Part 8: Graph Algorithms

Contents

[document finalized]

Graph Traversal

Lecture 32

Graph traversal algorithms provide a systematic way of exploring a given graph.

Can be used as a framework for designing efficient graph algorithms

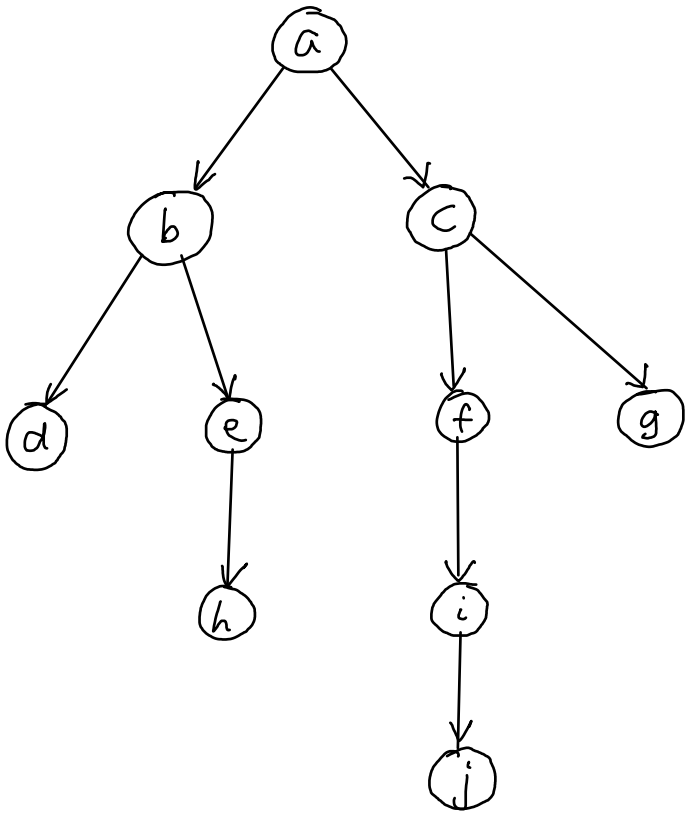
Tree Traversal Algorithms

Systematically visit all nodes in directed tree rooted at v:

```
function TreeSearch(T, v)
do something with v
for each child u of v in T do
  TreeSearch(T, u)
end
```

Example 1: Printing all nodes of a tree in order in which they are discovered by the search

```
function PrintTree(T, v)
print v
for each child u of v in T do
  PrintTree(T, u)
end
```



`PrintTree(T, a)` output: a b d e h c f i j g

What is the runtime of this algorithm?
Space requirements?

PrintTree is called exactly $|E|+1$ times: once to invoke it, and then once for each edge in the tree.

What is the maximum call-stack level l that can be reached?

$l \leq |E|$, because $|E|$ is the length of the longest possible path in T — because we only have $|E|$ edges to work with and there are no cycles.

So, in the worst case, we need additional space in the size of T

In general, however, l equals the height of v, which may be much smaller than $|E|$.

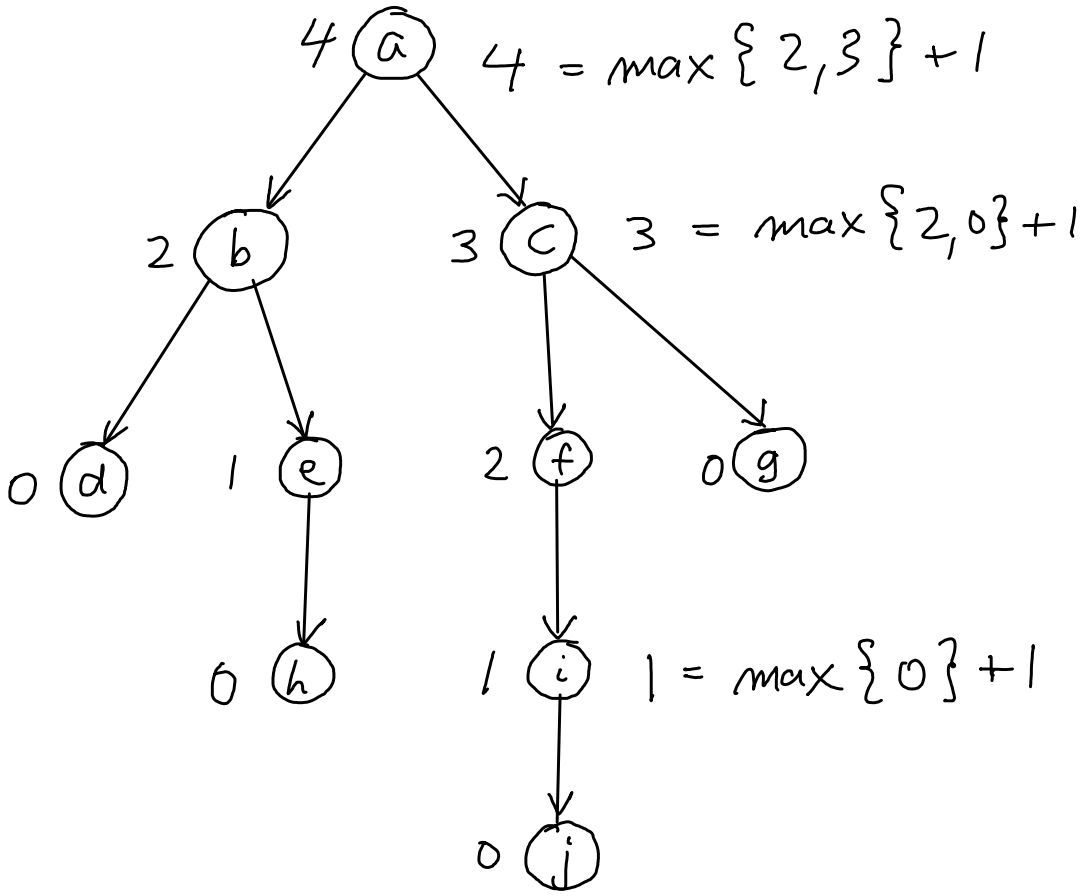
Example 2: Compute node heights

Recall: the height $h(v)$ of a node v in a tree is the maximum length of a path from v to a leaf (i.e., a node without child).

We can express this recursively:

$h(v) = 0$, if v is a leaf

$h(v) = 1 + \max\{h(u) \mid u \text{ is a child of } v\}$, if v is not a leaf



Recursive code, straight from the recurrence relation:

$h(v) = 0$, if v is a leaf

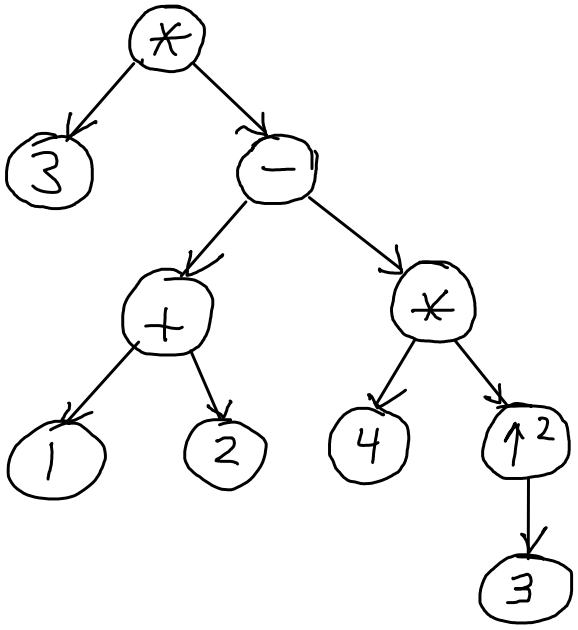
$h(v) = 1 + \max\{h(u) \mid u \text{ is a child of } v\}$, if v is not a leaf

```
function height(T, v)
h <- -1     // first height always greater,
            // 1 added at the end => also
            // works for leaves v
for each child u of v do
  g <- height(T, u)
  h <- max(h, g)
end
return h+1
```

Example 3: Evaluate expression trees

Arithmetic expressions can be viewed as binary trees:

$3(1+2-4\cdot 3^2) \sim$



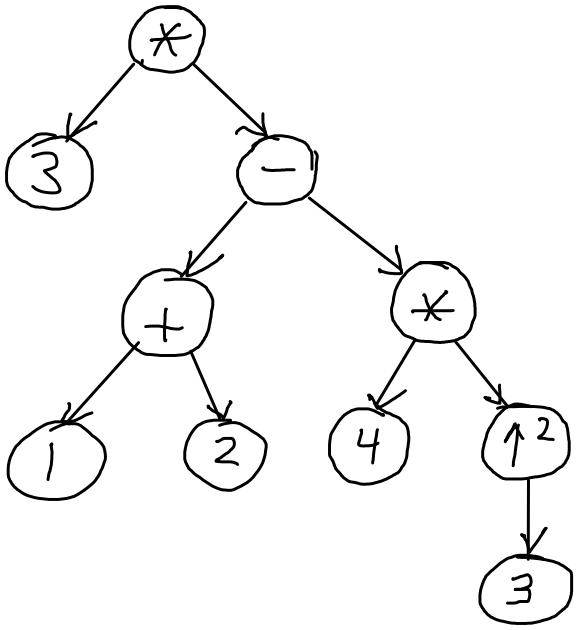
Leaves correspond to values and interior nodes correspond to functions applied to arguments that are the results of subexpressions

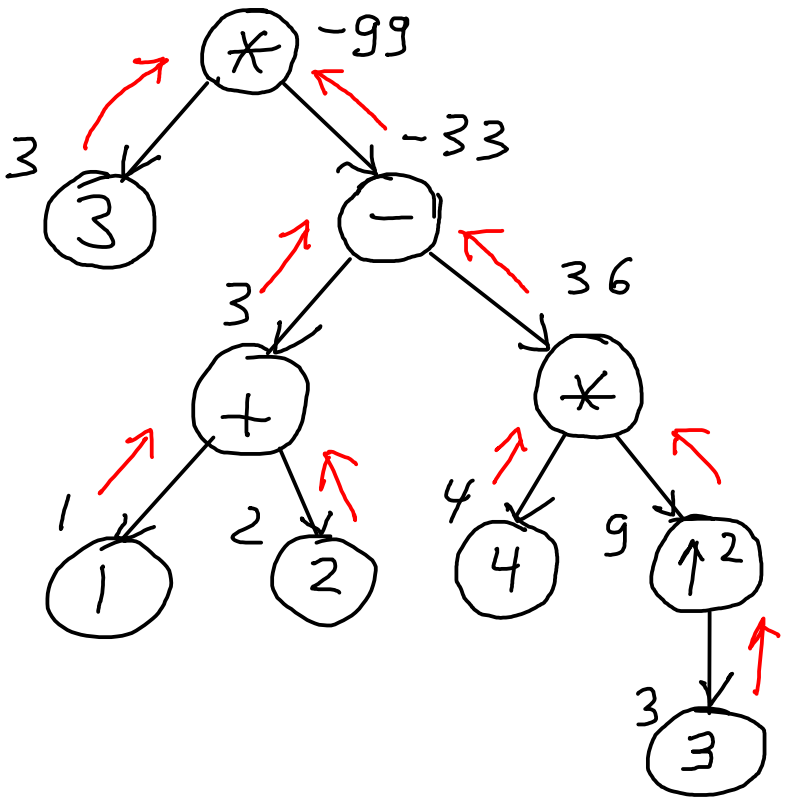
How can we determine the value of the root in a given expression tree?

```
// assume each node v carries the
// following information
//   v.left  : left child of v
//   v.right : right child of v
//   v.op    : operation in interior node
//   v.value : value at leaf
function eval(T, v)
if v is leaf then
  return v.value
end
vL <- eval(T, v.left)
if v has one child then
  return v.op(vL)
end
vR <- eval(T, v.right)
return v.op(vL, vR)
```

Application:

$3(1+2-4\cdot 3^2) \sim$





Depth-First Search

What happens when we apply `TreeSearch` to arbitrary graphs?

a) We don't reach all nodes in the graph if it is disconnected, and

a) we run into an infinite loop when a cycle can be reached from the start node!

How can we fix this?

- Use each node as start node in turn
- Memorize visited nodes and don't visit them again

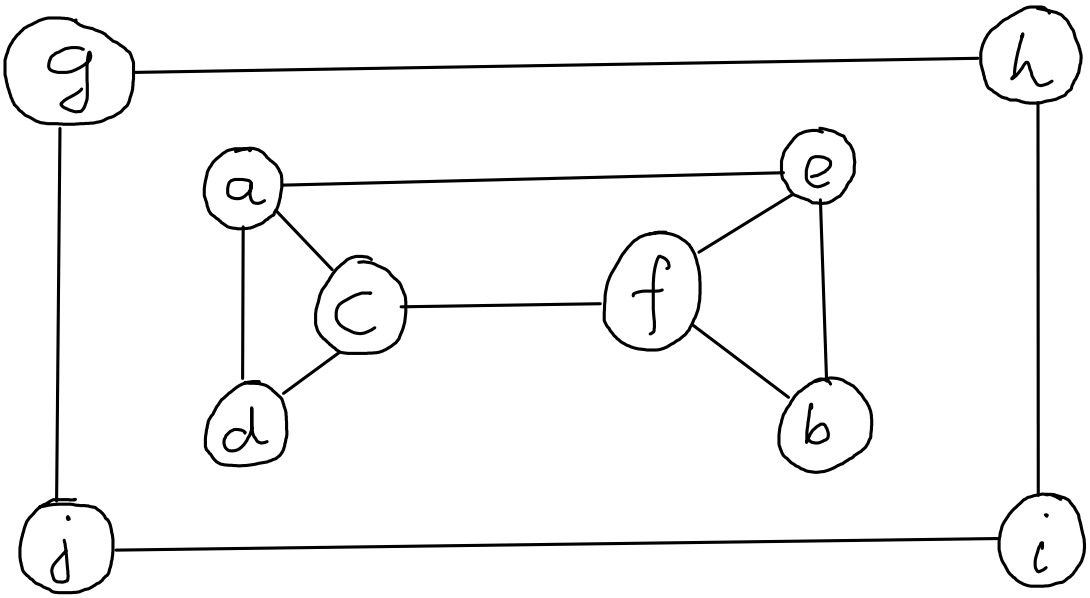
This method is called Depth-First Search (DFS), because it continues with the first child node of every node it visits and thus is going deeper and deeper until it hits a node that either has no child or whose children have been visited before. Then the search backtracks, returning to the most recent node it hasn't finished exploring.

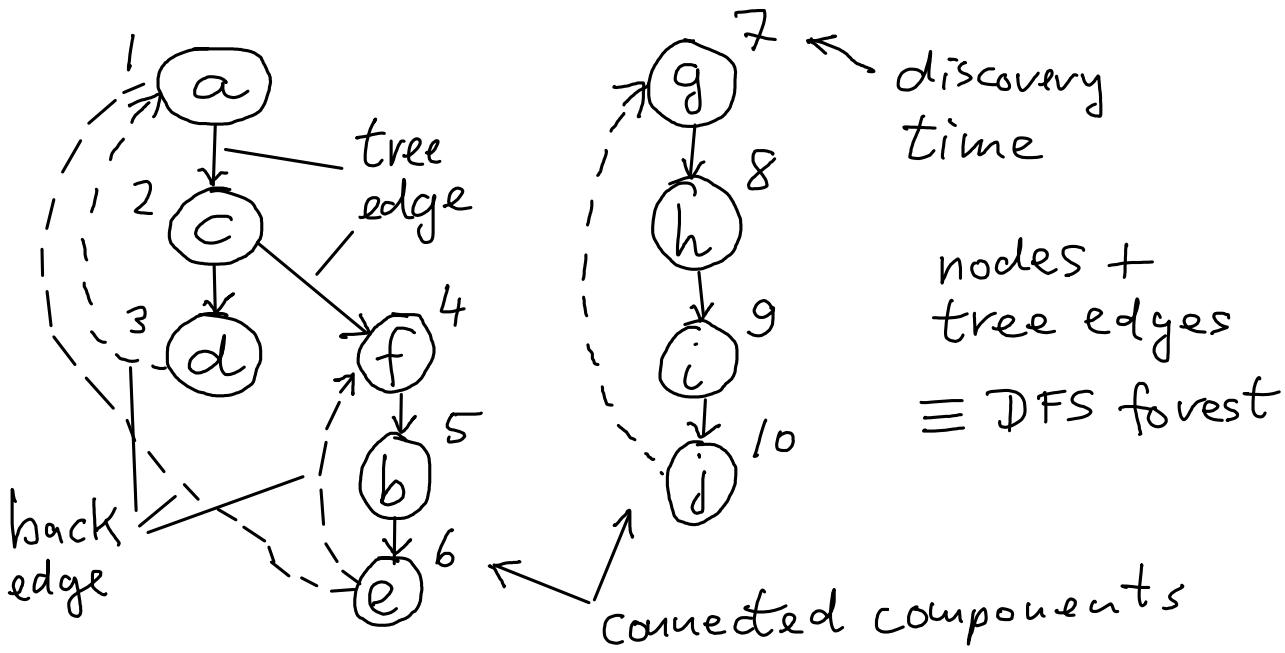
Pseudo Code:

```
// traverses graph in depth-first fashion
// uses discovery time array d:
//   d[v] = k <=>
//      node v is the k-th visited node
//   d[v] != 0  <=>  v has been visited
// also creates arrays: p (parent) and e (tree)
function DFS(G=(V,E))
for all v in V do
   d[v] <- 0              // not visited yet
   p[v] <- NIL            // no parent
end
t <- 0                    // discovery time
for each v in V do
  if d[v] = 0 then            // not yet visited?
    visit(G, d, p, e, t, v)  // visit node
  end
end

function visit(G, d, p, e, ref t, v)
t <- t + 1          // discovered new node
d[v] <- t           // set discovery time
for each u adjacent from v do
  if d[u] = 0 then
    p[u] <- v    // parent of u is v in DFS tree
    visit(G, d, p, e, t, u) // visit neighbour
  end
end
```

Example


g
h
a
e
c
f
d
b
j
i


1
a
tree edge
2
c
3
d
4
f
5
b
6
e
back edge
7
g
discovery time
8
h
9
i
10
j
nodes + tree edges
≡ DFS forest
connected components

Lecture 33

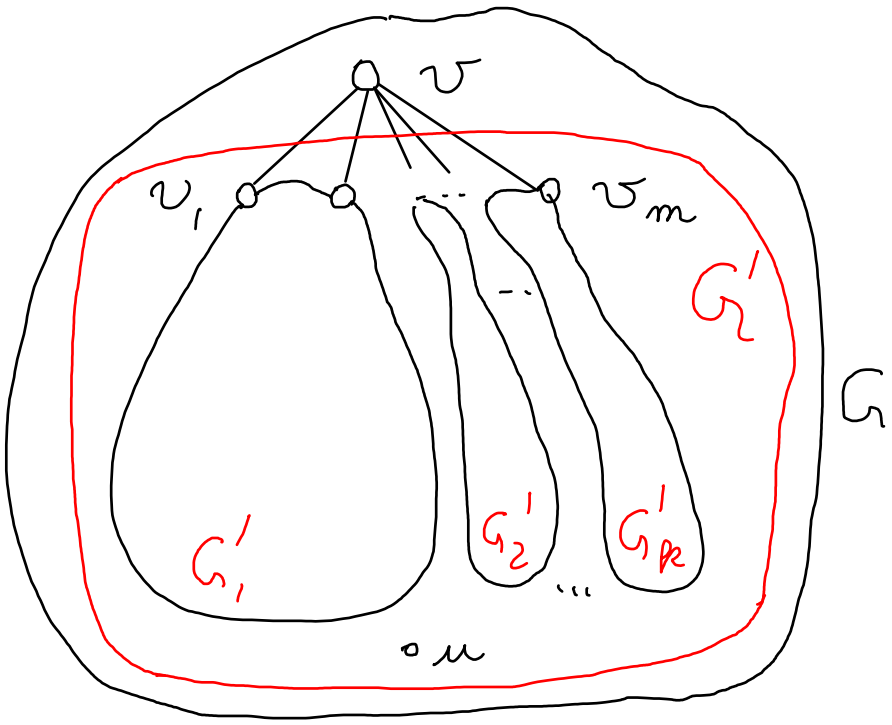
Theorem: Let $G = (V, E)$, $v \in V$ and $C(v)$ the set of vertices in the connected ncomponent of v. If $d[u] = 0$ for all $u \in C(v)$, then `visit(G,d,p,`v`,t)` discovers all nodes in $C(v)$.

Proof:

Induction on order n of G

$\underline{n = 1:}$ The only node v is discovered by the call.

$\underline{\leq n \rightarrow n + 1:}$ Consider graph G of order $n + 1$ and remove v together with all its edges to its neighbours $v_1, \ldots, v_m$.



What remains is graph G' with n nodes. The induction hypothesis applies to call `visit(`G_1`,d,p,`v_1`,t)`, i.e. this call discovers all nodes in the connected component G'_1 of v_1.

Note that in the original graph G, the discovered nodes would be the same, because v is marked visited and backedges to v don't interfere, because the search originating from v_1 will just skip them. This argument also applies to all following child `visit` calls, noting that there can't be any edges pointing into previously visited connected components.

So, after calling `visit` on all children, all nodes in the connected components $G'_1 \ldots G'_k$ of $v_1, \ldots, v_m$ in G' have been discovered.

But the graph induced by all G'_i plus v and its edges forms the connected component of v in G. This proves the claim for G. □

Applying this theorem to all nodes `visit` is called with in `DFS` proves that all nodes in G will be discovered.

Analogously, it can be shown that in directed graphs G, `visit(G,d,p,`v`,t)` discovers all nodes that are reachable from v in G by a directed path, provided that their d value was 0.

Theorem: Given a graph $G = (V, E)$ `DFS` runs in time $\Theta(|V| + |E|)$.

Proof:

The time spent in `DFS` is $\Theta(|V|)$ (two loops over V).

The total time spent in `visit` is $\Theta(|E|)$ because in the for loop each edge is visited exactly once. □

A `DFS` call creates a directed DFS forest which is made of visited nodes and so-called tree edges. Each tree is generated by a `visit` call from function `DFS`.

A **tree edge** (u, v) is formed whenever node v is first visited from u.

For undirected graphs, we call edge (u, v) a **back edge** if v is an ancestor (but not the parent) of u in the current DFS tree. Loops create back edges.

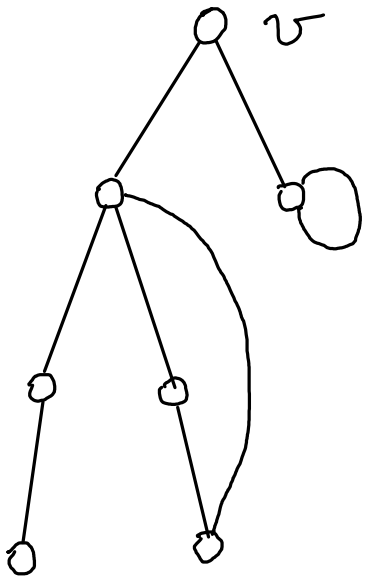
When DFS search is applied to directed graphs, there are two more edge types and a slightly changed definition of back edges:

Forward edges are those non-tree edges (u, v) that connect u to a descendant v in a DFS tree.

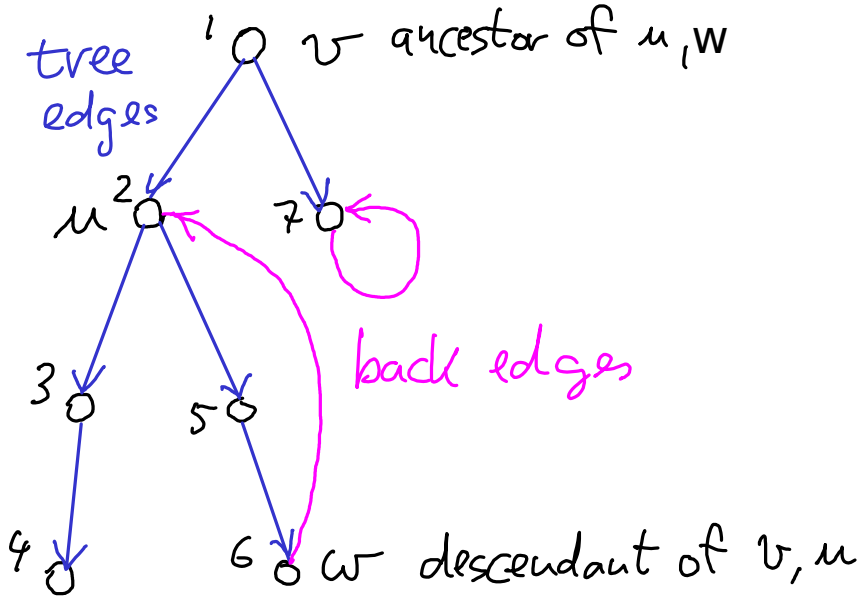
Directed edges pointing back to the parent node are considered back edges.

Cross edges are all other edges. They can go between vertices in the same DFS tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in differnt DFS trees.

DFS on undirected connected graph:

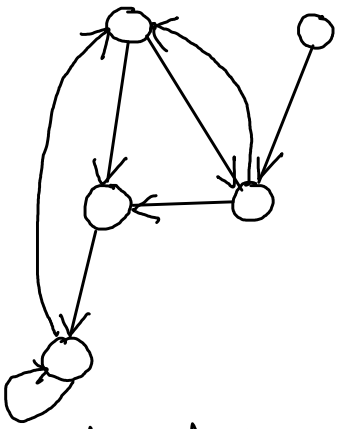


Undirected Graph

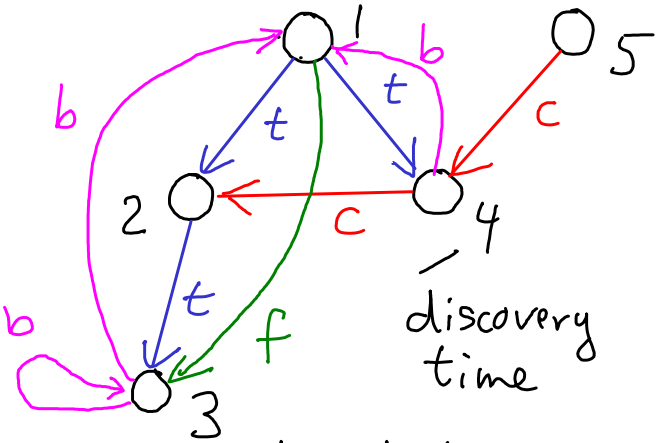


Directed DFS tree

DFS on directed graph:



directed graph



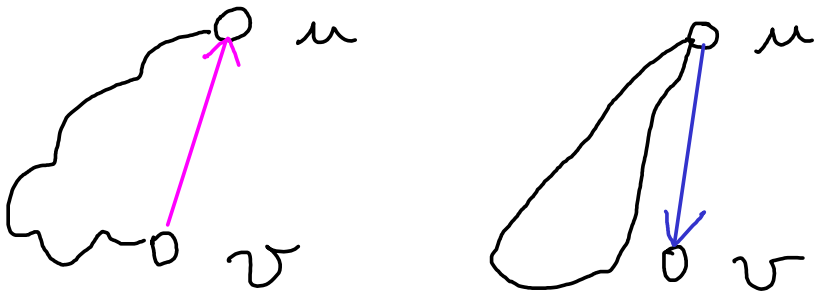
directed DFS forest

t tree edge
c cross edge
f forward edge
b back edge

When DFS is applied to undirected graphs $G = (V, E)$ only tree and back edges occur.

To see this consider any edge $\{u, v\}$ in E, for which u is discovered before v.

Then either v is discovered by searching another neighbour of u first — which means that (v, u) is a back edge — or not, in which case (u, v) is a tree edge.



How to detect back edges in DFS applied to undirected graphs?

Before calling `visit` on neighbour u of v, check whether u has been visited before and u is not the parent of v. If yes, the edge is a back edge. Otherwise, it isn't.

How to detect back edges in DFS applied to directed graphs?

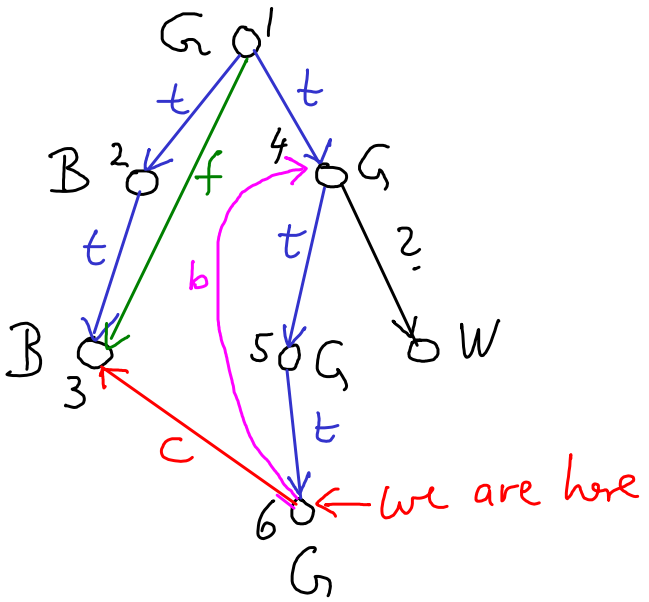
This is more tricky. Node u being visited before does not mean it is an ancestor of v in the current DFS tree. To test this, we need to know whether exploration from u isn't finished yet.

For this purpose we maintain a colour array c that tells us about the search status of nodes:

$c[v] =$ WHITE means v not visited yet

$c[v] =$ GREY means v is being visited

$c[v] =$ BLACK means visiting v is finished

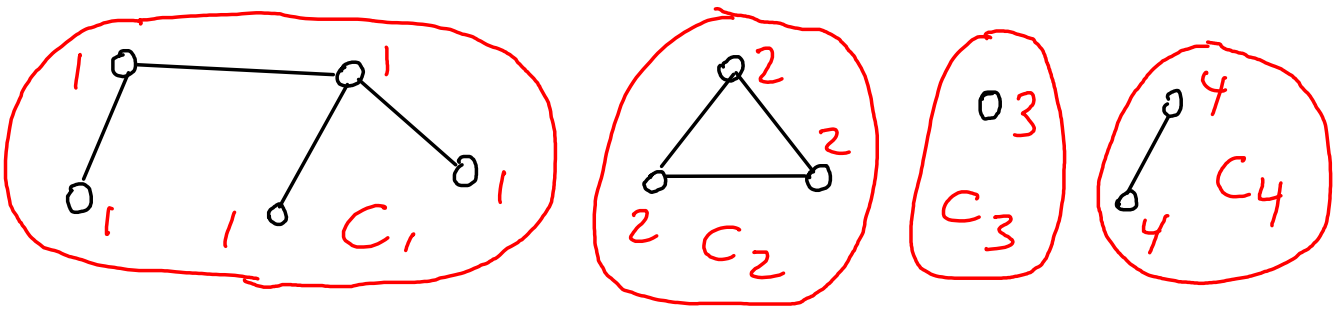


With this (v, u) is a back edge iff $c[v] =$ GREY, and (v, u) is a forward or cross edge iff $c[v] =$ BLACK.

```
// colour version of DFS
// for detecting back edges in digraphs
function DFS(G=(V,E))
for all v in V do
   d[v] <- 0              // not visited yet
   p[v] <- NIL            // no parent
   c[v] <- WHITE
end
t <- 0                    // discovery time
for each v in V do
  if d[v] = 0 then             // not yet visited?
    visit(G, d, p, e, c, t, v)  // visit node
  end
end

function visit(G, d, p, e, c, ref t, v)
t <- t + 1          // discovered new node
d[v] <- t           // set discovery time
c[v] <- GREY        // v being visited
for each u adjacent from v do
  if d[u] = 0 then
    p[u] <- v   // parent of u is v in DFS tree
    visit(G, d, p, e, c, t, u) // visit neighbour
  end
end
c[v] <- BLACK   // done visiting v
```

DFS Application 1: Connected Components



Given a graph $G = (\{1..n\}, E)$, how to compute array values $c[v]$ which indicate in which connected component node v resides?

Adapt DFS:

```
function Components(G=(V,E))
c[v] <- 0 for all v in V // not visited yet
k <- 0                   // current component
for each v in V do
  if c[v] = 0 then
    k <- k + 1 end       // next component
    co-visit(G, c, k, v) // visit nodes
  end
end
return c

function co-visit(G, c, k, v)
c[v] <- k          // v sits in component k
for each u adjacent from v do
  if c[v] = 0 then
    co-visit(G, c, k, u) // visit neighbour
  end
end
```

Runtime: $\Theta(|V| + |E|)$

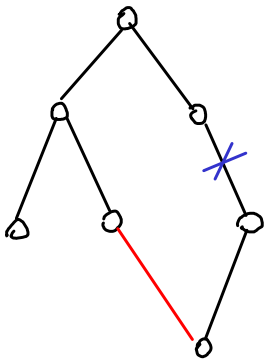
DFS Application 2: Cycle Test

Does graph $G = (V, E)$ have a cycle?

Graph Theory Result:

If $G = (V, E)$ is connected, then

$$G \text{ has cycle} \Leftrightarrow |E| \geq |V|$$



Tree: $|E| = |V| - 1$

connected graph with minimal number of edges

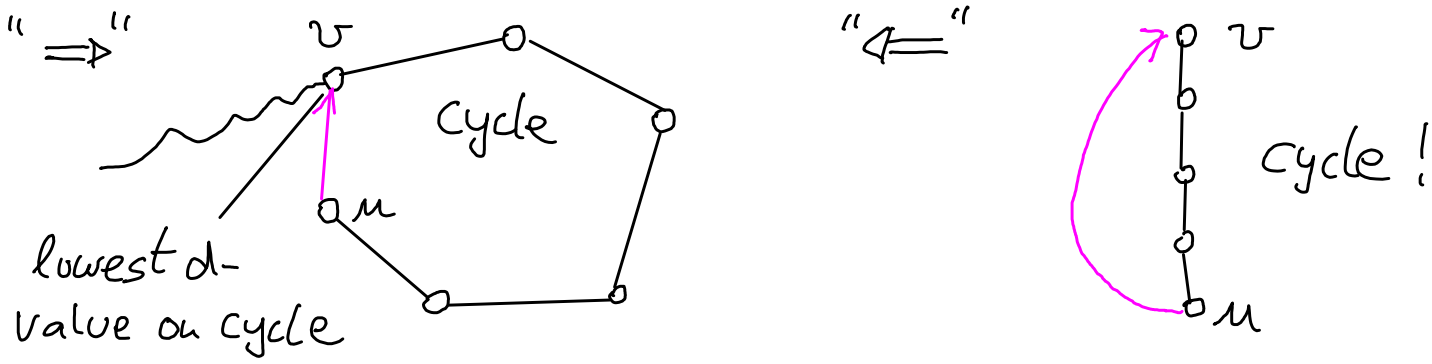
remove edge $\Rightarrow$ G' disconnected

add edge $\Rightarrow$ G' has cycle

So we could first compute the connected components, count nodes and edges in them, and the report a cycle if above inequality is true for at least one component.

We can also use DFS directly:

Theorem: G has cycle $\Leftrightarrow$ DFS applied to G encounters a back edge.



"$\Rightarrow$": Pick node v on a cycle with earliest discovery time. Then the DFS-subtree below v contains all nodes on the cycle, because all nodes on the cycle are reachable from v.

In particular the last node u in the cycle is reachable from v. Therefore, back edge (u, v) exists.

"$\Leftarrow$": The existence of back edge (u, v) by the definition means that there is a path from v to u in G and $\{u, v\} \in E$. So, G has a cycle. $\square$

Lecture 34

The Theorem also holds for directed graphs.

```
// returns true iff undirected G has a cycle
function HasCycle(G=(V,E))
for each v in V do
  d[v] <- 0 ; p[v] <- NIL // needed for back edge test
end
t <- 0                    // discovery time
for each v in V do        // visit node
  if d[v] = 0 then
    if cyc-visit(G, d, p, t, v) then return true end
  end
end
return false              // no cycle detected

function cyc-visit(G, d, p, ref t, v)
t <- t + 1          // discovered new node
d[v] <- t           // set discovery time
for each u adjacent from v do
  if d[u] > 0 then
    if u != p[v] then return true end   // back edge
  else
    if cyc-visit(G, d, t, u) then return true end
  end
end
return false
```

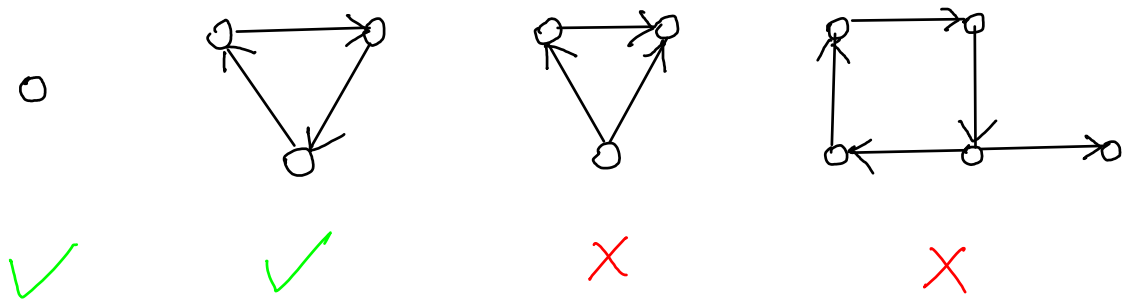
Worst-case runtime: $\Theta(|V| + |E|)$

For directed graphs, one needs to maintain the colour array and replace the `u != p[v]` test by `c[v] = GREY`.

DFS Application 3: Strong Connectedness

Is a directed graph $G = (V, E)$ strongly connected?

I.e., for all $u, v \in V$ with $u \neq v$ does there exist a directed path from u to v in G?



We can make use of the fact that `visit(G,d,p,t,`v`)` discovers all nodes that are reachable from v in G by directed paths.

In a first attempt, in `DFS` we simply reset `d` before each call to `visit(G,d,p,t,`v`)` and count how many nodes are visited in each call. If for at least one node this number is $< |V|$, we know that G is not strongly connected. Otherwise, it is.

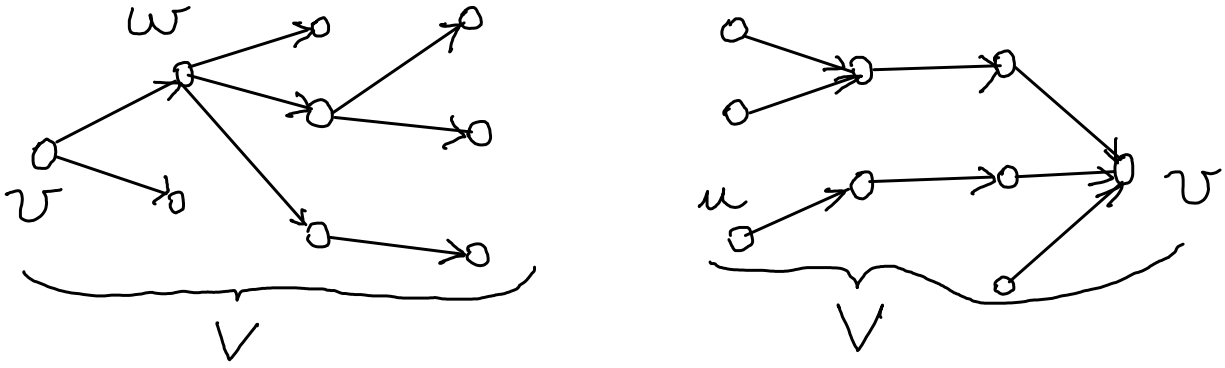
```
// return true iff G is strongly connected
function IsStronglyConnected(G=(V,E))
for each v in V do
  d[u] <- 0 for all u in V
  t <- 0
  visit(G, d, t, v) // no parent array needed
  if t < |V| then   // at least one node
    return false    // not reachable
  end
end
return true
```

The worst-case runtime is $\Theta(|V|^2 + |V||E|)$, which is rather slow.

It turns out that we don't need to run `visit` $|V|$ times.

Two times suffices.

Theorem: Directed graph $G = (V, E)$ is strongly connected iff for an arbitrary vertex $v \in V$ all other nodes are reachable from v and v is reachable from all other nodes by directed paths in G.



Proof: Let $v \in V$. If there is a node $u \neq v$ that can't be reached from v, or v can't be reached from u, then by definition G is not strongly connected.

Conversely, if all other nodes can be reached from v and v can be reached from all other nodes, then we can construct a path between any two different nodes u, w by first reaching v from u and then reaching w from v. Thus, G is strongly connected. □

A small obstacle remains: how do we compute the number of nodes from which v can be reached?

By reversing edges and running `visit` on the resulting graph!

Observation:

v can be reached from u by a path in G

$\Leftrightarrow$

u can be reached from v by a path in G'

Computing $G' = (V, E')$ from G can be accomplished in time $\Theta(|V| + |E|)$, by creating the adjacency lists of G' incrementally: add (v, u) to E' if $(u, v) \in E$.

With this we can now present the faster strong connectedness test:

```
function IsStronglyConnected2(G=(V,E))
pick v in V
d[u] <- 0 for all u in V
t <- 0
visit(G, d, t, v) // no parent array needed
if t < |V| then   // at least one node
  return false    // node not reachable
end
G' <- G with all edges reversed
d[u] <- 0 for all u in V
t <- 0
visit(G', d, t, v)
return t >= |V|
```

Worst-case runtime: $\Theta(|V| + |E|)$

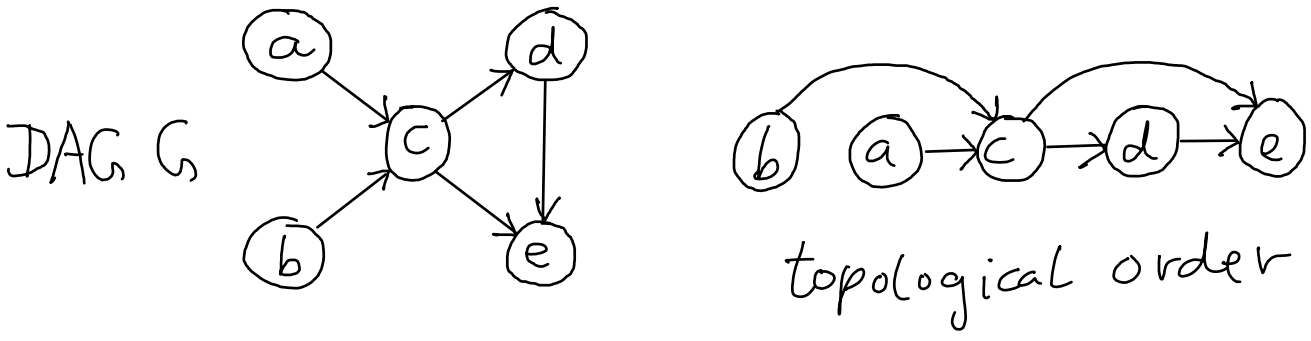
DFS Application 4: Topological Sorting

Consider the following scheduling problem: you are given a number of tasks T_i where each depends on other tasks to be finished before they can be executed.

In what order can the tasks be scheduled, such that when it comes to executing task T_i all of the tasks it depends on are already finished?

This problem can be modeled using directed acyclic graphs (DAGs). It is known as the **Topological Sorting Problem**:

Given a finite DAG G, compute a node ordering such that for each edge (u, v) in G, u is listed before v.



Cleary, if G had directed cycles, the problem has no solution. That's why we can limit the graphs to DAGs.

Is there a solution to the problem for all finite DAGs?

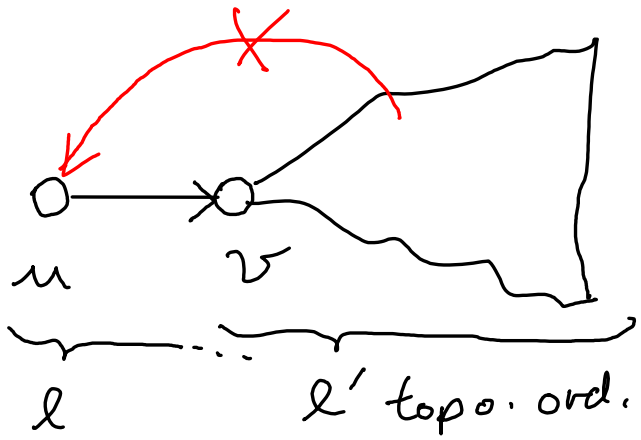
Yes.

This can be shown inductively (see A5.5).

Brute force approach: generate all $|V|!$ node orderings and check for violations. Infeasible for large graph orders.

Can we use DFS for solving this problem?

Intuition: after discovering all nodes reachable from a node u during which we created a topological ordering l' for them, we can prepend u to list l' to create a valid ordering l for this part of the DAG.



$\Rightarrow l = u\, l'$ is topological ordering

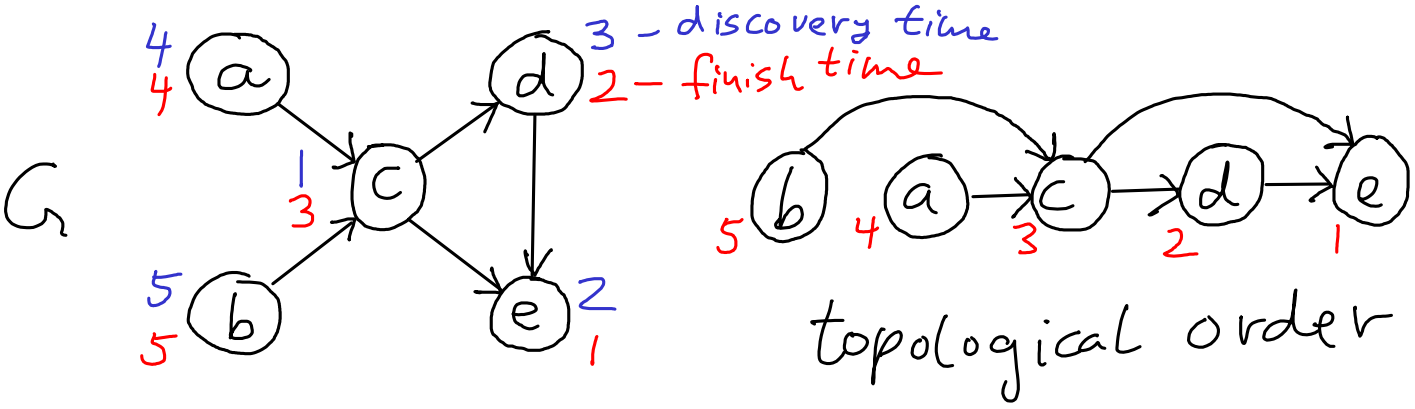
Adapted DFS function:

```
// assumes G is a DAG
function TopoSort(G=(V,E))
d[v] <- 0 for all v in V // not visited yet
t <- 0                   // discovery time
empty node list l
for each v in V do
  if d[v] = 0 then
    topo-visit(G, d, t, l, v) // visit node
  end
end
return l

function topo-visit(G, d, ref t, ref l, v)
t <- t + 1          // discovered new node
d[v] <- t           // set discovery time
for each u adjacent from v do
  if d[u] = 0 then
    topo-visit(G, d, t, l, u)
  end
end
// node v finished:
// prepend v to topological ordering
prepend(v, l)
```

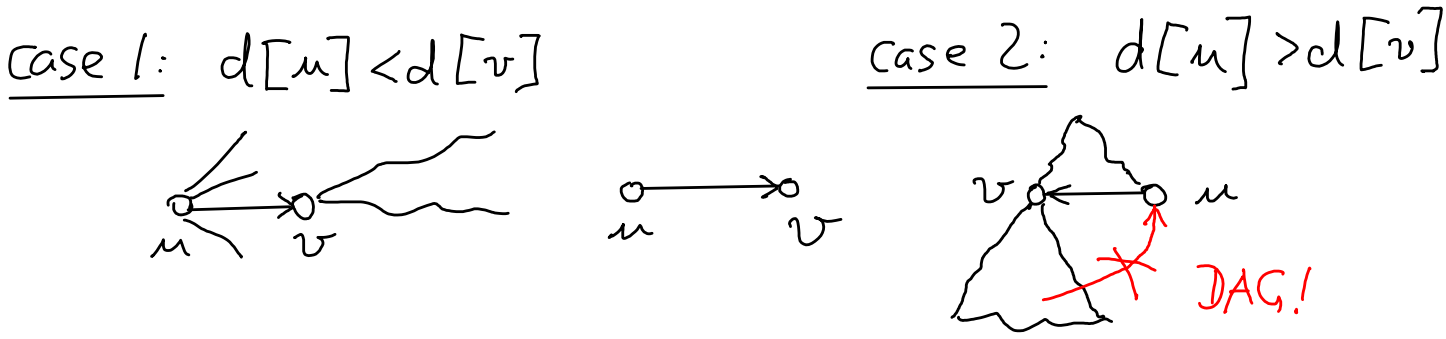
Runtime: $\Theta(|V| + |E|)$

Example:



Theorem: `TopoSort` applied to DAG $G = (V, E)$ computes a node list l so that for all $(u, v) \in E$, u is listed before v in l.

Proof: Suppose $(u, v) \in E$



Case 1: $d[u] < d[v]$

Then v is a descendant of u in the DFS tree rooted at u, because v is reachable from u.

Therefore, v is finished earlier than u, which means u preceeds v in l.

Case 2: $d[u] > d[v]$

I.e., u is discovered later than v.

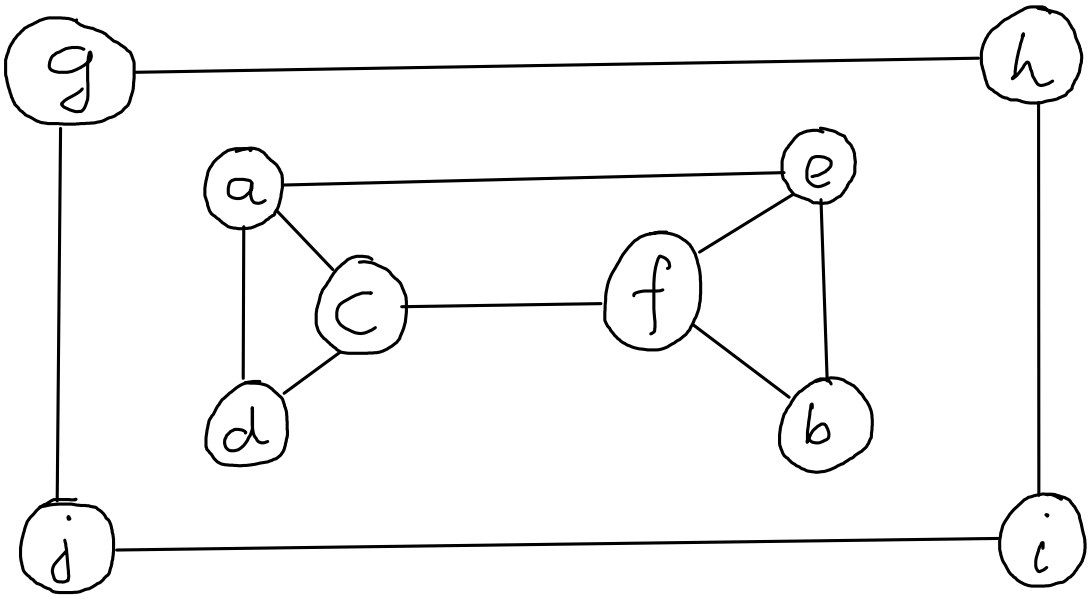
Then, u can't be a descendant of v in the DFS tree rooted at v.

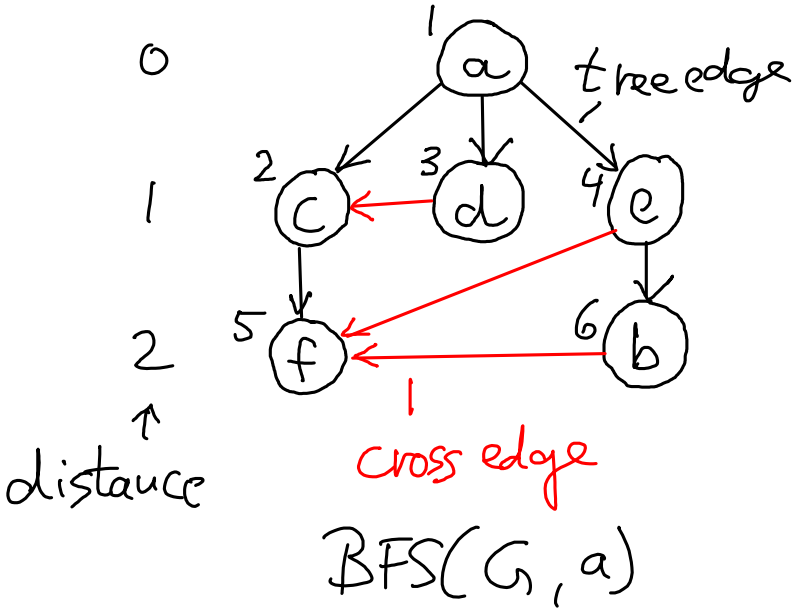
Otherwise, we would have a cycle $v \rightsquigarrow u \rightarrow v$.

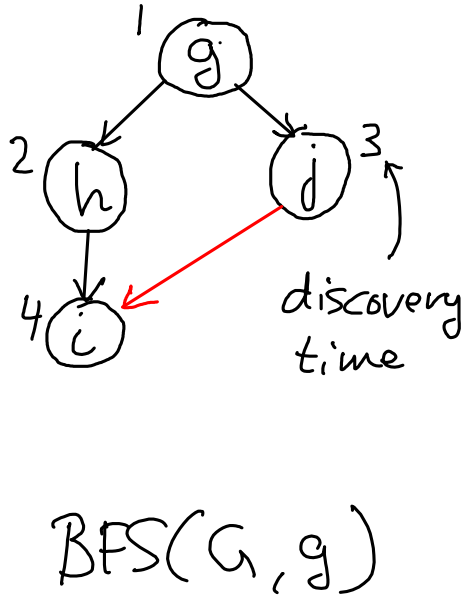
This means, that v is finished before u is discovered, and therefore u preceeds v in l. $\square$

Breadth-First Search Lecture 35

An alternative method for exploring nodes in a graph is level-by-level, i.e. visiting nodes by increased distance to a specific start node.







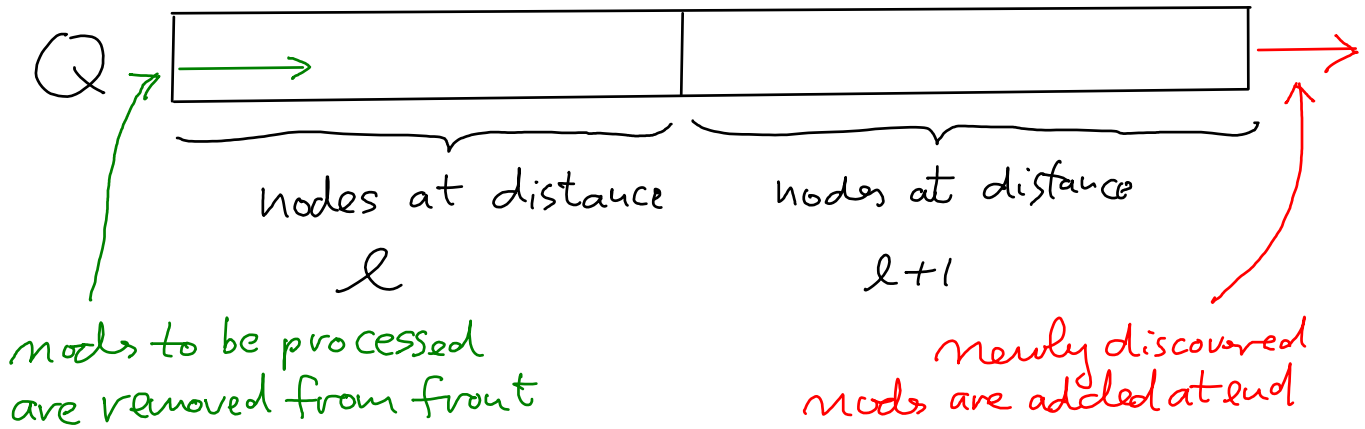
This node exploration method is called breadth-first search (BFS).

Instead of a recursive algorithm like DFS, we maintain the set of nodes yet to be searched in a first-in-first-out queue Q (see S1 problem 5 for an implementation).

Newly discovered nodes are added at the end of the queue, while nodes at the front are removed to look for adjacent nodes we have not discovered yet.

The process stops when the queue becomes empty. At this point we have visited all nodes that are reachable from the start node.

At any given time during the search the queue contains nodes of distance l from the start node at the front followed by 0 or more nodes at distance $l + 1$.



```
// traverses graph in breadth-first fashion
// starting from node s
// computes distance array d:
//   d[v] = minimal distance from s to v in G
// also creates arrays:
//   p (parent), e (tree), and c (colour)
function BFS(G=(V,E), s)
for all v in V do
   p[v] <- NIL // no parent
   c[v] <- WHITE ; d[v] <- infinity
end
c[s] <- GREY ; d[s] <- 0
QueueInit(Q)  // first-in-first-out data structure
QueueAdd(s, Q) // add s at the end of Q
while Q not empty do
  v <- QueueRemove(Q) // remove first element of Q
  for each u adjacent from v do
    if c[u] = WHITE then
      QueueAdd(u, Q)
      c[u] <- GREY  // u in Q
      d[u] <- d[v]+1// one step further away from s
      p[u] <- v     // parent of u is v in BFS tree
    end
  end
  c[v] <- BLACK     // done with v
end
```

Theorem:

The runtime of BFS is $\Theta(|V|+|E|)$, and BFS is correct, i.e. call $\texttt{BFS}(G=(V,E),s)$ terminates on all inputs, and upon termination it has discovered all nodes $v \in V$ reachable from s in G.

Further, $\texttt{d}[v] = \delta(s,v)$ (i.e. minimal path length from s to v, all edges have weight 1) and for all $v \neq s$ a shortest path from s to v can be obtained in reverse by iterating $v \leftarrow \texttt{p}[v]$ until s or NIL is reached.

Proof Sketch:

The initialization runtime is $\Theta(|V|)$. Moreover, every edge of G will be visited at most once in the inner loop because the originating nodes v will be marked GREY and BLACK and thus never discovered again. Thus, the total worst-case runtime when all nodes are being discovered (which indeed can happen, see below) is $\Theta(|V|+|E|)$.

Use induction on number of `pushFront` operations to prove the mininum distance claim, which also implies the reachabilty claim and the correctness of the path reconstruction loop. See CLRS for details. $\square$