Path-based depth-first search for strong and biconnected components

Author of the paper: Harold N. Gabow

Reported by: T.T. Liu D.P. Xu B.Y. Chen

May 28, 2017

Outline

Characterastics of Gabow's Algorithms

- One-pass algorithm. But for the algorithm of strong components, what we have learned from the textbook is a two-pass algorithm, by which we must traverse the whole graph twice.
- Lower time and space complexity. This algorithm only use stacks and an array, and do not employ a disjoint-set data structure.

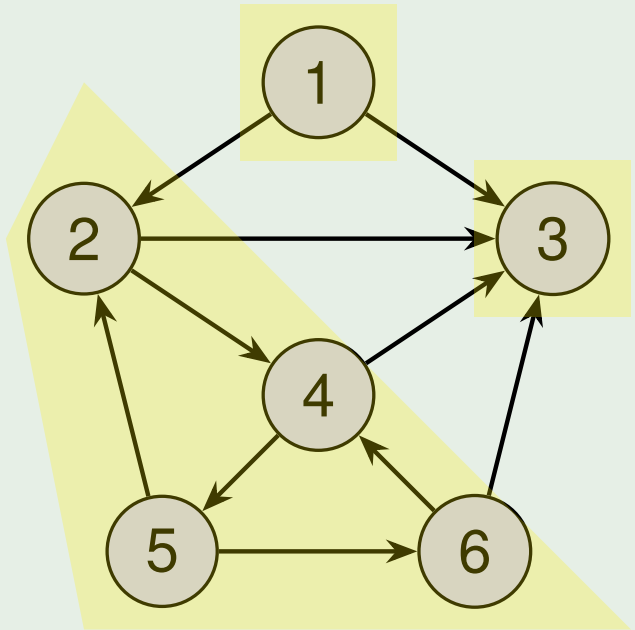
Outline

Review: What have we learned from the textbook?

Concepts of Strong Components

- Two mutually reachable vertices are in the same *strong component*.

Example



Review: What have we learned from the textbook?

Algorithms to Find Strong Components

- Idea: Run DFS twice. Once on the original graph G, once on the *tranposition* of G^T.
- Trick: Using *finishing times* of each vertex computed by the first DFS.
- Linear time complexity: $O(V + E)$

Outline

Purdom and Munro's High-Level Algorithm

Algorithm 1: Strong Components

```
H = G;
while H still has a vertex v do
    start a new path P = (v);
    while P is not empty do /* Grows path P                    */
        if the last vertex v_k of P has an edge (v_k, w) then
            if w ∉ P then /* w and v_i are identical.          */
                add w to P, as the new last vertex of P;
            else
                contract the cycle v_i, v_{i+1}, ··· , v_k, both in H and in P;
        else
            output v_k as a vertex of the strong component graph;
            delete v_k from both H and P;
```

Assessment

- Note that *contracting* means selecting one vertex as a representation and merging others rather than deleting them.
- Correctness: If no edge leaves v_k then v_k is a vertex of the finest acyclic contraction.

Assessment

- The time consumption of each statement in the pseudo-code is clear. Total time complexity is linear. except this statement:

`contract the cycle` $v_i, v_{i+1}, \cdots, v_k$, `both in` H `and in` P;

- Problem is how to merge in linear time while keeping the next time accessing this vertex still in constant time.
- Therefore, a good data structure for disjoint-set merging is needed usually.

Outline

His Contribution

- He gave a simple list-based implementation that achieves linear time.
- Use only stacks and arrays as data structure.
- Do not need a disjoint set merging data structure.

Data Structure Used in Algorithm

- In DFS, the path P from root to each node is almost always significant. So it is in this algorithm.
- A stack S contains the sequence of vertices in P.
- A stack B contains the boundaries between contracted vertices.
- An array $I[1 \ldots n]$ is used to store stack indices corresponding to vertices.

Contraction Makes Much Difference

- When contraction is executed, some vertices merges into a set.
- It is possible that several elements in stack S are in the same vertex in path P. More formal,

$$v_i = S[j] : B[i] \leq j < B[i+1]$$

- By the way, the formal definition of $I[v]$ is

$$I[j] = \begin{cases} 0, & \text{if } v \text{ has never been in P;} \\ j, & \text{if } v \text{ is currently in P and } S[j] = v; \\ c, & \text{if the strong component containing } v \text{ has} \\ & \text{been deleted and numbered as } c. \end{cases}$$

where c counts from $n+1$.

New algorithm to discover strong components

Procedure 2: STRONG(G)

empty stacks S and B;
for $v \in V$ **do**
  $I[v] = 0$;
$c = n$;
for $v \in V$ **do**
  if $I[v] = 0$ **then** /* vertex v has never been accessed yet */
    DFS(v);

New algorithm to discover strong components

Procedure 3: DFS(v)

PUSH(v, S); $I[v] =$TOP(S); PUSH$([v], B)$;
/* add v to the end of P */
for *egdes*$(v, w) \in E$ **do**
  if $I[v] = 0$ **then**
    DFS(w);
  else /* contract if necessary */
    while $I[w] < B[TOP[B]]$ **do**
      POP(B);
if $I[v] = B[TOP(B)]$ **then** /* number vertices of the next strong component */
  POP(B);
  $c = c + 1$;
  while $I[v] \leq TOP[S]$ **do**
    $I[$POP$(S)] = c$;

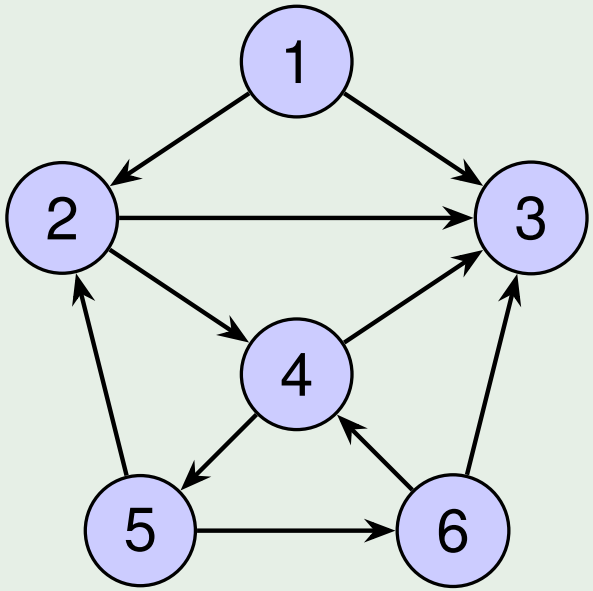
Demo: Gabow's strong components algorithm

Data in memory

B and S: stack. I: array.

B	S		I
		1	0
		2	0
		3	0
		4	0
		5	0
		6	0

Graph H



- Call Stack: STRONG()
- This state is the first after initialized. DFS(1) is going to be called.

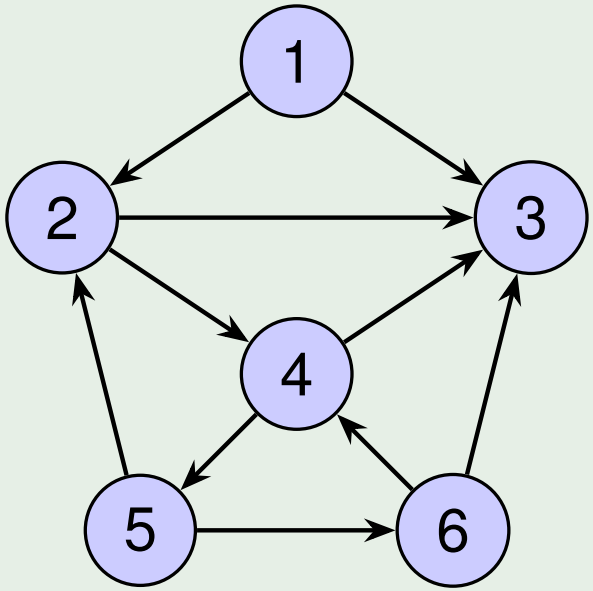
Demo: Gabow's strong components algorithm

Data in memory

B and S: stack. I: array.

B	S		I
1	1	1	1
		2	0
		3	0
		4	0
		5	0
		6	0

Graph H



- Call Stack: STRONG()→DFS(1)
- Code: `for edges(v,w)∈E do ...`
- $w = 2$.

Demo: Gabow's strong components algorithm

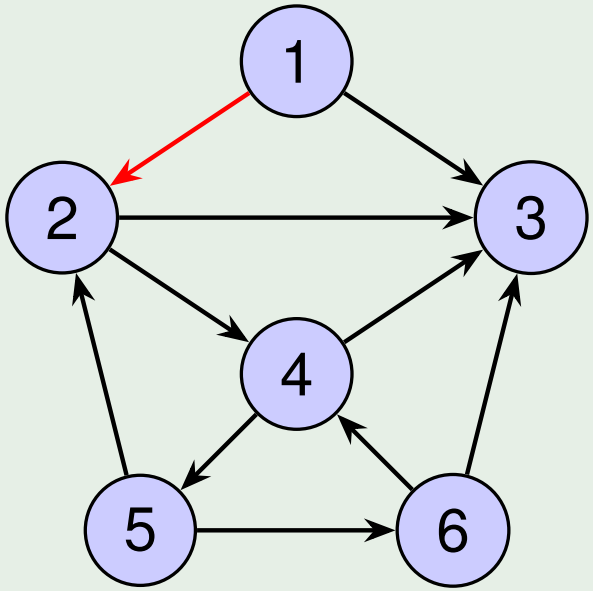
Data in memory

B and S: stack. I: array.

B	S
1	1
2	2

	I
1	1
2	2
3	0
4	0
5	0
6	0

Graph H



- Call Stack: STRONG()→DFS(1)→DFS(2)
- Code: `for edges(v,w)∈E do ...`
- $w = 3$.

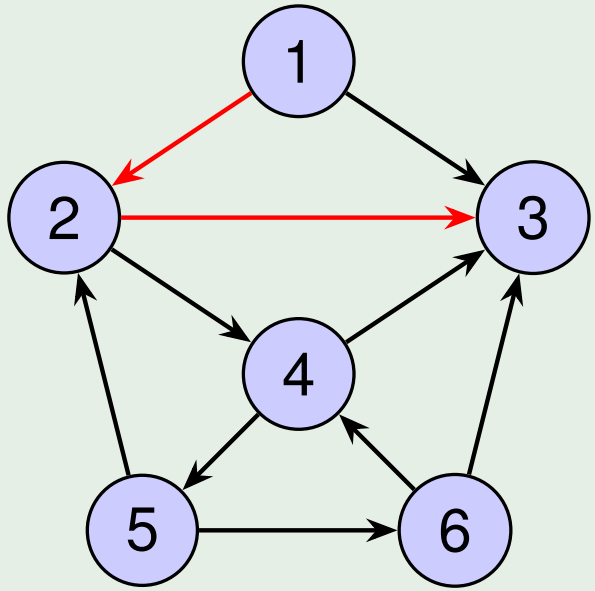
Demo: Gabow's strong components algorithm

Data in memory

B and S: stack. I: array.

B	S		I
1	1	1	1
2	2	2	2
3	3	3	3
		4	0
		5	0
		6	0

Graph H



- Call Stack: STRONG()→DFS(1)→DFS(2)→DFS(3)
- Code: `if I[v]=B[TOP(B)] then ...`
- Go back.

Demo: Gabow's strong components algorithm

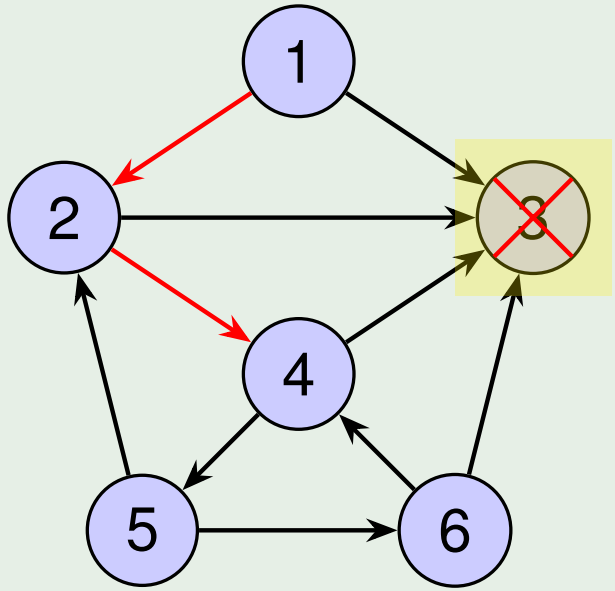
Data in memory

B and S: stack. I: array.

B	S
1	1
2	2
3	4

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

Graph H



- Call Stack: STRONG()→DFS(1)→DFS(2)→DFS(4)
- Code: `for edges(v,w)∈E do ...`
- $w = 5$.

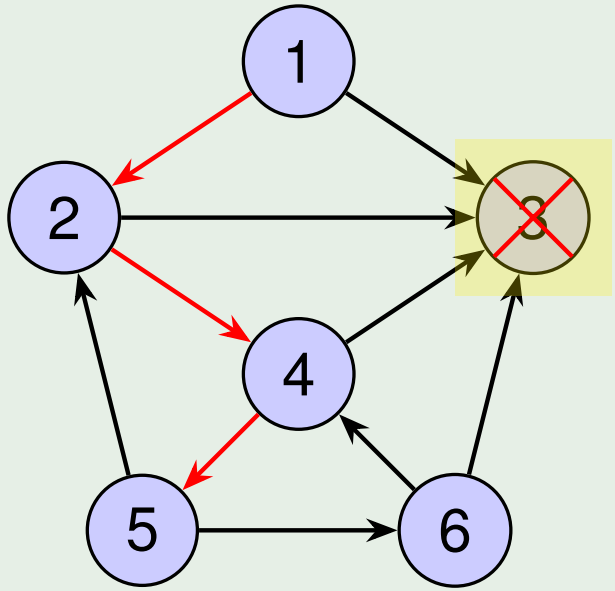
Demo: Gabow's strong components algorithm

Data in memory

B and S: stack. I: array.

B	S		I
1	1	1	1
2	2	2	2
3	4	3	7
4	5	4	3
		5	4
		6	0

Graph H



- Call Stack: $\cdots \rightarrow$DFS(1)$\rightarrow$DFS(2)$\rightarrow$DFS(4)$\rightarrow$DFS(5)
- Code: `for edges(v,w)∈E do ...`
- $w = 2$.

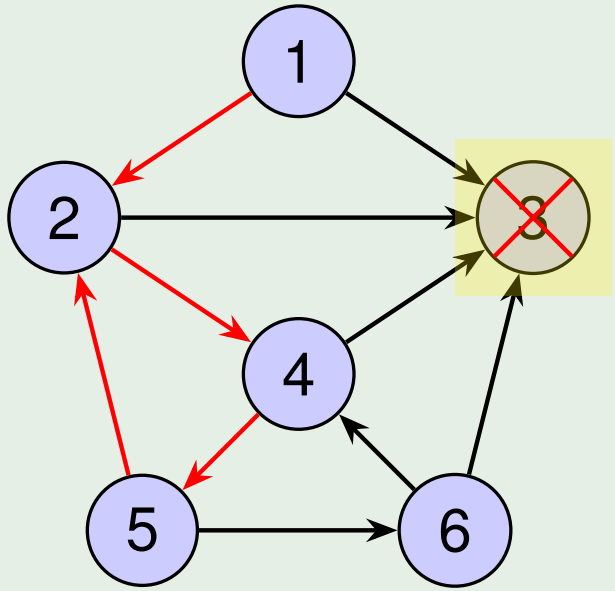
Demo: Gabow's strong components algorithm

Data in memory

B and S: stack. I: array.

B	S		I
1	1	1	1
2	2	2	2
3	4	3	7
4	5	4	3
		5	4
		6	0

Graph H



- Call Stack: $\cdots \rightarrow$DFS(1)$\rightarrow$DFS(2)$\rightarrow$DFS(4)$\rightarrow$DFS(5)
- Code: `while I[w]<B[TOP(B)] do POP(B);`
- Now, $w = 2$, contract!

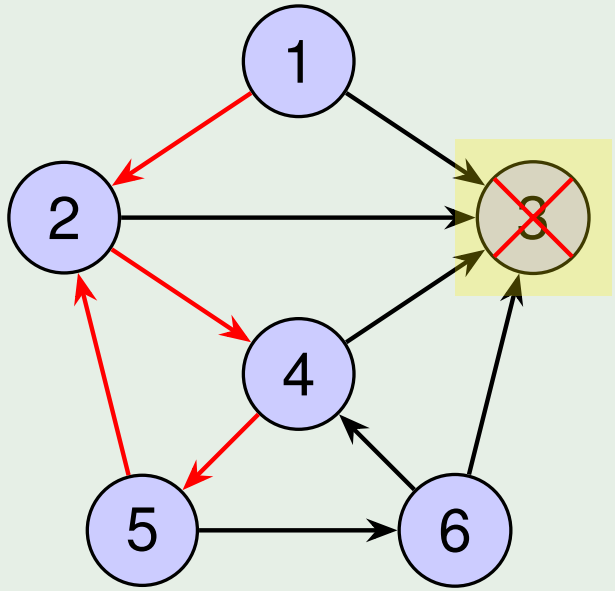
Demo: Gabow's strong components algorithm

Data in memory

B and S: stack. I: array.

B	S		I
1	1	1	1
2	2	2	2
	4	3	7
	5	4	3
		5	4
		6	0

Graph H



- Call Stack: $\cdots \rightarrow$DFS(1)$\rightarrow$DFS(2)$\rightarrow$DFS(4)$\rightarrow$DFS(5)
- Code: `while I[w]<B[TOP(B)] do POP(B);`
- Now, $w = 2$, contract!

Demo: Gabow's strong components algorithm

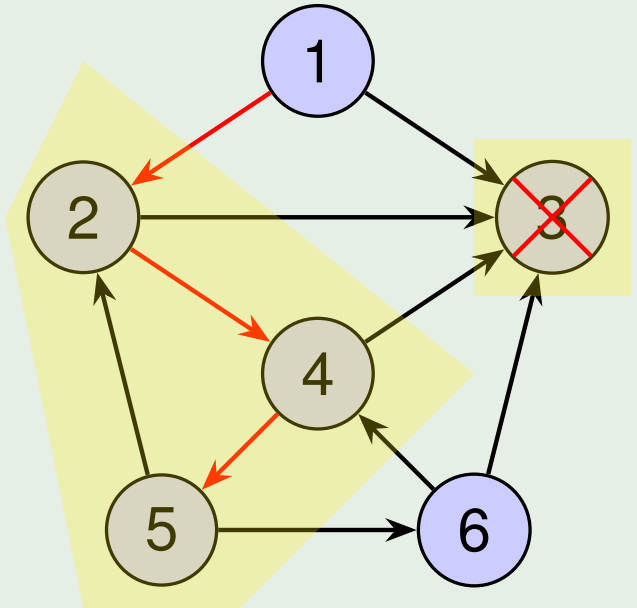
Data in memory

B and S: stack. I: array.

B	S
1	1
2	2
	4
	5

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

Graph H



- Call Stack: $\cdots \rightarrow$DFS(1)$\rightarrow$DFS(2)$\rightarrow$DFS(4)$\rightarrow$DFS(5)
- Code: `if I[w]=0 then DFS(w);`
- $w = 6$.

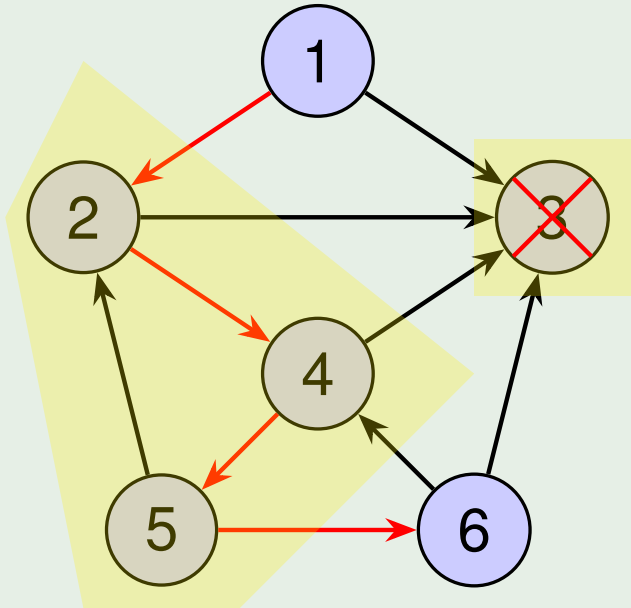
Demo: Gabow's strong components algorithm

Data in memory

B and S: stack. I: array.

B	S		I
1	1	1	1
2	2	2	2
5	4	3	7
	5	4	3
	6	5	4
		6	5

Graph H



- Call Stack: $\cdots \rightarrow$DFS(2)$\rightarrow$DFS(4)$\rightarrow$DFS(5)$\rightarrow$DFS(6)
- Code: `for edges(v,w)∈E do ...`
- $w = 4$.

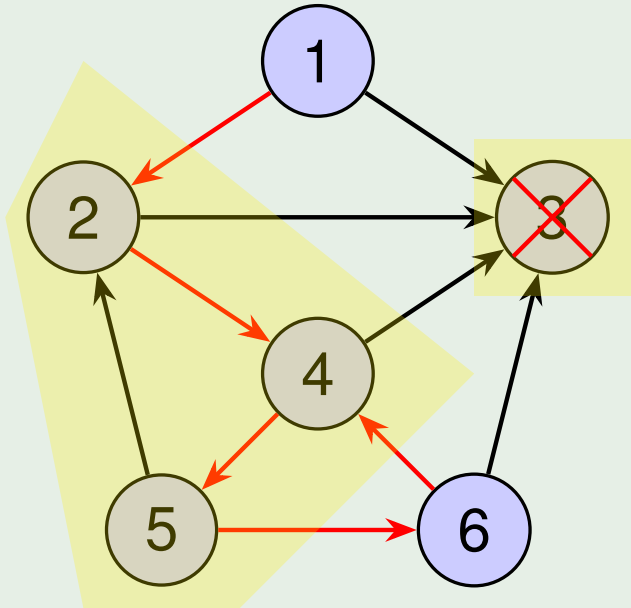
Demo: Gabow's strong components algorithm

Data in memory

B and S: stack. I: array.

B	S		I
1	1	1	1
2	2	2	2
5	4	3	7
	5	4	3
	6	5	4
		6	5

Graph H



- Call Stack: $\cdots \rightarrow$DFS(2)$\rightarrow$DFS(4)$\rightarrow$DFS(5)$\rightarrow$DFS(6)
- Code: `while I[w]<B[TOP(B)] do POP(B);`
- Now, $w = 4$, contract!

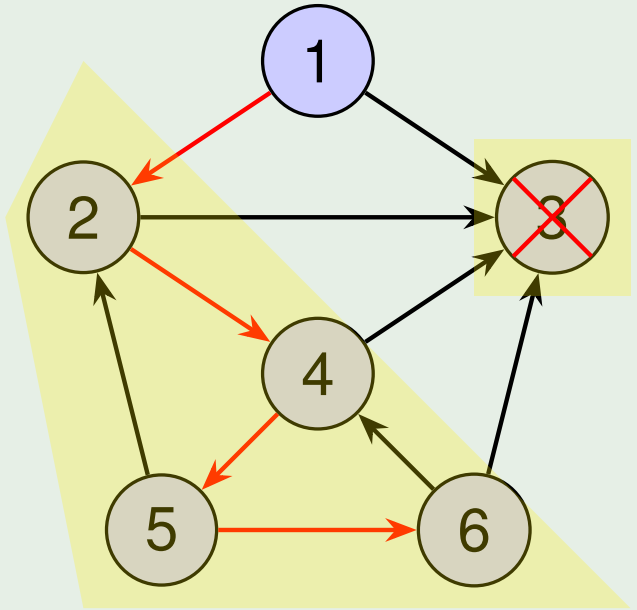
Demo: Gabow's strong components algorithm

Data in memory

B and S: stack. I: array.

B	S		I
1	1	1	1
2	2	2	2
	4	3	7
	5	4	3
	6	5	4
		6	5

Graph H



- Call Stack: $\cdots \rightarrow$DFS(2)$\rightarrow$DFS(4)$\rightarrow$DFS(5)$\rightarrow$DFS(6)
- Code: `if I[v]=B[TOP(B)] then ...`
- Go back.

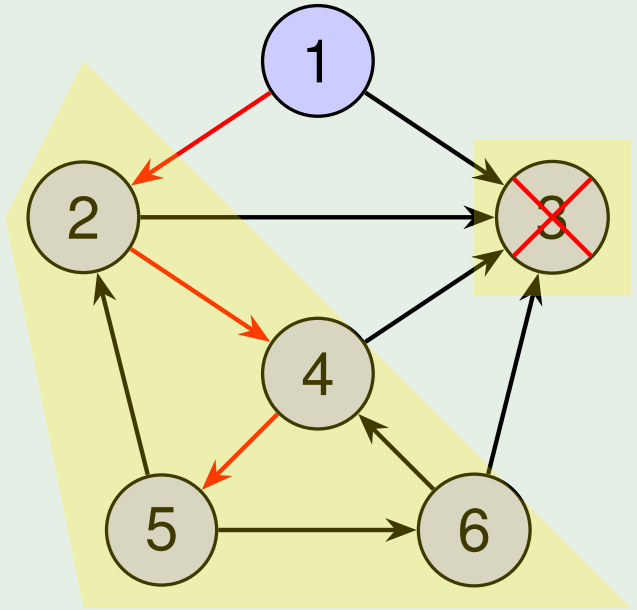
Demo: Gabow's strong components algorithm

Data in memory

B and S: stack. I: array.

B	S		I
1	1	1	1
2	2	2	2
	4	3	7
	5	4	3
	6	5	4
		6	5

Graph H



- Call Stack: $\cdots \rightarrow$DFS(2)$\rightarrow$DFS(4)$\rightarrow$DFS(5)
- Code: `if I[v]=B[TOP(B)] then ...`
- Go back.

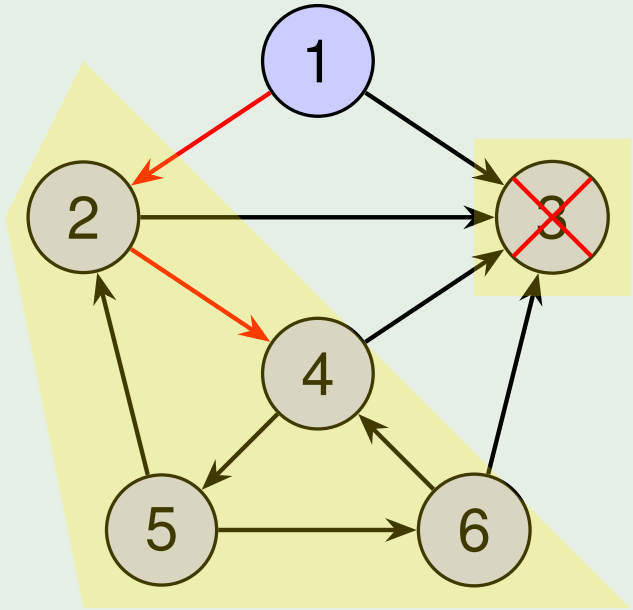
Demo: Gabow's strong components algorithm

Data in memory

B and S: stack. I: array.

B	S		I
1	1	1	1
2	2	2	2
	4	3	7
	5	4	3
	6	5	4
		6	5

Graph H



- Call Stack: $\cdots \rightarrow$DFS(2)$\rightarrow$DFS(4)
- Code: `if I[v]=B[TOP(B)] then ...`
- Go back.

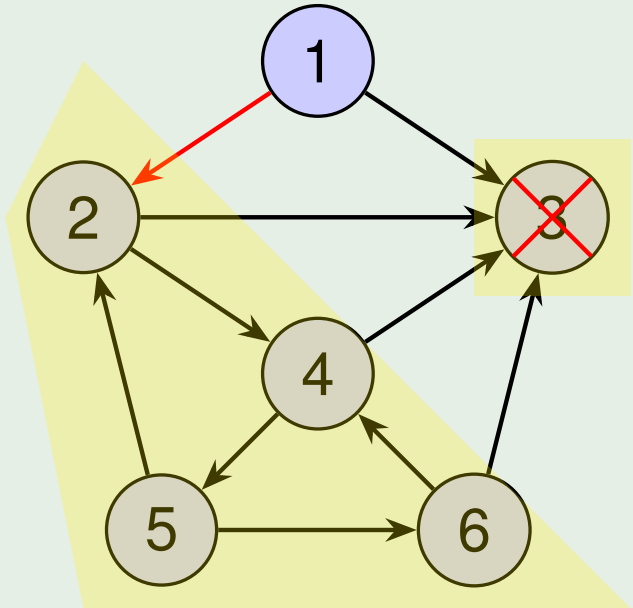
Demo: Gabow's strong components algorithm

Data in memory

B and S: stack. I: array.

B	S		I
1	1	1	1
2	2	2	2
	4	3	7
	5	4	3
	6	5	4
		6	5

Graph H



- Call Stack: STRONG()→DFS(1)→DFS(2)
- Code: `if I[v]=B[TOP(B)] then ...`
- Go back. But this time, Condition in last line is satisfied!

Demo: Gabow's strong components algorithm

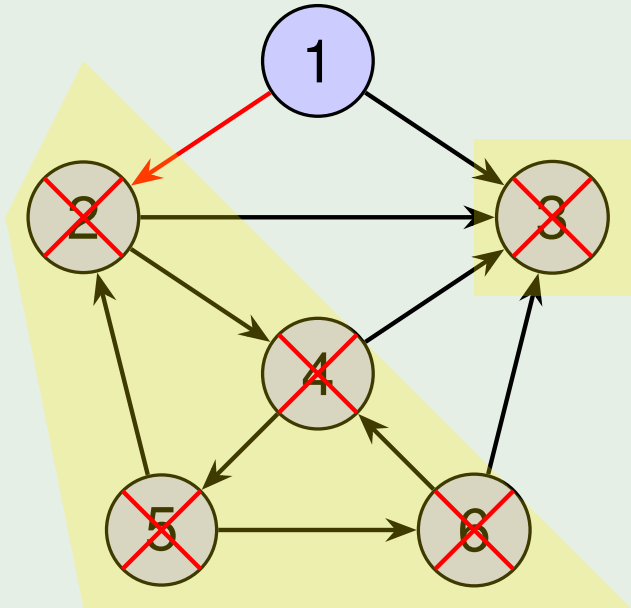
Data in memory

B and S: stack. I: array.

B	S
1	1

	I
1	1
2	8
3	7
4	8
5	8
6	8

Graph H



- Call Stack: STRONG()→DFS(1)→DFS(2)
- Code: `while I[v]≤TOP(S) do I[POP(S)]=c;`
- Pop 2 from B, while 2, 4, 5, 6 in S are also popped.

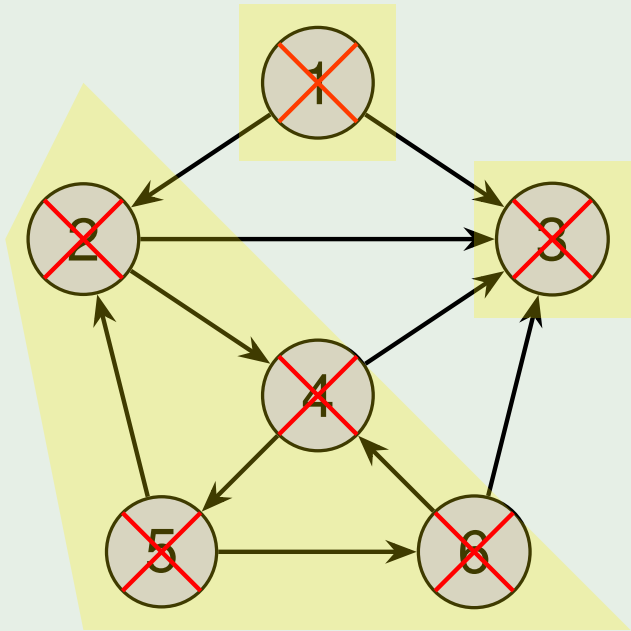
Demo: Gabow's strong components algorithm

Data in memory

B and S: stack. I: array.

B	S		I
		1	9
		2	8
		3	7
		4	8
		5	8
		6	8

Graph H



- Call Stack: STRONG()→DFS(1)
- Code: `while I[v]≤TOP(S) do I[POP(S)]=c;`
- Pop the last one both in B and in S. *Finished!!*

Outline

Correctness of Gabow's Strong Components Algorithm

Theorem (Correctness and Time Complexity)

When STRONG(G) halts each vertex $v \in V$ belongs to the strong component numbered $I[v]$. The time and space are both $O(V + E)$.

- The key of proof is to show that STRONG(G) is a valid implementation of the P&M's high-level algorithm.

Framework of STRONG(G)

Algorithm 4: Framework of High-Level Algorithm

$H = G$;
while H *still has a vertex* v **do**
    start a new path $P = (v)$;
    $\ldots$;

Procedure 5: STRONG(G)

empty stacks S and B;
for $v \in V$ **do**
    $I[v] = 0$;
$c = n$;
for $v \in V$ **do**
    if $I[v] = 0$ **then** `/* v has never been accessed */`
        DFS(v);

Growing Path P

Algorithm 6: A Part of High-Level Algorithm

if *the last vertex* v_k *of* P *has an edge* (v_k, w) **then**
  if $w \notin P$ **then** /* w and v_i are identical. */
    add w to P, as the new last vertex of P;
  else
    contract the cycle $v_i, v_{i+1}, \cdots, v_k$, both in H and in P;

Procedure 7: A Part of DFS(v)

for *egdes*$(v, w) \in E$ **do**
  if $I[v] = 0$ **then**
    DFS(w);
  else /* contract if necessary */
    while $I[w] < B[TOP[B]]$ **do**
      POP(B);

Having Found a Strong Components

Algorithm 8: A Part of High-Level Algorithm

if *the last vertex* v_k *of* P *has an edge* (v_k, w) **then**
    $\ldots$;
else
    output v_k as a vertex of the strong component graph;
    delete v_k from both H and P;

Procedure 9: A Part of DFS(v)

if $I[v] = B[TOP(B)]$ **then** `/* number vertices of the next strong component */`
    POP(B);
    $c = c + 1$;
    while $I[v] \leq TOP[S]$ **do**
        $I[\text{POP}(S)] = c$;

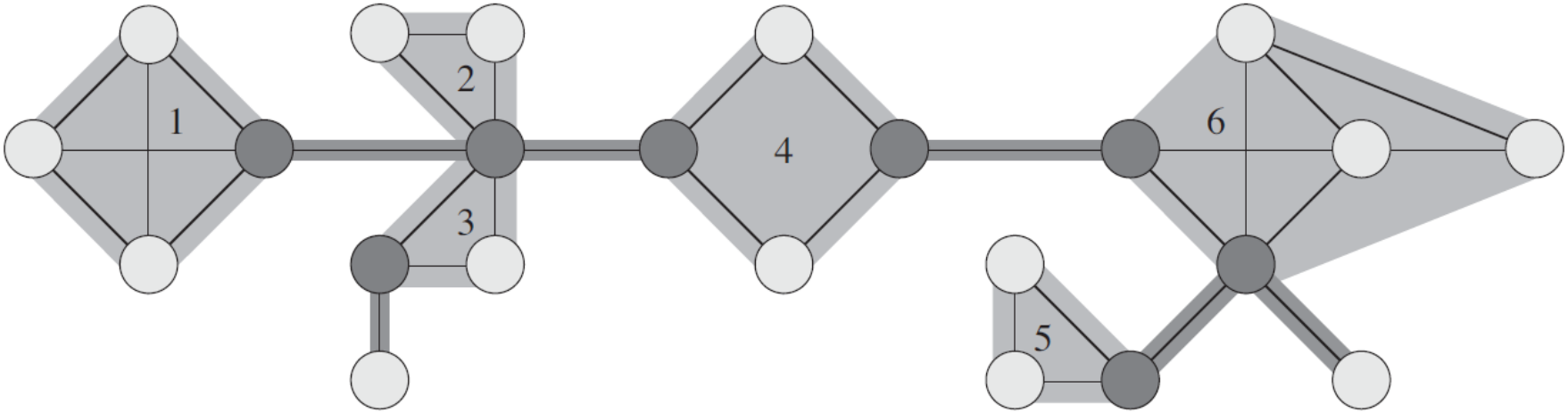
Time Complexity

- Every vertex is pushed onto and popped from each stack S, B exactly once. So the algorithm spends $O(1)$ time on each vertex or edge.
- Time Complexity: $O(V + E)$
- Intuitively, from another view, this algorithm is based on DFS, and no loop is executed on one vertex or one edge.

Outline

Concepts: Biconnected Component

- A *biconnected component* of G is a maximal set of edges such that any two edges in the set lie on a common simple cycle.

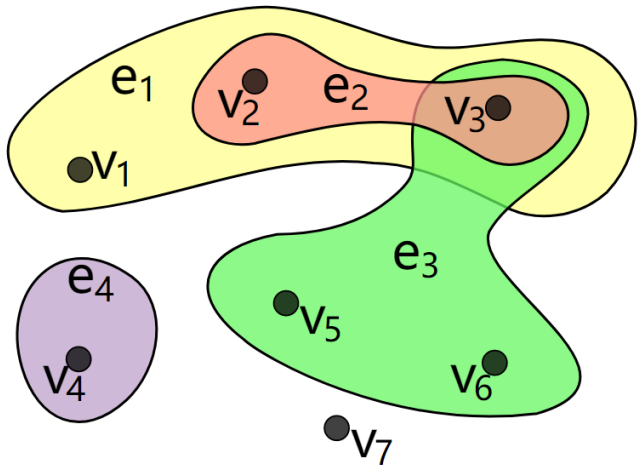


Concepts: Hypergraph

- A *hypergraph* $H = (V, E)$ is a generalization of a graph in which an edge can join any number of vertices.
- In the following hypergraph,

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$$

$$\begin{aligned} E &= \{e_1, e_2, e_3, e_4\} \\ &= \{\{v_1, v_2, v_3\}, \{v_2, v_3\}, \{v_3, v_5, v_6\}, \{v_4\}\} \end{aligned}$$



Concepts: Hypergraph

- Therefore, we need redefine the *edge*, *path*, *cycle*, $\cdots$, and nearly all concepts as long as it is relative to edge.
- A *path* is a sequence $(v_1, e_1, \cdots, v_k, e_k)$ of distinct vertices v_i and distinct edges e_i, $1 \leq i \leq k$, with $v_1 \in e_1$ and $v_i \in e_{i-1} \cap e_i$ for every $1 < i \leq k$.
- An important property:

$$v_{i+1} \in e_i - v_i, \quad 1 \leq i < k$$

- Merging a set of edges is to replace old edges with the new one:

$$e_{new} = \bigcup_{i=1}^{k} e_i$$

Summary

- The first main message of your talk in one or two lines.
- The second main message of your talk in one or two lines.
- Perhaps a third message, but not more than that.

- Outlook
  - Something you haven't solved.
  - Something else you haven't solved.