_0, v_1, v_2, \dots, v_t\}$ is the shortest path from s to u in G . We need to show $k = t$.

Toward a contradiction, suppose $t < k$. Then there must exist $(v_i, v_{i+1}) \in P'$ such that $|d[v_i] - d[v_{i+1}]| \geq 2$. This is impossible.

Shortest Path Problem



BFS algorithm solves the Single Source Shortest Path problem in $\Theta(n + m)$ time.

Connectivity Problem

Definition

- A graph $G = (V, E)$ is **connected** if for any two vertices u and v in G , there exists a path in G between u and v .
- A **connected component** of G is a **maximal subgraph** of G that is connected.
- G is connected if and only if it has exactly one connected component.

Connectivity Problem

Given $G = (V, E)$, is G a connected graph?
If not, find the connected components of G .

We can use BFS algorithm to solve the connectivity problem.

Connectivity Problem

In the **BFS** algorithm, delete the lines 2-3 (initialization of vertex variables).

Connectivity($G = (V, E)$)

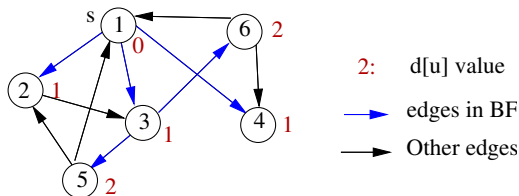
- 1 **for** each $i \in V$ **do**
- 2 $\text{color}[i] = \text{white}; d[i] = \infty; \pi[i] = \text{nil};$
- 3 $\text{count} = 0;$ (count will be the number of connected components)
- 4 **for** $i = 1$ **to** n **do**
- 5 **if** $\text{color}[i] = \text{white}$ **then**
- 6 **call** **BFS**(G, i); $\text{count} = \text{count} + 1$
- 7 **output** $\text{count};$
- 8 **end**

Connectivity Problem

- This algorithm outputs **count**, the number of connected components.
- If $\text{count} = 1$, G is connected. The algorithm also constructs a BFS tree.
- If $\text{count} > 1$, G is not connected. The algorithm also constructs a **BFS spanning forest F** of G . F is a collection of trees.
- Each tree corresponds to a connected component of G .

BFS for Directed Graphs

BFS algorithm can be applied to directed graphs without any change.



Definition

Let $G = (V, E)$ be a directed graph, T a spanning tree rooted at s . An edge $e = u \rightarrow v$ is called:

- **tree edge** if $e = u \rightarrow v \in T$.
- **backward edge** if u is a decedent of v .
- **forward edge** if u is an ancestor of v .
- **cross edge** if u and v are **unrelated**.

BFS for Directed Graphs: Property

Theorem

Let $G = (V, E)$ be a directed graph. Let T be the BFS tree constructed by BFS algorithm. Then there are no **forward edges** with respect to T .

Theorem

Let $d[u]$ be the value computed by BFS algorithm and T the BFS tree constructed by BFS algorithm. Then for each vertex $u \in V$,

- The tree path in T from s to u is the shortest path.
- $d[u]$ = the length of the shortest path from s to u .

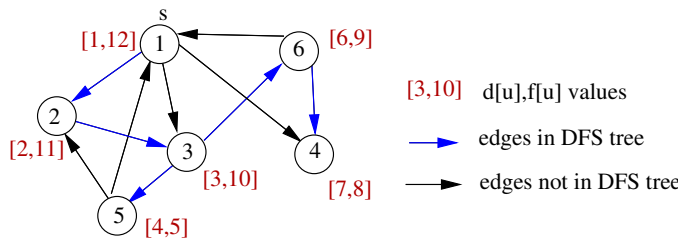
Depth First Search (DFS)

- Similar to BFS, **Depth First Search (DFS)** is another systematic way for visiting the vertices of a graph.
- It can be used on **directed** or **undirected graphs**. We discuss DFS for **directed graph** first.
- DFS has special properties, making it very useful in several applications.
- As a high level description, the only difference between BFS and DFS: replace the **queue Q** in BFS algorithm by a **stack S** . So it works like this:

High Level Description of DFS

- Start at the starting vertex s .
- Visit a neighbor u of s ; visit a neighbor v of u . . .
- Go as far as you can go, until reaching **a dead end**.
- Backtrack to a vertex that still has unvisited neighbors, and continue

DFS: Example



DFS: Recursive algorithm

- It is easier to describe the DFS by using a recursive algorithm.
- DFS also computes two variables for each vertex $u \in V$:
 - $d[u]$: The time when u is "discovered", i.e. pushed on the stack.
 - $f[u]$: the time when u is "finished", i.e. popped from the stack.
- These variables will be used in applications.

DFS: Recursive algorithm

DFS(G)

- 1 **for** each vertex $u \in V$ **do**
- 2 $\text{color}[u] \leftarrow \text{white}; \quad \pi[u] = \text{NIL}$
- 3 $\text{time} \leftarrow 0$
- 4 **for** each vertex $u \in V$ **do**
- 5 **if** $\text{color}[u] = \text{white}$ **then** **DFS-Visit**(u)

DFS-Visit(u)

- 1 $\text{color}[u] \leftarrow \text{grey}; \quad \text{time} \leftarrow \text{time} + 1; \quad d[u] \leftarrow \text{time}$
- 2 **for** each vertex $v \in \text{Adj}[u]$ **do**
- 3 **if** $\text{color}[v] = \text{white}$
- 4 **then** $\pi[v] \leftarrow u; \quad \text{DFS-Visit}(v)$
- 5 $\text{color}[u] \leftarrow \text{black}$
- 6 $f[u] \leftarrow \text{time} \leftarrow \text{time} + 1$

DFS: Properties

Let T be the DFS tree of G by DFS algorithm. Let $[d[u], f[u]]$ be the **time interval** computed by DFS algorithm. Let $u \neq v$ be any two vertices of G .

- The intervals of $[d[u], f[u]]$ and $[d[v], f[v]]$ are either **disjoint** or **one is contained in another**.
- $[d[u], f[u]]$ is **contained in** $[d[v], f[v]]$ if and only if u is a descendent of v with respect to T .

Classification of Edges

Let $G = (V, E)$ be a directed graph and T a spanning tree of G . The edge $e = u \rightarrow v$ of G can be classified as:

- **tree edge** if $e = u \rightarrow v \in T$.
- **back-edge** if $e \notin T$ and v is an **ancestor** of u .
- **forward-edge** if $e \notin T$ and u is an **ancestor** of v .
- **cross-edge** if $e \notin T$, v and u are **unrelated** with respect to T .

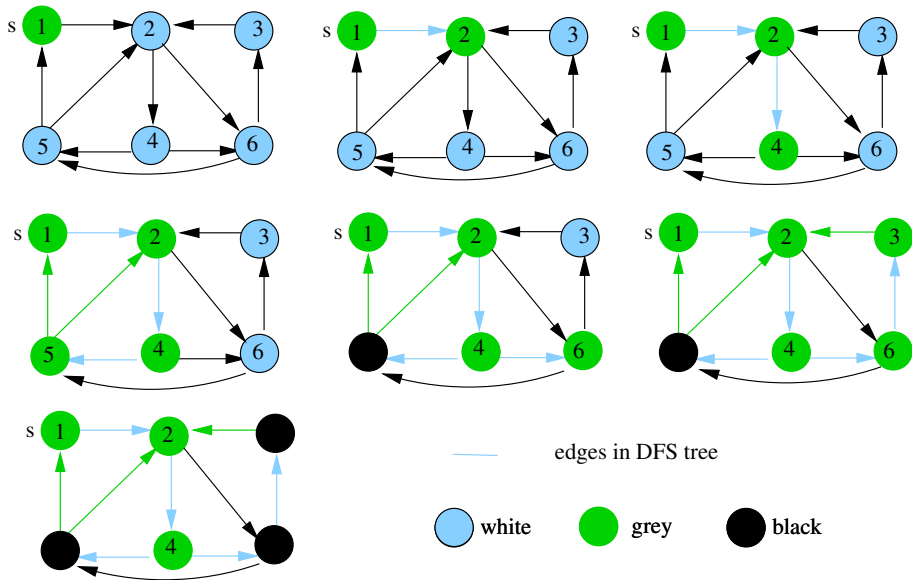
Classification of Edges

Let $G = (V, E)$ be a directed graph and T the spanning tree of G constructed by DFS algorithm. The classification of the edges can be done as follows.

- When $e = u \rightarrow v$ is first explored by DFS, color e by the $\text{color}[v]$.
- If $\text{color}[v]$ is white, then e is white and is a tree edge.
- If $\text{color}[v]$ is grey, then e is grey and is a back-edge.
- If $\text{color}[v]$ is black, then e is black and is either a forward- or a cross-edge.

For DFS tree of directed graphs, all four types of edges are possible.

DFS: Example



DFS: Applications

Definition

A directed graph $G = (V, E)$ is called a **directed acyclic graph (DAG for short)** if it contains no directed cycles.

DAG Testing

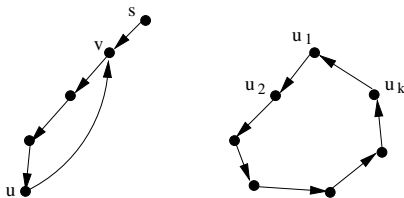
Given a directed graph $G = (V, E)$, test if G is a DAG or not.

Theorem

Let G be a directed graph, and T the DFS tree of G . Then G is DAG \iff there are no **back edges**.

DFS: Applications

Proof: \implies Suppose $e = u \rightarrow v$ is a back edge. Let P be the path in T from v to u . Then the directed path P followed by $e = u \rightarrow v$ is a directed cycle.



\Leftarrow Suppose $C = u_1 \rightarrow u_2 \rightarrow \dots \rightarrow u_k \rightarrow u_1$ is a directed cycle. Without loss of generality, assume u_1 is the first vertex visited by DFS. Then, the algorithm visits u_2, u_3, \dots, u_k , before it backtracks to u_1 . So $u_k \rightarrow u_1$ is a back edge.

DAG Testing in $\Theta(n + m)$ time

- 1 Run DFS on G . Mark the edges “white”, “grey” or “black”,
- 2 If there is a grey edge, report “ G is not a DAG”. If not “ G is a DAG”.

Topological Sort

Topological Sort

Let $G = (V, E)$ be a DAG. A **topological sort** of G assigns each vertex $v \in V$ a distinct number $L(v) \in [1..n]$ such that if $u \rightarrow v$ then $L(u) < L(v)$.

Note: If G is not a DAG, topological sort cannot exist.

Application

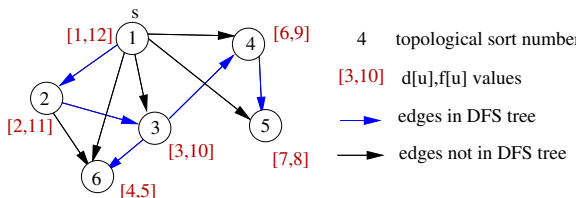
- The directed graph $G = (V, E)$ specifies a **job flow chart**.
- Each $v \in V$ is a job.
- If $u \rightarrow v$, then the job u must be done before the job v .
- A **topological sort** specifies the order to complete jobs.

Topological Sort

We can use DFS to find topological sort.

Topological-Sort-by-DFS(G)

- 1 Run DFS on G .
- 2 Number the vertices by decreasing order of $f[v]$ value. (This can be done as follows: During DFS, when a vertex v is finished, insert v in the front of a linked list.)



Clearly, this algorithm takes $\Theta(m + n)$ time.

Strong Connectivity

Definition

- A directed graph $G = (V, E)$ is **strongly connected** if for any two vertices u and v in V , there exists a directed path from u to v .
- A **strongly connected component of G** is a maximal subgraph of G that is strongly connected.

Strong Connectivity Problem

Given a directed graph G , find the strongly connected components of G .

Note: G is strongly connected if and only if it has exactly one strongly connected component.

Strong Connectivity

Application: Traffic Flow Map

- $G = (V, E)$ represents a street map.
- Each $v \in V$ is an intersection.
- Each edge $u \rightarrow v$ is a 1-way street from the intersection u to the intersection v .
- Can you reach from any intersection to any other intersection?
- This is so $\iff G$ is strongly connected.
- All intersections within each connected component can reach each other.

This problem can be solved by using DFS. Without it, it would be hard to solve efficiently.

Strong Connectivity

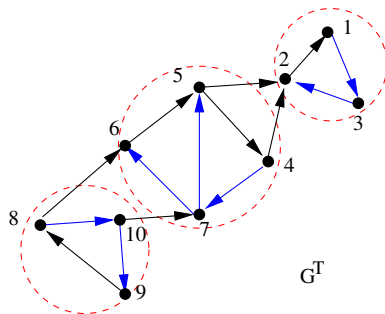
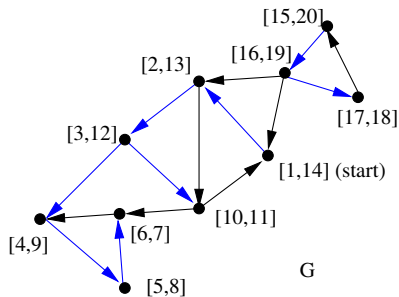
Strong-Connectivity-by-DFS(G)

- 1 Run DFS on G , compute $f[u]$ for all $u \in V$,
- 2 Order the vertices by decreasing $f[v]$ values.
- 3 Construct the **transpose graph G^T** , which is obtained from G by reversing the direction of all edges.
- 4 Run DFS on G^T , the vertices are considered in the order of decreasing $f[v]$ values.
- 5 The vertices in each tree in the DFS forest correspond to a strongly connected component of G .

Analysis:

- Steps 1 and 2: $\Theta(n + m)$ (step 2 is a part of step 1.)
- Step 3: $\Theta(n + m)$ (how?)
- Step 4 and 5: $\Theta(n + m)$ (step 5 is part of step 4.)

Strong Connectivity: Example



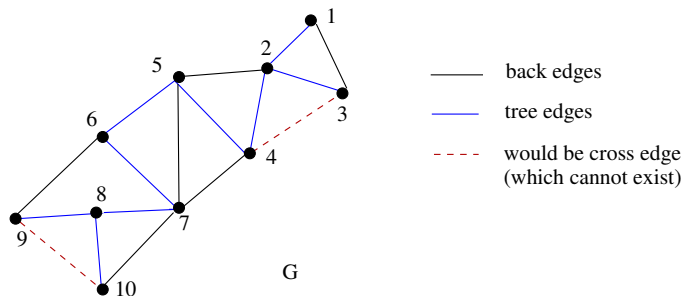
DFS for Undirected Graphs

- DFS algorithm can be used on an undirected graph $G = (V, E)$ without any change.
- It constructs a DFS tree T of G .
- Recall that: for an undirected graph $G = (V, E)$ and a spanning tree T of G , the edges of G can be classified as:
 - tree edges
 - back edges
 - cross edges

Theorem

Let G be an undirected graph, and T the DFS tree of G constructed by DFS algorithm. Then there are no cross edges.

DFS for Undirected Graphs



Summary: Edge Types

For Directed Graphs

	Tree	Forward	Backward	Cross
BFS	yes	no	yes	yes
DFS	yes	yes	yes	yes

For Undirected Graphs

	Tree	Back-edge	Cross
BFS	yes	no	yes
DFS	yes	yes	no

Biconnectivity Problem

Definition

Let $G = (V, E)$ be an **undirected connected graph**.

- A vertex $v \in V$ is a **cut vertex (also called articulation point)** if deleting v and its incident edges disconnects G .
- G is **biconnected** if it is connected and has no cut vertices.
- A **biconnected component** of G is a maximal subgraph of G that is biconnected.
- G is biconnected if and only if it has exactly one biconnected component.

Biconnectivity Problem

Given an undirected graph $G = (V, E)$, is G biconnected?

If not, find the cut vertices and the biconnected components of G .

Biconnectivity Problem

Application

- G represents a computer network.
- Each vertex is a computer site.
- Each edge is a communication link.
- If v is a cut vertex, then the failure of v will disconnect the whole network.
- The network can survive any single site failure if and only if G is biconnected.

Simple-Biconnectivity(G)

- 1 **for** each vertex $v \in V$ **do**
- 2 delete v and its incident edges from G
- 3 test if $G - \{v\}$ is connected

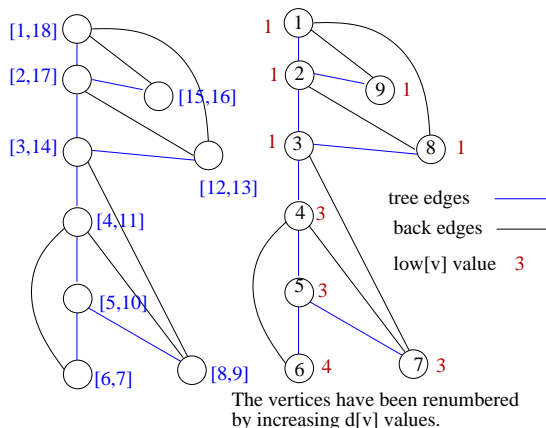
This algorithm takes $\Theta(n) \times \Theta(n + m) = \Theta(n(n + m))$ time.

Biconnectivity Problem

By using DFS, the problem can be solved in $O(n + m)$ time.

- Let T be the DFS tree of G .
- Re-number the vertices by increasing $d[v]$ values.
- For each vertex v , define:
low $[v]$ = the smallest vertex that can be reached from v or a descendent of v through a back edge.
- If v is a leaf of T , then $\text{low}[v] = \min \left\{ \overset{v}{\{w \mid (v, w) \text{ is a back-edge}\}} \right\}$
- If v is not a leaf of T , then $\text{low}[v] = \min \left\{ \overset{v}{\begin{array}{l} \{w \mid (v, w) \text{ is a back-edge}\} \\ \{\text{low}[t] \mid t \text{ is a son of } v\} \end{array}} \right\}$

Biconnectivity Problem



In this figure, low means closer to the root. So the root is the lowest vertex. $low[v]$ is the lowest vertex that can be reached from v or a descendent of v thru a single back edge.

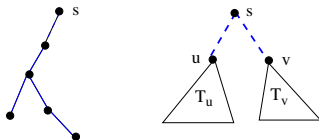
Biconnectivity Problem

Theorem

Let T be the DFS of $G = (V, E)$ rooted at the vertex s .

- 1 s is a cut vertex $\iff s$ has at least two sons in T .
- 2 A vertex $a \neq s$ is a cut vertex $\iff a$ has a son b such that $low[b] \geq a$.

Proof of (1): Suppose s has only one son. After deleting s , all other vertices are still connected by the remaining edges of T . So s is not a cut vertex.

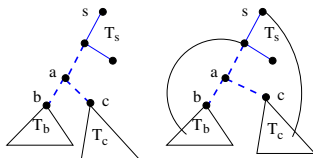


Suppose s has at least two sons u and v (there may be more). Let T_u be the subtree of T rooted at u and T_v be the subtree of T rooted at v . Because there are no cross edges, no edges connect T_u with T_v . So after s is deleted, T_u and T_v become disconnected. Hence s is a cut vertex.

Biconnectivity Problem

Proof of (2): Let T_s be the subtree of T above the vertex a . Let $b, c \dots$ be the sons of a . Let T_b, T_c, \dots be the subtree of T rooted at $b, c \dots$

- Suppose a has a son b with $\text{low}[b] \geq a$. Because $\text{low}[b] \geq a$, no vertex in T_b is connected to T_s . Because there are no cross edges, no edges connect vertices in T_b and T_c . So after a is deleted, T_b is disconnected from the rest of G . So a is a cut vertex.



- Suppose for every son b of a we have $\text{low}[b] < a$. This means that there is a back edge connecting a vertex in T_b to a vertex in T_s . So after a is deleted, all subtrees T_b, T_c, \dots are still connected to T_s , and G remains connected. So a is not a cut vertex.

Biconnectivity Problem

We can now describe the algorithm. For conceptual clarity, the algorithm is divided into several steps. Actually, all steps can be and should be incorporated into a single DFS run.

Biconnectivity-by-DFS(G)

- 1 Run DFS on G
- 2 Renumber the vertices by increasing $d[*]$ values.
- 3 For all $u \in V$, compute $\text{low}[u]$ as described before.
- 4 Identify the cut vertices according to the conditions in the theorem.

Analysis

- Steps 1 and 2: takes $\Theta(m + n)$ time.
- Step 3: $\text{low}[u]$ is the minimum of k values:
 - the $\text{low}[*]$ values for all sons of u .
 - the values for each back-edge from u .
 - 1 for u itself, we charge this to the edge between u and its parent.
 - So $k = \text{deg}(u)$.
 - We compute $\text{low}[*]$ in post order. When computing $\text{low}[u]$, all values needed have been computed already. So it takes $\Theta(\text{deg}(u))$ time to compute $\text{low}[u]$.
 - So the total time needed to compute $\text{low}[u]$ for all vertices is Θ of the number of edges of G . This is $\Theta(m)$.
- Step 4: The total time needed for checking these conditions for all vertices is $\Theta(n)$.

The Biconnectivity problem can be solved in $\Theta(n + m)$ time

- The DFS based Biconnectivity algorithm was discovered by Tarjan and Hopcroft in 1972. (See Problem 22-2, Page 558).
- They advocated the use of adjacent list representation over the adjacent matrix representation for solving complex graph problems in linear (i.e. $O(n + m)$) time.
- This DFS algorithm is a good example. Without using adjacent list representation, the problem would take at least $\Theta(n^2)$ time to solve.