3-6 Decompositions of Graphs

(DFS, DAG, Toposort, Cycle, SCC)

Hengfeng Wei

hfwei@nju.edu.cn

October 29, 2018

Robert Tarjan

John Hopcroft

"For fundamental achievements
in the design and analysis of algorithms and data structures."

— Turing Award, 1986

SIAM J. COMPUT.
Vol. 1, No. 2, June 1972

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", Robert Tarjan

SIAM J. COMPUT.
Vol. 1, No. 2, June 1972

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", Robert Tarjan

"DFS is a powerful technique with many applications."

The Hammer of DFS

Power of DFS:

Graph Traversal $\Longrightarrow$ Graph Decomposition

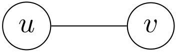
Power of DFS:

Graph Traversal $\Longrightarrow$ Graph Decomposition

Structure! Structure! Structure!

Graph *structure* induced by DFS:

states of 

types of 

Graph *structure* induced by DFS:

life time of v:

states of v

v : $\mathrm{d}[v], \mathrm{f}[v]$

types of u — v

$\mathrm{d}[v]$: Bicomp

$\mathrm{f}[v]$: Toposort, SCC

Definition (Classification of Edges)

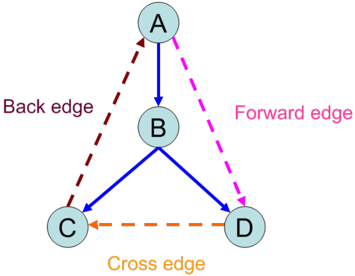
We can define four edge edges in terms of the depth-first forest G_π produced by a DFS on G:

Tree edge: edge in G_π

Back edge: $\rightarrow$ ancestor

Forward edge: $\rightarrow$ descendant (nontree edge)

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



DFS on Undirected Graphs (Problem 22.3-6)

Classifying an edge (u, v) as a tree edge or a back edge according to whether (u, v) or (v, u) is encountered first during the depth-first search

is equivalent to

classifying it according to the ordering of the four types in the classification scheme.

DFS on Undirected Graphs (Problem 22.3-6)

Classifying an edge (u, v) as a tree edge or a back edge according to whether (u, v) or (v, u) is encountered first during the depth-first search

is equivalent to

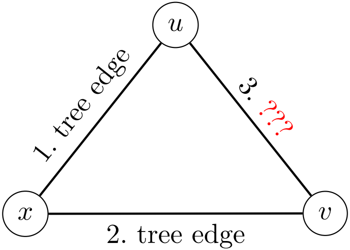
classifying it according to the ordering of the four types in the classification scheme.

DFS on Undirected Graphs (Problem 22.3-6)

Classifying an edge (u, v) as a tree edge or a back edge according to whether (u, v) or (v, u) is encountered first during the depth-first search

is equivalent to

classifying it according to the ordering of the four types in the classification scheme.



Theorem (Theorem 22.10)

In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge.

Proof.

Let (u, v) be an arbitrary edge of G, and suppose without loss of generality that $u.d < v.d$. Then the search must discover and finish v before it finishes u (while u is gray), since v is on u's adjacency list.

If the first time that the search explores edge (u, v), it is in the direction from u to v, then v is undiscovered (white) until that time, for otherwise the search would have explored this edge already in the direction from v to u. Thus, (u, v) becomes a tree edge.

If the search explores (u, v) first in the direction from v to u, then (u, v) is a back edge, since u is still gray at the time the edge is first explored. $\square$

Theorem (Theorem 22.10)

In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge.

Proof.

Let (u, v) be an arbitrary edge of G, and suppose without loss of generality that $u.d < v.d$. Then the search must discover and finish v before it finishes u (while u is gray), since v is on u's adjacency list.

If the first time that the search explores edge (u, v), it is in the direction from u to v, then v is undiscovered (white) until that time, for otherwise the search would have explored this edge already in the direction from v to u. Thus, (u, v) becomes a tree edge.

If the search explores (u, v) first in the direction from v to u, then (u, v) is a back edge, since u is still gray at the time the edge is first explored. $\square$


KEEP
CALM
AND
PROVE IT
KeepCalmAndPosters.com

DFS on Undirected Graphs (Problem 22.3-6)

Classifying an edge (u, v) as a tree edge or a back edge according to whether (u, v) or (v, u) is encountered first during the depth-first search

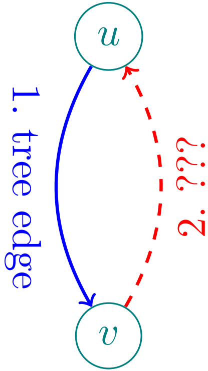
is equivalent to

classifying it according to the ordering of the four types in the classification scheme.

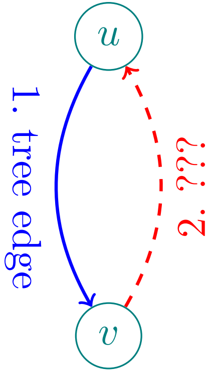
"First Types"	*vs.*	"First Time"
tree edge	$\Longleftrightarrow$	tree edge
back edge	$\Longleftrightarrow$	back edge

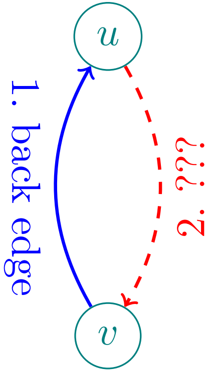
"First Types"	$\Longleftarrow$	"First Time"
tree edge	$\Longleftarrow$	tree edge
back edge	$\Longleftarrow$	back edge

"First Types"	$\Longleftarrow$	"First Time"
tree edge	$\Longleftarrow$	tree edge
back edge	$\Longleftarrow$	back edge

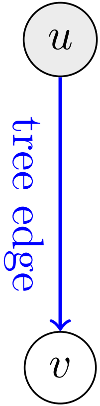


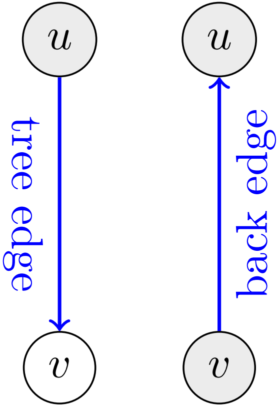
"First Types"	$\Longleftarrow$	"First Time"
tree edge	$\Longleftarrow$	tree edge
back edge	$\Longleftarrow$	back edge

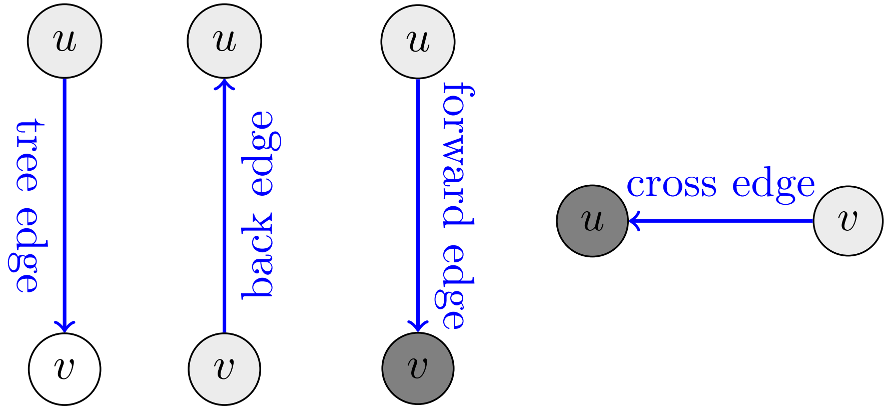




"First Types"	$\Longrightarrow$	"First Time"
tree edge	$\Longrightarrow$	tree edge
back edge	$\Longrightarrow$	back edge


u
tree edge
v


u
tree edge
v
u
back edge
v


u
v
tree edge
u
v
back edge
u
v
forward edge
u
v
cross edge

Edge Types and Lifetime of Vertices in DFS

$$\forall u \to v:$$

- tree/forward edge: $[_u \; [_v \;]_v \;]_u$
- back edge: $[_v \; [_u \;]_u \;]_v$
- cross edge: $[_v \;]_v \; [_u \;]_u$

Edge Types and Lifetime of Vertices in DFS

$$\forall u \to v :$$

- tree/forward edge: $[_u \ [_v \ ]_v \ ]_u$
- back edge: $[_v \ [_u \ ]_u \ ]_v$
- cross edge: $[_v \ ]_v \ [_u \ ]_u$

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

Edge Types and Lifetime of Vertices in DFS

$$\forall u \to v :$$

- tree/forward edge: $[_u \ [_v \ ]_v \ ]_u$
- back edge: $[_v \ [_u \ ]_u \ ]_v$
- cross edge: $[_v \ ]_v \ [_u \ ]_u$

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

Edge Types and Lifetime of Vertices in DFS

$$\forall u \to v :$$

- tree/forward edge: $[_u \; [_v \;]_v \;]_u$
- back edge: $[_v \; [_u \;]_u \;]_v$
- cross edge: $[_v \;]_v \; [_u \;]_u$

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

$$\mathrm{f}[u] < \mathrm{f}[v] \iff$$

Edge Types and Lifetime of Vertices in DFS

$$\forall u \to v :$$

- tree/forward edge: $[_u\ [_v\ ]_v\ ]_u$
- back edge: $[_v\ [_u\ ]_u\ ]_v$
- cross edge: $[_v\ ]_v\ [_u\ ]_u$

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

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

Edge Types and Lifetime of Vertices in DFS

$$\forall u \to v:$$

- tree/forward edge: $[_u\ [_v\ ]_v\ ]_u$
- back edge: $[_v\ [_u\ ]_u\ ]_v$
- cross edge: $[_v\ ]_v\ [_u\ ]_u$

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

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

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

On digraphs:

$$\nexists \text{ back edge} \iff \text{DAG} \iff \exists \text{ topo. ordering}$$

On digraphs:

$$\nexists \text{ back edge} \iff \text{DAG} \iff \exists \text{ topo. ordering}$$

Toposort by Tarjan (probably), 1976

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

On digraphs:

$$\nexists \text{ back edge} \iff \text{DAG} \iff \exists \text{ topo. ordering}$$

TOPOSORT by Tarjan (probably), 1976

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

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

Kahn's TOPOSORT algorithm (1962; Problem 22.4-5)

- Queue Q for source vertices $(\text{in}[v] = 0)$
- Repeat: $\textsc{Dequeue}(\exists u \in Q)$, output u

  delete u and $u \to v$ from Q,

  $\textsc{Enqueue}(v)$ if $\text{in}[v] = 0$

Kahn's TOPOSORT algorithm (1962; Problem 22.4-5)

- Queue Q for source vertices ($\text{in}[v] = 0$)
- Repeat: DEQUEUE($\exists u \in Q$), output u

  delete u and $u \to v$ from Q,

  ENQUEUE(v) if $\text{in}[v] = 0$

$$O(m + n)$$

Kahn's TOPOSORT algorithm (1962; Problem 22.4-5)

- Queue Q for source vertices ($\text{in}[v] = 0$)
- Repeat: $\textsc{Dequeue}(\exists u \in Q)$, output u

  delete u and $u \to v$ from Q,

  $\textsc{Enqueue}(v)$ if $\text{in}[v] = 0$

$$O(m + n)$$

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 22.4-5)

- Queue Q for source vertices ($\text{in}[v] = 0$)
- Repeat: DEQUEUE($\exists u \in Q$), output u

  delete u and $u \to v$ from Q,

  ENQUEUE(v) if $\text{in}[v] = 0$

$$O(m + n)$$

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?

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)

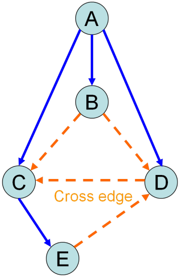
	Digraph	Undirected graph
DFS	back edge $\iff$ cycle	back edge $\iff$ cycle
BFS		cross edge $\iff$ cycle

Cycle detection (Problem 5.8 – 1)

	Digraph	Undirected graph
DFS	back edge $\Longleftrightarrow$ cycle	back edge $\Longleftrightarrow$ cycle
BFS	back edge $\Longrightarrow$ cycle cycle $\not\Longrightarrow$ back edge	cross edge $\Longleftrightarrow$ cycle

Cycle detection (Problem 5.8 – 1)

	Digraph	Undirected graph
DFS	back edge $\Longleftrightarrow$ cycle	back edge $\Longleftrightarrow$ cycle
BFS	back edge $\Longrightarrow$ cycle cycle $\not\Longrightarrow$ back edge	cross edge $\Longleftrightarrow$ cycle



Cycle Deletion (Problem 22.4-3)

Whether an undirected graph G contains a cycle?

$$O(|V|)$$

Cycle Deletion (Problem 22.4-3)

Whether an undirected graph G contains a cycle?

$$O(|V|)$$

$$\text{tree: } |E| = |V| - 1 \implies \text{check } |E| \geq |V|$$

Theorem (Digraph as DAG)

Every digraph is a dag of its SCCs.

Theorem (Digraph as DAG)

Every digraph is a dag of its SCCs.

Two tiered structure of digraphs:

digraph $\equiv$ a dag of SCCs

SCC: equivalence class over reachability

$\boxed{\text{digraph} \equiv \text{a dag of SCCs}}$

Kosaraju's SCC algorithm, 1978

"SCCs can be topo-sorted

in decreasing order of their highest finish time."

$\boxed{\text{digraph} \equiv \text{a dag of SCCs}}$

Kosaraju's SCC algorithm, 1978

"SCCs can be topo-sorted

in decreasing order of their highest finish time."

The vertice with the highest finish time is in a source SCC.

$\boxed{\text{digraph} \equiv \text{a dag of SCCs}}$

Kosaraju's SCC algorithm, 1978

"SCCs can be topo-sorted

in decreasing order of their highest finish time."

The vertice with the highest finish time is in a source SCC.

(I) DFS on G; DFS on G^T

$\boxed{\text{digraph} \equiv \text{a dag of SCCs}}$

Kosaraju's SCC algorithm, 1978

"SCCs can be topo-sorted
in decreasing order of their highest finish time."

The vertice with the highest finish time is in a source SCC.

(I) DFS on G; DFS on G^T

(II) DFS on G^T; DFS on G

$$\boxed{\text{digraph} \equiv \text{a dag of SCCs}}$$

Kosaraju's SCC algorithm, 1978

"SCCs can be topo-sorted

in decreasing order of their highest finish time."

The vertice with the highest finish time is in a source SCC.

(I) DFS on G; DFS/BFS on G^T

(II) DFS on G^T; DFS/BFS on G

Semiconnected Digraph (Problem 4.14)

$$\forall u, v \in V : u \rightsquigarrow v \lor v \rightsquigarrow u$$

Semiconnected Digraph (Problem 4.14)

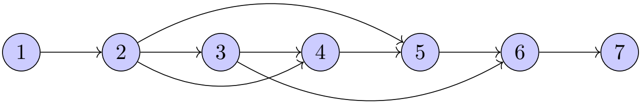
$$\forall u, v \in V : u \rightsquigarrow v \vee v \rightsquigarrow u$$

$$\boxed{\text{digraph} \equiv \text{a dag of SCCs}}$$

Semiconnected Digraph (Problem 4.14)

$$\forall u, v \in V : u \rightsquigarrow v \lor v \rightsquigarrow u$$

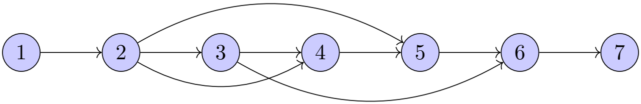
$\boxed{\text{digraph} \equiv \text{a dag of SCCs}}$



Semiconnected Digraph (Problem 4.14)

$$\forall u, v \in V : u \rightsquigarrow v \lor v \rightsquigarrow u$$

$\boxed{\text{digraph} \equiv \text{a dag of SCCs}}$



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

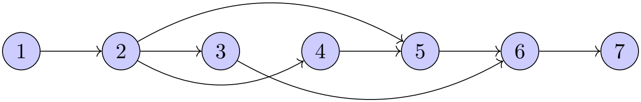
$$\boxed{\text{DAG: Semiconnected} \iff \exists!\ \text{topo. ordering}}$$

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

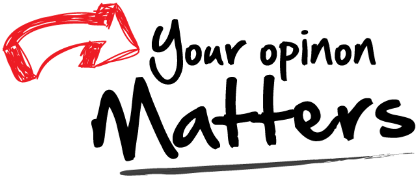
Tarjan's TOPOSORT + Check edges (v_i, v_{i+1})

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

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




THANK YOU



Office 302

Mailbox: H016

hfwei@nju.edu.cn