

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

June 3, 2017



Outline

- 1 Introduction
- 2 Strong Components
 - Reviews
 - Purdom and Munro's High-Level Algorithm
 - Contribution
 - Discussion
- 3 Biconnected Components
 - Review
 - High-Level Algorithm
 - Gabow's Algorithms



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 two stacks and an array, and do not employ a disjoint-set data structure.



Outline

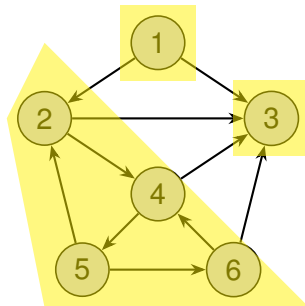
- 1 Introduction
- 2 **Strong Components**
 - **Reviews**
 - Purdom and Munro's High-Level Algorithm
 - Contribution
 - Discussion
- 3 Biconnected Components
 - Review
 - High-Level Algorithm
 - Gabow's Algorithms



Review: What have we learned from the textbook?

Concepts of Strong Components

- Two **mutually reachable** vertices are in the same *strong component*.
- It is a **equivalence relation**.



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 its *transpose* G^T .
- Trick: Using *finishing times* of each vertex computed by the first DFS.
- Linear time complexity: $O(V + E)$
- Proposed by S. Rao Kosaraju, known as the *Kosaraju's Algorithm*.



Outline

- 1 Introduction
- 2 Strong Components
 - Reviews
 - Purdom and Munro's High-Level Algorithm
 - Contribution
 - Discussion
- 3 Biconnected Components
 - Review
 - High-Level Algorithm
 - Gabow's Algorithms

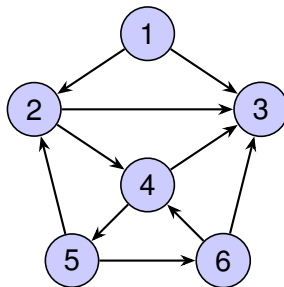


Purdom and Munro's High-Level Algorithm: Plain text

- Initially H is the given graph G . If H has no vertices stop. Otherwise *start a new path P* by choosing a vertex v and setting $P = (v)$. Continue by growing P as follows.
- To grow the path $P = (v_1, \dots, v_k)$ choose an edge (v_k, w) directed from the last vertex of P and do the following:
 - If $w \notin P$, *add w to P* , making it the new last vertex of P . Continue growing P .
 - If $w \in P$, say $w = v_i$, contract the cycle v_i, v_{i+1}, \dots, v_k , both in H and in P . P is now a path in the new graph H . Continue growing P .
 - If no edge leaves v_k , output v_k as a vertex of the strong component graph. Delete v_k from both H and P . If P is now nonempty continue growing P . Otherwise try to start a new path P .



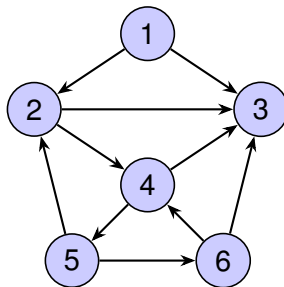
Demo: P&M's High-Level Algorithm



- Path $P = \{ \}$
- Initially, $H = G$.



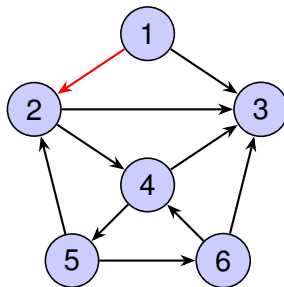
Demo: P&M's High-Level Algorithm



- Path $P = \{\{1\}\}$
- Grow P by adding v_1 .



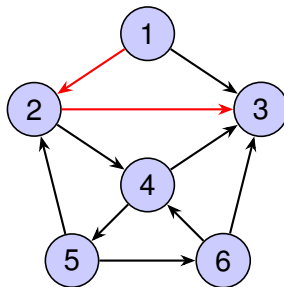
Demo: P&M's High-Level Algorithm



- Path $P = \{\{1\}, \{2\}\}$
- Grow P by adding v_2 .



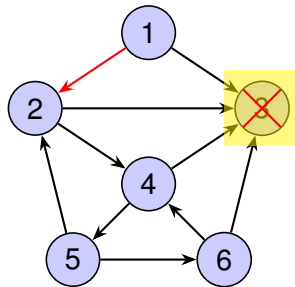
Demo: P&M's High-Level Algorithm



- Path $P = \{ \{1\}, \{2\}, \{3\} \}$
- Grow P by adding v_3 .



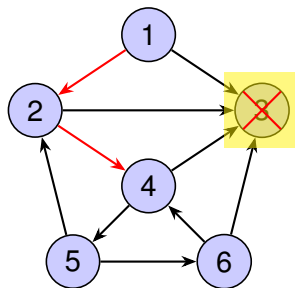
Demo: P&M's High-Level Algorithm



- Path $P = \{\{1\}, \{2\}\}$
- As v_3 is isolated, no edge leaves from v_3 , so just **delete** it.



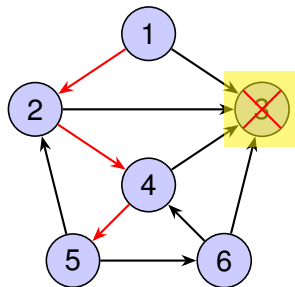
Demo: P&M's High-Level Algorithm



- Path $P = \{ \{1\}, \{2\}, \{4\} \}$
- Grow P by adding v_4 .



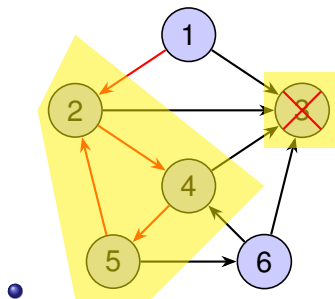
Demo: P&M's High-Level Algorithm



- Path $P = \{ \{1\}, \{2\}, \{4\}, \{5\} \}$
- Grow P by adding v_5 .



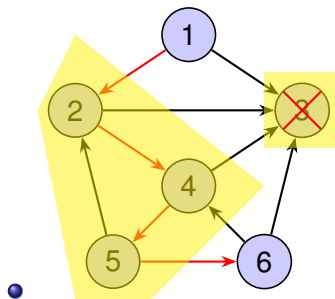
Demo: P&M's High-Level Algorithm



- Path $P = \{ \{1\}, \{2, 4, 5\} \}$
- The cycle v_2, v_4, v_5 in P is detected. Contract this cycle.



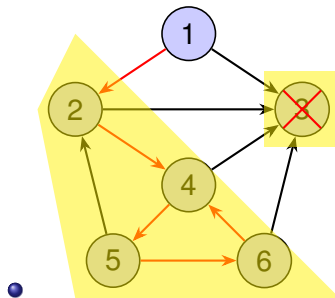
Demo: P&M's High-Level Algorithm



- Path $P = \{ \{1\}, \{2, 4, 5\}, 6 \}$
- Grow P by adding v_6 .



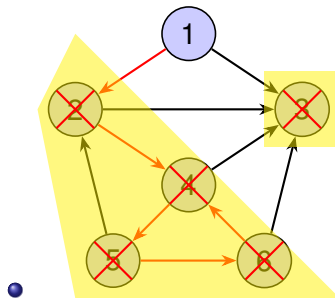
Demo: P&M's High-Level Algorithm



- Path $P = \{ \{1\}, \{2, 4, 5, 6\} \}$
- The cycle $\{v_2, v_4, v_5\}, v_6, v_4$ in P is detected. **Contract** this cycle.



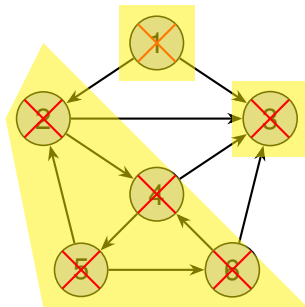
Demo: P&M's High-Level Algorithm



- Path $P = \{ \{1\}, \{2, 4, 5, 6\} \}$
- No edge leaves from $\{v_2, v_4, v_5, v_6\}$, so we **delete** it.



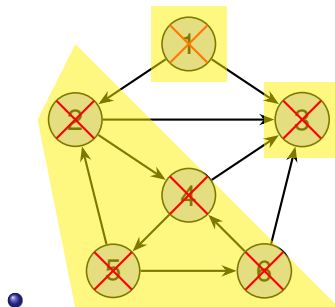
Demo: P&M's High-Level Algorithm



- Path $P = \{ \{1\} \}$
- No edge leaves from $\{v_1\}$, so we **delete** it.



Demo: P&M's High-Level Algorithm



- Path $P = \{ \}$
- Now graph H is **empty**, which has no vertex.

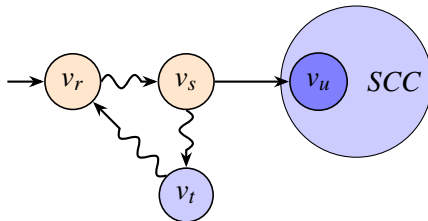


Correctness

- Correctness: If no edge leaves v_k then v_k is a vertex of the finest acyclic contraction.
- Easy to prove by contradiction: Assume there is an edge leaves $v_k \dots$



How to implement by DFS?



- Assume the current node is v_s which has two adjacent nodes. The current path is $P = (\dots, v_r, \dots, v_s)$.
- For the node v_u incident from v_s but also in the SCC , after running Sub-DFS() on this node, v_u will be removed with the SCC .



Pseudo Code

Algorithm 1: Strong components: Main-DFS(G) (DFS caller)

$H = G$;

while H still has a vertex v **do**

 | Sub-DFS(v); /* start a new path $P = (v)$ */



Pseudo Code

Algorithm 2: Strong components: Sub-DFS(v) (DFS callee)

add the v as the new last vertex of path P ;

for $w \in \{\text{vertices adjacent to } v\}$ **do**

if $w \notin P$ **then**

 Sub-DFS(w);

else $/* w = v_i, \text{ and } v = v_k \quad */$

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

if *no edge leaves v and v is the last DFS-finished vertex in a SCC* **then**

 output v as a vertex of the strong component graph;

 delete v from both H and P ;



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}, \dots, 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

- 1 Introduction
- 2 **Strong Components**
 - Reviews
 - Purdom and Munro's High-Level Algorithm
 - **Contribution**
 - Discussion
- 3 Biconnected Components
 - Review
 - High-Level Algorithm
 - Gabow's Algorithms



Gabow's 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 $I[1 \dots n]$ is used to store stack S 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

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

where c counts from $n + 1$.



New Algorithm to Discover Strong Components

Procedure 3: 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 4: DFS(v)

```

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

```



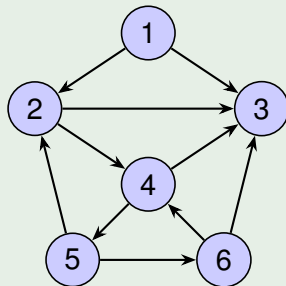
Demo: Gabow's Strong Components Algorithm

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.



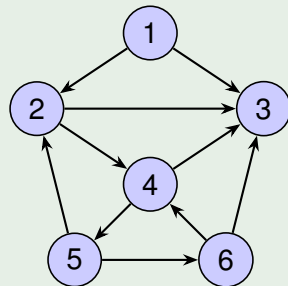
Demo: Gabow's Strong Components Algorithm

Data in memory

B and S: stack. I: array.

B	S	I
1	1	1
		2
		3
		4
		5
		6

Graph H



- Call stack: $\text{STRONG}() \rightarrow \text{DFS}(1)$
- Code: **for** edges $(v, w) \in E$ **do** ...
- $w = 2$.



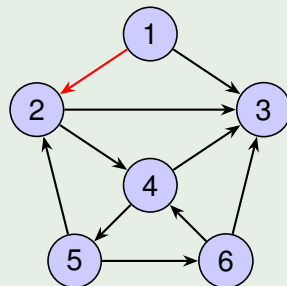
Demo: Gabow's Strong Components Algorithm

Data in memory

B and S: stack. I: array.

B	S	I
1	→ 1	1
2	→ 2	2
		3
		4
		5
		6

Graph H



- Call stack: `STRONG()` → `DFS(1)` → `DFS(2)`
- Code: **for** edges $(v, w) \in E$ **do** ...
- $w = 3$.



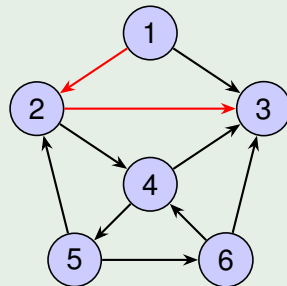
Demo: Gabow's Strong Components Algorithm

Data in memory

B and S: stack. I: array.

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

Graph H



- Call stack: $\text{STRONG}() \rightarrow \text{DFS}(1) \rightarrow \text{DFS}(2) \rightarrow \text{DFS}(3)$
- Code: **if** $I[v] = B[\text{TOP}(B)]$ **then** ...
- Go back.



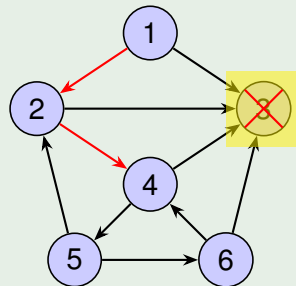
Demo: Gabow's Strong Components Algorithm

Data in memory

B and S: stack. I: array.

B		S		I
1	→	1		1
2	→	2		2
3	→	4		3
				7
				3
				0
				0

Graph H



- Call stack: STRONG() → DFS(1) → DFS(2) → DFS(4)
- Code: **for** edges $(v, w) \in E$ **do** ...
- $w = 5$.



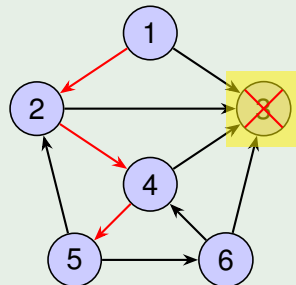
Demo: Gabow's Strong Components Algorithm

Data in memory

B and S: stack. I: array.

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

Graph H



- Call stack: $\dots \rightarrow \text{DFS}(1) \rightarrow \text{DFS}(2) \rightarrow \text{DFS}(4) \rightarrow \text{DFS}(5)$
- Code: **for** edges $(v, w) \in E$ **do** \dots
- $w = 2$.



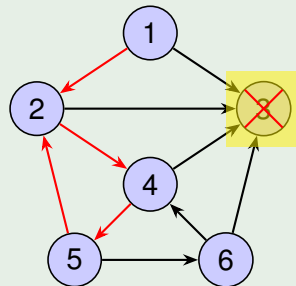
Demo: Gabow's Strong Components Algorithm

Data in memory

B and S: stack. I: array.

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

Graph H



- Call stack: $\dots \rightarrow \text{DFS}(1) \rightarrow \text{DFS}(2) \rightarrow \text{DFS}(4) \rightarrow \text{DFS}(5)$
- Code: **while** $I[w] < B[\text{TOP}(B)]$ **do** $\text{POP}(B)$;
- Now, $w = 2$, contract!



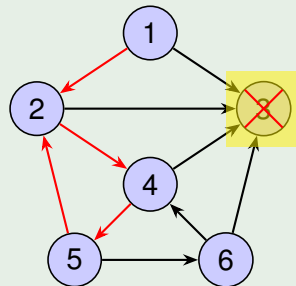
Demo: Gabow's Strong Components Algorithm

Data in memory

B and S: stack. I: array.

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

Graph H



- Call stack: $\dots \rightarrow \text{DFS}(1) \rightarrow \text{DFS}(2) \rightarrow \text{DFS}(4) \rightarrow \text{DFS}(5)$
- Code: **while** $I[w] < B[\text{TOP}(B)]$ **do** $\text{POP}(B)$;
- Now, $w = 2$, contract!



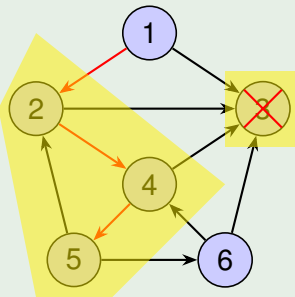
Demo: Gabow's Strong Components Algorithm

Data in memory

B and S: stack. I: array.

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

Graph H



- Call stack: $\dots \rightarrow \text{DFS}(1) \rightarrow \text{DFS}(2) \rightarrow \text{DFS}(4) \rightarrow \text{DFS}(5)$
- Code: **if** $I[w] = 0$ **then** $\text{DFS}(w)$;
- $w = 6$.



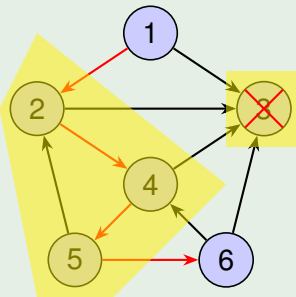
Demo: Gabow's Strong Components Algorithm

Data in memory

B and S: stack. I: array.

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

Graph H



- Call stack: $\dots \rightarrow \text{DFS}(2) \rightarrow \text{DFS}(4) \rightarrow \text{DFS}(5) \rightarrow \text{DFS}(6)$
- Code: **for** edges $(v, w) \in E$ **do** \dots
- $w = 4.$



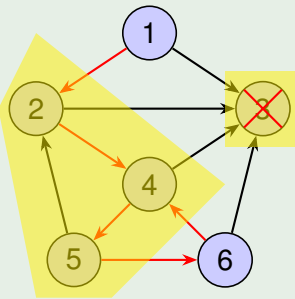
Demo: Gabow's Strong Components Algorithm

Data in memory

B and S: stack. I: array.

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

Graph H



- Call stack: $\dots \rightarrow \text{DFS}(2) \rightarrow \text{DFS}(4) \rightarrow \text{DFS}(5) \rightarrow \text{DFS}(6)$
- Code: **while** $I[w] < B[\text{TOP}(B)]$ **do** $\text{POP}(B)$;
- Now, $w = 4$, contract!



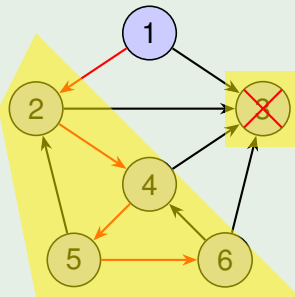
Demo: Gabow's Strong Components Algorithm

Data in memory

B and S: stack. I: array.

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

Graph H



- Call stack: $\dots \rightarrow \text{DFS}(2) \rightarrow \text{DFS}(4) \rightarrow \text{DFS}(5) \rightarrow \text{DFS}(6)$
- Code: **if** $I[v] = B[\text{TOP}(B)]$ **then** \dots
- Go back.



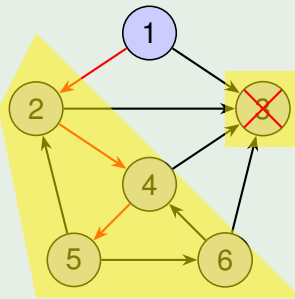
Demo: Gabow's Strong Components Algorithm

Data in memory

B and S: stack. I: array.

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

Graph H



- Call stack: $\dots \rightarrow \text{DFS}(2) \rightarrow \text{DFS}(4) \rightarrow \text{DFS}(5)$
- Code: **if** $I[v] = B[\text{TOP}(B)]$ **then** \dots
- Go back.



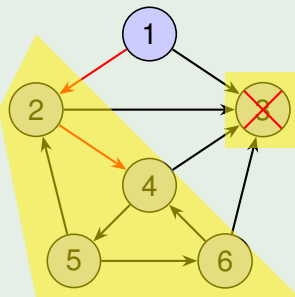
Demo: Gabow's Strong Components Algorithm

Data in memory

B and S: stack. I: array.

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

Graph H



- Call stack: $\dots \rightarrow \text{DFS}(2) \rightarrow \text{DFS}(4)$
- Code: **if** $I[v] = B[\text{TOP}(B)]$ **then** \dots
- Go back.



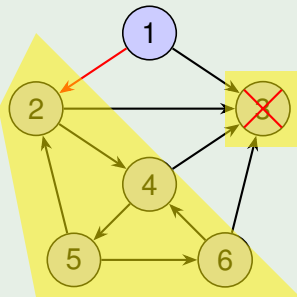
Demo: Gabow's Strong Components Algorithm

Data in memory

B and S: stack. I: array.

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

Graph H



- Call stack: $\text{STRONG}() \rightarrow \text{DFS}(1) \rightarrow \text{DFS}(2)$
- Code: **if** $I[v] = B[\text{TOP}(B)]$ **then** ...
- Go back. But this time, **Condition in last line is satisfied!**



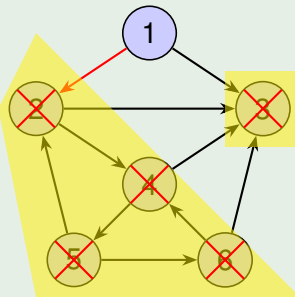
Demo: Gabow's Strong Components Algorithm

Data in memory

B and S: stack. I: array.

B	S	I
1	1	1
		2
		3
		4
		5
		6

Graph H



- Call stack: $\text{STRONG}() \rightarrow \text{DFS}(1) \rightarrow \text{DFS}(2)$
- Code: **while** $I[v] \leq \text{TOP}(S)$ **do** $I[\text{POP}(S)] = c$;
- Pop 2 from B, while 2, 4, 5, 6 in S are also popped.

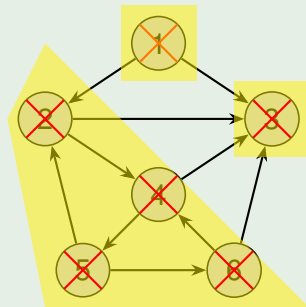
Demo: Gabow's Strong Components Algorithm

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: $\text{STRONG}() \rightarrow \text{DFS}(1)$
- Code: **while** $I[v] \leq \text{TOP}(S)$ **do** $I[\text{POP}(S)] = c$;
- Pop the last one both in B and in S. *Finished!!*

Outline

1 Introduction

2 Strong Components

- Reviews
- Purdom and Munro's High-Level Algorithm
- Contribution
- **Discussion**

3 Biconnected Components

- Review
- High-Level Algorithm
- Gabow's Algorithms



Correctness of Gabow's Strong Components Algorithm

Theorem (Correctness and 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 5: Strong components: Main-DFS(G) (DFS caller)

$$H = G;$$
while H still has a vertex v **do**

```
Sub-DFS(v); /* start a new path  $P = (v)$  */
```

Procedure 6: 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 7: A Part of High-Level Algorithm

```

for  $w \in \{\text{vertices adjacent to } v\}$  do
    if  $w \notin P$  then
        | Sub-DFS( $w$ );
    else /*  $w = v_i$ , and  $v = v_k$  */
        | contract the cycle  $v_i, v_{i+1}, \dots, v_k$ , both in  $H$  and in  $P$ ;
    */
    
```

Procedure 8: A Part of DFS(v)

```

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



Having Found a Strong Components

Algorithm 9: A Part of High-Level Algorithm

if *no edge leaves v and v is the last DFS-finished vertex in a SCC*
then
 | output v as a vertex of the strong component graph;
 | delete v from both H and P ;

Procedure 10: 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**
 | | $I[POP(S)] = c$;



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.



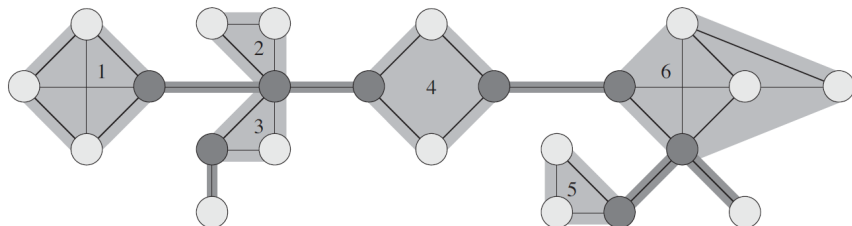
Outline

- 1 Introduction
- 2 Strong Components
 - Reviews
 - Purdom and Munro's High-Level Algorithm
 - Contribution
 - Discussion
- 3 Biconnected Components
 - **Review**
 - High-Level Algorithm
 - Gabow's Algorithms



Review: 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.



Outline

- 1 Introduction
- 2 Strong Components
 - Reviews
 - Purdom and Munro's High-Level Algorithm
 - Contribution
 - Discussion
- 3 Biconnected Components
 - Review
 - **High-Level Algorithm**
 - Gabow's Algorithms



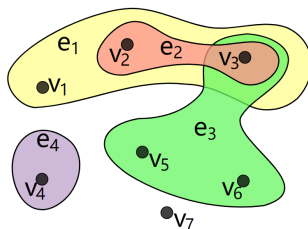
Concepts: Hypergraph

- 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,

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

$$E = \{e_1, e_2, e_3, e_4\}$$

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



Concepts: Hypergraph

- Therefore, we need redefine the *edge*, *path*, *cycle*, \dots , and nearly all concepts as long as it is relative to edge.
- A *path* is a sequence $(v_1, e_1, \dots, v_k, e_k)$ 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:

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

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

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



Additional Concepts We Need

- The *block hypergraph* H of G is the hypergraph formed by merging the edges of each biconnected component of G .
- The set of *all vertices in edges of* P is denoted

$$V(P) = \bigcup_{i=1}^k e_i$$



High-Level Algorithm in Plain Text

- Initially H is the given graph G . If H has no edges stop. Otherwise start a new path P by choosing an edge $\{v, w\}$ and setting $P = (v, \{v, w\})$. Continue by growing P .
- To grow the path $P = (v_1, e_1, \dots, v_k, e_k)$ choose an edge $\{v, w\} \neq e_k$ with $v \in e_k - \{v_k\}$ and do:
 - If $w \notin V(P)$, add $v, \{v, w\}$ to the end of P . Continue growing P .
 - If $w \in V(P)$, say $w \in e_i - \{v_{i+1}\}$, merge the edges of the cycle $w, e_i, v_{i+1}, e_{i+1}, \dots, v_k, e_k, v, \{v, w\}$ to a new edge $e = \bigcup_{j=i}^k e_j$, both in H and in P . Continue growing P .
 - If no edge leaves $e_k - \{v_k\}$, output e_k as an edge of the block hypergraph. Delete e_k from H and delete (v_k, e_k) from P . If P is now nonempty continue growing P . Otherwise try to start a new path P .



Pseudo Code

Algorithm 11: Biconnected Components: Main-DFS (DFS caller)

$H = G;$

while H still has an edge $\{v, w\}$ **do**

 | Sub-DFS(v); /* start a new path $P = (v, \{v, w\})$ */



Pseudo Code

Algorithm 12: Biconnected Components: Sub-DFS (DFS callee)

```

add the  $v$  as the new last vertex of path  $P$ ;
for  $w \in \{\text{vertices adjacent to } v\}$  do /* Grows path  $P$  */
    if  $w \notin V\{P\}$  then
        add  $\{v, w\}$  to the end of  $P$ , as the new last edge of  $P$ ;
        Sub-DFS( $w$ );
        remove the edge  $\{v, w\}$  if necessary;
    else /*  $w \in e_i - \{v_{i+1}\}$ , but most likely  $w \neq v_i$  */
        replace the cycle  $w, e_i, v_{i+1}, e_{i+1}, \dots, v_k, e_k, v$  to a new edge
             $e = \bigcup_{j=i}^k e_j$ , both in  $H$  and in  $P$ ;
if no edge leaves  $e_k - v_k$  then
    output  $e_k$  as an edge of the block hypergraph;
    delete  $e_k$  from  $H$  and delete  $(v_k, e_k)$  from  $P$ ;

```



Correctness

- When v , $\{v, w\}$ is added to P the result is a valid path, by the condition $v \in e_k - \{v_k\}$. When edges are merged they form a valid cycle, by the condition $\{v, w\} \neq e_k$.
- The algorithm correctly forms the finest acyclic merging of G , it finds the block hypergraph as desired.



Outline

- 1 Introduction
- 2 Strong Components
 - Reviews
 - Purdom and Munro's High-Level Algorithm
 - Contribution
 - Discussion
- 3 Biconnected Components
 - Review
 - High-Level Algorithm
 - **Gabow's Algorithms**



Data Structure Used in Algorithm

- A **stack** S contains the vertices $V(P)$.
- A **stack** B contains the boundaries between edges of P , *two vertices per boundary*.
- An array $I[1 \dots n]$ is used to store stack indices corresponding to vertices.
- All of above are similar (*but not the same*) to these in strong components.



Algorithms

Procedure 13: BICONN(G)

empty stacks S and B ;

for $v \in V$ **do**

$I[v] = 0$;

$c = n$;

for $v \in V$ **do**

if $I[v] = 0$ *and* v *is not isolated* **then**
 DFS(v);



Representations

- *Open arrows*: They point to the vertices v_i of P .

$$v_i = S[B[2i - 1]], \quad i = 1, \dots, k$$

Filled arrows: They demarcate the sets $e_i - v_i$; these sets are the "nonfirst" vertices of edges e_i of P .

$$e_i - \{v_i\} = \{S[j] : B[2i] \leq j < B[2i + 2]\}, \quad i = 1, \dots, k$$

- *Biconnected components*: Each edge $\{v, w\}$ belongs to the biconnected component with number $\min\{I[v], I[w]\}$.



Algorithms

Procedure 14: DFS(v)

```
PUSH( $v, S$ );  $I[v] = \text{TOP}(S)$ ;  
if  $I[v] > 1$  then /* create a filled arrow on  $B$  */  
    | PUSH( $I[v], B$ );  
for  $\text{edges}\{v, w\} \in E$  do  
    | if  $I[w] = 0$  then /* create an open arrow on  $B$  */  
        | PUSH( $I[v], B$ ); DFS( $w$ );  
        | else /* possible merge */  
            | while  $I[v] > 1$  and  $I[w] < B[\text{TOP}(B) - 1]$  do  
                | POP( $B$ ); POP( $B$ );  
if  $I[v] = 1$  then  
    |  $I[\text{POP}(S)] = c$ ;  
else if  $I[v] = B[\text{TOP}(B)]$  then  
    | POP( $B$ ); POP( $B$ );  $c = c + 1$ ;  
    | while  $I[v] \leq \text{TOP}(S)$  do  $I[\text{POP}(S)] = c$ ;
```



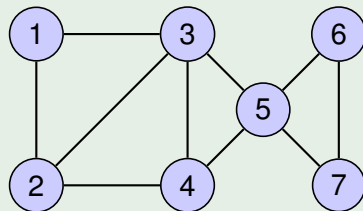
Demo: Gabow's biconnected components algorithm

Data in memory

B and S: stack. I: array.

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

Graph



- Procedure: BICONN(G)
- $P = \{ \}$



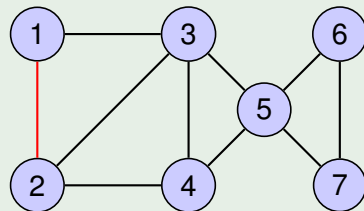
Demo: Gabow's biconnected components algorithm

Data in memory

B and S: stack. I: array.

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

Graph



- Procedure: DFS(1): $w=2$
- $P = \{v_1, \{v_1, v_2\}\}$



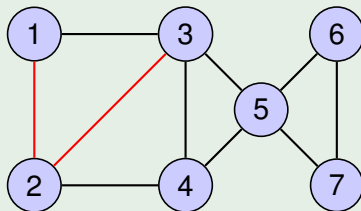
Demo: Gabow's biconnected components algorithm

Data in memory

B and S: stack. I: array.

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

Graph



- Procedure: DFS(2): w=3
- $P = \{v_1, \{v_1, v_2\}, v_2, \{v_2, v_3\}\}$

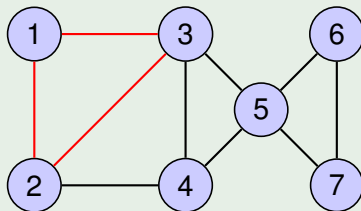
Demo: Gabow's biconnected components algorithm

Data in memory

B and S: stack. I: array.

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

Graph



- Procedure: DFS(3): $w=1$
- $P = \{v_1, \{v_1, v_2, v_3\}\}$



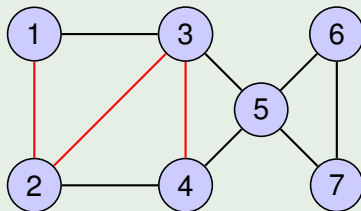
Demo: Gabow's biconnected components algorithm

Data in memory

B and S: stack. I: array.

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

Graph



- Procedure: DFS(3): w=4
- $P = \{v_1, \{v_1, v_2, v_3\}, v_3, \{v_3, v_4\}\}$

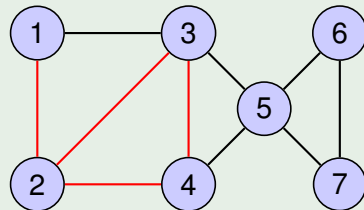
Demo: Gabow's biconnected components algorithm

Data in memory

B and S: stack. I: array.

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

Graph



• Procedure: DFS(4): $w=2$

• $P = \{v_1, \{v_1, v_2, v_3, v_4\}\}$

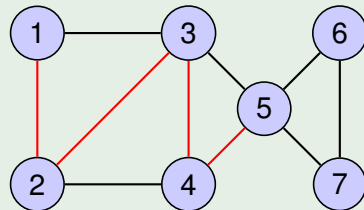
Demo: Gabow's biconnected components algorithm

Data in memory

B and S: stack. I: array.

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

Graph



- Procedure: DFS(4): w=5
- $P = \{v_1, \{v_1, v_2, v_3, v_4\}, v_4, \{v_4, v_5\}\}$



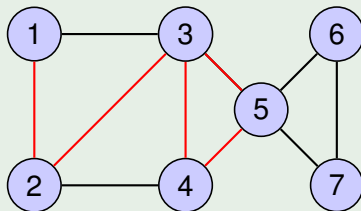
Demo: Gabow's biconnected components algorithm

Data in memory

B and S: stack. I: array.

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

Graph



- Procedure: DFS(5): w=3
- $P = \{v_1, \{v_1, v_2, v_3, v_4, v_5\}\}$

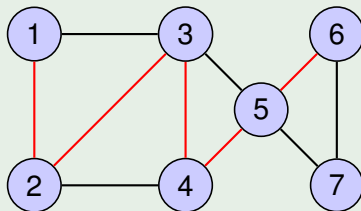
Demo: Gabow's biconnected components algorithm

Data in memory

B and S: stack. I: array.

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

Graph



- Procedure: DFS(5): w=6
- $P = \{v_1, \{v_1, v_2, v_3, v_4, v_5\}, v_5, \{v_5, v_6\}\}$

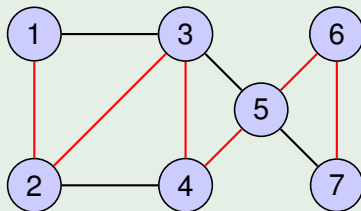
Demo: Gabow's biconnected components algorithm

Data in memory

B and S: stack. I: array.

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

Graph



• Procedure: DFS(6): w=7

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

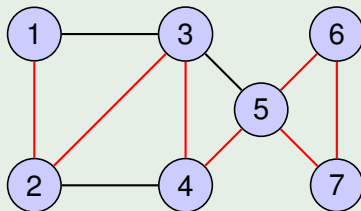
Demo: Gabow's biconnected components algorithm

Data in memory

B and S: stack. I: array.

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

Graph



- Procedure: DFS(7): w=5
- $P = \{v_1, \{v_1, v_2, v_3, v_4, v_5\}, v_5, \{v_5, v_6, v_7\}\}$



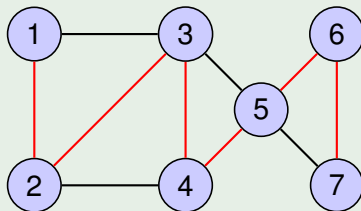
Demo: Gabow's biconnected components algorithm

Data in memory

B and S: stack. I: array.

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

Graph



• Procedure: DFS(7): End

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

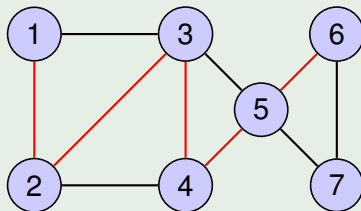
Demo: Gabow's biconnected components algorithm

Data in memory

B and S: stack. I: array.

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

Graph



• Procedure: DFS(6): End

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

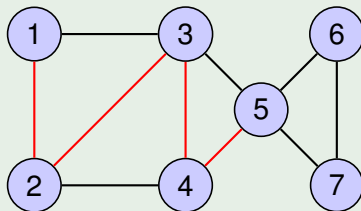
Demo: Gabow's biconnected components algorithm

Data in memory

B and S: stack. I: array.

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

Graph



• Procedure: DFS(5): End(No operation when w=7)

• $P = \{v_1, \{v_1, v_2, v_3, v_4, v_5\}\} \quad \{v_5, v_6, v_7\}$

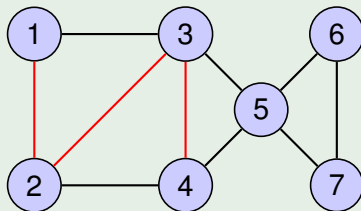
Demo: Gabow's biconnected components algorithm

Data in memory

B and S: stack. I: array.

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

Graph



• Procedure: DFS(4): End

• $P = \{v_1, \{v_1, v_2, v_3, v_4, v_5\}\} \quad \{v_5, v_6, v_7\}$

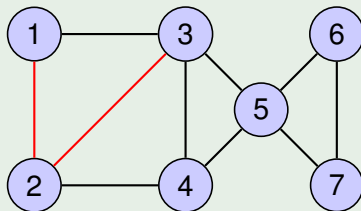
Demo: Gabow's biconnected components algorithm

Data in memory

B and S: stack. I: array.

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

Graph



• Procedure: DFS(3): End(No operation when $w=5$)

• $P = \{v_1, \{v_1, v_2, v_3, v_4, v_5\}\} \quad \{v_5, v_6, v_7\}$

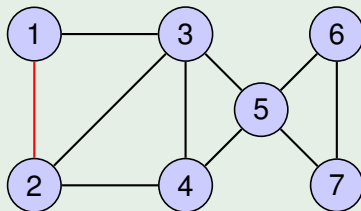
Demo: Gabow's biconnected components algorithm

Data in memory

B and S: stack. I: array.

B	S	I
	1	1
		2
		3
		4
		5
		6
		7

Graph



- Procedure: DFS(2): End

- $P = \{ \} \quad \{v_1, v_2, v_3, v_4, v_5\} \quad \{v_5, v_6, v_7\}$

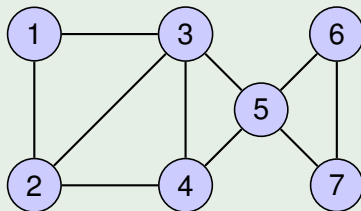
Demo: Gabow's biconnected components algorithm

Data in memory

B and S: stack. I: array.

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

Graph



- Procedure: DFS(1): End
- $P = \{ \} \quad \{v_1, v_2, v_3, v_4, v_5\} \quad \{v_5, v_6, v_7\}$

Correctness

- In order to keep the completeness, the correctness is given as follow.

Theorem (Correctness and complexity)

When $BICONN(G)$ halts any edge $\{v, w\} \in E$ belongs to the biconnected component numbered $\min\{I[v], I[w]\}$. The time and space are both $O(V + E)$.

