Algorithms and Datastructures

Lecture 10

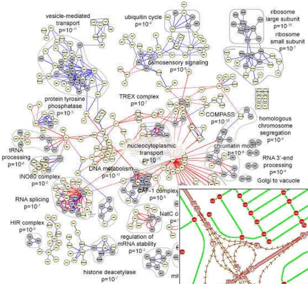
Manfred Jaeger

AALBORG UNIVERSITET

Graphs

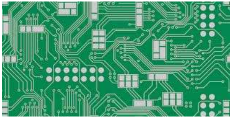
Graphs and Networks everywhere:

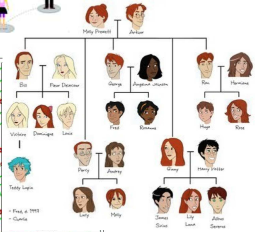
Biological Network



Social Network

Circuit

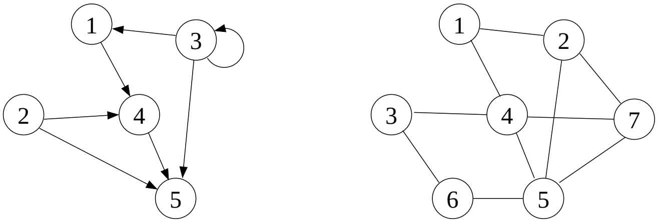




Pedigree

Road Network

The mathematical model for connected entities:



directed undirected

A **Graph** is a pair (V, E) where

- $V = \{v_1, \ldots, v_n\}$ is a set of nodes (a.k.a. vertices).
- E is a set of ordered pairs (v_i, v_j) of vertices (**directed** graph), or unordered sets $\{v_i, v_j\}$ of vertices (**undirected** graph)
- Undirected graphs contain no self loops (edges (v, v))

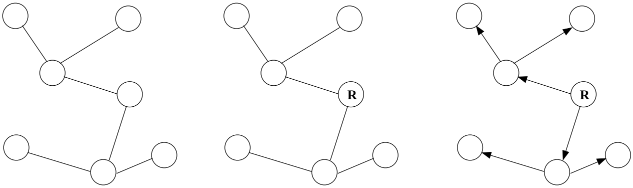
A graph is **weighted** if there also is a function

$$w : E \to \mathbb{R}$$

Cycles

Path: Sequence of nodes $\langle v_0, \ldots, v_k \rangle$ so that $(v_i, v_{i+1}) \in E$ $(i = 0, \ldots, k-1)$.

Simple Cycle: Path with $v_0 = v_k$, and all vertices $v_1, \ldots, v_k$ are distinct. For undirected graphs: $k \geq 3$.



(a) *(b)* *(c)*

Trees

Tree (a): Undirected graph without simple cycles

Rooted Tree (b): Tree with distinguished *Root* node

Directed Tree (c): Directed graph with distinguished *Root* node R, so that every node $v \in V$ is reachable from R by a unique path.

A Graph ADT

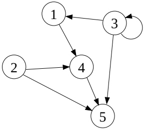
Data: graph data (V, E)

Operations

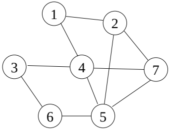
Name	Specification
void *addNode*(v)	adds new node v to V
void *addEdge*(v, w)	adds new edge (v, w) (or $\{v, w\}$) to E
void *removeNode*(v)	remove node v from V
void *removeEdge*(v, w)	remove edge (v, w) (or $\{v, w\}$) from E
list(Nodes) *allNodes*()	returns a list of all nodes in V
list(Nodes) *neighbors*(v)	returns a list of all nodes w with $(v, w) \in E$ ($\{v, w\} \in E$)
boolean *testEdge*(v, w)	returns *true* if $(v, w) \in E$ ($\{v, w\} \in E$)

☞ Not necessarily a complete list (Graph ADT does not have a generally accepted, normative definition, like Stack, Queue ...)!

☞ For weighted graphs also *get*, *set* operations for the weights.



	1	2	3	4	5
1	0	0	0	1	0
2	0	0	0	1	1
3	1	0	1	0	1
4	0	0	0	0	1
5	0	0	0	0	0



	1	2	3	4	5	6	7
1	0	1	0	1	0	0	0
2	1	0	0	0	1	0	1
3	0	0	0	1	0	1	0
4	1	0	1	0	1	0	1
5	0	1	0	1	0	1	1
6	0	0	1	0	1	0	0
7	0	1	0	1	1	0	0

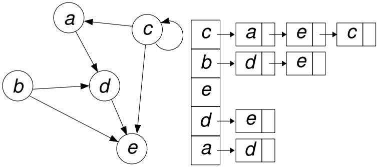
- Nodes identified with indices $1, \ldots, n$
- Edge set E stored as an $n \times n$-dimensional **adjacency matrix** A with

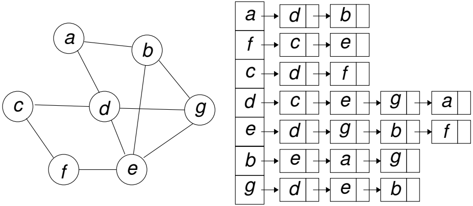
$$A[i,j] = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{if } (i,j) \notin E \end{cases}$$

For weighted graphs:

$$A[i,j] = \begin{cases} w(i,j) & \text{if } (i,j) \in E \\ nil & \text{if } (i,j) \notin E \end{cases}$$

(alternative: 0 for non-existing edges)





- Data stored in array of linked lists
- List elements contain identifier of (or pointer to) node
- Does not require that nodes possess integer identifiers $1, \ldots, n$.

Call nodes **indexed**, if nodes $v \in V$ have attribute $v.index \in 1, \ldots, |V|$ (for adjacency matrix nodes need to be indexed).

Operation	Adjacency Matrix	Adjacency List Indexed	Adjacency List Not Indexed
void *addEdge*(v, w)	$O(1)$	$O(1)$	$O(\|V\|)$
list(Nodes) *allNodes*()	$O(1)$	$O(1)$	$O(1)$
list(Nodes) *neighbors*(v)	$O(\|V\|)$	$O(1)$	$O(\|V\|)$
boolean *testEdge*(v, w)	$O(1)$	$O(\|E\|)$	$O(\|V\| + \|E\|)$

Space: Adjacency Matrix: $O(|V|^2)$; Adjacency List: $O(|V| + |E|)$. (but matrix has much lower constant factor!).

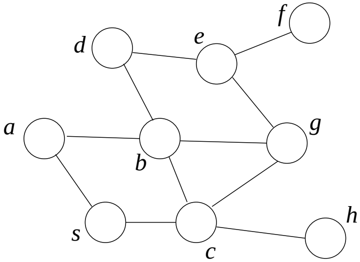
Breadth-first Search

input : Graph G with *source* vertex $s \in V$

output: Setting of node attributes *color*, d, and π.

BFS(G, s)

```
for each u ∈ G.allnodes() \ {s} do
    u.color = white
    u.d = ∞
    u.π = nil
s.color = gray
s.d = 0
s.π = nil
Q = newQueue
ENQUEUE(Q, s)
while Q ≠ ∅ do
    u = Dequeue(Q)
    for each v ∈ neighbors(u) do
        if v.color = white then
            v.color = gray
            v.d=u.d+1
            v.π = u
            ENQUEUE(Q, v)
    u.color = black
```

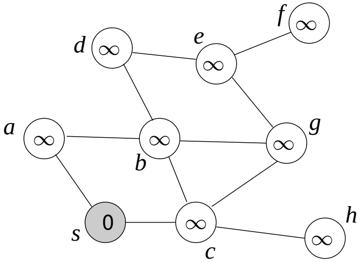


input : Graph G with *source* vertex $s \in V$

output: Setting of node attributes *color*, *d*, and π.

BFS(G, s)

for *each* $u \in G.allnodes() \setminus \{s\}$ **do**
    $u.color = white$
    $u.d = \infty$
    $u.\pi = nil$
$s.color = gray$
$s.d = 0$
$s.\pi = nil$
$Q = newQueue$
ENQUEUE(Q, s)
while $Q \neq \emptyset$ **do**
    $u = Dequeue(Q)$
    for *each* $v \in neighbors(u)$ **do**
        if *v.color = white* **then**
            v.color = gray
            v.d=u.d+1
            $v.\pi = u$
            ENQUEUE(Q, v)
    $u.color = black$



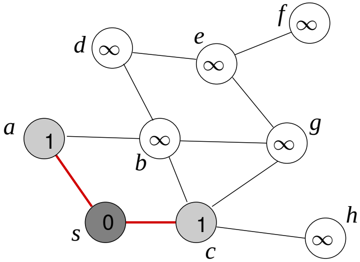
Q:
s

input : Graph *G* with *source* vertex $s \in V$

output: Setting of node attributes *color*, *d*, and π.

BFS(G, s)

```
for each u ∈ G.allnodes() \ {s} do
    u.color = white
    u.d = ∞
    u.π = nil
s.color = gray
s.d = 0
s.π = nil
Q = newQueue
ENQUEUE(Q, s)
while Q ≠ ∅ do
    u = Dequeue(Q)
    for each v ∈ neighbors(u) do
        if v.color = white then
            v.color = gray
            v.d=u.d+1
            v.π = u
            ENQUEUE(Q, v)
    u.color = black
```



Q:

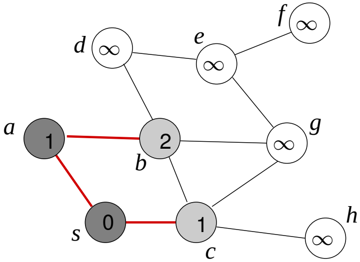
a
c

input : Graph G with *source* vertex $s \in V$

output: Setting of node attributes *color*, *d*, and π.

BFS(G, s)

```
for each u ∈ G.allnodes() \ {s} do
    u.color = white
    u.d = ∞
    u.π = nil
s.color = gray
s.d = 0
s.π = nil
Q = newQueue
ENQUEUE(Q, s)
while Q ≠ ∅ do
    u = Dequeue(Q)
    for each v ∈ neighbors(u) do
        if v.color = white then
            v.color = gray
            v.d=u.d+1
            v.π = u
            ENQUEUE(Q, v)
    u.color = black
```



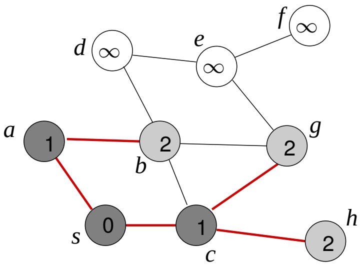
Q:

c
b

input : Graph G with *source* vertex $s \in V$
output: Setting of node attributes *color*, *d*, and π.

BFS(G, s)

```
for each u ∈ G.allnodes() \ {s} do
    u.color = white
    u.d = ∞
    u.π = nil
s.color = gray
s.d = 0
s.π = nil
Q = newQueue
ENQUEUE(Q, s)
while Q ≠ ∅ do
    u = Dequeue(Q)
    for each v ∈ neighbors(u) do
        if v.color = white then
            v.color = gray
            v.d=u.d+1
            v.π = u
            ENQUEUE(Q, v)
    u.color = black
```



Q:

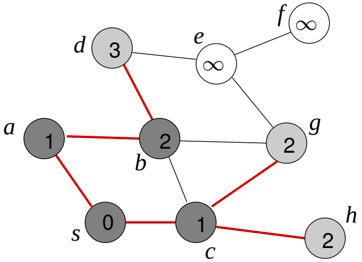
b
g
h

input : Graph G with *source* vertex $s \in V$

output: Setting of node attributes *color*, d, and π.

BFS(G, s)

for *each* $u \in G.allnodes() \setminus \{s\}$ **do**
  $u.color = white$
  $u.d = \infty$
  $u.\pi = nil$
$s.color = gray$
$s.d = 0$
$s.\pi = nil$
$Q = newQueue$
ENQUEUE(Q, s)
while $Q \neq \emptyset$ **do**
  $u = Dequeue(Q)$
  for *each* $v \in neighbors(u)$ **do**
    if *v.color = white* **then**
      v.color = gray
      v.d=u.d+1
      $v.\pi = u$
      ENQUEUE(Q, v)
  $u.color = black$



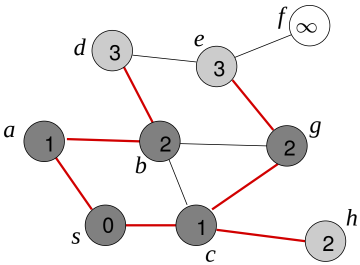
Q:

g
h
d

input : Graph G with *source* vertex $s \in V$
output: Setting of node attributes *color*, *d*, and π.

BFS(G, s)

for *each* $u \in G.allnodes() \setminus \{s\}$ **do**
    $u.color = white$
    $u.d = \infty$
    $u.\pi = nil$
$s.color = gray$
$s.d = 0$
$s.\pi = nil$
$Q = newQueue$
ENQUEUE(Q, s)
while $Q \neq \emptyset$ **do**
    $u = Dequeue(Q)$
    for *each* $v \in neighbors(u)$ **do**
        if *v.color = white* **then**
            v.color = gray
            v.d=u.d+1
            $v.\pi = u$
            ENQUEUE(Q, v)
    $u.color = black$



Q:

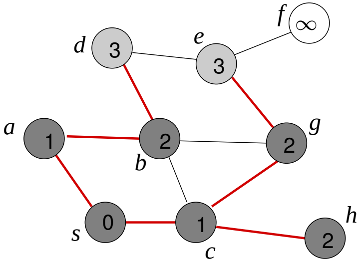
h
d
e

input : Graph G with *source* vertex $s \in V$

output: Setting of node attributes *color*, *d*, and π.

BFS(G, s)

```
for each u ∈ G.allnodes() \ {s} do
    u.color = white
    u.d = ∞
    u.π = nil
s.color = gray
s.d = 0
s.π = nil
Q = newQueue
ENQUEUE(Q, s)
while Q ≠ ∅ do
    u = Dequeue(Q)
    for each v ∈ neighbors(u) do
        if v.color = white then
            v.color = gray
            v.d=u.d+1
            v.π = u
            ENQUEUE(Q, v)
    u.color = black
```



Q:

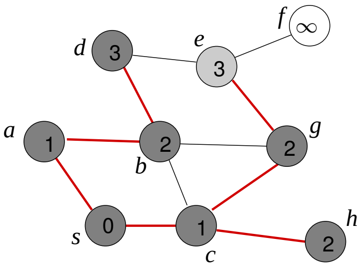
d
e

input : Graph G with *source* vertex $s \in V$

output: Setting of node attributes *color*, *d*, and π.

BFS(G, s)

for *each* $u \in G.allnodes() \setminus \{s\}$ **do**
    $u.color = white$
    $u.d = \infty$
    $u.\pi = nil$
$s.color = gray$
$s.d = 0$
$s.\pi = nil$
$Q = newQueue$
ENQUEUE(Q, s)
while $Q \neq \emptyset$ **do**
    $u = Dequeue(Q)$
    for *each* $v \in neighbors(u)$ **do**
        if *v.color = white* **then**
            v.color = gray
            v.d=u.d+1
            $v.\pi = u$
            ENQUEUE(Q, v)
    $u.color = black$



Q:

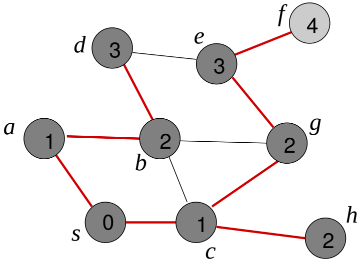
e

input : Graph G with *source* vertex $s \in V$

output: Setting of node attributes *color*, d, and π.

BFS(G, s)

```
for each u ∈ G.allnodes() \ {s} do
    u.color = white
    u.d = ∞
    u.π = nil
s.color = gray
s.d = 0
s.π = nil
Q = newQueue
ENQUEUE(Q, s)
while Q ≠ ∅ do
    u = Dequeue(Q)
    for each v ∈ neighbors(u) do
        if v.color = white then
            v.color = gray
            v.d=u.d+1
            v.π = u
            ENQUEUE(Q, v)
    u.color = black
```



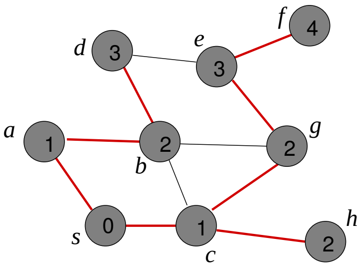
Q:

f

input : Graph G with *source* vertex $s \in V$
output: Setting of node attributes *color*, *d*, and π.

BFS(G, s)

for *each* $u \in G.allnodes() \setminus \{s\}$ **do**
    $u.color = white$
    $u.d = \infty$
    $u.\pi = nil$
$s.color = gray$
$s.d = 0$
$s.\pi = nil$
$Q = newQueue$
ENQUEUE(Q, s)
while $Q \neq \emptyset$ **do**
    $u = Dequeue(Q)$
    for *each* $v \in neighbors(u)$ **do**
        if *v.color = white* **then**
            v.color = gray
            v.d=u.d+1
            $v.\pi = u$
            ENQUEUE(Q, v)
    $u.color = black$



Q:

BFS(G, s)

```
1  for each u ∈ G.allnodes() \ {s} do
2      u.color = white
3      u.d = ∞
4      u.π = nil
5  s.color = gray
6  s.d = 0
7  s.π = nil
8  Q = newQueue
9  ENQUEUE(Q, s)
10 while Q ≠ ∅ do
11     u = Dequeue(Q)
12     for each v ∈ neighbors(u) do
13         if v.color = white then
14             v.color = gray
15             v.d=u.d+1
16             v.π = u
17             ENQUEUE(Q, v)
18     u.color = black
```

Aggregate Analysis

Analysing loops: Instead of bounding the time for each single loop iteration, and summing over all iterations:

- directly bound the total time spent on all loop iterations

BFS Analysis

Total time spent:

- lines **10,11**: $O(|V|)$ (each node is enqueued/dequeued at most once).
- lines **13-17**: $O(|E|)$ (each edge is processed once (directed graph), or twice (undirected graph)).
- line **12**: $O(|V|)$ if computation of *neighbors*(u) is $O(1)$, $O(|V|^2)$ if *neighbors*(u) is $O(|V|)$.

Total time $O(|V| + |E|)$ if *neighbors*(u) computable in $O(1)$, otherwise $O(|V|^2)$.

Shortest Path

For unweighted graph G: shortest path distance $\delta(s, v)$ from s to v is the minimum number of edges on paths from s to v (∞ if no path exists).

Shortest Paths from BFS

After termination of $BFS(G, s)$:

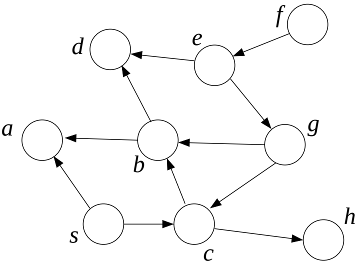
- for all $v \in V$: $v.d = \delta(s, v)$
- for all v with $v.d \neq \infty$: a shortest path from s to v is given by backward-tracing the π pointers from v to s.

Depth-first Search

```
DFS(G)
1  for each u ∈ G.allnodes() do
2      u.color = white
3      u.π = nil
4  time = 0
5  for each u ∈ G.allnodes() do
6      if u.color = white then
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = gray
4  for each v ∈ neighbors(u) do
5      if v.color=white then
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = black
9  time = time + 1
10 u.f = time
```

Snapshots at line **4** of DFS-VISIT.
Nodes marked with $u.d/u.f$:



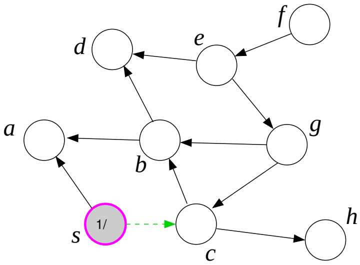
DFS(G)

```
1  for each u ∈ G.allnodes() do
2      u.color = white
3      u.π = nil
4  time = 0
5  for each u ∈ G.allnodes() do
6      if u.color = white then
7          DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1  time = time + 1
2  u.d = time
3  u.color = gray
4  for each v ∈ neighbors(u) do
5      if v.color=white then
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = black
9  time = time + 1
10 u.f = time
```

Snapshots at line **4** of DFS-VISIT.
Nodes marked with $u.d/u.f$:



current node u

π edge

next $v \in neighbors(u)$

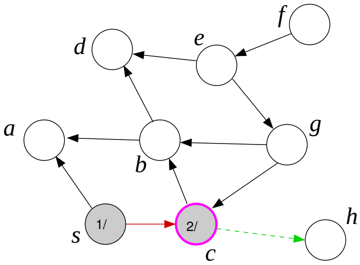
DFS(G)

```
for each u ∈ G.allnodes() do
    u.color = white
    u.π = nil
time = 0
for each u ∈ G.allnodes() do
    if u.color = white then
        DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1  time = time + 1
2  u.d = time
3  u.color = gray
4  for each v ∈ neighbors(u) do
5      if v.color=white then
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = black
9  time = time + 1
10 u.f = time
```

Snapshots at line **4** of DFS-VISIT.
Nodes marked with $u.d/u.f$:



current node u

π edge

next $v \in neighbors(u)$

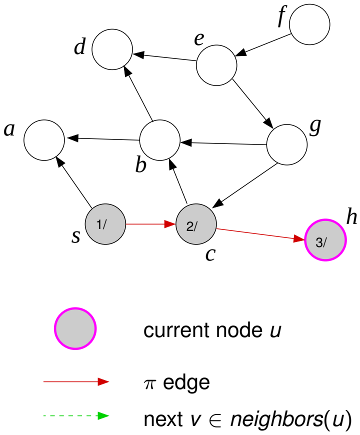
DFS(G)

```
1  for each u ∈ G.allnodes() do
2      u.color = white
3      u.π = nil
4  time = 0
5  for each u ∈ G.allnodes() do
6      if u.color = white then
7          DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1  time = time + 1
2  u.d = time
3  u.color = gray
4  for each v ∈ neighbors(u) do
5      if v.color=white then
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = black
9  time = time + 1
10 u.f = time
```

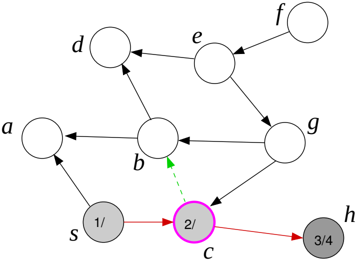
Snapshots at line **4** of DFS-VISIT.
Nodes marked with $u.d/u.f$:



```
DFS(G)
for each u ∈ G.allnodes() do
    u.color = white
    u.π = nil
time = 0
for each u ∈ G.allnodes() do
    if u.color = white then
        DFS-Visit(G, u)
```

```
DFS-Visit(G, u)
1  time = time + 1
2  u.d = time
3  u.color = gray
4  for each v ∈ neighbors(u) do
5      if v.color=white then
6          v.π = u
7          DFS-Visit(G, v)
8  u.color = black
9  time = time + 1
10 u.f = time
```

Snapshots at line **4** of DFS-Visit.
Nodes marked with $u.d/u.f$:



current node u

π edge

next $v \in neighbors(u)$

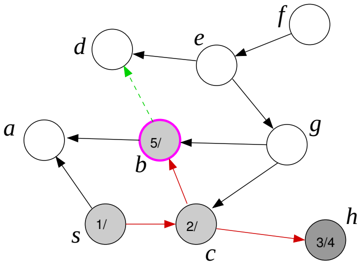
<u>DFS(G)</u>

1 **for each** $u \in G.allnodes()$ **do**
2 $\quad u.color = white$
3 $\quad u.\pi = nil$
4 $time = 0$
5 **for each** $u \in G.allnodes()$ **do**
6 $\quad$ **if** $u.color = white$ **then**
7 $\qquad$ DFS-VISIT(G, u)

<u>DFS-VISIT(G, u)</u>

1 $time = time + 1$
2 $u.d = time$
3 $u.color = gray$
4 **for each** $v \in neighbors(u)$ **do**
5 $\quad$ **if** $v.color{=}white$ **then**
6 $\qquad v.\pi = u$
7 $\qquad$ DFS-VISIT(G, v)
8 $u.color = black$
9 $time = time + 1$
10 $u.f = time$

Snapshots at line **4** of DFS-VISIT.
Nodes marked with $u.d/u.f$:



current node u

π edge

next $v \in neighbors(u)$

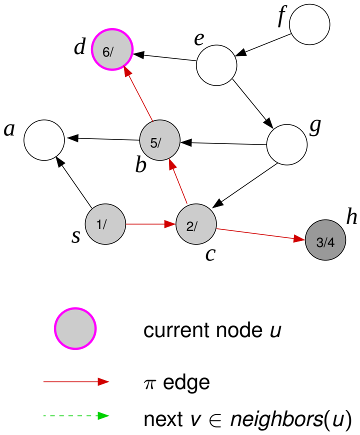
DFS(G)

```
1  for each u ∈ G.allnodes() do
2      u.color = white
3      u.π = nil
4  time = 0
5  for each u ∈ G.allnodes() do
6      if u.color = white then
7          DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1  time = time + 1
2  u.d = time
3  u.color = gray
4  for each v ∈ neighbors(u) do
5      if v.color=white then
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = black
9  time = time + 1
10 u.f = time
```

Snapshots at line **4** of DFS-VISIT.
Nodes marked with $u.d/u.f$:



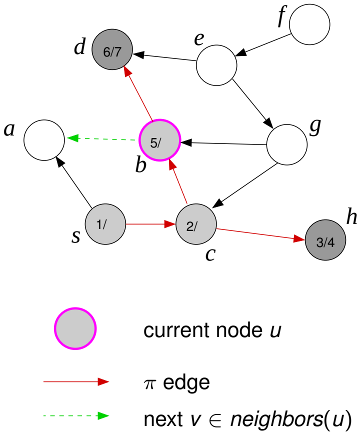
DFS(G)

1 **for** *each* $u \in G.allnodes()$ **do**
2 $\quad u.color = white$
3 $\quad u.\pi = nil$
4 $time = 0$
5 **for** *each* $u \in G.allnodes()$ **do**
6 $\quad$ **if** $u.color = white$ **then**
7 $\quad\quad$ DFS-VISIT(G, u)

DFS-VISIT(G, u)

1 $time = time + 1$
2 $u.d = time$
3 $u.color = gray$
4 **for** *each* $v \in neighbors(u)$ **do**
5 $\quad$ **if** $v.color = white$ **then**
6 $\quad\quad v.\pi = u$
7 $\quad\quad$ DFS-VISIT(G, v)
8 $u.color = black$
9 $time = time + 1$
10 $u.f = time$

Snapshots at line **4** of DFS-VISIT.
Nodes marked with $u.d/u.f$:



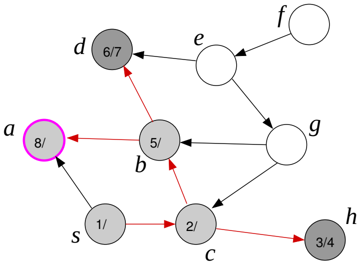
DFS(G)

1 **for each** $u \in G.allnodes()$ **do**
2 $\quad u.color = white$
3 $\quad u.\pi = nil$
4 $time = 0$
5 **for each** $u \in G.allnodes()$ **do**
6 $\quad$ **if** $u.color = white$ **then**
7 $\quad\quad$ DFS-VISIT(G, u)

DFS-VISIT(G, u)

1 $time = time + 1$
2 $u.d = time$
3 $u.color = gray$
4 **for each** $v \in neighbors(u)$ **do**
5 $\quad$ **if** $v.color = white$ **then**
6 $\quad\quad v.\pi = u$
7 $\quad\quad$ DFS-VISIT(G, v)
8 $u.color = black$
9 $time = time + 1$
10 $u.f = time$

Snapshots at line **4** of DFS-VISIT.
Nodes marked with $u.d/u.f$:



current node u

π edge

next $v \in neighbors(u)$

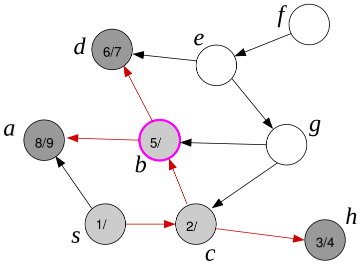
DFS(G)

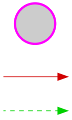
```
1  for each u ∈ G.allnodes() do
2      u.color = white
3      u.π = nil
4  time = 0
5  for each u ∈ G.allnodes() do
6      if u.color = white then
7          DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1  time = time + 1
2  u.d = time
3  u.color = gray
4  for each v ∈ neighbors(u) do
5      if v.color=white then
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = black
9  time = time + 1
10 u.f = time
```

Snapshots at line **4** of DFS-VISIT.
Nodes marked with $u.d/u.f$:





current node u

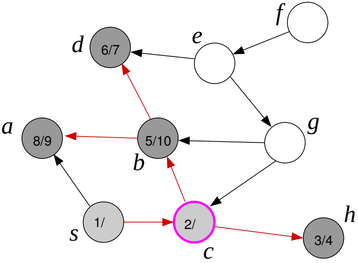
π edge

next $v \in neighbors(u)$

```
DFS(G)
1  for each u ∈ G.allnodes() do
2      u.color = white
3      u.π = nil
4  time = 0
5  for each u ∈ G.allnodes() do
6      if u.color = white then
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = gray
4  for each v ∈ neighbors(u) do
5      if v.color=white then
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = black
9  time = time + 1
10 u.f = time
```

Snapshots at line **4** of DFS-VISIT.
Nodes marked with $u.d/u.f$:



current node u

π edge

next $v \in neighbors(u)$

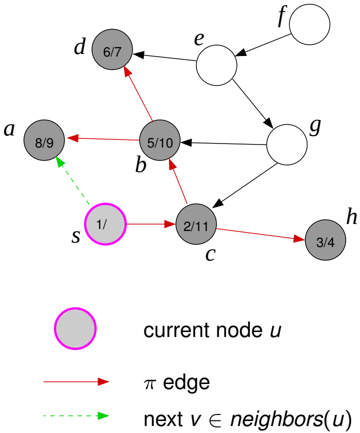
DFS(G)

```
for each u ∈ G.allnodes() do
    u.color = white
    u.π = nil
time = 0
for each u ∈ G.allnodes() do
    if u.color = white then
        DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1  time = time + 1
2  u.d = time
3  u.color = gray
4  for each v ∈ neighbors(u) do
5      if v.color=white then
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = black
9  time = time + 1
10 u.f = time
```

Snapshots at line **4** of DFS-VISIT.
Nodes marked with $u.d/u.f$:



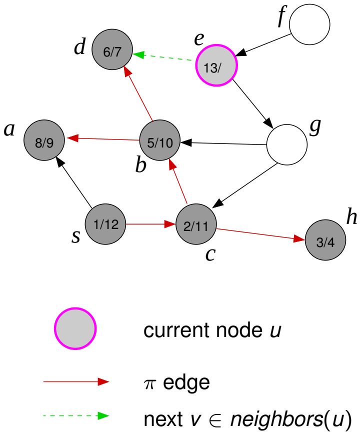
DFS(G)

```
1  for each u ∈ G.allnodes() do
2      u.color = white
3      u.π = nil
4  time = 0
5  for each u ∈ G.allnodes() do
6      if u.color = white then
7          DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1  time = time + 1
2  u.d = time
3  u.color = gray
4  for each v ∈ neighbors(u) do
5      if v.color=white then
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = black
9  time = time + 1
10 u.f = time
```

Snapshots at line **4** of DFS-VISIT.
Nodes marked with $u.d/u.f$:



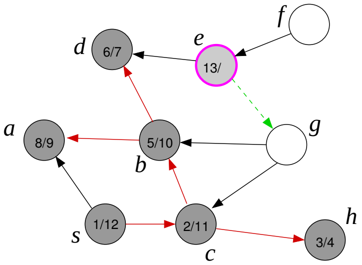
DFS(G)

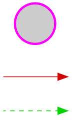
```
1  for each u ∈ G.allnodes() do
2      u.color = white
3      u.π = nil
4  time = 0
5  for each u ∈ G.allnodes() do
6      if u.color = white then
7          DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1  time = time + 1
2  u.d = time
3  u.color = gray
4  for each v ∈ neighbors(u) do
5      if v.color=white then
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = black
9  time = time + 1
10 u.f = time
```

Snapshots at line **4** of DFS-VISIT.
Nodes marked with $u.d/u.f$:





current node u

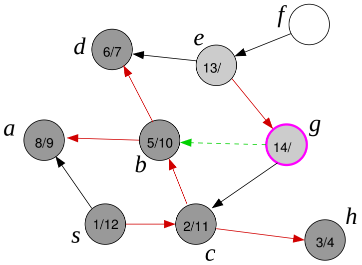
π edge

next $v \in neighbors(u)$

```
DFS(G)
for each u ∈ G.allnodes() do
    u.color = white
    u.π = nil
time = 0
for each u ∈ G.allnodes() do
    if u.color = white then
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = gray
4  for each v ∈ neighbors(u) do
5      if v.color=white then
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = black
9  time = time + 1
10 u.f = time
```

Snapshots at line **4** of DFS-VISIT.
Nodes marked with $u.d/u.f$:



current node u

π edge

next $v \in neighbors(u)$

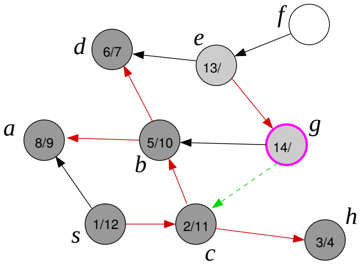
DFS(G)

```
1  for each u ∈ G.allnodes() do
2      u.color = white
3      u.π = nil
4  time = 0
5  for each u ∈ G.allnodes() do
6      if u.color = white then
7          DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
1  time = time + 1
2  u.d = time
3  u.color = gray
4  for each v ∈ neighbors(u) do
5      if v.color=white then
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = black
9  time = time + 1
10 u.f = time
```

Snapshots at line **4** of DFS-VISIT.
Nodes marked with $u.d/u.f$:



current node u

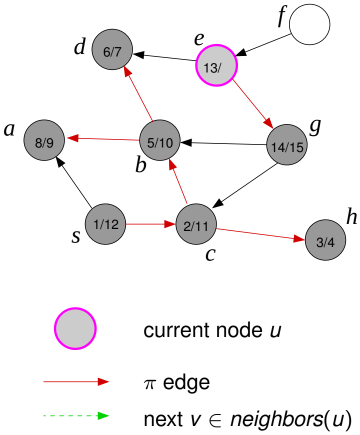
π edge

next $v \in neighbors(u)$

```
DFS(G)
1  for each u ∈ G.allnodes() do
2      u.color = white
3      u.π = nil
4  time = 0
5  for each u ∈ G.allnodes() do
6      if u.color = white then
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = gray
4  for each v ∈ neighbors(u) do
5      if v.color=white then
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = black
9  time = time + 1
10 u.f = time
```

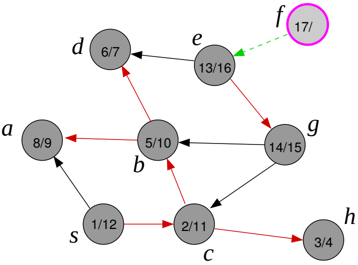
Snapshots at line **4** of DFS-VISIT.
Nodes marked with $u.d/u.f$:



```
    DFS(G)
 1  for each u ∈ G.allnodes() do
 2      u.color = white
 3      u.π = nil
 4  time = 0
 5  for each u ∈ G.allnodes() do
 6      if u.color = white then
 7          DFS-VISIT(G, u)
```

```
    DFS-VISIT(G, u)
 1  time = time + 1
 2  u.d = time
 3  u.color = gray
 4  for each v ∈ neighbors(u) do
 5      if v.color=white then
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = black
 9  time = time + 1
10  u.f = time
```

Snapshots at line **4** of DFS-VISIT.
Nodes marked with $u.d/u.f$:



current node u

π edge

next $v \in neighbors(u)$

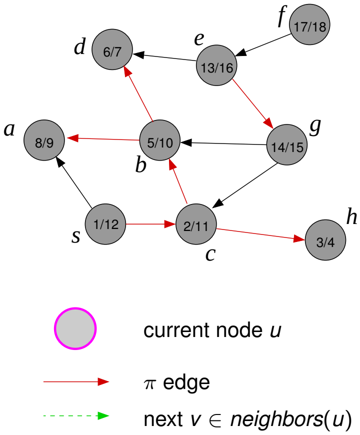
DFS(G)

1 **for each** $u \in G.allnodes()$ **do**
2 $\quad u.color = white$
3 $\quad u.\pi = nil$
4 $time = 0$
5 **for each** $u \in G.allnodes()$ **do**
6 $\quad$ **if** $u.color = white$ **then**
7 $\quad\quad$ DFS-VISIT(G, u)

DFS-VISIT(G, u)

1 $time = time + 1$
2 $u.d = time$
3 $u.color = gray$
4 **for each** $v \in neighbors(u)$ **do**
5 $\quad$ **if** $v.color = white$ **then**
6 $\quad\quad v.\pi = u$
7 $\quad\quad$ DFS-VISIT(G, v)
8 $u.color = black$
9 $time = time + 1$
10 $u.f = time$

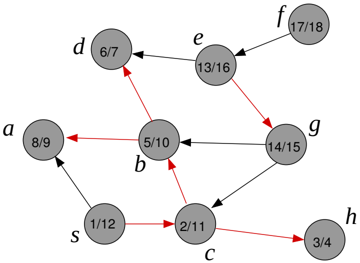
Snapshots at line **4** of DFS-VISIT.
Nodes marked with $u.d/u.f$:

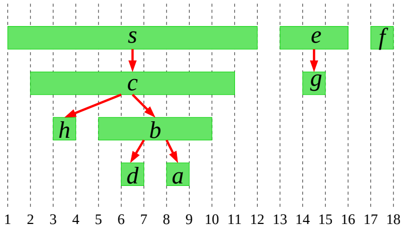


Similar aggregate analysis as for BFS:

- $O(|V| + |E|)$ if *neighbors*(u) computable in $O(1)$
- $O(|V|^2)$ if *neighbors*(u) computable in $O(|V|)$

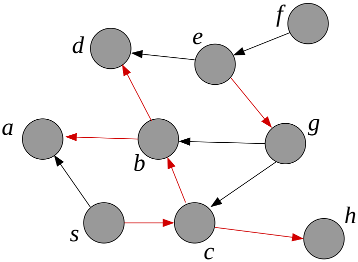
The π-edges define a set of rooted trees (a forest):



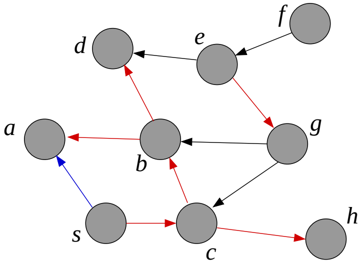


Right: tree structure with nodes spanning time interval $[v.d, v.f]$

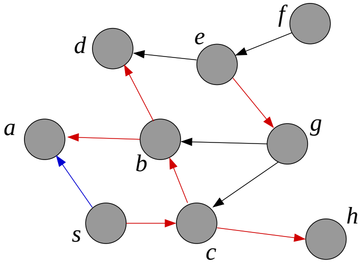
☞ The exact structure depends on the order in which nodes are enumerated in *allnodes*() and *neighbors*(u).



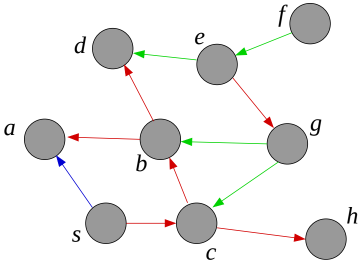
- Tree edges ($v.color = white$ in line **5**)



- Tree edges ($v.color = white$ in line **5**)
- Forward Edges: Edges inside one tree, leading from ancestor to descendant ($v.color = black$ in line **5**)



- Tree edges ($v.color = white$ in line **5**)
- Forward Edges: Edges inside one tree, leading from ancestor to descendant ($v.color = black$ in line **5**)
- Back Edges: Edges inside one tree, leading from descendant to ancestor ($v.color = gray$ in line **5**)

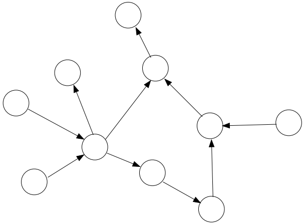


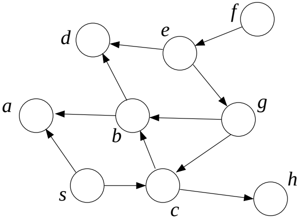
- Tree edges ($v.color = white$ in line **5**)
- Forward Edges: Edges inside one tree, leading from ancestor to descendant ($v.color = black$ in line **5**)
- Back Edges: Edges inside one tree, leading from descendant to ancestor ($v.color = gray$ in line **5**)
- Cross Edges: All other edges ($v.color = black$ in line **5**)

☞ this classification is determined by the node orderings in *allnodes*() and *neighbors*(u); *not* describing inherent edge properties.

Directed Acyclic Graphs (DAG)

Directed Graph without any cycles:





Typical Application: nodes represent tasks/events, edges temporal or logistic precedence

DAG Test by DFS

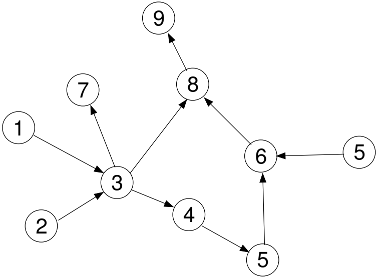
A directed graph is acyclic if and only if a DFS does not produce any Back edges (no gray nodes encountered in line **4**)

Typical Application: determine whether set of tasks is feasible

A **topological sort** of a *DAG* is an ordering of the vertices, so that

$$(u, v) \in E \;\Rightarrow\; u < v$$

Nodes labelled with index in a topological sort:

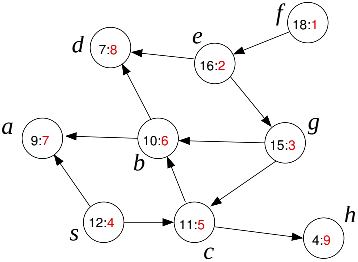


Typical application: construct a *feasible schedule* to carry out all tasks.

Topological Sort and DFS Finishing Times

The finishing times $v.f$ computed by DFS on a DAG define a reverse topological sort.

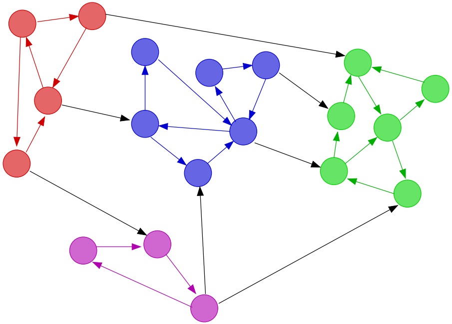
Example: Nodes with finishing times and index in topological sort:



Definition

A *strongly connected component (SCC)* of G is *maximal* subgraph C of G, such that for all $u, v \in C$ there exists a path from u to v.

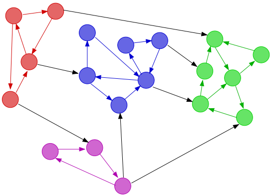
Example: graph with its SCCs shown in color :

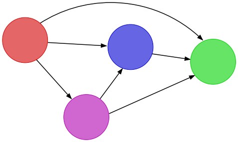


☞ For every directed graph G there is a unique partitioning into SCCs.

The **Component Graph** of the directed graph $G = (V, E)$ is the graph

- whose nodes are the SCCs $C_1, \dots, C_k$ of G, and
- there is an edge (C_i, C_j) if and only if there are nodes $u \in C_i$ and $v \in C_j$ with $(u, v) \in E$





☞ The Component Graph is a DAG!

DFS Trees and SCCs

- All nodes from SCC will be contained in a single tree constructed by DFS
- But: a single tree may contain nodes from several SCCs

Goal: apply DFS such that the DFS trees are exactly the SCCs.

Transpose Graph

The transpose G^T of a graph $G = (V, E)$ is the graph with the same vertex set V as G, and all edges reversed, i.e. G^T has the edge set

$$E^T = \{(u, v) \mid (v, u) \in E\}$$

☞ G and G^T have the same SCCs.

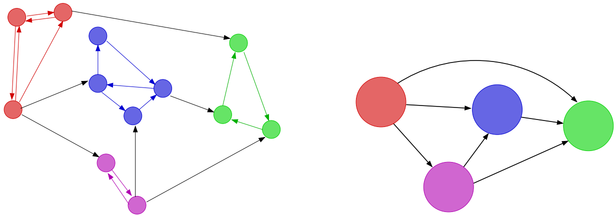
Step 1: Perform a standard DFS to compute finishing times $v.f$ for all vertices

Step 2: Turn G into G^T.

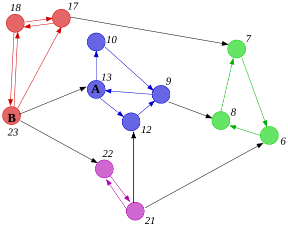
Step 3: Perform a DFS on G^T, where in the top-level loop nodes are processed in decreasing order of their $v.f$ values

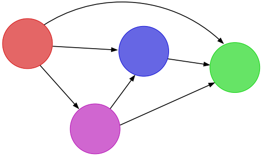
Step 4: Return the trees obtained in the second DFS

Step 1: Perform a standard DFS to compute finishing times $v.f$ for all vertices



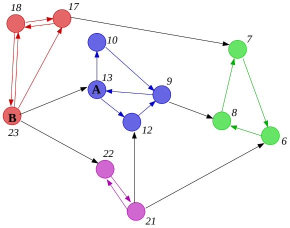
Step 1: Perform a standard DFS to compute finishing times $v.f$ for all vertices

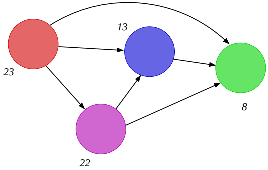




- Finishing times computed (top loop first started at **A**, then **B**; edges processed in clockwise order)

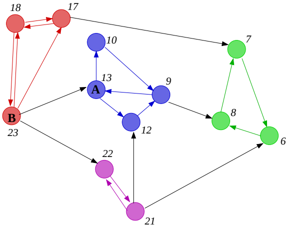
Step 1: Perform a standard DFS to compute finishing times $v.f$ for all vertices

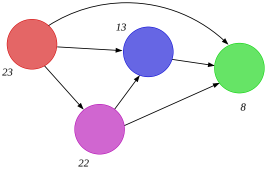




- Finishing times computed (top loop first started at **A**, then **B**; edges processed in clockwise order)
- SCCs C labeled with $f(C) = max_{u \in C}\, u.f$

Step 1: Perform a standard DFS to compute finishing times $v.f$ for all vertices

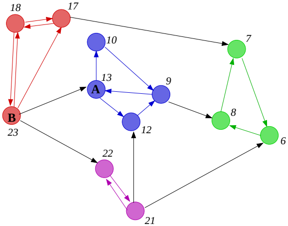


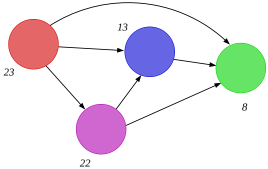


- Finishing times computed (top loop first started at **A**, then **B**; edges processed in clockwise order)
- SCCs C labeled with $f(C) = max_{u \in C}\, u.f$

☞ If there is an edge (C, C') in the component graph, then $f(C) > f(C')$.

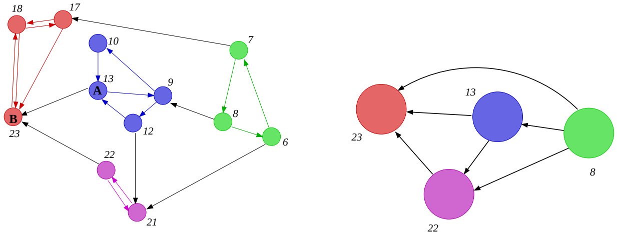
Step 2: Turn G into G^T.





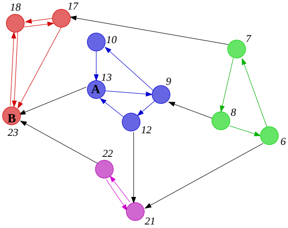
☞ Only the reversal of the edges between different SCCs matters!

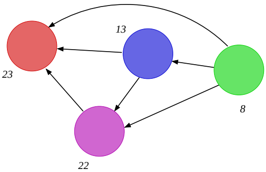
Step 2: Turn G into G^T.



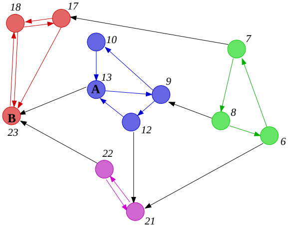
☞ Only the reversal of the edges between different SCCs matters!

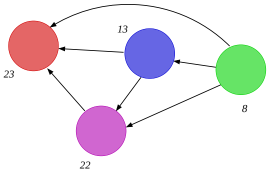
Step 3: Perform a DFS on G^T, where in the top-level loop nodes are processed in decreasing order of their $v.f$ values





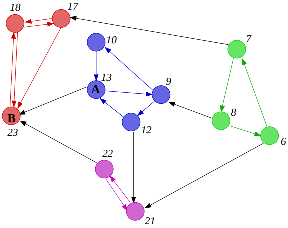
Step 3: Perform a DFS on G^T, where in the top-level loop nodes are processed in decreasing order of their $v.f$ values

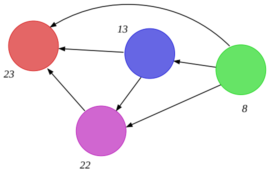




- The first node processed in the top-level loop belongs to a SCC whose nodes do not have any descendants in other SCCs

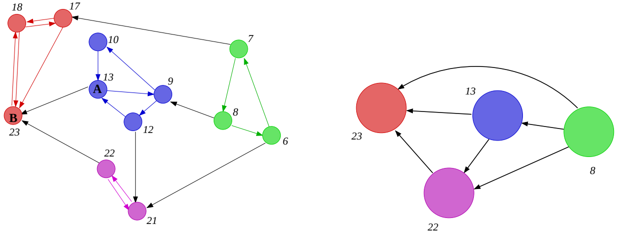
Step 3: Perform a DFS on G^T, where in the top-level loop nodes are processed in decreasing order of their $v.f$ values





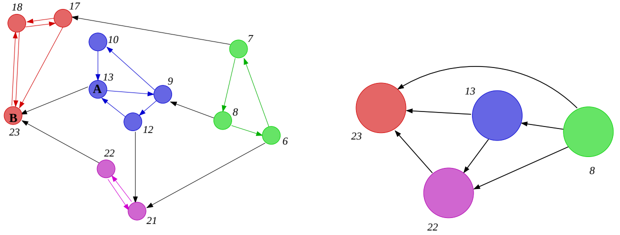
- The first node processed in the top-level loop belongs to a SCC whose nodes do not have any descendants in other SCCs
- $\rightsquigarrow$ the first DFS tree consists of the nodes of that component

Step 3: Perform a DFS on G^T, where in the top-level loop nodes are processed in decreasing order of their $v.f$ values



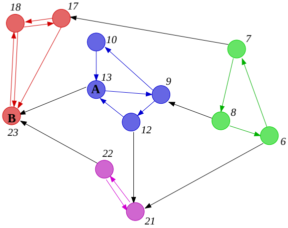
- The first node processed in the top-level loop belongs to a SCC whose nodes do not have any descendants in other SCCs
- $\rightsquigarrow$ the first DFS tree consists of the nodes of that component
- The second node processed in the top-level loop belongs to a SCC whose nodes can only have black descendants in other SCCs

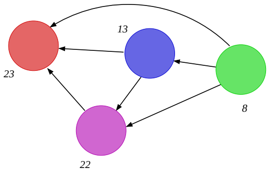
Step 3: Perform a DFS on G^T, where in the top-level loop nodes are processed in decreasing order of their $v.f$ values



- The first node processed in the top-level loop belongs to a SCC whose nodes do not have any descendants in other SCCs
- $\rightsquigarrow$ the first DFS tree consists of the nodes of that component
- The second node processed in the top-level loop belongs to a SCC whose nodes can only have black descendants in other SCCs
- $\rightsquigarrow$ the second DFS tree consists of the nodes of that component

Step 3: Perform a DFS on G^T, where in the top-level loop nodes are processed in decreasing order of their $v.f$ values





- The first node processed in the top-level loop belongs to a SCC whose nodes do not have any descendants in other SCCs
- $\rightsquigarrow$ the first DFS tree consists of the nodes of that component
- The second node processed in the top-level loop belongs to a SCC whose nodes can only have black descendants in other SCCs
- $\rightsquigarrow$ the second DFS tree consists of the nodes of that component
- . . . etc