CSE 101
Introduction to Data Structures and Algorithms
Graph Algorithms

Most of the algorithms we cover operate on the *adjacency list* representation of a graph, which was described in the preceding handout. A useful side effect of this choice is that the algorithms, as expressed in pseudo-code, operate correctly on both directed and undirected graphs.

Given a graph G (directed or undirected) and two vertices $x, y \in V(G)$, we define the *distance* $\delta(x, y)$ from *x* to *y*, as follows.

$$\delta(x, y) = \begin{cases} \text{minimum length of an } x\text{-}y \text{ path} & \text{if } y \text{ is reachable from } x \\ \infty & \text{if } G \text{ contains no } x\text{-}y \text{ path} \end{cases}$$

Note that when we speak of an *x*-*y* path in the directed case, we always mean a *directed x*-*y* path. Similarly, the reachability relation is slightly different for directed graphs. To say that *y* is *reachable* from *x* in the directed case, means that G contains a *directed x*-*y* path.

With this definition we can now state the *Single Source Shortest Paths* (SSSP) problem:

> Given a graph *G* (directed or undirected), and a fixed $s \in V(G)$, called the *source vertex*: (1) determine $\delta(s, x)$ for all $x \in V(G)$, and (2) determine a shortest *s*-*x* path in *G* for all those x that are reachable from *s*.

Observe that if *G* contains any *s*-*x* path, there must exist one of minimum length. In general though, there may be more than one shortest *s*-*x* path in *G*, and in fact the number can be quite large. So whereas a solution to part (1) of SSSP is necessarily unique, a solution to part (2) may not be.

Breadth First Search
The Breadth First Search (BFS) algorithm is a very efficient solution to SSSP. It takes as input, the adjacency list representation of a graph. Its output is stored in data structures that encode solutions to parts (1) and (2) of SSSP. Among these data are three attributes for each vertex $x \in V(G)$.

- $\text{color}[x]$: (white, gray, black) keeps track of the progress of BFS, interpreted as follows:
    - $\text{color}[x] = \text{white}$: *x* is undiscovered,
    - $\text{color}[x] = \text{gray}$: *x* is discovered, but some neighbors of *x* may not be discovered,
    - $\text{color}[x] = \text{black}$: *x* is finished, i.e. x and all its neighbors are discovered.
- $\text{distance}[x]$: when BFS is complete, stores the distance $\delta(s, x)$.
- $\text{parent}[x]$: stores the predecessor of *x* along a shortest *s*-*x* path, if it exists, `nil` otherwise.

If we use positive integers to label our vertices, then we may regard $V(G) = \{1, 2, 3, \ldots, n\}$, where $n = |V(G)|$. Then the above attributes are naturally implemented as arrays, as the notation would suggest. The BFS algorithm also uses a FIFO queue Q to manage vertices during execution.

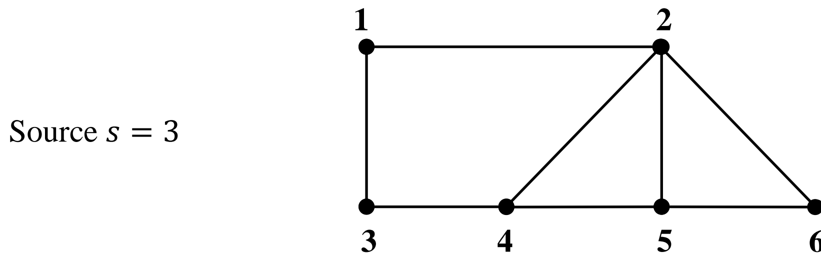
BFS systematically discovers every vertex reachable from the source vertex *s*. For every positive integer k, it discovers all vertices at distance k from *s*, before it discovers any vertices at distance $k + 1$ from *s*, hence the name *Breadth-First Search*.

<u>BFS(G, s)</u>

1. for $x \in$ V(G) $-$ {s}
2.   color[x] = white
3. $d[x] = \infty$
4. $p[x]$ = nil
5. color[s] = gray   // discover the source s
6. $d[s] = 0$
7. $p[s]$ = nil
8. $Q = \emptyset$ // construct a new empty queue
9. Enqueue(Q, s)
10. while $Q \neq \emptyset$
11. x = Dequeue(Q)
12.   for $y \in$ adj[x]
13.   if color[y] == white   // y is undiscovered
14.   color[y] = gray   // discover y
15. $d[y] = d[x] + 1$
16. $p[y] = x$
17.   Enqueue(Q, y)
18.   color[x] = black   // finish x

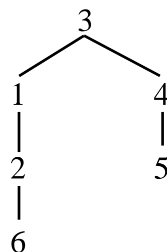
Lines 1-7 initialize all vertex attributes, while lines 8-9 initialize the FIFO queue Q. Then while loop (lines 10-18) systematically explores all vertices reachable from the source s.

Example 1



		color[ ]	d[ ]	p[ ]
1:	2 3	~~w~~ ~~g~~ b	~~∞~~ 1	~~nil~~ 3
2:	1 4 5 6	~~w~~ ~~g~~ b	~~∞~~ 2	~~nil~~ 1
3:	1 4	~~g~~ b	0	nil
4:	2 3 5	~~w~~ ~~g~~ b	~~∞~~ 1	~~nil~~ 3
5:	2 4 6	~~w~~ ~~g~~ b	~~∞~~ 2	~~nil~~ 4
6:	2 5	~~w~~ ~~g~~ b	~~∞~~ 3	~~nil~~ 2

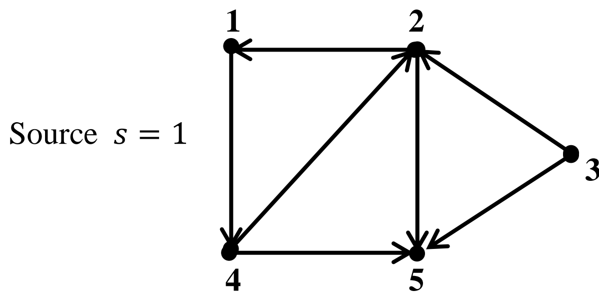
Q: 3 1 4 2 5 6   BFS Tree:



An inspection of the pseudo-code shows that all vertices reachable from the source are discovered (grayed), enqueued, their undiscovered neighbors are discovered, then finished (blackened) and dequeued. The test on line 13 guarantees that such vertices enter the queue exactly once, since no vertex is ever whitened. Vertices not reachable from the source are never discovered and remain white when the algorithm halts. At all times, the queue contains the set of gray vertices.

The following example shows BFS operating on a directed graph.

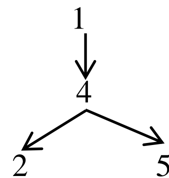
Example 2



		color[ ]	d[ ]	p[ ]
1:	4	~~g~~ b	0	nil
2:	1 5	~~w~~ ~~g~~ b	~~∞~~ 2	~~nil~~ 4
3:	2 5	w	~~∞~~	nil
4:	2 5	~~w~~ ~~g~~ b	~~∞~~ 1	~~nil~~ 1
5:		~~w~~ ~~g~~ b	~~∞~~ 2	~~nil~~ 4

Q: 1 4 2 5

BFS Tree:



Once BFS is complete, the *Predecessor Subgraph* (or *BFS tree*) is defined to be $T = (V_p, E_p)$, where

$$V_p = \{\, x \in V(G) \mid p[x] \neq \text{nil} \,\} \cup \{s\}$$

and

$$E_p = \{\, (p[x], x) \mid p[x] \neq \text{nil} \,\}.$$

The edge $(p[x], x)$ is understood to be an *ordered pair* in the directed case, and an *unordered pair* $\{p[x], x\}$ in the undirected case. If the subgraph (V_p, E_p) contained a cycle, then there was a "cycle" of discovery events during the execution of BFS. But this is impossible since each vertex is discovered at most once, as noted above, and hence (V_p, E_p) is acyclic. We see directly from the definition that $|E_p| = |V_p| - 1$. Therefore by Lemma 6 in the Graph Theory handout, (V_p, E_p) is a tree.

Note that the BFS tree can depend on the order in which adjacency lists were processed, a detail that is not specified in the BFS pseudo-code. We adopt the convention to process adjacency lists in increasing numerical order by vertex label, unless instructed otherwise.

Exercise 1
Alter the order of the adjacency lists (either by altering the labels or by altering the convention) in the last two examples, then re-run BFS. Also re-run BFS with different sources.

Exercise 2
Make up other examples (both directed and undirected) and run BFS on them.

After BFS is complete, the following recursive algorithm prints a shortest s-x path, provided x is reachable from s.

PrintPath(G, s, x) pre: BFS(G, s) has been run
1. if $x == s$
2. print(s)
3. else if p$[x]$ == nil
4. print$(x$, " is not reachable from ", $s)$
5. else
6. PrintPath$(G, s, \text{p}[x])$
7. print(x)

See Theorem 22.5 and preceding lemmas for a proof that BFS, together with PrintPath, correctly solves the SSSP problem.

To analyze the runtime of BFS, let $n = |V(G)|$ and $m = |E(G)|$. Initialization (lines 1-9) incurs cost $\Theta(n)$. Queue operations also have cost $\Theta(n)$ (in worst case), since each vertex enters the queue at most once. Scanning adjacency lists has cost equal to the total length of all adjacency lists, which is the sum of the vertex degrees. By the Handshake Lemma this is

$$\sum_{x \in V(G)} \deg(x) = \begin{Bmatrix} m & \text{directed case} \\ 2m & \text{undirected case} \end{Bmatrix} = \Theta(m)$$

Thus the runtime of BFS is $\Theta(n + m)$. This is linear in the number of bytes needed to store the adjacency list representation.

Depth First Search
Depth First Search (DFS) is often used as a subroutine in other algorithms that determine structural details of a graph. It does this by searching deeper into a graph whenever possible, a strategy that is in some sense "orthogonal" to that of BFS. Like BFS, it requires that vertices have certain attributes. For each $x \in V(G)$:

- color$[x]$: (white, gray, black) keeps track of the progress of DFS, with interpretation similar to that of BFS.
- p$[x]$: parent of x in the predecessor subgraph.
- d$[x]$: discover time of x.
- f$[x]$: finish time of x.

The time stamps $d[x]$ and $f[x]$ can be used by other algorithms to deduce aspects of the structure of G. DFS(G) itself calls the recursive subroutine Visit(x), which discovers and finishes x, along with all as yet

undiscovered vertices reachable from x. In the following pseudo-code, time is a variable ranging from 1 to $2n$, where $n = |V(G)|$. It is local to DFS(), but static over all recusive calls to Visit().

```
DFS(G)
1. for all x ∈ V(G)
2.     color[x] = white
3.     p[x] = nil
4. time = 0
5. for all x ∈ V(G)
6.     if color[x] == white
7.         Visit(x)

Visit(x)
1. d[x] = (+ + time)        // discover x
2. color[x] = gray
3. for all y ∈ adj[x]
4.     if color[y] == white
5.         p[y] = x
6.         Visit(y)
7. color[x] = black
8. f[x] = (+ + time)        // finish x
```

The *predecessor subgraph* $G_p = (V_p, E_p)$ is now defined as

$$V_p = V(G)$$

and

$$E_p = \{\ (p[x], x) \mid p[x] \neq \text{nil}\ \}.$$

As before the pair $(p[x], x)$ denotes an ordered pair in the case that G is a directed graph, and an unordered pair if G is undirected. Again observe that since no vertex is ever whitened, and only white vertices are discovered, there can be no "cycle" of discoveries, and hence G_p is acyclic. However V_p is the entire vertex set of G, so unlike in BFS, the predecessor subgraph may not be a tree. Thus G_p is also called a *DFS forest*. Just as for BFS, the predecessor subgraph depends on the order in which vertices are processed. Our convention for DFS is (unless otherwise stated) to execute loop 5-7 in DFS() and loop 3-6 in Visit() in increasing order by vertex label.

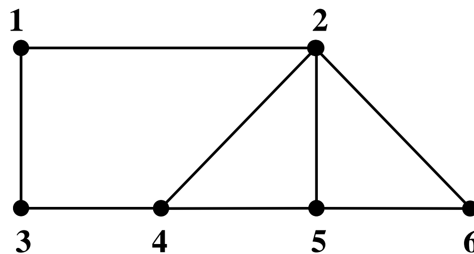
Notice every vertex in G is visited exactly once, but there are two ways for this to happen. A vertex can be visited from line 7 of DFS(), or from line 6 of Visit() itself. Only in the second case (line 6 of Visit()) does the vertex receive a predecessor. In the first case (line 7 of DFS()), the vertex has nil parent, and is therefore the root of a tree in the DFS forest. Hence the number of such trees is the number of vertices of the first kind.

One other comparison between BFS() and DFS() is worthwhile at this point. Notice that in addition to being recursive, DFS() has no "source" vertex. We could supply a source s to DFS(), then start by visiting that vertex, replacing the main loop 5-7 with a single call to Visit(s). The resulting algorithm, like BFS(), would discover only those vertices reachable from s. Likewise, BFS() could be re-written without a source, looping over all vertices in the graph, and discovering all of them. It seems both algorithms have two

versions: the first version having a source and discovering only those vertices reachable from it, and the second version with no source and discovering all vertices in the graph. Our text (CLRS) has chosen to present version 1 of BFS() and version 2 of DFS(). As an exercise, the reader should write the "other" version of each of these two algorithms.

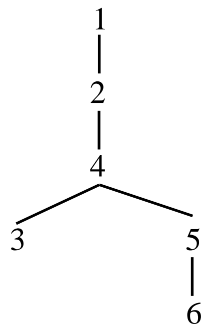
Finally, observe that there are many ways to search a graph or digraph. What distinguishes all graph search algorithms is the policy by which the algorithm decides, at each vertex, whether to "go broad" (visit another vertex on the current adjacency list), or to "go deep" (visit an undiscovered neighbor of a vertex on the current adjacency list). BFS() and DFS() lie at the two extremes of these policies. In BFS() we always broaden the search before going deeper, and in DFS() the priority is reversed. Many other policies are possible. For instance, the decision (broad or deep) can be made to depend on what "information" the algorithm finds at a given vertex.

Example 3



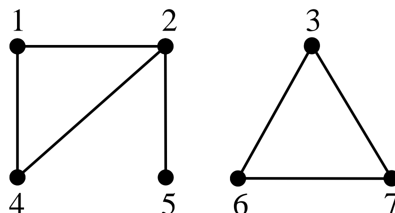
		color[ ]	$d[\,]$	$f[\,]$	$p[\,]$
1:	2 3	~~w~~ ~~g~~ b	1	12	nil
2:	1 4 5 6	~~w~~ ~~g~~ b	2	11	1
3:	1 4	~~w~~ ~~g~~ b	4	5	*4*
4:	2 3 5	~~w~~ ~~g~~ b	3	10	2
5:	2 4 6	~~w~~ ~~g~~ b	6	9	4
6:	2 5	~~w~~ ~~g~~ b	7	8	5

DFS Forest:



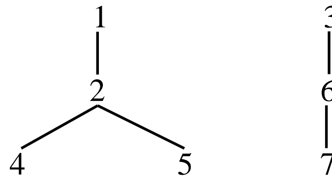
When DFS is run on an undirected graph G, the DFS forest will have as many trees as G has connected components, where each tree spans one component.

Example 4



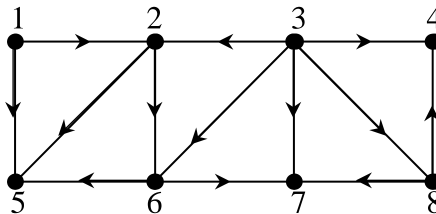
		$d[\,]$	$f[\,]$	$p[\,]$
1:	2 4	1	8	nil
2:	1 4 5	2	7	1
3:	6 7	9	14	nil
4:	1 2	3	4	2
5:	2	5	6	2
6:	3 7	10	13	3
7:	3 6	11	12	6

DFS Forest:



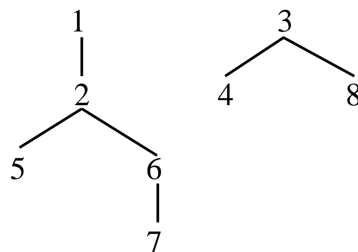
One might hope that when run on a digraph, DFS would find the strongly connected components. Unfortunately this is not the case. We'll need a more complicated algorithm to find strong components, one involving two separate calls to DFS (on different digraphs).

Example 5



	adj[ ]	$d[\,]$	$f[\,]$	$p[\,]$
1:	2 5	1	10	nil
2:	5 6	2	9	1
3:	2 4 6 7 8	11	16	nil
4:		12	13	3
5:		3	4	2
6:	5 7	5	8	2
7:		6	7	6
8:	4 7	14	15	3

DFS Forest:



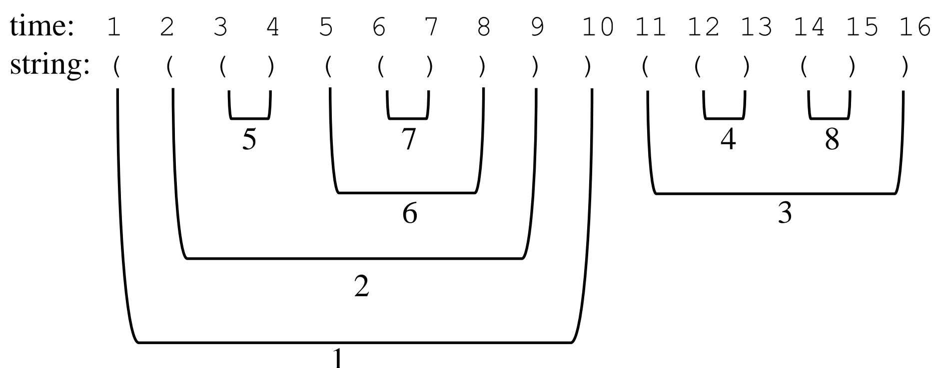
Both BFS and DFS are used to determine structural aspects of a graph or digraph. We know that BFS solves the SSSP problem, but what problem does DFS solve? The answer is that it solves *many problems*. Before we get to those, let us analyze the run time of DFS. As before let $n = |V(G)|$ and $m = |E(G)|$.

Initialization (loop 1-3 of DFS) has cost $\Theta(n)$, as does the main loop (5-7 of DFS), disregarding the call to Visit. As previously noted, each vertex is visited exactly once, so each adjacency list is scanned (loop 3-6 of Visit) once. All other operations in Visit have constant cost, so the cost of visiting all vertices is the sum of the lengths of the adjacency lists, which is

$$\sum_{x \in V(G)} \deg(x) = \begin{Bmatrix} m & \text{directed case} \\ 2m & \text{undirected case} \end{Bmatrix} = \Theta(m)$$

The total cost of DFS is therefore $\Theta(n+m)$, again linear in the number of bytes need to store the adjacency list representation of G.

What structural information about G can be deduced from the fields $d[]$, $f[]$ and $p[]$ after a call to DFS? First note that the time stamps and the DFS-forest store the *same* information in different form. To see this, create a parenthesis string, with "(" for each discover time, and ")" for each finish time. Illustrating on the last example, we have



Observe that this string is not only *well formed* (in the sense that the parentheses match up), but also matching pairs of parentheses correspond to the discover and finish times of one and the same vertex. This string can therefore be thought of as the precessor subgraph G_p in other notation. This is the contentent of the following Theorem

Theorem (Parenthesis)
Suppose $x, y \in V(G)$ and $d[x] < d[y]$. Then exactly one of the following hold.

(1) $d[x] < f[x] < d[y] < f[y]$
( ) ( )

or

(2) $d[x] < d[y] < f[y] < f[x]$
(( ))

Proof: See CLRS Chapter 22, page 606.

Item (2) is equivalent to saying: y is discovered while x is gray, and also to the statement: y is a descendant of x. Therefore item (1) is holds if and only if y is *not* a descendant of x, so either x and y lie in different trees in the DFS forest, or they are cousins in the same tree.

Theorem (White Path)
Let $x, y \in V(G)$. Then y is a descendant of x (so (2) holds) if and only if at time $d[x]$, G contains an x-y path consisting entirely of white vertices.

Proof: See CLRS Chapter 22, page 608.

Think of DFS as being a movie, a sequence of still pictures, one for each time stamp. Then the White Path Theorem says that certain frames in this movie (i.e. the discover times) predict the structure the DFS forest when the movie is over.

DFS can also be used to classify the edges of G as follows

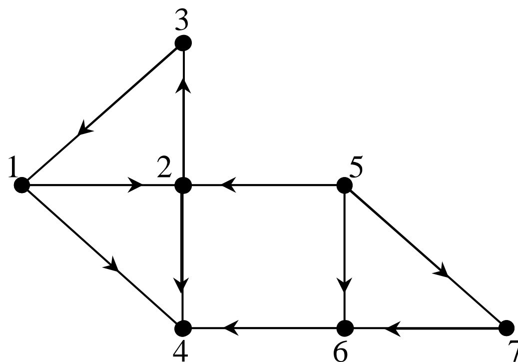
- *Tree Edges*: belong to the DFS forest G_p
- *Back Edges*: join a vertex to an ancestor in G_p
- *Forward Edges*: join a vertex to a descendant (other than a child) in G_p
- *Cross Edges*: join vertices in different trees, or join cousins in the same tree in G_p

Note that the distinction between back and forward edges only exists for directed graphs. In an undirected graph, there are therefore only 3 catagories: tree, back and cross.

Again following DFS in the last example we have

Tree: $(1,2), (2,5), (2,6), (6,7), (3,4), (3,8)$
Back:
Forward: $(1,5)$
Cross: $(3,2), (3,6), (3,7), (8,7),$ $(6,5), (8,4)$
tree to tree (for $(3,2), (3,6), (3,7), (8,7)$) cousin-to-cousin (for $(6,5), (8,4)$)

Example 6



Following our standard conventions (main loop of DFS and adjacency lists in Visit are processced in numerical order), we obtain:

Tree: (1, 2), (2, 3), (2, 4), 5, 6), (5, 7)
Back: (3, 1)
Forward: (1, 4)
Cross: (5, 2), (6, 4), (7, 6)

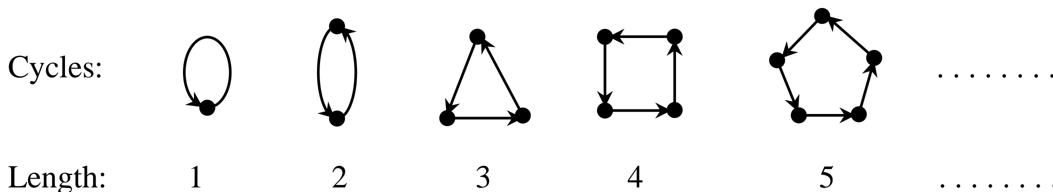
Exercise 3
Run DFS on all previous examples in this handout, and classify edges. Alter the labels (or change our convention as to loop order), and see how these changes alter the predecessor subgraph G_p and the classification of edges.

Exercise 4
Modify the DFS algorithm so that it prints out each edge together with its classification. Do this separately for directed graphs, then for undirected graphs. See pages 609-610 and problem 22.3-10 in CLRS for hints on how to do this.

Exercise 3 above would seem to indicate that DFS is not finding structual information that is intrinsinc to the graph itself, but rather to its labeling, and to the order in which we process those labels. In Fact, this is not the case.

Exercise 5
Modify the DFS algorithm by so that when run it on an undirected graph, it determines the reachability relation. In other words, after the modified DFS is run, one can tell exactly which vertices are reachable from which other vertices, as well as how many connected components the graph contains. See problem 22.3-12 for hints on how to do this.

Topological Sort
From now on we assume that $G = (V, E)$ is a digraph. We say that G is *acyclic* if and only if G contains no directed cycles.



Example 7

Such a graph is sometimes called a DAG (*Directed Acyclic Graph*). We can tell if digraph is a DAG by running DFS and classifying the edges.

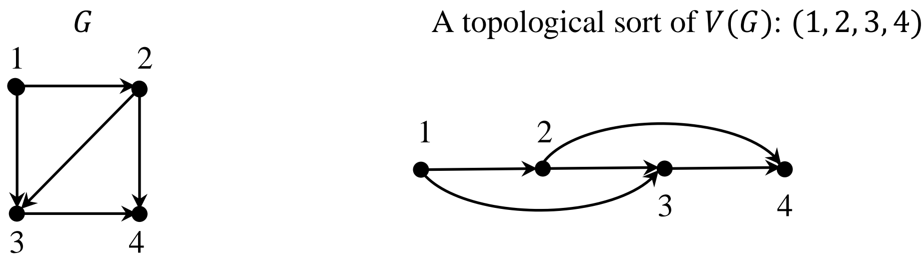
Lemma
A digraph G is acyclic if and only if DFS(G) yields no back edges. Equivalently, G contains a back edge if and only if it contains a directed cycle.

Proof:
A path consisting of tree edges from an ancestor to a descendant, followed by a back edge up to the ancestor, constitutes a directed cycle in G. This shows that the existence of a back edge implies the existence of a directed cycle.

Now assume G contains a directed cycle, call it C. Let y be the first vertex on C to be discovered during the execution of $\text{DFS}(G)$, and let x be the vertex preceding y along C. We claim that (x, y) is the desired back edge. Indeed, at time $d[y]$ the vertices of C form a white y-x path in G. By the white path theorem, x will be a descendent of y when $\text{DFS}(G)$ is complete. Therefore (x, y) is a back edge, as claimed. We've shown that the existence of a directed cycle implies the existence of a back edge, completing the proof. ■

Definition
Let $G = (V, E)$ be a DAG. A *topological sort* of the vertex set V is a linear ordering of the vertices such that if $(x, y) \in E$, then x appears before y in the ordering. In other words, it is a way of arranging the vertices along a line so that all directed edges go in the same direction.

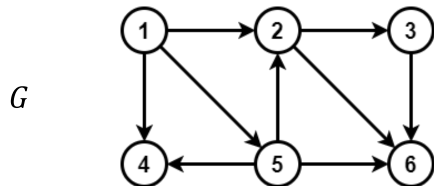
Example 8



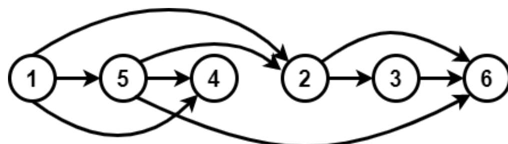
Notice that a digraph that contains a directed cycle cannot have a topological sort, since no matter how its vertices are arranged on a line, some edge along the cycle must go the wrong direction. Conversely, every DAG admits at least one topological sort, and possibly more than one.

There is an efficent algorithm to produce a topological sort of any DAG. Just execute $\text{DFS}(G)$, and as vertices finish, push them onto a stack S. When the algorithm is complete, the sequence S (from top to bottom) gives a topological sort of $V(G)$. Equivalently, run $\text{DFS}(G)$, and then sort $V(G)$ by decreasing finish times.

Example 9



Topological sort of $V(G)$: $(1, 5, 4, 2, 3, 6)$



Exercise 6
This DAG has 3 more topological sorts. Find them.

Exercise 7
How can you sort an array using topological sort? Write the algorithm.

Strongly Connected Components
Let $G = (V, E)$ be a digraph. We say $y \in V$ is *reachable* from $x \in V$ if and only if G contains a directed x-y path. We say G is *strongly connected* if and only if every vertex is reachable from every other vertex. In other words, G contains both a directed x-y path and a directed y-x path for all $x, y \in V$.

Similarly, a subset $U \subseteq V$ is said to be *strongly connected* if and only if every vertex in U is reachable from every other vertex in U. We call $U \subseteq V$ a *strongly connected component* (SCC) of G if and only if U is both

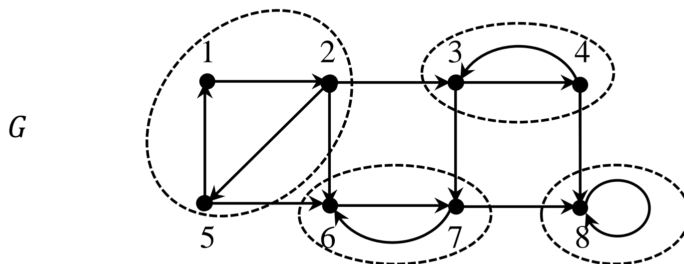
(1) strongly connected, and

(2) maximal with respect to (1)

Part (2) of the definition means that whenever there is a set W with $U \subsetneq W \subseteq V$, then W is not strongly connected. Equivalently, no vertex can be added to U without losing the strong connectivity property. Obviously G is itself strongly connected if and only if it contains exactly one strongly connected component, namely $V = V(G)$.

Example 10

G



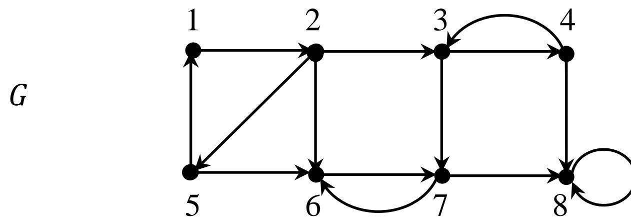
Strongly connected components: $\{1, 2, 5\}, \{3, 4\}, \{6, 7\}, \{8\}$

The DFS algorithm provides an efficient technique for finding the strongly connected components of a digraph G.

- Run $\text{DFS}(G)$, and as vertices finish, push them onto a stack S.
- Compute the *transpose* G^T of G, i.e. reverse all directed edges.
- Run $\text{DFS}(G^T)$, processing vertices in the main loop of DFS by decreasing finish times from the first call, i.e. pop vertices off the stack S.

When the second call to DFS is complete, the vertices in each tree of the resulting DFS forest constitute the strongly connected components of G. (See Theorem 22.16 on p. 619 of CLRS for a proof of this fact.)

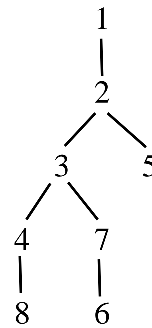
Example 11
Using the previous example



we run DFS on G, pushing vertices onto a stack as we finish. The main loop of DFS is executed in default order, namely ascending numerical order by vertex label.

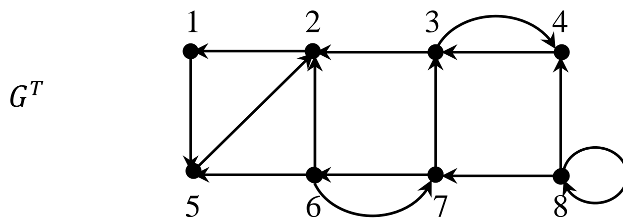
	adj[ ]	$d[\,]$	$f[\,]$	$p[\,]$
1:	2	1	16	nil
2:	3 5 6	2	15	1
3:	4 7	3	12	2
4:	3 8	4	7	3
5:	1 6	13	14	2
6:	7	9	10	7
7:	6 8	8	11	3
8:	8	5	6	4

DFS Forest



Stack
1
2
5
3
7
6
4
8

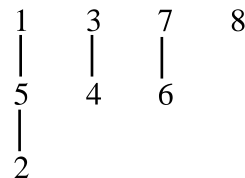
Now we compute the transpose of G,



then run DFS on G^T, this time executing the main loop of DFS in the top to bottom stack order the previous run. For later use, we also push vertices onto a stack in this second call.

	adj[ ]	$d[\,]$	$f[\,]$	$p[\,]$
1:	5	1	6	nil
2:	1	3	4	5
5:	2	2	5	1
3:	2 4	7	10	nil
7:	3 6	11	14	nil
6:	2 5 7	12	13	7
4:	3	8	9	3
8:	4 7 8	15	16	nil

DFS Forest



Stack
8
7
6
3
4
1
5
2

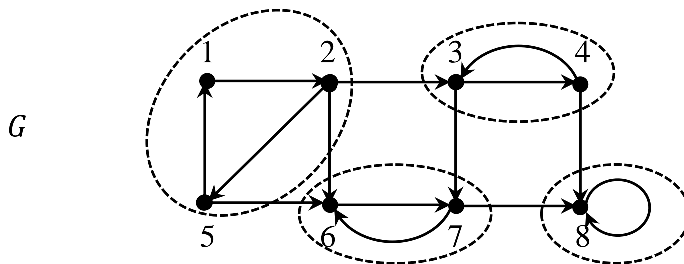
The strongly connected components of G are now the vertices in the trees of the DFS forest.

$$C_1 = \{1, 2, 5\}$$
$$C_2 = \{3, 4\}$$
$$C_3 = \{6, 7\}$$
$$C_4 = \{8\}$$

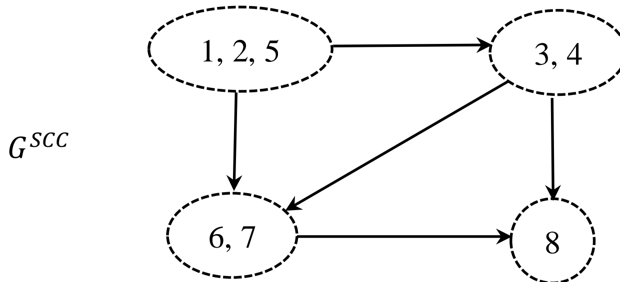
Notice that the strongly connected components of G and G^T are one and the same. Observe also that we can use the second stack to easily identify the strong components of G. Just read the stack from bottom to top. Notice that the boundary between two strong components is marked by a vertex with nil parent, which is printed above in bold.

Definition
The *component graph* of a digraph G (also called the *condensation graph*), denoted G^{SCC}, is the digraph whose vertices are the strongly connected components of G, and which has a directed edge from component C_i to component C_j if and only if there exist $x \in C_i$ and $y \in C_j$ such that $(x, y) \in E(G)$.

Continuing the previous example, we have



and its component graph



Observe that G^{SCC} is necessarily acyclic. Indeed, if G^{SCC} contained a cycle, then G would contain a cycle out of, then back into one of its strong components. This would violate the maximality condition for a strongly connected component. Since G^{SCC} is a DAG, we can determine a topological sort of its vertices (which you'll recall are the strong components of G.) But notice that the procedure outlined above has already done that, and this is why we constructed the second stack. As we read vertices in G from bottom to top, we separate strong components by noting which vertices have nil parents, and then automatically obtain a topological sort of G^{SCC}. In fact the second call to DFS was, while finding the strong components of G, also performing a topological sort of G^{SCC}.