CITS2200 Data Structures and Algorithms

Topic 12

Tree and Graph Traversals

- Tree traversals
- Bredth First Search
- Depth First Search
- Topological Sort

Reading: Weiss Section 18.4

1. Tree Traversals

Why traverse?

- search for a particular item
- test equality (isomorphism)
- copy
- create
- display

We'll consider two of the simplest and most common techniques:

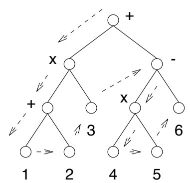
depth-first — follow branches from root to leaves

breadth-first (level-order) — visit nodes level by level

1.1 Depth-first Traversal

Preorder Traversal: Common garden "left to right", "backtracking", depth-first search!

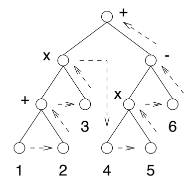
```
if(!t.isEmpty()) {
   visit root of t;
   perform preorder traversal of left subtree;
   perform preorder traversal of right subtree;
}
```



(Generates a *prefix expression* $+ \times + 1\ 2\ 3 - \times\ 4\ 5\ 6$

Postorder Traversal

```
if(!t.isEmpty()) {
   perform postorder traversal of left subtree;
   perform postorder traversal of right subtree;
   visit root of t;
}
```



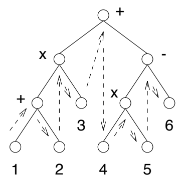
(Generates a *postfix expression*

$$1\ 2 + 3 \times 4\ 5 \times 6 - +$$

Also non-ambiguous — as used by, eg. HP calculators.)

Inorder Traversal

```
if(!t.isEmpty()) {
   perform inorder traversal of left subtree;
   visit root of t;
   perform inorder traversal of right subtree;
}
```



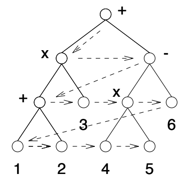
(Generates an *infix expression*

$$1 + 2 \times 3 + 4 \times 5 - 6$$

Common, easy to read, but ambiguous.)

1.2 Level-order (Breadth-first) Traversal

Starting at root, visit nodes level by level (left to right):



Doesn't suit recursive approach. Have to jump from subtree to subtree.

- need to keep track of subtrees yet to be visited — ie need a data structure to hold (windows to) subtrees (or Orchard)
- each internal node visited spawns two new subtrees
- new subtrees visited *only after* those already waiting

$\Rightarrow$ Queue of (windows to) subtrees!

Algorithm

```
place tree (root window) in empty queue q;
while (!q.isEmpty()) {
   dequeue first item;
   if (!external node) {
      visit its root node;
      enqueue left subtree (root window);
      enqueue right subtree (root window);
   }
}
```


1.3 Traversal Analysis

Time

The traversals we have outlined all take $O(n)$ time for a binary tree of size n.

Since all n nodes must be visited, we require $O(n)$ time
$\Rightarrow$ asymptotic performance cannot be improved.

Space

Depth-first: Recursive implementation requires memory (from Java's local variable stack) for each method call $\Rightarrow$ proportional to height of tree

- worst case: skinny, size n implies height n
- expected case: much better (depends on distribution considered — see Wood Section 5.3.3)
- best case: *exercise*...

Iterative implementation is also possible.

Level-order: Require memory for queue.

Depends on tree *width* — maximum number of nodes on a single level.

Maximum length of queue is bounded by twice the width.

- best case: skinny, width 2
- worst case: *exercise*...

Breadth-first graph search

A Bredth-first search in a graph is a little more complicated than a level-order traversal of a tree, because we must make sure that we do not visit the same node twice.

Breadth-first search is a simple but extremely important technique for searching a graph. This search technique starts from a given vertex v and constructs a spanning tree for G, called the *breadth-first tree*. It uses a (first-in, first-out) *queue* as its main data structure.

Following CLRS (section 22.2), as the search progresses, we will divide the vertices of the graph into three categories, *black* vertices which are the vertices that have been fully examined and incorporated into the tree, *grey* vertices which are the vertices that have been seen (because they are adjacent to a tree vertex) and placed on the queue, and *white* vertices, which have not yet been examined.

Queues

Recall that a *queue* is a first-in-first-out buffer.

Items are *pushed* (or enqueued) onto the end of the queue, and items can be *popped* (or dequeued) from the front of the queue.

A Queue is commonly implemented using either a block representation, or a linked representation.

We will assume that the push and pop operations can be performed in constant time. You may also assume that we can examine the first element of the queue, and decide if the queue is empty, all in constant time (i.e. $O(1)$).

Breadth-first search initialization

The final breadth-first tree will be stored as an array called π where $\pi[x]$ is the immediate parent of x in the spanning tree. Of course, as v is the root of this tree, $\pi[v]$ will remain undefined (or **nil** in CLRS).

To initialize the search we mark the colour of every vertex as *white* and the queue is empty. Then the first step is to mark the colour of v to be *grey*, put $\pi[v]$ to be undefined.

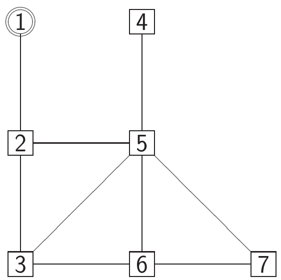
Breadth-first search repetitive step

Then the following procedure is repeated until the queue, Q, is empty.

procedure BFS(v)
  Push v on to the tail of Q
  while Q is not empty
    Pop vertex w from the head of Q
    for each vertex x adjacent to w **do**
      if *colour*[x] is *white* **then**
        $\pi[x] \leftarrow w$
        colour[x] $\leftarrow$ *grey*
        Push x on to the tail of Q
      end if
    end for
    colour[w] $\leftarrow$ *black*
  end while

At the end of the search, every vertex in the graph will have colour *black* and the parent or predecessor array π will contain the details of the breadth-first search tree.

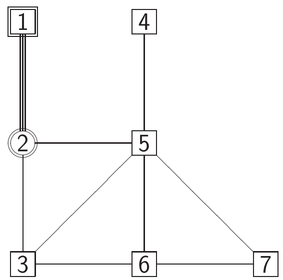
Example of breadth-first search



Head
↓
queue 1

x	*colour*[x]	$\pi[x]$
1	*grey*	**undef**
2	*white*	
3	*white*	
4	*white*	
5	*white*	
6	*white*	
7	*white*	

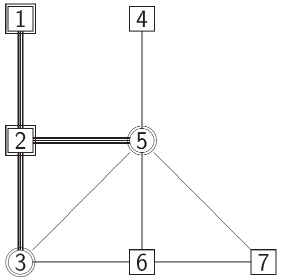
After visiting vertex 1



Head
↓
queue 1 2

x	*colour*[x]	$\pi[x]$
1	*black*	**undef**
2	*grey*	1
3	*white*	
4	*white*	
5	*white*	
6	*white*	
7	*white*	

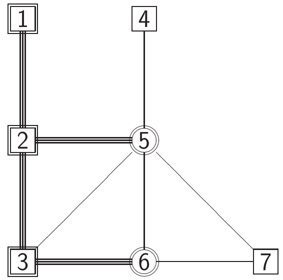
After visiting vertex 2



Head
↓
queue 1 2 3 5

x	*colour*[x]	$\pi[x]$
1	*black*	**undef**
2	*black*	1
3	*grey*	2
4	*white*	
5	*grey*	2
6	*white*	
7	*white*	

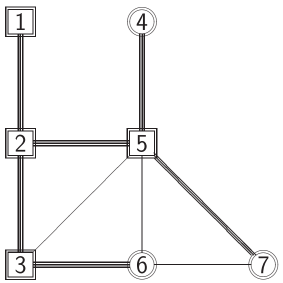
After visiting vertex 3



Head
↓
queue 1 2 3 5 6

x	*colour*[x]	$\pi[x]$
1	*black*	**undef**
2	*black*	1
3	*black*	2
4	*white*	
5	*grey*	2
6	*grey*	3
7	*white*	

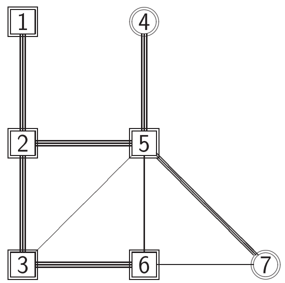
After visiting vertex 5



Head
↓
queue 1 2 3 5 6 4 7

x	*colour*[x]	$\pi[x]$
1	*black*	**undef**
2	*black*	1
3	*black*	2
4	*grey*	5
5	*black*	2
6	*grey*	3
7	*grey*	5

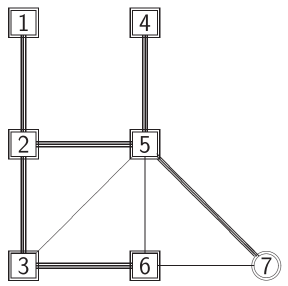
After visiting vertex 6



Head
↓
queue 1 2 3 5 6 4 7

x	*colour*[x]	$\pi[x]$
1	*black*	**undef**
2	*black*	1
3	*black*	2
4	*grey*	5
5	*black*	2
6	*black*	3
7	*grey*	5

After visiting vertex 4

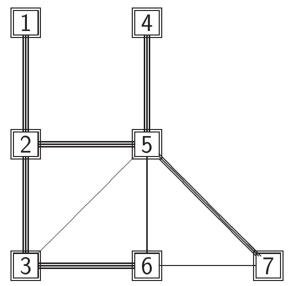


Head

queue 1 2 3 5 6 4 7

x	*colour[x]*	$\pi[x]$
1	*black*	**undef**
2	*black*	1
3	*black*	2
4	*black*	5
5	*black*	2
6	*black*	3
7	*grey*	5

After visiting vertex 7



Head

queue 1 2 3 5 6 4 7

x	*colour[x]*	$\pi[x]$
1	*black*	**undef**
2	*black*	1
3	*black*	2
4	*black*	5
5	*black*	2
6	*black*	3
7	*black*	5

At termination

At the termination of breadth-first search every vertex in the same connected component as v is a black vertex and the array π contains details of a spanning tree for that component — the breadth-first tree.

Time analysis

During the breadth-first search each vertex is enqueued once and dequeued once. As each enqueueing/dequeueing operation takes constant time, the queue manipulation takes $\Theta(V)$ time. At the time the vertex is dequeued, the adjacency list of that vertex is completely examined. Therefore we take $\Theta(E)$ time examining all the adjacency lists and the total time is $\Theta(V + E)$.

Uses of BFS

Breadth-first search is particularly useful for certain simple tasks such as determining whether a graph is connected, or finding the distance between two vertices.

The vertices of G are examined in order of increasing distance from v — first v, then its neighbours, then the vertices at distance 2 from v and so on. The spanning tree constructed provides a shortest path from any vertex back to v just by following the array π.

Therefore it is simple to modify the breadth-first search to provide an array of distances *dist* where *dist[u]* is the distance of the vertex u from the source vertex v.

Breadth-first search finding distances

To initialize the search we mark the colour of every vertex as *white* and the queue is empty. Then the first step is to mark the colour of v to be *grey*, set $\pi[v]$ to be undefined, set *dist*[v] to be 0, and add v to the queue, Q. Then we repeat the following procedure.

while Q is not empty
    Pop vertex w from the head of Q
    for each vertex x adjacent to w **do**
        if *colour*[x] is *white* **then**
            dist[x] $\leftarrow$ *dist*[w]+1
            $\pi[x] \leftarrow w$
            colour[x] $\leftarrow$ *grey*
            Push x on to the tail of Q
        end if
    end for
    colour[w] $\leftarrow$ *black*
end while

Depth-first graph search

Depth-first search is another important technique for searching a graph. Similarly to breadth-first search it also computes a spanning tree for the graph, but the tree is very different.

The structure of depth-first search is naturally *recursive* so we will give a recursive description of it. Nevertheless it is useful and important to consider the non-recursive implementation of the search.

The fundamental idea behind depth-first search is to visit the next unvisited vertex, thus extending the current path as far as possible. When the search gets stuck in a "corner" we back up along the path until a new avenue presents itself (this is called *backtracking*).

Basic recursive depth-first search

The following recursive program computes the depth-first search tree for a graph G starting from the source vertex v.

To initialize the search we mark the colour of every vertex as *white*. Then we call the recursive routine DFS(v) where v is the source vertex.

procedure DFS(w)
    colour[w] $\leftarrow$ *grey*
    for each vertex x adjacent to w **do**
        if *colour*[x] is *white* **then**
            $\pi[x] \leftarrow$ *w*
            DFS(x)
        end if
    end for
    colour[w] $\leftarrow$ *black*

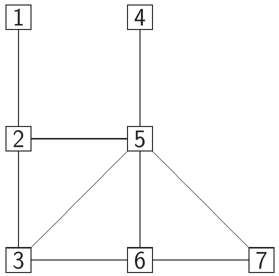
At the end of this depth-first search procedure we have produced a spanning tree containing every vertex in the connected component containing v.

A Non-recursive DFS

A non-recursive DFS requires a *stack* to record the previously visited vertices.

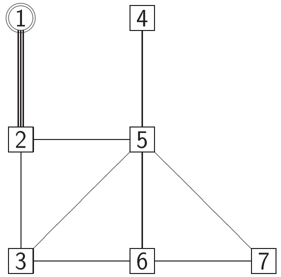
procedure DFS(w)
    initialize stack S
    push w onto S
    while S not empty **do**
        $x \leftarrow$ pop off S
        if *colour*[x]=*white* **then**
            colour[x] $\leftarrow$ *black*
            for each vertex y adjacent to x **do**
                if *colour*[y] is *white* **then**
                    push y onto S
                    $\pi[y] \leftarrow$ *x*
                end if
            end for
        end if
    end while

Example of depth-first search



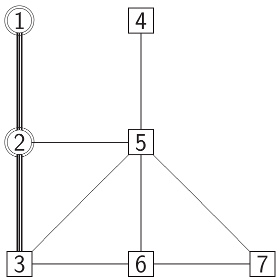
x	*colour*[x]	$\pi[x]$
1	*white*	**undef**
2	*white*	
3	*white*	
4	*white*	
5	*white*	
6	*white*	
7	*white*	

Immediately prior to calling DFS(2)



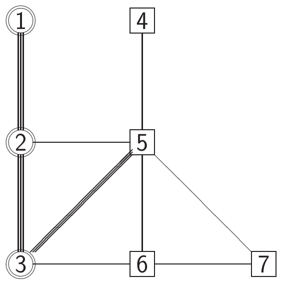
x	*colour*[x]	$\pi[x]$
1	*grey*	**undef**
2	*white*	1
3	*white*	
4	*white*	
5	*white*	
6	*white*	
7	*white*	

Immediately prior to calling DFS(3)



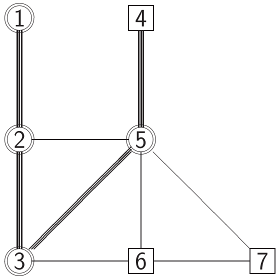
x	*colour*[x]	$\pi[x]$
1	*grey*	**undef**
2	*grey*	1
3	*white*	2
4	*white*	
5	*white*	
6	*white*	
7	*white*	

Immediately prior to calling DFS(5)



x	*colour*[x]	$\pi[x]$
1	*grey*	**undef**
2	*grey*	1
3	*grey*	2
4	*white*	
5	*white*	3
6	*white*	
7	*white*	

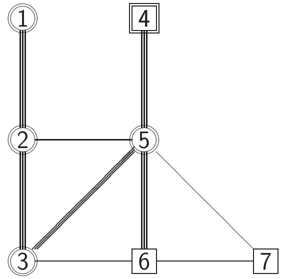
Immediately prior to calling DFS(4)



x	*colour*[x]	$\pi[x]$
1	*grey*	**undef**
2	*grey*	1
3	*grey*	2
4	*white*	5
5	*grey*	3
6	*white*	
7	*white*	

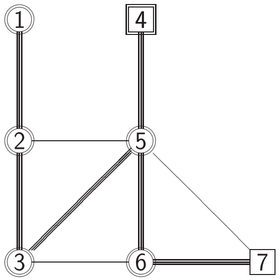
Immediately prior to calling DFS(6)

Now the call to DFS(4) actually finishes without making any more recursive calls so we return to examining the neighbours of vertex 5, the next of which is vertex 6.



x	*colour*[x]	$\pi[x]$
1	*grey*	**undef**
2	*grey*	1
3	*grey*	2
4	*black*	5
5	*grey*	3
6	*white*	5
7	*white*	

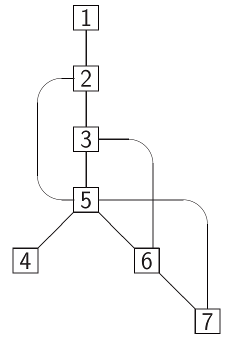
Immediately prior to calling DFS(7)



x	*colour*[x]	$\pi[x]$
1	*grey*	**undef**
2	*grey*	1
3	*grey*	2
4	*black*	5
5	*grey*	3
6	*grey*	5
7	*white*	6

The depth-first search tree

After completion of the search we can draw the depth-first search tree for this graph:



In this picture the slightly thicker straight edges are the **tree edges** (see later) and the remaining edges are the **back edges** — the back edges arise when we examine an edge (u, v) and discover that its endpoint v no longer has the colour *white*

Analysis of DFS

The running time of DFS is easy to analyse as follows.

First we observe that the routine DFS(w) is called exactly once for each vertex w; during the execution of this routine we perform only constant time array accesses, and run through the adjacency list of w once.

Running through the adjacency list of each vertex exactly once takes $O(E)$ time overall, and hence the total time taken is $O(V + E)$.

In fact, we can say more and observe that because every vertex and every edge are examined precisely once in both BFS and DFS, the time taken is $O(V + E)$.

Discovery and finish times

The operation of depth-first search actually gives us more information than simply the depth-first search tree; we can assign two times to each vertex.

procedure DFS(w)
  colour[w] ← *grey*
  discovery[w] ← *time*
  time ← *time+1*
  for each vertex x adjacent to w **do**
    if *colour[x]* is *white* **then**
      $\pi[x]$ ← *w*
      DFS(x)
    end if
  end for
  colour[w] ← *black*
  finish[w] ← *time*
  time ← *time+1*

The parenthesis property

This assigns to each vertex a *discovery* time, which is the time at which it is first discovered, and a *finish* time, which is the time at which all its neighbours have been searched and it no longer plays any further role in the search.

The discovery and finish times satisfy a property called the *parenthesis property*.

Imagine writing down an expression consisting entirely of labelled parentheses — at the time of discovering vertex u we open a parenthesis $(_u$ and a the time of finishing with u we close the parenthesis $_u)$.

Then the resulting expression is a well-formed expression with correctly nested parentheses.

For our example depth-first search we get:

$$(_1\ (_2\ (_3\ (_4\ (_5\ _5)\ (_6\ (_7\ _7)\ _6)\ _4)\ _3)\ _2)\ _1)$$

Depth-first search for directed graphs

A depth-first search on an undirected graph produces a classification of the edges of the graph into *tree edges*, or *back edges*. For a directed graph, there are further possibilities. The same depth-first search algorithm can be used to classify the edges into four types:

tree edges If the procedure DFS(u) calls DFS(v) then (u, v) is a tree edge

back edges If the procedure DFS(u) explores the edge (u, v) but finds that v is an already visited ancestor of u, then (u, v) is a back edge

forward edges If the procedure DFS(u) explores the edge (u, v) but finds that v is an already visited descendant of u, then (u, v) is a forward edge

cross edges All other edges are cross-edges

Topological sort

We shall consider a classic simple application of depth-first search.

Definition A *directed acyclic graph (dag)* is a directed graph with no directed cycles.

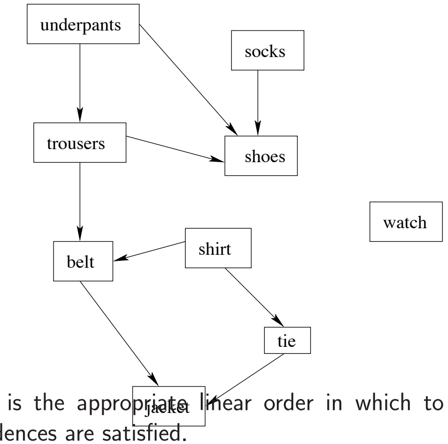
Theorem In a depth-first search of a dag there are no back edges.

Consider now some complicated process in which various jobs must be completed before others are started. We can model this by a graph D where the vertices are the jobs to be completed and there is an edge from job u to job v if job u must be completed before job v is started. Our aim is to find some linear ordering of the jobs such that they can be completed without violating any of the constraints.

This is called finding a *topological sort* of the dag D.

Example of a dag to be topologically sorted

For example, consider this dag describing the stages of getting dressed and the dependency between items of clothing (from CLRS, page 550).



What is the appropriate linear order in which to do these jobs so that all the precedences are satisfied.

Algorithm for TOPOLOGICAL SORT

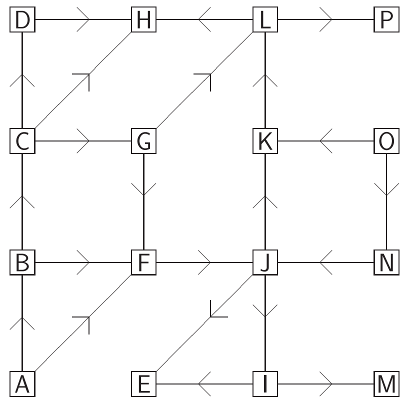
The algorithm for topological sort is an extremely simple application of depth-first search.

Algorithm

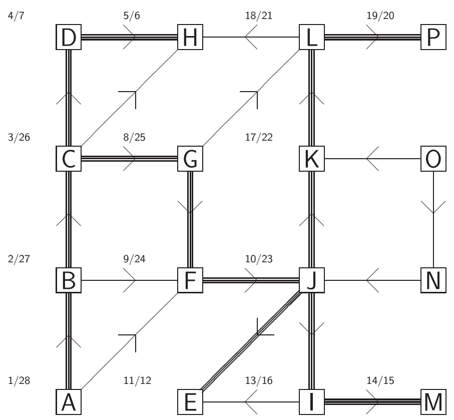
Apply the depth-first search procedure to find the finishing times of each vertex. As each vertex is finished, put it onto the *front* of a linked list.

At the end of the depth-first search the linked list will contain the vertices in topologically sorted order.

Doing the topological sort

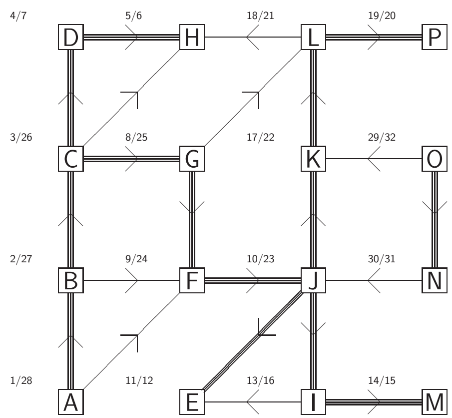


After the first depth-first search



Notice that there is a component that has not been reached by the depth-first search.

After the entire search



The final topological sort is:
$O - N - A - B - C - G - F - J - K - L - P - I - M - E - D - H$

Analysis and correctness

Time analysis of the algorithm is very easy — to the $\Theta(V+E)$ time for the depth-first search we must add $\Theta(V)$ time for the manipulation of the linked list. Therefore the total time taken is again $\Theta(V+E)$.

Proof of topological sort

Suppose DFS has calculated the finish times of a dag $G = (V, E)$. For any pair of adjacent vertices $u, v \in V$ (implying $(u, v) \in E$) then we just need to show $f[v] < f[u]$ (the destination vertex v must finish first).

For each edge (u, v) explored by DFS of G consider the colour of vertex v.

GREY: v can never be grey since v should therefore be an ancestor of u and so the graph would be cyclic.

Proof (contd)

WHITE: v is a descendant of u so we will set its time now but we are still exploring u so we will set its finished time at some point in the future (and so therefore $f[v] < f[u]$). (refer back to the psuedocode).

BLACK: v has already been visited and so its finish time must have been set earlier, whereas we are still exploring u and so we will set its finish time in the future (and so again $f[v] < f[u]$).

Since for every edge in G there are two possible destination vertex colours and in each case we can show $f[v] < f[u]$, we have shown that this property applies to every connected vertex in G.

See CLRS (theorem 22.11) for a more thorough treatment.

Other uses for DFS

DFS is the standard algorithmic method for solving the following two problems:

Strongly connected components In a directed graph D a strongly connected component is a maximal subset S of the vertices such that for any two vertices u, $v \in S$ there is a directed path from u to v and from v to u.

Depth-first search can be used on a digraph to find strongly connected components in time $\Theta(V + E)$.

Articulation points In a connected, undirected graph, an *articulation point* is a vertex whose removal disconnects the graph.

Depth-first search can be used on a graph to find all the articulation points in time $\Theta(V + E)$.

Summary

1. Searching may occur *breadth* first (BFS) or *depth* first (DFS).
2. DFS and BFS create a spanning tree from any graph.
3. BFS visits the vertices nearest to the source first. It can be used to determine whether a graph is connected.
4. DFS visits the vertices furtherest to the source first. It can be used to perform a topological sort.