Algorithms: DFS, STRONGLY CONNECTED COMPONENTS, FLOWS

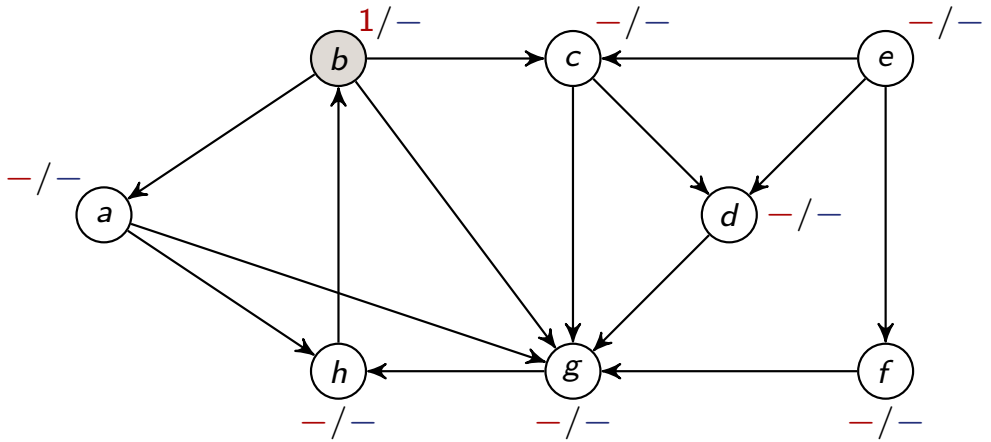
Ola Svensson

EPFL School of Computer and Communication Sciences

Lecture 15, 09.04.2024

Pseudocode of DFS

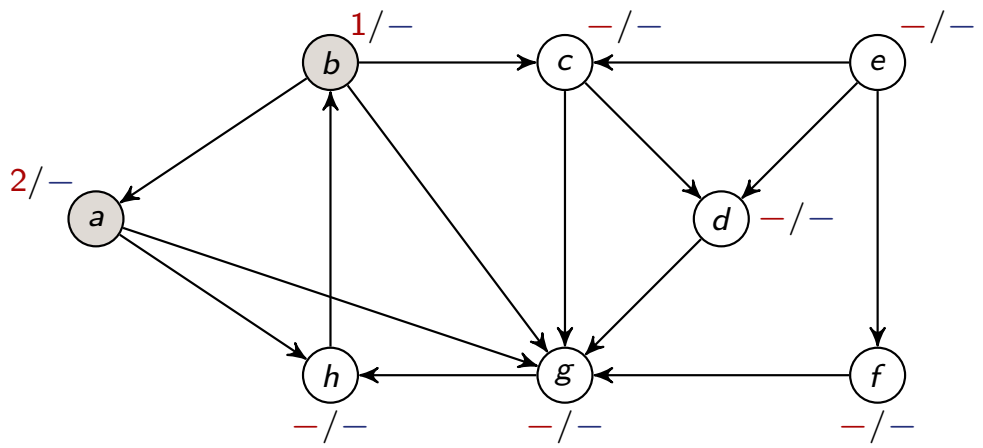
```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                    // discover u
  for each v ∈ G.Adj[u]             // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                        // finish u
```



time = 1

Pseudocode of DFS

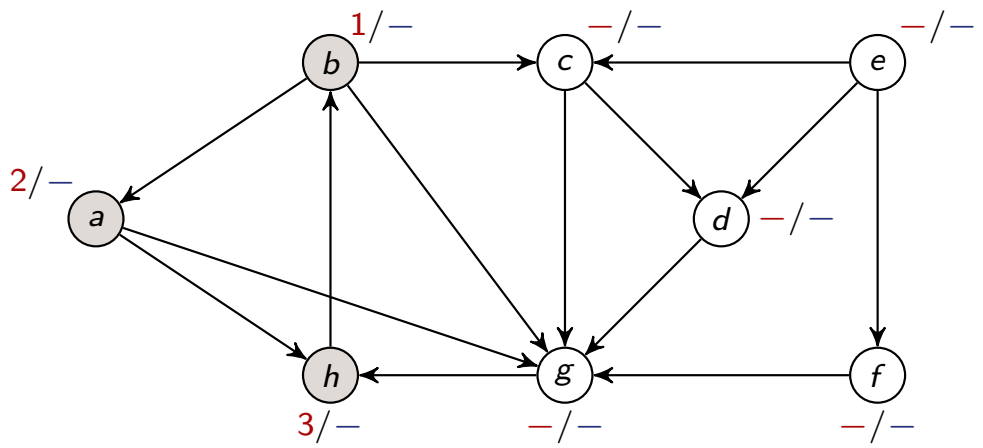
```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                    // discover u
  for each v ∈ G.Adj[u]             // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                        // finish u
```



time = 2

Pseudocode of DFS

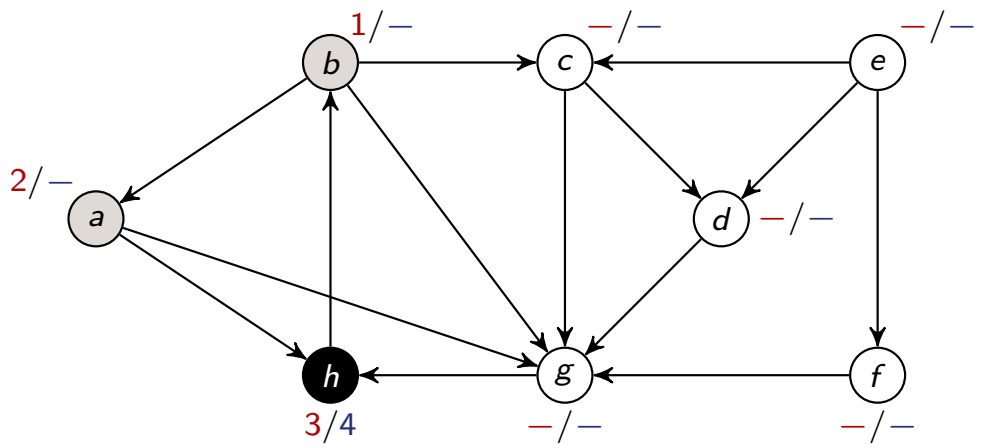
```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                    // discover u
  for each v ∈ G.Adj[u]             // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                        // finish u
```



time = 3

Pseudocode of DFS

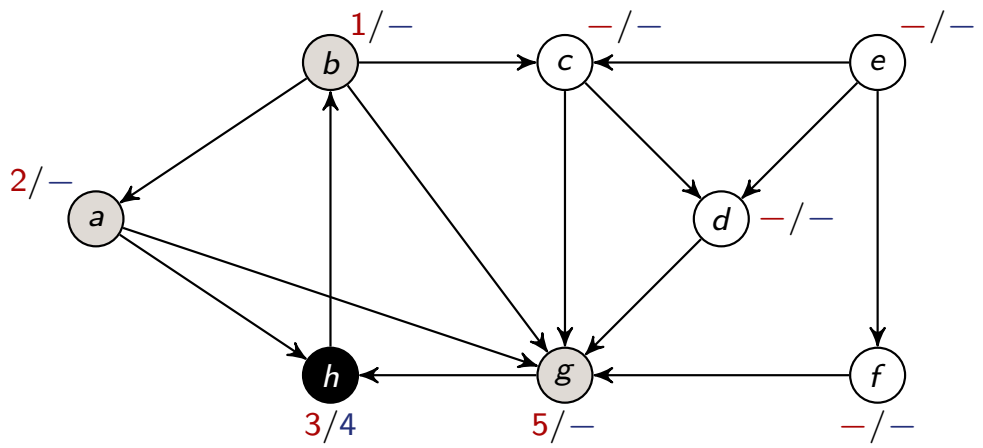
```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                    // discover u
  for each v ∈ G.Adj[u]             // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                        // finish u
```



time = 4

Pseudocode of DFS

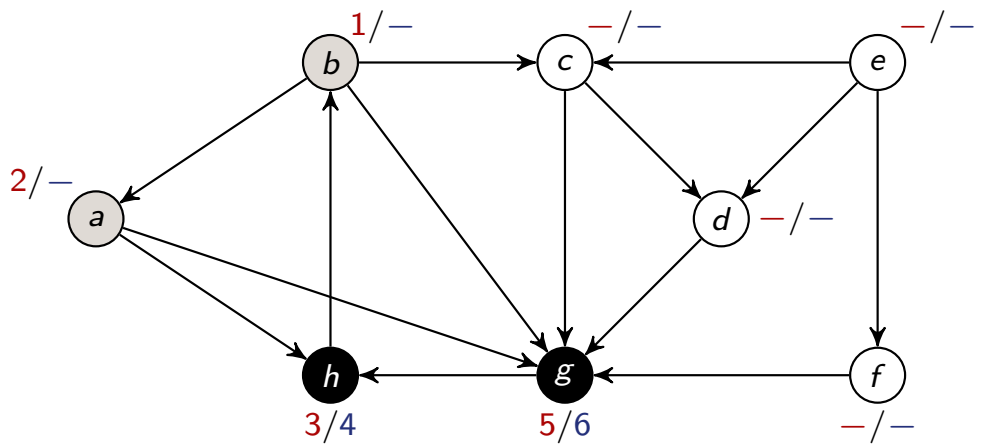
```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                    // discover u
  for each v ∈ G.Adj[u]             // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                        // finish u
```



time = 5

Pseudocode of DFS

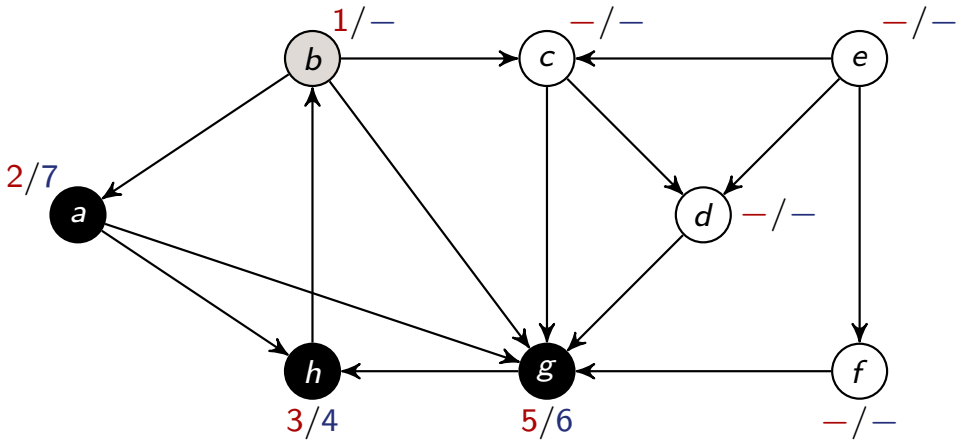
```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                    // discover u
  for each v ∈ G.Adj[u]             // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                        // finish u
```



time = 6

Pseudocode of DFS

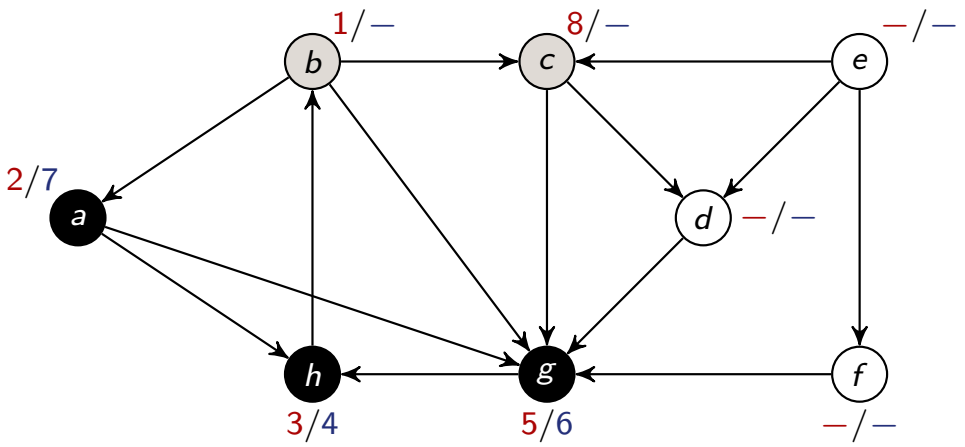
```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                    // discover u
  for each v ∈ G.Adj[u]             // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                        // finish u
```



time = 7

Pseudocode of DFS

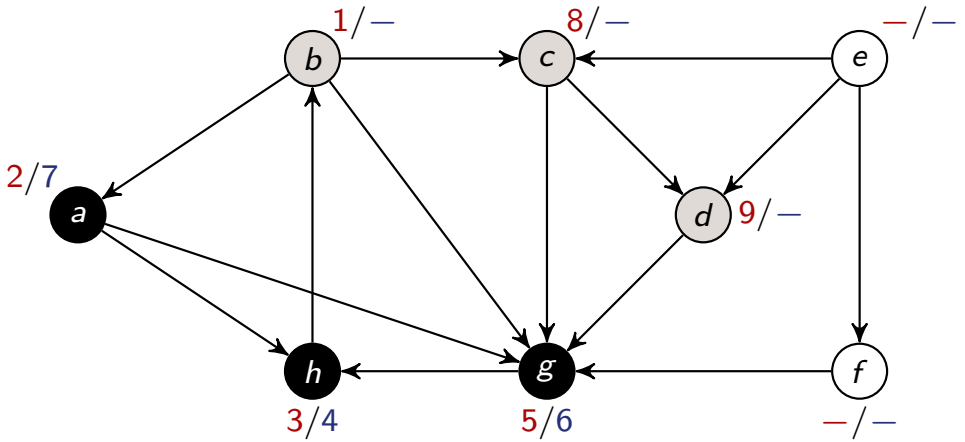
```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                    // discover u
  for each v ∈ G.Adj[u]             // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                        // finish u
```



time = 8

Pseudocode of DFS

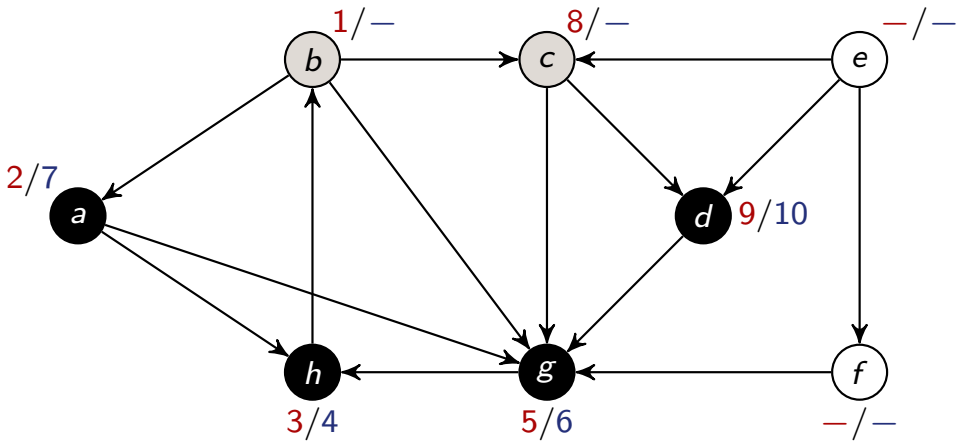
```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                    // discover u
  for each v ∈ G.Adj[u]             // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                        // finish u
```



time = 9

Pseudocode of DFS

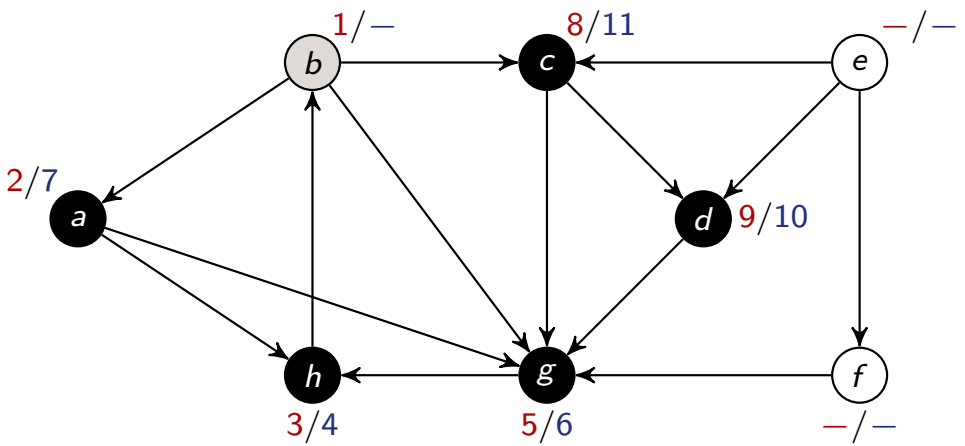
```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                    // discover u
  for each v ∈ G.Adj[u]             // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                        // finish u
```



time = 10

Pseudocode of DFS

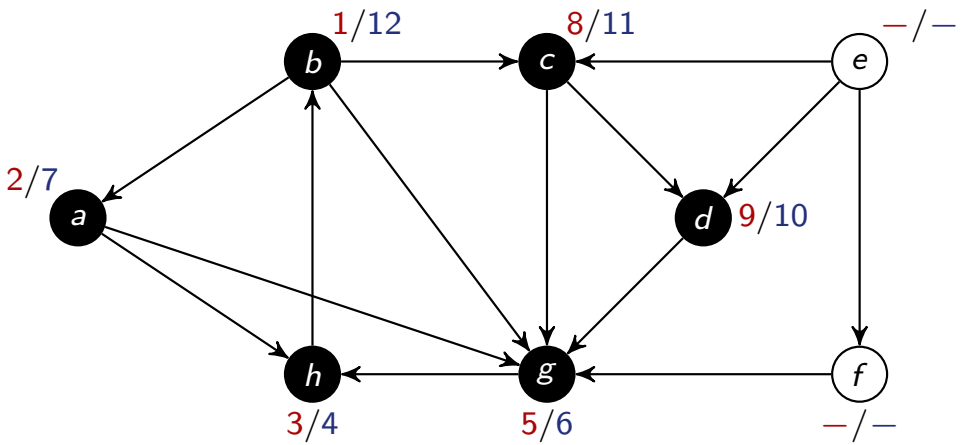
```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                    // discover u
  for each v ∈ G.Adj[u]             // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                        // finish u
```



time = 11

Pseudocode of DFS

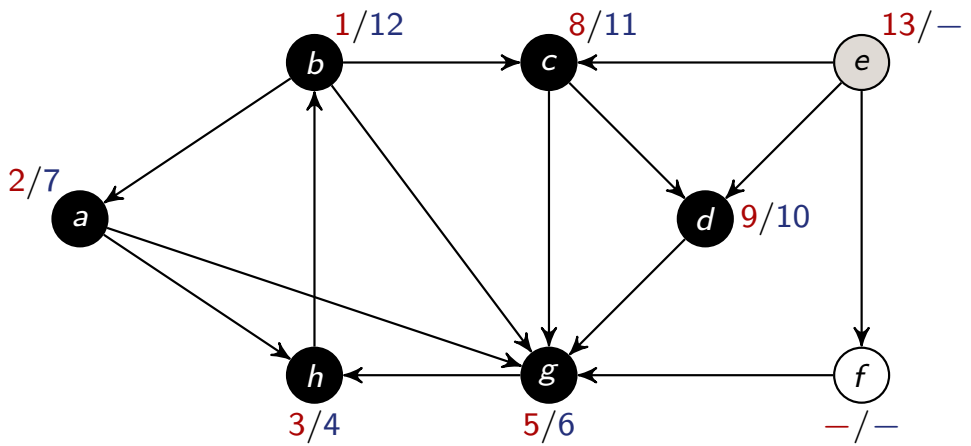
```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                    // discover u
  for each v ∈ G.Adj[u]             // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                        // finish u
```



time = 12

Pseudocode of DFS

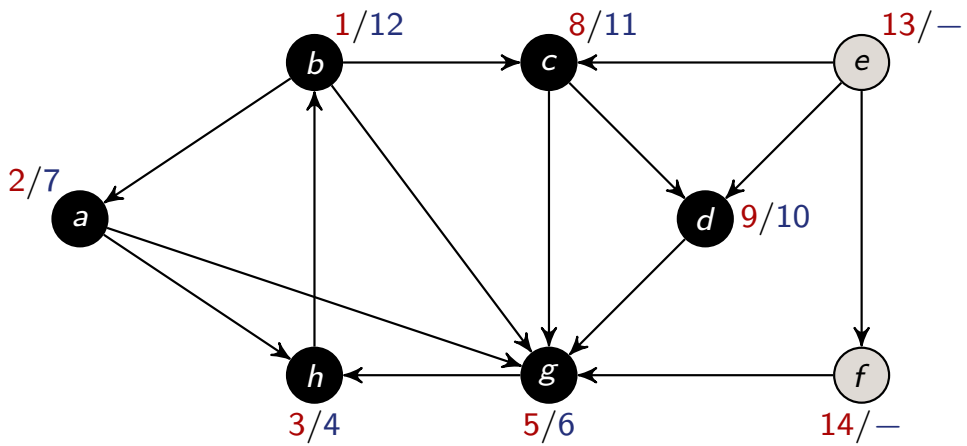
```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                    // discover u
  for each v ∈ G.Adj[u]             // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                        // finish u
```



time = 13

Pseudocode of DFS

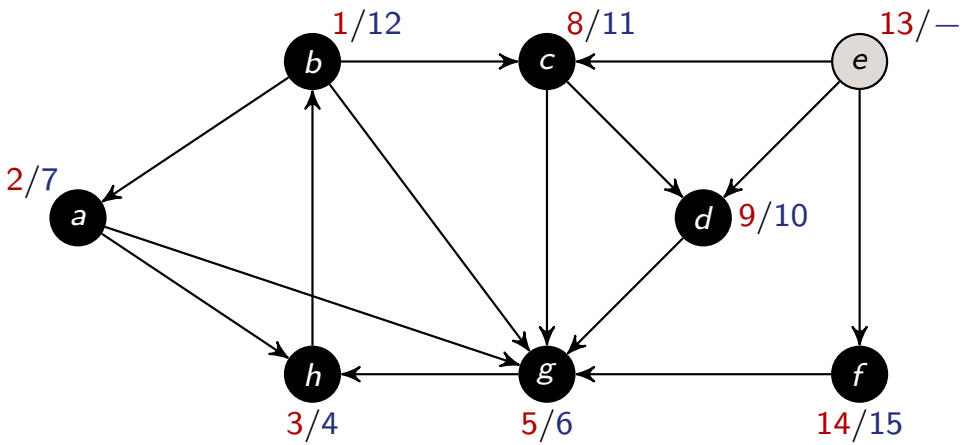
```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                    // discover u
  for each v ∈ G.Adj[u]             // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                        // finish u
```



time = 14

Pseudocode of DFS

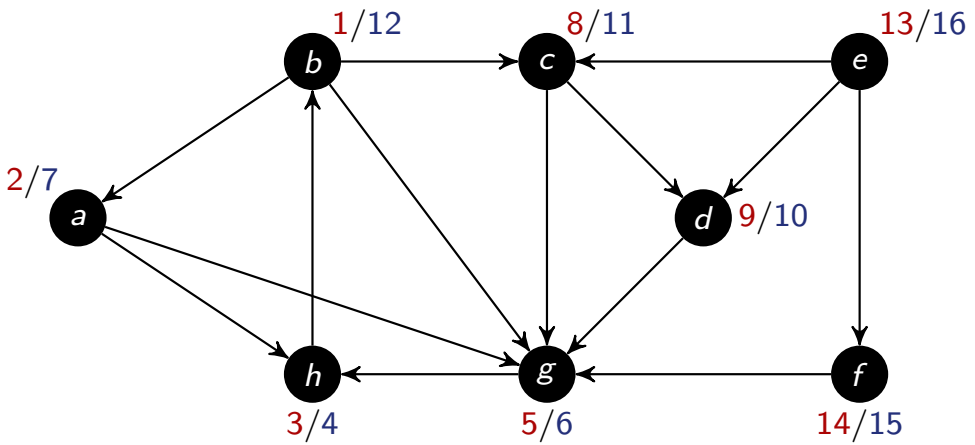
```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                    // discover u
  for each v ∈ G.Adj[u]             // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                        // finish u
```



time = 15

Pseudocode of DFS

```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                    // discover u
  for each v ∈ G.Adj[u]             // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                        // finish u
```



time = 16

Runtime Analysis

```
DFS(G)
  for each u ∈ G.V
      u.color = WHITE
  time = 0
  for each u ∈ G.V
      if u.color == WHITE
          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                  // discover u
  for each v ∈ G.Adj[u]           // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                      // finish u
```

Runtime Analysis

```
DFS(G)
  for each u ∈ G.V
      u.color = WHITE
  time = 0
  for each u ∈ G.V
      if u.color == WHITE
          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                  // discover u
  for each v ∈ G.Adj[u]           // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                      // finish u
```

- Color each vertex white takes time $\Theta(V)$

Runtime Analysis

```
DFS(G)
  for each u ∈ G.V
      u.color = WHITE
  time = 0
  for each u ∈ G.V
      if u.color == WHITE
          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                  // discover u
  for each v ∈ G.Adj[u]           // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                      // finish u
```

- Color each vertex white takes time $\Theta(V)$
- Note that DFS-VISIT is called once for each vertex (when it is colored gray from white)

Runtime Analysis

```
DFS(G)
  for each u ∈ G.V
      u.color = WHITE
  time = 0
  for each u ∈ G.V
      if u.color == WHITE
          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                  // discover u
  for each v ∈ G.Adj[u]           // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                      // finish u
```

- Color each vertex white takes time $\Theta(V)$
- Note that DFS-VISIT is called once for each vertex (when it is colored gray from white)
- When DFS-VISIT(u) is called the **for** loop runs at most {#neighbors of u} times

Runtime Analysis

```
DFS(G)
  for each u ∈ G.V
      u.color = WHITE
  time = 0
  for each u ∈ G.V
      if u.color == WHITE
          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                  // discover u
  for each v ∈ G.Adj[u]           // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                      // finish u
```

- Color each vertex white takes time $\Theta(V)$
- Note that DFS-VISIT is called once for each vertex (when it is colored gray from white)
- When DFS-VISIT(u) is called the **for** loop runs at most {#neighbors of u} times
- Therefore the total time DFS-VISIT is run is

$$\sum_{u \in V} \{\#\text{neighbors of } u\} = \Theta(E)$$

Runtime Analysis

```
DFS(G)
  for each u ∈ G.V
      u.color = WHITE
  time = 0
  for each u ∈ G.V
      if u.color == WHITE
          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY                  // discover u
  for each v ∈ G.Adj[u]           // explore (u, v)
      if v.color == WHITE
          DFS-VISIT(v)
  u.color = BLACK
  time = time + 1
  u.f = time                      // finish u
```

- Color each vertex white takes time $\Theta(V)$
- Note that DFS-VISIT is called once for each vertex (when it is colored gray from white)
- When DFS-VISIT(u) is called the **for** loop runs at most {#neighbors of u} times
- Therefore the total time DFS-VISIT is run is

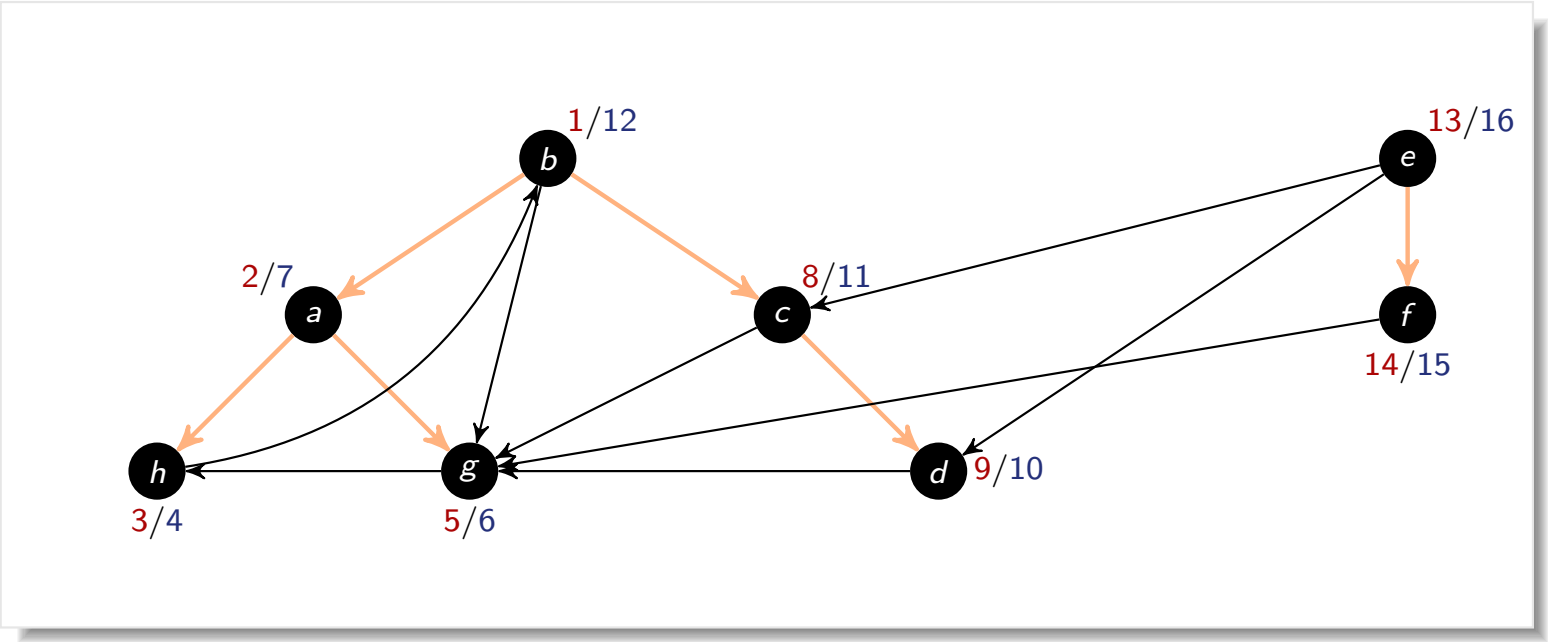
$$\sum_{u \in V} \{\#\text{neighbors of } u\} = \Theta(E)$$

Total Time: $\Theta(V + E)$

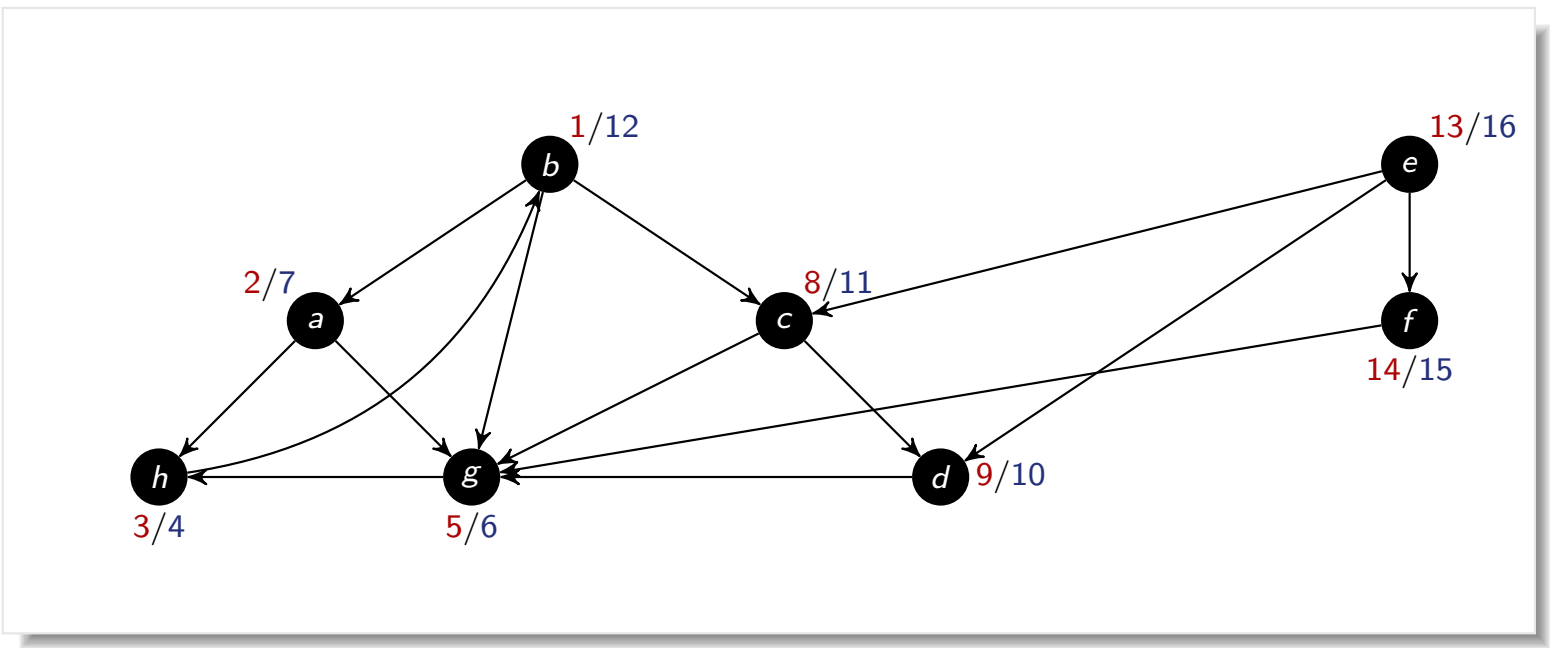
PROPERTIES OF DFS

Output of DFS

DFS forms a depth-first forest comprised of > 1 depth-first trees. Each tree is made of edges (u, v) such that u is gray and v is white when (u, v) is explored.

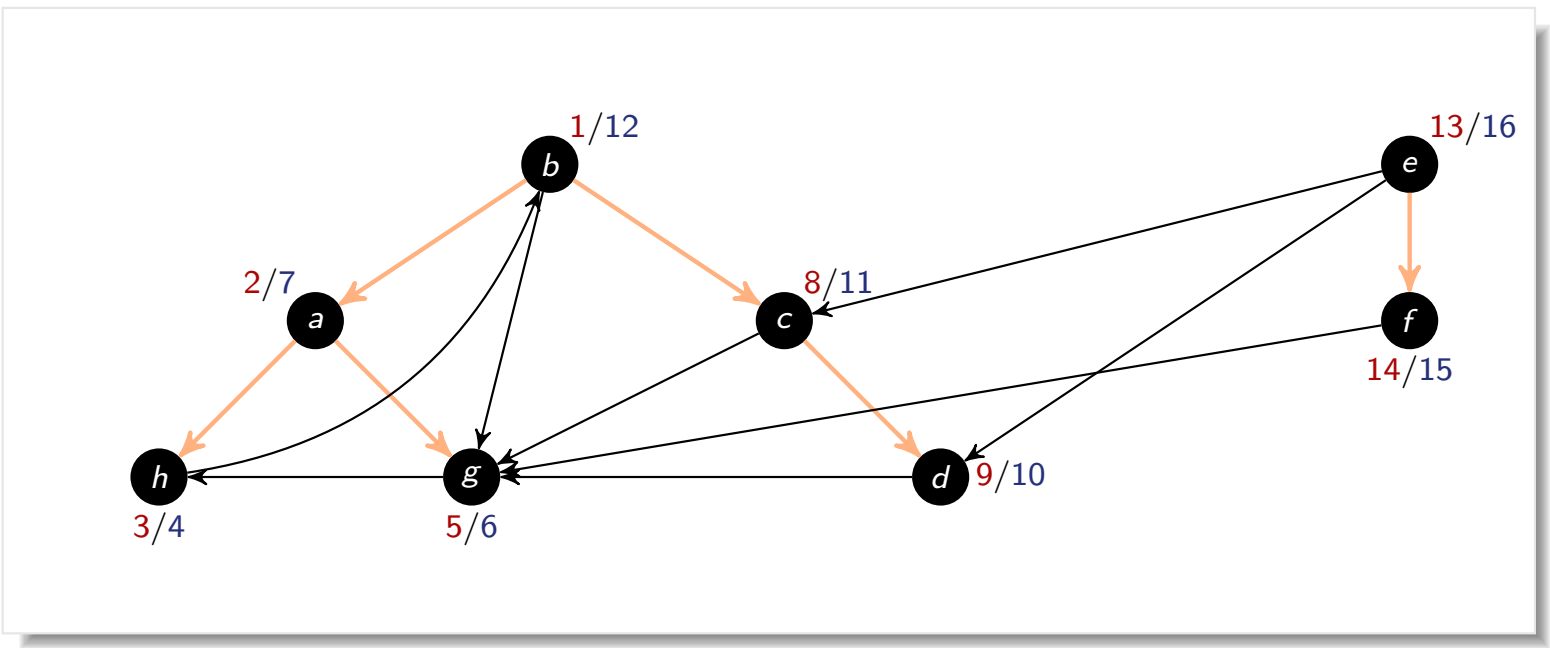


Classification of edges



Classification of edges

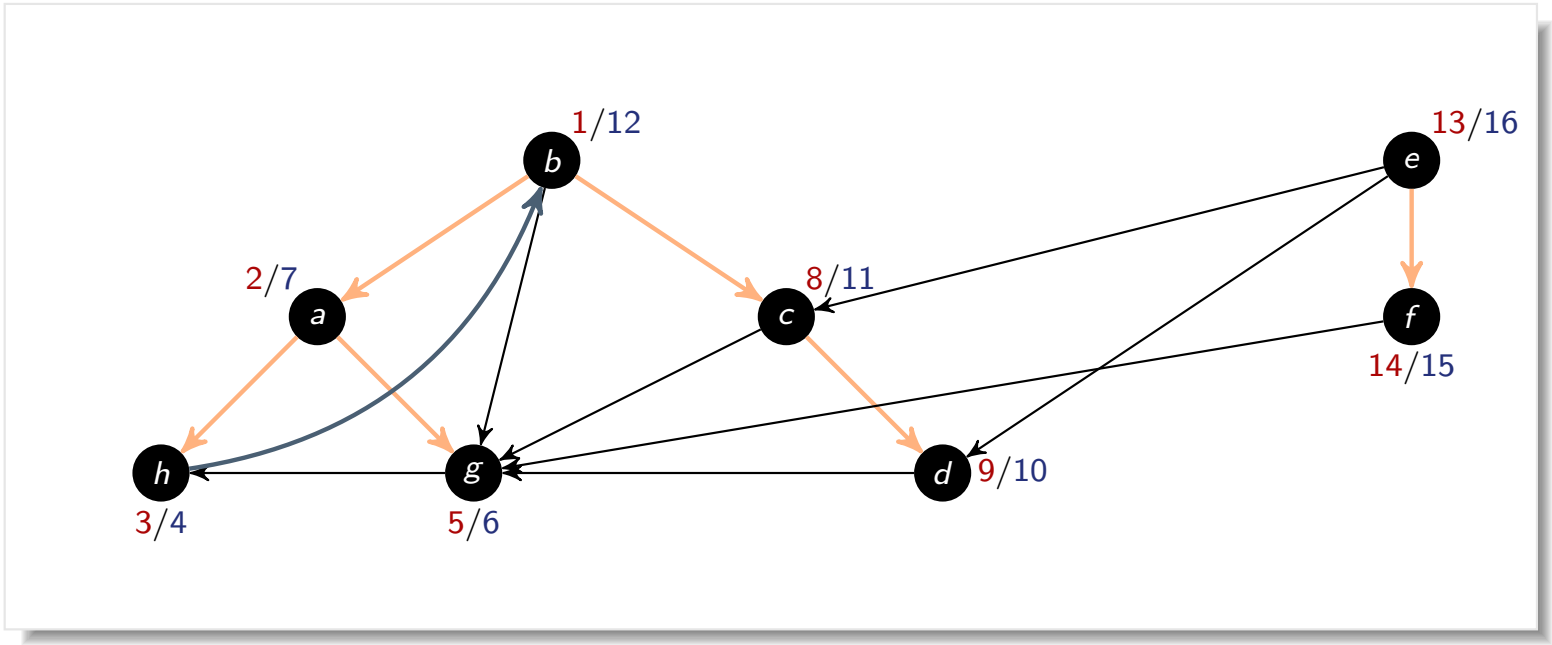
Tree edge: In the depth-first forest, found by exploring (u, v)



Classification of edges

Tree edge: In the depth-first forest, found by exploring (u, v)

Back edge: (u, v) where u is a descendant of v

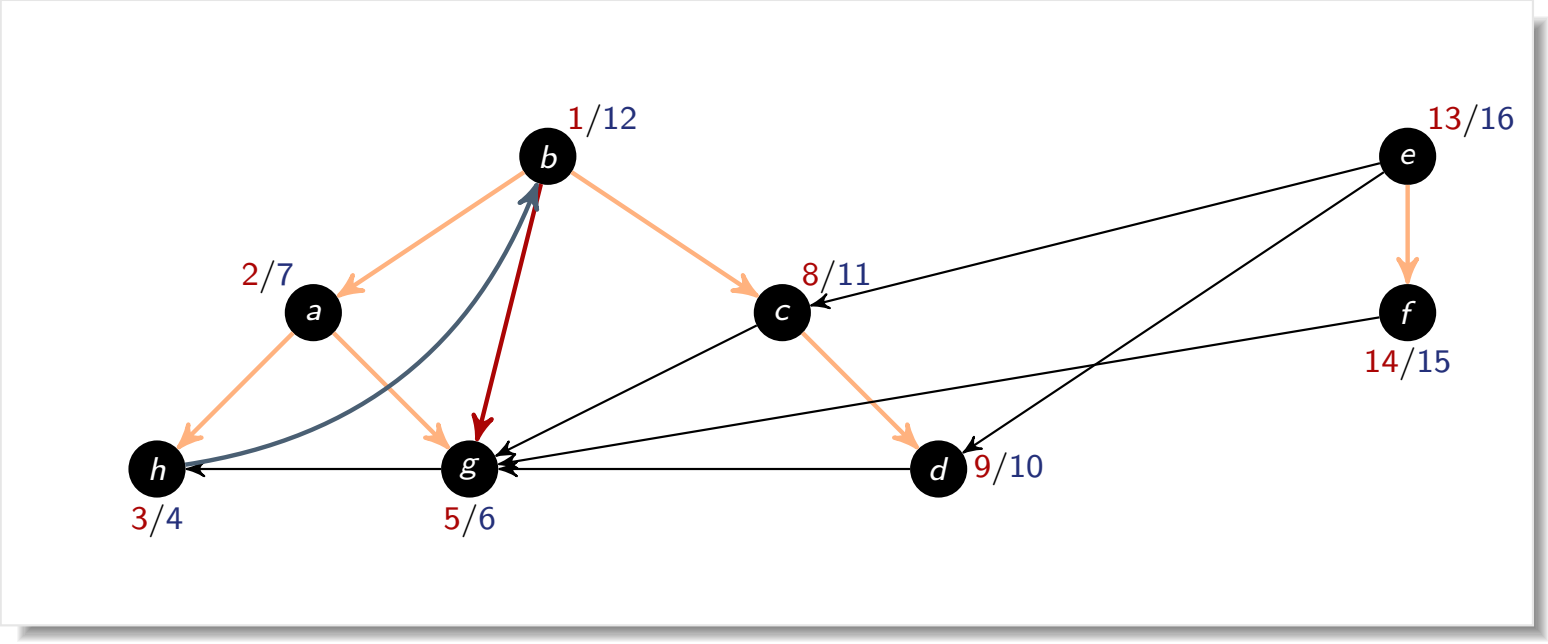


Classification of edges

Tree edge: In the depth-first forest, found by exploring (u, v)

Back edge: (u, v) where u is a descendant of v

Forward edge: (u, v) where v is a descendant of u, but not a tree edge



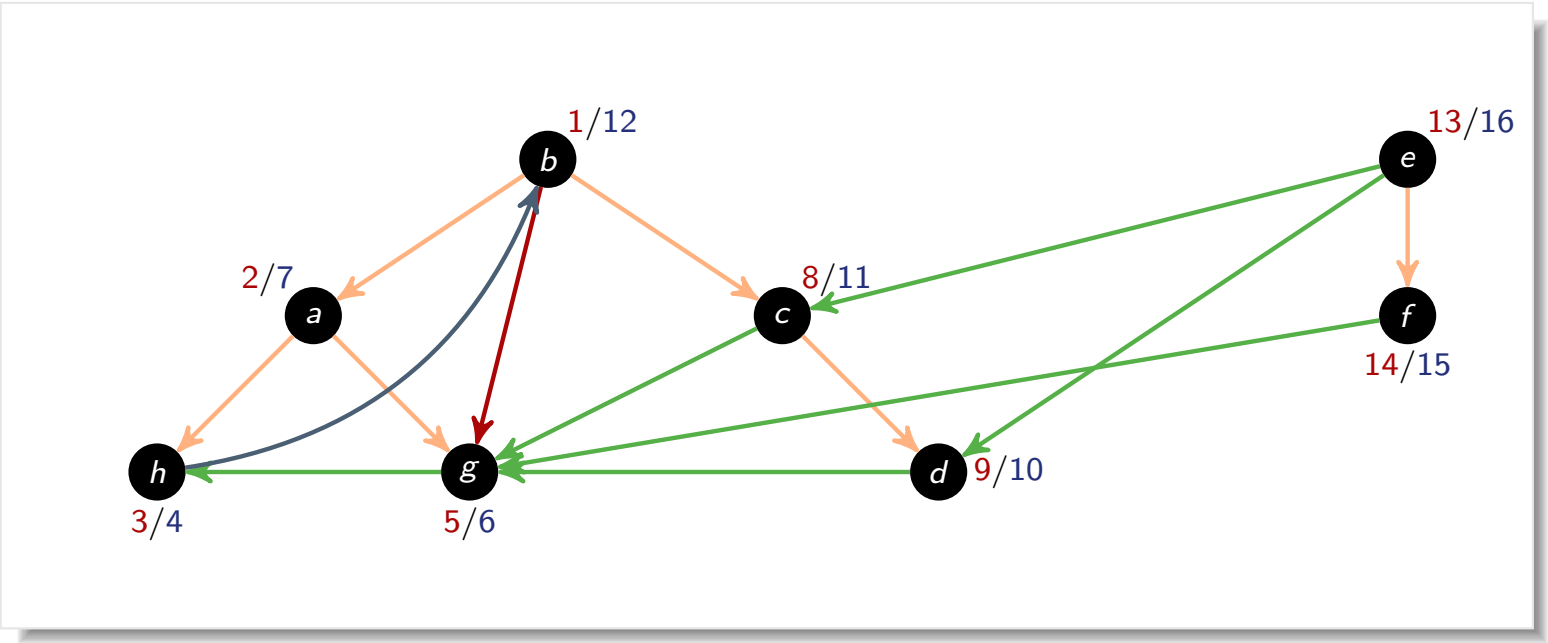
Classification of edges

Tree edge: In the depth-first forest, found by exploring (u, v)

Back edge: (u, v) where u is a descendant of v

Forward edge: (u, v) where v is a descendant of u, but not a tree edge

Cross edge: any other edge



Classification of edges

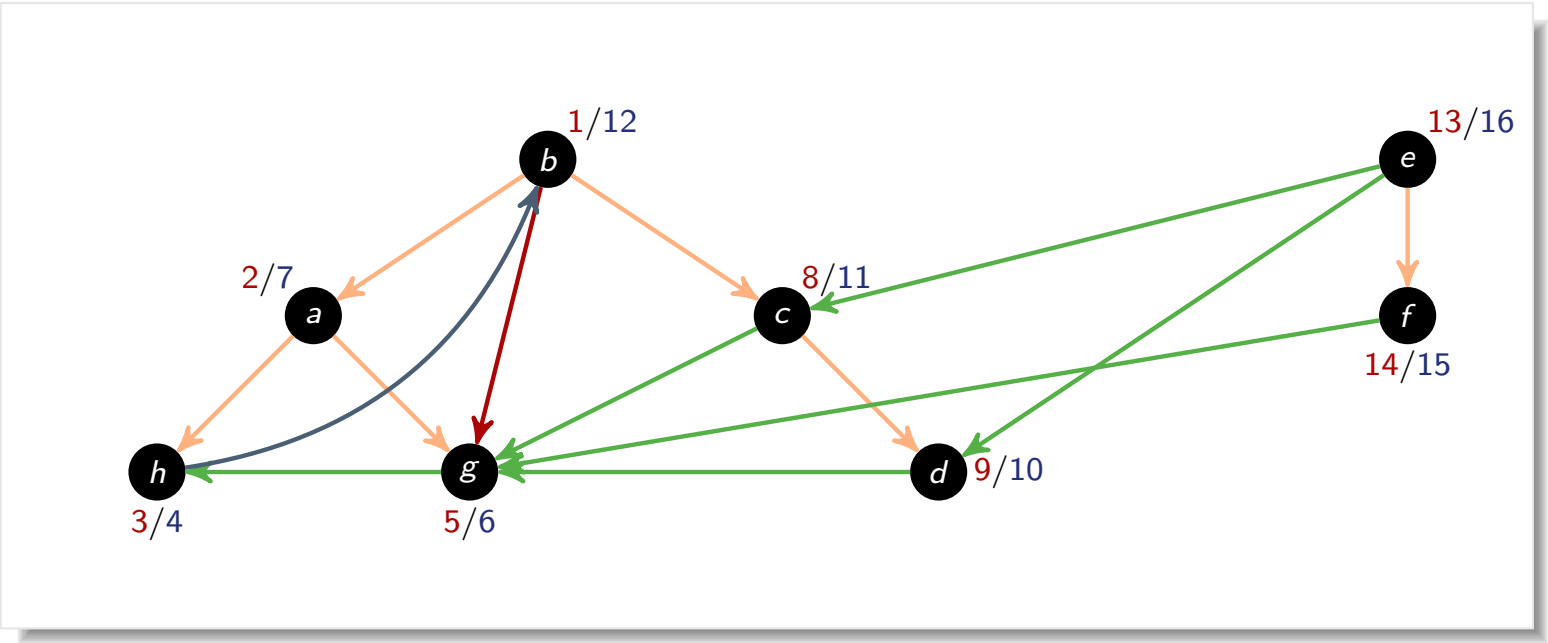
Tree edge: In the depth-first forest, found by exploring (u, v)

Back edge: (u, v) where u is a descendant of v

Forward edge: (u, v) where v is a descendant of u, but not a tree edge

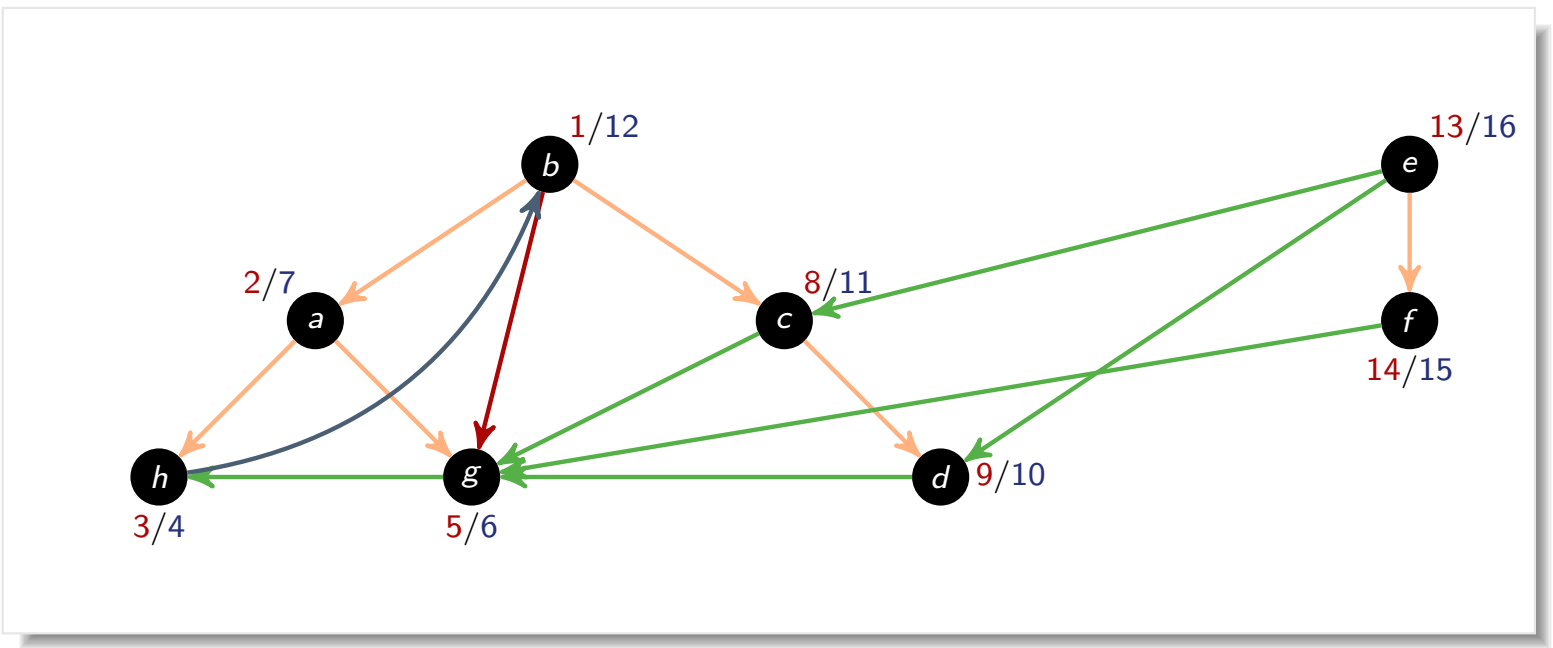
Cross edge: any other edge

In DFS of an undirected graph we get only tree and back/forward (we call them back) edges, no cross edges. Why?



Parenthesis theorem

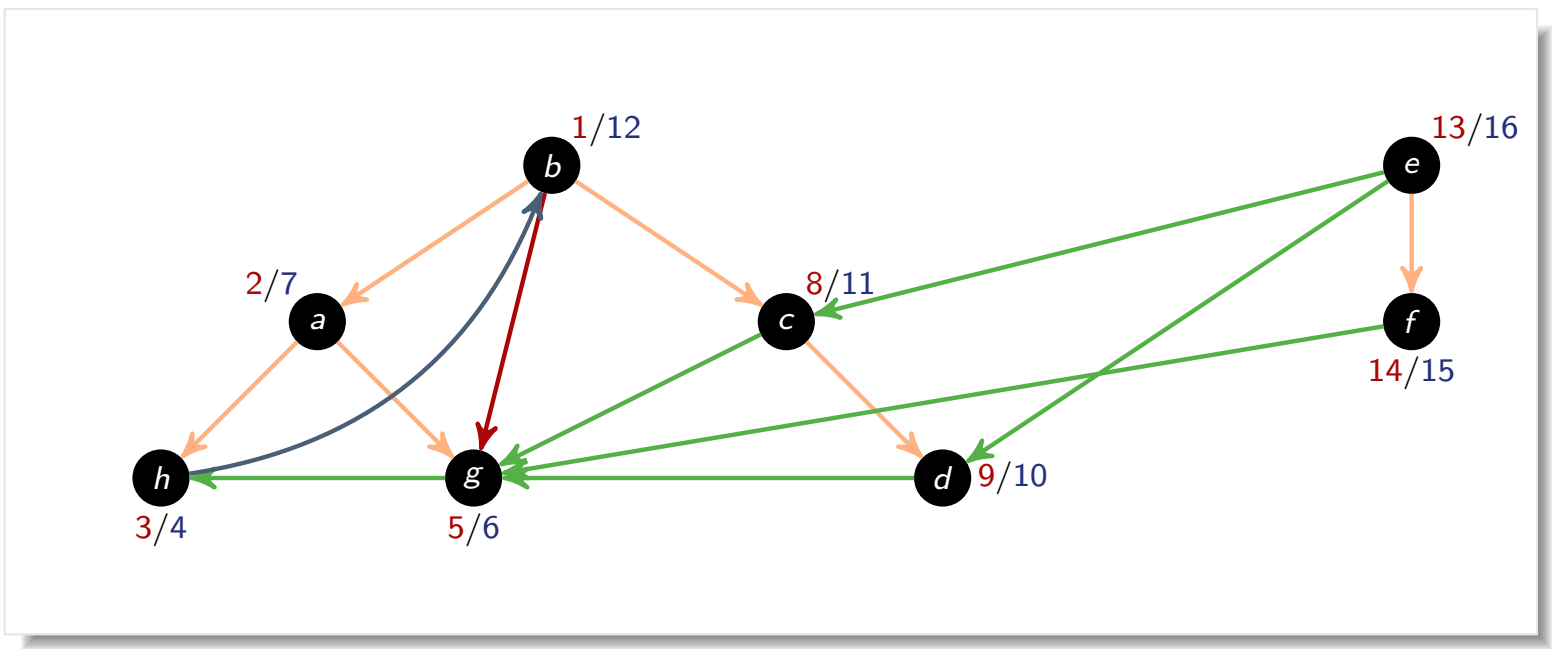
For all u, v exactly one of the following holds



Parenthesis theorem

For all u, v exactly one of the following holds

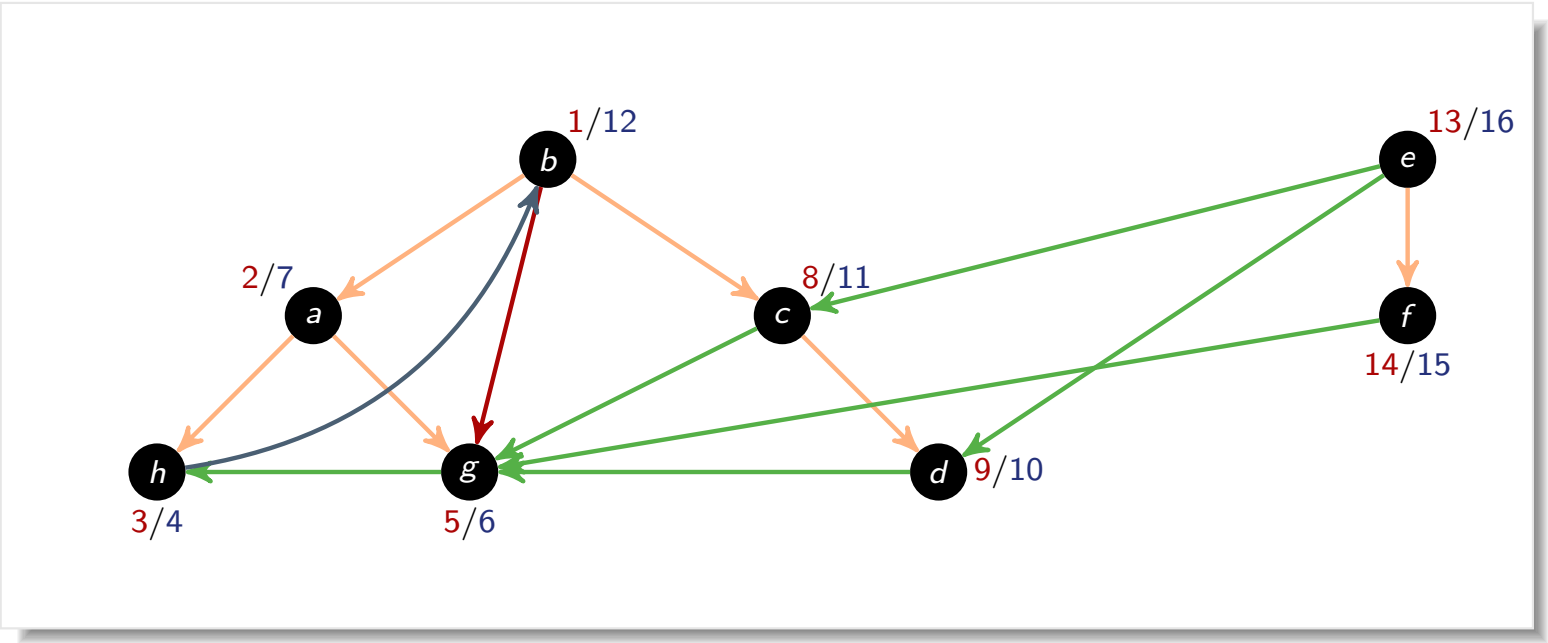
1. $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint neither of u and v are descendant of each other



Parenthesis theorem

For all u, v exactly one of the following holds

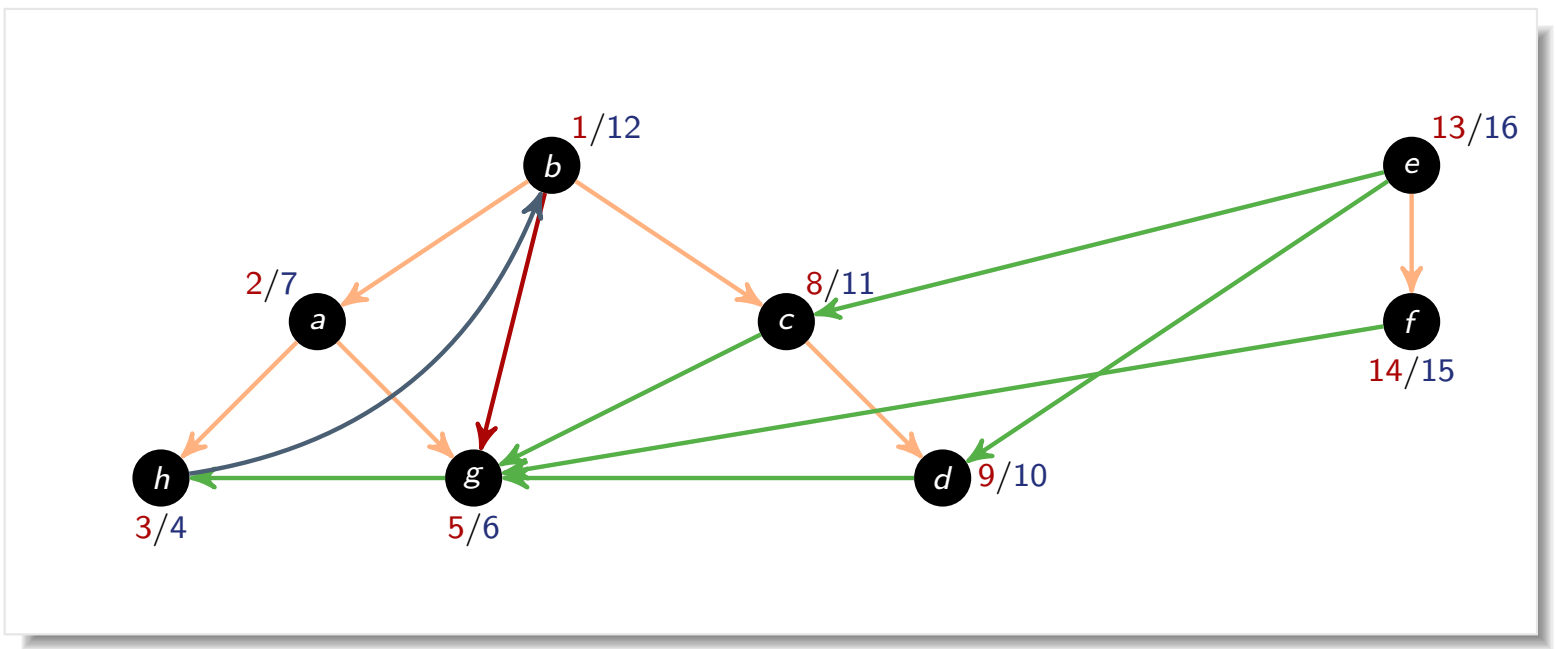
1. $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint neither of u and v are descendant of each other
2. $u.d < v.d < v.f < u.f$ and v is a descendant of u



Parenthesis theorem

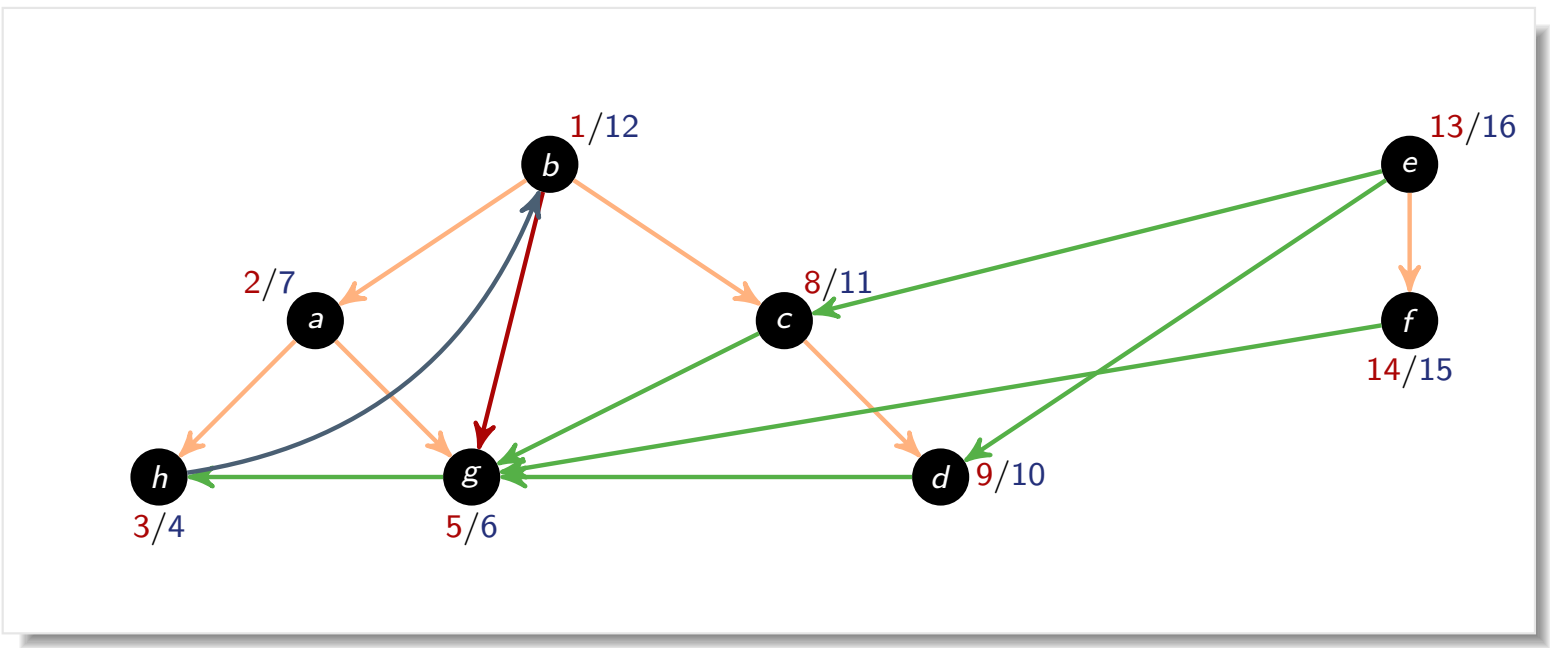
For all u, v exactly one of the following holds

1. $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint neither of u and v are descendant of each other
2. $u.d < v.d < v.f < u.f$ and v is a descendant of u
3. $v.d < u.d < u.f < v.f$ and u is a descendant of v.



White-path theorem

Vertex v is a descendant of u if and only if at time $u.d$ there is a path from u to v consisting of only white vertices (except for u, which was just colored gray)



TOPOLOGICAL SORT

Application of DFS

Lecture 15, 09.04.2024

Topological sort

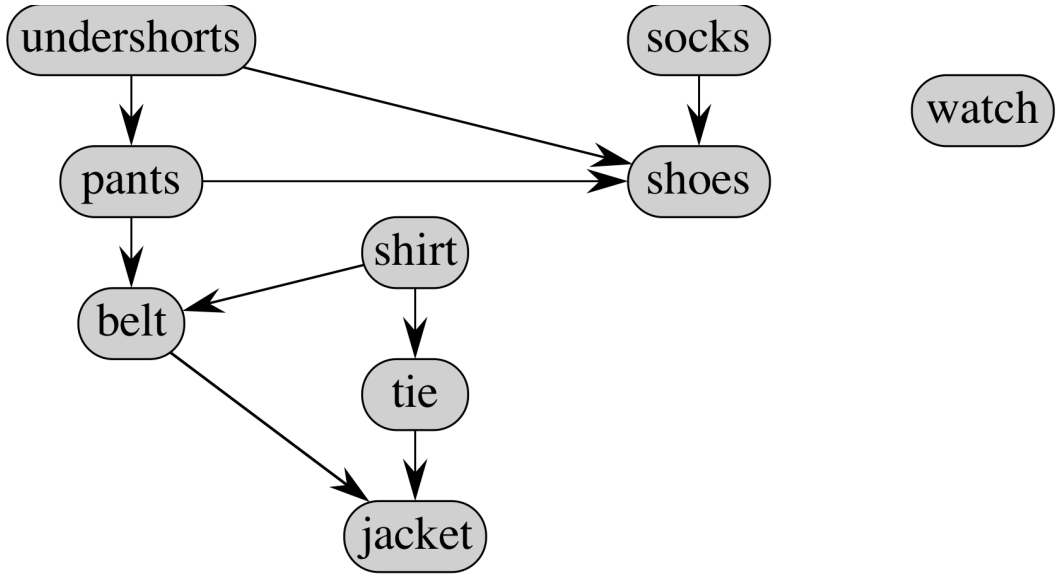
Definition

INPUT: A directed acyclic graph (DAG) $G = (V, E)$

OUTPUT: a linear ordering of vertices such that if $(u, v) \in E$, then u appears somewhere before v

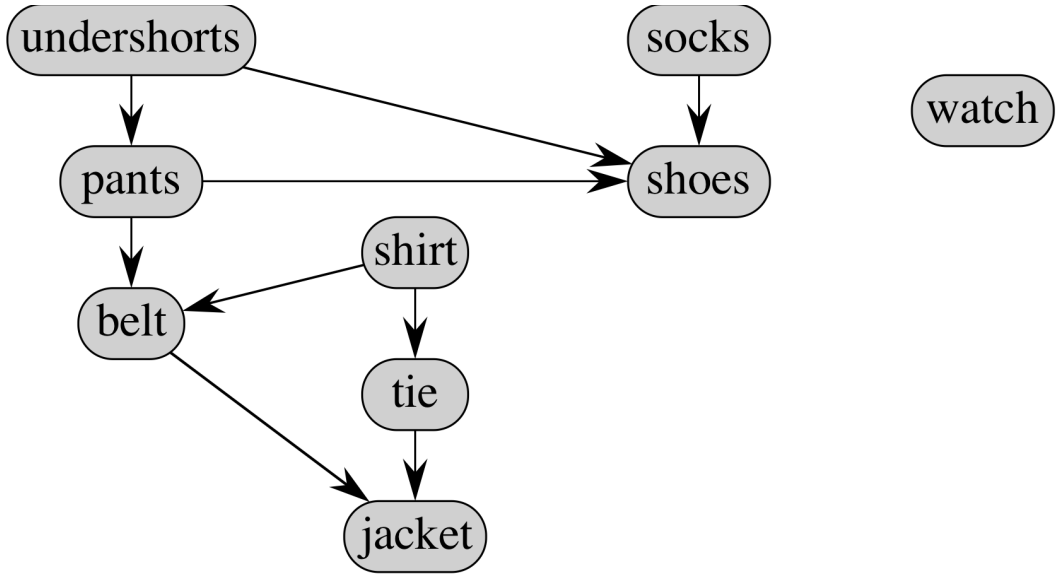
Example

Getting dressed in the morning:

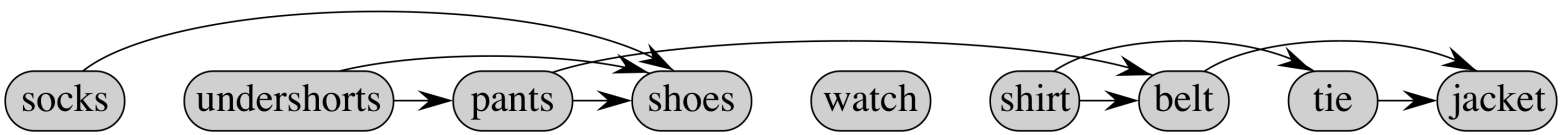


Example

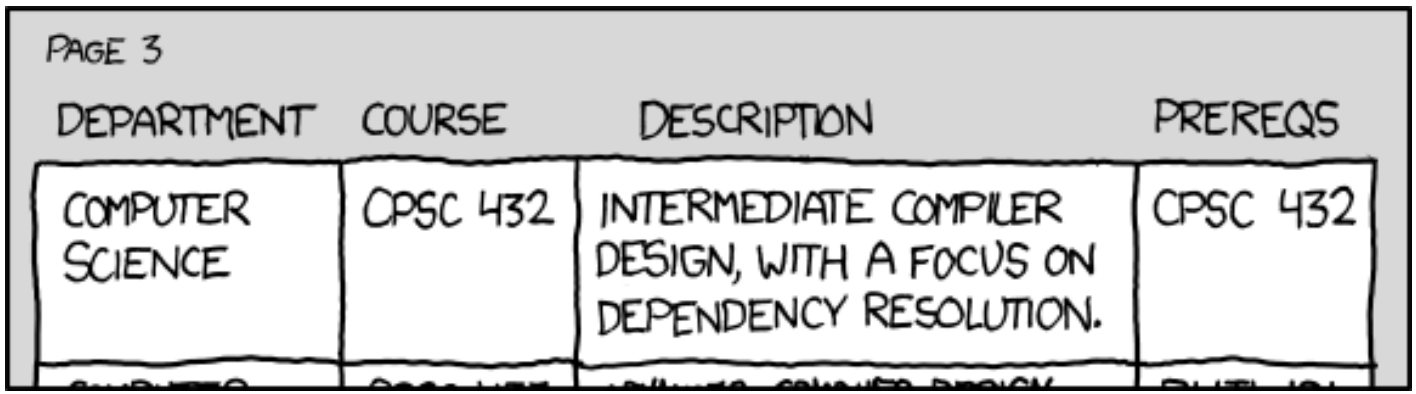
Getting dressed in the morning:



in which order?



First: when is a directed graph acyclic?



First: when is a directed graph acyclic?

First: when is a directed graph acyclic?

Lemma

A directed graph G is acyclic if and only if a DFS of G yields no back edges

First: when is a directed graph acyclic?

Lemma

A directed graph G is acyclic if and only if a DFS of G yields no back edges

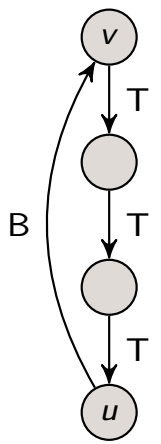
Proof. First show that back-edge implies cycle

First: when is a directed graph acyclic?

Lemma

A directed graph G is acyclic if and only if a DFS of G yields no back edges

Proof. First show that back-edge implies cycle
Suppose there is a back edge (u, v). Then v is ancestor of u in depth-first forest. Therefore there is a path from v to u, which creates a cycle.



First: when is a directed graph acyclic?

Lemma

A directed graph G is acyclic if and only if a DFS of G yields no back edges

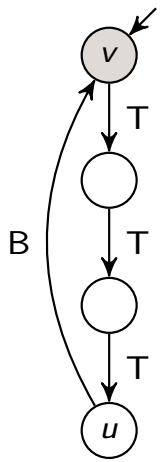
Proof. Second show that cycle implies back-edge

First: when is a directed graph acyclic?

Lemma

A directed graph G is acyclic if and only if a DFS of G yields no back edges

Proof. Second show that cycle implies back-edge
Let v be the first vertex discovered in the cycle C and let (u, v) be the preceding edge in C. At time $v.d$ vertices in C form a white-path from v to u and hence u is a descendant of v.



Algorithm for topological sort

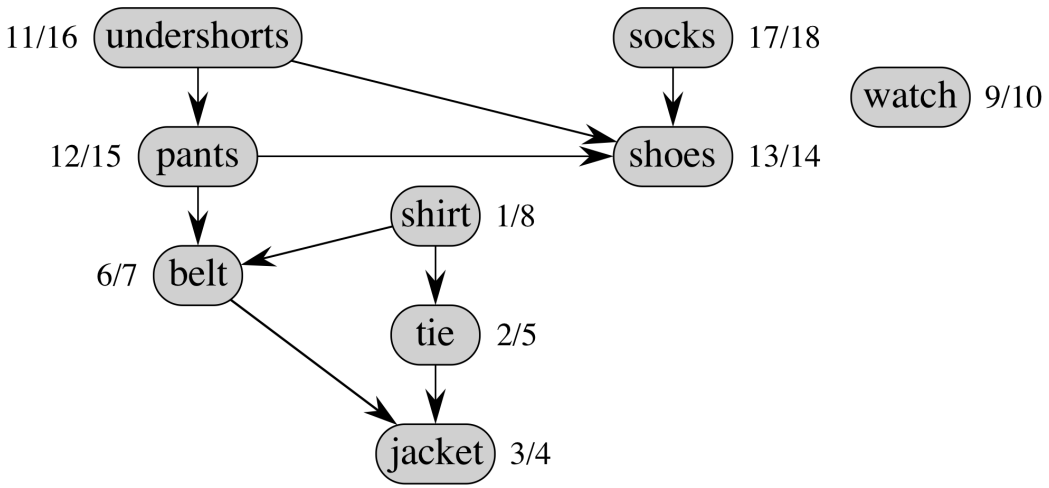
Algorithm for topological sort

TOPOLOGICAL-SORT(G):
1. Call $DFS(G)$ to compute finishing times $v.f$ for all $v \in G.V$
2. Output vertices in order of *decreasing* finishing times

Algorithm for topological sort

TOPOLOGICAL-SORT(G):
1. Call $DFS(G)$ to compute finishing times $v.f$ for all $v \in G.V$
2. Output vertices in order of *decreasing* finishing times

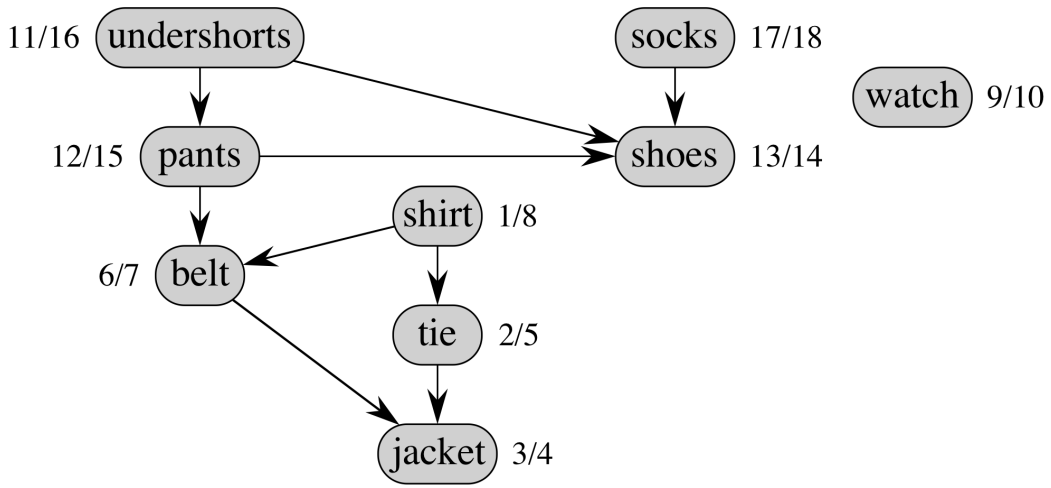
Example

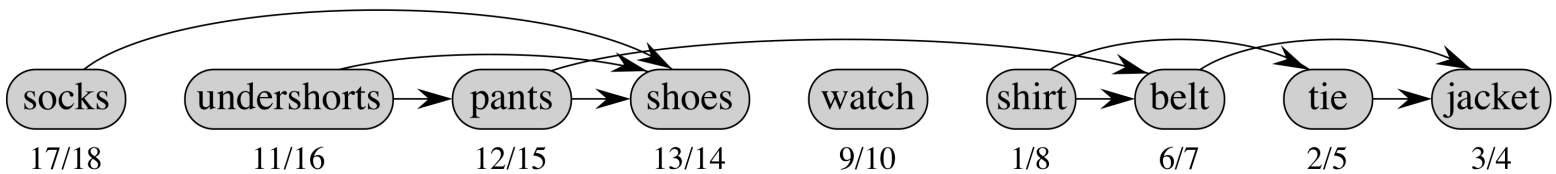


Algorithm for topological sort

TOPOLOGICAL-SORT(G):
1. Call $DFS(G)$ to compute finishing times $v.f$ for all $v \in G.V$
2. Output vertices in order of *decreasing* finishing times

Example





Time Analysis

TOPOLOGICAL-SORT(G):
1. Call $DFS(G)$ to compute finishing times $v.f$ for all $v \in G.V$
2. Output vertices in order of *decreasing* finishing times

Time Analysis

TOPOLOGICAL-SORT(G):
1. Call $DFS(G)$ to compute finishing times $v.f$ for all $v \in G.V$
2. Output vertices in order of *decreasing* finishing times

Do not need to sort by finishing times

- Can just output vertices as they are finished and understand that we want the reverse of the list

Time Analysis

TOPOLOGICAL-SORT(G):
1. Call $DFS(G)$ to compute finishing times $v.f$ for all $v \in G.V$
2. Output vertices in order of *decreasing* finishing times

Do not need to sort by finishing times

- Can just output vertices as they are finished and understand that we want the reverse of the list
- Or put them onto the front of a linked list as they are finished. When done, the list contains vertices in topologically sorted order.

Time:

Time Analysis

TOPOLOGICAL-SORT(G):
1. Call $DFS(G)$ to compute finishing times $v.f$ for all $v \in G.V$
2. Output vertices in order of *decreasing* finishing times

Do not need to sort by finishing times

- Can just output vertices as they are finished and understand that we want the reverse of the list
- Or put them onto the front of a linked list as they are finished. When done, the list contains vertices in topologically sorted order.

Time: $\Theta(V + E)$ (same as DFS)

Correctness

Need to show that if $(u, v) \in E$ then $v.f < u.f$

When we explore (u, v) what are the colors of u and v?

Correctness

Need to show that if $(u, v) \in E$ then $v.f < u.f$

When we explore (u, v) what are the colors of u and v?

- u is gray

Correctness

Need to show that if $(u, v) \in E$ then $v.f < u.f$

When we explore (u, v) what are the colors of u and v?

- u is gray
- Is v gray, too?

Correctness

Need to show that if $(u, v) \in E$ then $v.f < u.f$

When we explore (u, v) what are the colors of u and v?

- u is gray
- Is v gray, too?
  - No, because then v would be ancestor of u which implies that there is a back edge so the graph is not acyclic (by previous Lemma)

Correctness

Need to show that if $(u, v) \in E$ then $v.f < u.f$

When we explore (u, v) what are the colors of u and v?

- u is gray
- Is v gray, too?
  - No, because then v would be ancestor of u which implies that there is a back edge so the graph is not acyclic (by previous Lemma)
- Is v white?
  - Then becomes descendant of u. By parenthesis theorem, $u.d < v.d < v.f < u.f$

Correctness

Need to show that if $(u, v) \in E$ then $v.f < u.f$

When we explore (u, v) what are the colors of u and v?

- u is gray
- Is v gray, too?
  - No, because then v would be ancestor of u which implies that there is a back edge so the graph is not acyclic (by previous Lemma)
- Is v white?
  - Then becomes descendant of u. By parenthesis theorem, $u.d < v.d < v.f < u.f$
- Is v black?
  - Then v is already finished. Since we are exploring (u, v), we have not yet finished u. Therefore, $v.f < u.f$.

Correctness

Need to show that if $(u, v) \in E$ then $v.f < u.f$

When we explore (u, v) what are the colors of u and v?

- u is gray
- Is v gray, too?
  - No, because then v would be ancestor of u which implies that there is a back edge so the graph is not acyclic (by previous Lemma)
- Is v white?
  - Then becomes descendant of u. By parenthesis theorem, $u.d < v.d < v.f < u.f$
- Is v black?
  - Then v is already finished. Since we are exploring (u, v), we have not yet finished u. Therefore, $v.f < u.f$.

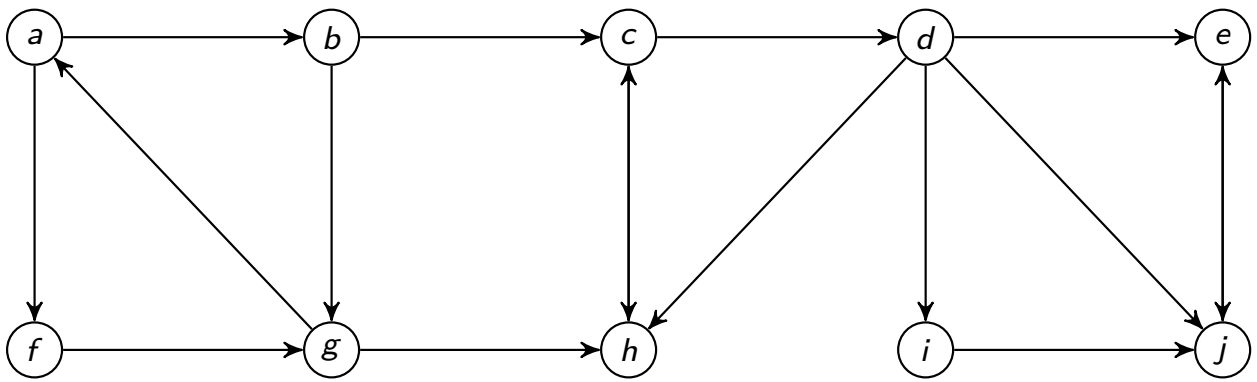
STRONGLY CONNECTED COMPONENTS
(A magic algorithm)

Lecture 15, 09.04.2024

What is a Strongly Connected Component?

Definition: A strongly connected component (SCC) of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

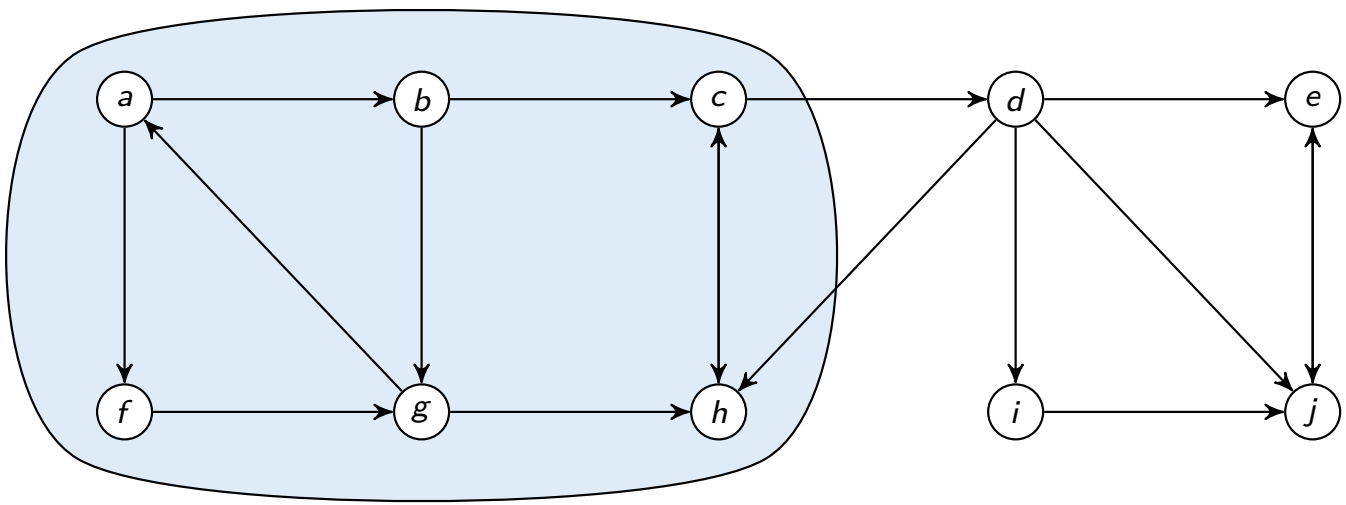
Example:



What is a Strongly Connected Component?

Definition: A strongly connected component (SCC) of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

Example:

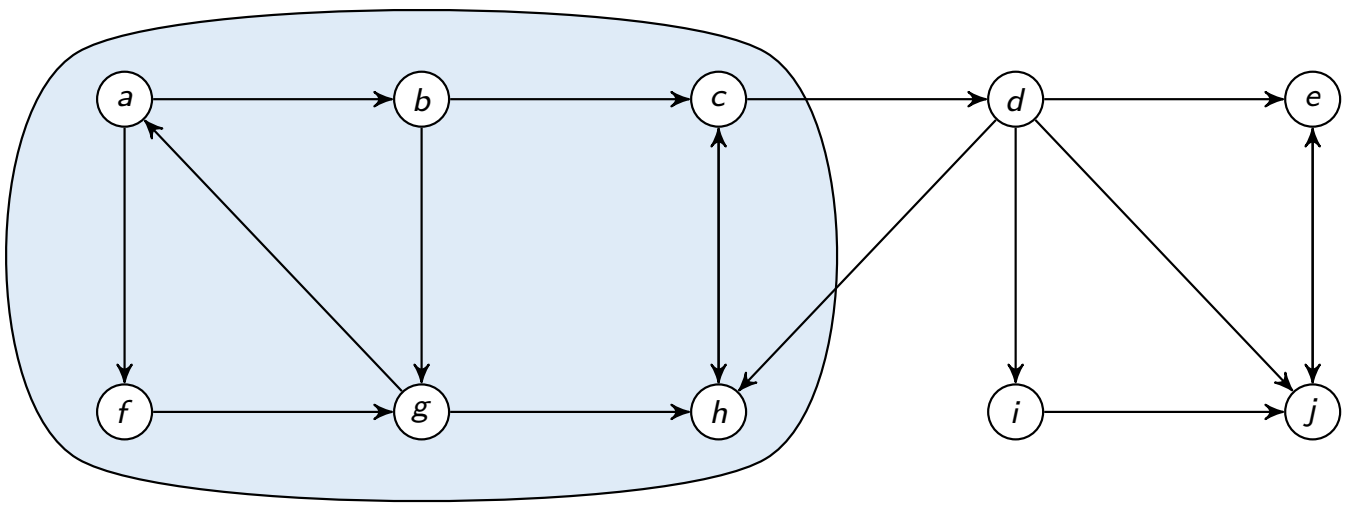


Is this a SCC?

What is a Strongly Connected Component?

Definition: A strongly connected component (SCC) of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

Example:

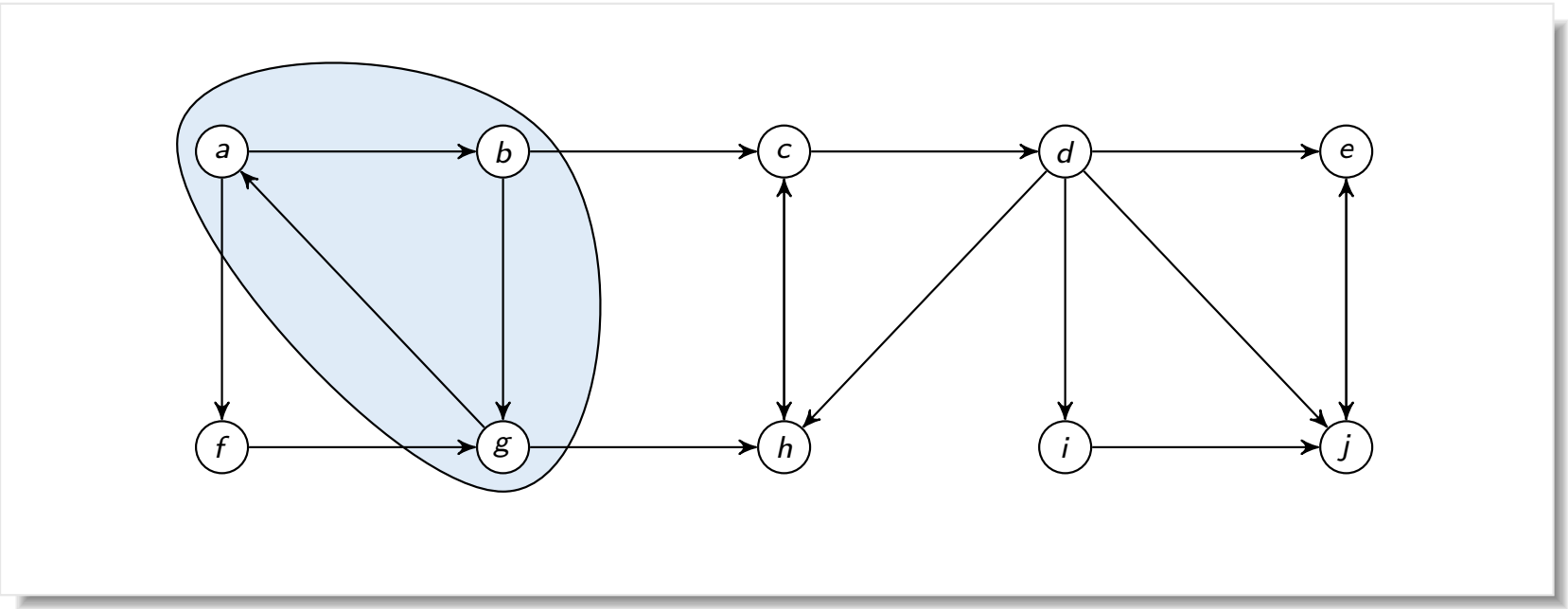


Is this a SCC? NO, because e.g. $c \not\rightsquigarrow b$

What is a Strongly Connected Component?

Definition: A strongly connected component (SCC) of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

Example:

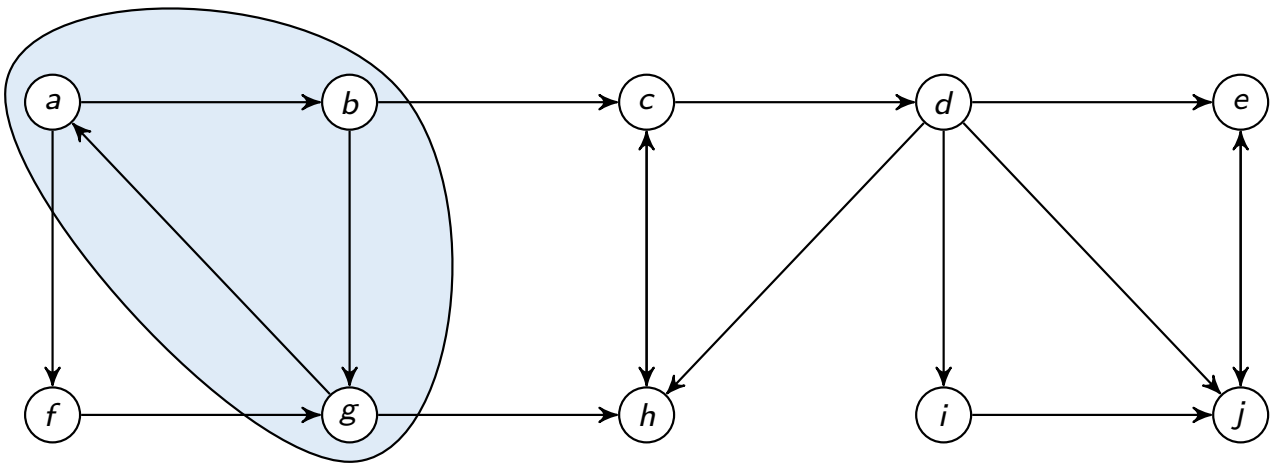


Is this a SCC?

What is a Strongly Connected Component?

Definition: A strongly connected component (SCC) of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

Example:

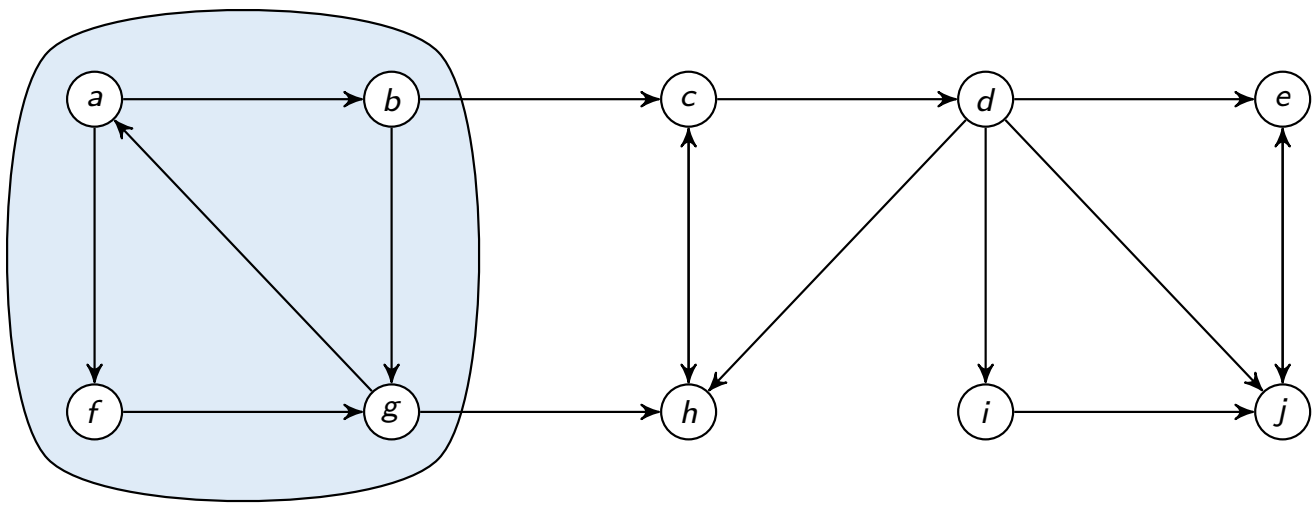


Is this a SCC? NO, because not maximal

What is a Strongly Connected Component?

Definition: A strongly connected component (SCC) of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

Example:

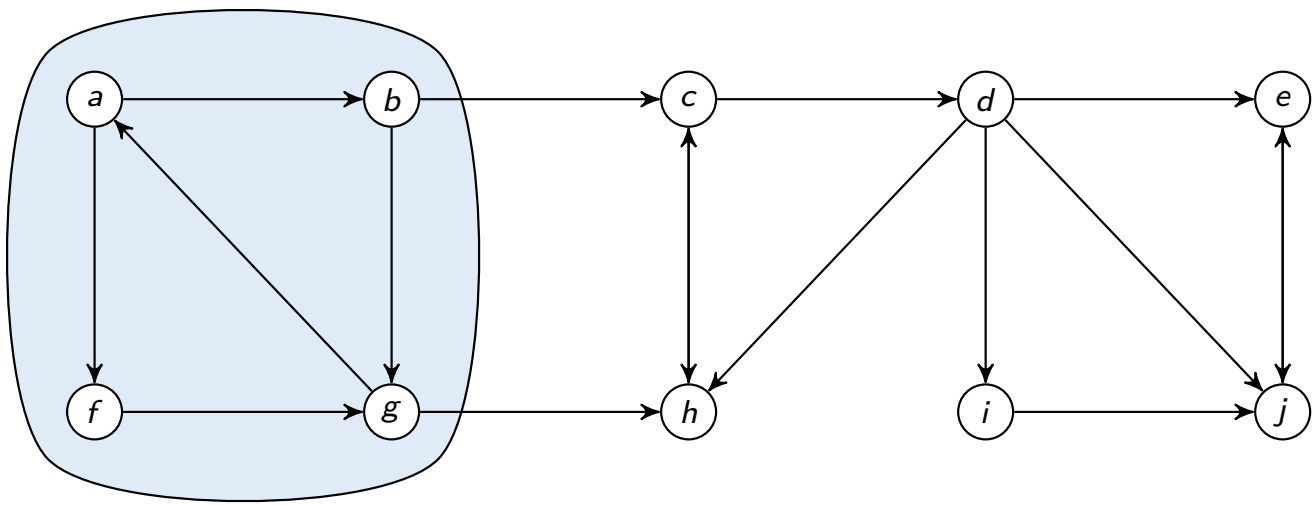


Is this a SCC?

What is a Strongly Connected Component?

Definition: A strongly connected component (SCC) of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

Example:

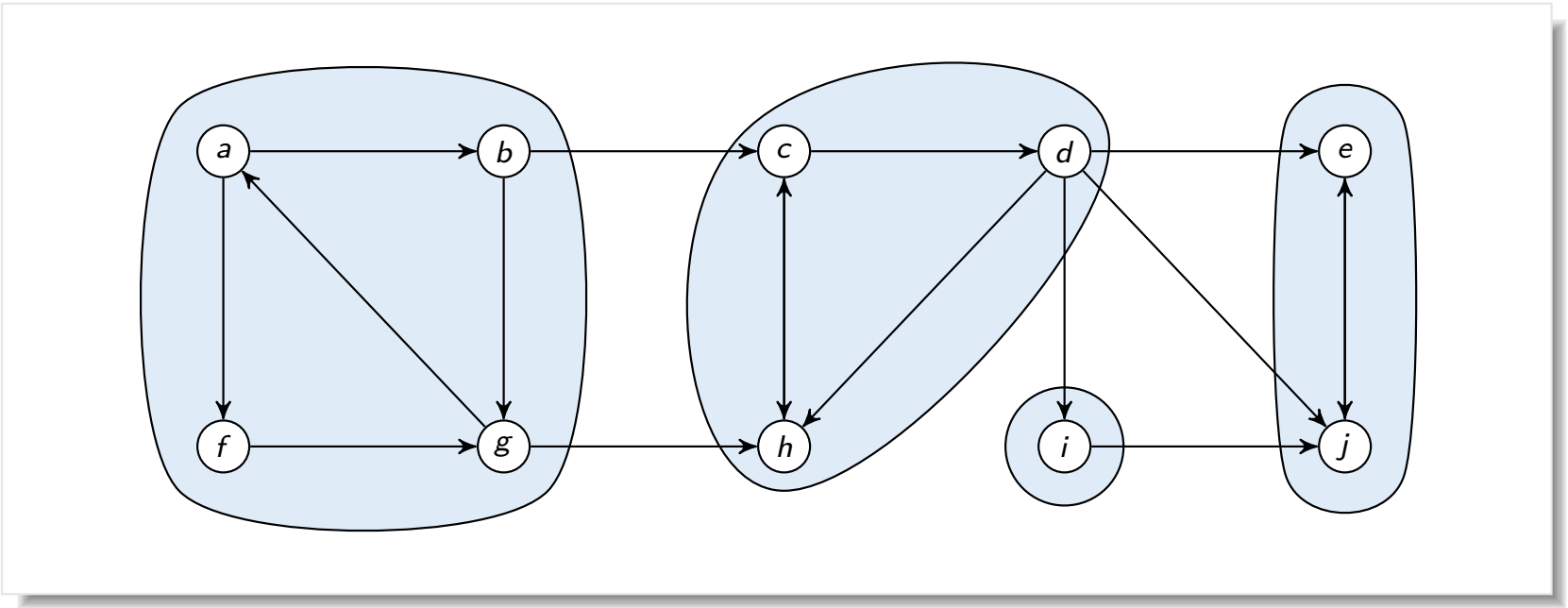


Is this a SCC? YES!

What is a Strongly Connected Component?

Definition: A strongly connected component (SCC) of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

Example:

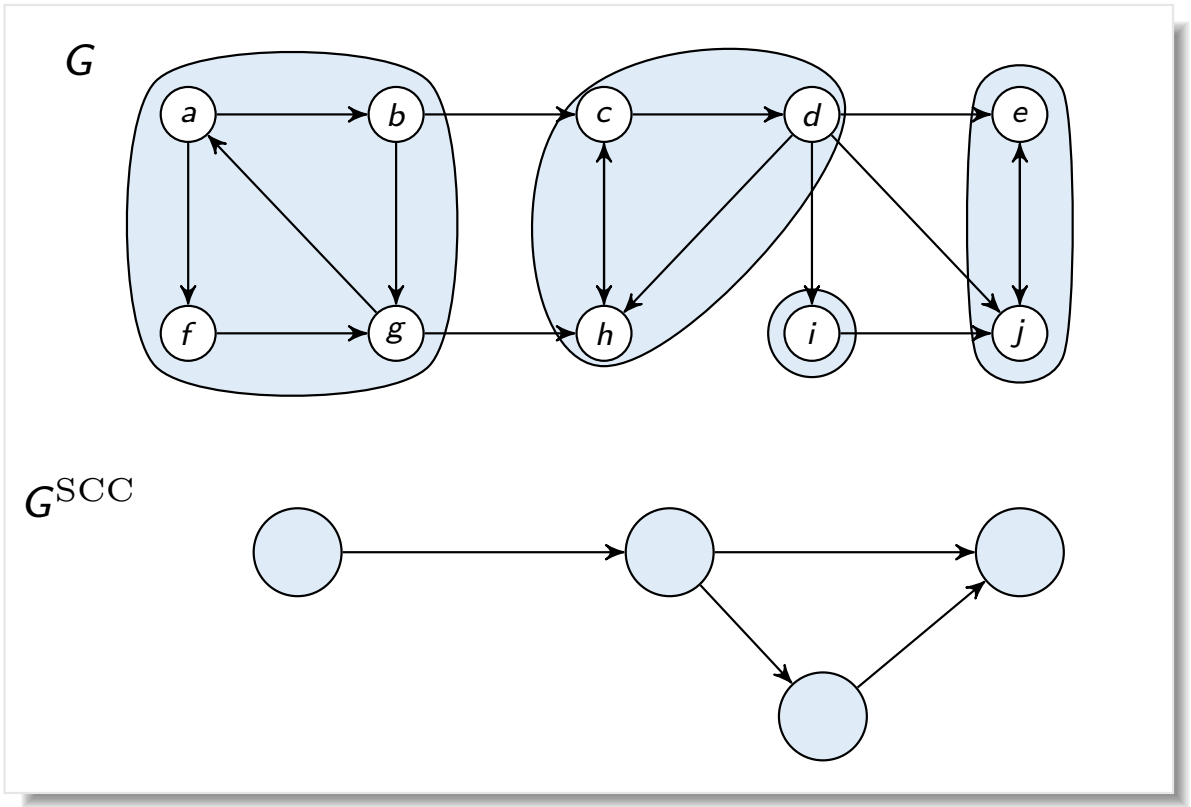


A depiction of all SCCs of the graph

Component Graph

For a digraph $G = (V, E)$, its component graph $G^{\mathrm{SCC}} = (V^{\mathrm{SCC}}, E^{\mathrm{SCC}})$ is defined by:

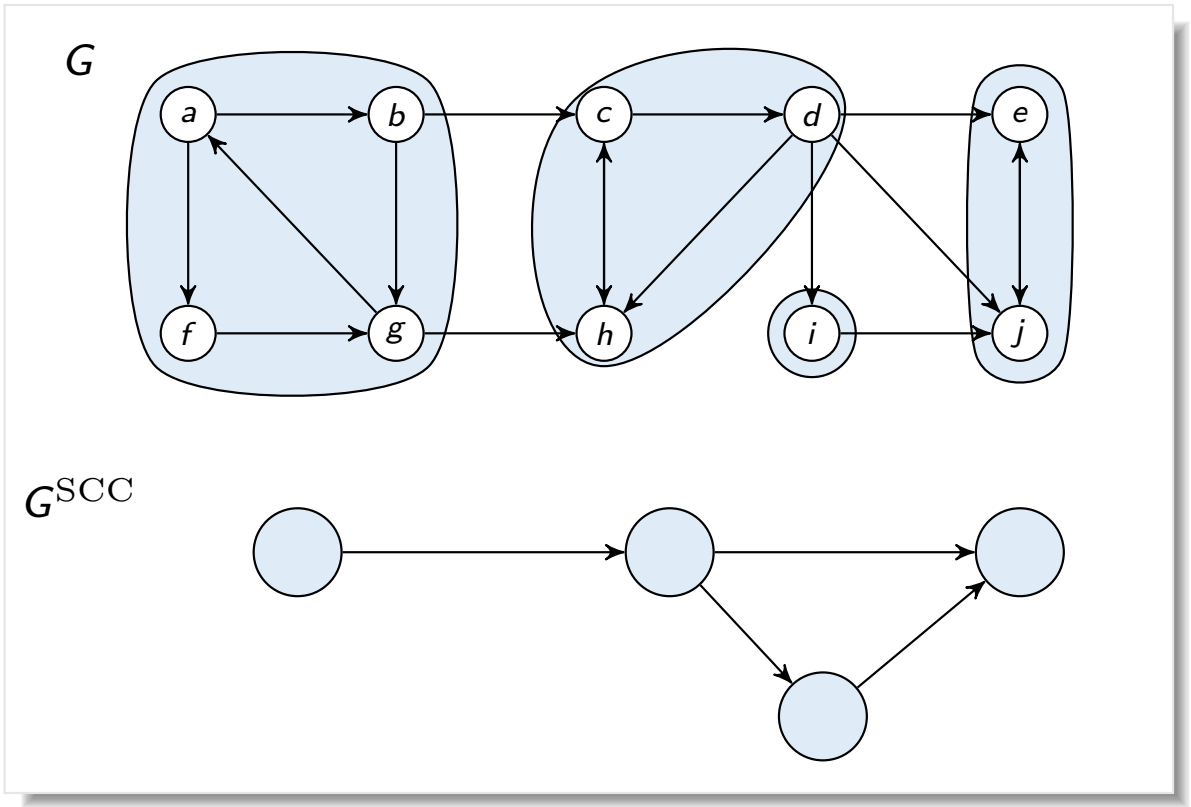
- V^{SCC} has a vertex for each SCC in G;
- E^{SCC} has an edge if there's an edge between the corresponding SCC's in G.



Component Graph

For a digraph $G = (V, E)$, its component graph $G^{\mathrm{SCC}} = (V^{\mathrm{SCC}}, E^{\mathrm{SCC}})$ is defined by:

- V^{SCC} has a vertex for each SCC in G;
- E^{SCC} has an edge if there's an edge between the corresponding SCC's in G.



Lemma: G^{SCC} is a DAG.

Magic Algorithm

$\mathrm{SCC}(G)$:

1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.

Magic Algorithm

$\mathrm{SCC}(G)$:

1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.

Graph G^T is the transpose of G:

- $G^T = (V, E), E^T = \{(u, v) : (v, u) \in E\}$.
- G^T is G with all edges reversed.

Magic Algorithm

$\mathrm{SCC}(G)$:

1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.

Graph G^T is the transpose of G:

- $G^T = (V, E), E^T = \{(u, v) : (v, u) \in E\}$.
- G^T is G with all edges reversed.

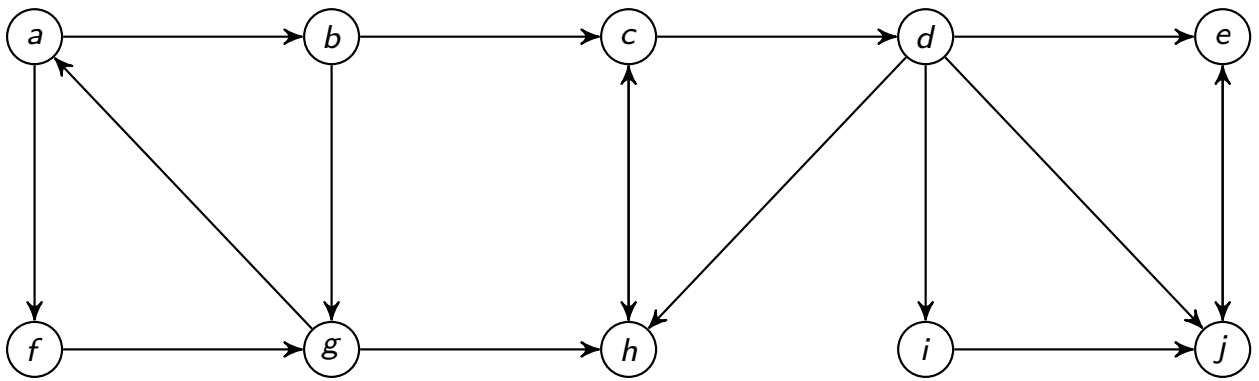
Observations:

- Can create G^T in $\Theta(V + E)$ time if using adjacency lists.
- G and G^T has the same SCCs.

Magic Algorithm

$\mathrm{SCC}(G)$:

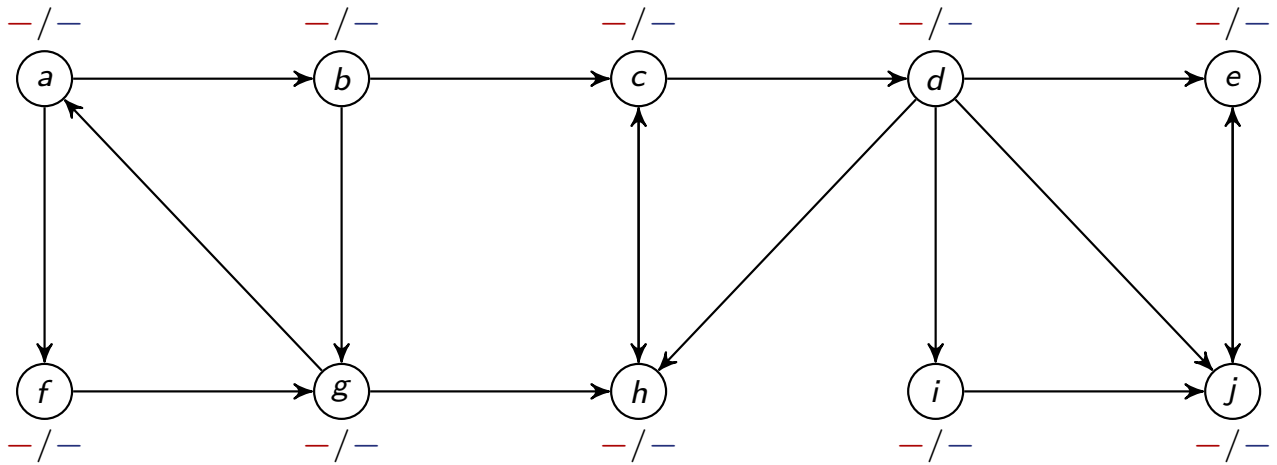
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

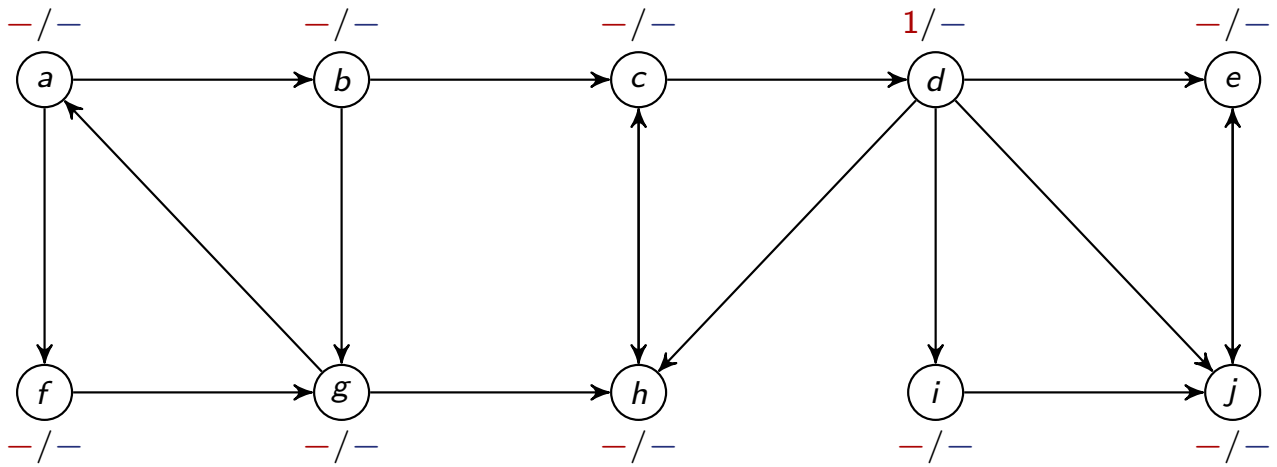
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

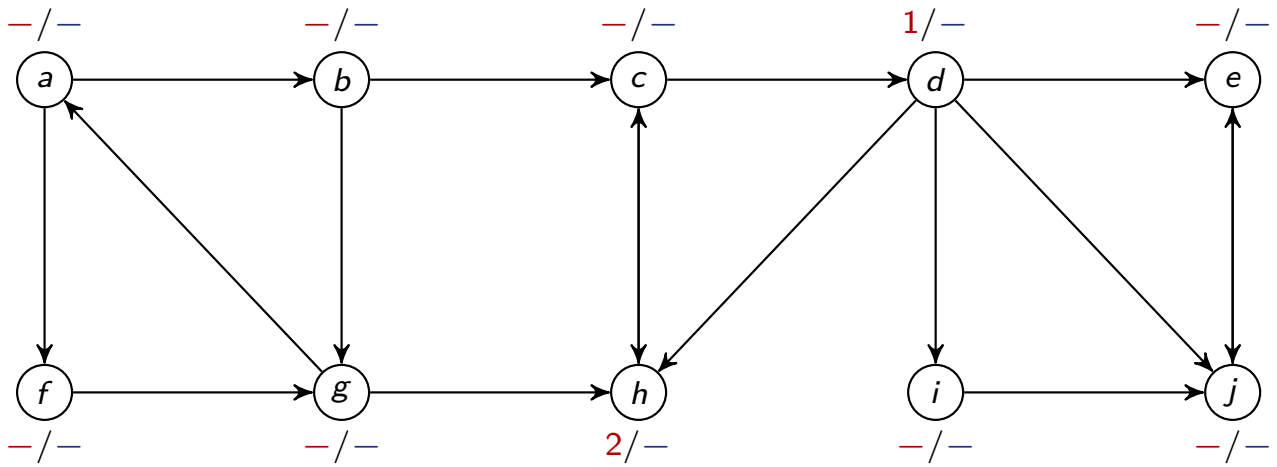
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

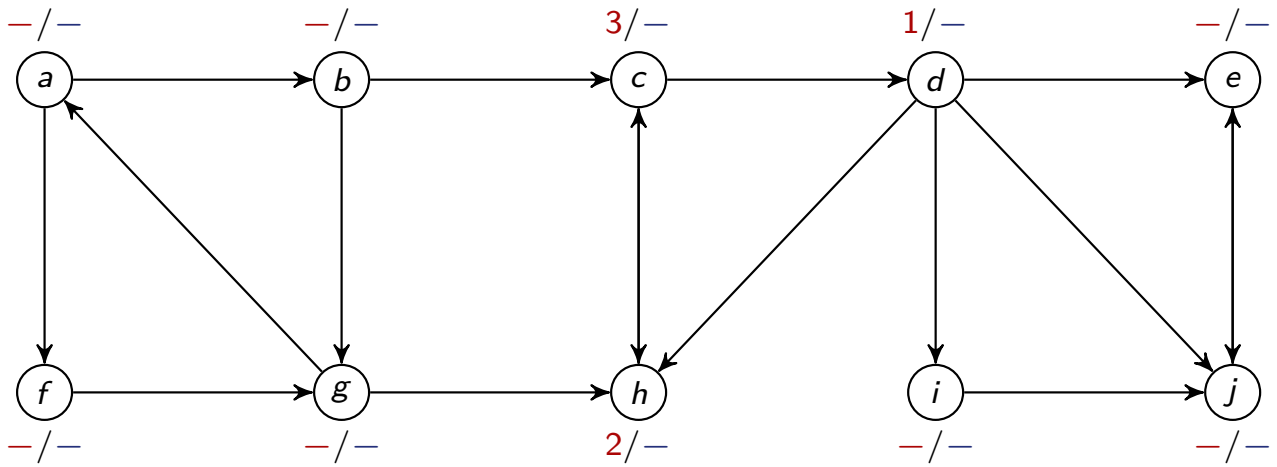
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

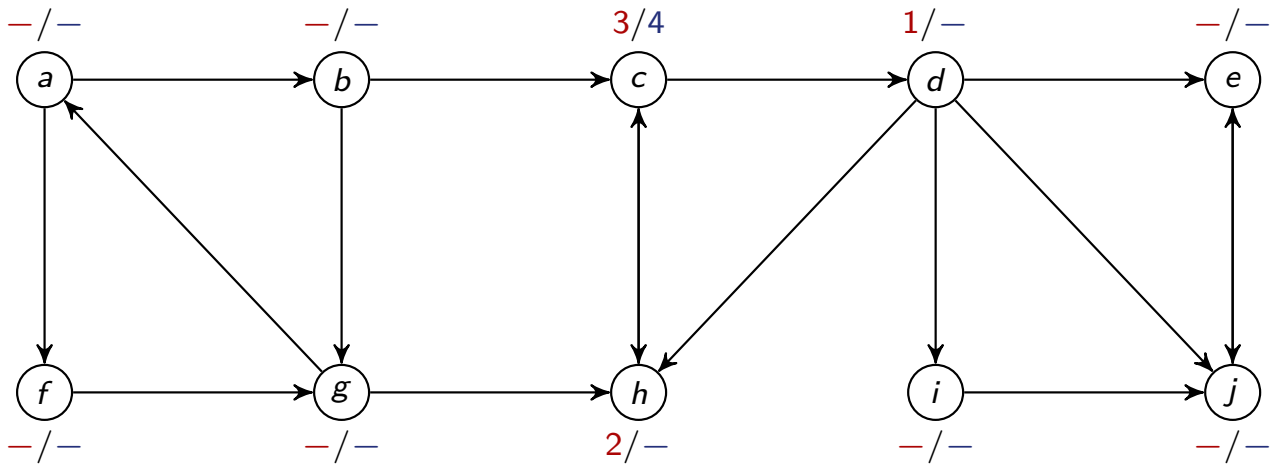
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

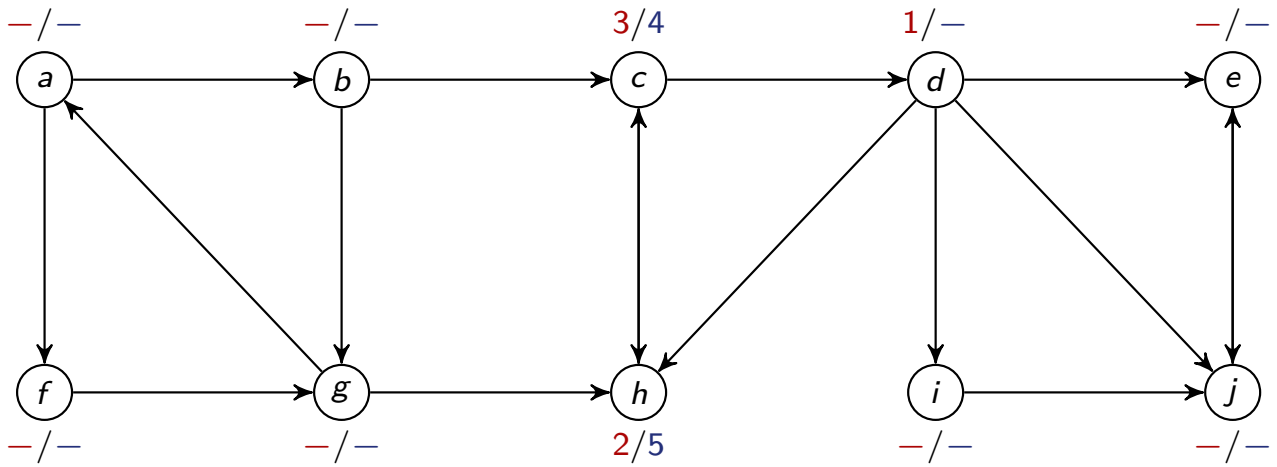
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

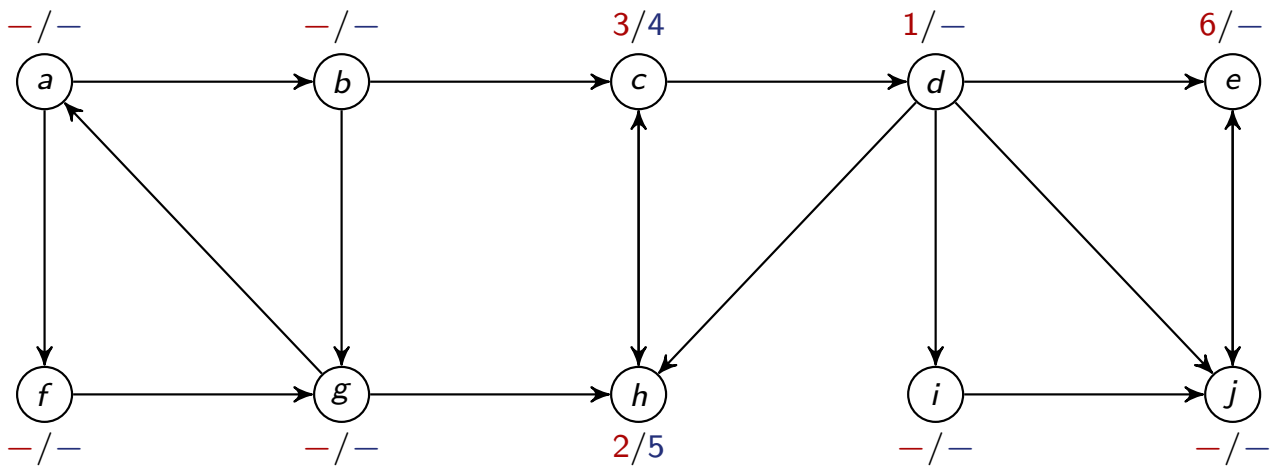
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

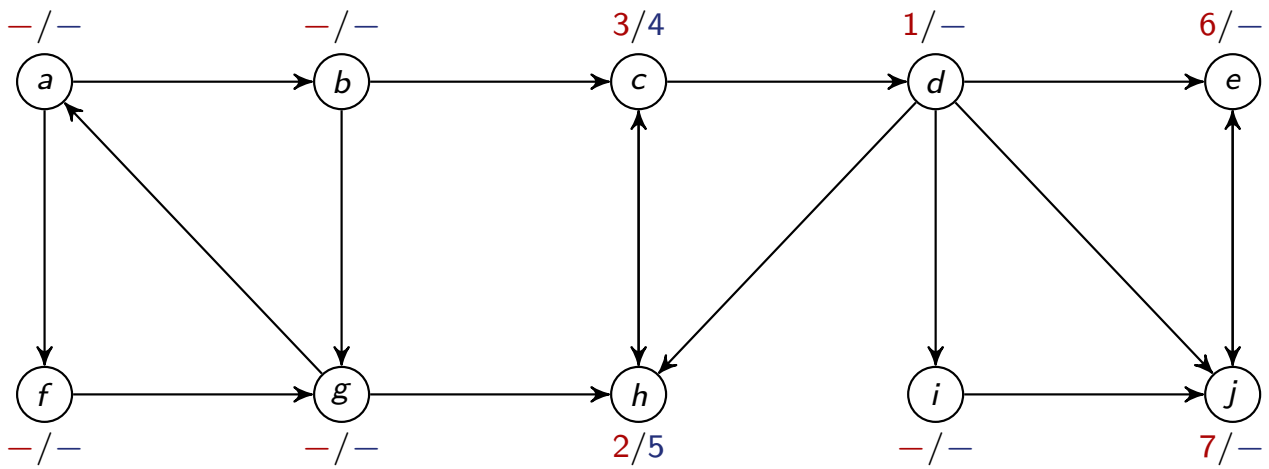
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

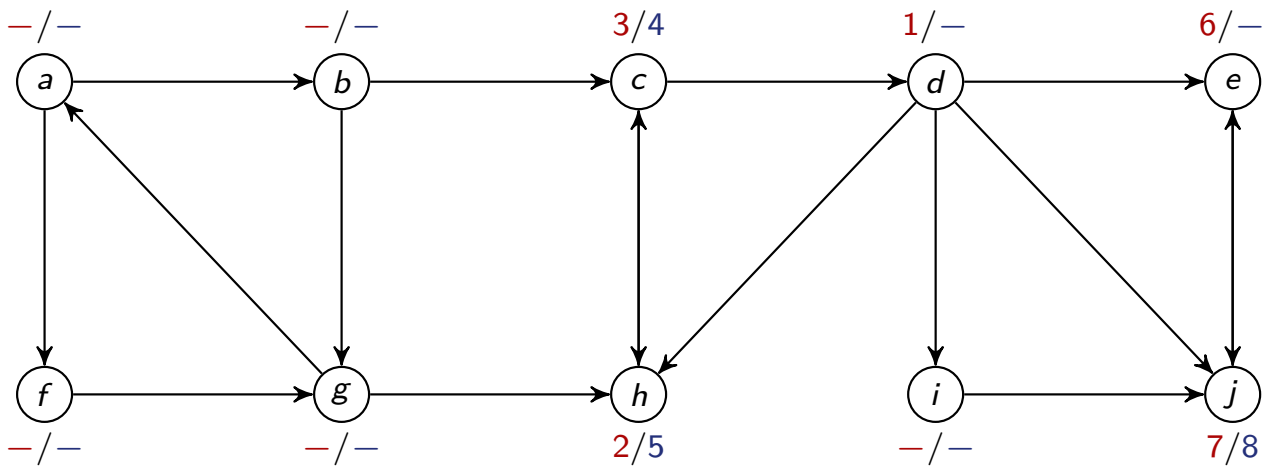
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

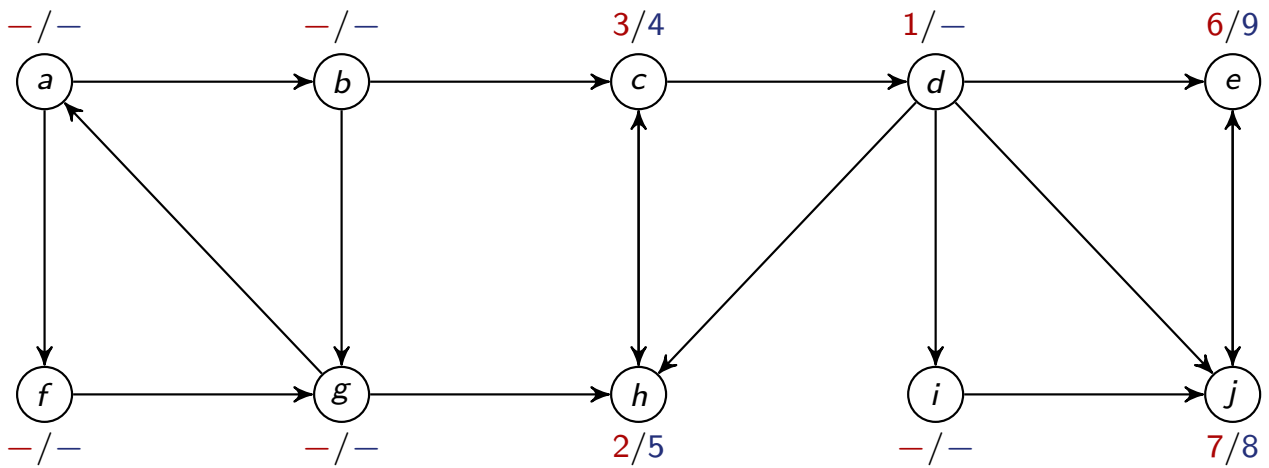
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

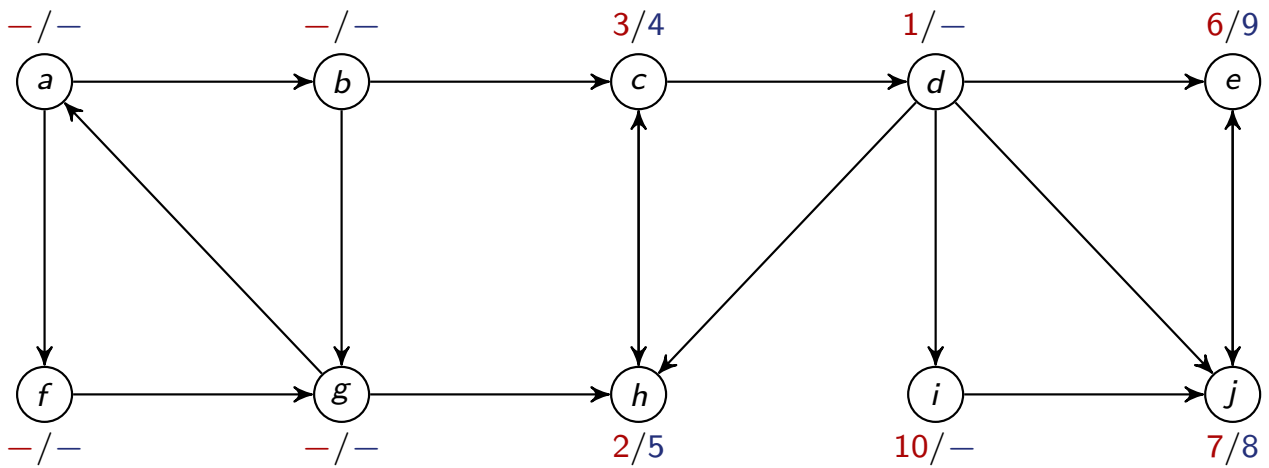
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

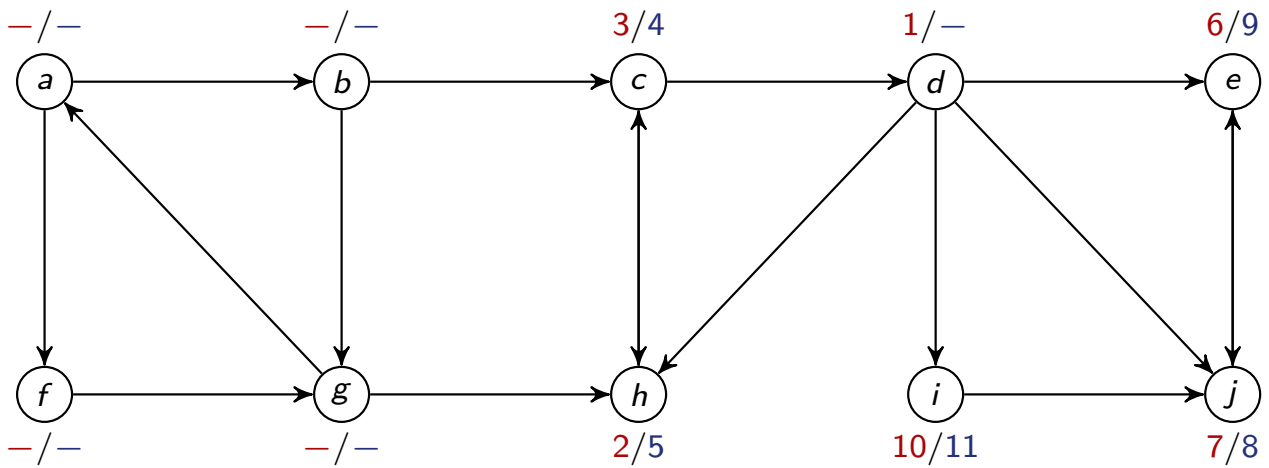
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

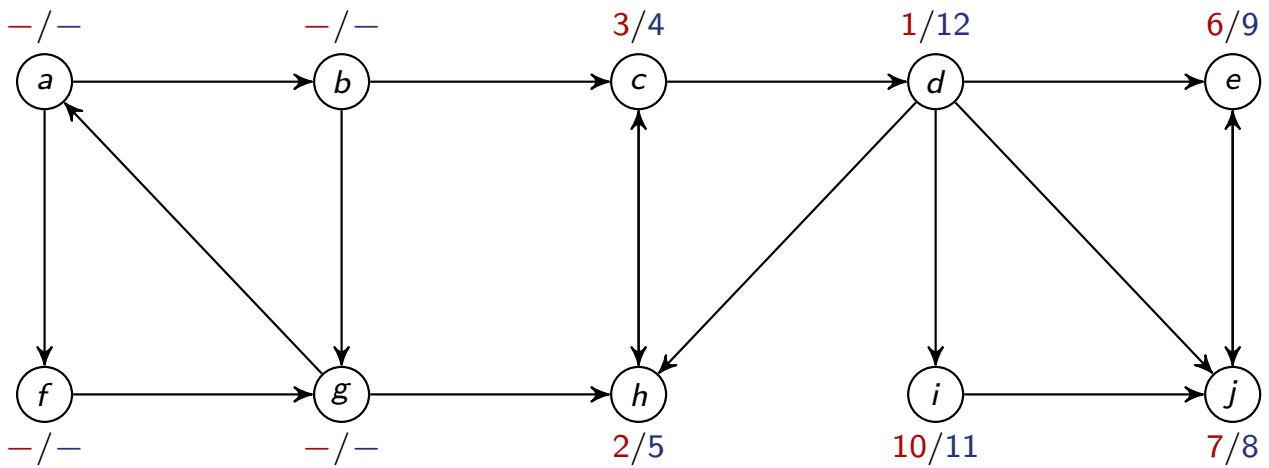
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

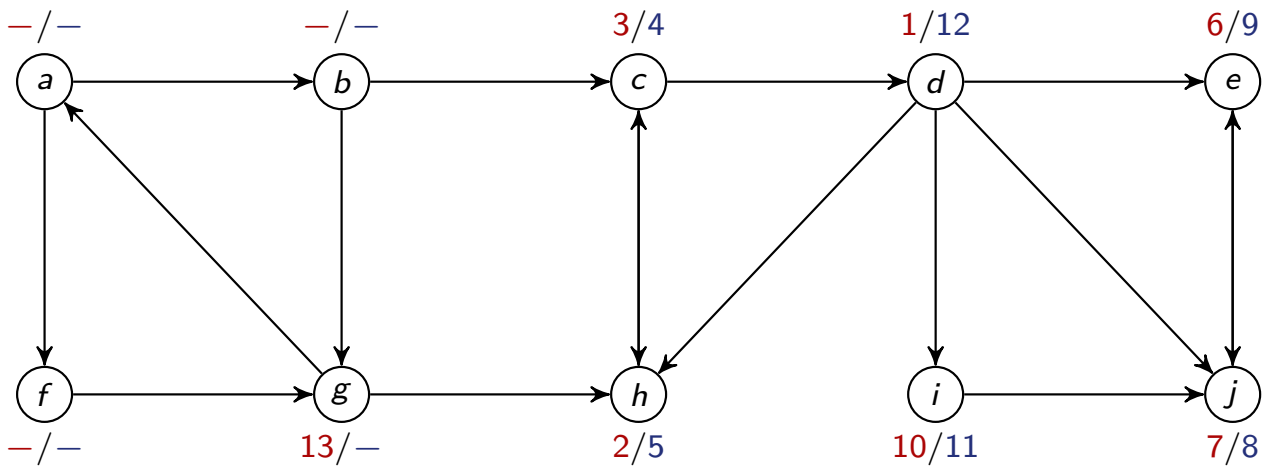
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

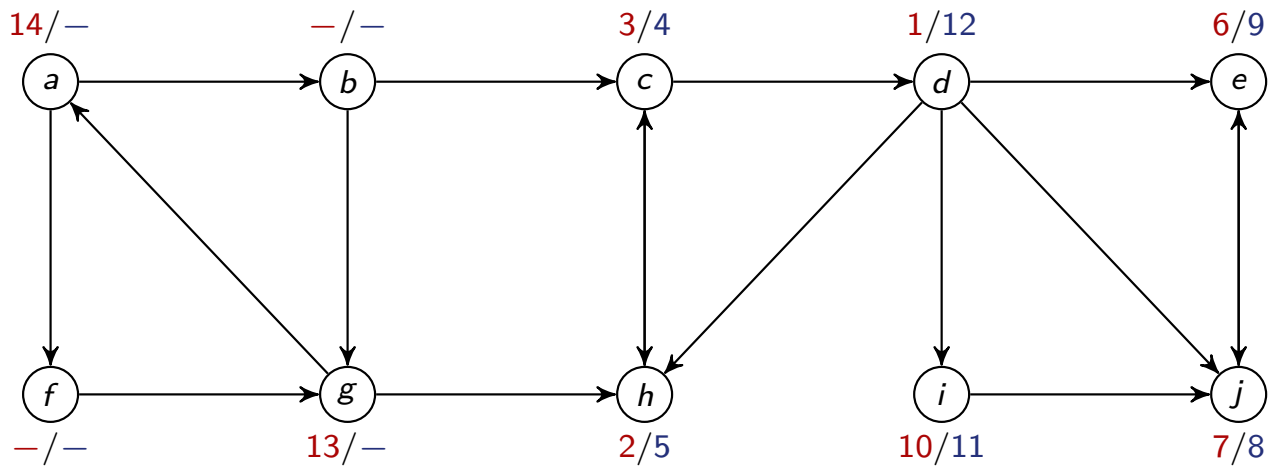
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

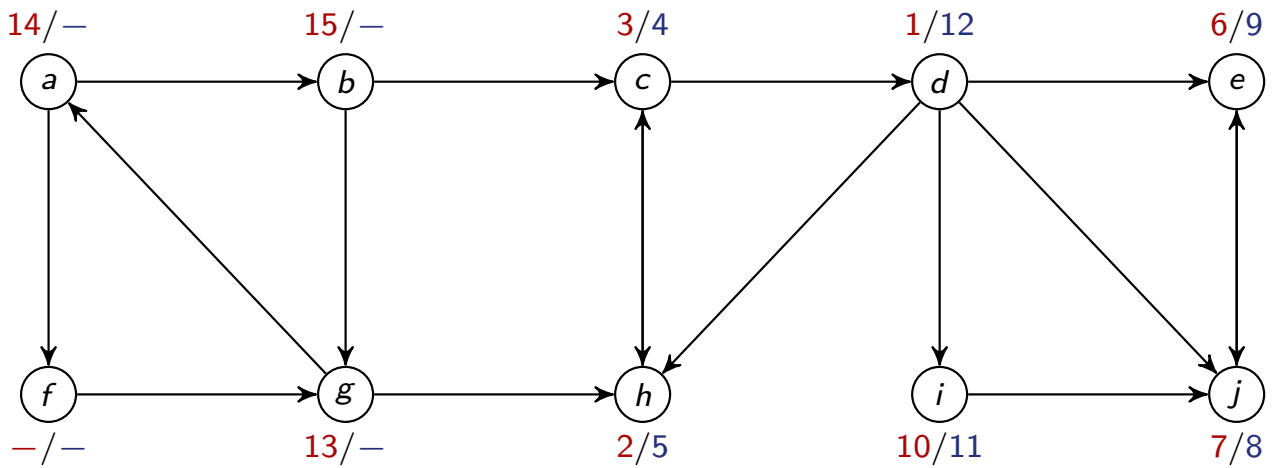
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

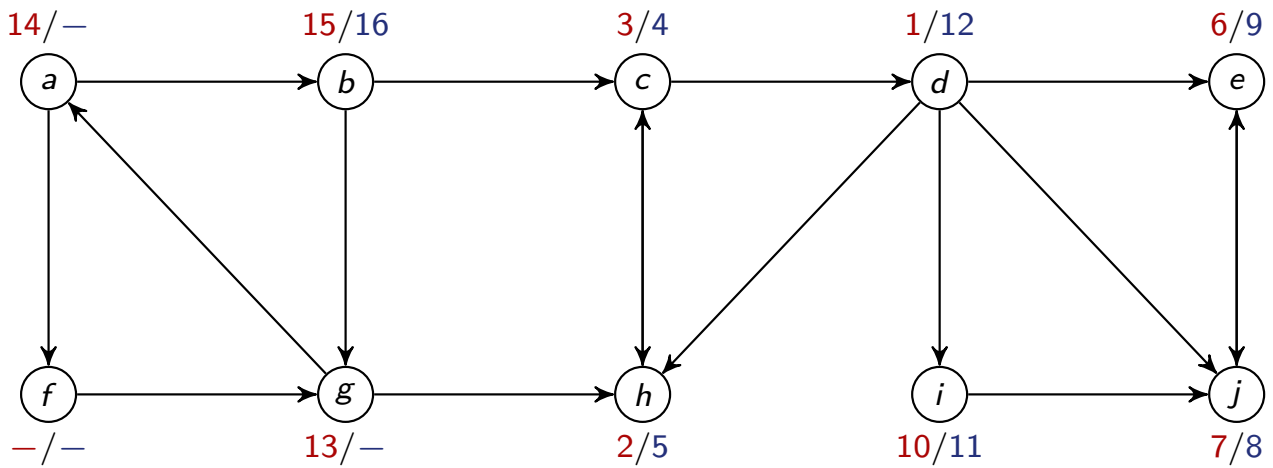
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

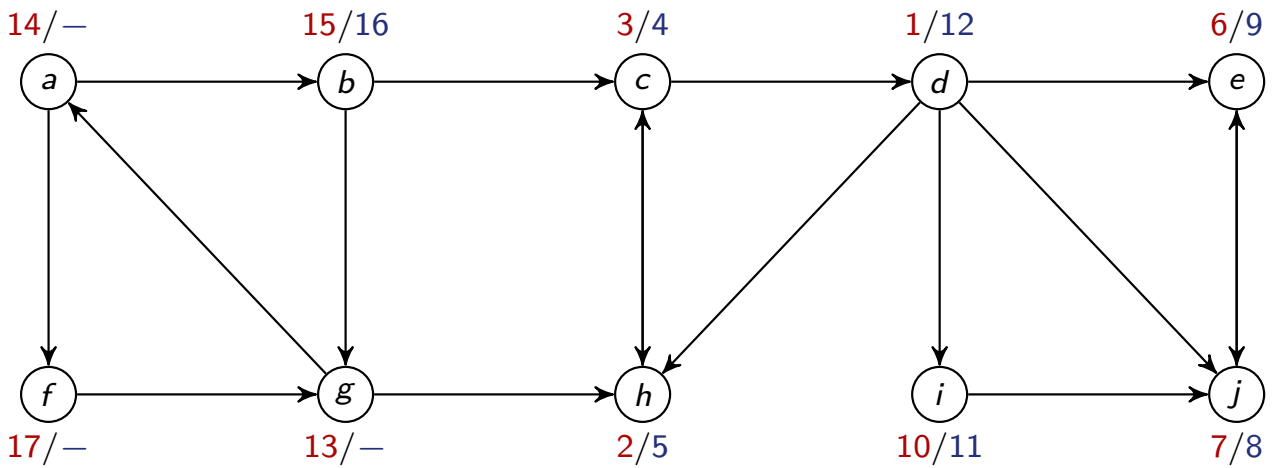
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

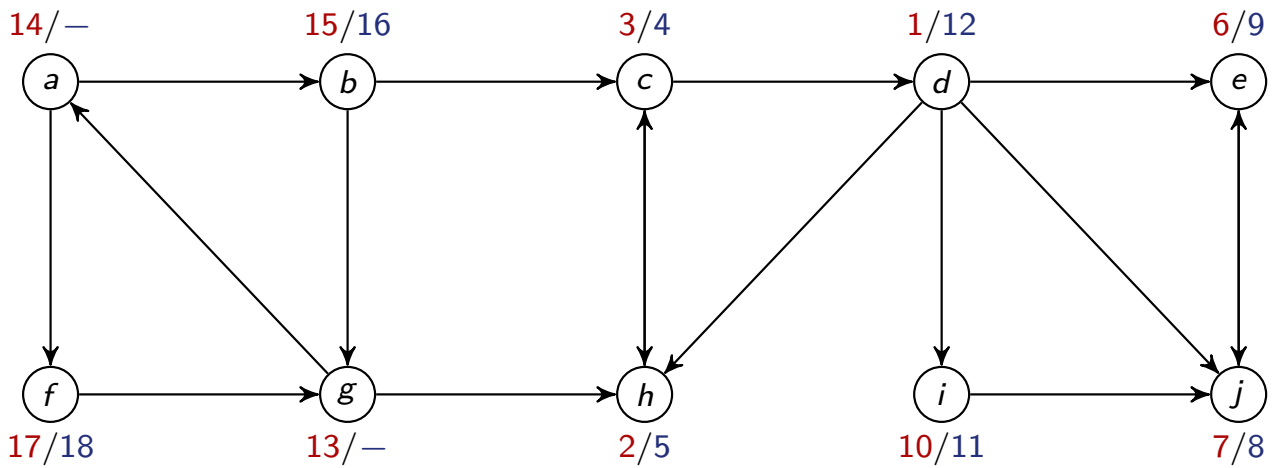
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

SCC(G):

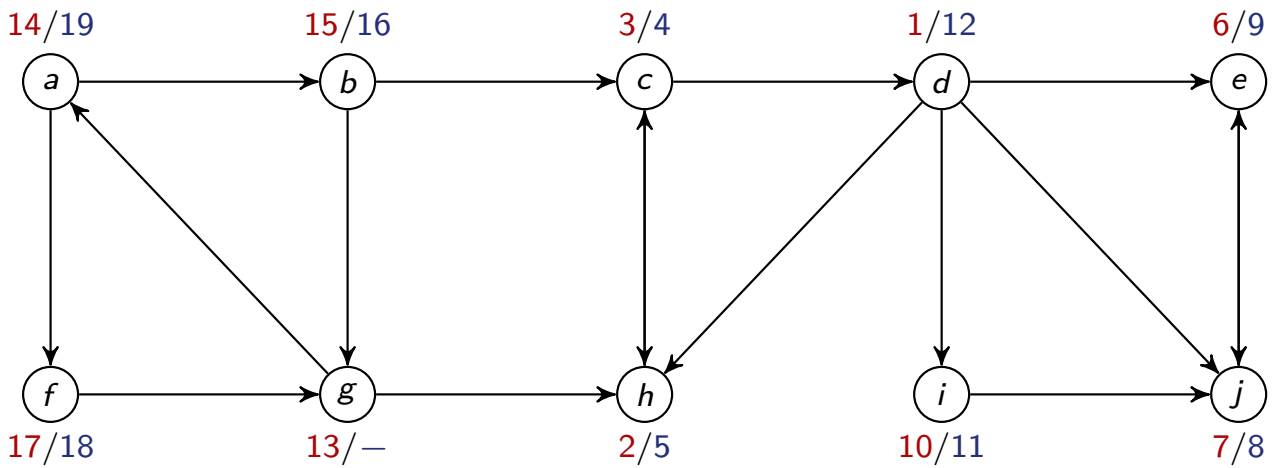
1. Call DFS(G) to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call DFS(G^T) but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

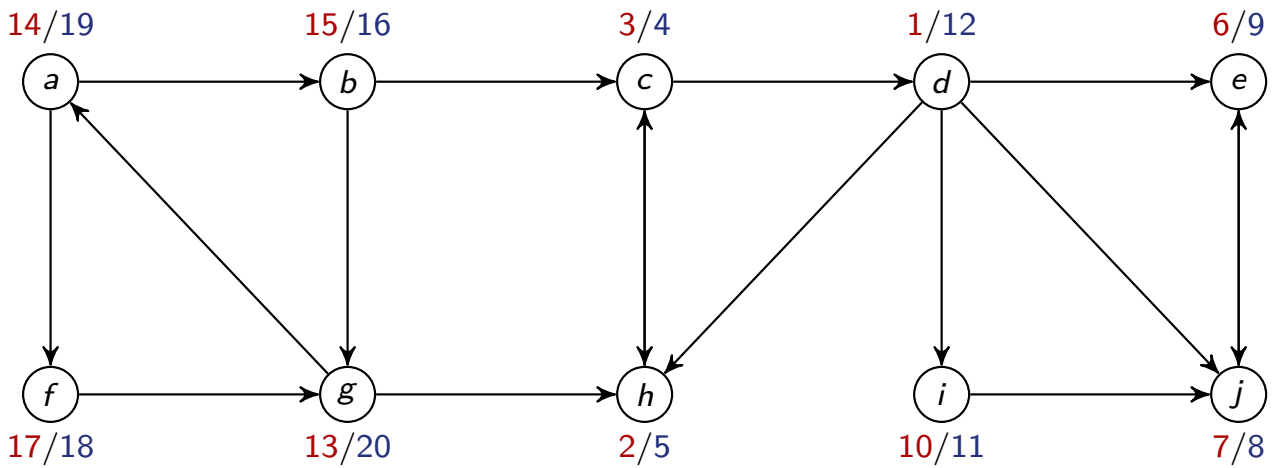
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

SCC(G):

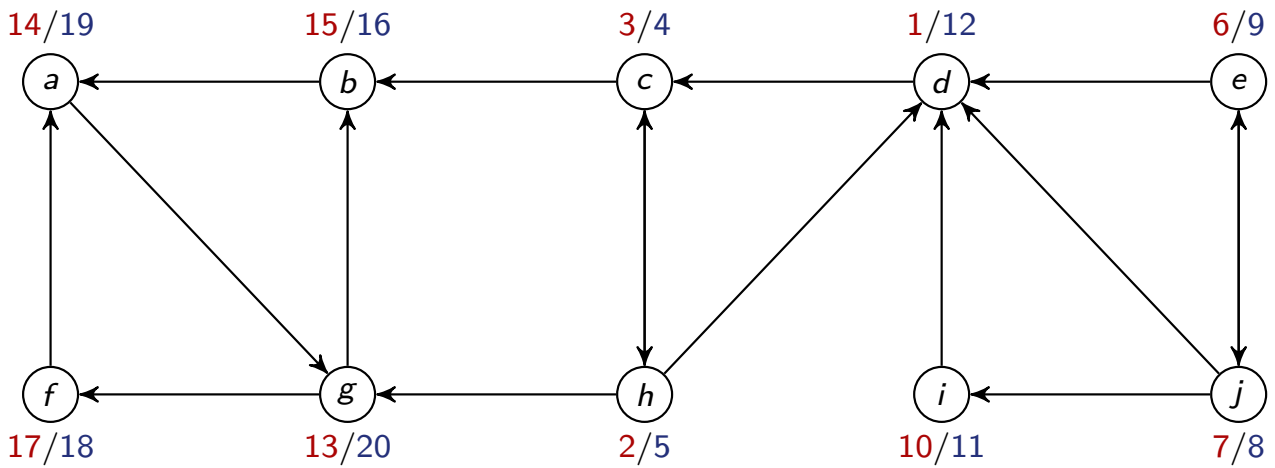
1. Call DFS(G) to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call DFS(G^T) but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

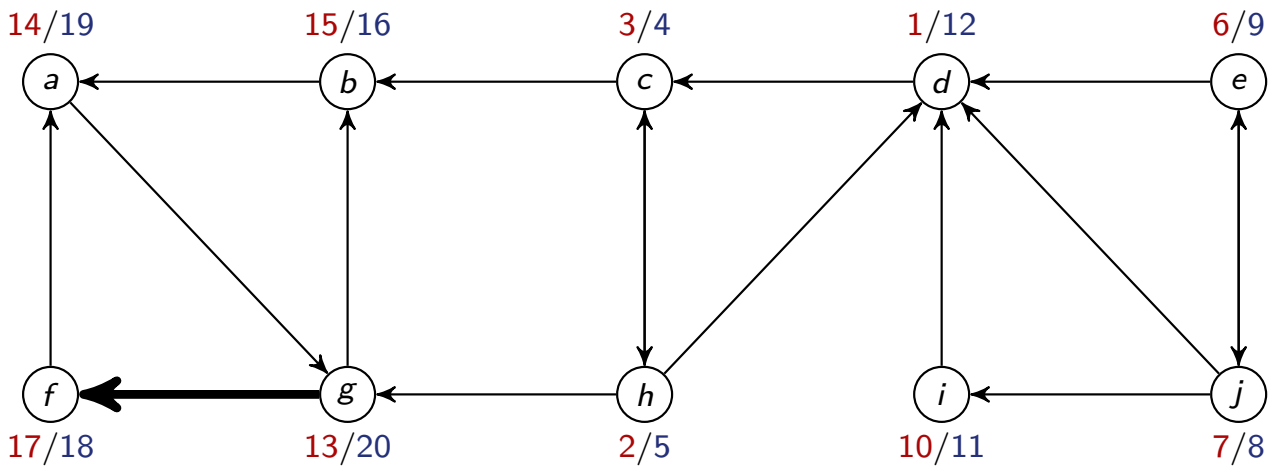
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

SCC(G):

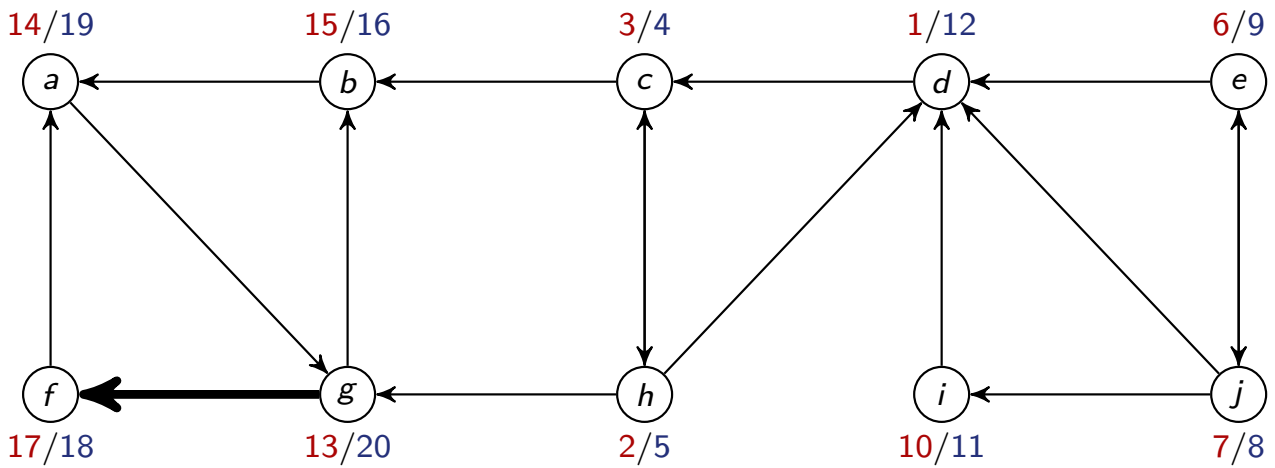
1. Call DFS(G) to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call DFS(G^T) but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

SCC(G):

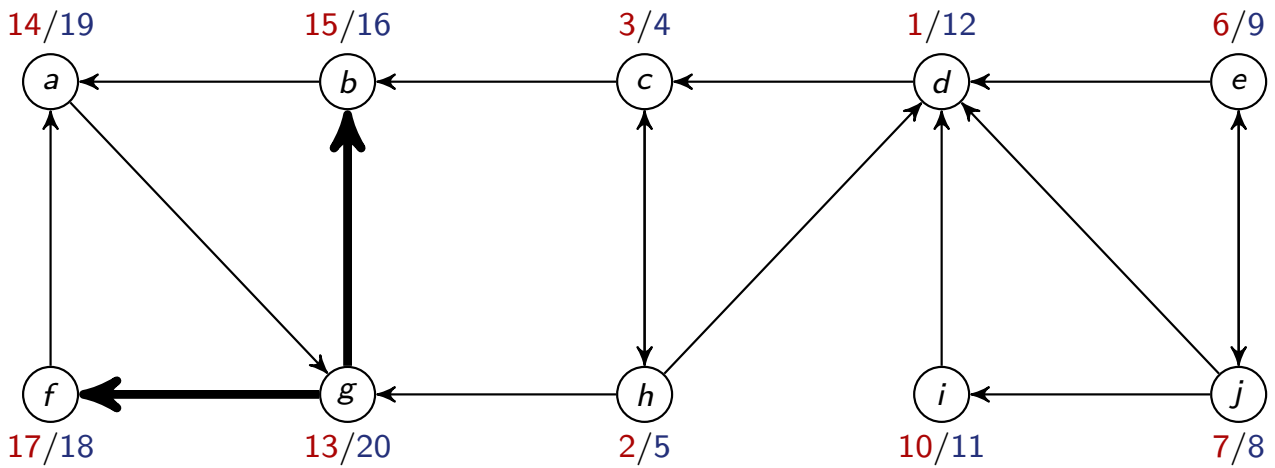
1. Call DFS(G) to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call DFS(G^T) but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

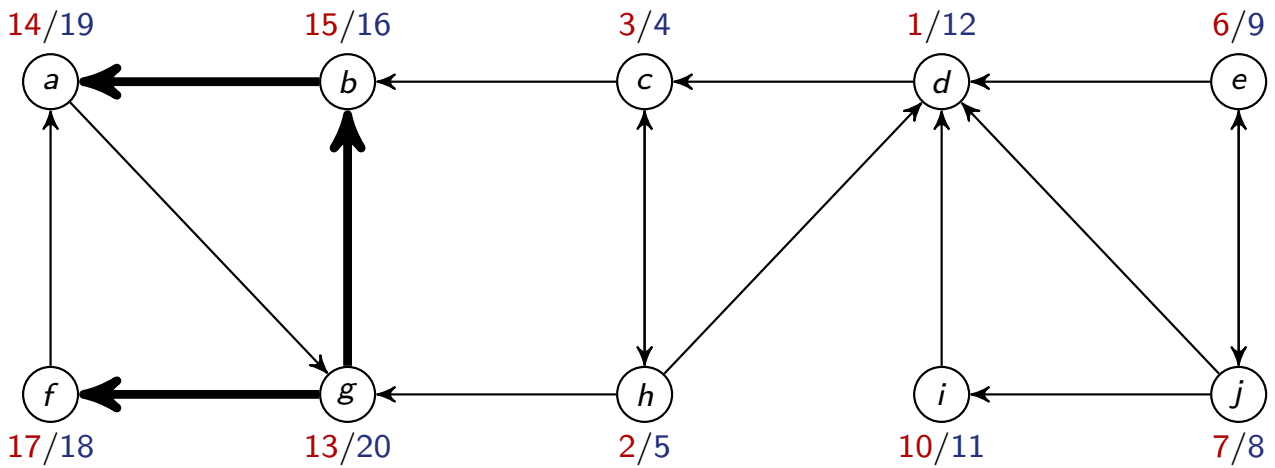
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

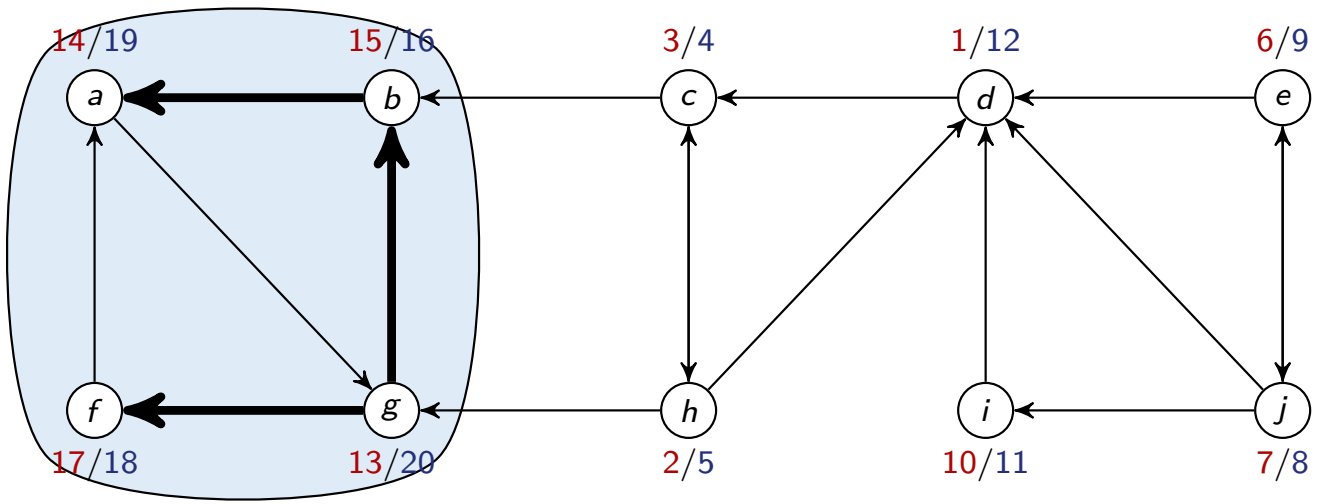
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

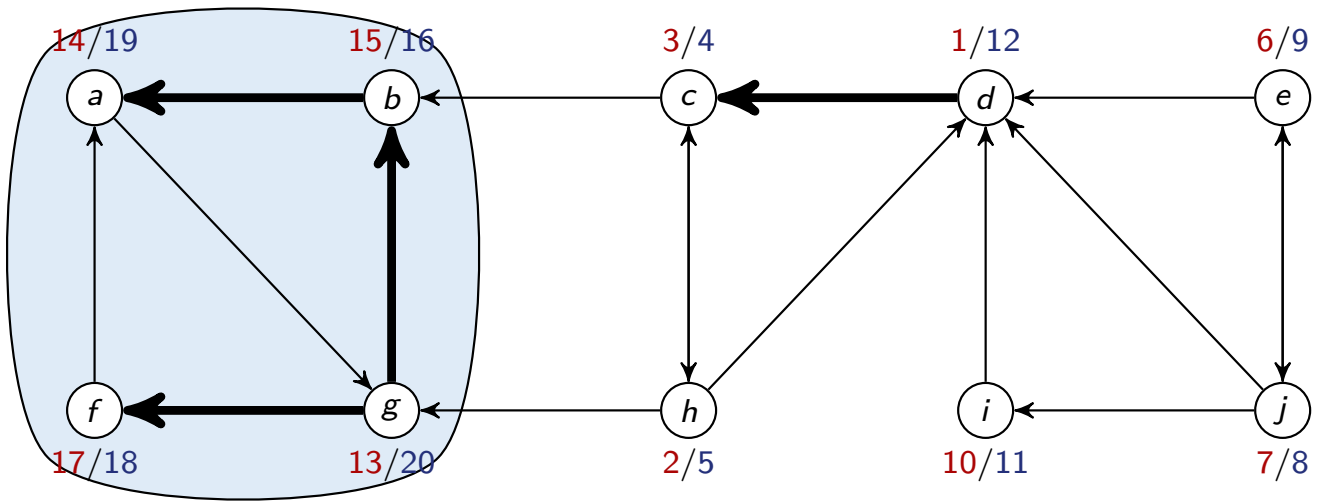
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

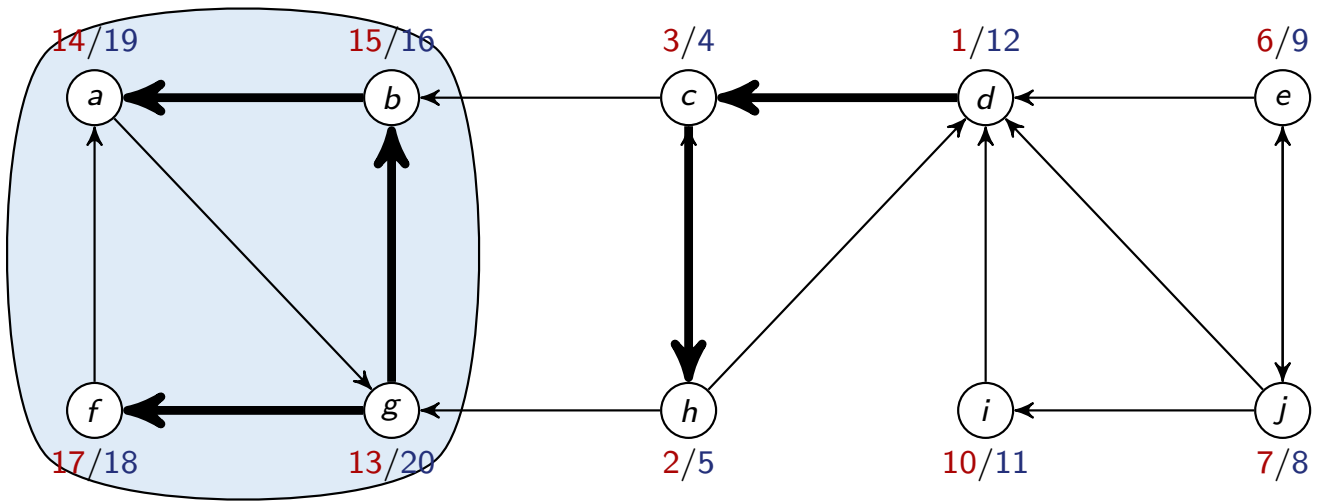
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

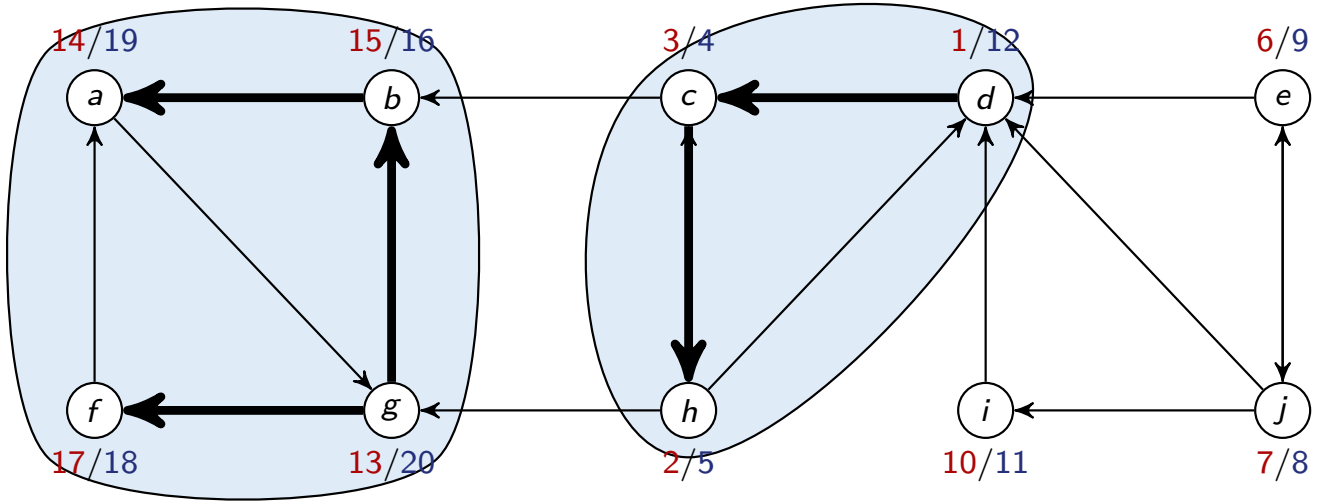
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

SCC(G):

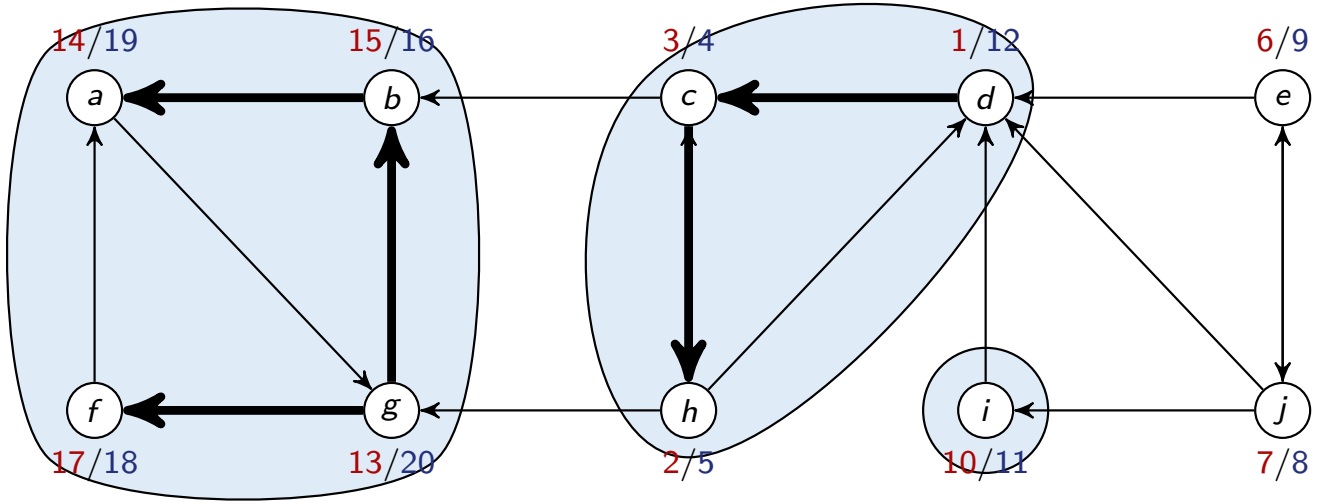
1. Call DFS(G) to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call DFS(G^T) but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

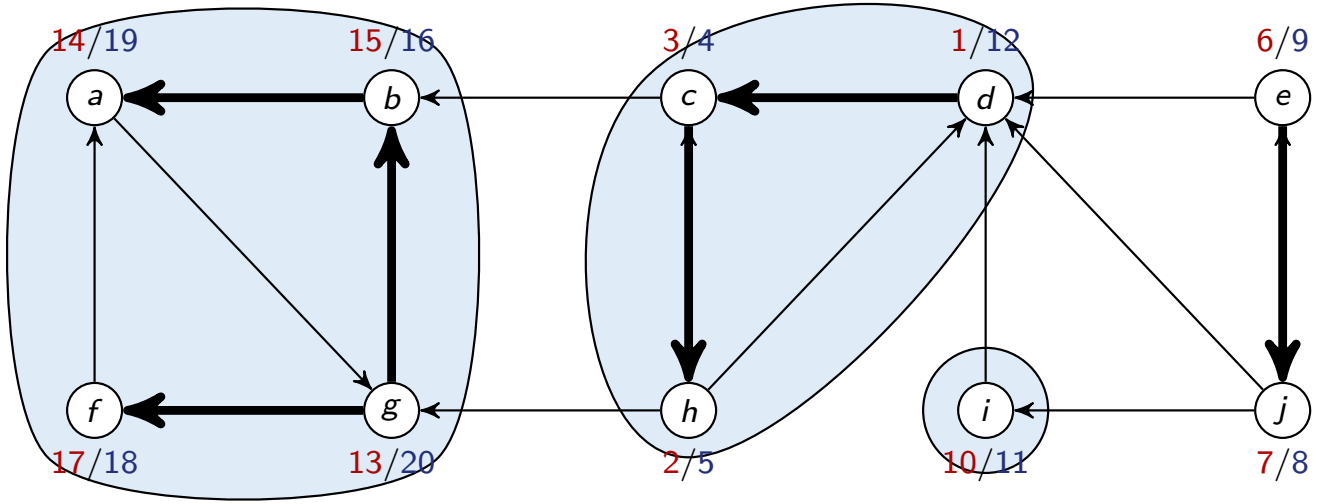
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

$\mathrm{SCC}(G)$:

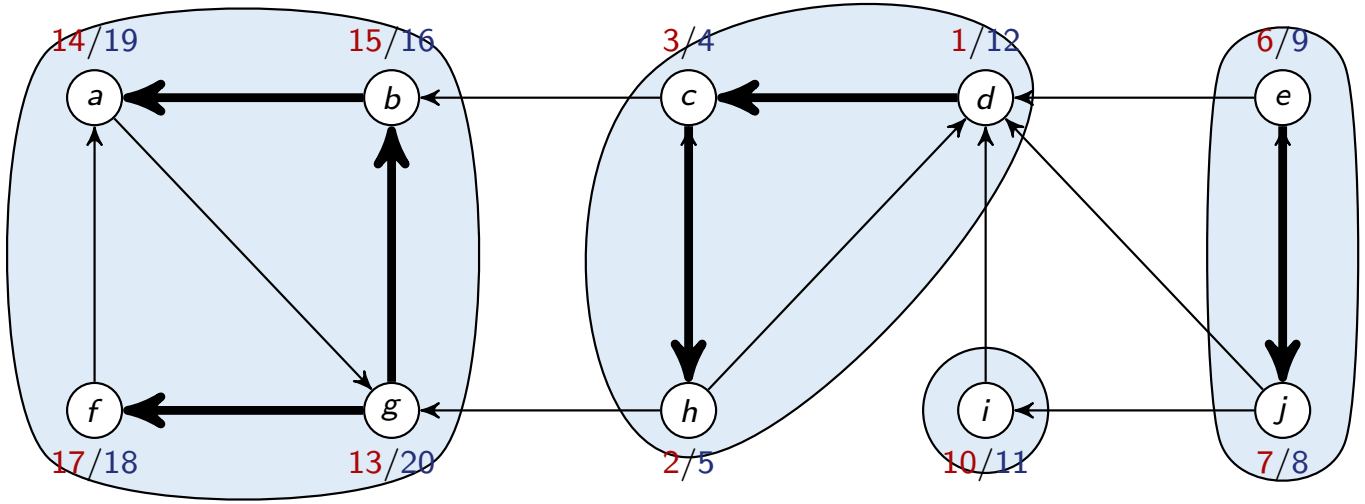
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Magic Algorithm

SCC(G):

1. Call DFS(G) to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call DFS(G^T) but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.



Analysis

$\mathrm{SCC}(G)$:

1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.

Runtime analysis:

Analysis

$\mathrm{SCC}(G)$:

1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.

Runtime analysis: Each step takes $\Theta(V + E)$ so total running time is $\Theta(V + E)$

Analysis

$\mathrm{SCC}(G)$:

1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.

Runtime analysis: Each step takes $\Theta(V + E)$ so total running time is $\Theta(V + E)$

Why does it work?

Analysis

$\mathrm{SCC}(G)$:

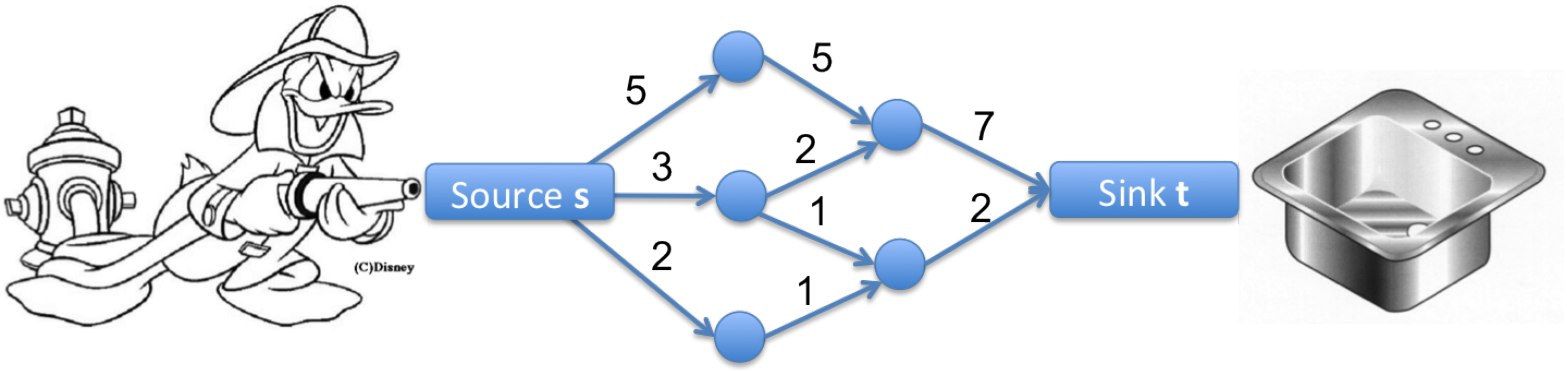
1. Call $\mathrm{DFS}(G)$ to compute finishing times $u.f$ for all u.
2. Compute G^T
3. Call $\mathrm{DFS}(G^T)$ but in the main loop, consider vertices in order of decreasing $u.f$ (as computed in first DFS).
4. Output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC.

Runtime analysis: Each step takes $\Theta(V + E)$ so total running time is $\Theta(V + E)$

Why does it work? Intuition:

- The first DFS orders SCC's in topological order (recall G^{SCC} is acyclic)
- Second DFS then outputs the vertices in each SCC

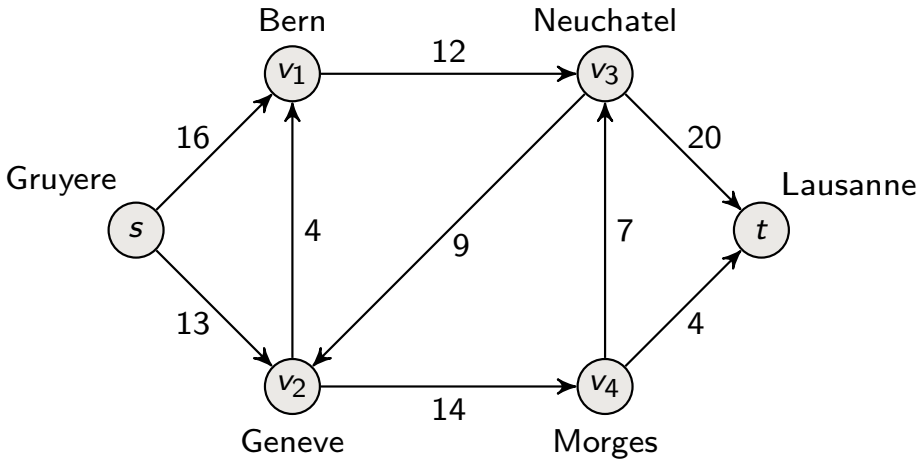
Formal proof in book



FLOW NETWORKS

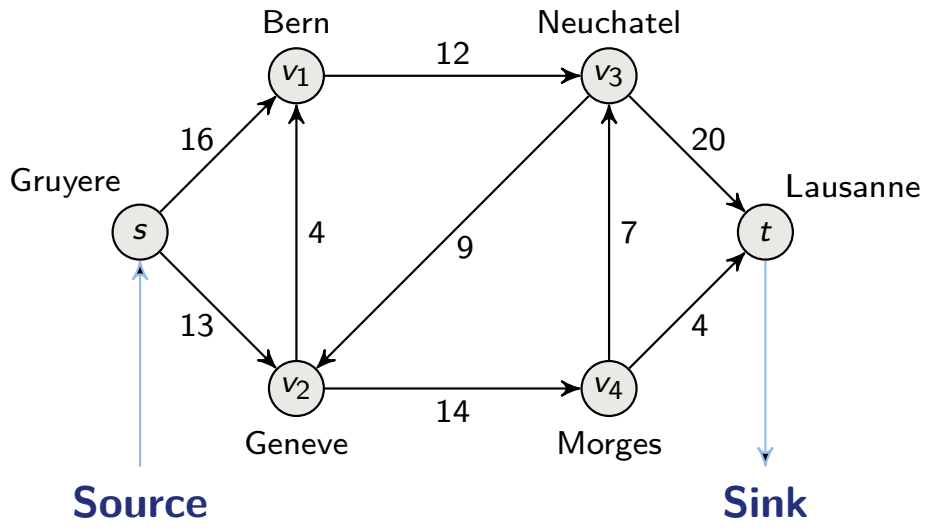
Flow Network

Transfer as much cheese as possible from Gruyere to Lausanne



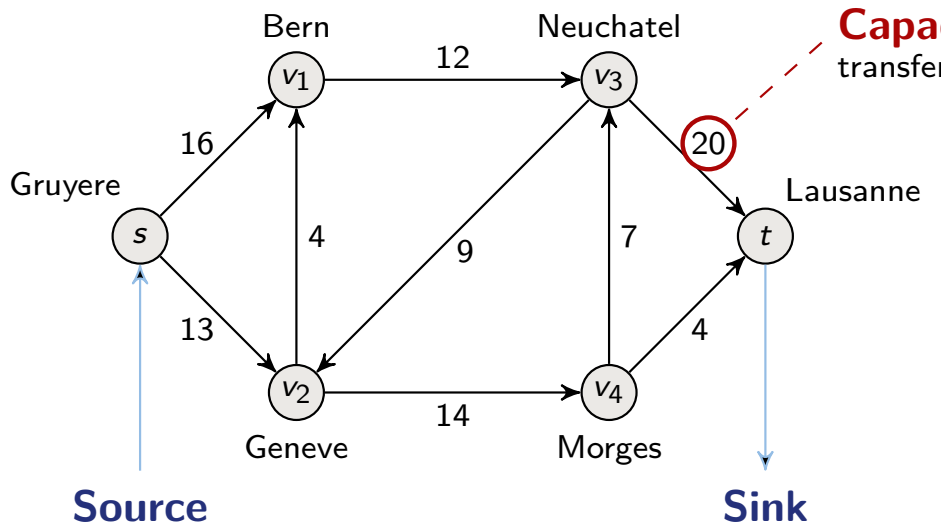
Flow Network

Transfer as much cheese as possible from Gruyere to Lausanne



Flow Network

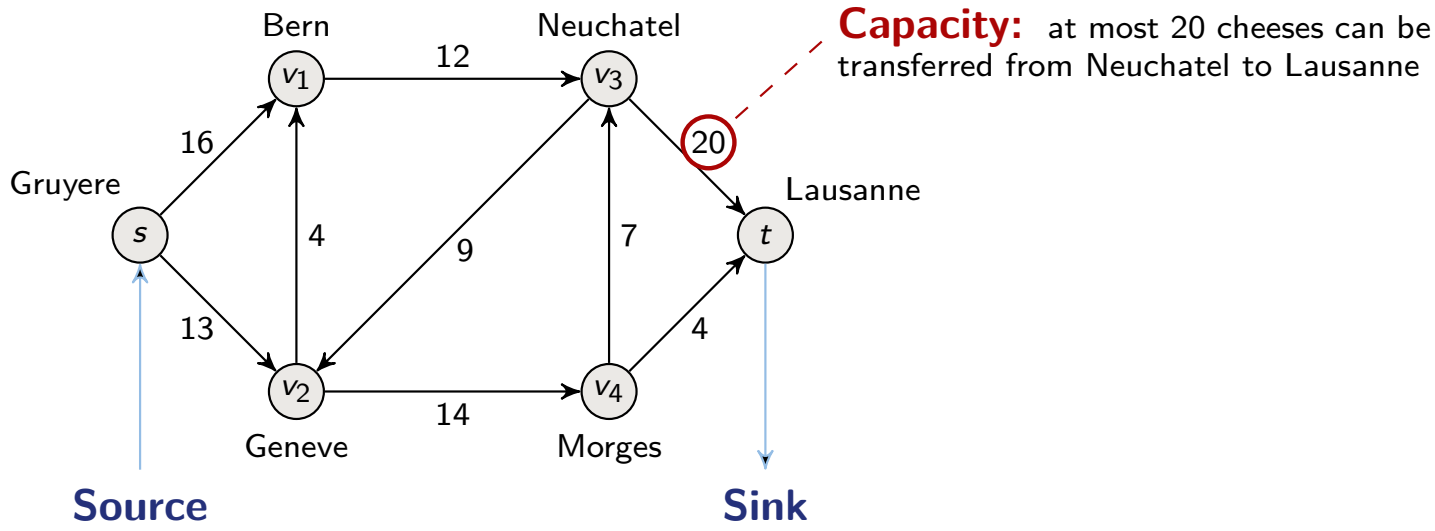
Transfer as much cheese as possible from Gruyere to Lausanne



Capacity: at most 20 cheeses can be transferred from Neuchatel to Lausanne

Flow Network

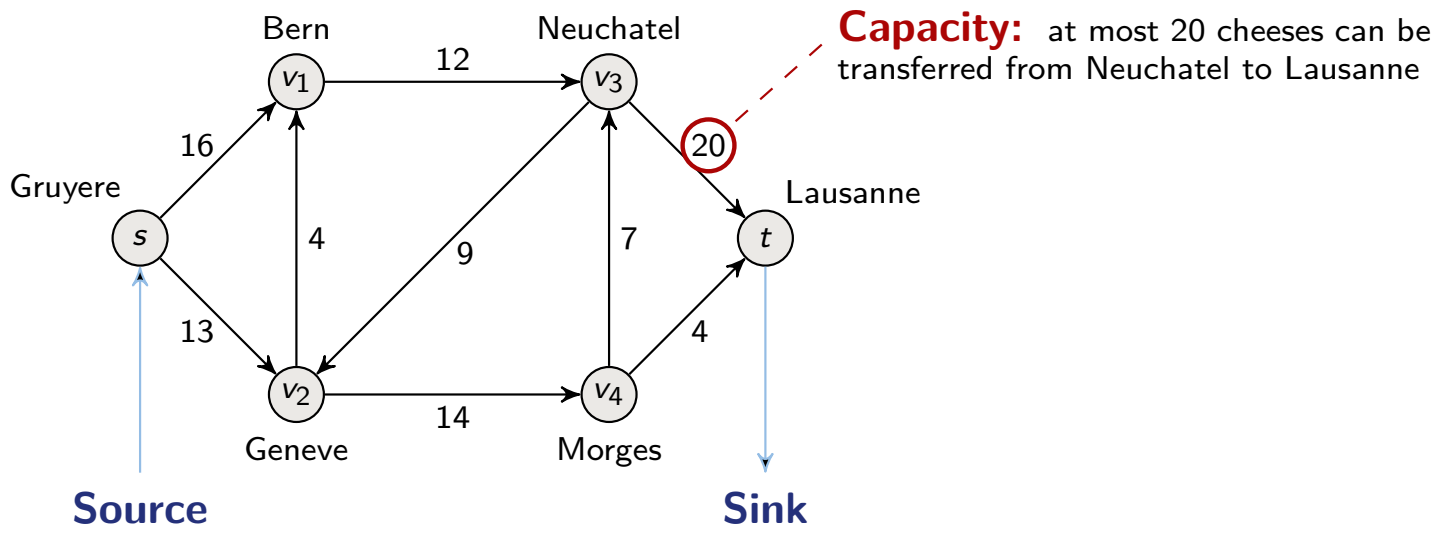
Transfer as much cheese as possible from Gruyere to Lausanne



- a graph to model flow through edges (pipes)

Flow Network

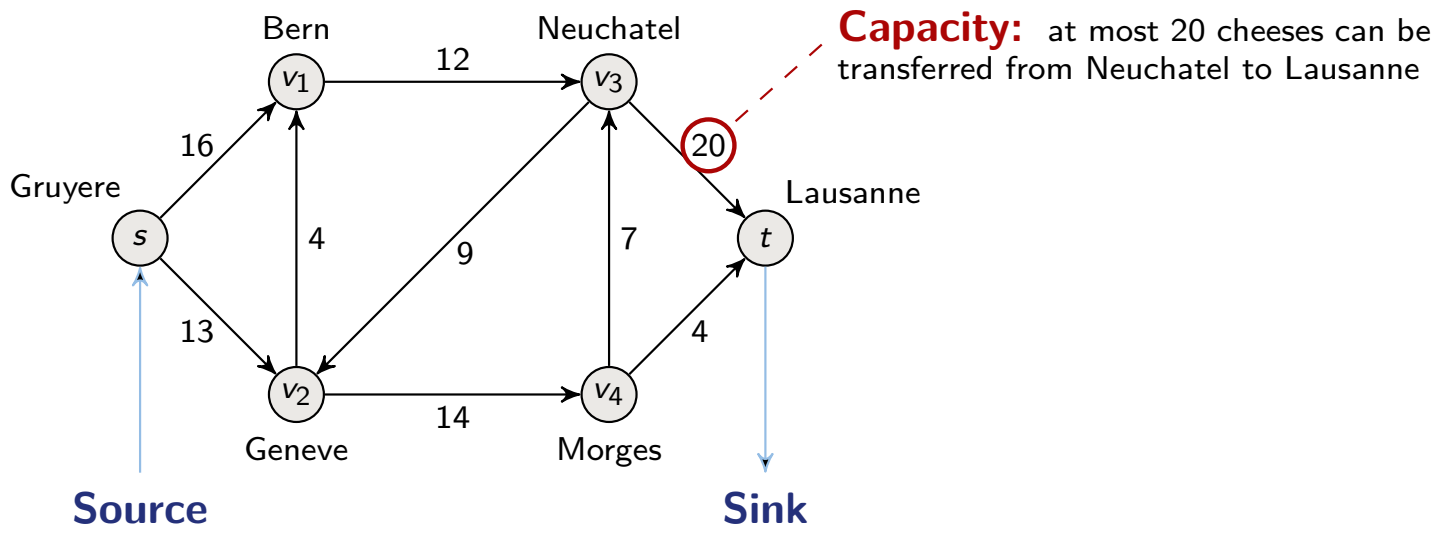
Transfer as much cheese as possible from Gruyere to Lausanne



- a graph to model flow through edges (pipes)
- each edge has a capacity an upper bound on the flow rate (pipes have different sizes)

Flow Network

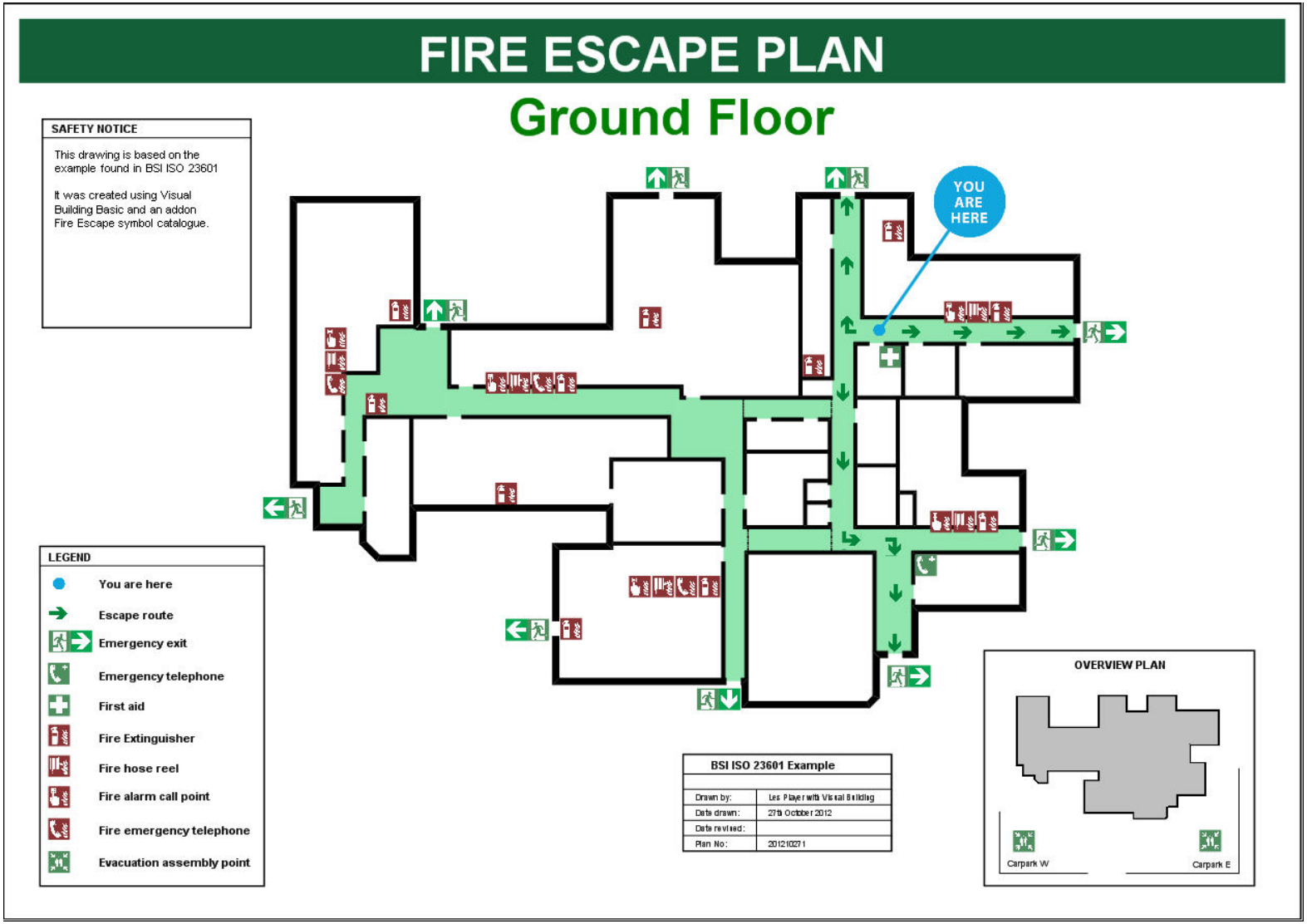
Transfer as much cheese as possible from Gruyere to Lausanne



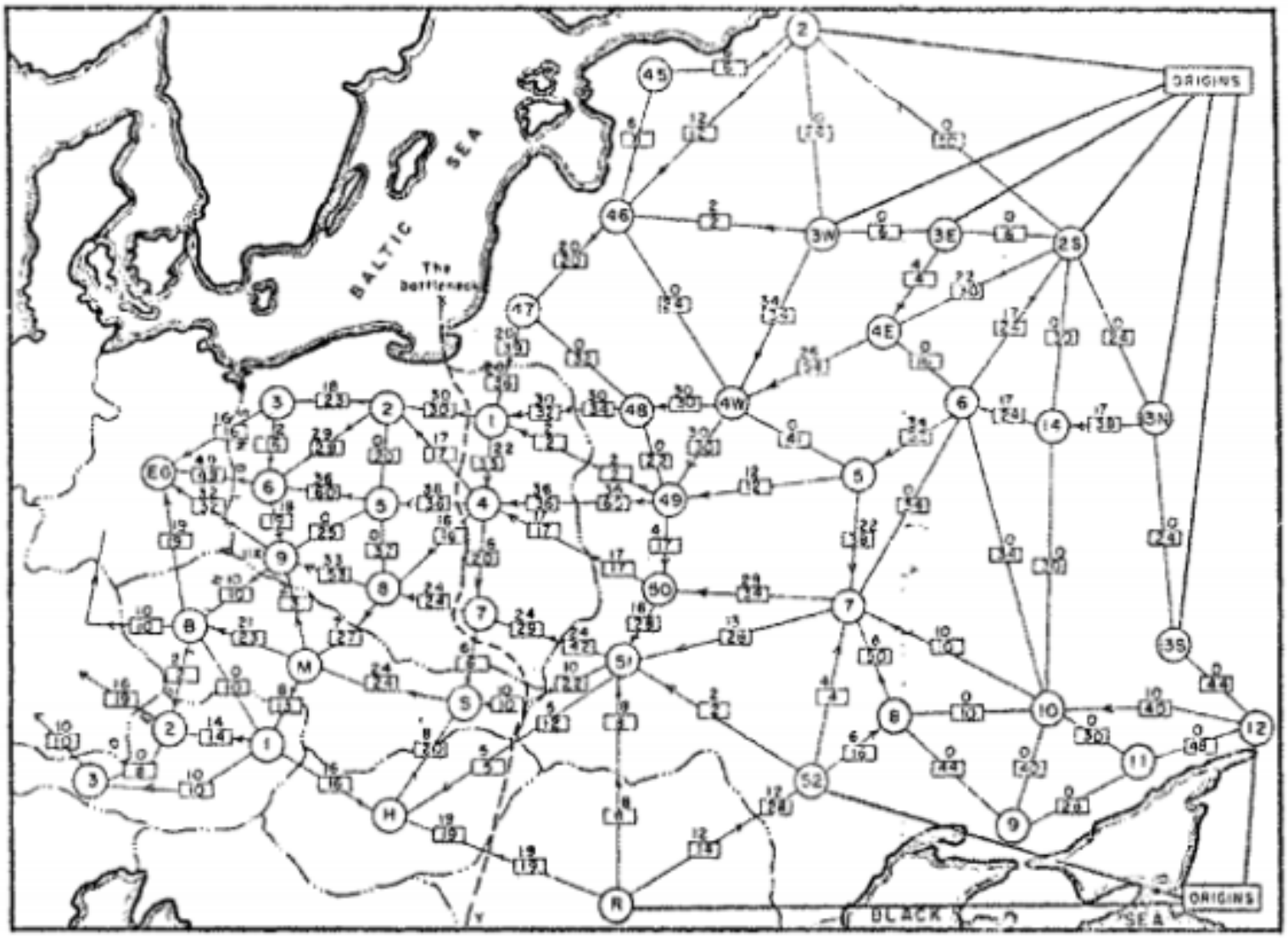
- a graph to model flow through edges (pipes)
- each edge has a capacity an upper bound on the flow rate (pipes have different sizes)
- Want to maximize rate of flow from the source to the sink

Tons of applications

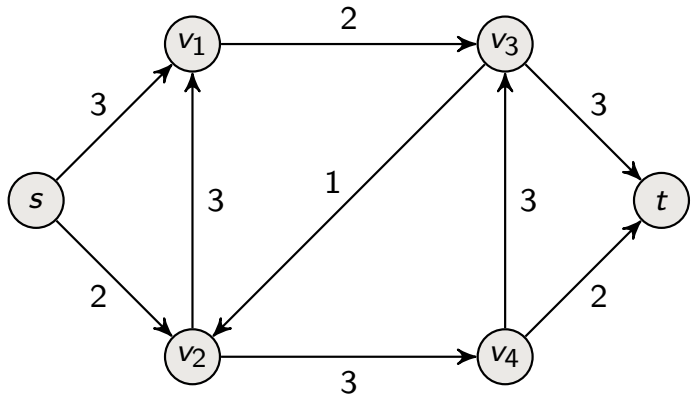
Tons of applications



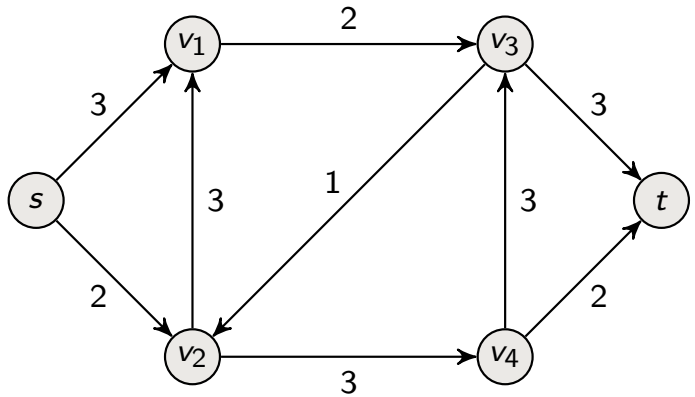
Tons of applications



Flow Network (formally)

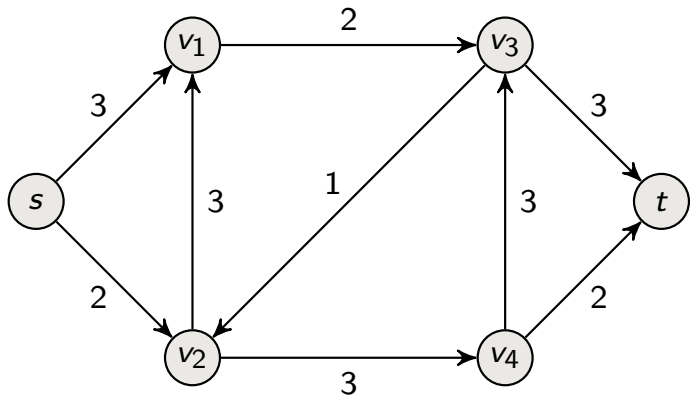


Flow Network (formally)



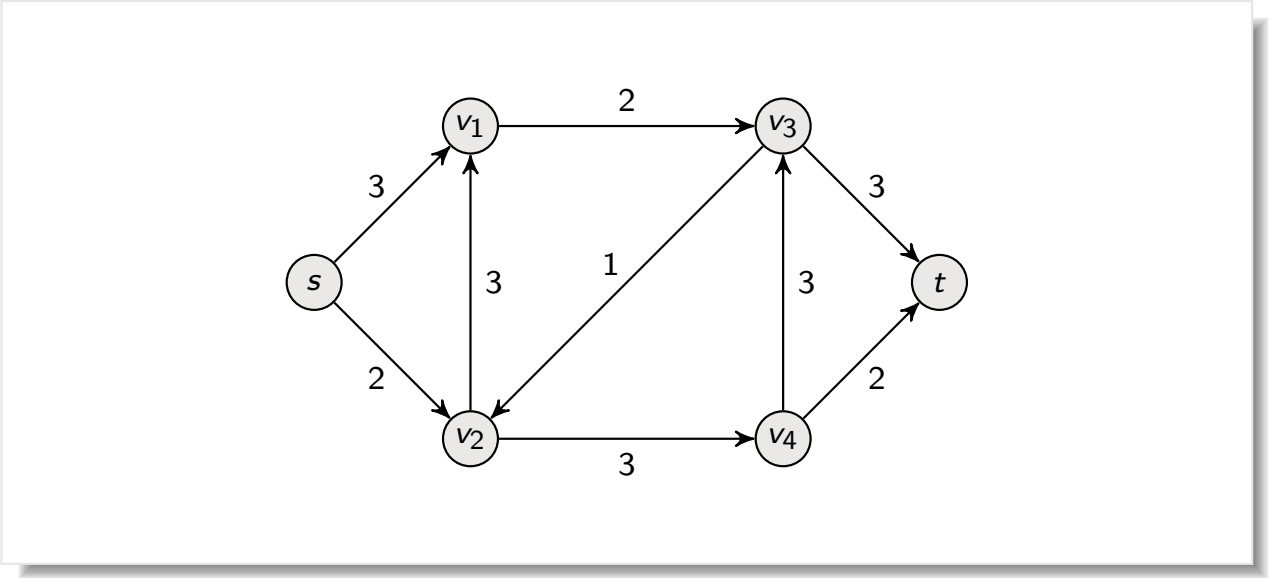
- Directed graph $G = (V, E)$

Flow Network (formally)



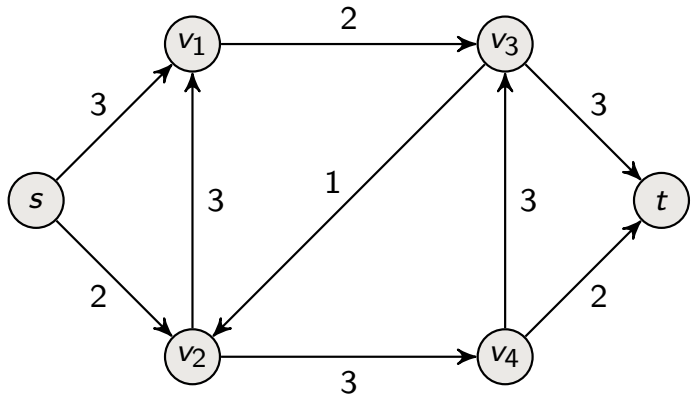
- Directed graph $G = (V, E)$
- Each edge (u, v) has a capacity $c(u, v) \geq 0$ ($c(u, v) = 0$ if $(u, v) \notin E$)

Flow Network (formally)



- Directed graph $G = (V, E)$
- Each edge (u, v) has a capacity $c(u, v) \geq 0$ ($c(u, v) = 0$ if $(u, v) \notin E$)
- Source s and sink t (flow goes from s to t)

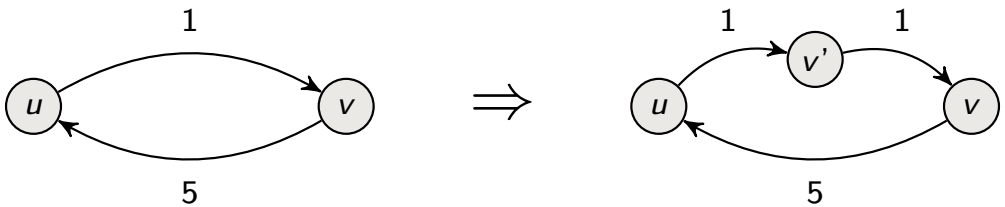
Flow Network (formally)



- Directed graph $G = (V, E)$
- Each edge (u, v) has a capacity $c(u, v) \geq 0$ ($c(u, v) = 0$ if $(u, v) \notin E$)
- Source s and sink t (flow goes from s to t)
- No antiparallel edges (assumed w.l.o.g. for simplicity)

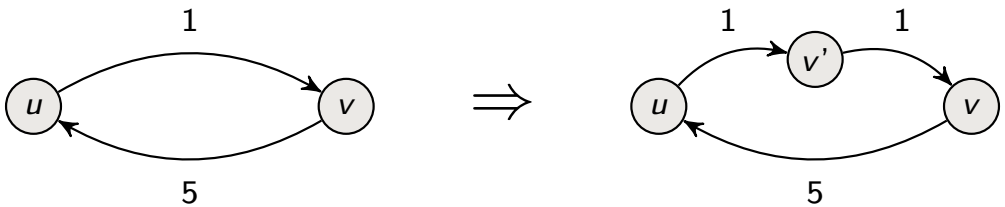
Why is “no antiparallel edges” w.l.o.g.?

Why is “no antiparallel edges” w.l.o.g.?



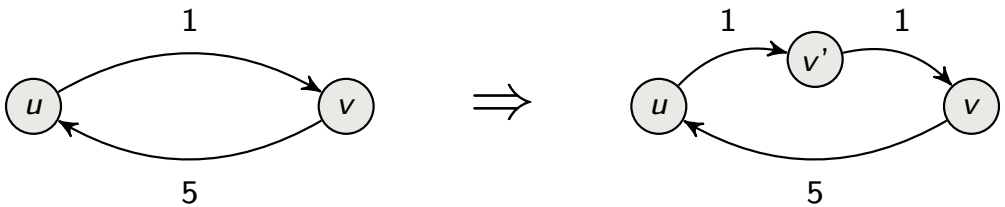
- If there are two parallel edges (u, v) and (v, u), choose one of them say (u, v)

Why is “no antiparallel edges” w.l.o.g.?



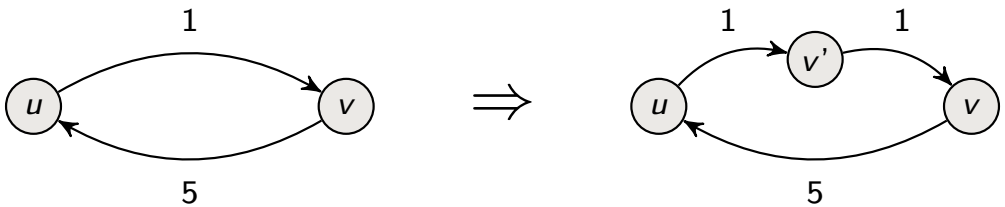
- If there are two parallel edges (u, v) and (v, u), choose one of them say (u, v)
- Create a new vertex v'

Why is “no antiparallel edges” w.l.o.g.?



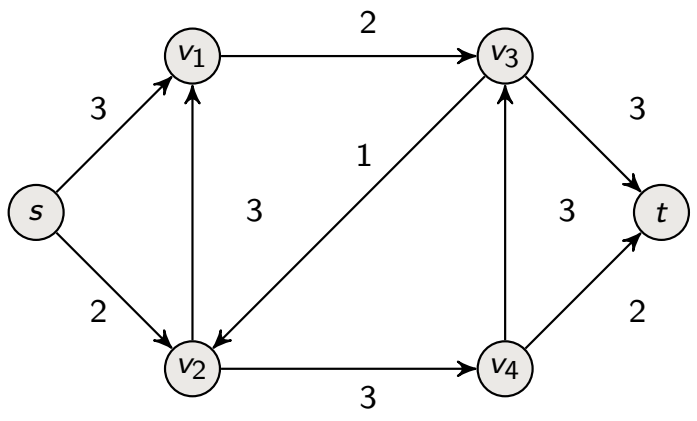
- If there are two parallel edges (u, v) and (v, u), choose one of them say (u, v)
- Create a new vertex v'
- Replace (u, v) by two new edges (u, v') and (v', v) with $c(u, v') = c(v', u) = c(u, v)$

Why is "no antiparallel edges" w.l.o.g.?



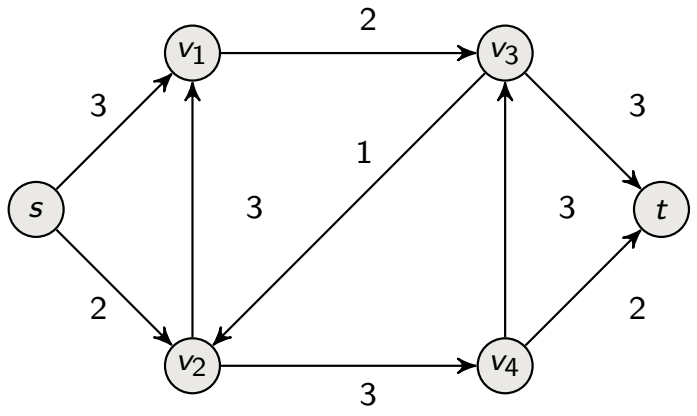
- If there are two parallel edges (u, v) and (v, u), choose one of them say (u, v)
- Create a new vertex v'
- Replace (u, v) by two new edges (u, v') and (v', v) with $c(u, v') = c(v', u) = c(u, v)$
- Repeat this $O(E)$ times to get an equivalent flow network with no antiparallel edges.

Definition of a flow



A flow is a function $f : V \times V \to \mathbb{R}$ satisfying:

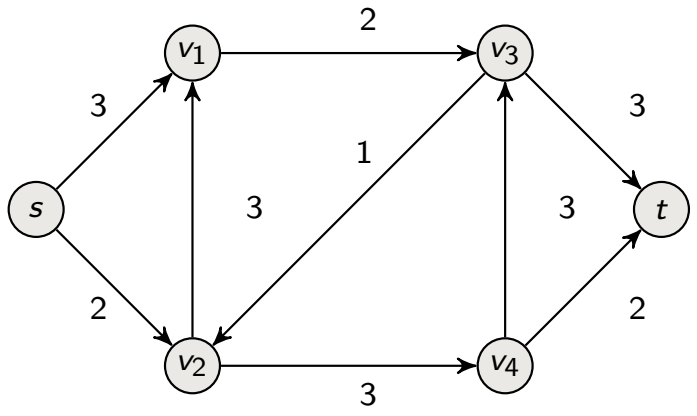
Definition of a flow



A flow is a function $f : V \times V \to \mathbb{R}$ satisfying:

Capacity constraint: For all $u, v \in V : 0 \leq f(u, v) \leq c(u, v)$

Definition of a flow



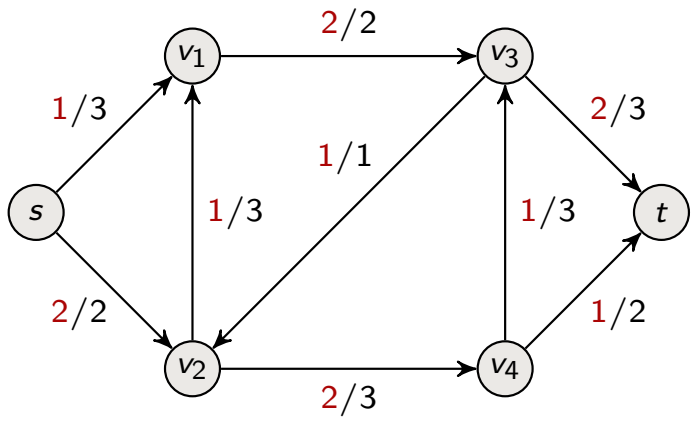
A flow is a function $f : V \times V \to \mathbb{R}$ satisfying:

Capacity constraint: For all $u, v \in V$: $0 \leq f(u, v) \leq c(u, v)$

Flow conservation: For all $u \in V \setminus \{s, t\}$,

$$\underbrace{\sum_{v \in V} f(v, u)}_{\text{flow into } u} = \underbrace{\sum_{v \in V} f(u, v)}_{\text{flow out of } u}$$

Definition of a flow



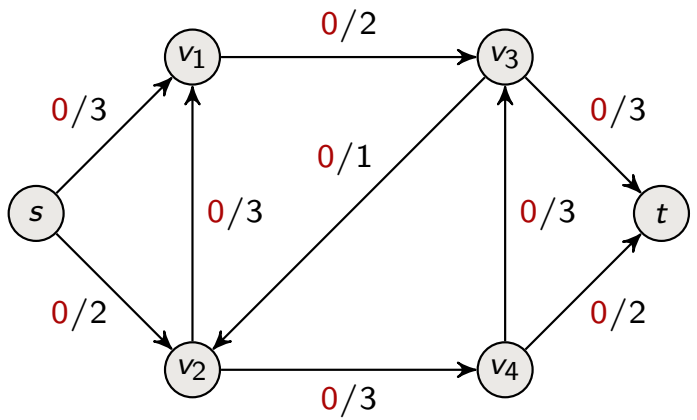
A flow is a function $f : V \times V \to \mathbb{R}$ satisfying:

Capacity constraint: For all $u, v \in V : 0 \leq f(u, v) \leq c(u, v)$

Flow conservation: For all $u \in V \setminus \{s, t\}$,

$$\underbrace{\sum_{v \in V} f(v, u)}_{\text{flow into } u} = \underbrace{\sum_{v \in V} f(u, v)}_{\text{flow out of } u}$$

Definition of a flow



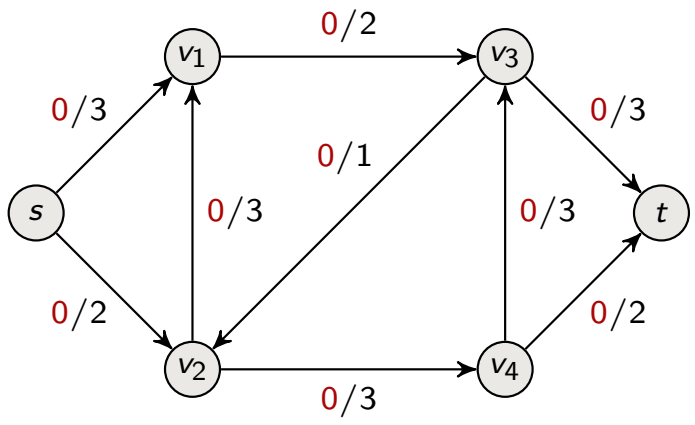
A flow is a function $f : V \times V \to \mathbb{R}$ satisfying:

Capacity constraint: For all $u, v \in V : 0 \leq f(u, v) \leq c(u, v)$

Flow conservation: For all $u \in V \setminus \{s, t\}$,

$$\underbrace{\sum_{v \in V} f(v, u)}_{\text{flow into } u} = \underbrace{\sum_{v \in V} f(u, v)}_{\text{flow out of } u}$$

Value of a flow

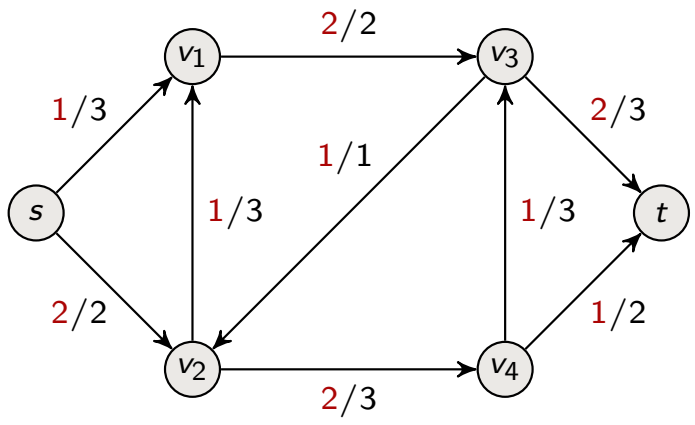


Value of a flow $f = |f|$

$$= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

$$= \text{flow out of source} - \text{flow into source}$$

Value of a flow

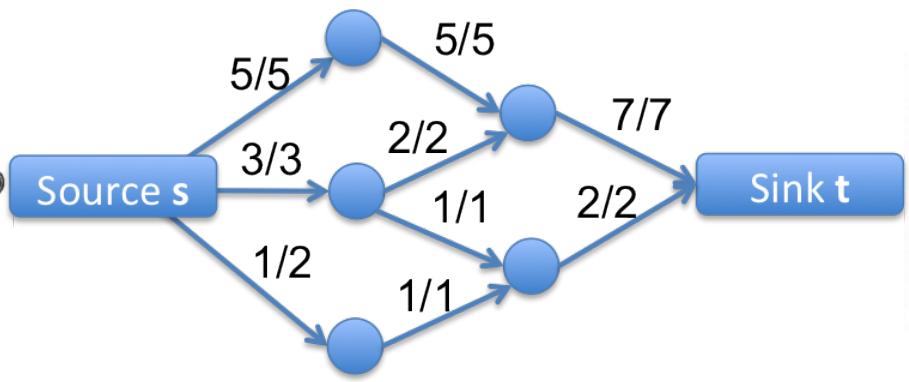


Value of a flow $f = |f|$

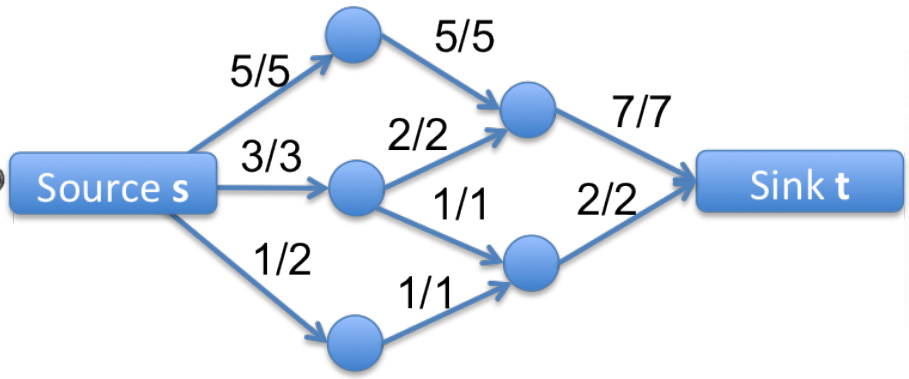
$$= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

$$= \text{flow out of source} - \text{flow into source}$$

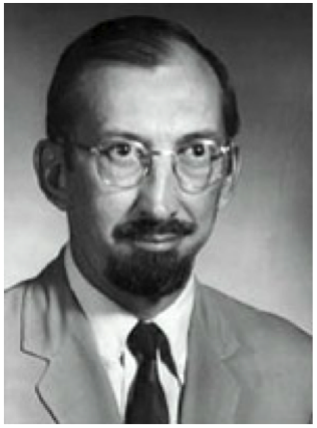
What's the value of this flow?



What's the value of this flow? 9



L. R. Ford, Jr. (1927-)



D, R, Fulkerson (1924-1976)

MAXIMUM-FLOW PROBLEM

Ford-Fulkerson Method

The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while** exists an augmenting path p in the residual network G_f
3. augment flow f along p
4. **return** f

The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while** exists an **augmenting path** p in the **residual network** G_f
3. **augment flow** f along p
4. **return** f

The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while** exists an **augmenting path** p in the **residual network** G_f
3. **augment flow** f along p
4. **return** f

Basic idea:

The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while** exists an **augmenting path** p in the **residual network** G_f
3. **augment flow** f along p
4. **return** f

Basic idea:

- As long as there is a path from source to sink, with available capacity on all edges in the path

The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

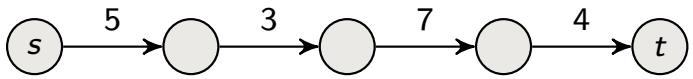
1. Initialize flow f to 0
2. **while** exists an **augmenting path** p in the **residual network** G_f
3. **augment flow** f along p
4. **return** f

Basic idea:

- As long as there is a path from source to sink, with available capacity on all edges in the path
- send flow along one of these paths and then we find another path and so on

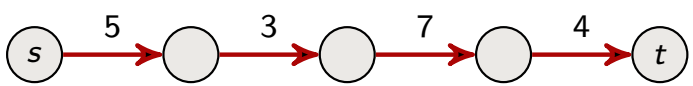
Applying the basic idea to examples

- As long as there is a path from source to sink, with available capacity on all edges in the path
- send flow along one of these paths and then we find another path and so on



Applying the basic idea to examples

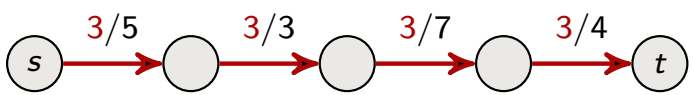
- As long as there is a path from source to sink, with available capacity on all edges in the path
- send flow along one of these paths and then we find another path and so on



Exists a path **p** from s to t with remaining capacity
$\Rightarrow$ Push flow on **p**

Applying the basic idea to examples

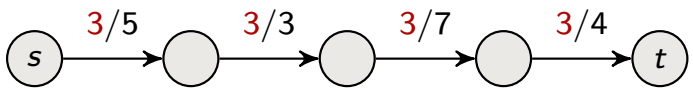
- As long as there is a path from source to sink, with available capacity on all edges in the path
- send flow along one of these paths and then we find another path and so on



Exists a path **p** from s to t with remaining capacity
$\Rightarrow$ Push flow on **p**

Applying the basic idea to examples

- As long as there is a path from source to sink, with available capacity on all edges in the path
- send flow along one of these paths and then we find another path and so on

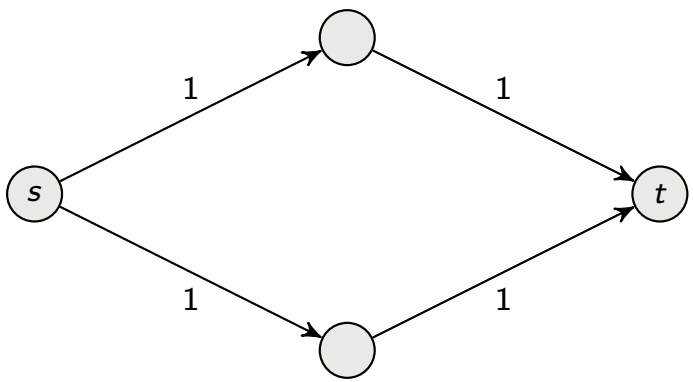


No path from s to t
with remaining capacity

and the flow is maximum

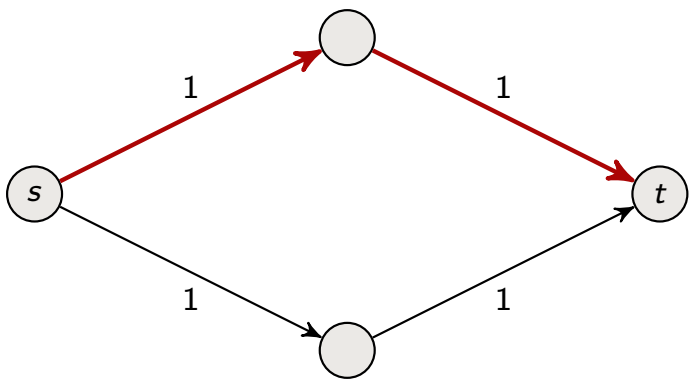
Applying the basic idea to examples

- As long as there is a path from source to sink, with available capacity on all edges in the path
- send flow along one of these paths and then we find another path and so on



Applying the basic idea to examples

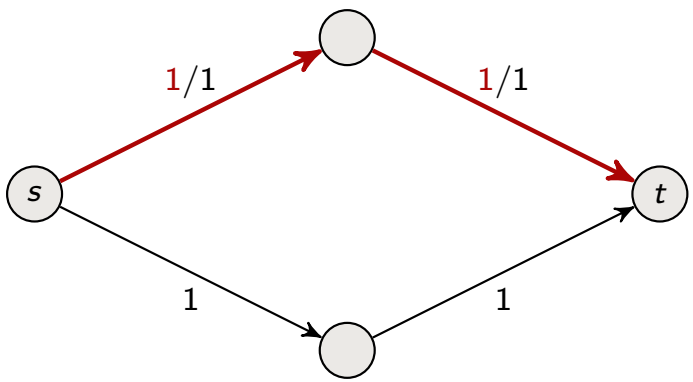
- As long as there is a path from source to sink, with available capacity on all edges in the path
- send flow along one of these paths and then we find another path and so on



Exists a path **p** from s to t with remaining capacity
$\Rightarrow$ Push flow on **p**

Applying the basic idea to examples

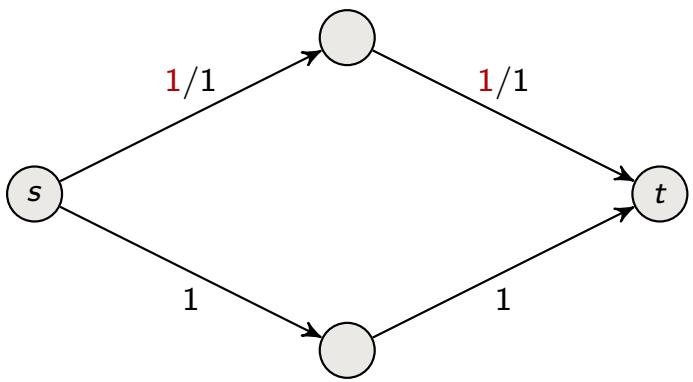
- As long as there is a path from source to sink, with available capacity on all edges in the path
- send flow along one of these paths and then we find another path and so on



Exists a path **p** from s to t with remaining capacity
$\Rightarrow$ Push flow on **p**

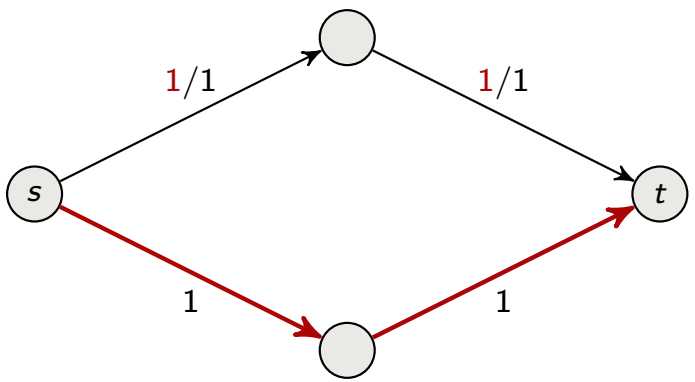
Applying the basic idea to examples

- As long as there is a path from source to sink, with available capacity on all edges in the path
- send flow along one of these paths and then we find another path and so on



Applying the basic idea to examples

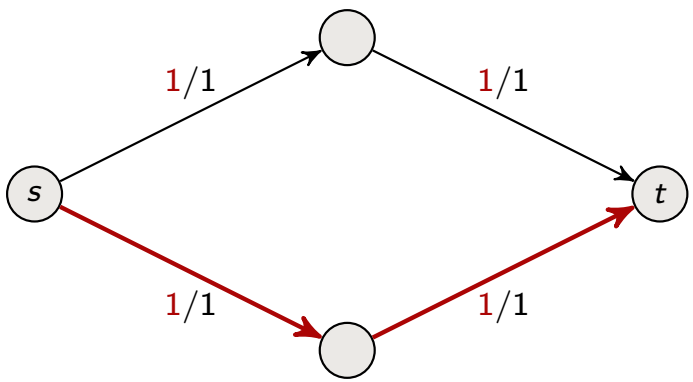
- As long as there is a path from source to sink, with available capacity on all edges in the path
- send flow along one of these paths and then we find another path and so on



Exists a path **p** from s to t with remaining capacity
$\Rightarrow$ Push flow on **p**

Applying the basic idea to examples

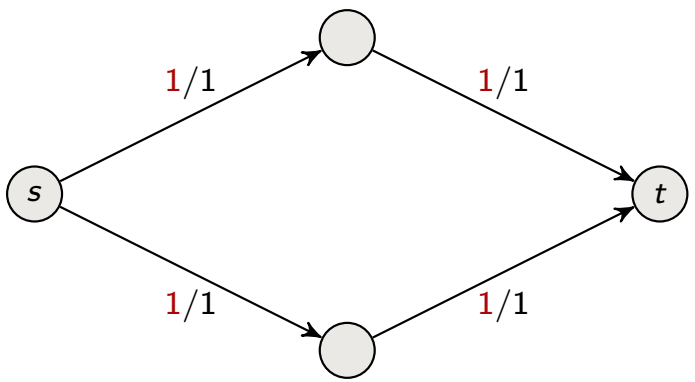
- As long as there is a path from source to sink, with available capacity on all edges in the path
- send flow along one of these paths and then we find another path and so on



Exists a path **p** from s to t with remaining capacity
$\Rightarrow$ Push flow on **p**

Applying the basic idea to examples

- As long as there is a path from source to sink, with available capacity on all edges in the path
- send flow along one of these paths and then we find another path and so on

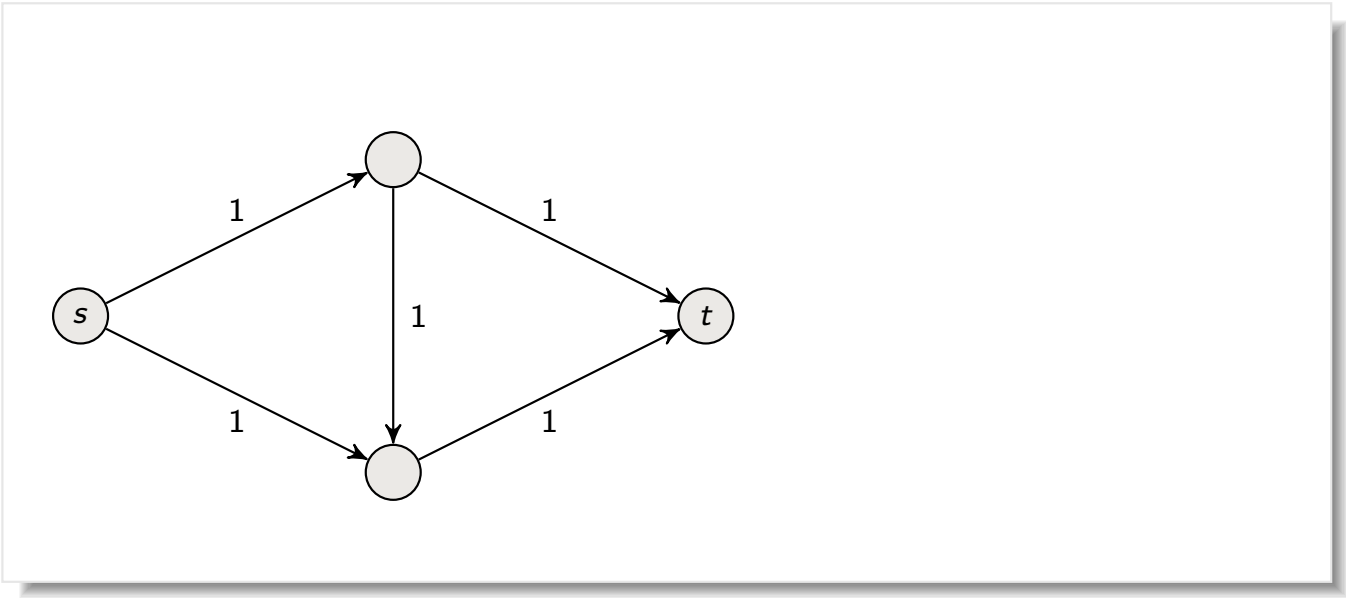


No path from s to t
with remaining capacity

and the flow is maximum

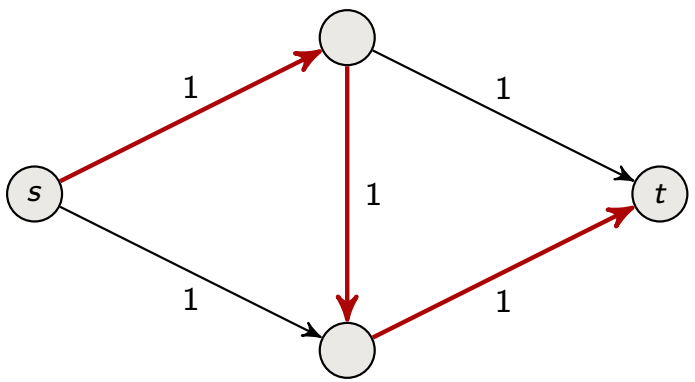
Applying the basic idea to examples

- As long as there is a path from source to sink, with available capacity on all edges in the path
- send flow along one of these paths and then we find another path and so on



Applying the basic idea to examples

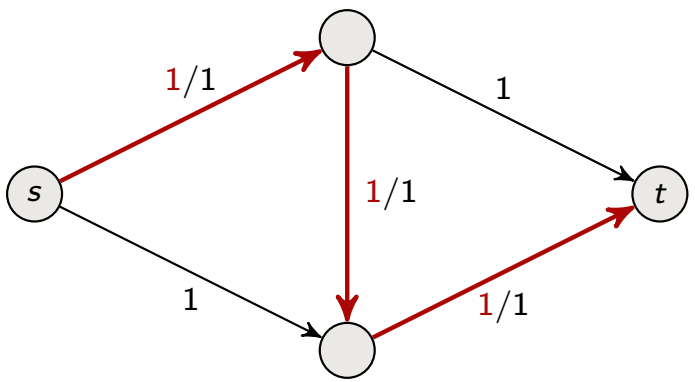
- As long as there is a path from source to sink, with available capacity on all edges in the path
- send flow along one of these paths and then we find another path and so on



Exists a path **p** from s to t with remaining capacity
$\Rightarrow$ Push flow on **p**

Applying the basic idea to examples

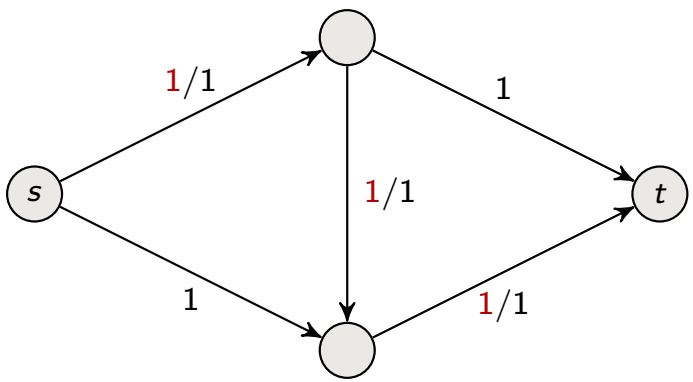
- As long as there is a path from source to sink, with available capacity on all edges in the path
- send flow along one of these paths and then we find another path and so on



Exists a path **p** from s to t with remaining capacity
$\Rightarrow$ Push flow on **p**

Applying the basic idea to examples

- As long as there is a path from source to sink, with available capacity on all edges in the path
- send flow along one of these paths and then we find another path and so on

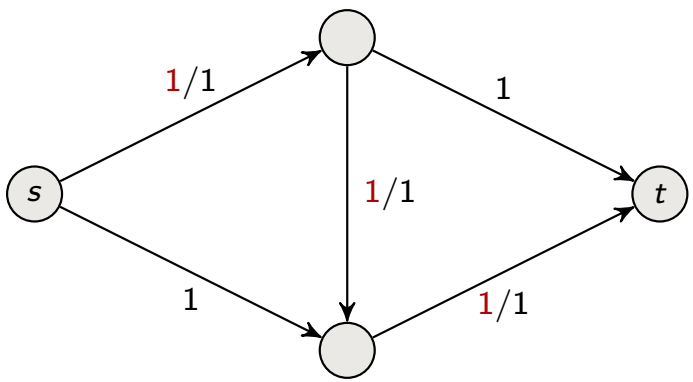


No path from s to t
with remaining capacity

but the flow is **not maximum**

Applying the basic idea to examples

- As long as there is a path from source to sink, with available capacity on all edges in the path
- send flow along one of these paths and then we find another path and so on



No path from s to t
with remaining capacity

but the flow is **not maximum**

What went wrong? How can we fix it?

The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while** exists an augmenting path p in the **residual network** G_f
3. augment flow f along p
4. **return** f

Residual network

- Given a flow f and a network $G = (V, E)$
- the residual network consists of edges with capacities that represent how we can change the flow on the edges

Residual network

- Given a flow f and a network $G = (V, E)$
- the residual network consists of edges with capacities that represent how we can change the flow on the edges

Residual capacity:

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E \\ f(v, u) & \text{if } (v, u) \in E \\ 0 & \text{otherwise} \end{cases}$$

Residual network

- Given a flow f and a network $G = (V, E)$
- the residual network consists of edges with capacities that represent how we can change the flow on the edges

Residual capacity:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise} \end{cases}$$

Amount of capacity left

Amount of flow that can be reversed

Residual network

- Given a flow f and a network $G = (V, E)$
- the residual network consists of edges with capacities that represent how we can change the flow on the edges

Residual capacity:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise} \end{cases}$$

Amount of capacity left

Amount of flow that can be reversed

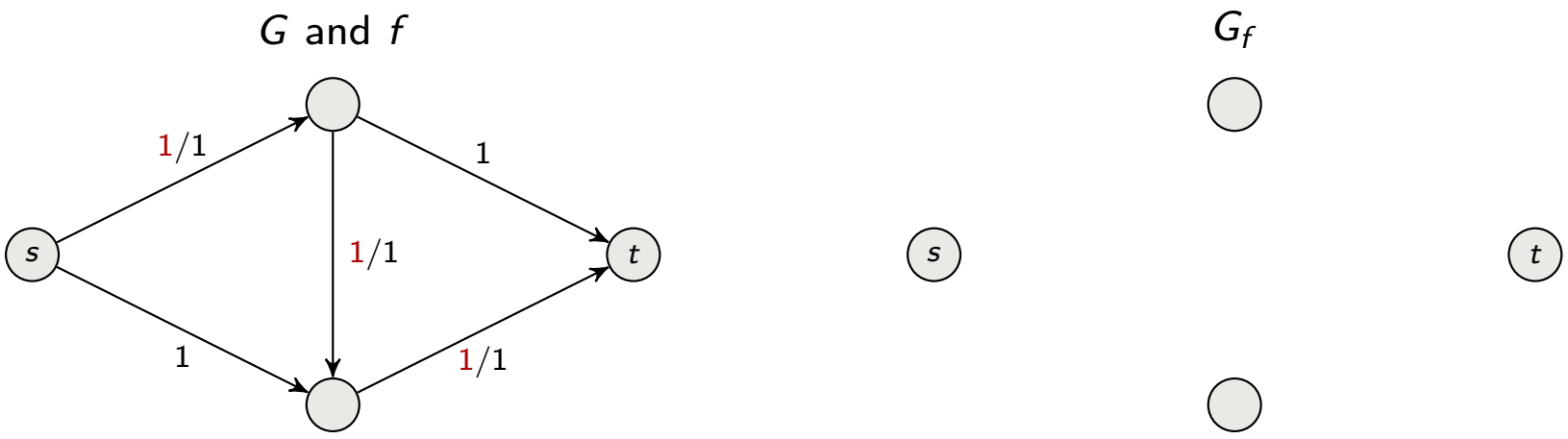
Residual network:

$G_f = (V, E_f)$ where $E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$

Examples

Residual network: $G_f = (V, E_f)$ where $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$ and

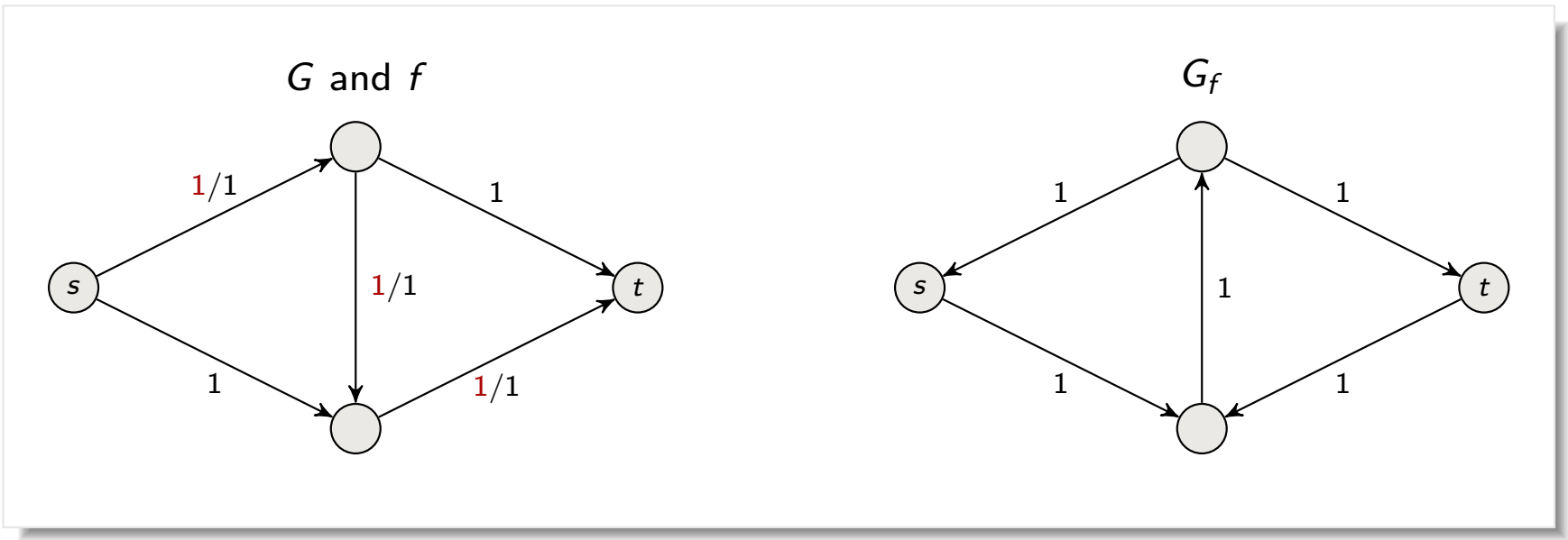
$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E \\ f(v, u) & \text{if } (v, u) \in E \\ 0 & \text{otherwise} \end{cases}$$



Examples

Residual network: $G_f = (V, E_f)$ where $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$ and

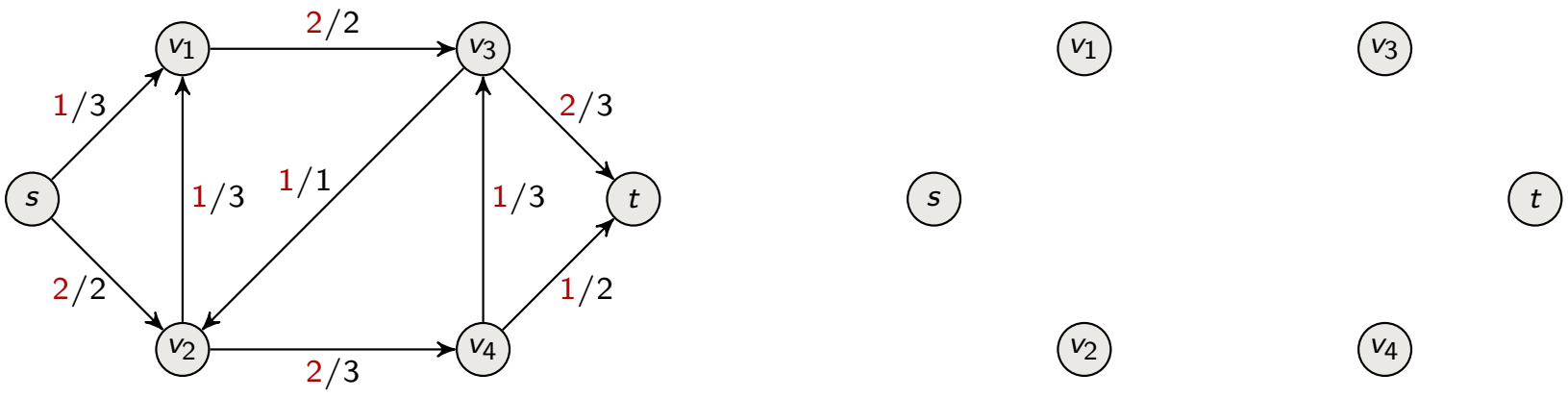
$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E \\ f(v, u) & \text{if } (v, u) \in E \\ 0 & \text{otherwise} \end{cases}$$



Examples

Residual network: $G_f = (V, E_f)$ where $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$ and

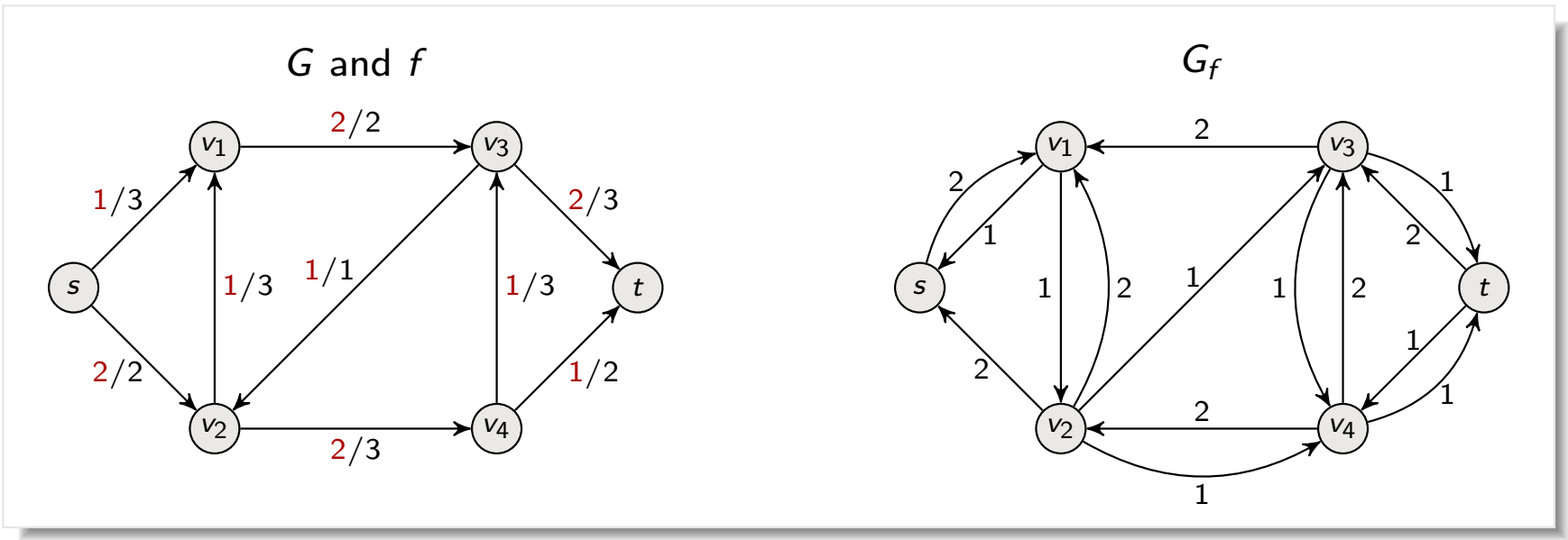
$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E \\ f(v, u) & \text{if } (v, u) \in E \\ 0 & \text{otherwise} \end{cases}$$



Examples

Residual network: $G_f = (V, E_f)$ where $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$ and

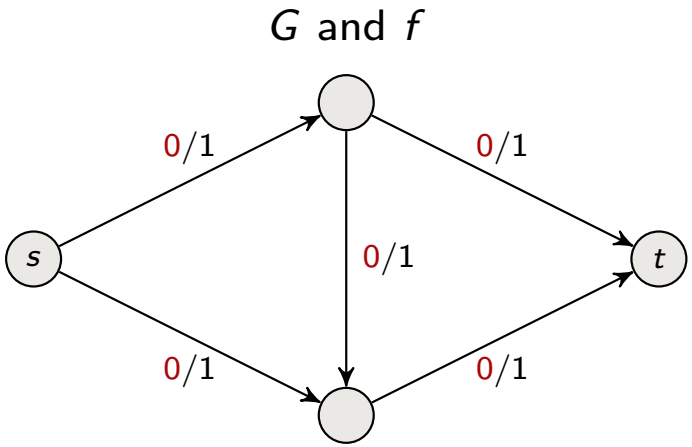
$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E \\ f(v, u) & \text{if } (v, u) \in E \\ 0 & \text{otherwise} \end{cases}$$



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. **Initialize flow f to 0**
2. **while** exists an augmenting path p in the residual network G_f
3. augment flow f along p
4. **return** f

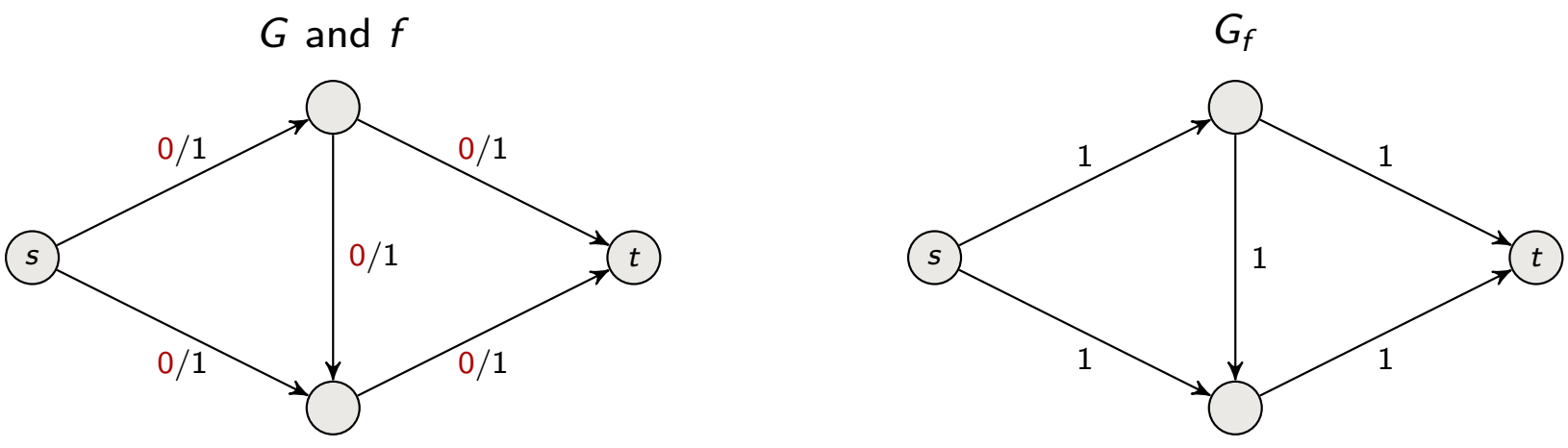


The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f

Augmenting path = simple path from s to t

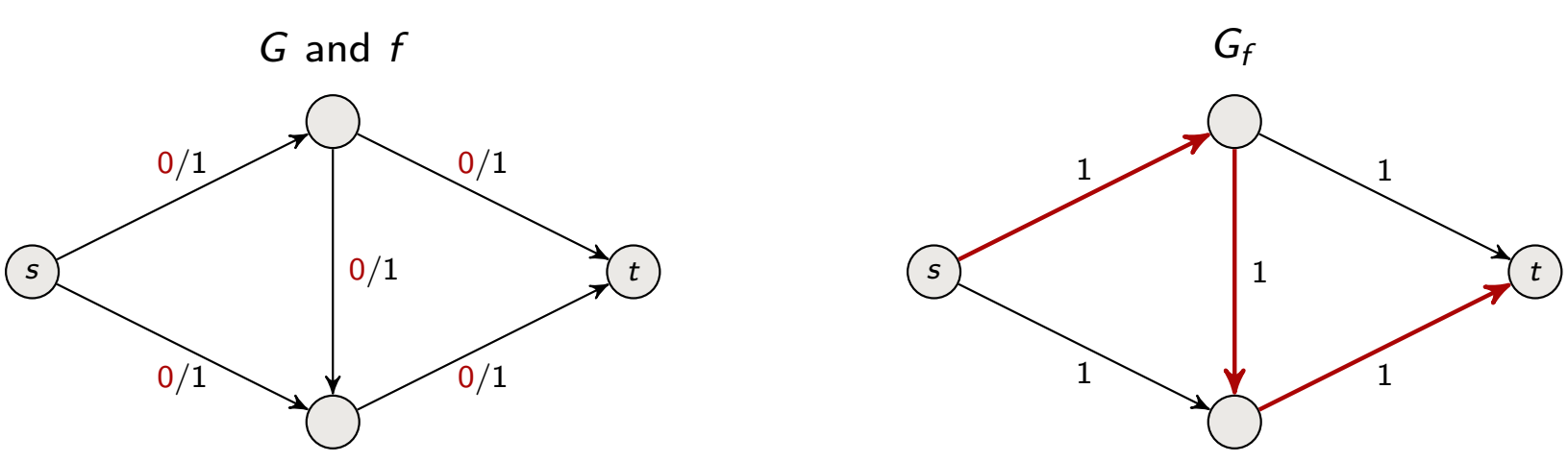


The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f

Exists augmenting path **p**

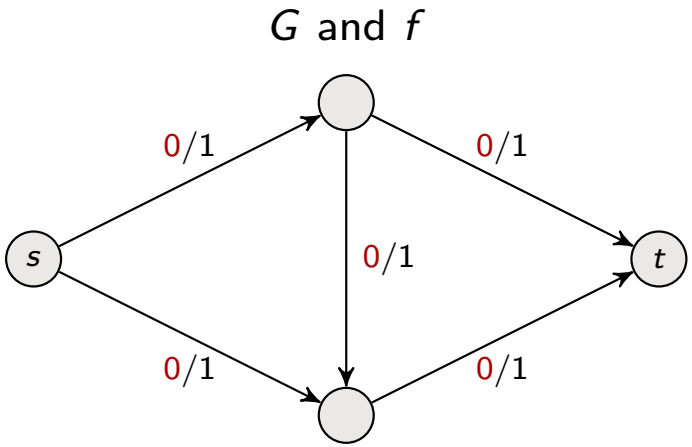


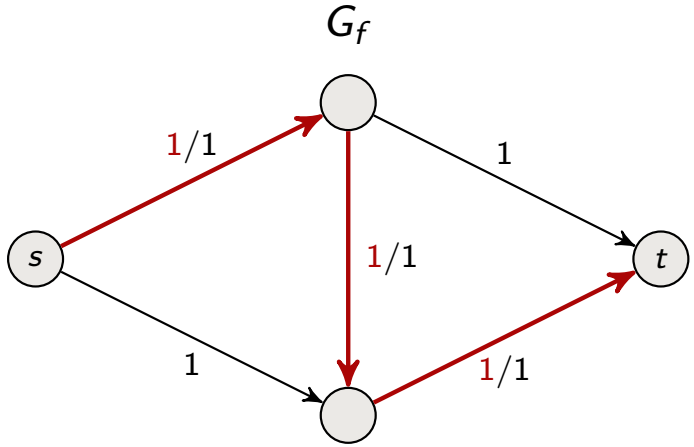
The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f

Exists augmenting path **p** with flow $\mathbf{f_p}$ of value = min capacity on **p**



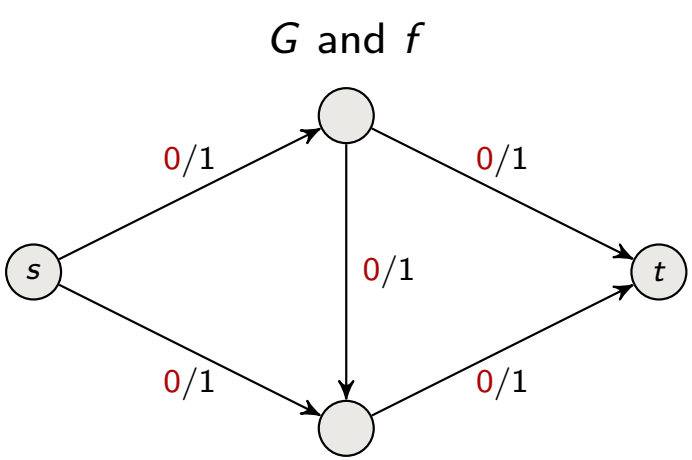


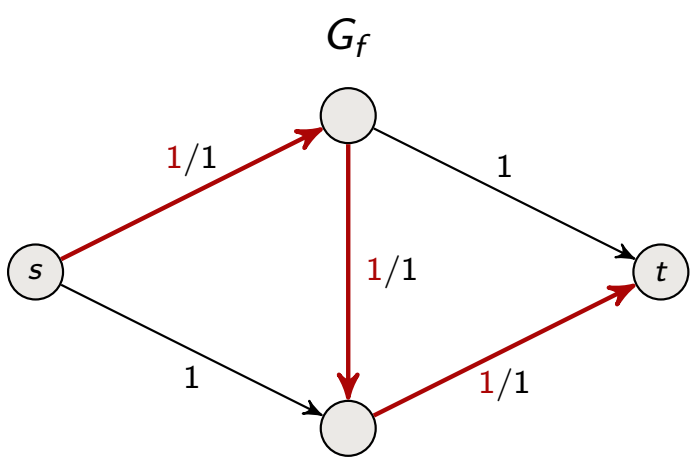
The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while** exists an augmenting path p in the residual network G_f
3. **augment flow f along p**
4. **return** f

f is updated by changing the flow on an edge $(\mathbf{u}, \mathbf{v})$ by $\mathbf{f_p}(\mathbf{u}, \mathbf{v}) - \mathbf{f_p}(\mathbf{v}, \mathbf{u})$



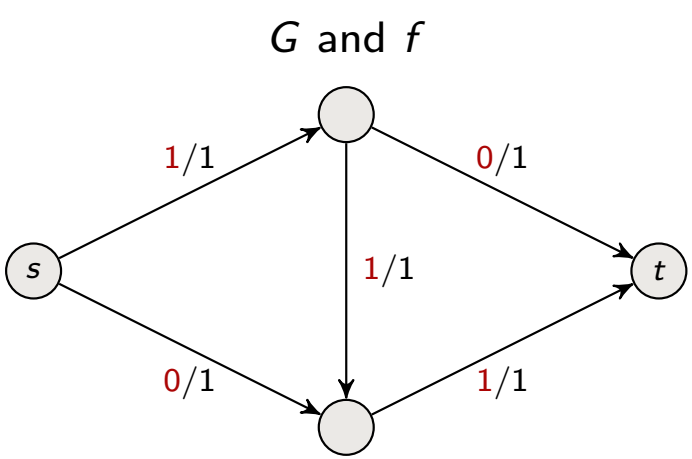


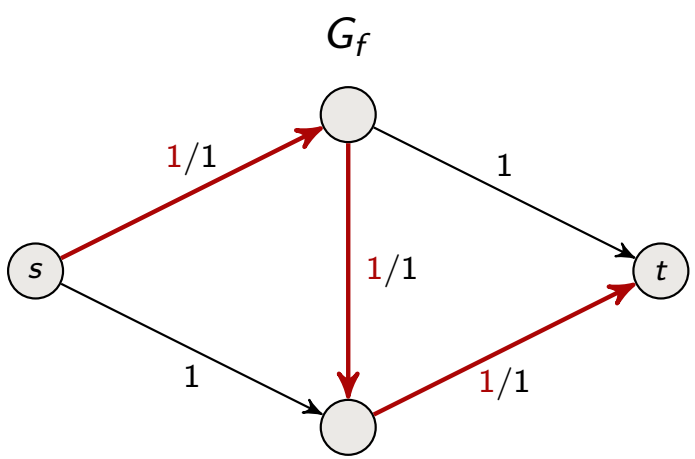
The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while** exists an augmenting path p in the residual network G_f
3. **augment flow f along p**
4. **return** f

f is updated by changing the flow on an edge $(\mathbf{u}, \mathbf{v})$ by $\mathbf{f_p}(\mathbf{u}, \mathbf{v}) - \mathbf{f_p}(\mathbf{v}, \mathbf{u})$

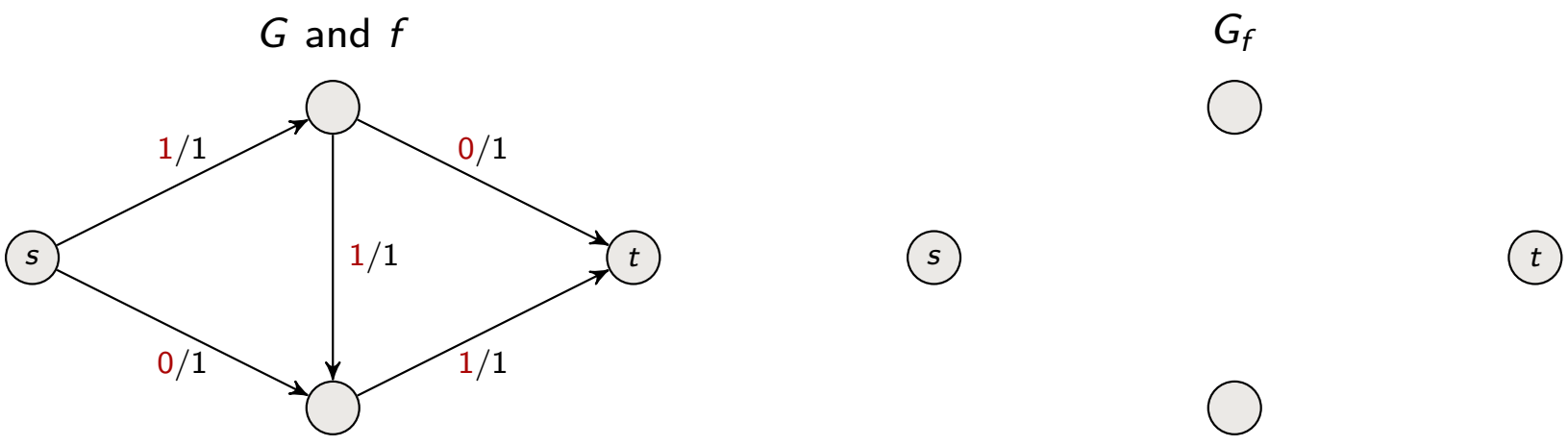




The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

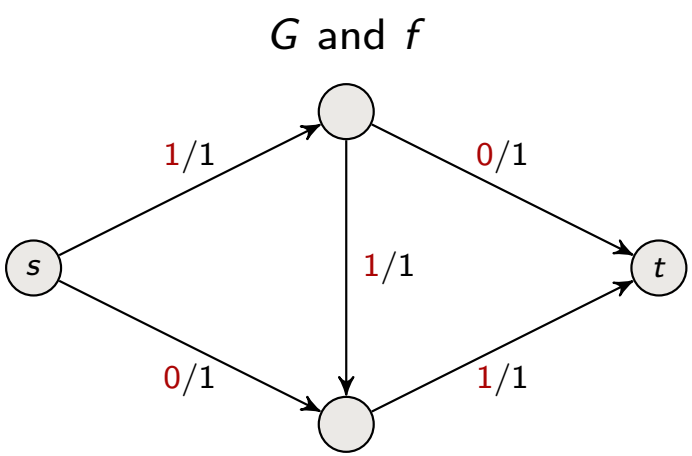
1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f

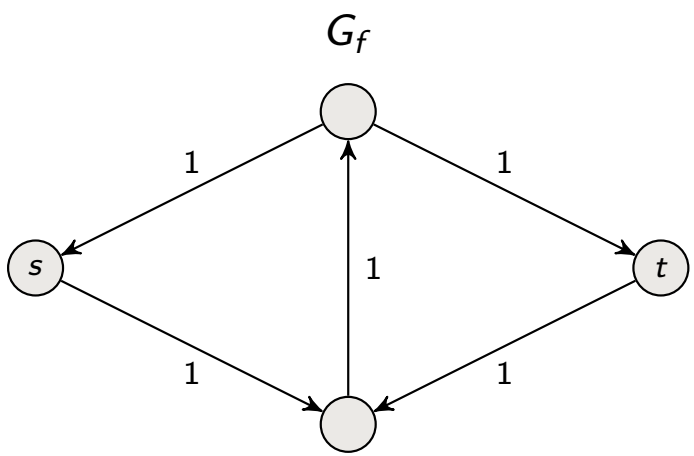


The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f

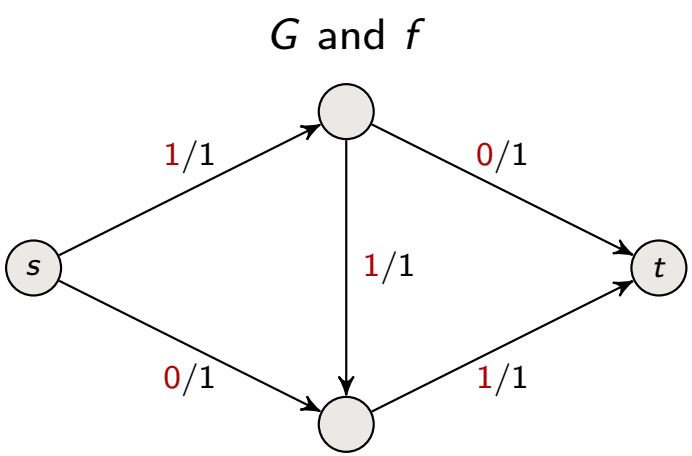


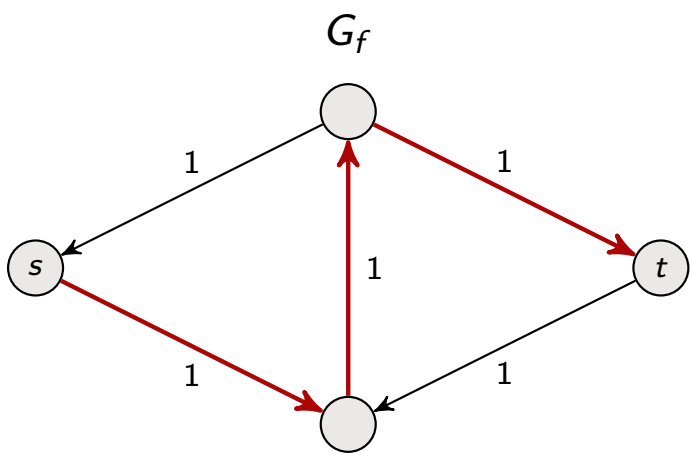


The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f

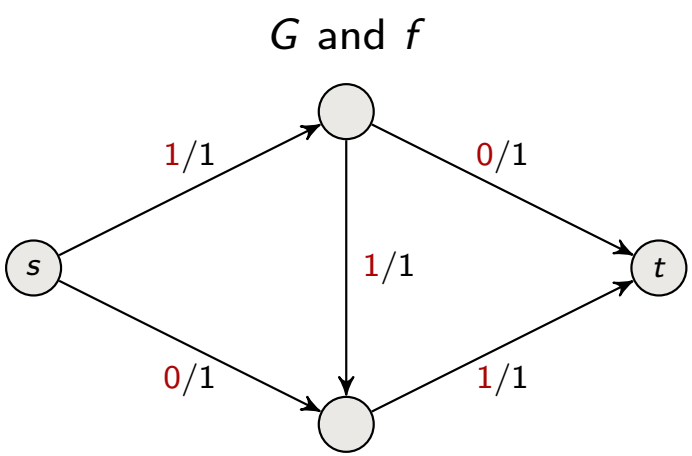


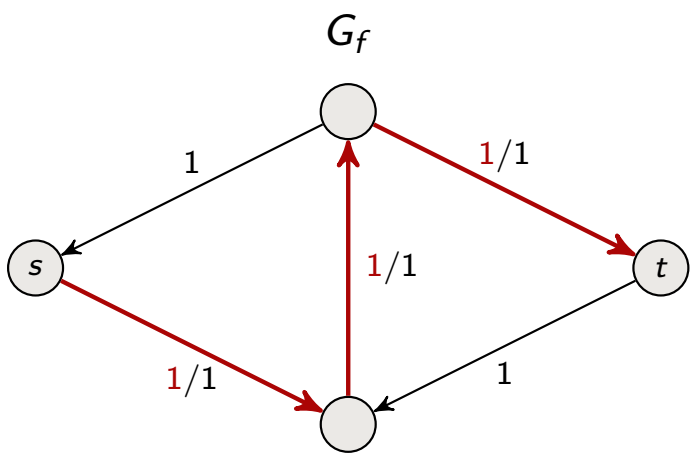


The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f

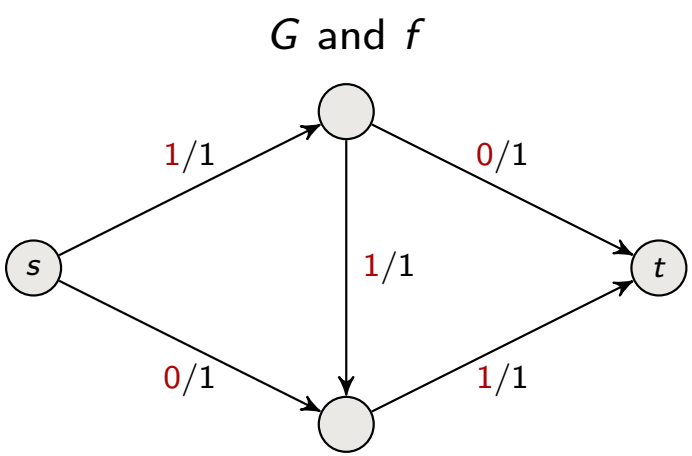


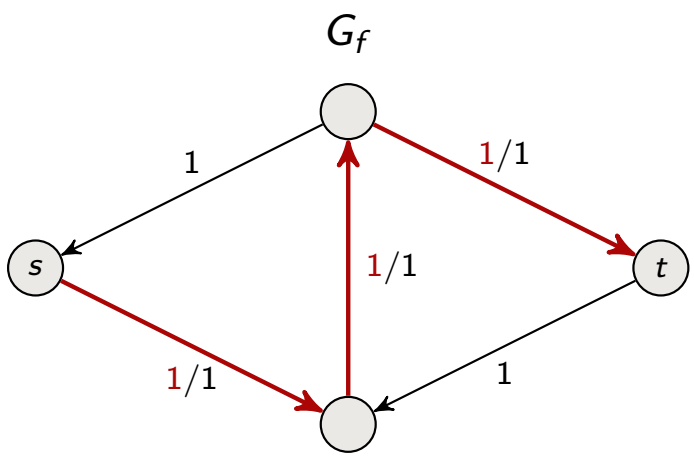


The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while** exists an augmenting path p in the residual network G_f
3. **augment flow f along p**
4. **return** f

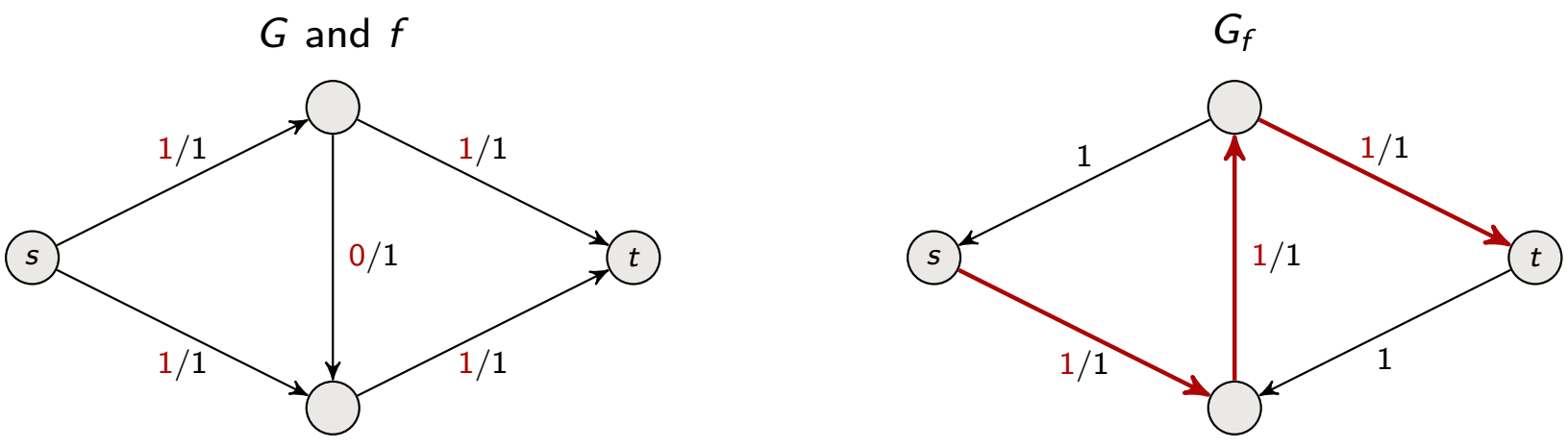




The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

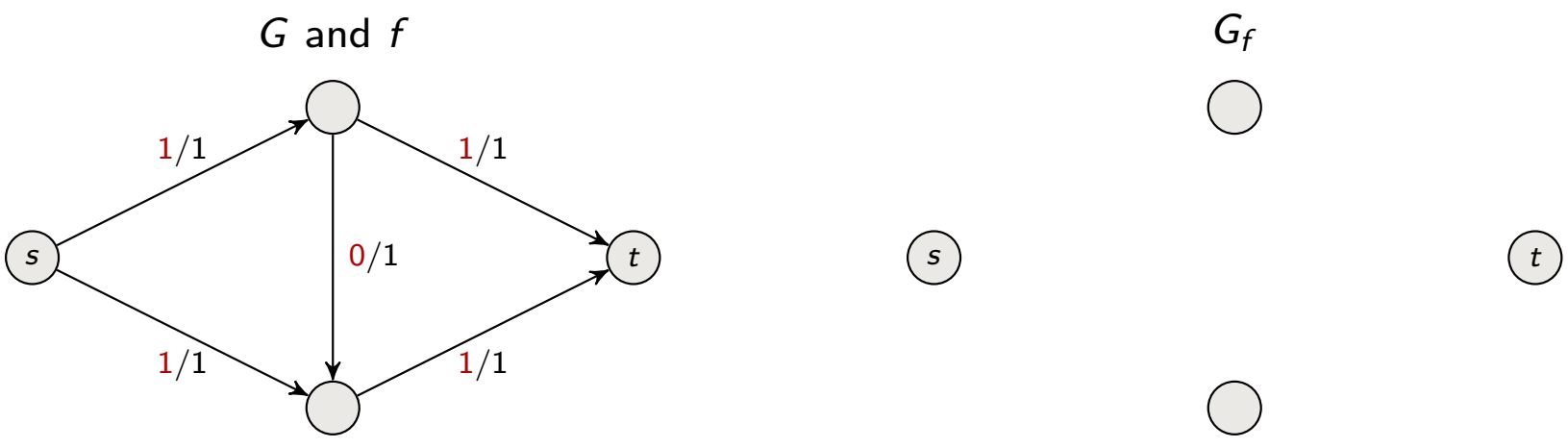
1. Initialize flow f to 0
2. **while** exists an augmenting path p in the residual network G_f
3. **augment flow f along p**
4. **return** f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

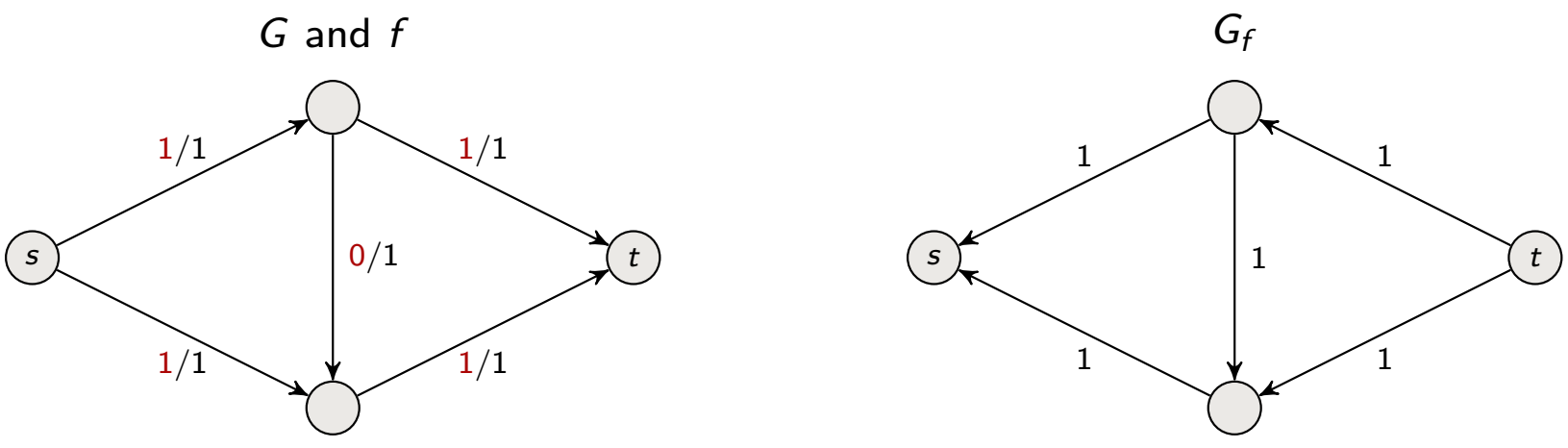
1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f



The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f



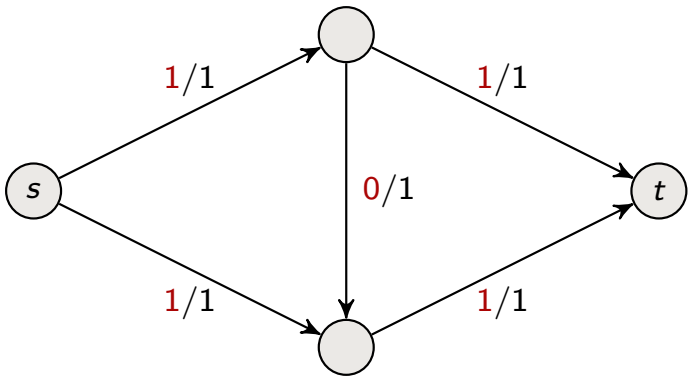
The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

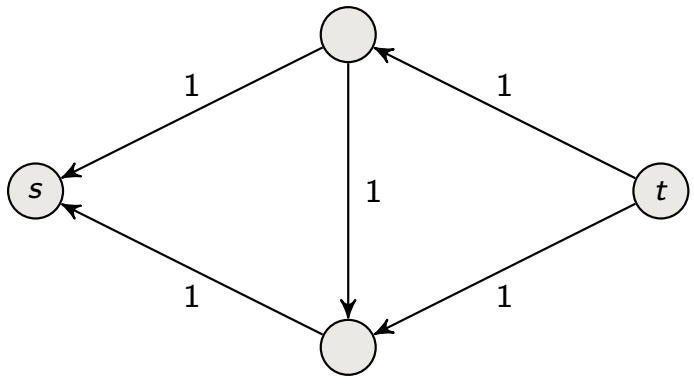
1. Initialize flow f to 0
2. **while exists an augmenting path p in the residual network G_f**
3. augment flow f along p
4. **return** f

No augmenting path and flow of value 2 is optimal

G and f



G_f



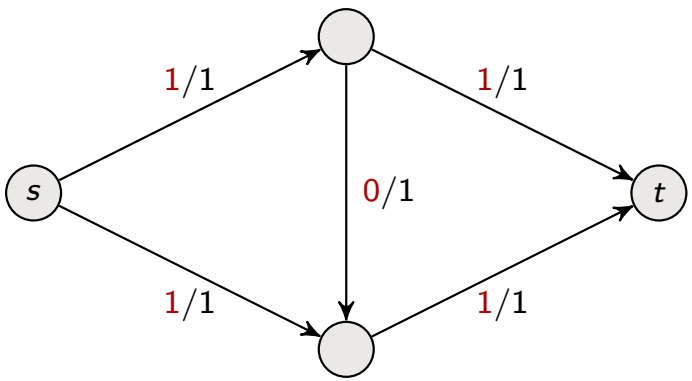
The Ford-Fulkerson Method'54

FORD-FULKERSON-METHOD(G, s, t):

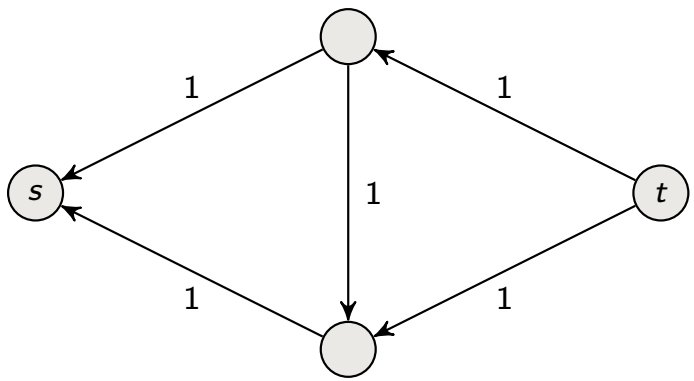
1. Initialize flow f to 0
2. **while** exists an augmenting path p in the residual network G_f
3. augment flow f along p
4. **return** f

No augmenting path and flow of value 2 is optimal

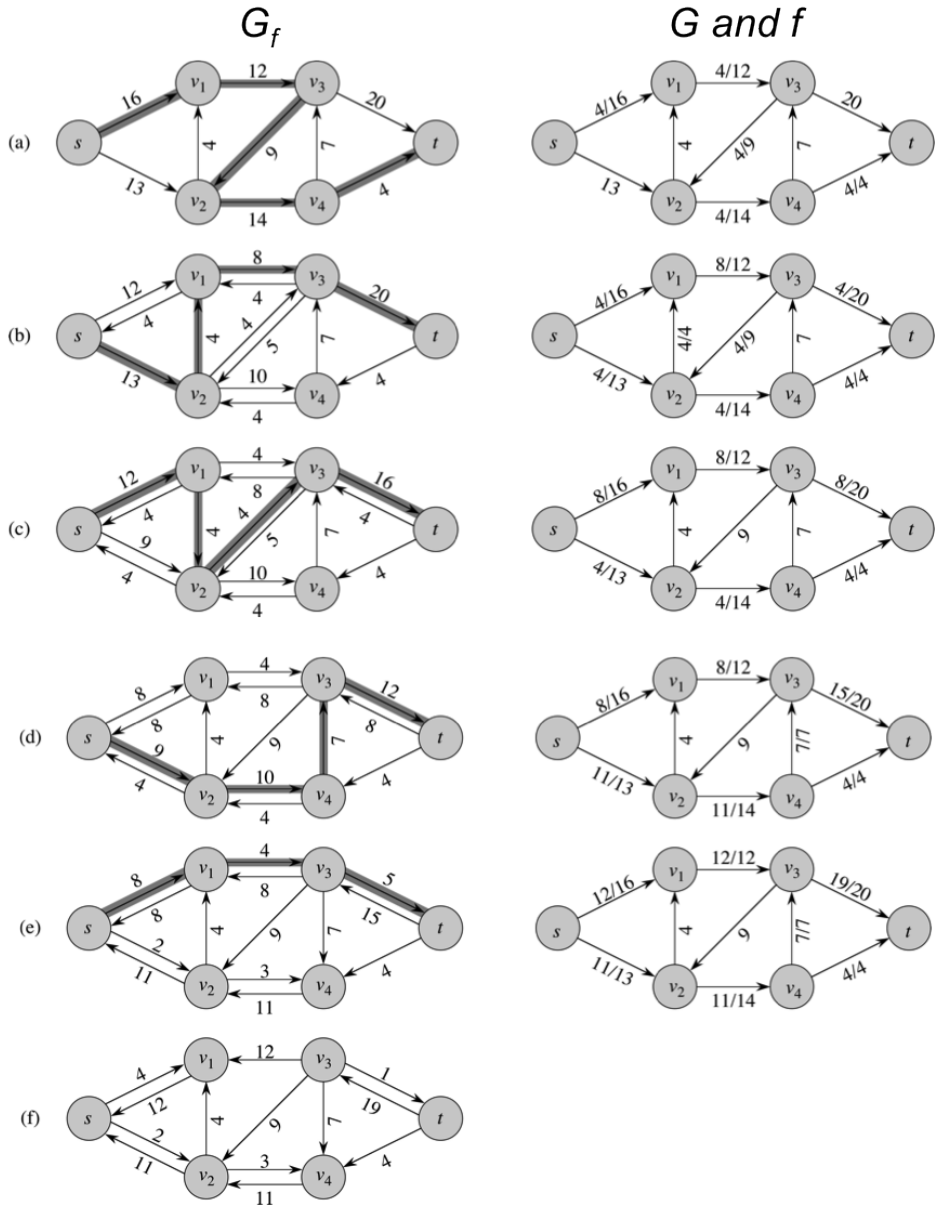
G and f



G_f



Study and understand
Example!



Summary

- Graphs fundamental object to study
- Two natural ways of traversing a graph: breadth-first search and depth-first search
- Topological sort of acyclic graphs by applying DFS and then order according to decreasing finishing times
- Strongly connected components
- Flow Networks