

# Decompositions of Graphs

Hengfeng Wei

hfwei@nju.edu.cn

June 12, 2018





John Hopcroft



Robert Tarjan



John Hopcroft



Robert Tarjan

*“For fundamental achievements in the design and analysis of algorithms and data structures.”*

*— Turing Award, 1986*

## DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS\*

ROBERT TARJAN†

**Abstract.** The value of depth-first search or “backtracking” as a technique for solving problems is illustrated by two examples. An improved version of an algorithm for finding the strongly connected components of a directed graph and an algorithm for finding the biconnected components of an undirect graph are presented. The space and time requirements of both algorithms are bounded by  $k_1V + k_2E + k_3$  for some constants  $k_1, k_2$ , and  $k_3$ , where  $V$  is the number of vertices and  $E$  is the number of edges of the graph being examined.

**Key words.** Algorithm, backtracking, biconnectivity, connectivity, depth-first, graph, search, spanning tree, strong-connectivity.

- ▶ “Depth-First Search And Linear Graph Algorithms” by [Robert Tarjan](#).

## DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS\*

ROBERT TARJAN†

**Abstract.** The value of depth-first search or “backtracking” as a technique for solving problems is illustrated by two examples. An improved version of an algorithm for finding the strongly connected components of a directed graph and an algorithm for finding the biconnected components of an undirect graph are presented. The space and time requirements of both algorithms are bounded by  $k_1V + k_2E + k_3$  for some constants  $k_1, k_2$ , and  $k_3$ , where  $V$  is the number of vertices and  $E$  is the number of edges of the graph being examined.

**Key words.** Algorithm, backtracking, biconnectivity, connectivity, depth-first, graph, search, spanning tree, strong-connectivity.

*“DFS is a powerful technique with many applications.”*

- ▶ “Depth-First Search And Linear Graph Algorithms” by Robert Tarjan.

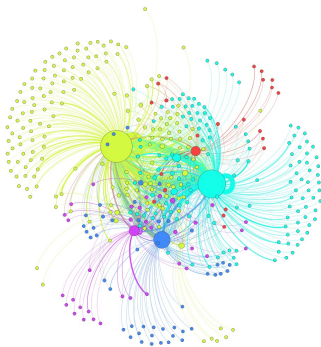
Power of DFS:

Graph Traversal  $\implies$  Graph Decomposition

Power of DFS:


Graph Traversal  $\implies$  Graph Decomposition

*Structure! Structure! Structure!*



Graph *structure* induced by DFS:

states of 

types of 



## Graph *structure* induced by DFS:

states of  $v$

types of  $u$  —  $v$

life time of  $v$ :

$v : d[v], f[v]$

$d[v]$ : BICOMP

$f[v]$ : TOPOSORT, SCC

## Definition (Classifying edges)

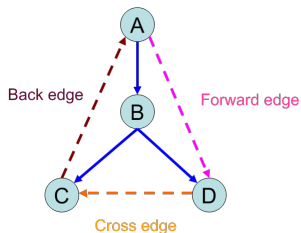
Given a DFS traversal  $\implies$  DFS tree:

Tree edge:  $\rightarrow$  child

Back edge:  $\rightarrow$  ancestor

Forward edge:  $\rightarrow$  *nonchild* descendant

Cross edge:  $\rightarrow (\neg \text{ancestor}) \wedge (\neg \text{descendant})$



## Definition (Classifying edges)

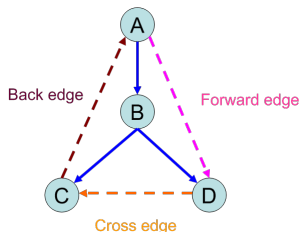
Given a DFS traversal  $\implies$  DFS tree:

Tree edge:  $\rightarrow$  child

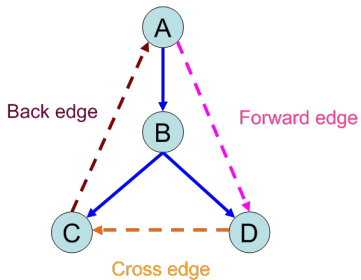
Back edge:  $\rightarrow$  ancestor

Forward edge:  $\rightarrow$  *nonchild* descendant

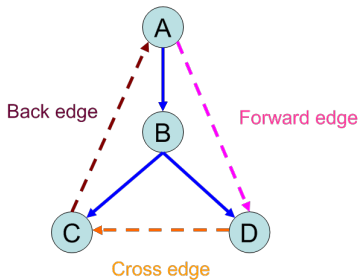
Cross edge:  $\rightarrow (\neg \text{ancestor}) \wedge (\neg \text{descendant})$



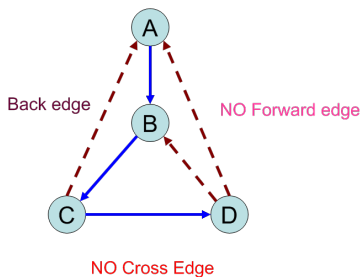
- ▶ Also applicable to BFS
- ▶ w.r.t. DFS/BFS trees



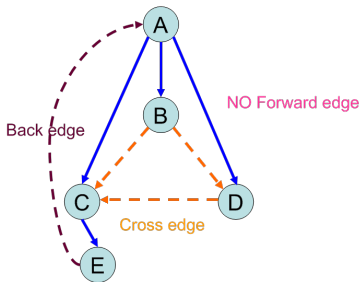
DFS on directed graph



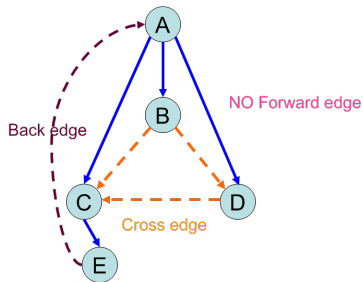
DFS on directed graph



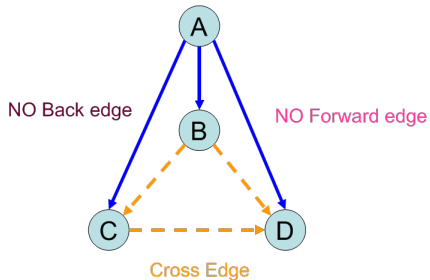
DFS on undirected graph



BFS on directed graph



BFS on directed graph



BFS on undirected graph (Problem 5.1)

## DFS tree and BFS tree coincide (Problem 5.7)

Undirected connected graph  $G = (V, E), v \in V$

DFS tree  $T$  from  $v \equiv$  BFS tree  $T'$  from  $v$



## DFS tree and BFS tree coincide (Problem 5.7)

Undirected connected graph  $G = (V, E), v \in V$

DFS tree  $T$  from  $v \equiv$  BFS tree  $T'$  from  $v$

$$G \equiv T$$

## DFS tree and BFS tree coincide (Problem 5.7)

Undirected connected graph  $G = (V, E), v \in V$

DFS tree  $T$  from  $v \equiv$  BFS tree  $T'$  from  $v$

$$G \equiv T$$

Proof.

$G_{\text{DFS}}$ : tree + back vs.  $G_{\text{BFS}}$ : tree + cross



## DFS tree and BFS tree coincide (Problem 5.7)

Undirected connected graph  $G = (V, E), v \in V$

DFS tree  $T$  from  $v \equiv$  BFS tree  $T'$  from  $v$

$$G \equiv T$$

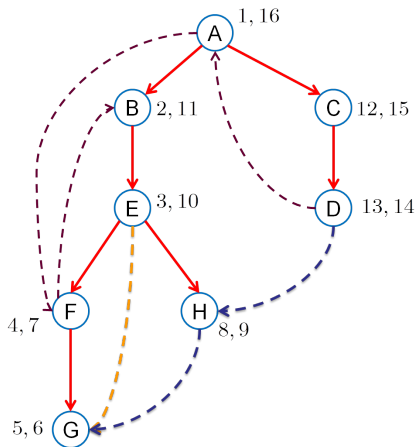
Proof.

$G_{\text{DFS}}$ : tree + back vs.  $G_{\text{BFS}}$ : tree + cross



$Q$  : What if  $G$  is a digraph?

## Lift time of vertices in DFS



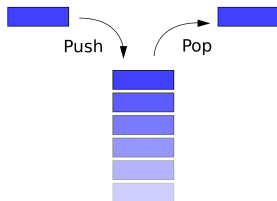
## Theorem (Disjoint or Contained (Problem 4.2 : (1)&(2)))

$$\forall u, v : [u]_u \cap [v]_v = \emptyset \vee ([u]_u \subset [v]_v \vee [v]_v \subset [u]_u)$$

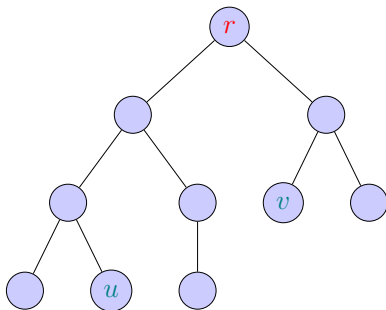
## Theorem (Disjoint or Contained (Problem 4.2 : (1)&(2)))

$$\forall u, v : [u]_u \cap [v]_v = \emptyset \vee ([u]_u \subset [v]_v \vee [v]_v \subset [u]_u)$$

Proof.

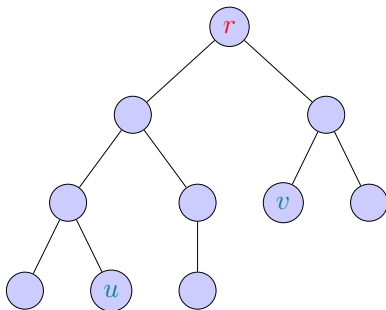


## Preprocessing for ancestor/descendant relation (Problem 5.6)



*Q : Is  $u$  an ancestor of  $v$ ?  $O(1)$*

## Preprocessing for ancestor/descendant relation (Problem 5.6)

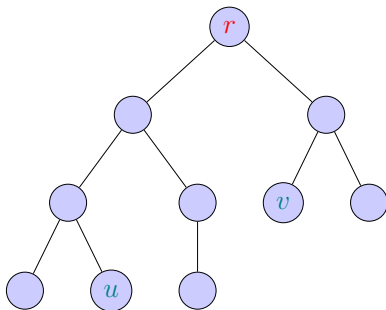


*$Q$  : Is  $u$  an ancestor of  $v$ ?  $O(1)$*

$v : d[v], f[v]$



## Preprocessing for ancestor/descendant relation (Problem 5.6)



*Q : Is  $u$  an ancestor of  $v$ ?  $O(1)$*

$v : d[v], f[v]$

*Q : # of descendants of any  $v$ ?*

## Edge types and life time of vertices in DFS (Problem 4.5)

$$\forall u \rightarrow v :$$

- ▶ tree/forward edge:  $[u \text{ (red)} [v \text{ (blue)} ]v ]u \text{ (red)}$
- ▶ back edge:  $[v [u \text{ (red)} ]u ]v$
- ▶ cross edge:  $[v ]v [u \text{ (red)} ]u$

## Edge types and life time of vertices in DFS (Problem 4.5)

$$\forall u \rightarrow v :$$

- ▶ tree/forward edge:  $[u \text{ (red)} [v \text{ (blue)} ]v ]u \text{ (red)}$
- ▶ back edge:  $[v [u \text{ (red)} ]u ]v$
- ▶ cross edge:  $[v ]v [u \text{ (red)} ]u$

$$f[v] < d[u] \iff \text{edge}$$

## Edge types and life time of vertices in DFS (Problem 4.5)

$$\forall u \rightarrow v :$$

- ▶ tree/forward edge:  $[u \text{ (red)} [v \text{ (blue)} ]v ]u \text{ (red)}$
- ▶ back edge:  $[v [u \text{ (red)} ]u ]v$
- ▶ cross edge:  $[v ]v [u \text{ (red)} ]u$

$$f[v] < d[u] \iff \text{cross edge}$$

## Edge types and life time of vertices in DFS (Problem 4.5)

$$\forall u \rightarrow v :$$

- ▶ tree/forward edge:  $[u \text{ (red)} [v \text{ (blue)} ]v ]u \text{ (red)}$
- ▶ back edge:  $[v [u \text{ (red)} ]u ]v$
- ▶ cross edge:  $[v ]v [u \text{ (red)} ]u$

$$f[v] < d[u] \iff \text{cross edge}$$

$$f[u] < f[v] \iff$$

## Edge types and life time of vertices in DFS (Problem 4.5)

$$\forall u \rightarrow v :$$

- ▶ tree/forward edge:  $[u \text{ (red)} [v \text{ (blue)} ]v \text{ (blue)} ]u \text{ (red)}$
- ▶ back edge:  $[v [u \text{ (red)} ]u \text{ (red)} ]v$
- ▶ cross edge:  $[v ]v [u \text{ (red)} ]u \text{ (red)}$

$$f[v] < d[u] \iff \text{cross edge}$$

$$f[u] < f[v] \iff \text{back edge}$$

## Edge types and life time of vertices in DFS (Problem 4.5)

$$\forall u \rightarrow v :$$

- ▶ tree/forward edge:  $[u \text{ (red)} [v \text{ (blue)} ]v ]u$
- ▶ back edge:  $[v [u \text{ (red)} ]u ]v$
- ▶ cross edge:  $[v ]v [u \text{ (red)} ]u$

$$f[v] < d[u] \iff \text{cross edge}$$

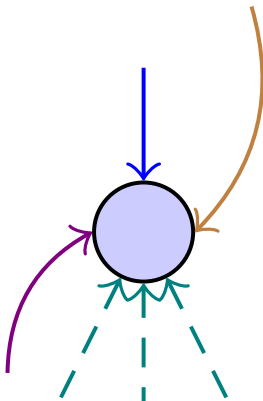
$$f[u] < f[v] \iff \text{back edge}$$

$$\nexists \text{ cycle} \implies \boxed{u \rightarrow v \iff f[v] < f[u]}$$

DFS from the perspective of a single node:



DFS from the perspective of a single node:



## Height and diameter of tree (Problem 5.4)

Binary tree  $T = (V, E)$  with  $|V| = n$  and the root  $r$ :

- (I) Height  $H(T)$  in  $O(n)$
- (II) Diameter  $D(T)$  in  $O(n)$

## Height and diameter of tree (Problem 5.4)

Binary tree  $T = (V, E)$  with  $|V| = n$  and the root  $r$ :

- (I) Height  $H(T)$  in  $O(n)$
- (II) Diameter  $D(T)$  in  $O(n)$

$$\left\{ \begin{array}{l} H(T) = \max(H(L_T), H(R_T)) + 1, \end{array} \right.$$

## Height and diameter of tree (Problem 5.4)

Binary tree  $T = (V, E)$  with  $|V| = n$  and the root  $r$ :

- (I) Height  $H(T)$  in  $O(n)$
- (II) Diameter  $D(T)$  in  $O(n)$

$$\begin{cases} H(T) = 0, & T \text{ is a leaf} \\ H(T) = \max(H(L_T), H(R_T)) + 1, & \text{o.w.} \end{cases}$$

## Height and diameter of tree (Problem 5.4)

Binary tree  $T = (V, E)$  with  $|V| = n$  and the root  $r$ :

- (I) Height  $H(T)$  in  $O(n)$
- (II) Diameter  $D(T)$  in  $O(n)$

$$\begin{cases} H(T) = 0, & T \text{ is a leaf} \\ H(T) = \max(H(L_T), H(R_T)) + 1, & \text{o.w.} \end{cases}$$

$$\begin{cases} D(T) = 0, & T \text{ is a leaf} \\ D(T) = \max(D(L_T), D(R_T), \end{cases} \quad \text{o.w.}$$

## Height and diameter of tree (Problem 5.4)

Binary tree  $T = (V, E)$  with  $|V| = n$  and the root  $r$ :

- (I) Height  $H(T)$  in  $O(n)$
- (II) Diameter  $D(T)$  in  $O(n)$

$$\begin{cases} H(T) = 0, & T \text{ is a leaf} \\ H(T) = \max(H(L_T), H(R_T)) + 1, & \text{o.w.} \end{cases}$$

$$\begin{cases} D(T) = 0, & T \text{ is a leaf} \\ D(T) = \max(D(L_T), D(R_T), \underbrace{H(L_T) + H(R_T) + 2}_{\text{through the root}}), & \text{o.w.} \end{cases}$$

Binary tree  $T = (V, E)$  with  $|V| = n$  and the root  $r$

Binary tree  $T = (V, E)$  with  $|V| = n$  and the root  $r$

$Q$  : Diameter of a *tree without* a designated root



Binary tree  $T = (V, E)$  with  $|V| = n$  and the root  $r$

$Q$  : Diameter of a *tree without* a designated root



$Q$  : Diameter of a tree *without* a designated root

$Q$  : Diameter of a tree *without* a designated root

A beautiful algorithm:

- ▶ Pick any  $u$
- ▶ Run BFS from  $u$ , obtain the farthest  $v$
- ▶ Run BFS from  $v$ , obtain the farthest  $w$   
 $(v, w)$

$Q$  : Diameter of a tree *without* a designated root

A beautiful algorithm:

- ▶ Pick any  $u$
- ▶ Run BFS from  $u$ , obtain the farthest  $v$
- ▶ Run BFS from  $v$ , obtain the farthest  $w$   
 $(v, w)$

Your Job: Prove it!

$Q$  : Diameter of a tree *without* a designated root

A beautiful algorithm:

- ▶ Pick any  $u$
- ▶ Run BFS from  $u$ , obtain the farthest  $v$
- ▶ Run BFS from  $v$ , obtain the farthest  $w$   
 $(v, w)$

Your Job: Prove it!

Back to our original problem with  $u \leftarrow r$ .

## Cycle detection (Problem 5.8 – 1)

	Digraph	Undirected graph
DFS		
BFS		

## Cycle detection (Problem 5.8 – 1)

	Digraph	Undirected graph
DFS	back edge $\iff$ cycle	
BFS		

## Cycle detection (Problem 5.8 – 1)

	Digraph	Undirected graph
DFS	back edge $\iff$ cycle	back edge $\iff$ cycle
BFS		



## Cycle detection (Problem 5.8 – 1)

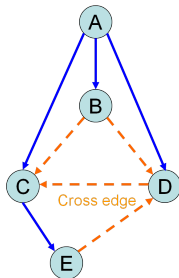
	Digraph	Undirected graph
DFS	back edge $\iff$ cycle	back edge $\iff$ cycle
BFS		cross edge $\iff$ cycle

## Cycle detection (Problem 5.8 – 1)

	Digraph	Undirected graph
DFS	back edge $\iff$ cycle	back edge $\iff$ cycle
BFS	back edge $\implies$ cycle cycle $\not\Rightarrow$ back edge	cross edge $\iff$ cycle

## Cycle detection (Problem 5.8 – 1)

	Digraph	Undirected graph
DFS	back edge $\iff$ cycle	back edge $\iff$ cycle
BFS	back edge $\implies$ cycle cycle $\not\Rightarrow$ back edge	cross edge $\iff$ cycle



## Evasiveness of acyclicity of **undirected** graphs (Problem 5.8 – 2)

Evasiveness  $\triangleq$  check  $\binom{n}{2}$  edges (adjacency matrix)

## Evasiveness of acyclicity of **undirected** graphs (Problem 5.8 – 2)

Evasiveness  $\triangleq$  check  $\binom{n}{2}$  edges (adjacency matrix)

$Q$  : Is **acyclicity** evasive?

## Evasiveness of acyclicity of **undirected** graphs (Problem 5.8 – 2)

Evasiveness  $\triangleq$  check  $\binom{n}{2}$  edges (adjacency matrix)

$Q$  : Is **acyclicity** evasive?

By Adversary Argument.



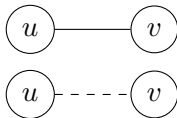
## Evasiveness of acyclicity of **undirected** graphs (Problem 5.8 – 2)

Evasiveness  $\triangleq$  check  $\binom{n}{2}$  edges (adjacency matrix)

$Q$  : Is **acyclicity** evasive?

By Adversary Argument.

Adversary  $\mathcal{A}$ :



Algorithm  $\mathcal{A}$ :

CHECKEDGE( $u, v$ )

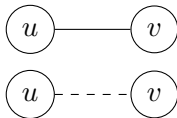
## Evasiveness of acyclicity of **undirected** graphs (Problem 5.8 – 2)

Evasiveness  $\triangleq$  check  $\binom{n}{2}$  edges (adjacency matrix)

$Q$  : Is **acyclicity** evasive?

By Adversary Argument.

Adversary  $\mathcal{A}$ :



Algorithm  $\mathcal{A}$ :

CHECKEDGE( $u, v$ )

Hint: Kruskal





$$\text{CHECKEDGE}(u, v) \leftarrow \begin{array}{c} u \text{ --- } v \end{array}$$

$$\iff$$

$$\# \text{ cycle} \in G + \begin{array}{c} u \text{ --- } v \end{array}$$



$$\begin{aligned} \text{CHECKEDGE}(u, v) \leftarrow & \text{---} \begin{array}{c} \text{---} u \text{---} v \text{---} \end{array} \\ & \iff \\ & \# \text{ cycle} \in G + \begin{array}{c} \text{---} u \text{---} v \text{---} \end{array} \end{aligned}$$

*Q* : Why adjacency matrix?

## After-class Exercise: Evasiveness of connectivity of undirected graphs

Evasiveness  $\triangleq$  check  $\binom{n}{2}$  edges (adjacency matrix)

$Q$  : Is connectivity evasive?

## After-class Exercise: Evasiveness of connectivity of undirected graphs

Evasiveness  $\triangleq$  check  $\binom{n}{2}$  edges (adjacency matrix)

$Q$  : Is connectivity evasive?



Hint: Anti-Kruskal

## Orientation of undirected graph (Problem 4.13)

- ▶ undirected (connected) graph  $G$
- ▶ edges oriented *s.t.*

$$\forall v, \text{in}[v] \geq 1$$

## Orientation of undirected graph (Problem 4.13)

- ▶ undirected (connected) graph  $G$
- ▶ edges oriented s.t.

$$\forall v, \text{in}[v] \geq 1$$

orientation  $\iff \exists$  cycle  $C$

## Orientation of undirected graph (Problem 4.13)

- ▶ undirected (connected) graph  $G$
- ▶ edges oriented s.t.

$$\forall v, \text{in}[v] \geq 1$$

orientation  $\iff \exists$  cycle  $C$

DFS from  $v \in C$

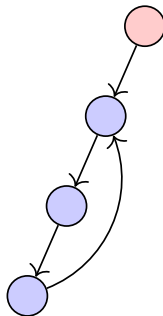
## Orientation of undirected graph (Problem 4.13)

- ▶ undirected (connected) graph  $G$
- ▶ edges oriented s.t.

$$\forall v, \text{in}[v] \geq 1$$

orientation  $\iff \exists$  cycle  $C$

DFS from  $v \in C$





## Shortest cycle of undirected graph (Problem 4.12)

A **WRONG** DFS-based algorithm:

$$\forall v : \text{level}[v]$$

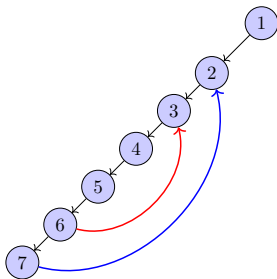
Back edge  $u \rightarrow v : \text{level}[u] - \text{level}[v] + 1$

## Shortest cycle of undirected graph (Problem 4.12)

A **WRONG** DFS-based algorithm:

$$\forall v : \text{level}[v]$$

Back edge  $u \rightarrow v : \text{level}[u] - \text{level}[v] + 1$



## Shortest cycle of digraph (Problem 4.12)

A DFS-based algorithm:

$$\forall v : \text{level}[v]$$

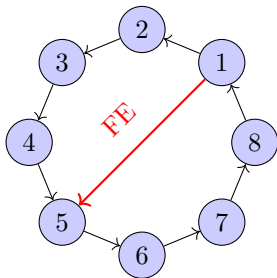
$$\text{Back edge } u \rightarrow v : \text{level}[u] - \text{level}[v] + 1$$

## Shortest cycle of digraph (Problem 4.12)

A **WRONG** DFS-based algorithm:

$$\forall v : \text{level}[v]$$

Back edge  $u \rightarrow v : \text{level}[u] - \text{level}[v] + 1$



On digraphs:

$\nexists$  back edge  $\iff$  DAG

On digraphs:

$\nexists$  back edge  $\iff$  DAG  $\iff \exists$  topo. ordering

On digraphs:

$\nexists$  back edge  $\iff$  DAG  $\iff \exists$  topo. ordering

TOPOSORT by Tarjan (probably), 1976

$\nexists$  cycle  $\implies$   $u \rightarrow v \iff f[v] < f[u]$

On digraphs:

$\nexists$  back edge  $\iff$  DAG  $\iff \exists$  topo. ordering

TOPOSORT by Tarjan (probably), 1976

$\nexists$  cycle  $\implies \boxed{u \rightarrow v \iff f[v] < f[u]}$

Sort vertices in *decreasing* order of their *finish* times.



## Kahn's TOPOSORT algorithm (1962; Problem 4.16)

- ▶ Queue  $Q$  for source vertices ( $\text{in}[v] = 0$ )
- ▶ Repeat: DEQUEUE( $u \in Q$ ), delete  $u$  and  $u \rightarrow v$  from  $Q$ ,  
output  $u$ , ENQUEUE( $v$ ) if  $\text{in}[v] = 0$

## Kahn's TOPOSORT algorithm (1962; Problem 4.16)

- ▶ Queue  $Q$  for source vertices ( $\text{in}[v] = 0$ )
- ▶ Repeat: DEQUEUE( $u \in Q$ ), delete  $u$  and  $u \rightarrow v$  from  $Q$ ,  
output  $u$ , ENQUEUE( $v$ ) if  $\text{in}[v] = 0$

## Lemma (Correctness of Kahn's TOPOSORT)

*Every DAG has at least one source (and at least one sink vertex).*

## Kahn's TOPOSORT algorithm (1962; Problem 4.16)

- ▶ Queue  $Q$  for source vertices ( $\text{in}[v] = 0$ )
- ▶ Repeat: DEQUEUE( $u \in Q$ ), delete  $u$  and  $u \rightarrow v$  from  $Q$ , output  $u$ , ENQUEUE( $v$ ) if  $\text{in}[v] = 0$

## Lemma (Correctness of Kahn's TOPOSORT)

*Every DAG has at least one source (and at least one sink vertex).*

$Q$  : What if  $G$  is *not* a DAG?

## Taking courses in few semesters (Problem 5.14)

- ▶  $n$  courses
- ▶  $m$  of  $c_1 \rightarrow c_2$ : prerequisite
- ▶ Goal: taking courses in few semesters

## Taking courses in few semesters (Problem 5.14)

- ▶  $n$  courses
- ▶  $m$  of  $c_1 \rightarrow c_2$ : prerequisite
- ▶ Goal: taking courses in few semesters

Critical path *OR* Longest path using DFS in  $O(n + m)$

## Taking courses in few semesters (Problem 5.14)

- ▶  $n$  courses
- ▶  $m$  of  $c_1 \rightarrow c_2$ : prerequisite
- ▶ Goal: taking courses in few semesters

Critical path *OR* Longest path using DFS in  $O(n + m)$

For general digraph, **LONGEST-PATH** is NP-hard.

## Line up (Problem 4.22)

1.  $i$  hates  $j$ :  $i \succ j$
2.  $i$  hates  $j$ :  $\#i < \#j$

TOPOSORT

Critical path

## Hamiltonian path in DAG (Problem 4.14)

HP: path visiting each vertex once

$Q : \exists \text{ HP in a DAG in } O(n + m)$



## Hamiltonian path in DAG (Problem 4.14)

HP: path visiting each vertex once

$Q : \exists \text{ HP in a DAG in } O(n + m)$

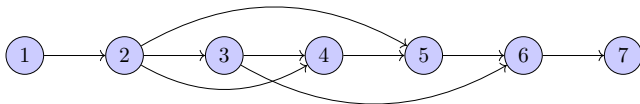
For general (di)graph, HP is NP-hard.

## Hamiltonian path in DAG (Problem 4.14)

HP: path visiting each vertex once

$Q : \exists \text{ HP in a DAG in } O(n + m)$

For general (di)graph, HP is NP-hard.

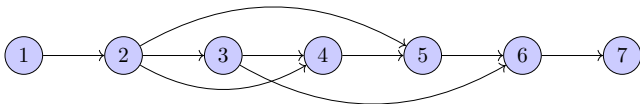


## Hamiltonian path in DAG (Problem 4.14)

HP: path visiting each vertex once

$Q : \exists \text{ HP in a DAG in } O(n + m)$

For general (di)graph, HP is NP-hard.



DAG:  $\exists \text{ HP} \iff \exists! \text{ topo. ordering}$

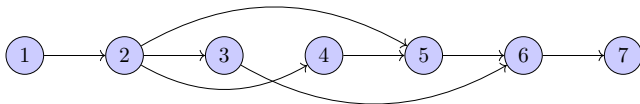
DAG:  $\exists$  HP  $\iff$   $\exists!$  topo. ordering

DAG:  $\exists$  HP  $\iff \exists!$  topo. ordering

Tarjan's TOPOSORT + Check edges  $(v_i, v_{i+1})$

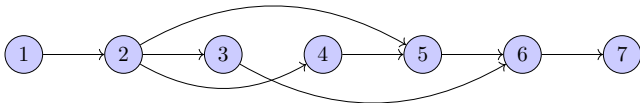
DAG:  $\exists$  HP  $\iff \exists!$  topo. ordering

Tarjan's TOPOSORT + Check edges  $(v_i, v_{i+1})$



DAG:  $\exists$  HP  $\iff \exists!$  topo. ordering

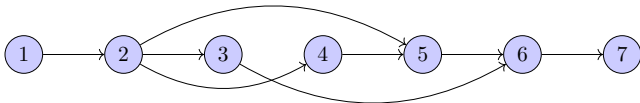
Tarjan's TOPOSORT + Check edges  $(v_i, v_{i+1})$



Kahn's TOPOSORT (Problem 4.16)

DAG:  $\exists$  HP  $\iff \exists!$  topo. ordering

Tarjan's TOPOSORT + Check edges  $(v_i, v_{i+1})$



Kahn's TOPOSORT (Problem 4.16)

$$|Q| \leq 1$$



## Theorem (Digraph as DAG (Problem 4.6))

*Every digraph is a dag of its SCCs.*

## Theorem (Digraph as DAG (Problem 4.6))

*Every digraph is a dag of its SCCs.*

Two tiered structure of digraphs:

digraph  $\equiv$  a dag of SCCs

SCC: equivalence class over reachability

digraph  $\equiv$  a dag of SCCs

Kosaraju SCC algorithm, 1978

*“SCCs can be topo-sorted  
in **decreasing** order of their highest **finish** time.”*

digraph  $\equiv$  a dag of SCCs

Kosaraju SCC algorithm, 1978

*“SCCs can be topo-sorted  
in **decreasing** order of their highest **finish** time.”*

- (I) DFS on  $G$ ; DFS/BFS on  $G^T$
- (II) DFS on  $G^T$ ; DFS/BFS on  $G$

## Kosaraju SCC algorithm, 1978 (Problem 4.7)

1st DFS  $\xRightarrow{?}$  BFS

2nd DFS  $\xRightarrow{?}$  BFS

## Kosaraju SCC algorithm, 1978 (Problem 4.7)

1st DFS  $\xRightarrow{?}$  BFS

2nd DFS  $\xRightarrow{?}$  BFS

1st DFS: **toposort** between SCCs

2nd DFS: **reachability** within an SCC

## Kosaraju SCC algorithm, 1978 (Problem 4.7)

1st DFS  $\xRightarrow{?}$  BFS

2nd DFS  $\xRightarrow{?}$  BFS

1st DFS: **toposort** between SCCs

2nd DFS: **reachability** within an SCC

digraph  $\equiv$  a **dag** of SCCs

## One-to-all reachability in a digraph (Problem 5.12)

$$v : v \rightsquigarrow^? \forall u$$

$$\exists? v : v \rightsquigarrow \forall u$$



## One-to-all reachability in a digraph (Problem 5.12)

$$v : v \rightsquigarrow^? \forall u$$

$$\exists? v : v \rightsquigarrow \forall u$$

SCC

$$\exists! \text{ source vertex } v \iff v \rightsquigarrow \forall u$$

## One-to-all reachability in a digraph (Problem 5.12)

$$v : v \rightsquigarrow^? \forall u$$

$$\exists? v : v \rightsquigarrow \forall u$$

SCC

$$\exists! \text{ source vertex } v \iff v \rightsquigarrow \forall u$$

$$\iff : \exists! \text{ source}$$

## One-to-all reachability in a digraph (Problem 5.12)

$$v : v \rightsquigarrow^? \forall u$$

$$\exists? v : v \rightsquigarrow \forall u$$

SCC

$$\exists! \text{ source vertex } v \iff v \rightsquigarrow \forall u$$

$$\iff : \exists! \text{ source}$$

$$\implies : \text{By contradiction.}$$

$$\exists u : v \not\rightsquigarrow u \wedge \text{in}[u] > 0 \implies \exists \text{ cycle}$$

## Impacts of vertices in a digraph (Problem 4.18)

$$\text{impact}(v) = |\{w \neq v : v \rightsquigarrow w\}|$$

- ▶  $\arg \min_v \text{impact}(v)$
- ▶  $\arg \max_v \text{impact}(v)$

## Impacts of vertices in a digraph (Problem 4.18)

$$\text{impact}(v) = |\{w \neq v : v \rightsquigarrow w\}|$$

- ▶  $\arg \min_v \text{impact}(v)$
- ▶  $\arg \max_v \text{impact}(v)$

$\arg \min_v \text{impact}(v) \in \text{sink SCC of smallest cardinality}$

## Impacts of vertices in a digraph (Problem 4.18)

$$\text{impact}(v) = |\{w \neq v : v \rightsquigarrow w\}|$$

- ▶  $\arg \min_v \text{impact}(v)$
- ▶  $\arg \max_v \text{impact}(v)$

$\arg \min_v \text{impact}(v) \in \text{sink SCC of smallest cardinality}$

$\arg \min_v \text{impact}(v) \in \text{source SCC}$

## Impacts of vertices in a digraph (Problem 4.18)

$$\text{impact}(v) = |\{w \neq v : v \rightsquigarrow w\}|$$

- ▶  $\arg \min_v \text{impact}(v)$
- ▶  $\arg \max_v \text{impact}(v)$

$\arg \min_v \text{impact}(v) \in \text{sink SCC of smallest cardinality}$

$\arg \min_v \text{impact}(v) \in \text{source SCC}$

$Q : \forall v : \text{computing } \text{impact}(v)$

## 2SAT (Problem 4.23)

$$I : (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_1 \vee x_2) \wedge (\overline{x_3} \vee x_4) \wedge (\overline{x_1} \vee x_4)$$



## 2SAT (Problem 4.23)

$$I : (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_1 \vee x_2) \wedge (\overline{x_3} \vee x_4) \wedge (\overline{x_1} \vee x_4)$$

$$\alpha \vee \beta \equiv \overline{\alpha} \rightarrow \beta \equiv \overline{\beta} \rightarrow \alpha$$

## 2SAT (Problem 4.23)

$$I : (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_1 \vee x_2) \wedge (\overline{x_3} \vee x_4) \wedge (\overline{x_1} \vee x_4)$$

$$\alpha \vee \beta \equiv \overline{\alpha} \rightarrow \beta \equiv \overline{\beta} \rightarrow \alpha$$

Implication graph  $G_I$ .

## 2SAT (Problem 4.23)

$$I : (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_1 \vee x_2) \wedge (\overline{x_3} \vee x_4) \wedge (\overline{x_1} \vee x_4)$$

$$\alpha \vee \beta \equiv \overline{\alpha} \rightarrow \beta \equiv \overline{\beta} \rightarrow \alpha$$

Implication graph  $G_I$ .

## Theorem (2SAT)

$$\exists \text{ SCC } \exists x : v_x \in \text{SCC} \wedge v_{\overline{x}} \in \text{SCC} \iff I \text{ is not satisfiable.}$$

## 2SAT (Problem 4.23)

$$I : (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_1 \vee x_2) \wedge (\overline{x_3} \vee x_4) \wedge (\overline{x_1} \vee x_4)$$

$$\alpha \vee \beta \equiv \overline{\alpha} \rightarrow \beta \equiv \overline{\beta} \rightarrow \alpha$$

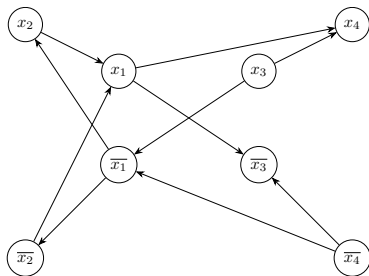
Implication graph  $G_I$ .

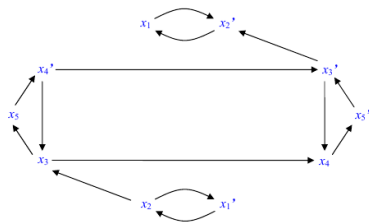
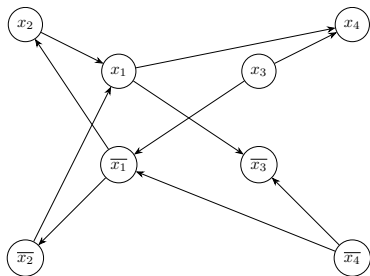
## Theorem (2SAT)

$$\exists \text{ SCC } \exists x : v_x \in \text{SCC} \wedge v_{\overline{x}} \in \text{SCC} \iff I \text{ is not satisfiable.}$$

## Reference:

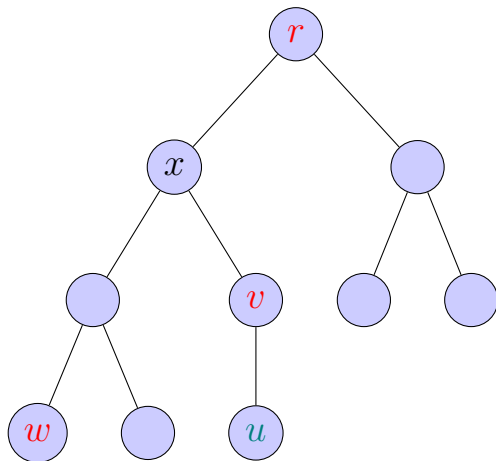
- ▶ “A Linear-time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas” by Bengt Aspvall, Michael Plass, and Robert Tarjan, 1979.





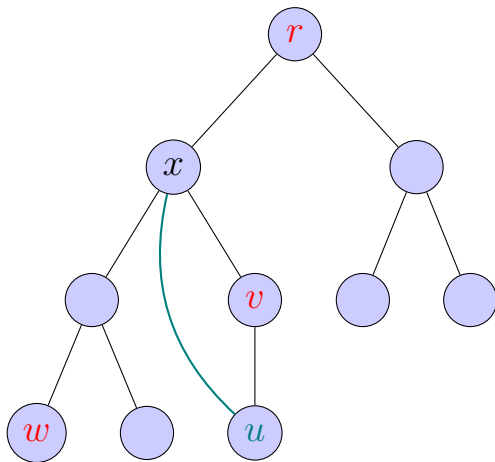
# BICOMP: Back!

# BICOMP: Back!

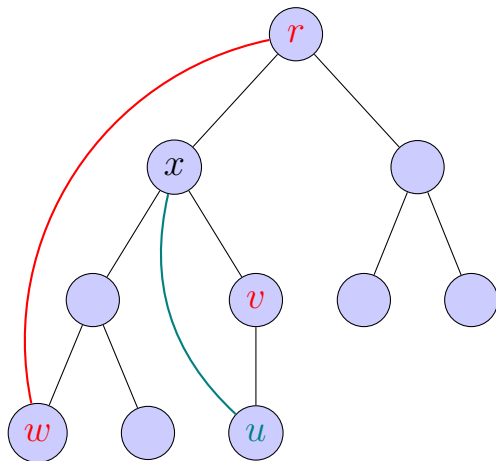




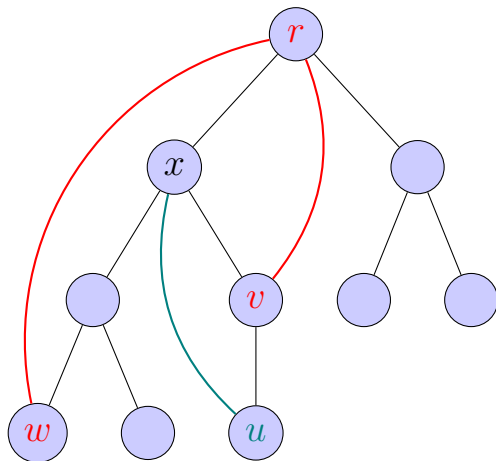
# BICOMP: Back!



# BICOMP: Back!

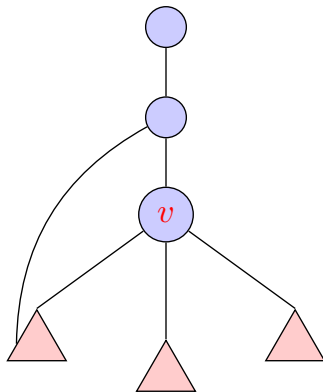


# BICOMP: Back!



- (I) When and how to **update**  $\text{back}[v]$ ?
- (II) When and how to **identify** a bicomponent?

- (I) When and how to **update**  $\text{back}[v]$ ?
- (II) When and how to **identify** a bicomponent?



## Initialization of $\text{back}[v]$ (Problem 4.9)

$$\text{back}[v] = d[v] \text{ vs. } \text{back}[v] = \infty \vee 2(n+1)$$

## Initialization of $\text{back}[v]$ (Problem 4.9)

$$\text{back}[v] = d[v] \text{ vs. } \text{back}[v] = \infty \vee 2(n+1)$$

$\text{back}[v]$ : the earliest reachable ancestor of  $v$

## Initialization of $\text{back}[v]$ (Problem 4.9)

$$\text{back}[v] = d[v] \text{ vs. } \text{back}[v] = \infty \vee 2(n+1)$$

$\text{back}[v]$ : the earliest reachable ancestor of  $v$

tree edge ( $\rightarrow v$ ):  $\text{back}[v] = d[v]$

back edge ( $v \rightarrow w$ ):  $\text{back}[v] = \min\{\text{back}[v], d[w]\}$

backtracking from  $w$ :  $\text{back}[v] = \min\{\text{back}[v], \text{back}[w] = w\text{Back}\}$



## Initialization of $\text{back}[v]$ (Problem 4.9)

$$\text{back}[v] = d[v] \text{ vs. } \text{back}[v] = \infty \vee 2(n+1)$$

$\text{back}[v]$ : the earliest reachable ancestor of  $v$

tree edge ( $\rightarrow v$ ):  $\text{back}[v] = d[v]$

back edge ( $v \rightarrow w$ ):  $\text{back}[v] = \min\{\text{back}[v], d[w]\}$

backtracking from  $w$ :  $\text{back}[v] = \min\{\text{back}[v], \text{back}[w] = w\text{Back}\}$

Proof.

if ever updated

## Initialization of $\text{back}[v]$ (Problem 4.9)

$$\text{back}[v] = d[v] \text{ vs. } \text{back}[v] = \infty \vee 2(n+1)$$

$\text{back}[v]$ : the earliest reachable ancestor of  $v$

tree edge ( $\rightarrow v$ ):  $\text{back}[v] = d[v]$

back edge ( $v \rightarrow w$ ):  $\text{back}[v] = \min\{\text{back}[v], d[w]\}$

backtracking from  $w$ :  $\text{back}[v] = \min\{\text{back}[v], \text{back}[w] = w\text{Back}\}$

Proof.

if never updated:

if ever updated

## Initialization of $\text{back}[v]$ (Problem 4.9)

$$\text{back}[v] = d[v] \text{ vs. } \text{back}[v] = \infty \vee 2(n+1)$$

$\text{back}[v]$ : the earliest reachable ancestor of  $v$

tree edge ( $\rightarrow v$ ):  $\text{back}[v] = d[v]$

back edge ( $v \rightarrow w$ ):  $\text{back}[v] = \min\{\text{back}[v], d[w]\}$

backtracking from  $w$ :  $\text{back}[v] = \min\{\text{back}[v], \text{back}[w] = w\text{Back}\}$

Proof.

if never updated:

if ever updated

$$w\text{Back} = \infty > d[v] \text{ vs.}$$

## Initialization of $\text{back}[v]$ (Problem 4.9)

$$\text{back}[v] = d[v] \text{ vs. } \text{back}[v] = \infty \vee 2(n+1)$$

$\text{back}[v]$ : the earliest reachable ancestor of  $v$

tree edge ( $\rightarrow v$ ):  $\text{back}[v] = d[v]$

back edge ( $v \rightarrow w$ ):  $\text{back}[v] = \min\{\text{back}[v], d[w]\}$

backtracking from  $w$ :  $\text{back}[v] = \min\{\text{back}[v], \text{back}[w] = \text{wBack}\}$

Proof.

if never updated:

if ever updated

$$\text{wBack} = \infty > d[v] \text{ vs. } \text{wBack} = d[w] > d[v]$$



## Root cutnode $v$ (Problem 4.8)

$$v \text{ is a cutnode} \iff \text{out}[v] \geq 2$$

## Root cutnode $v$ (Problem 4.8)

$v$  is a cutnode  $\iff \text{out}[v] \geq 2$

