CS 341: Algorithms

University of Waterloo
Éric Schost
eschost@uwaterloo.ca

Module 3: breadth-first search, depth-first search

Goals

This module:

- basics on **undirected graphs**
- undirected BFS and applications (shortest paths, bipartite graphs, connected components)
- undirected DFS and applications (cut vertices)
- basics on **directed graphs**
- directed DFS and applications (testing for cycles, topological sort, strongly connected components)

Undirected graphs

Definition, notation: a graph $\boldsymbol{G}$ is pair $(\boldsymbol{V}, \boldsymbol{E})$:

- V is a finite set, whose elements are called vertices
- E is a finite set, whose elements are unordered pairs of distinct vertices, and are called edges.

Convention: $\boldsymbol{n}$ is the number of vertices, $\boldsymbol{m}$ is the number of edges.

Undirected graphs

Definition, notation: a graph $\boldsymbol{G}$ is pair $(\boldsymbol{V}, \boldsymbol{E})$:

- V is a finite set, whose elements are called vertices
- E is a finite set, whose elements are unordered pairs of distinct vertices, and are called edges.

Convention: $\boldsymbol{n}$ is the number of vertices, $\boldsymbol{m}$ is the number of edges.

Data structures:

- **adjacency list:** an array $A[1..n]$ s.t. $A[v]$ is the linked list of all edges connected to v.
  $\boldsymbol{2m}$ list cells, total size $\boldsymbol{\Theta(n+m)}$, but testing if an edge exists is not $O(1)$
- **adjacency matrix:** a $(0,1)$ matrix M of size $n \times n$, with $M[v,w] = 1$ iff $\{v,w\}$ is an edge.
  size $\boldsymbol{\Theta(n^2)}$, but testing if an edge exists is $O(1)$

Connected graphs, path, cycles, trees

Definition:

- path: a sequence $v_1, \ldots, v_k$ of vertices, with $\{v_i, v_{i+1}\}$ in E for all i.
  $k = 1$ is OK.
- connected graph: $G = (V, E)$ such that for all v, w in V, there is a path $v \rightsquigarrow w$
- cycle: a path $\boldsymbol{v_1, \ldots, v_k, v_1}$ with $\boldsymbol{k \geq 3}$ and v_i's pairwise distinct
- tree: a connected graph without any cycle
- rooted tree: a tree with a special vertex called **root**

Subgraphs, connected components

Definition:

- subgraph of $G = (V, E)$: a graph $G' = (V', E')$, where
  - $V' \subset V$
  - $E' \subset E$, with all edges E' joining vertices from V'
- connected component of $G = (V, E)$
  - a connected subgraph of G
  - that is not contained in a larger connected subgraph of G

Let $G_i = (V_i, E_i)$, $i = 1, \ldots, s$ be the connected components of $G = (V, E)$.

- the V_i's are a partition of V, with $\sum_i n_i = n$ $\qquad$ $\boldsymbol{n_i = |V_i|}$
- the E_i's are a partition of E, with $\sum_i m_i = m$ $\qquad$ $\boldsymbol{m_i = |E_i|}$

Breadth-first search

Breadth-first exploration of a graph

BFS(G, s)

G: a graph with n vertices, given by adjacency lists

s: a vertex from G

```
let Q be an empty queue
let visited be an array of size n, with all entries set to false
enqueue(s, Q)
visited[s] ← true
while Q not empty do
    v ← dequeue(Q)
    for all w neighbours of v do
        if visited[w] is false
            enqueue(w, Q)
            visited[w] ← true
```

Complexity

Anaysis:

- each vertex is enqueued at most once
- so each vertex is dequeued at most once $\quad$ $\boldsymbol{O(n)}$ for steps 5-6
- so each adjacency list is read at most once

For all v, write d_v = number of neighbours of v = length of $A[v]$ = **degree** of v.

Then total cost at step 7 is

$$O\left(\sum_v d_v\right) = O(m)$$

cf. the adjacency array A has $2m$ cells (handshaking lemma)

Total: $\boldsymbol{O(n+m)}$

Correctness

Claim

For all vertices v, if visited$[v]$ is true at the end, there is a path $s \rightsquigarrow v$ in G

Proof. Let $s = v_0, \ldots, v_K$ be the vertices for which visited is set to true, in this order. We prove: **for all i, there is a path $s \rightsquigarrow v_i$**, by induction.

Correctness

Claim

For all vertices v, if visited$[v]$ is true at the end, there is a path $s \rightsquigarrow v$ in G

Proof. Let $s = v_0, \ldots, v_K$ be the vertices for which visited is set to true, in this order. We prove: **for all i, there is a path $s \rightsquigarrow v_i$**, by induction.

- OK for $i = 0$
- suppose true for $v_0, \ldots, v_{i-1}$.

Correctness

Claim

For all vertices v, if visited$[v]$ is true at the end, there is a path $s \rightsquigarrow v$ in G

Proof. Let $s = v_0, \ldots, v_K$ be the vertices for which visited is set to true, in this order. We prove: **for all i, there is a path $s \rightsquigarrow v_i$**, by induction.

- OK for $i = 0$
- suppose true for $v_0, \ldots, v_{i-1}$.

  when visited$[v_i]$ is set to true, we are examining the neighbours of a certain v_j, $\boldsymbol{j < i}$.

Correctness

Claim

For all vertices v, if visited$[v]$ is true at the end, there is a path $s \rightsquigarrow v$ in G

Proof. Let $s = v_0, \dots, v_K$ be the vertices for which visited is set to true, in this order. We prove: **for all i, there is a path $s \rightsquigarrow v_i$**, by induction.

- OK for $i = 0$
- suppose true for $v_0, \dots, v_{i-1}$.

  when visited$[v_i]$ is set to true, we are examining the neighbours of a certain v_j, $\boldsymbol{j < i}$.

  by assumption, there is a path $s \rightsquigarrow v_j$

Correctness

Claim

For all vertices v, if visited$[v]$ is true at the end, there is a path $s \rightsquigarrow v$ in G

Proof. Let $s = v_0, \dots, v_K$ be the vertices for which visited is set to true, in this order. We prove: **for all i, there is a path $s \rightsquigarrow v_i$**, by induction.

- OK for $i = 0$
- suppose true for $v_0, \dots, v_{i-1}$.

  when visited$[v_i]$ is set to true, we are examining the neighbours of a certain v_j, $\boldsymbol{j < i}$.

  by assumption, there is a path $s \rightsquigarrow v_j$

  because $\{v_j, v_i\}$ is in E, there is a path $s \rightsquigarrow v_i$

Correctness

Claim

For all vertices v, if there is a path $s \rightsquigarrow v$ in G, $\textsf{visited}[v]$ is true at the end

Proof. Let $v_0 = s, \ldots, v_k = v$ be a path $s \rightsquigarrow v$. We prove $\textsf{visited}[v_i]$ is true for all i, by induction.

- $\textsf{visited}[v_0]$ is true
- if $\textsf{visited}[v_i]$ is true, we will examine all neighbours v of v_i

Correctness

Claim

For all vertices v, if there is a path $s \rightsquigarrow v$ in G, $\mathsf{visited}[v]$ is true at the end

Proof. Let $v_0 = s, \ldots, v_k = v$ be a path $s \rightsquigarrow v$. We prove $\mathsf{visited}[v_i]$ is true for all i, by induction.

- $\mathsf{visited}[v_0]$ is true
- if $\mathsf{visited}[v_i]$ is true, we will examine all neighbours v of v_i

  so at the end of Step 7, all $\mathsf{visited}[v]$ will be true, for v neighbour of v_i

Correctness

Claim

For all vertices v, if there is a path $s \rightsquigarrow v$ in G, $\mathsf{visited}[v]$ is true at the end

Proof. Let $v_0 = s, \ldots, v_k = v$ be a path $s \rightsquigarrow v$. We prove $\mathsf{visited}[v_i]$ is true for all i, by induction.

- $\mathsf{visited}[v_0]$ is true
- if $\mathsf{visited}[v_i]$ is true, we will examine all neighbours v of v_i

  so at the end of Step 7, all $\mathsf{visited}[v]$ will be true, for v neighbour of v_i

  in particular, $\mathsf{visited}[v_{i+1}]$ will be true

Correctness

Summary

For all vertices v, there is a path $s \rightsquigarrow v$ in G **if and only if** visited$[v]$ is true at the end

Applications

- testing if there is a path $s \rightsquigarrow v$
- testing if G is connected

in $\boldsymbol{O(n+m)}$.

Correctness

Summary

For all vertices v, there is a path $s \rightsquigarrow v$ in G **if and only if** visited$[v]$ is true at the end

Applications

- testing if there is a path $s \rightsquigarrow v$
- testing if G is connected

in $\boldsymbol{O(n+m)}$.

Exercise

For a connected graph, $m \geq n - 1$.

Keeping track of parents and levels

```
BFS(G, s)
    let Q be an empty queue
    let parent be an array of size n, with all entries set to NIL
    let level be an array of size n, with all entries set to ∞
    enqueue(s, Q)
    parent[s] ← s
    level[s] ← 0
    while Q not empty do
        v ← dequeue(Q)
        for all w neighbours of v do
            if parent[w] is NIL
                enqueue(w, Q)
                parent[w] ← v
                level[w] ← level[v] + 1
```

BFS tree

Definition: the **BFS tree** T is the subgraph made of:

- all w such that $\mathsf{parent}[w] \neq \textbf{NIL}$.
- all edges $\{w, \mathsf{parent}[w]\}$, for w as above (except $w = s$)

BFS tree

Definition: the **BFS tree** T is the subgraph made of:

- all w such that $\mathsf{parent}[w] \neq \mathbf{NIL}$.
- all edges $\{w, \mathsf{parent}[w]\}$, for w as above (except $w = s$)

Claim

The BFS tree T is a tree

BFS tree

Definition: the **BFS tree** T is the subgraph made of:

- all w such that $\mathsf{parent}[w] \neq$ **NIL**.
- all edges $\{w, \mathsf{parent}[w]\}$, for w as above (except $w = s$)

Claim

The BFS tree T is a tree

Proof: by induction on the vertices for which $\mathsf{parent}[v]$ is not **NIL**

- when we set $\mathsf{parent}[s] \leftarrow s$, only one vertex, no edge.
- suppose true before we set $\mathsf{parent}[w] \leftarrow v$

  v was in T before, w was not, so we add one vertex w and one edge $\{v, w\}$ to T

  so T remains a tree

BFS tree

Definition: the **BFS tree** T is the subgraph made of:

- all w such that $\mathsf{parent}[w] \neq \textbf{NIL}$.
- all edges $\{w, \mathsf{parent}[w]\}$, for w as above (except $w = s$)

Claim

The BFS tree T is a tree

Proof: by induction on the vertices for which $\mathsf{parent}[v]$ is not **NIL**

- when we set $\mathsf{parent}[s] \leftarrow s$, only one vertex, no edge.
- suppose true before we set $\mathsf{parent}[w] \leftarrow v$

  v was in T before, w was not, so we add one vertex w and one edge $\{v, w\}$ to T

  so T remains a tree

Remark: we make it a **rooted** tree by choosing s as root

Shortest paths from the BFS tree

Sub-claim 1

The levels in the queue are non-decreasing

Proof: by induction, they are always as $[\boldsymbol{\ell}, \ldots, \boldsymbol{\ell}]$ or as $[\boldsymbol{\ell}, \ldots, \boldsymbol{\ell}, \boldsymbol{\ell} + \mathbf{1}, \ldots, \boldsymbol{\ell} + \mathbf{1}]$

Shortest paths from the BFS tree

Sub-claim 1

The levels in the queue are non-decreasing

Proof: by induction, they are always as $[\boldsymbol{\ell}, \ldots, \boldsymbol{\ell}]$ or as $[\boldsymbol{\ell}, \ldots, \boldsymbol{\ell}, \boldsymbol{\ell} + \mathbf{1}, \ldots, \boldsymbol{\ell} + \mathbf{1}]$

Sub-claim 2

For all vertices u, v, if there is an edge $\{u, v\}$, then $\textsf{level}[v] \leq \textsf{level}[u] + 1$.

Shortest paths from the BFS tree

Sub-claim 1

The levels in the queue are non-decreasing

Proof: by induction, they are always as $[\boldsymbol{\ell}, \dots, \boldsymbol{\ell}]$ or as $[\boldsymbol{\ell}, \dots, \boldsymbol{\ell}, \boldsymbol{\ell} + \mathbf{1}, \dots, \boldsymbol{\ell} + \mathbf{1}]$

Sub-claim 2

For all vertices u, v, if there is an edge $\{u, v\}$, then $\mathsf{level}[v] \leq \mathsf{level}[u] + 1$.

Proof: when we dequeue u,

- either we already saw the parent of v
- or u becomes the parent of v

Shortest paths from the BFS tree

Sub-claim 1

The levels in the queue are non-decreasing

Proof: by induction, they are always as $[\boldsymbol{\ell}, \ldots, \boldsymbol{\ell}]$ or as $[\boldsymbol{\ell}, \ldots, \boldsymbol{\ell}, \boldsymbol{\ell}+\mathbf{1}, \ldots, \boldsymbol{\ell}+\mathbf{1}]$

Sub-claim 2

For all vertices u, v, if there is an edge $\{u, v\}$, then $\mathsf{level}[v] \leq \mathsf{level}[u] + 1$.

Proof: when we dequeue u,

- either we already saw the parent of v $\quad \mathsf{level}[\mathsf{parent}[v]] \leq \mathsf{level}[u]$
- or u becomes the parent of v

Shortest paths from the BFS tree

Sub-claim 1

The levels in the queue are non-decreasing

Proof: by induction, they are always as $[\boldsymbol{\ell}, \ldots, \boldsymbol{\ell}]$ or as $[\boldsymbol{\ell}, \ldots, \boldsymbol{\ell}, \boldsymbol{\ell}+\mathbf{1}, \ldots, \boldsymbol{\ell}+\mathbf{1}]$

Sub-claim 2

For all vertices u, v, if there is an edge $\{u, v\}$, then $\mathsf{level}[v] \leq \mathsf{level}[u] + 1$.

Proof: when we dequeue u,

- either we already saw the parent of v $\qquad \mathsf{level}[\mathsf{parent}[v]] \leq \mathsf{level}[u]$
- or u becomes the parent of v $\qquad \mathsf{level}[\mathsf{parent}[v]] = \mathsf{level}[u]$

Shortest paths from the BFS tree

Sub-claim 1

The levels in the queue are non-decreasing

Proof: by induction, they are always as $[\boldsymbol{\ell}, \ldots, \boldsymbol{\ell}]$ or as $[\boldsymbol{\ell}, \ldots, \boldsymbol{\ell}, \boldsymbol{\ell} + \mathbf{1}, \ldots, \boldsymbol{\ell} + \mathbf{1}]$

Sub-claim 2

For all vertices u, v, if there is an edge $\{u, v\}$, then $\mathsf{level}[v] \leq \mathsf{level}[u] + 1$.

Proof: when we dequeue u,

- either we already saw the parent of v — $\mathsf{level}[\mathsf{parent}[v]] \leq \mathsf{level}[u]$
- or u becomes the parent of v — $\mathsf{level}[\mathsf{parent}[v]] = \mathsf{level}[u]$
- but $\mathsf{level}[\mathsf{parent}[v]] = \mathsf{level}[v] - 1$

Shortest paths from the BFS tree

Claim

For all v in G:

- there is a path $s \rightsquigarrow v$ in G iff there is a path $s \rightsquigarrow v$ in T
- if so, the path in T is a shortest path and $\mathsf{level}[v] = \mathrm{dist}(s, v)$

Shortest paths from the BFS tree

Claim

For all v in G:

- there is a path $s \rightsquigarrow v$ in G iff there is a path $s \rightsquigarrow v$ in T
- if so, the path in T is a shortest path and $\textsf{level}[v] = \text{dist}(s, v)$

Proof. First item: $\Longrightarrow$ was proved before

Shortest paths from the BFS tree

Claim

For all v in G:

- there is a path $s \rightsquigarrow v$ in G iff there is a path $s \rightsquigarrow v$ in T
- if so, the path in T is a shortest path and $\mathsf{level}[v] = \mathrm{dist}(s, v)$

Proof. First item: $\implies$ was proved before, $\impliedby$ obvious.

Shortest paths from the BFS tree

Claim

For all v in G:

- there is a path $s \rightsquigarrow v$ in G iff there is a path $s \rightsquigarrow v$ in T
- if so, the path in T is a shortest path and $\mathsf{level}[v] = \mathrm{dist}(s, v)$

Proof. First item: $\implies$ was proved before, $\impliedby$ obvious.

Second item:

- $\mathrm{dist}(s, v) \leq \mathsf{level}[v]$ (follow the path on T)

Shortest paths from the BFS tree

Claim

For all v in G:

- there is a path $s \rightsquigarrow v$ in G iff there is a path $s \rightsquigarrow v$ in T
- if so, the path in T is a shortest path and $\mathsf{level}[v] = \mathrm{dist}(s, v)$

Proof. First item: $\implies$ was proved before, $\impliedby$ obvious.

Second item:

- $\mathrm{dist}(s, v) \leq \mathsf{level}[v]$ (follow the path on T)
- take the **shortest** path $\boldsymbol{s = v_0 \to v_1 \to v_2 \to \cdots \to v_k = v}$ $\quad k = \mathrm{dist}(s, v)$

Shortest paths from the BFS tree

Claim

For all v in G:

- there is a path $s \rightsquigarrow v$ in G iff there is a path $s \rightsquigarrow v$ in T
- if so, the path in T is a shortest path and $\mathsf{level}[v] = \mathrm{dist}(s, v)$

Proof. First item: $\implies$ was proved before, $\impliedby$ obvious.

Second item:

- $\mathrm{dist}(s, v) \leq \mathsf{level}[v]$ (follow the path on T)
- take the **shortest** path $\boldsymbol{s = v_0 \to v_1 \to v_2 \to \cdots \to v_k = v}$ $\quad k = \mathrm{dist}(s, v)$

  $\mathsf{level}[v_0] = 0$

Shortest paths from the BFS tree

Claim

For all v in G:

- there is a path $s \rightsquigarrow v$ in G iff there is a path $s \rightsquigarrow v$ in T
- if so, the path in T is a shortest path and $\mathsf{level}[v] = \mathrm{dist}(s, v)$

Proof. First item: $\implies$ was proved before, $\impliedby$ obvious.

Second item:

- $\mathrm{dist}(s, v) \leq \mathsf{level}[v]$ (follow the path on T)
- take the **shortest** path $\boldsymbol{s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k = v}$ $\qquad k = \mathrm{dist}(s, v)$

  $\mathsf{level}[v_0] = 0$

  so $\mathsf{level}[v_1] \leq 1$ $\qquad$ sub-claim 2

Shortest paths from the BFS tree

Claim

For all v in G:

- there is a path $s \rightsquigarrow v$ in G iff there is a path $s \rightsquigarrow v$ in T
- if so, the path in T is a shortest path and $\textsf{level}[v] = \text{dist}(s, v)$

Proof. First item: $\implies$ was proved before, $\impliedby$ obvious.

Second item:

- $\text{dist}(s, v) \le \textsf{level}[v]$ (follow the path on T)
- take the **shortest** path $\boldsymbol{s = v_0 \to v_1 \to v_2 \to \cdots \to v_k = v}$ $\qquad k = \text{dist}(s, v)$

  $\textsf{level}[v_0] = 0$

  so $\textsf{level}[v_1] \le 1$ $\qquad$ sub-claim 2

  so $\textsf{level}[v_2] \le 2$ $\qquad$ sub-claim 2

Shortest paths from the BFS tree

Claim

For all v in G:

- there is a path $s \rightsquigarrow v$ in G iff there is a path $s \rightsquigarrow v$ in T
- if so, the path in T is a shortest path and $\mathsf{level}[v] = \mathrm{dist}(s, v)$

Proof. First item: $\implies$ was proved before, $\impliedby$ obvious.

Second item:

- $\mathrm{dist}(s, v) \leq \mathsf{level}[v]$ (follow the path on T)
- take the **shortest** path $\boldsymbol{s = v_0 \to v_1 \to v_2 \to \cdots \to v_k = v}$ $\quad k = \mathrm{dist}(s, v)$

  $\mathsf{level}[v_0] = 0$

  so $\mathsf{level}[v_1] \leq 1$ $\quad$ sub-claim 2

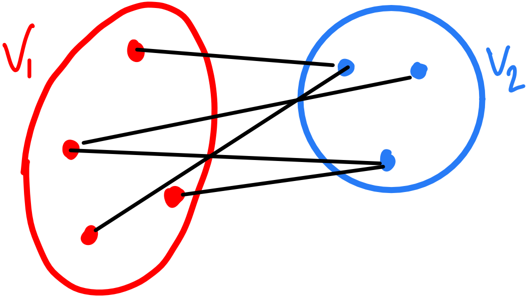
  so $\mathsf{level}[v_2] \leq 2$ $\quad$ sub-claim 2

  $\ldots$ so $\mathsf{level}[v_k] \leq k = \mathrm{dist}(s, v)$ $\quad$ sub-claim 2

Bipartite graphs

Definition

- a graph $G = (V, E)$ is **bipartite** if there is a partition $V = \boldsymbol{V_1} \cup \boldsymbol{V_2}$ such that all edges have one end in V_1 and one end in V_2.



Using BFS to test bipartite-ness

Claim.

Suppose G connected, run BFS from any s, and set

- V_1 = vertices with odd level
- V_2 = vertices with even level.

Then G is bipartite if and only all edges have one end in V_1 and one end in V_2 (testable in $O(n+m)$)

Using BFS to test bipartite-ness

Claim.

Suppose G connected, run BFS from any s, and set

- V_1 = vertices with odd level
- V_2 = vertices with even level.

Then G is bipartite if and only all edges have one end in V_1 and one end in V_2 (testable in $\boldsymbol{O(m)}$)

Using BFS to test bipartite-ness

Claim.

Suppose G connected, run BFS from any s, and set

- V_1 = vertices with odd level
- V_2 = vertices with even level.

Then G is bipartite if and only all edges have one end in V_1 and one end in V_2 (testable in $O(m)$)

Proof. $\Longleftarrow$ obvious.

Using BFS to test bipartite-ness

Claim.

Suppose G connected, run BFS from any s, and set

- V_1 = vertices with odd level
- V_2 = vertices with even level.

Then G is bipartite if and only all edges have one end in V_1 and one end in V_2 (testable in $O(m)$)

Proof. $\Longleftarrow$ obvious.

For $\Longrightarrow$, let W_1, W_2 be a bipartition. Because paths alternate between W_1, W_2:

- V_1 (= vertices with odd level) is included in W_1 (say)
- V_2 (= vertices with even level) is included in W_2

So $V_1 = W_1$ and $V_2 = W_2$.

Computing the connected components

Idea: add an outer loop that runs BFS on successive vertices

Exercise

Fill in the details.

Complexity:

- each pass of BFS $O(n_i + m_i)$, if $G_i(V_i, E_i)$ is the ith connected component
- total $O(n + m)$

Depth-first search

Going depth-first

The idea:

- travel as deep as possible, as long as you can
- when you can't go further, backtrack.

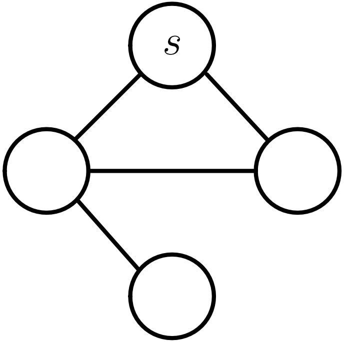
DFS implementations are based on stacks, either implicitly (recursion) or explicitly (as with queues for BFS).

Going depth-first

The idea:

- travel as deep as possible, as long as you can
- when you can't go further, backtrack.

DFS implementations are based on stacks, either implicitly (recursion) or explicitly (as with queues for BFS).

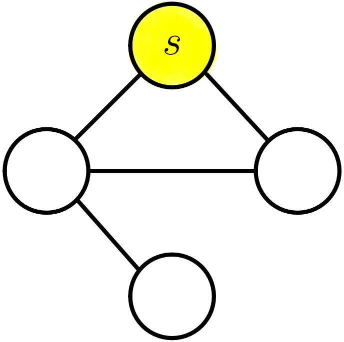


Going depth-first

The idea:

- travel as deep as possible, as long as you can
- when you can't go further, backtrack.

DFS implementations are based on stacks, either implicitly (recursion) or explicitly (as with queues for BFS).

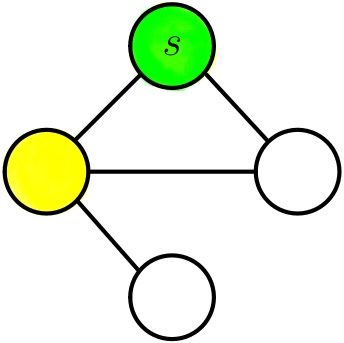


Going depth-first

The idea:

- travel as deep as possible, as long as you can
- when you can't go further, backtrack.

DFS implementations are based on stacks, either implicitly (recursion) or explicitly (as with queues for BFS).

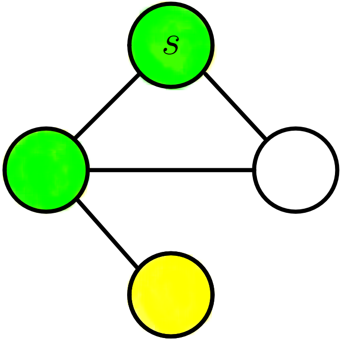


Going depth-first

The idea:

- travel as deep as possible, as long as you can
- when you can't go further, backtrack.

DFS implementations are based on stacks, either implicitly (recursion) or explicitly (as with queues for BFS).

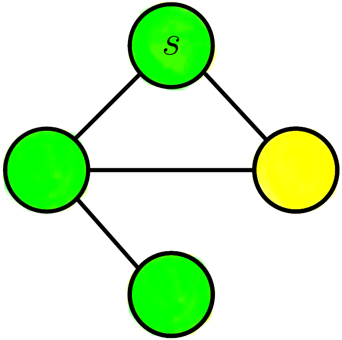


Going depth-first

The idea:

- travel as deep as possible, as long as you can
- when you can't go further, backtrack.

DFS implementations are based on stacks, either implicitly (recursion) or explicitly (as with queues for BFS).

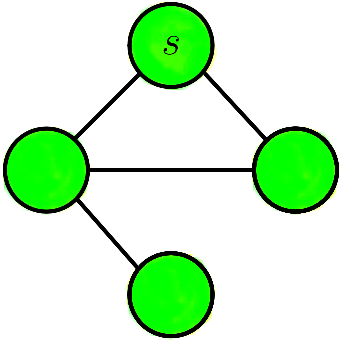


Going depth-first

The idea:

- travel as deep as possible, as long as you can
- when you can't go further, backtrack.

DFS implementations are based on stacks, either implicitly (recursion) or explicitly (as with queues for BFS).



Recursive algorithm

DFS(G)
G: a graph with n vertices, given by adjacency lists
1. let visited be an array of size n, with all entries set to **false**
2. **for all** v in G
3. **if** visited$[v]$ is **false**
4. **explore**(v)

explore(v)
1. visited$[v]$ = **true**
2. **for all** w neighbour of v **do**
3. **if** visited$[w]$ = **false**
4. **explore**(w)

Remark: can add parent array as in BFS

The white path lemma

Claim

When we start exploring a vertex v, any w that can be connected to v by an **unvisited** path will be visited **explore**(v) is finished.

The white path lemma

Claim

When we start exploring a vertex v, any w that can be connected to v by an **unvisited** path will be visited **explore**(v) is finished.

Proof. Let $\boldsymbol{v_0 = v, \ldots, v_k = w}$ be a path $v \rightsquigarrow w$, with $v_1, \ldots, v_k$ not visited yet.
We prove: all v_i's are visited before **explore**(v) is finished.

The white path lemma

Claim

When we start exploring a vertex v, any w that can be connected to v by an **unvisited** path will be visited **explore**(v) is finished.

Proof. Let $\boldsymbol{v_0 = v, \ldots, v_k = w}$ be a path $v \rightsquigarrow w$, with $v_1, \ldots, v_k$ not visited yet.
We prove: all v_i's are visited before **explore**(v) is finished.

True for $i = 0$.

The white path lemma

Claim

When we start exploring a vertex v, any w that can be connected to v by an **unvisited** path will be visited **explore**(v) is finished.

Proof. Let $\boldsymbol{v_0 = v, \ldots, v_k = w}$ be a path $v \rightsquigarrow w$, with $v_1, \ldots, v_k$ not visited yet.
We prove: all v_i's are visited before **explore**(v) is finished.

True for $i = 0$.

Suppose true for $i < k$.

The white path lemma

Claim

When we start exploring a vertex v, any w that can be connected to v by an **unvisited** path will be visited **explore**(v) is finished.

Proof. Let $\boldsymbol{v_0 = v, \ldots, v_k = w}$ be a path $v \rightsquigarrow w$, with $v_1, \ldots, v_k$ not visited yet.
We prove: all v_i's are visited before **explore**(v) is finished.

True for $i = 0$.

Suppose true for $i < k$. When we visit v_i, **explore**(v) is not finished, and v_{i+1} is one of its neighbours.

The white path lemma

Claim

When we start exploring a vertex v, any w that can be connected to v by an **unvisited** path will be visited **explore**(v) is finished.

Proof. Let $\boldsymbol{v_0 = v, \ldots, v_k = w}$ be a path $v \rightsquigarrow w$, with $v_1, \ldots, v_k$ not visited yet.
We prove: all v_i's are visited before **explore**(v) is finished.

True for $i = 0$.

Suppose true for $i < k$. When we visit v_i, **explore**(v) is not finished, and v_{i+1} is one of its neighbours.

- **if** visited$[\boldsymbol{v_{i+1}}]$ **is true when we reach Step 3**
  OK: it means we visited it

The white path lemma

Claim

When we start exploring a vertex v, any w that can be connected to v by an **unvisited** path will be visited **explore**(v) is finished.

Proof. Let $\boldsymbol{v_0 = v, \ldots, v_k = w}$ be a path $v \rightsquigarrow w$, with $v_1, \ldots, v_k$ not visited yet.
We prove: all v_i's are visited before **explore**(v) is finished.

True for $i = 0$.

Suppose true for $i < k$. When we visit v_i, **explore**(v) is not finished, and v_{i+1} is one of its neighbours.

- **if** visited$[\boldsymbol{v_{i+1}}]$ **is true when we reach Step 3**
  OK: it means we visited it
- **else, we will visit it just now**
  OK: it will be done before **explore**(v) is finished

Another basic property

Claim

If w is visited during **explore**(v), there is a path $\boldsymbol{v} \leadsto \boldsymbol{w}$.

Another basic property

Claim

If w is visited during **explore**(v), there is a path $\boldsymbol{v} \rightsquigarrow \boldsymbol{w}$.

Proof. Same as for BFS.

Consequences

Previous properties: after we call **explore** at $v_1, \ldots, v_k$ in **DFS**, we have visited exactly the connected components containing $v_1, \ldots, v_k$

Consequences

Previous properties: after we call **explore** at $v_1, \ldots, v_k$ in **DFS**, we have visited exactly the connected components containing $v_1, \ldots, v_k$

Shortest paths: no

Runtime: still $\boldsymbol{O(n+m)}$

Iterative version?

```
explore(s)
    let S be an empty stack
    push(s, S)
    visited[s] ← true
    while S not empty do
        v ← pop(S)
        for all w neighbours of v do
            if visited[w] is false
                push(w, S)
                visited[w] ← true
```

Still depth-first?

Iterative version?

```
explore(s)
    let S be an empty stack
    push(s, S)
    visited[s] ← true
    while S not empty do
        v ← pop(S)
        for all w neighbours of v do
            if visited[w] is false
                push(w, S)
                visited[w] ← true
```

Still depth-first?

Exercise: fix this.

Trees, forest, ancestors and descendants

Previous observation:

- DFS(G) gives a partition of G into vertex-disjoint rooted trees $T_1, \ldots, T_k$ (DFS forest)

Definition. Suppose the DFS forest is $T_1, \ldots, T_k$ and let u, v be two vertices

- u is an **ancestor** of v if they are on the same T_i and u is on the path root $\rightsquigarrow v$
- equivalent: v is a **descendant** of u

Key property

Claim

All edges in G connect a vertex to one of its descendants or ancestors.

Key property

Claim

All edges in G connect a vertex to one of its descendants or ancestors.

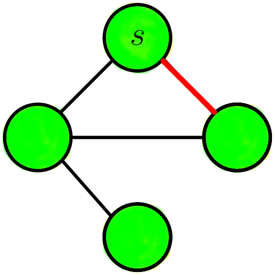
Proof. Let $\{v, w\}$ be an edge, and suppose we visit v first.

Then when we visit v, (v, w) is an unvisited path between v and w, so w will become a descendant of v (white path lemma)

Back edges

Definition.

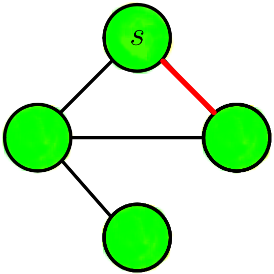
- a **back edge** is an edge in G connecting an ancestor to a descendant, which is **not** a tree edge.



Back edges

Definition.

- a **back edge** is an edge in G connecting an ancestor to a descendant, which is **not** a tree edge.



Observation

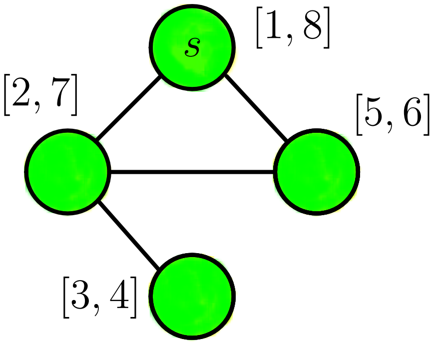
All edges are either **tree edges** or **back edges** (key property).

Start and finish times

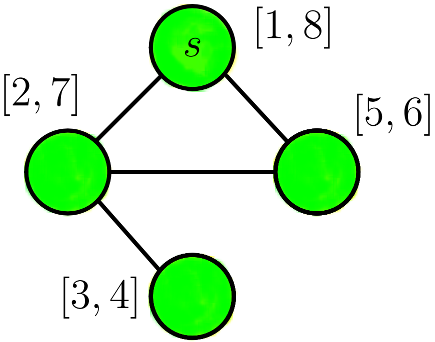
Set a variable t to 1 initially, create two arrays start and finish, and change **explore**:

```
explore(v)
    visited[v] = true
    start[v] = t
    t++
    for all w neighbour of v do
        if visited[w] = false
            explore(w)
    finish[v] = t
    t++
```

Example



Example



Observation:

- these intervals are either contained in one another, or disjoint

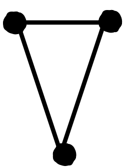
Cut vertices

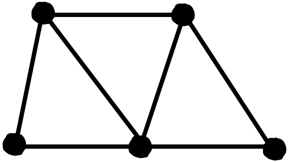
Biconnectivity

Definition: $G = (V, E)$ biconnected if

- G is connected
- G stays connected if we remove any vertex (and all edges that contain it)

Two biconnected graphs:

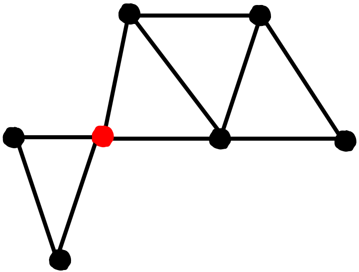




Cut vertices

Definition: for G connected, a vertex v in G is a cut vertex if removing v (and all edges that contain it) makes G disconnected.

Also called articulation points

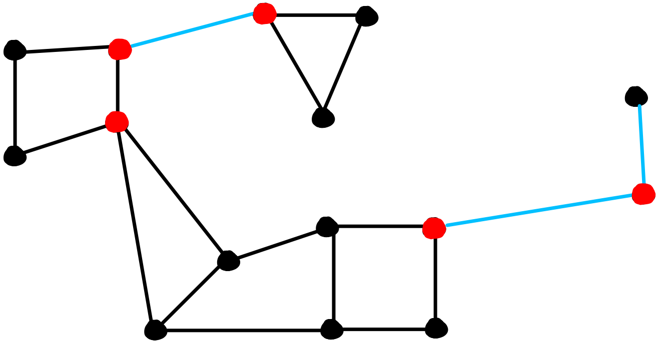


biconnected $\iff$ no cut vertex

Aside: the shape of a connected undirected graph

Call biconnected component a biconnected subgraph that is not contained in a larger one

Then G can be seen as a tree of biconnected components connected at cut vertices



Blue edges are **cut edges:** removing them makes the graph disconnected

Finding the cut vertices (G connected)

Setup: we start from a rooted DFS tree T, knowing parent and level.

Warm-up

The root s is a cut vertex if and only if it has more than one child.

Proof.

- if s has one child, removing s leaves T connected. So s not a cut vertex.

Finding the cut vertices (G connected)

Setup: we start from a rooted DFS tree T, knowing parent and level.

Warm-up

The root s is a cut vertex if and only if it has more than one child.

Proof.

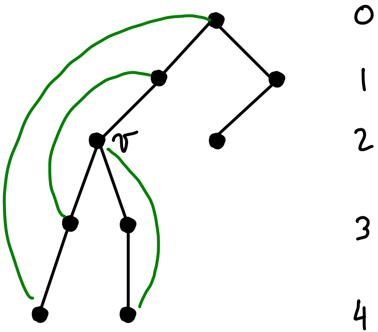
- if s has one child, removing s leaves T connected. So s not a cut vertex.
- suppose s has subtrees $S_1, \ldots, S_k$, $k > 1$.

  Key property: no edge connecting S_i to S_j for $i \neq j$. So removing s creates k connected components.

Finding the cut vertices (G connected)

Definition: for a vertex v, let

- $a(v) = \min\{\mathsf{level}[w],\ \{v, w\} \text{ edge}\}$
- $m(v) = \min\{a(w),\ w \text{ descendant of } v\}$



$a(v) = 1$

$m(v) = 0$

Using the values $m(v)$

Claim

For any v (except the root), v is a cut vertex if and only if it has a child w with $m(w) \geq \text{level}[v]$.

Proof

- Take a child w of v, let T_w be the subtree at w. Let also T_v be the subtree at v.

Using the values $m(v)$

Claim

For any v (except the root), v is a cut vertex if and only if it has a child w with $m(w) \geq \mathsf{level}[v]$.

Proof

- Take a child w of v, let T_w be the subtree at w. Let also T_v be the subtree at v.
- If $m(w) < \mathsf{level}[v]$, then there is an edge from T_w to a vertex above v. After removing v, T_w remains connected to the root.

Using the values $m(v)$

Claim

For any v (except the root), v is a cut vertex if and only if it has a child w with $m(w) \geq \text{level}[v]$.

Proof

- Take a child w of v, let T_w be the subtree at w. Let also T_v be the subtree at v.
- If $m(w) < \text{level}[v]$, then there is an edge from T_w to a vertex above v. After removing v, T_w remains connected to the root.
- If $m(w) \geq \text{level}[v]$, then all edges originating from T_w end in T_v.

Using the values $m(v)$

Claim

For any v (except the root), v is a cut vertex if and only if it has a child w with $m(w) \geq \mathsf{level}[v]$.

Proof

- Take a child w of v, let T_w be the subtree at w. Let also T_v be the subtree at v.
- If $m(w) < \mathsf{level}[v]$, then there is an edge from T_w to a vertex above v. After removing v, T_w remains connected to the root.
- If $m(w) \geq \mathsf{level}[v]$, then all edges originating from T_w end in T_v.

  Proof: any edge originating from a vertex x in T_w ends at a level at least $\mathsf{level}[v]$, and connects x to one of its ancestors or descendants (key property)

Using the values $m(v)$

Claim

For any v (except the root), v is a cut vertex if and only if it has a child w with $m(w) \geq \text{level}[v]$.

Proof

- Take a child w of v, let T_w be the subtree at w. Let also T_v be the subtree at v.
- If $m(w) < \text{level}[v]$, then there is an edge from T_w to a vertex above v. After removing v, T_w remains connected to the root.
- If $m(w) \geq \text{level}[v]$, then all edges originating from T_w end in T_v.

  Proof: any edge originating from a vertex x in T_w ends at a level at least $\text{level}[v]$, and connects x to one of its ancestors or descendants (key property)

  So after removing v, T_w is disconnected from the root (except if v is the root)

Computing the values $m(v)$

Observation:

- if v has children $\boldsymbol{w_1, \ldots, w_k}$, then $\boldsymbol{m(v) = \min\{a(v), m(w_1), \ldots, m(w_k)\}}$

Computing the values $m(v)$

Observation:

- if v has children $w_1, \dots, w_k$, then $m(v) = \min\{a(v), m(w_1), \dots, m(w_k)\}$

Consequence:

- computing $a(v)$ is $O(d_v)$ — d_v = degree of v
- knowing all $m(w_1), \dots, m(w_k)$, we get $m(v)$ in $O(d_v)$
- so all values $m(v)$ can be computed in $O(m)$
  (remember $O(n+m) = O(m)$ when G connected)

Computing the values $m(v)$

Observation:

- if v has children $w_1, \ldots, w_k$, then $m(v) = \min\{a(v), m(w_1), \ldots, m(w_k)\}$

Consequence:

- computing $a(v)$ is $O(d_v)$ — $d_v =$ degree of v
- knowing all $m(w_1), \ldots, m(w_k)$, we get $m(v)$ in $O(d_v)$
- so all values $m(v)$ can be computed in $O(m)$
  (remember $O(n + m) = O(m)$ when G connected)

testing the cut-vertex condition at v is $O(d_v)$, total $O(m)$

Exercise

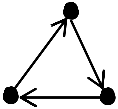
write the pseudo-code

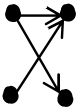
Directed graphs

Directed graphs basics

Definition:

- $G = (V, E)$ as in the undirected case, with the difference that edges are (directed) pairs (v, w)
  - edges also called arcs
  - usually, we allow loops, with $v = w$
  - v is the source node, w is the target
- a path is a sequence $v_1, \ldots, v_k$ of vertices, with (v_i, v_{i+1}) in E for all i. $k = 1$ is OK.
- a cycle is a path $v_1, \ldots, v_k, v_1$, $\boldsymbol{k \geq 1}$
- a directed acyclic graph (DAG) is a directed graph with no cycle





Directed graphs basics

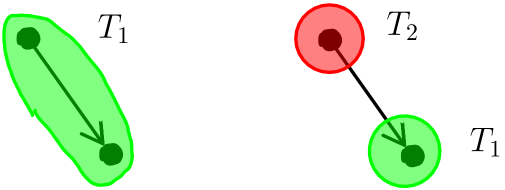
Data structures

- adjacency lists
- adjacency matrix (not symmetric anymore)

BFS and DFS for directed graphs

The algorithms work without any change. We will focus on DFS. Still true:

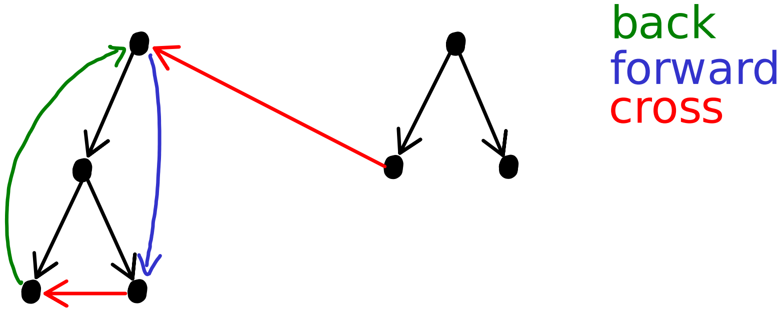
- we obtain a partition of V into vertex-disjoint trees $T_1, \ldots, T_k$
- when we start exploring a vertex v, any w with an unvisited path $v \rightsquigarrow w$ becomes a descendant of v (white path lemma)
- properties of start and finish times
- but there can exist edges connecting the trees T_i



Classification of edges

Suppose we have a DFS forest. Edges of G are one of the following:

- **tree edges**
- **back edges:** from descendant to ancestor
- **forward edges:** from ancestor to descendant (but not tree edge)
- **cross edges:** all others



(depends on the order of vertices we chose in the main DFS loop)

Classification of edges

explore(v)
1. visited$[v]$ = **true**
2. start$[v] = t$, t++
3. **for all** w neighbour of v **do**
4. **if** visited$[w]$ = **false**
5. **explore**(w) (v, w) **tree edge**
6. finish$[v] = t$, t++

If w was visited:

- if w not finished, (v, w) **back edge**
- else if **start**$[v]$ < **start**$[w]$ < **finish**$[w]$, (v, w) **forward edge**
- else, **start**$[w]$ < **finish**$[w]$ < **start**$[v]$, (v, w) **cross edge**

Testing acyclicity

Claim

G has a cycle if and only if there is a back edge in the DFS forest

Proof

- Suppose there is a back edge (v, w). Then v is a descendant of w, so there is a path $w \rightsquigarrow v$, and a cycle $w \rightsquigarrow v \rightarrow w$
- Suppose there is a cycle $\boldsymbol{v_1, \ldots, v_{k-1}, v_k = v_1}$. Up to renumbering, assume we find v_1 first in the DFS.

  Starting from v_1, we will reach v_{k-1} (white path lemma). We check the edge (v_{k-1}, v_1), and v_1 is not finished. So back edge.

Consequence: acyclicity test in $\boldsymbol{O(n+m)}$

Strong connectivity

Definition. A directed graph G is strongly connected if for all v, w in G, there is a path $\boldsymbol{v} \rightsquigarrow \boldsymbol{w}$ (and thus a path $\boldsymbol{w} \rightsquigarrow \boldsymbol{v}$).

Strong connectivity

Definition. A directed graph G is strongly connected if for all v, w in G, there is a path $\boldsymbol{v} \leadsto \boldsymbol{w}$ (and thus a path $\boldsymbol{w} \leadsto \boldsymbol{v}$).

Algorithm:

- call **explore twice**, starting from a same vertex s
- edges reversed the second time

Strong connectivity

Definition. A directed graph G is strongly connected if for all v, w in G, there is a path $\boldsymbol{v \leadsto w}$ (and thus a path $\boldsymbol{w \leadsto v}$).

Algorithm:

- call **explore twice**, starting from a same vertex s
- edges reversed the second time

Correctness:

- first run tells whether for all v, there is a path $s \leadsto v$
- second one tells whether for all v, there is a path $s \leadsto v$ in the reverse graph (which is a path $v \leadsto s$ in G)

Consequence: test in $\boldsymbol{O(n+m)}$

Structure of directed graphs

Definition: a **strongly connected component** of G is

- a subgraph of G
- which is strongly connected
- but not contained in a larger strongly connected subgraph of G.

Exercise

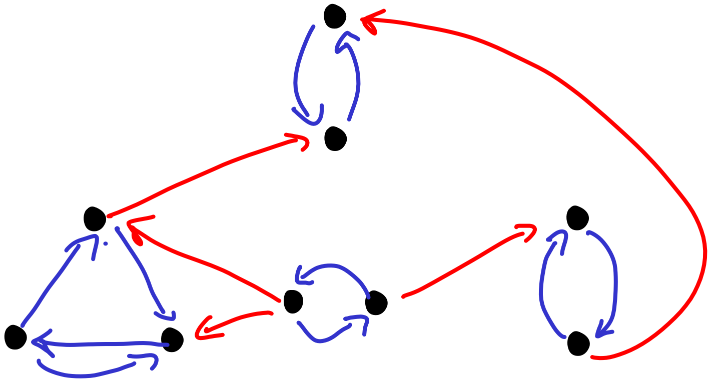
The vertices of strongly connected components form a partition of V.

Exercise

v and w are in the same strongly connected component if and only if there are paths $v \rightsquigarrow w$ and $w \rightsquigarrow v$.

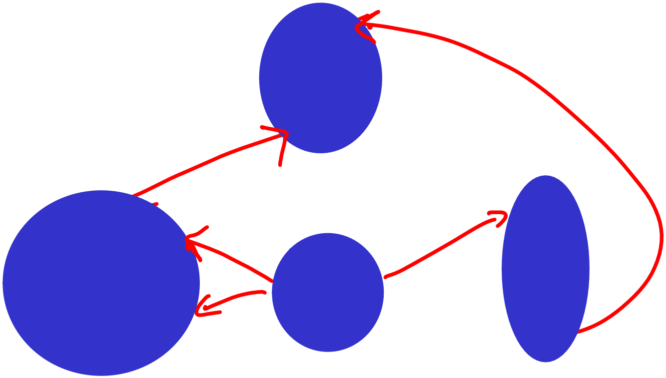
Structure of directed graphs

A directed graph G can be seen as a DAG of disjoint strongly connected components.



Structure of directed graphs

A directed graph G can be seen as a DAG of disjoint strongly connected components.



Kosaraju's algorithm for strongly connected components

Definition: for a directed graph $G = (V, E)$, the **reverse** (or **transpose**) graph $G^T = (V, E^T)$ is the graph with same vertices, and reversed edges.

SCC(G)
1. run a DFS on G and record finish times
2. run a DFS on G^T, with vertices ordered in **decreasing finish time**
3. return the trees in the DFS forest of G^T

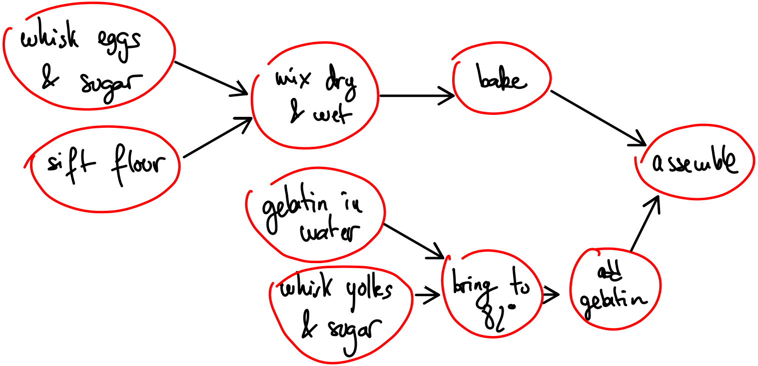
Complexity: $\boldsymbol{O(n+m)}$ (don't forget the time to reverse G)

Exercise

check that the strongly connected components of G and G^T are the same

Topological ordering

Definition: Suppose $G = (V, E)$ is a DAG. A **topological order** is an ordering $<$ of V such that for any edge (v, w), we have $v < w$.



No such order if there are cycles.

From a DFS forest

Observation:

- start times do not help
- finish times do, but we have to reverse their order

From a DFS forest

Claim

Suppose that V is ordered using the reverse of the finishing order: $v < w \iff \textsf{finish}[w] < \textsf{finish}[v]$.

This is a topological order.

Proof. Have to prove: for any edge (v, w), $\textsf{finish}[w] < \textsf{finish}[v]$.

- if we discover v before w, w will become a descendant of v (white path lemma), and we finish exploring it before we finish v.
- if we discover w before v, because there is no path $w \rightsquigarrow v$ (G is a DAG), we will finish w before we start v.

Consequence: topological order in $O(n + m)$.

Kosaraju's algorithm: proof of correctness (bonus material)

Correctness 1/3

Want to prove: for any vertices v, w, the following are equivalent.

(1) v and w and in the same strongly connected component of G

(2) v and w and in the same tree in the DFS forest of G^T (with vertices ordered in decreasing finish time)

Correctness 1/3

Want to prove: for any vertices v, w, the following are equivalent.

(1) v and w and in the same strongly connected component of G

(2) v and w and in the same tree in the DFS forest of G^T (with vertices ordered in decreasing finish time)

Proof of $1 \implies 2$ (order of the vertices does not matter here)
Let C be the strongly connected component of G that contains v and w

Correctness 1/3

Want to prove: for any vertices v, w, the following are equivalent.

(1) v and w and in the same strongly connected component of G

(2) v and w and in the same tree in the DFS forest of G^T (with vertices ordered in decreasing finish time)

Proof of $1 \implies 2$ (order of the vertices does not matter here)

Let C be the strongly connected component of G that contains v and w

Let s be the first vertex of C that we visit in the DFS of G^T

- there is a path $s \rightsquigarrow v$ in G^T
- all vertices on this path are in C (easy)
- so they are all unvisited when we arrive at s
- so v becomes a descendant of s **white path lemma**
- same for w

Correctness 2/3

Proof of $2 \implies 1$.
Let T be the tree in the DFS forest of G^T that contains v and w, with root s

We prove that for every vertex t in T, s **and** t **are in the same strongly connected component of** G**.**

Correctness 2/3

Proof of $2 \implies 1$.
Let T be the tree in the DFS forest of G^T that contains v and w, with root s

We prove that for every vertex t in T, s **and** t **are in the same strongly connected component of** G**.**

(1) for all t in T, there is a path $\boldsymbol{s \rightsquigarrow t}$ in G^T, so there is a path $\boldsymbol{t \rightsquigarrow s}$ in G

Correctness 2/3

Proof of $2 \implies 1$.

Let T be the tree in the DFS forest of G^T that contains v and w, with root s

We prove that for every vertex t in T, s **and** t **are in the same strongly connected component of** G**.**

(1) for all t in T, there is a path $\boldsymbol{s \rightsquigarrow t}$ in G^T, so there is a path $\boldsymbol{t \rightsquigarrow s}$ in G

(2) now we prove: for all t in T, t is a descendant of s in the DFS forest of G (this gives a path $\boldsymbol{s \rightsquigarrow t}$ in G)

Correctness 3/3

Want to prove: for all t in T, t is a descendant of s in the DFS forest of G.

Correctness 3/3

Want to prove: for all t in T, t is a descendant of s in the DFS forest of G.

By induction: suppose it is true for some t in T, and prove it is true for its children.

So let u be a child of t in T.

Correctness 3/3

Want to prove: for all t in T, t is a descendant of s in the DFS forest of G.

By induction: suppose it is true for some t in T, and prove it is true for its children. So let u be a child of t in T.

- $\mathsf{start}[s] \leq \mathsf{start}[t] < \mathsf{finish}[t] \leq \mathsf{finish}[s]$ induction assumption
- by definition of s, $\mathsf{finish}[u] < \mathsf{finish}[s]$

Correctness 3/3

Want to prove: for all t in T, t is a descendant of s in the DFS forest of G.

By induction: suppose it is true for some t in T, and prove it is true for its children. So let u be a child of t in T.

- $\mathsf{start}[s] \leq \mathsf{start}[t] < \mathsf{finish}[t] \leq \mathsf{finish}[s]$ induction assumption
- by definition of s, $\mathsf{finish}[u] < \mathsf{finish}[s]$, so our options are
  (1) $\mathsf{start}[s] < \mathsf{start}[u] < \mathsf{finish}[u] < \mathsf{finish}[s]$ [( )]
  (2) $\mathsf{start}[u] < \mathsf{finish}[u] < \mathsf{start}[s] < \mathsf{finish}[s]$ ( ) [ ]

Correctness 3/3

Want to prove: for all t in T, t is a descendant of s in the DFS forest of G.

By induction: suppose it is true for some t in T, and prove it is true for its children. So let u be a child of t in T.

- $\mathsf{start}[s] \leq \mathsf{start}[t] < \mathsf{finish}[t] \leq \mathsf{finish}[s]$ induction assumption
- by definition of s, $\mathsf{finish}[u] < \mathsf{finish}[s]$, so our options are
  (1) $\mathsf{start}[s] < \mathsf{start}[u] < \mathsf{finish}[u] < \mathsf{finish}[s]$ [( )]
  (2) $\mathsf{start}[u] < \mathsf{finish}[u] < \mathsf{start}[s] < \mathsf{finish}[s]$ ( ) [ ]
- if (2), with our induction assumption, we get $\mathsf{start}[u] < \mathsf{start}[t]$
- since (t, u) is in T, (u, t) is in G. With previous item, we get: t is a descendant of u in the DFS of G (white path)

Correctness 3/3

Want to prove: for all t in T, t is a descendant of s in the DFS forest of G.

By induction: suppose it is true for some t in T, and prove it is true for its children. So let u be a child of t in T.

- $\mathsf{start}[s] \leq \mathsf{start}[t] < \mathsf{finish}[t] \leq \mathsf{finish}[s]$ induction assumption
- by definition of s, $\mathsf{finish}[u] < \mathsf{finish}[s]$, so our options are
  (1) $\mathsf{start}[s] < \mathsf{start}[u] < \mathsf{finish}[u] < \mathsf{finish}[s]$ [( )]
  (2) $\mathsf{start}[u] < \mathsf{finish}[u] < \mathsf{start}[s] < \mathsf{finish}[s]$ ( ) [ ]
- if (2), with our induction assumption, we get $\mathsf{start}[u] < \mathsf{start}[t]$
- since (t, u) is in T, (u, t) is in G. With previous item, we get: t is a descendant of u in the DFS of G (white path)
- this gives $\mathsf{start}[u] < \mathbf{start}[t] < \mathsf{finish}[t] < \mathbf{finish}[u]$

Correctness 3/3

Want to prove: for all t in T, t is a descendant of s in the DFS forest of G.

By induction: suppose it is true for some t in T, and prove it is true for its children. So let u be a child of t in T.

- $\text{start}[s] \leq \text{start}[t] < \text{finish}[t] \leq \text{finish}[s]$ induction assumption
- by definition of s, $\text{finish}[u] < \text{finish}[s]$, so our options are
  (1) $\text{start}[s] < \text{start}[u] < \text{finish}[u] < \text{finish}[s]$ [( )]
  (2) $\text{start}[u] < \text{finish}[u] < \text{start}[s] < \text{finish}[s]$ ( ) [ ]
- if (2), with our induction assumption, we get $\text{start}[u] < \text{start}[t]$
- since (t, u) is in T, (u, t) is in G. With previous item, we get: t is a descendant of u in the DFS of G (white path)
- this gives $\text{start}[u] < \textbf{start}[t] < \text{finish}[t] < \textbf{finish}[u]$
- but also $\textbf{finish}[u] < \text{start}[s] < \textbf{start}[t]$ from (2) and induction assumption

Correctness 3/3

Want to prove: for all t in T, t is a descendant of s in the DFS forest of G.

By induction: suppose it is true for some t in T, and prove it is true for its children. So let u be a child of t in T.

- $\text{start}[s] \leq \text{start}[t] < \text{finish}[t] \leq \text{finish}[s]$ induction assumption
- by definition of s, $\text{finish}[u] < \text{finish}[s]$, so our options are
  (1) $\text{start}[s] < \text{start}[u] < \text{finish}[u] < \text{finish}[s]$ [( )]
  (2) $\text{start}[u] < \text{finish}[u] < \text{start}[s] < \text{finish}[s]$ ( ) [ ]
- if (2), with our induction assumption, we get $\text{start}[u] < \text{start}[t]$
- since (t, u) is in T, (u, t) is in G. With previous item, we get: t is a descendant of u in the DFS of G (white path)
- this gives $\text{start}[u] < \textbf{start}[t] < \text{finish}[t] < \textbf{finish}[u]$
- but also $\textbf{finish}[u] < \text{start}[s] < \textbf{start}[t]$ from (2) and induction assumption
- so (2) impossible, and we must have (1)
- shows that u is a descendant of s in the DFS forest of G