# COMP90038 Algorithms and Complexity

## Graph Traversal

Michael Kirley

Lecture 8

Semester 1, 2016

# Breadth-First and Depth-First Traversal

There are two natural approaches to the traversal of a graph.

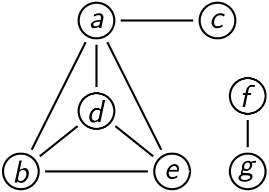
Suppose we have a graph and we want to explore all its nodes systematically. Suppose we start from node $v$ and $v$ has neighbouring nodes $x$, $y$ and $z$.

In a breadth-first approach we, roughly, explore $x$, $y$ and $z$ before exploring any of their neighboring nodes.

In a depth-first approach, we may explore, say, $x$ first, but then, before exploring $y$ and $z$, we first explore one of $x$'s neighbours, then one of its neighbours, and so on.

(This is really hard to express in English—we do need pseudo-code!)

# Depth-First Search



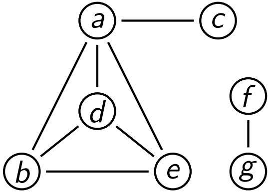

Both graph traversal methods rely on marking nodes as they are visited—so that we can avoid revisiting nodes.

Depth-first search is based on backtracking.

Neighbouring nodes are considered in, say, alphabetical order.

For the example graph, nodes are visited in the order $a, b, d, e, c, f, g$.

# Depth-First Search: The Traversal Stack

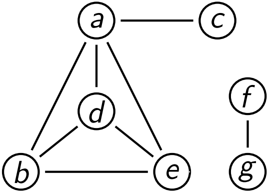



DFS corresponds to using a stack discipline for keeping track of where we are in the overall process.

Here is how the "where-we-came-from" stack develops for the example:

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | $e$ | | | | | | | | | |
| | | $d$ | $d$ | $d$ | | | | | | | | |
| | $b$ | $b$ | $b$ | $b$ | $b$ | | $c$ | | | | $g$ | |
| $a$ | $a$ | $a$ | $a$ | $a$ | $a$ | $a$ | $a$ | $a$ | | $f$ | $f$ | $f$ |
| — | — | — | — | — | — | — | — | — | — | — | — | — |

# Depth-First Search: The Traversal Stack

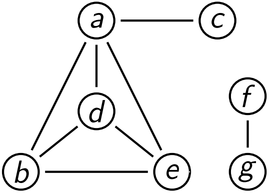



Levitin uses a more compact notation for the stack's history. Here is how the stack develops, in Levitin's notation:

| | | |
|---|---|---|
| $e_{4,1}$ | | |
| $d_{3,2}$ | | |
| $b_{2,3}$ | $c_{5,4}$ | $g_{7,6}$ |
| $a_{1,5}$ | | $f_{6,7}$ |

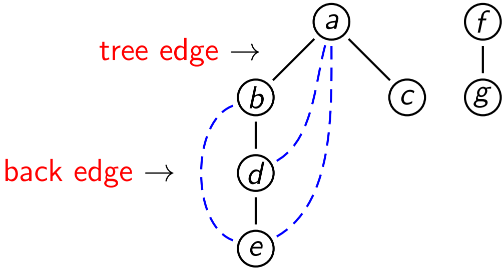
The first subscripts give the order in which nodes are pushed, the second the order in which they are popped off the stack.

# Depth-First Search: The Depth-First Search Forest




Another useful tool for depicting a DF traversal is the DFS tree (for a connected graph).

More generally, we get a DFS forest:



# Depth-First Search: The Algorithm

**function** $\text{DFS}(\langle V, E \rangle)$
    mark each node in $V$ with 0
    $count \leftarrow 0$
    **for** each $v$ in $V$ **do**
        **if** $v$ is marked 0 **then**
            $\textsc{DfsExplore}(v)$

**function** $\textsc{DfsExplore}(v)$
    $count \leftarrow count + 1$
    mark $v$ with $count$
    **for** each edge $(v, w)$ **do**        $\triangleright$ $w$ is $v$'s neighbour
        **if** $w$ is marked with 0 **then**
            $\textsc{DfsExplore}(w)$

This works both for directed and undirected graphs.

# Depth-First Search: The Algorithm

The "marking" of nodes is usually done by maintaining a separate array, `mark`, indexed by $V$.

For example, when we wrote "mark $v$ with *count*", that would be implemented as "`mark[v] := count`".

How to find the nodes adjacent to $v$ depends on the graph representation used.

Using an adjacency matrix `adj`, we need to consider `adj[v,w]` for each `w` in $V$. Here the complexity of graph traversal is $\Theta(|V|^2)$.

Using adjacency lists, for each $v$, we traverse the list `adj[v]`.

In this case, the complexity of traversal is $\Theta(|V| + |E|)$. Why?

# Applications of Depth-First Search

It is easy to adapt the DFS algorithm so that it can decide whether a graph is connected.

How?

# Applications of Depth-First Search

It is easy to adapt the DFS algorithm so that it can decide whether a graph is connected.

How? ✎

It is also easy to adapt it so that it can decide whether a graph has a cycle.

How? ✎

# Applications of Depth-First Search

It is easy to adapt the DFS algorithm so that it can decide whether a graph is connected.

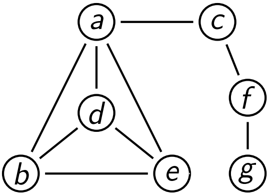
How? ✎

It is also easy to adapt it so that it can decide whether a graph has a cycle.

How? ✎

In terms of DFS forests, how can we tell if we have traversed a dag?

# Breadth-First Search



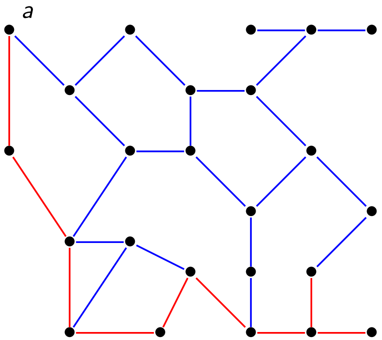

Breadth-first search proceeds in a concentric manner, visiting all nodes that are one step away from the start node, then all those that are two steps away (except those that were already visited), and so on.

Again, neighbouring nodes are considered in, say, alphabetical order.

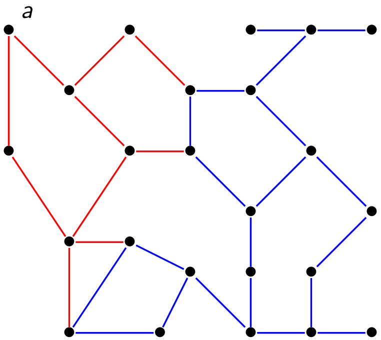
For the example graph, nodes are visited in the order $a, b, c, d, e, f, g$.

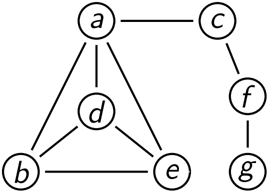
# Depth-First Search vs Breadth-First

Typical depth-first search:




Typical breadth-first search:





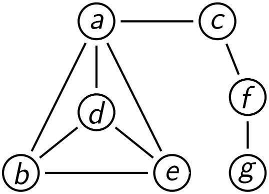

BFS uses a queue discipline for keeping track of pending tasks.

How the queue develops for the example:

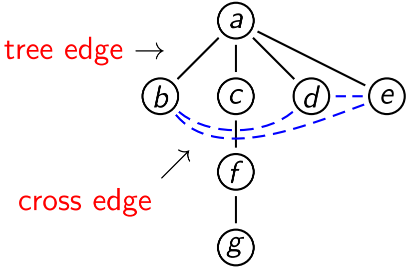
| | | | |
|---|---|---|---|
| $a_1$ | | | |
| $b_2$ | $c_3$ | $d_4$ | $e_5$ |
| $c_3$ | $d_4$ | $e_5$ | |
| $d_4$ | $e_5$ | $f_6$ | |
| $e_5$ | $f_6$ | | |
| $f_6$ | | | |
| $g_7$ | | | |

The subscript again is Levitin's; it gives the order in which nodes are processed.

# The Breadth-First Search Forest




Here is the BFS tree for the example:




In general, we may get a BFS forest.

# Breadth-First Search: The Algorithm

**function** $\mathrm{BFS}(\langle V, E\rangle)$
  mark each node in $V$ with 0
  $count \leftarrow 0$, $init$(queue) ▷ create an empty queue
  **for** each $v$ in $V$ **do**
    **if** $v$ is marked 0 **then**
      $count \leftarrow count + 1$
      mark $v$ with $count$
      $inject(queue, v)$ ▷ queue containing just $v$
      **while** $queue$ is non-empty **do**
        $u \leftarrow eject(queue)$ ▷ dequeues $u$
        **for** each edge $(u, w)$ adjacent to $u$ **do**
          **if** $w$ is marked with 0 **then**
            $count \leftarrow count + 1$
            mark $w$ with $count$
            $inject(queue, w)$ ▷ enqueues $w$

# Breadth-First Search: The Algorithm

BFS has the same complexity as DFS.

Again, the same algorithm works for directed graphs as well.

Certain problems are most easily solved by adapting BFS.

For example, given a graph and two nodes, $a$ and $b$ in the graph, how would you find the length of the shortest path from $a$ to $b$?

# Topological Sorting

We mentioned scheduling problems and their representation by directed graphs.

Assume a directed edge from $a$ to $b$ means that task $a$ must be completed before $b$ can be started.

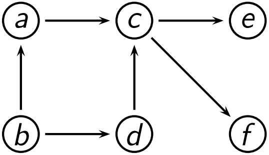
Then the graph has to be a dag.

Assume the tasks are carried out by a single person, unable to multi-task.

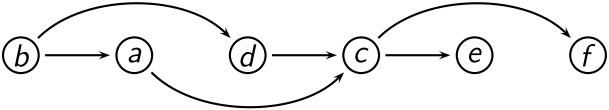
Then we should try to linearize the graph, that is, order the nodes in a sequence $v_1, v_2, \ldots, v_n$ such that for each edge $(v_i, v_j) \in E$, we have $i < j$.

# Topological Sorting: Example

There are four different ways to linearize the following graph.




Here is one:



# Topological Sorting Algorithm 1

We can solve the top-sort problem with depth-first search:

1. Perform DFS and note the order in which nodes are popped off the stack.
2. List the nodes in the reverse of that order.

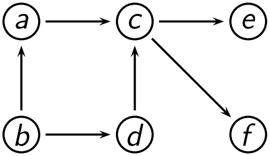
This works because of the stack discipline.

If $(u, v)$ is an edge then it is possible (for some way of deciding ties) to arrive at a DFS stack with $u$ sitting below $v$.

Taking the "reverse popping order" ensures that $u$ is listed before $v$.

# Topological Sorting Example Again

Using the DFS method and resolving ties by using alphabetical order, the graph gives rise to the traversal stack shown on the right (the popping order shown in red):




| | | |
|---|---|---|
| $e_{3,1}$ | $f_{4,2}$ | |
| $c_{2,3}$ | | $d_{6,5}$ |
| $a_{1,4}$ | | $b_{5,6}$ |

Taking the nodes in reverse popping order yields $b, d, a, c, f, e$.

# Topological Sorting Algorithm 2

An alternative method would be to repeatedly select a random source in the graph (that is, a node with no incoming edges), list it, and remove it from the graph.

This is a very natural approach, but it has the drawback that we repeatedly need to scan the graph for a source.

However, it exemplifies the general principle of decrease-and-conquer.

# Next Week

So next we turn our attention to the "decrease and conquer" principle (Levitin Chapter 4).