Connected Components

Martin Hoefer

Algorithms and Data Structures

Winter 2016

Subgraphs, Paths, Cycles

A (simple directed) graph $G = (V, E)$ has no multi-edges or loops. G contains a graph $G' = (V', E')$ if $V' \subseteq V$ and $E' \subseteq E$. We call G' a subgraph of G and denote $G' \subseteq G$.

- A (directed) path of length k is a graph $P_k = (V, E)$ with $V = \{v_1, \ldots, v_{k+1}\}$ and $E = \{(v_i, v_{i+1}) \mid i = 1, \ldots, k\}$, where we require $E = |k|$. We call P_k a (v_1, v_{k+1})-path.
- The extension of P_k denoted $C_{k+1} = (V, E \cup \{(v_{k+1}, v_1)\}$, $(v_{k+1}, v_1) \notin E$, is a (directed) cycle/circle of length $k + 1$.
- The graphs P_k and C_{k+1} are called simple if $|V| = k + 1$, i.e., no vertex appears twice on the path or in the cycle.
- An undirected path or cycle is the corresponding undirected graph. We also use the name for all directed graphs that result from switching the direction of arbitrarily many edges.

Connected Components

- A graph $G = (V, E)$ is called strongly connected if it contains a (v, w)-path and a (w, v)-path, for every pair $v, w \in V$.
- G is weakly connected if the symmetric graph $G' = (V, E')$ with $E' = E \cup \{(w, v) \mid (v, w) \in E\}$ is strongly connected.
- A simple undirected graph is k-connected (k-node connected) if removal of at most $k - 1$ arbitrary vertices (and all incident edges) keeps the resulting graph connected. The graph is k-edge connected if removal of $k - 1$ arbitrary edges keeps the resulting graph connected.
- For a simple graph, an inclusion-maximal weakly connected (strongly connected, k-connected, k-edge connected) subgraph is called weakly (strongly, k-, k-edge) connected component. For weakly connected component we often simply use connected component.

Connected Components

Determining connected components is fundamental in many applications analyzing the structure of networks. Powerful and fast algorithms for computing such components can be given as specialization of the following general framework based on depth-first search (DFS).

The framework uses the following input and data structures:

Input:	Directed Graph $G = (V, E)$
Data Structures:	Stack S (for vertices on the DFS path)
	Vertex array `incoming` (for first incoming edge)
	Vertex and Edge Markings

In the following, the functions root, traverse and backtrack are implemented differently depending on the type of component under consideration.

Directed Depth-First Search (DFS)

foreach $s \in V$ **do**
  if s *is unmarked* **then**
    mark s, set `incoming`$[s] \leftarrow$ `nil`
    push $s \rightarrow S$
    $\longrightarrow$ root(s)
    while S *not empty* **do**
      $v \leftarrow top(S)$
      if $\exists$ *unmarked* $\boxed{e = (v, w) \in E}$ **then**
        mark e
        if w *unmarked* **then**
          mark w, set `incoming`$[w] \leftarrow e$
          push $w \rightarrow S$
        $\longrightarrow$ traverse(v, e, w)
      else
        $w \leftarrow pop(S)$
        $\longrightarrow$ backtrack(w, `incoming`$[w]$, $top(S)$)

DFS Numbers

Let $v_1, \ldots, v_n$ be the sequence in which vertices are marked.

- $DFS(v_i) = i$ is the DFS-number of v_i
- The DFS-number $DFS((v, w)) = DFS(v)$ of edge (v, w) is the DFS-number of the vertex from which it was traversed.
- We define the DFS order $\preceq$ on $V \cup E$ by

$$DFS(p) \leq DFS(q) \quad \Leftrightarrow \quad p \preceq q \qquad \text{for all } p, q \in V \cup E$$

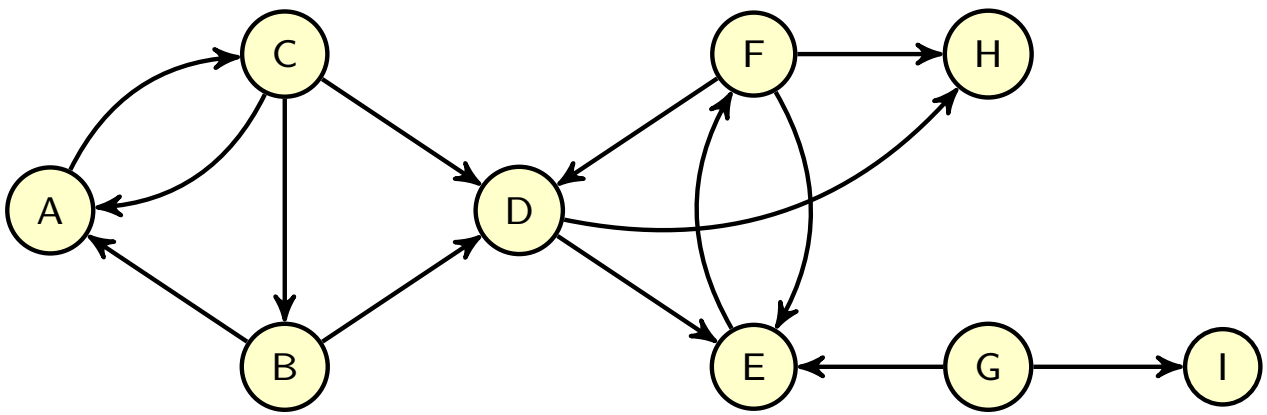
$$\text{and} \quad DFS(p) < DFS(q) \quad \Leftrightarrow \quad p \prec q \qquad \text{for all } p, q \in V \cup E.$$

Edges

The edges are classified as follows. If an edge (v, w) is marked, it becomes a

- tree edge if w is unmarked
- back edge if w is marked, $w \preceq v$ and $w \in S$,
- cross edge if w is marked, $w \prec v$ and $w \notin S$, and
- forward edge if w is marked and $v \prec w$.

Example:

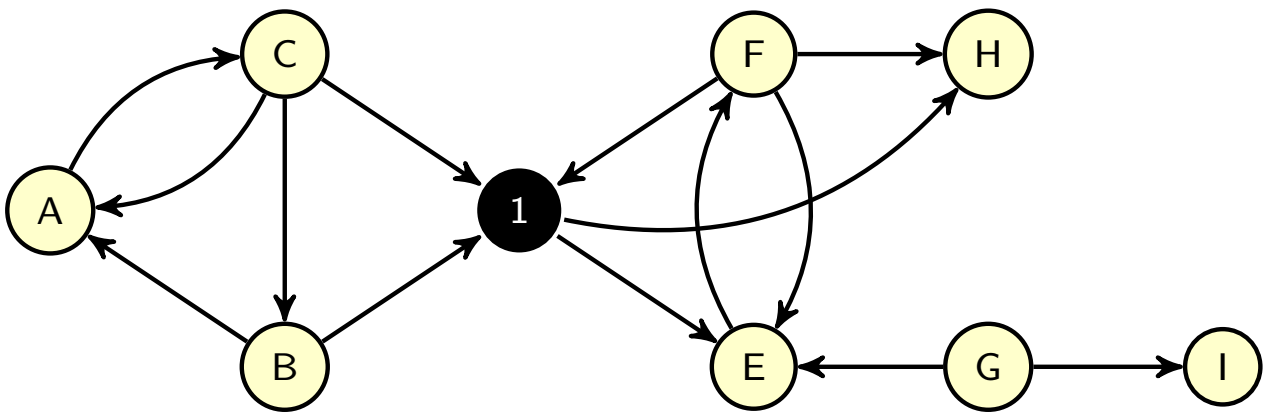


Edges

The edges are classified as follows. If an edge (v, w) is marked, it becomes a

- tree edge if w is unmarked
- back edge if w is marked, $w \preceq v$ and $w \in S$,
- cross edge if w is marked, $w \prec v$ and $w \notin S$, and
- forward edge if w is marked and $v \prec w$.

Example:

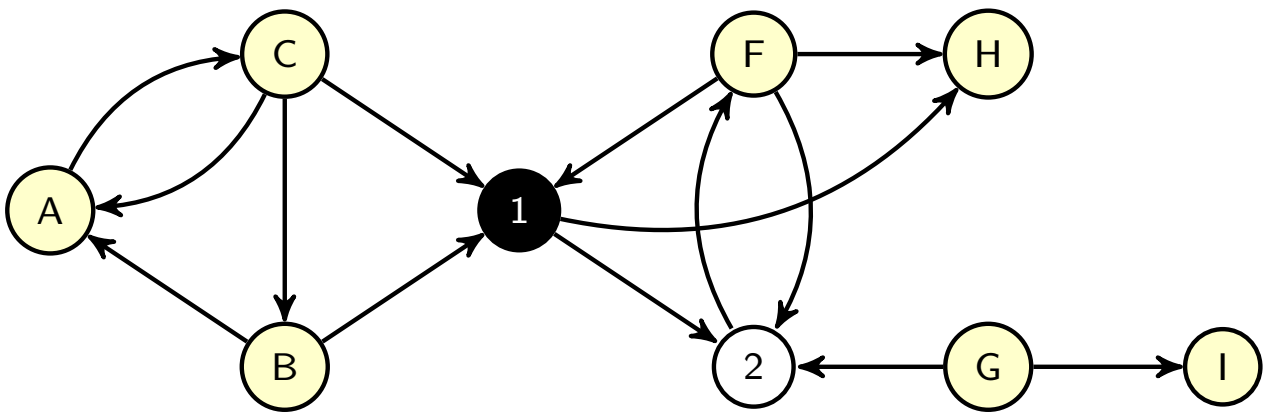


Edges

The edges are classified as follows. If an edge (v, w) is marked, it becomes a

- tree edge if w is unmarked
- back edge if w is marked, $w \preceq v$ and $w \in S$,
- cross edge if w is marked, $w \prec v$ and $w \notin S$, and
- forward edge if w is marked and $v \prec w$.

Example:

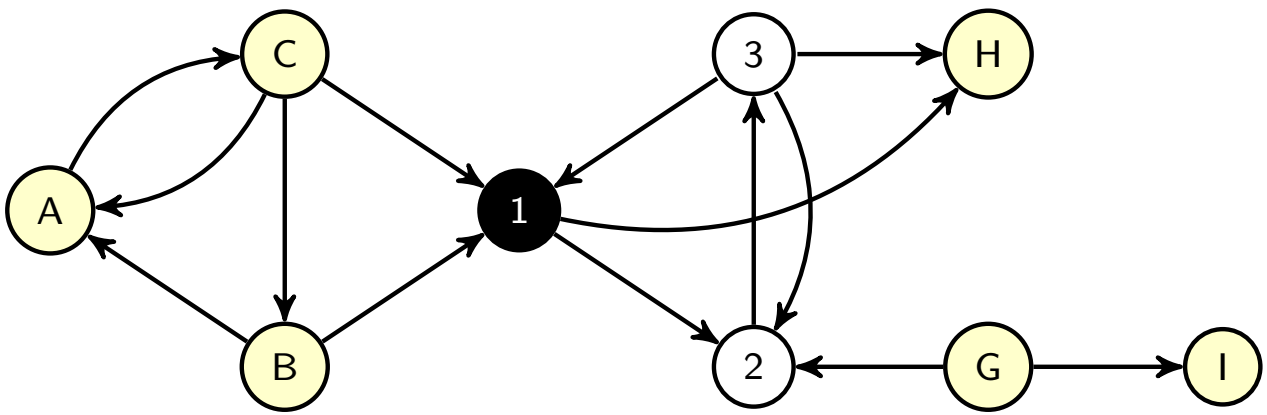


Edges

The edges are classified as follows. If an edge (v, w) is marked, it becomes a

- tree edge if w is unmarked
- back edge if w is marked, $w \preceq v$ and $w \in S$,
- cross edge if w is marked, $w \prec v$ and $w \notin S$, and
- forward edge if w is marked and $v \prec w$.

Example:

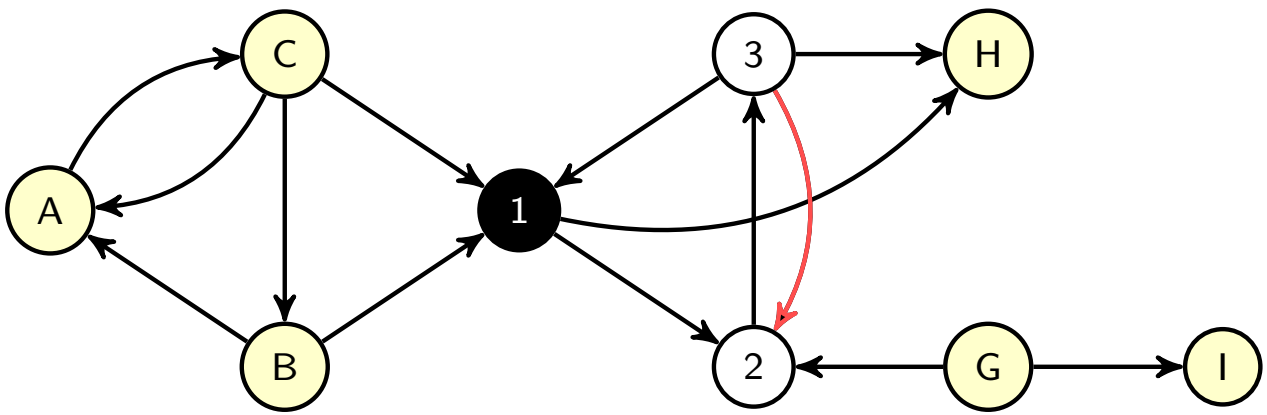


Edges

The edges are classified as follows. If an edge (v, w) is marked, it becomes a

- tree edge if w is unmarked
- back edge if w is marked, $w \preceq v$ and $w \in S$,
- cross edge if w is marked, $w \prec v$ and $w \notin S$, and
- forward edge if w is marked and $v \prec w$.

Example:

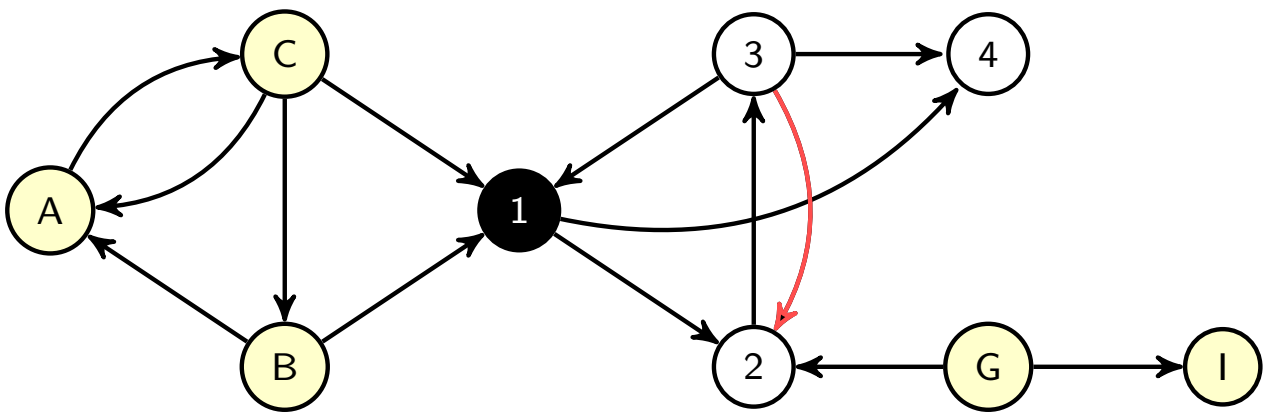


Edges

The edges are classified as follows. If an edge (v, w) is marked, it becomes a

- tree edge if w is unmarked
- back edge if w is marked, $w \preceq v$ and $w \in S$,
- cross edge if w is marked, $w \prec v$ and $w \notin S$, and
- forward edge if w is marked and $v \prec w$.

Example:

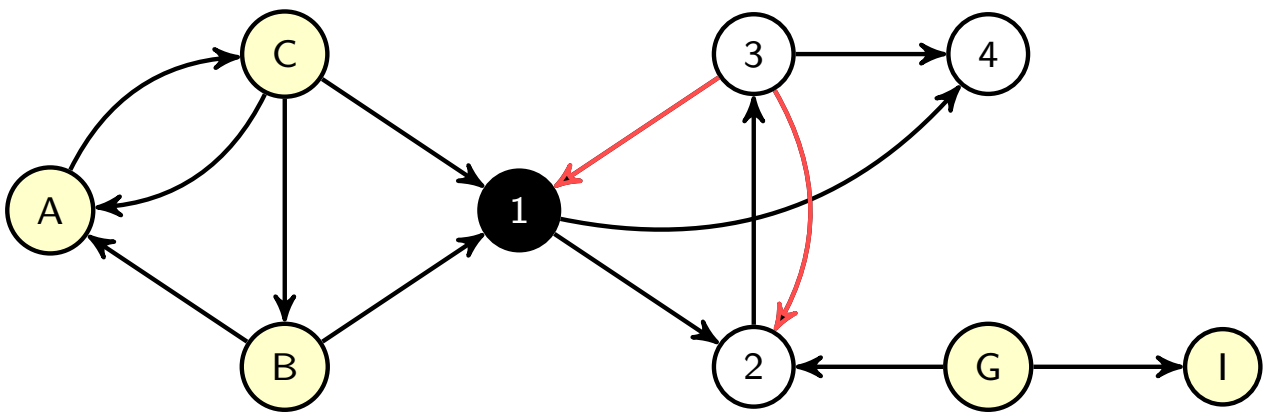


Edges

The edges are classified as follows. If an edge (v, w) is marked, it becomes a

- tree edge if w is unmarked
- back edge if w is marked, $w \preceq v$ and $w \in S$,
- cross edge if w is marked, $w \prec v$ and $w \notin S$, and
- forward edge if w is marked and $v \prec w$.

Example:

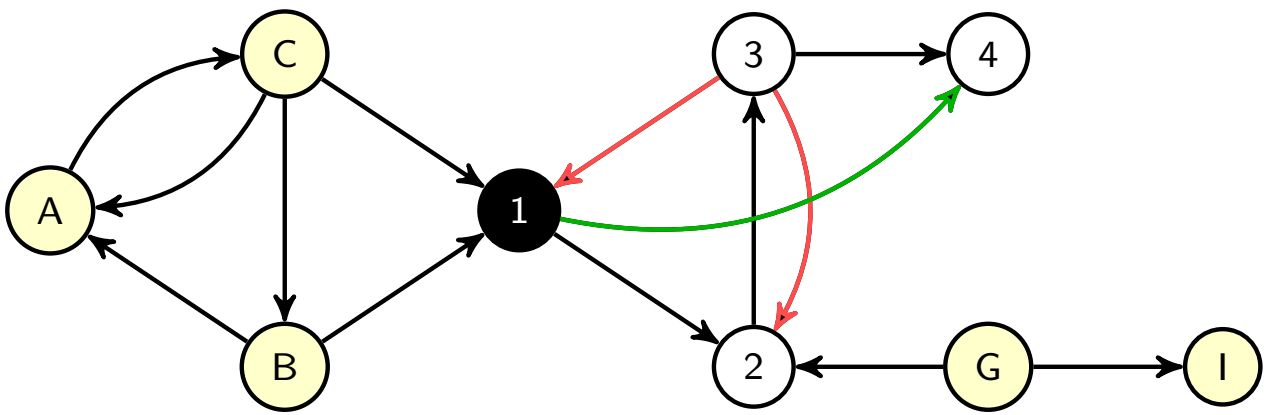


Edges

The edges are classified as follows. If an edge (v, w) is marked, it becomes a

- tree edge if w is unmarked
- back edge if w is marked, $w \preceq v$ and $w \in S$,
- cross edge if w is marked, $w \prec v$ and $w \notin S$, and
- forward edge if w is marked and $v \prec w$.

Example:

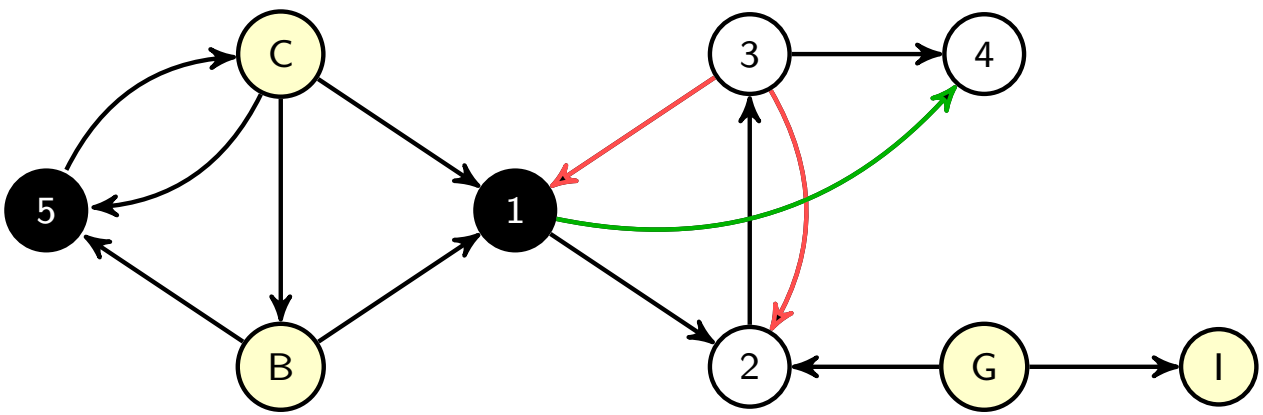


Edges

The edges are classified as follows. If an edge (v, w) is marked, it becomes a

- tree edge if w is unmarked
- back edge if w is marked, $w \preceq v$ and $w \in S$,
- cross edge if w is marked, $w \prec v$ and $w \notin S$, and
- forward edge if w is marked and $v \prec w$.

Example:

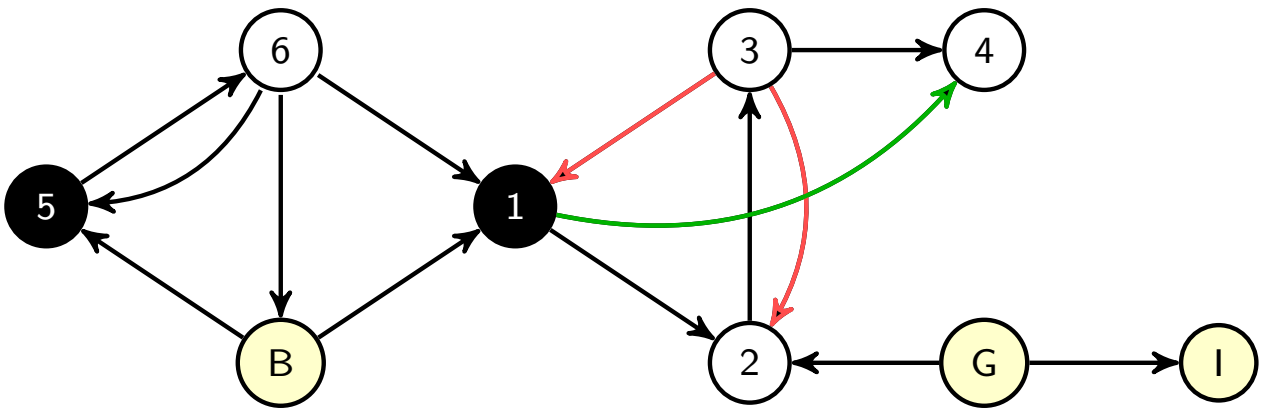


Edges

The edges are classified as follows. If an edge (v, w) is marked, it becomes a

- tree edge if w is unmarked
- back edge if w is marked, $w \preceq v$ and $w \in S$,
- cross edge if w is marked, $w \prec v$ and $w \notin S$, and
- forward edge if w is marked and $v \prec w$.

Example:

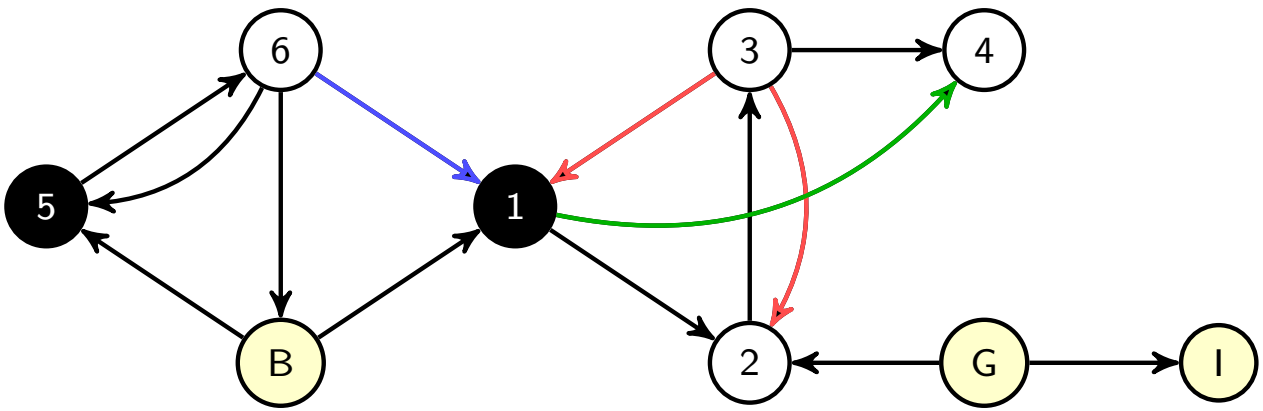


Edges

The edges are classified as follows. If an edge (v, w) is marked, it becomes a

- tree edge if w is unmarked
- back edge if w is marked, $w \preceq v$ and $w \in S$,
- cross edge if w is marked, $w \prec v$ and $w \notin S$, and
- forward edge if w is marked and $v \prec w$.

Example:

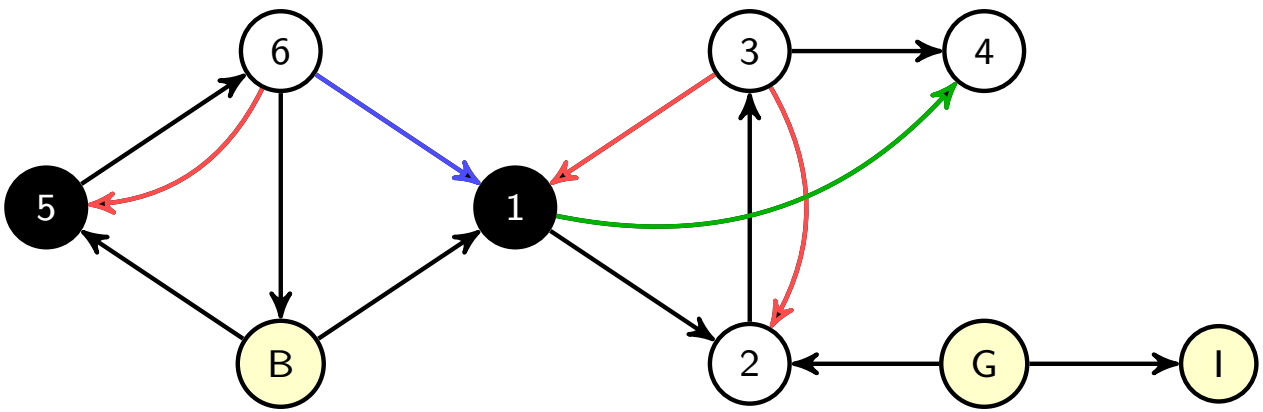


Edges

The edges are classified as follows. If an edge (v, w) is marked, it becomes a

- tree edge if w is unmarked
- back edge if w is marked, $w \preceq v$ and $w \in S$,
- cross edge if w is marked, $w \prec v$ and $w \notin S$, and
- forward edge if w is marked and $v \prec w$.

Example:

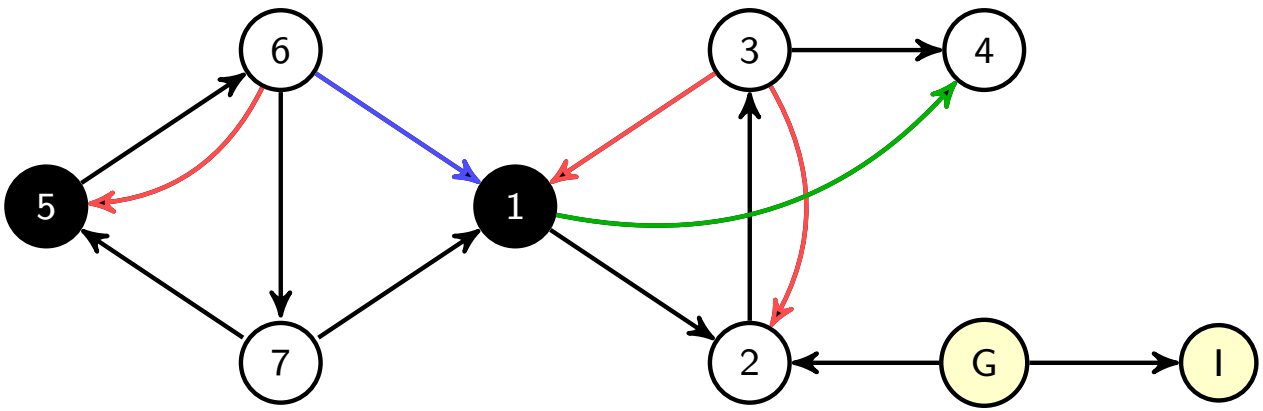


Edges

The edges are classified as follows. If an edge (v, w) is marked, it becomes a

- tree edge if w is unmarked
- back edge if w is marked, $w \preceq v$ and $w \in S$,
- cross edge if w is marked, $w \prec v$ and $w \notin S$, and
- forward edge if w is marked and $v \prec w$.

Example:

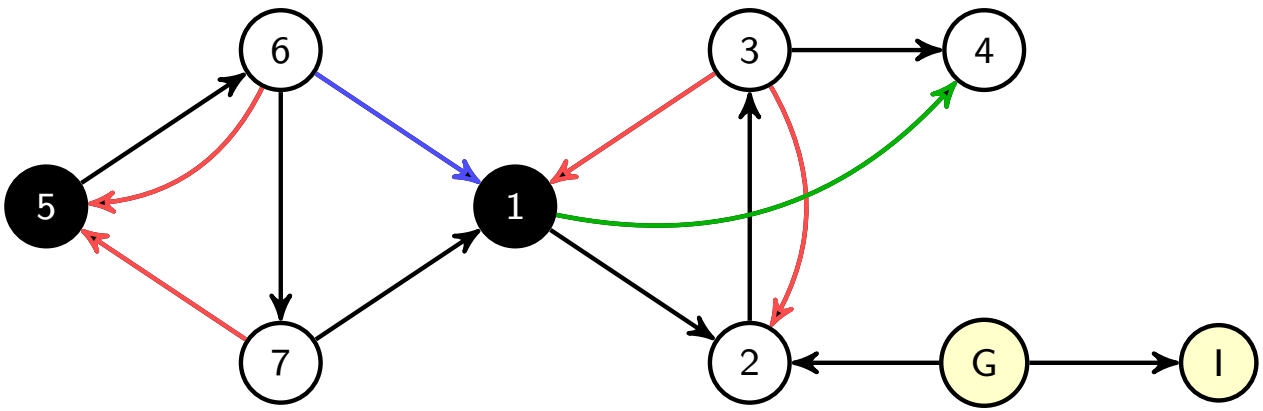


Edges

The edges are classified as follows. If an edge (v, w) is marked, it becomes a

- tree edge if w is unmarked
- back edge if w is marked, $w \preceq v$ and $w \in S$,
- cross edge if w is marked, $w \prec v$ and $w \notin S$, and
- forward edge if w is marked and $v \prec w$.

Example:

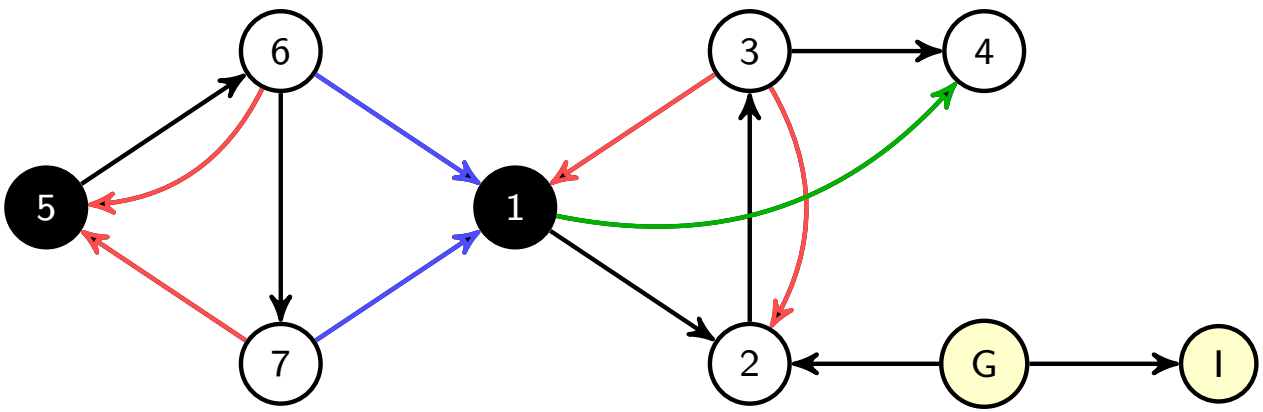


Edges

The edges are classified as follows. If an edge (v, w) is marked, it becomes a

- tree edge if w is unmarked
- back edge if w is marked, $w \preceq v$ and $w \in S$,
- cross edge if w is marked, $w \prec v$ and $w \notin S$, and
- forward edge if w is marked and $v \prec w$.

Example:

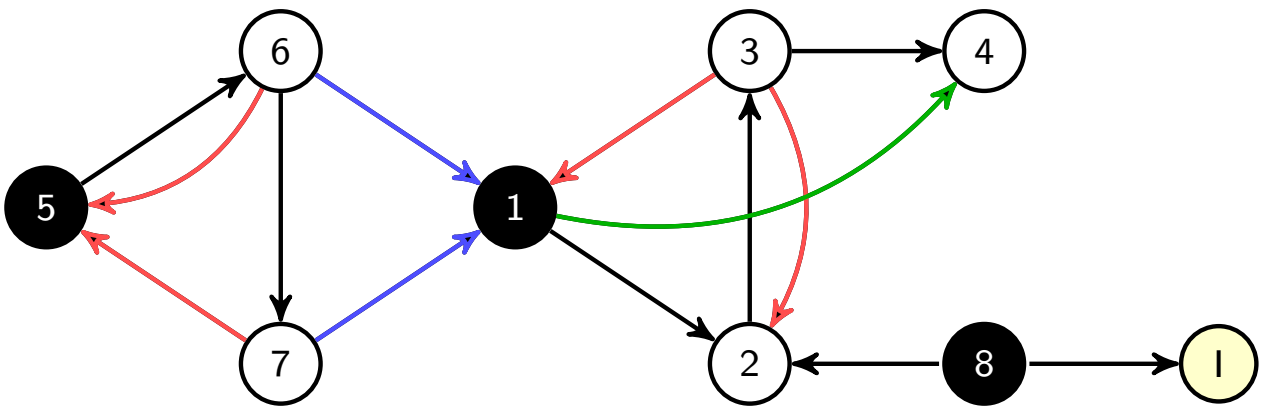


Edges

The edges are classified as follows. If an edge (v, w) is marked, it becomes a

- tree edge if w is unmarked
- back edge if w is marked, $w \preceq v$ and $w \in S$,
- cross edge if w is marked, $w \prec v$ and $w \notin S$, and
- forward edge if w is marked and $v \prec w$.

Example:

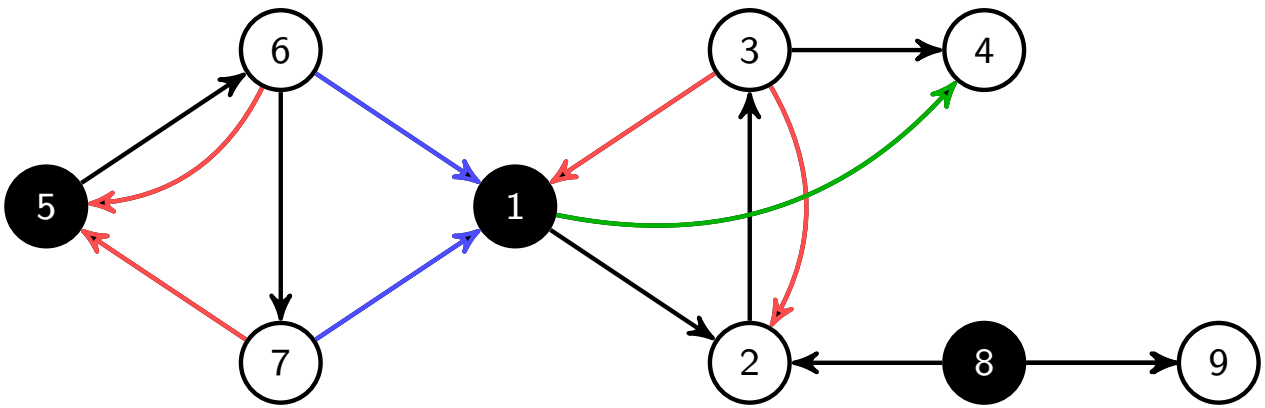


Edges

The edges are classified as follows. If an edge (v, w) is marked, it becomes a

- tree edge if w is unmarked
- back edge if w is marked, $w \preceq v$ and $w \in S$,
- cross edge if w is marked, $w \prec v$ and $w \notin S$, and
- forward edge if w is marked and $v \prec w$.

Example:

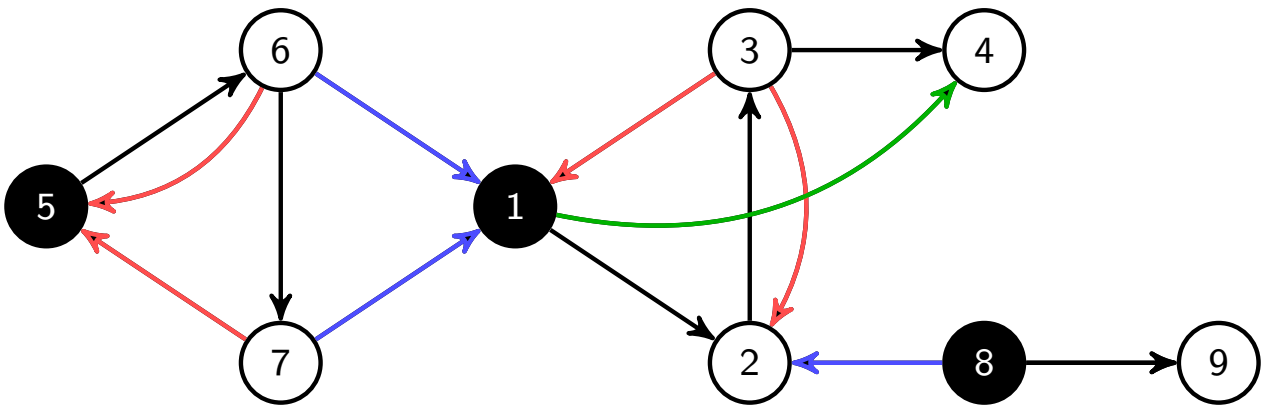


Edges

The edges are classified as follows. If an edge (v, w) is marked, it becomes a

- tree edge if w is unmarked
- back edge if w is marked, $w \preceq v$ and $w \in S$,
- cross edge if w is marked, $w \prec v$ and $w \notin S$, and
- forward edge if w is marked and $v \prec w$.

Example:

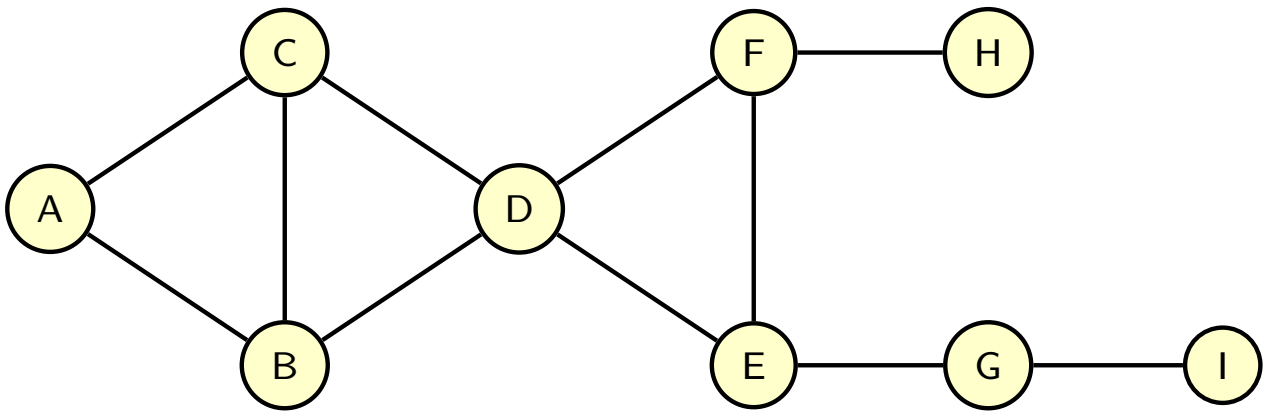


Example

When replacing the box in the framework by

$$\boxed{e = (v, w) \in E \textbf{ or } e = (w, v) \in E} \ ,$$

we get undirected DFS where edges are traversed independently of their direction.

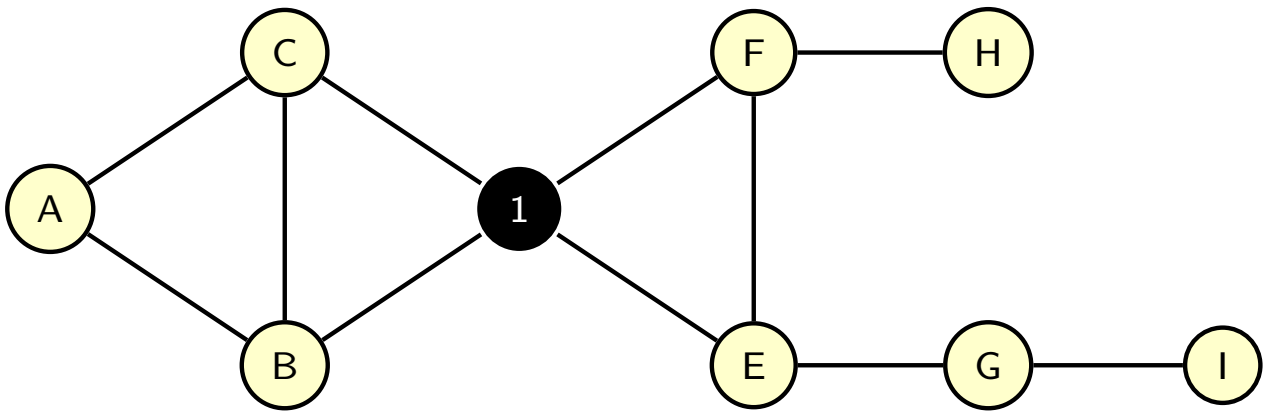


Example

When replacing the box in the framework by

$$\boxed{e = (v, w) \in E \textbf{ or } e = (w, v) \in E} \ ,$$

we get undirected DFS where edges are traversed independently of their direction.

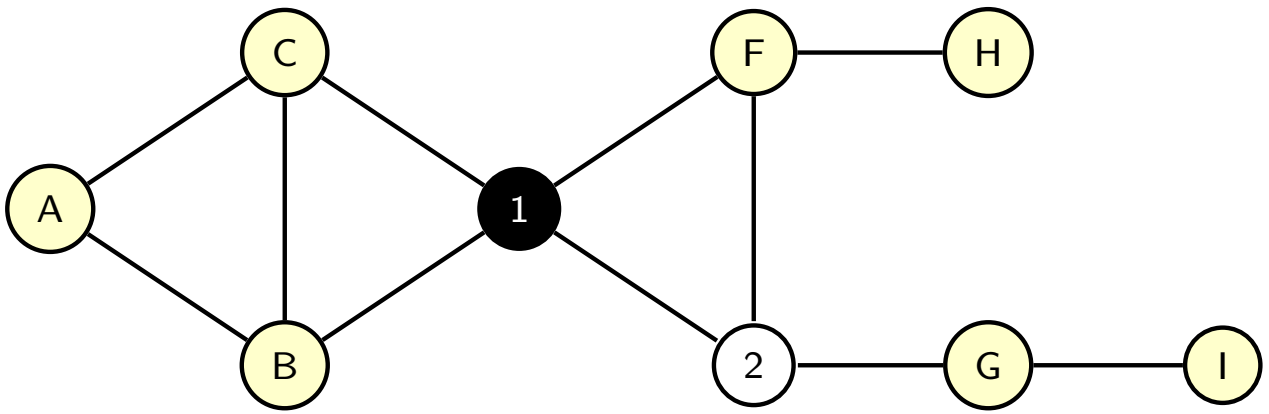


Example

When replacing the box in the framework by

$$\boxed{e = (v, w) \in E \textbf{ or } e = (w, v) \in E} \ ,$$

we get undirected DFS where edges are traversed independently of their direction.

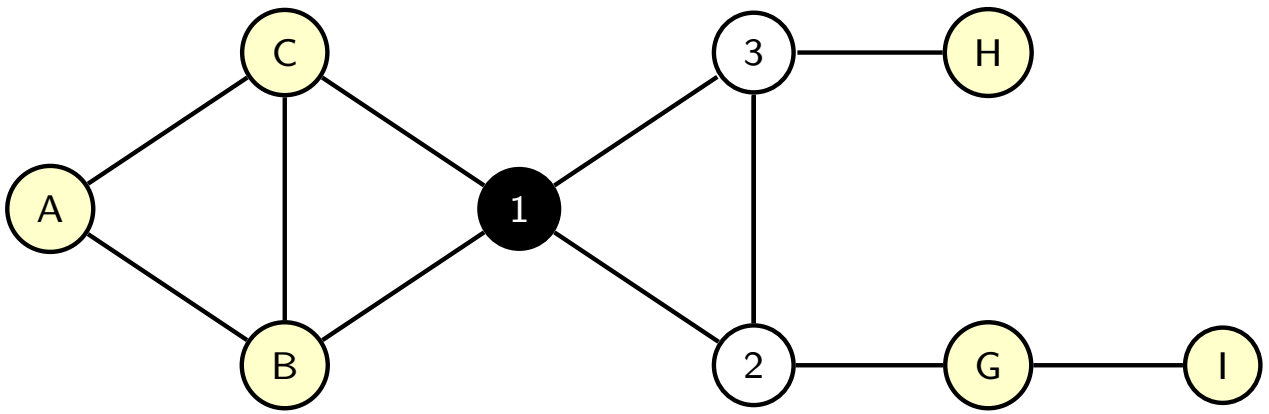


Example

When replacing the box in the framework by

$$\boxed{e = (v, w) \in E \textbf{ or } e = (w, v) \in E} \ ,$$

we get undirected DFS where edges are traversed independently of their direction.

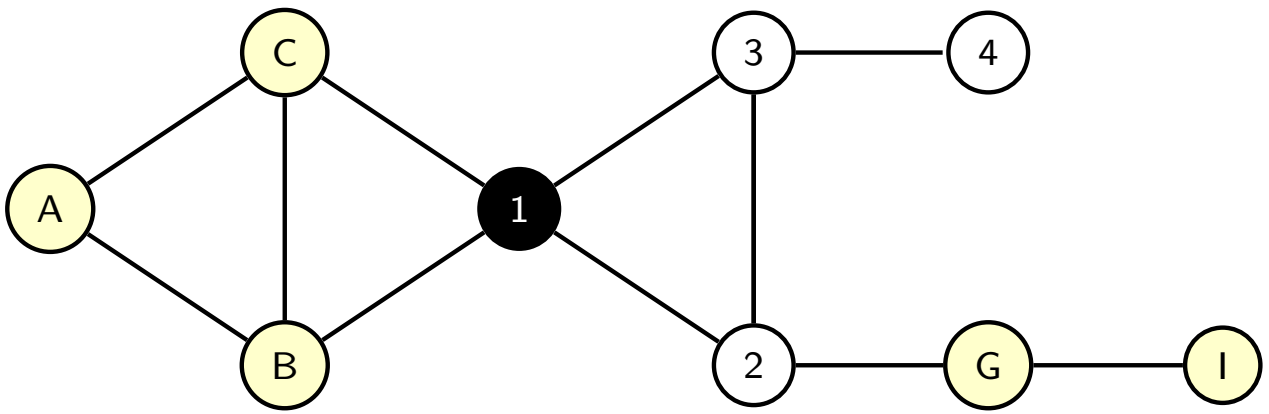


Example

When replacing the box in the framework by

$$\boxed{e = (v, w) \in E \textbf{ or } e = (w, v) \in E} \ ,$$

we get undirected DFS where edges are traversed independently of their direction.

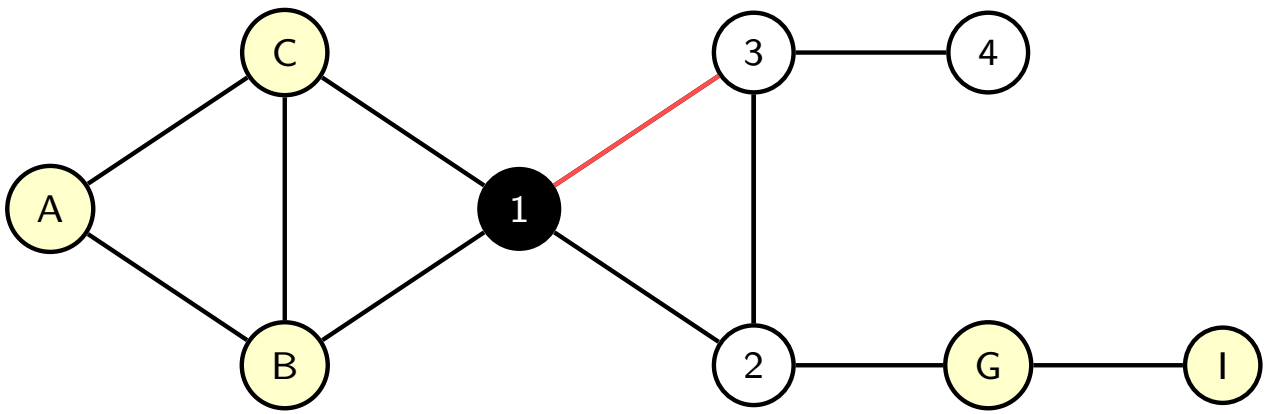


Example

When replacing the box in the framework by

$$\boxed{e = (v, w) \in E \textbf{ or } e = (w, v) \in E} \ ,$$

we get undirected DFS where edges are traversed independently of their direction.

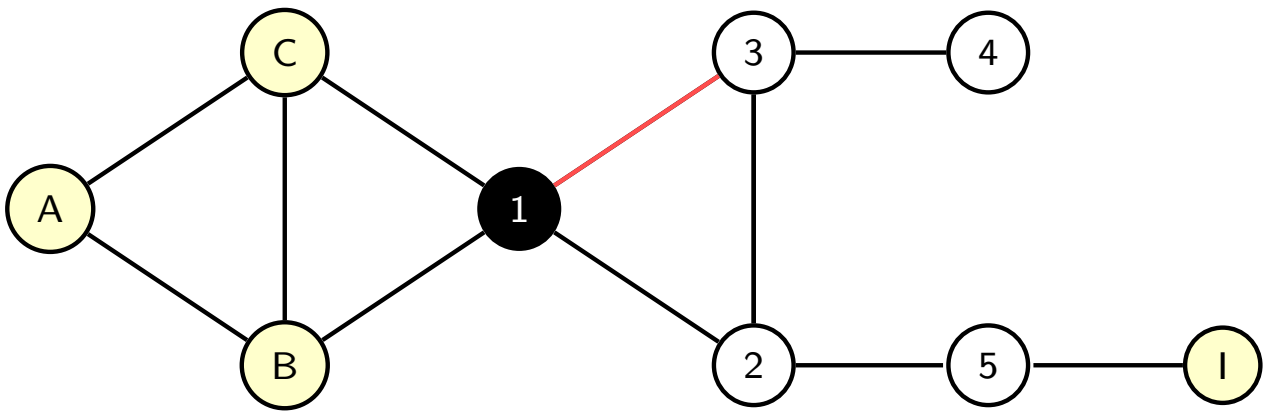


Example

When replacing the box in the framework by

$$\boxed{e = (v, w) \in E \textbf{ or } e = (w, v) \in E} \ ,$$

we get undirected DFS where edges are traversed independently of their direction.

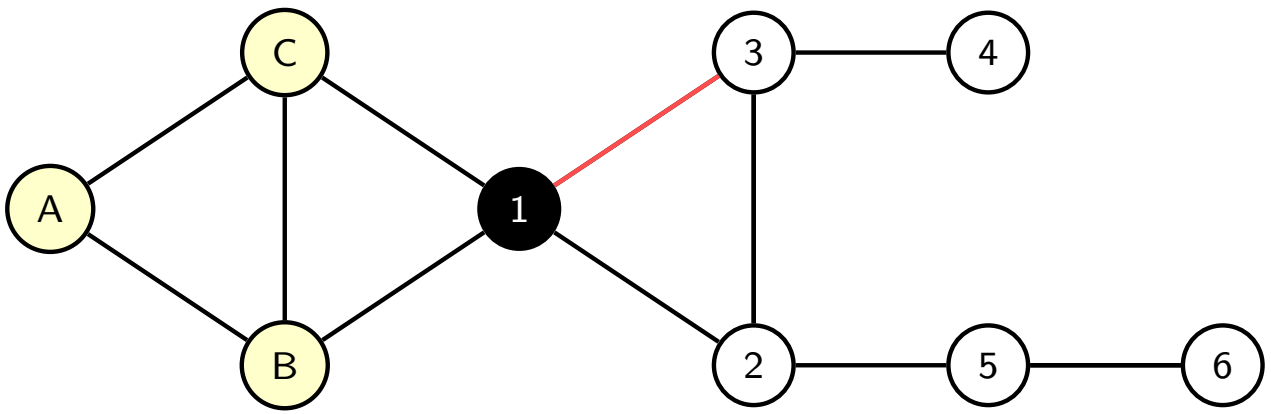


Example

When replacing the box in the framework by

$$\boxed{e = (v, w) \in E \textbf{ or } e = (w, v) \in E} \ ,$$

we get undirected DFS where edges are traversed independently of their direction.

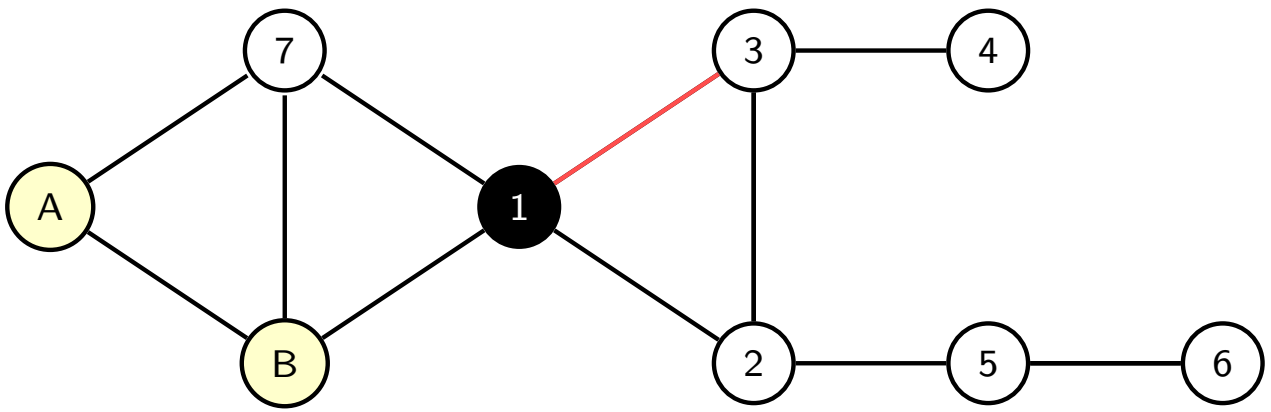


Example

When replacing the box in the framework by

$$\boxed{e = (v, w) \in E \textbf{ or } e = (w, v) \in E} \ ,$$

we get undirected DFS where edges are traversed independently of their direction.

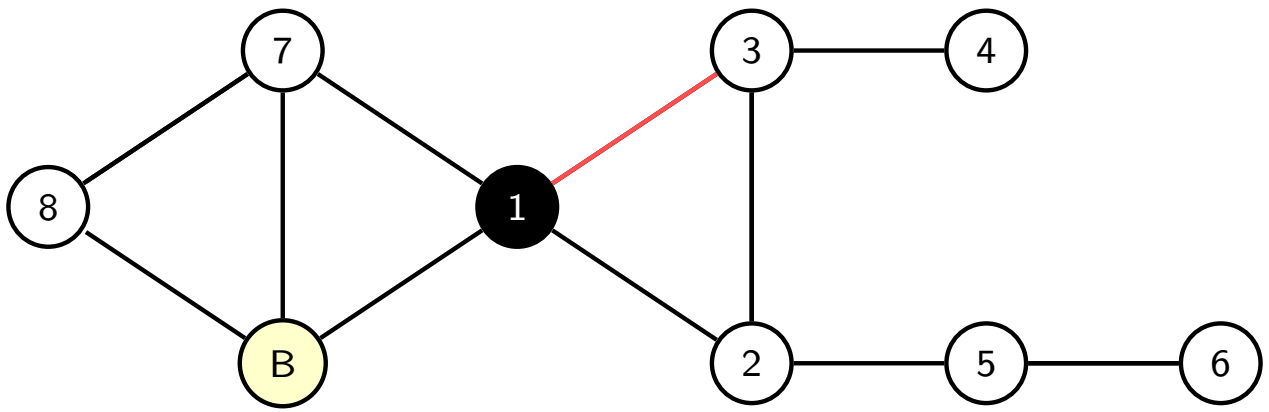


Example

When replacing the box in the framework by

$$\boxed{e = (v, w) \in E \textbf{ or } e = (w, v) \in E} \ ,$$

we get undirected DFS where edges are traversed independently of their direction.

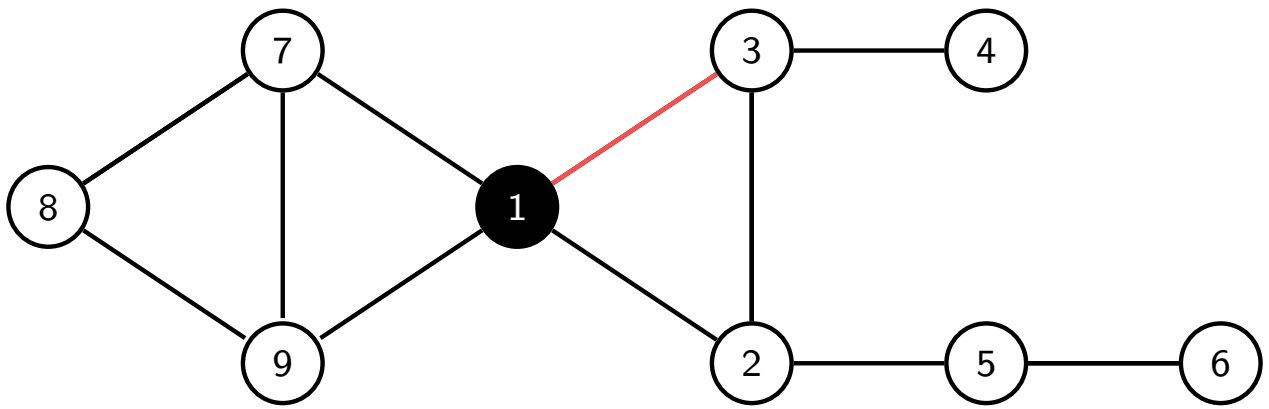


Example

When replacing the box in the framework by

$$\boxed{e = (v, w) \in E \textbf{ or } e = (w, v) \in E} \ ,$$

we get undirected DFS where edges are traversed independently of their direction.

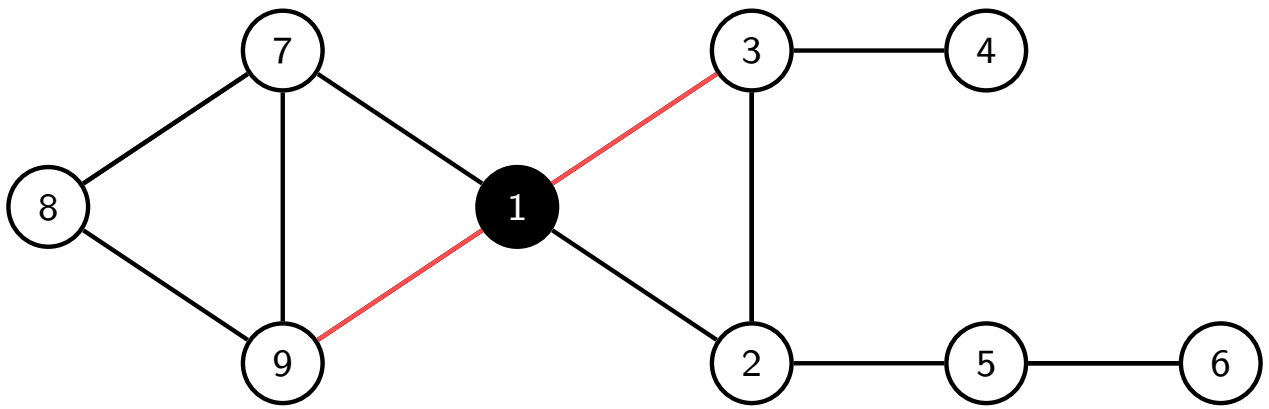


Example

When replacing the box in the framework by

$$\boxed{e = (v, w) \in E \textbf{ or } e = (w, v) \in E} \ ,$$

we get undirected DFS where edges are traversed independently of their direction.

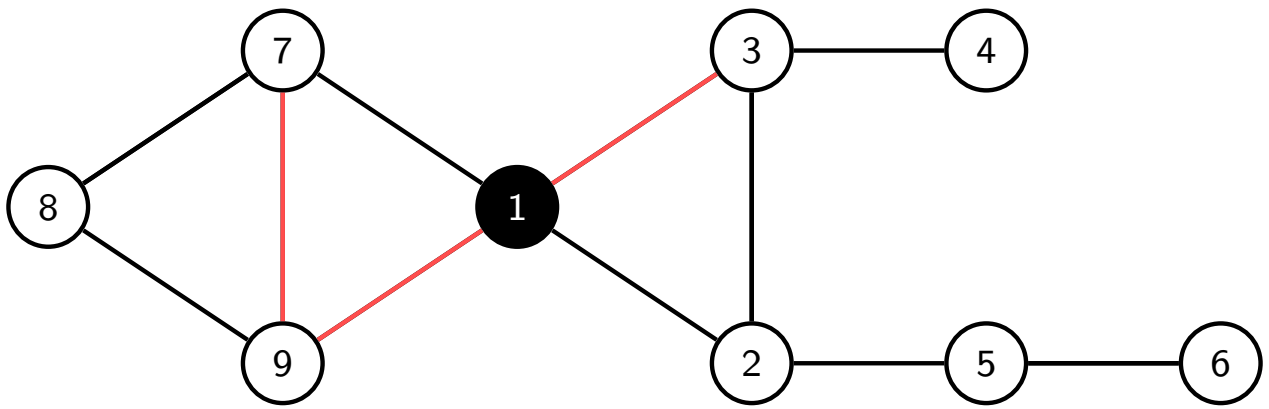


Example

When replacing the box in the framework by

$$\boxed{e = (v, w) \in E \textbf{ or } e = (w, v) \in E} \ ,$$

we get undirected DFS where edges are traversed independently of their direction.

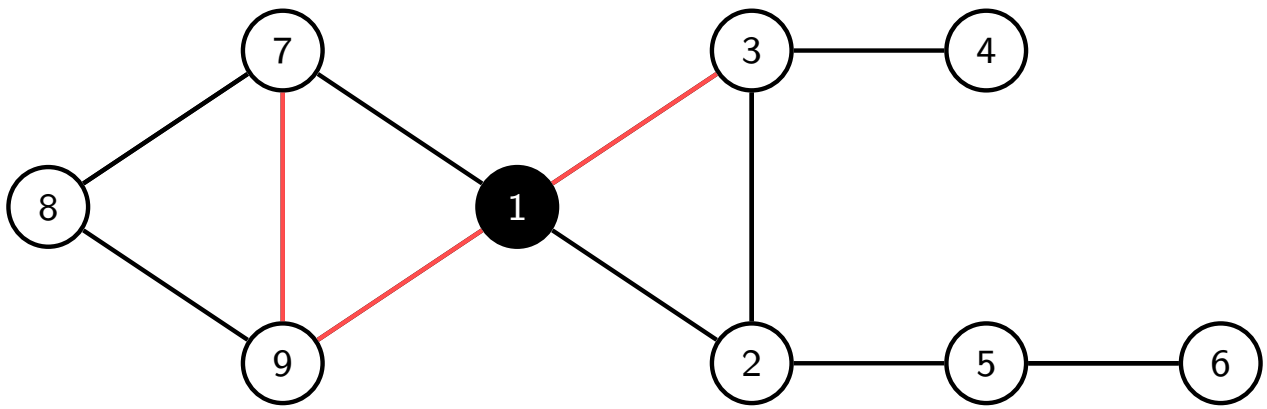


Example

When replacing the box in the framework by

$$\boxed{e = (v, w) \in E \textbf{ or } e = (w, v) \in E} \ ,$$

we get undirected DFS where edges are traversed independently of their direction.



In undirected DFS there are neither cross nor forward edges.

(Weakly) Connected Components

The following algorithms all use the DFS framework. They can all be implemented in $O(n+m)$ time and $O(n+m)$ space. We thus concentrate only on the implementation of the missing functions. For (weakly) connected components, the functions are straightforward.

A component c is represented by the first vertex. The output is a vertex- and edge array `component` that points to the representing vertices.

root(s)
  $c \leftarrow s$; $\texttt{component}[s] \leftarrow c$

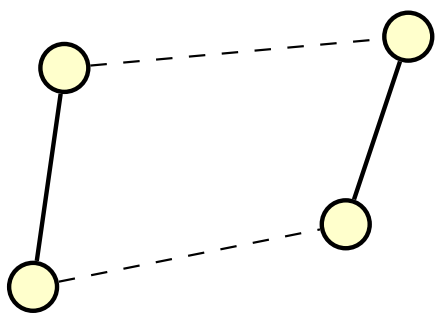
traverse(v, e, w)
  $\texttt{component}[e] \leftarrow c$
  $\texttt{component}[w] \leftarrow c$

2-Connected Components

Lemma

For an undirected graph G, edges $e, e' \in E$ are in a 2-connected component if and only if there is a simple undirected cycle that contains both e and e'. The set of 2-connected components forms a partition of E.

Proof:
Obviously, if the cycle exists, e and e' are in a 2-connected component.
For the other direction, consider e and e' and subpartition each by a new vertex. The resulting component is still 2-connected. The existence of a simple undirected cycle comes from *Menger's Theorem*: In a 2-connected undirected multigraph with at least two edges there are two vertex-disjoint paths between every pair of vertices.

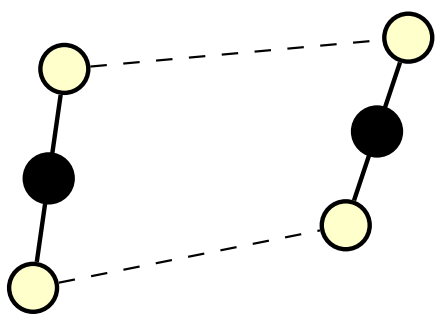


2-Connected Components

Lemma

For an undirected graph G, edges $e, e' \in E$ are in a 2-connected component if and only if there is a simple undirected cycle that contains both e and e'. The set of 2-connected components forms a partition of E.

Proof:
Obviously, if the cycle exists, e and e' are in a 2-connected component.
For the other direction, consider e and e' and subpartition each by a new vertex. The resulting component is still 2-connected. The existence of a simple undirected cycle comes from *Menger's Theorem*: In a 2-connected undirected multigraph with at least two edges there are two vertex-disjoint paths between every pair of vertices.



Proof (contd)

To show the partition statement, we contradict the maximality of components. If an edge is in two 2-connected components, the existence of a simple undirected cycle to each edge from every component implies that there is a simple cycle for *every pair* of edges from the union of the components. Thus, the union of the components is 2-connected as well, which contradicts the maximality. □

Undirected DFS for 2-Connected Components

We use a stack S_E for edges in unfinished (called "open") components and stack C for open components (represented by first edge). Output is array `component` pointing to the respresenting edges.

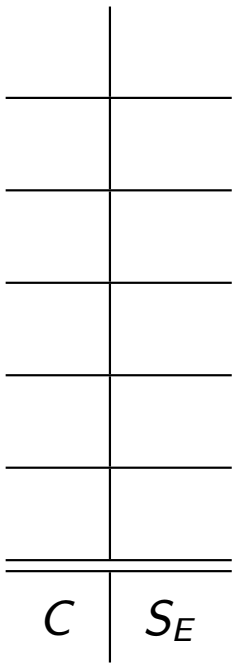
```
traverse(v, e, w)
    if e is loop then
        component[e] ← e
    else
        push e → S_E
        if e is tree edge then
            push e → C
        if e is back edge then
            while w ≺ top(C) do
                pop(C)
```

```
backtrack(w, e, v)
    if e = top(C) and e ≠ nil then
        pop(C)
        repeat
            e' ← pop(S_E)
            component[e'] ← e
        until e' = e
```

Example

1

Stacks

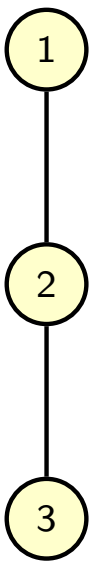


Example

Stacks after traverse(1,(1,2),2)

(1,2)	(1,2)
C	S_E

Example



Stacks after traverse(2,(2,3),3)

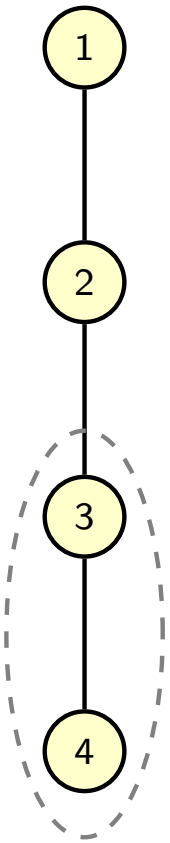
(2,3)	(2,3)
(1,2)	(1,2)
C	S_E

Example

Stacks after traverse(3,(3,4),4)

(3,4)	(3,4)
(2,3)	(2,3)
(1,2)	(1,2)
C	S_E

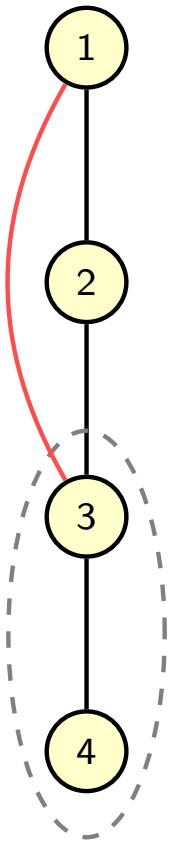
Example



Stacks after backtrack(4,(3,4),3)

(2,3)	(2,3)
(1,2)	(1,2)
C	S_E

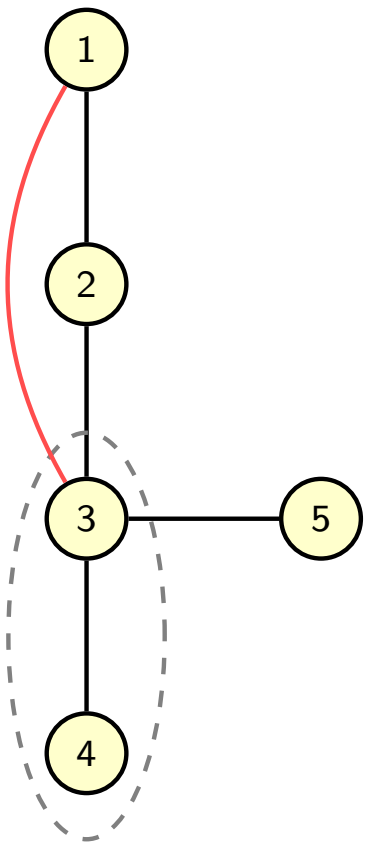
Example



Stacks after traverse(3,(3,1),1)

	(3,1)
	(2,3)
(1,2)	(1,2)
C	S_E

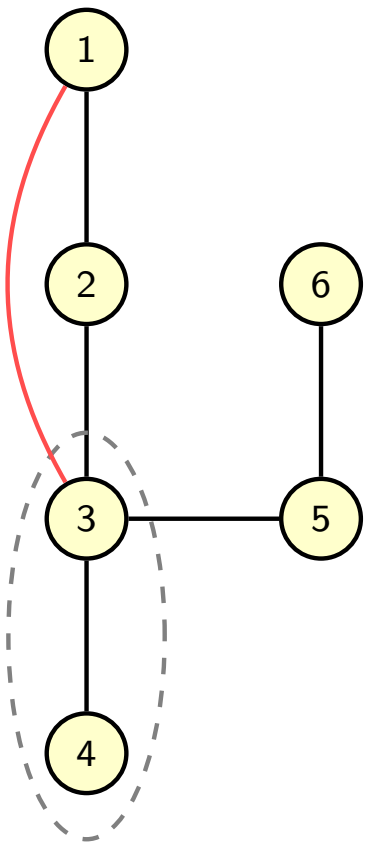
Example



Stacks after traverse(3,(3,5),5)

	(3,5)
	(3,1)
(3,5)	(2,3)
(1,2)	(1,2)
C	S_E

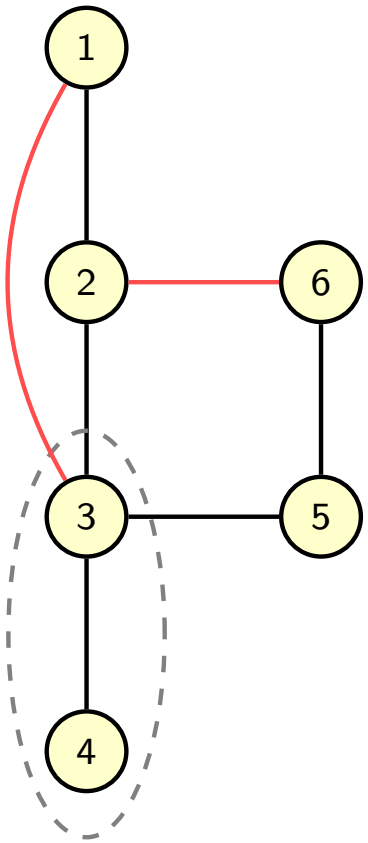
Example



Stacks after traverse(5,(5,6),6)

	(5,6)
	(3,5)
(5,6)	(3,1)
(3,5)	(2,3)
(1,2)	(1,2)
C	S_E

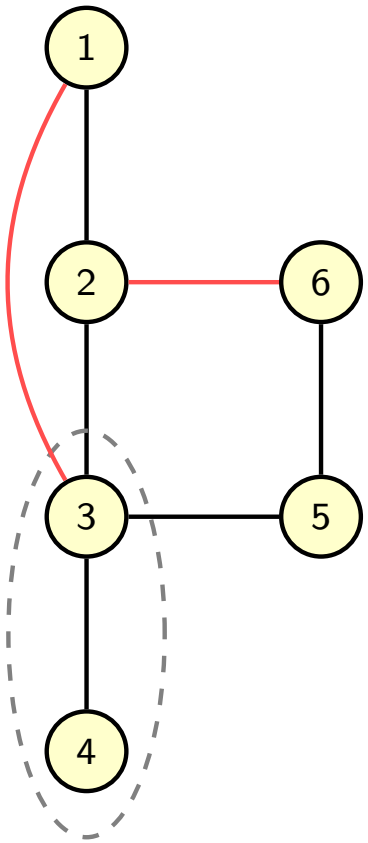
Example



Stacks after traverse(6,(6,2),2)

	(6,2)
	(5,6)
	(3,5)
	(3,1)
	(2,3)
(1,2)	(1,2)
C	S_E

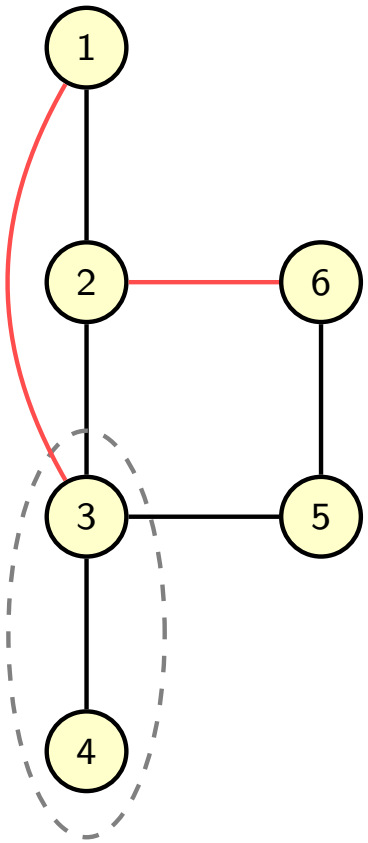
Example



Stacks after backtrack(6,(5,6),5)

	(6,2)
	(5,6)
	(3,5)
	(3,1)
	(2,3)
(1,2)	(1,2)
C	S_E

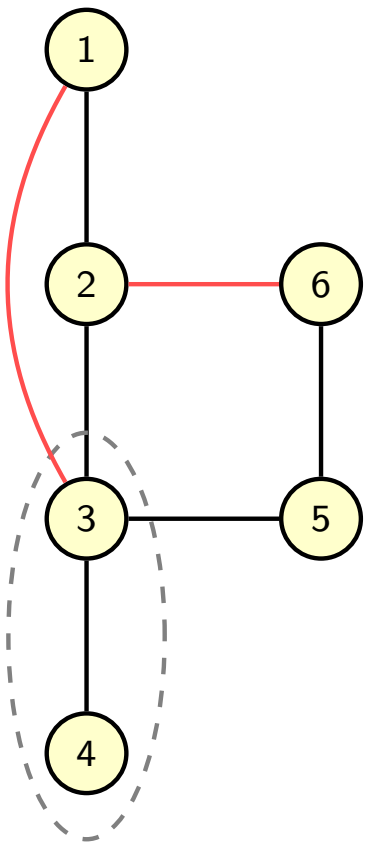
Example



Stacks after backtrack(5,(3,5),3)

	(6,2)
	(5,6)
	(3,5)
	(3,1)
	(2,3)
(1,2)	(1,2)
C	S_E

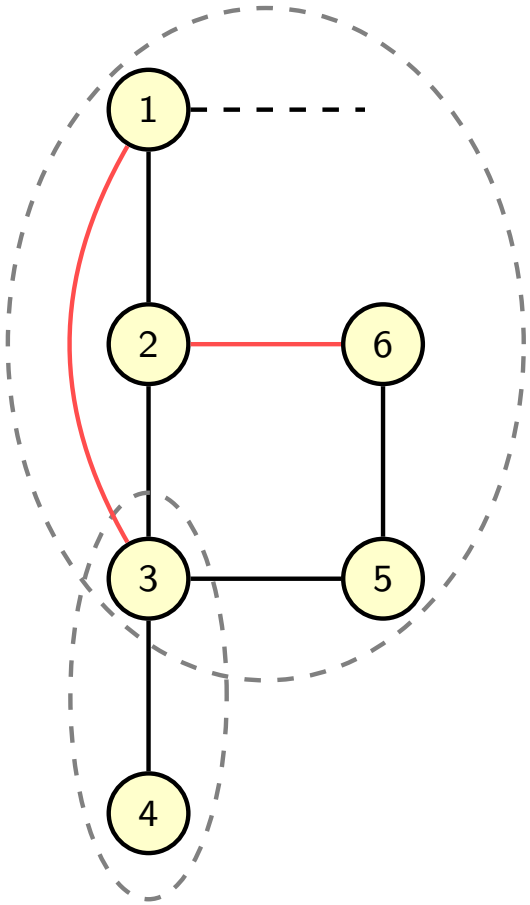
Example



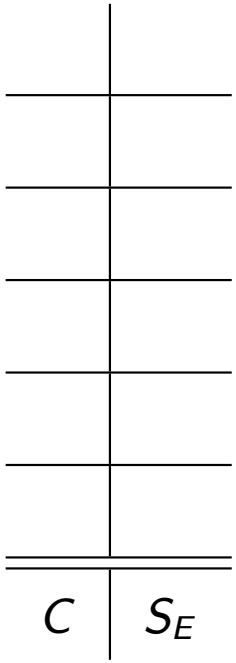
Stacks after backtrack(3,(2,3),2)

	(6,2)
	(5,6)
	(3,5)
	(3,1)
	(2,3)
(1,2)	(1,2)
C	S_E

Example



Stacks after backtrack(2,(1,2),1)
(...and so on)



Correctness and Running Time

Theorem

The DFS Algorithm computes the 2-connected components of an undirected graph in time $O(n+m)$.

Proof:

The DFS framework has linear running time, and for traverse and backtrack in total we also have linear running time. This follows by the fact every edge is placed at most once on S_E and C.

For correctness we use induction over the calls of traverse and backtrack. We call a marked tree edge open if there has been no backtracking over it, all other edges are closed. At any point, open edges induce a path. For every 2-connected component there is a unique first edge, and a component is open/closed if the first edge is open/closed.

Correctness Proof

After the first t calls of traverse and backtrack let G_t be the subgraph induced by marked edges. Denote open components, their edge sets and their first edges by $G_t^{(i)}$, $E_t^{(i)}$ and $e_t^{(i)}$, for $1 \leq i \leq k_t$, in the order in which $e_t^{(i)}$ were marked. We show the following invariants:

- The edges of a closed component point to their first edge.
- On C we have $e_t^{(1)} \prec \ldots \prec e_t^{(k_t)}$ in this order.
- On S_E we have the edges from $E_t^{(1)}, \ldots, E_t^{(k_t)}$ in this order.

All conditions are true in the beginning. Assume they hold until $t-1$ and consider step t in two cases.

Correctness Proof

Case 1: (Call of traverse(v, e, w))

- If e is a loop, becomes a separate component.
- If e is a tree edge, then w is a vertex of degree 1 in G_t and e the only edge of a new open component.

$\Rightarrow$ For these edges the invariants continue to hold.

- If e is a back edge but no loop, it closes a simple cycle by traversing tree edges up to w. All these edges belong to the same 2-connected component of G_t. As all edges e' with $w \prec e'$ are removed from C, the invariant also holds here.

Correctness Proof

Case 2: (Call of backtrack(w, e, v))
If e=`nil`, then w is a vertex that was put on S in the outer loop of the DFS. Thus, all edges of the (weakly) connected component and all 2-connected components were closed, hence C and S_E are empty.

Otherwise edge e becomes closed. For C containing the representative edges, e is the first edge of the open component with highest index if and only if it is on top of C. As there are no cross or forward edges in undirected DFS, the component cannot grow and is therefore closed. For S_E containing the edges of the component, the repeat-loop deletes the correct edges from the stack. □

Undirected DFS for 2-Edge Connected Components

We use a stack S_V for vertices in open components and stack C for open components (represented by first vertex). Output is array `component` pointing to the respresenting vertices.

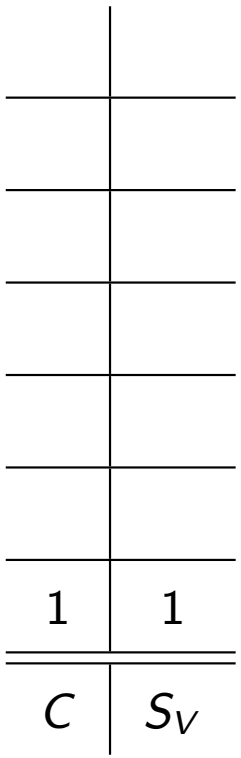
root(s)
- push $s \to S_V$
- push $s \to C$

traverse(v, e, w)
- **if** *e is tree edge* **then**
  - push $w \to S_V$
  - push $w \to C$
- **if** *e is back edge* **then**
  - **while** $w \prec top(C)$ **do**
    - $pop(C)$

backtrack(w, e, v)
- **if** $w = top(C)$ **then**
  - $pop(C)$
  - **repeat**
    - $u \leftarrow pop(S_V)$
    - `component`$[u] \leftarrow w$
  - **until** $u = w$

Example

Stacks after root(1)

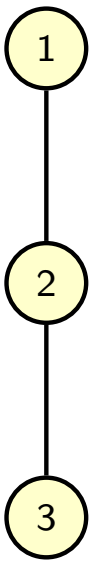


Example

Stacks after traverse(1,(1,2),2)

2	2
1	1
C	S_V

Example



Stacks after traverse(2,(2,3),3)

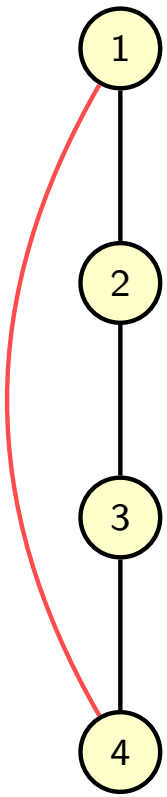
3	3
2	2
1	1
C	S_V

Example

Stacks after traverse(3,(3,4),4)

4	4
3	3
2	2
1	1
C	S_V

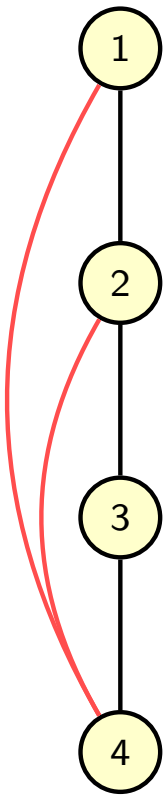
Example



Stacks after traverse(4,(4,1),1)

	4
	3
	2
1	1
C	S_V

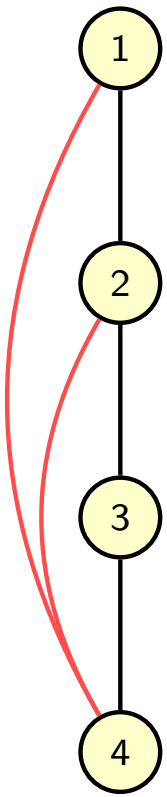
Example



Stacks after traverse(4,(4,2),2)

	4
	3
	2
1	1
C	S_V

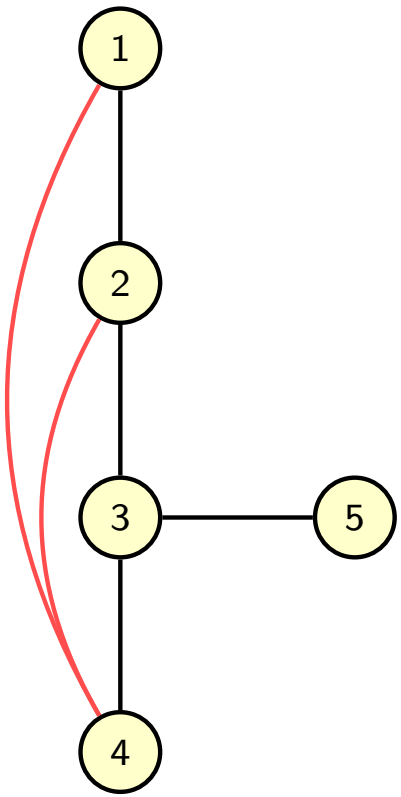
Example



Stacks after backtrack(4,(3,4),3)

	4
	3
	2
1	1
C	S_V

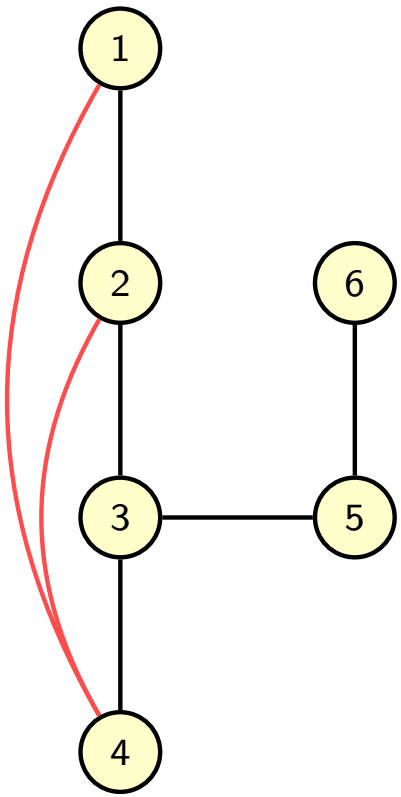
Example



Stacks after traverse(3,(3,5),5)

	5
	4
	3
5	2
1	1
C	S_V

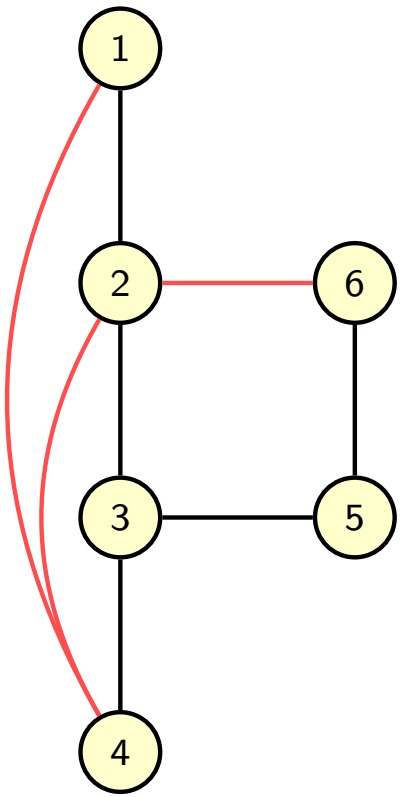
Example



Stacks after traverse(5,(5,6),6)

	6
	5
	4
6	3
5	2
1	1
C	S_V

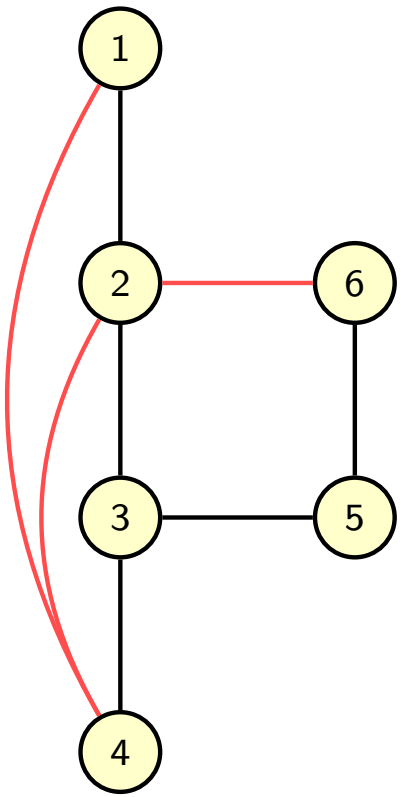
Example



Stacks after traverse(6,(6,2),2)

	6
	5
	4
	3
	2
1	1
C	S_V

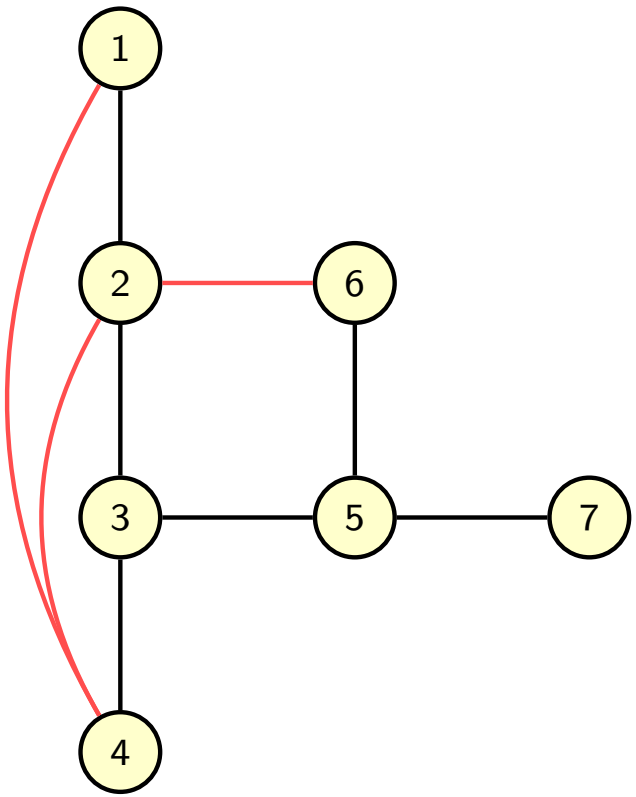
Example



Stacks after backtrack(6,(5,6),5)

	6
	5
	4
	3
	2
	1
1	1
C	S_V

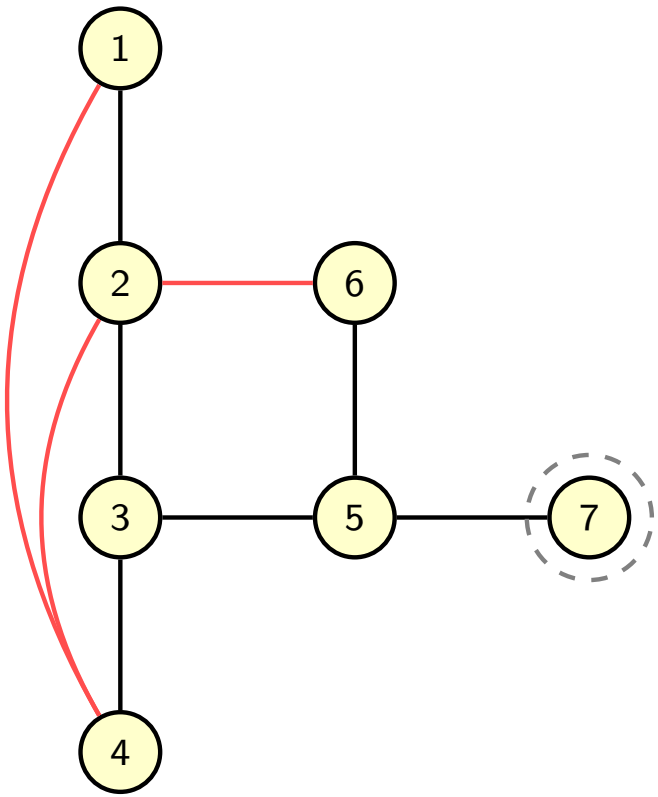
Example



Stacks after traverse(5,(5,7),7)

	7
	6
	5
	4
	3
7	2
1	1
C	S_V

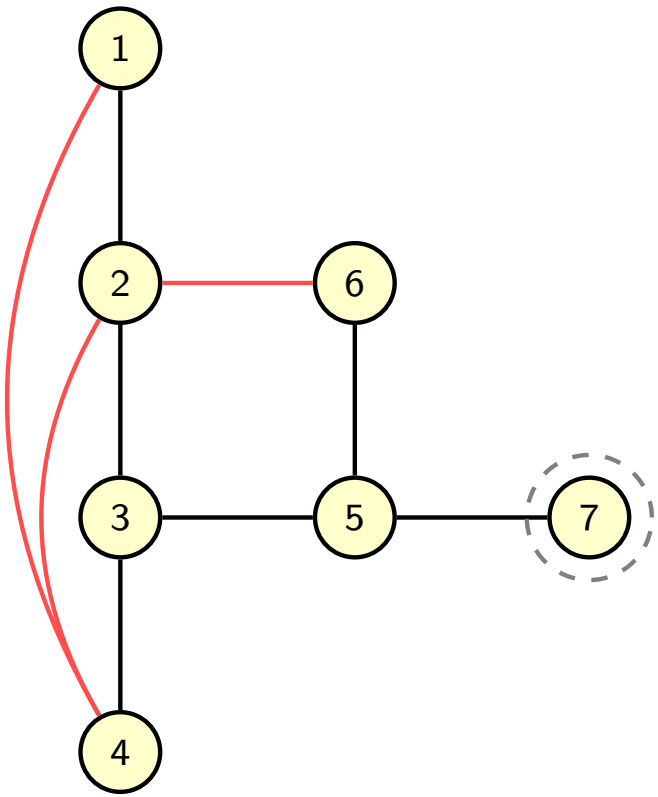
Example



Stacks after backtrack(7,(5,7),6)

	6
	5
	4
	3
	2
1	1
C	S_V

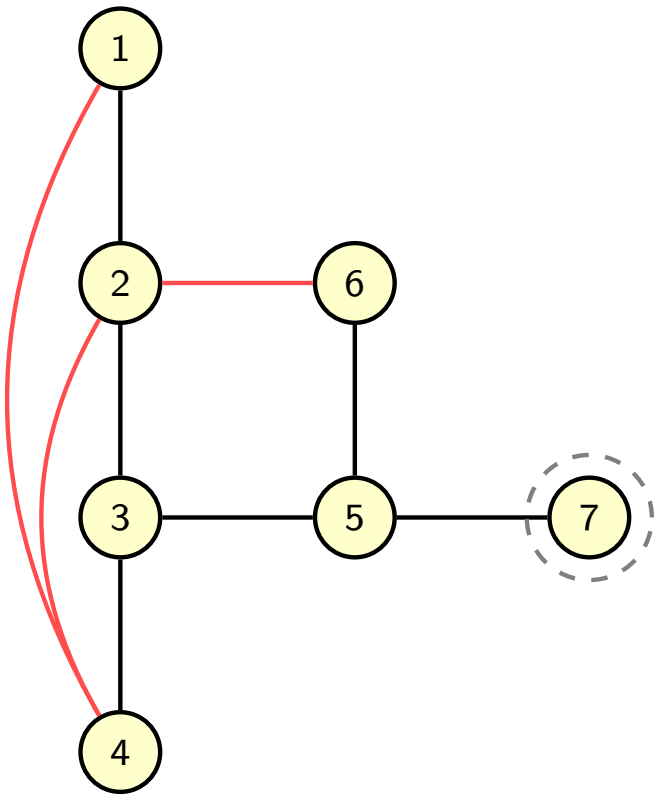
Example



Stacks after backtrack(5,(3,5),3)

	6
	5
	4
	3
	2
1	1
C	S_V

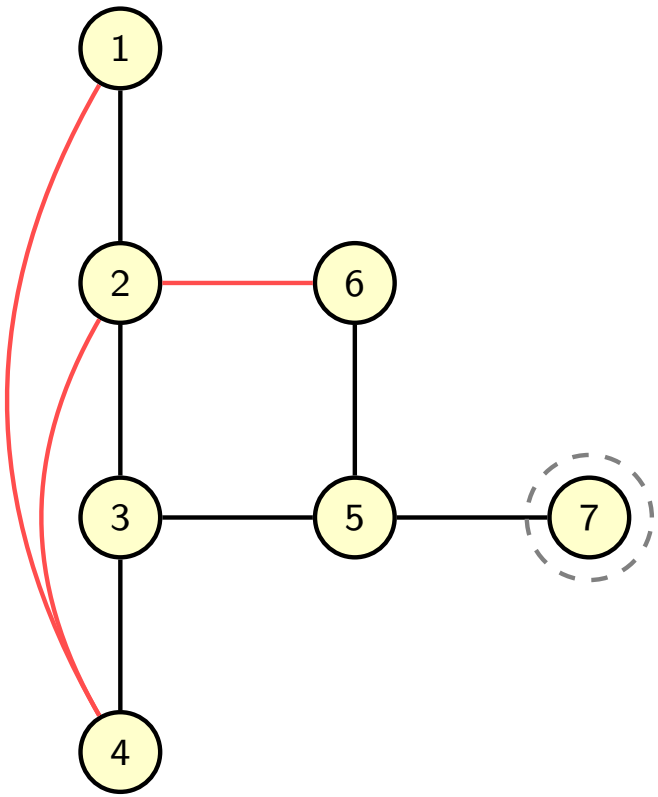
Example



Stacks after backtrack(3,(2,3),2)

	6
	5
	4
	3
	2
1	1
C	S_V

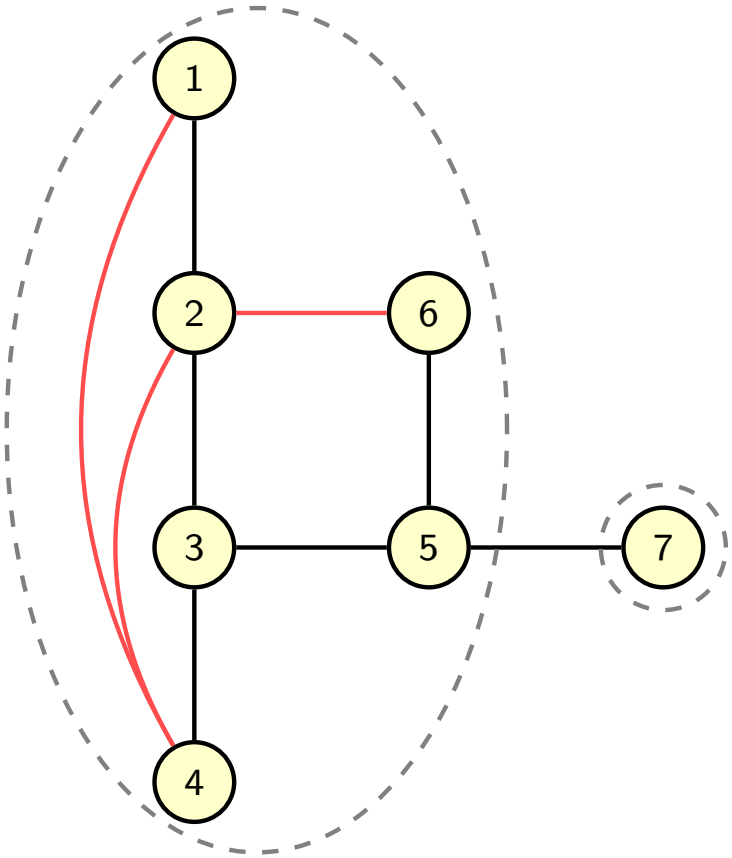
Example



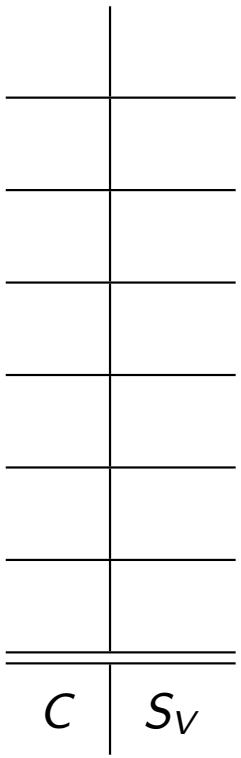
Stacks after backtrack(2,(1,2),1)

	6
	5
	4
	3
	2
1	1
C	S_V

Example



Stacks after backtrack(1,`nil`,`nil`)



Correctness

Lemma

For an undirected graph G, two nodes $u, v \in V$ are in a 2-edge connected component if and only if there is a closed sequence (cycle with repetition of vertices) of edges that contains both. The set of 2-edge connected components forms a partition of V.

Theorem

The DFS Algorithm computes the 2-edge connected components of an undirected graph in time $O(n + m)$.

The proof of the theorem is left as an exercise.

Directed DFS for Strongly Connected Components

The methods are very similar to the ones for 2-edge connected components. This time, however, we use the framework for directed DFG and change the term in the box as shown below.

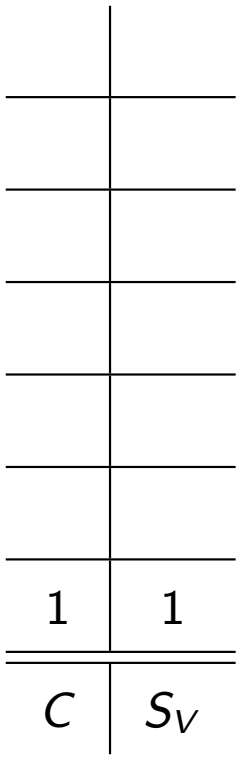
root(s)
- push $s \to S_V$
- push $s \to C$

traverse(v, e, w)
- **if** *e is tree edge* **then**
  - push $w \to S_V$
  - push $w \to C$
- **if** *e is back edge*
- **or** *e is cross edge with* $w \in S_V$
- **then**
  - **while** $w \prec top(C)$ **do**
    - $pop(C)$

backtrack(w, e, v)
- **if** $w = top(C)$ **then**
  - $pop(C)$
  - **repeat**
    - $u \leftarrow and$
    - $\texttt{component}[u] \leftarrow w$
  - **until** $u = w$

Example

Stacks after root(1)

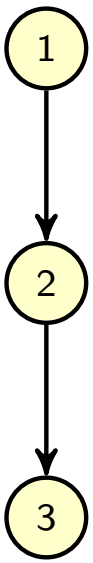


Example

Stacks after traverse(1,(1,2),2)

2	2
1	1
C	S_V

Example



Stacks after traverse(2,(2,3),3)

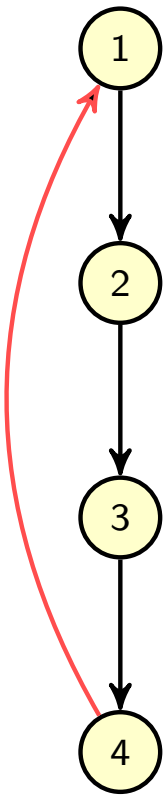
3	3
2	2
1	1
C	S_V

Example

Stacks after traverse(3,(3,4),4)

4	4
3	3
2	2
1	1
C	S_V

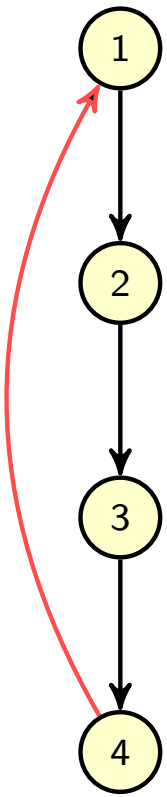
Example



Stacks after traverse(4,(4,1),1)

	4
	3
	2
1	1
C	S_V

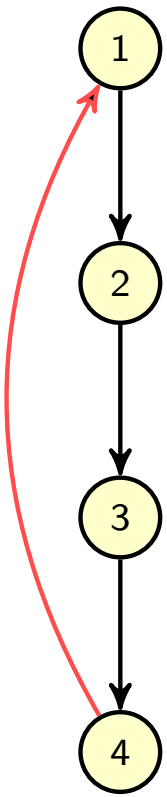
Example



Stacks after backtrack(4,(3,4),3)

	4
	3
	2
1	1
C	S_V

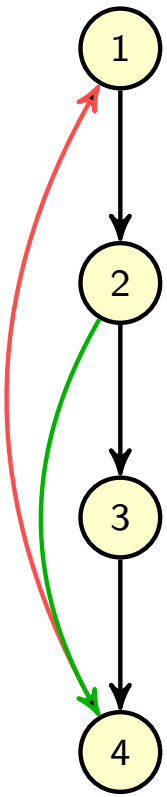
Example



Stacks after backtrack(3,(2,3),2)

	4
	3
	2
1	1
C	S_V

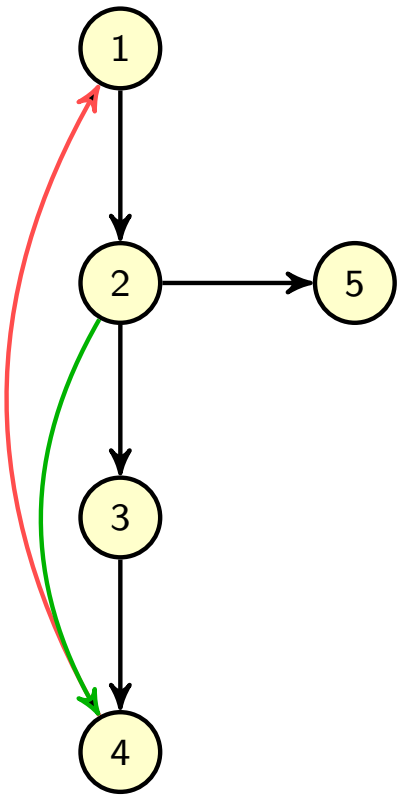
Example



Stacks after traverse(2,(2,4),4)

	4
	3
	2
1	1
C	S_V

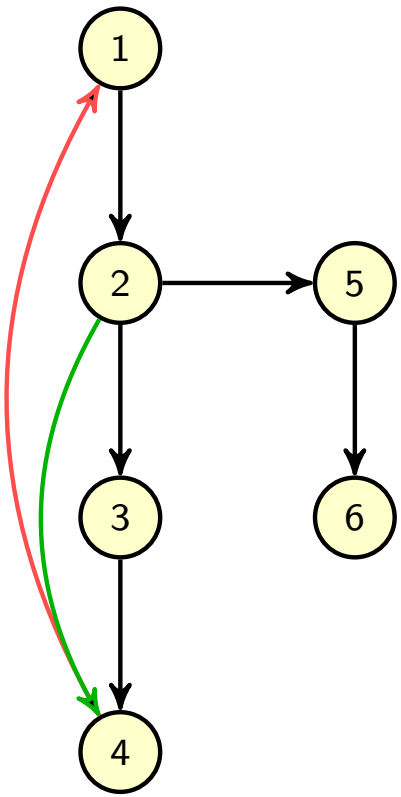
Example



Stacks after traverse(2,(2,5),5)

	5
	4
	3
5	2
1	1
C	S_V

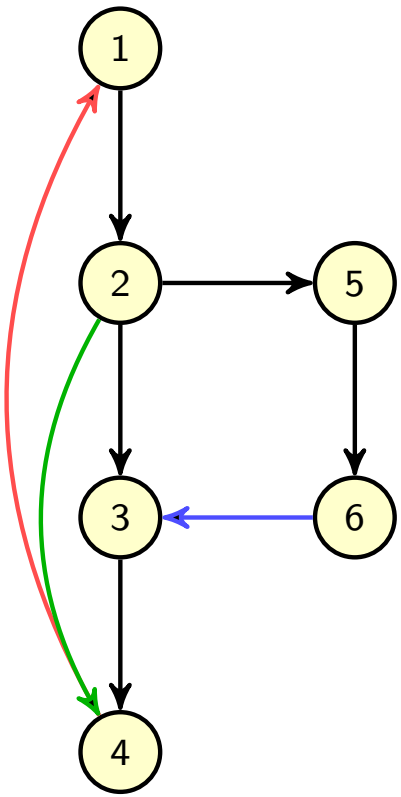
Example



Stacks after traverse(5,(5,6),6)

	6
	5
	4
6	3
5	2
1	1
C	S_V

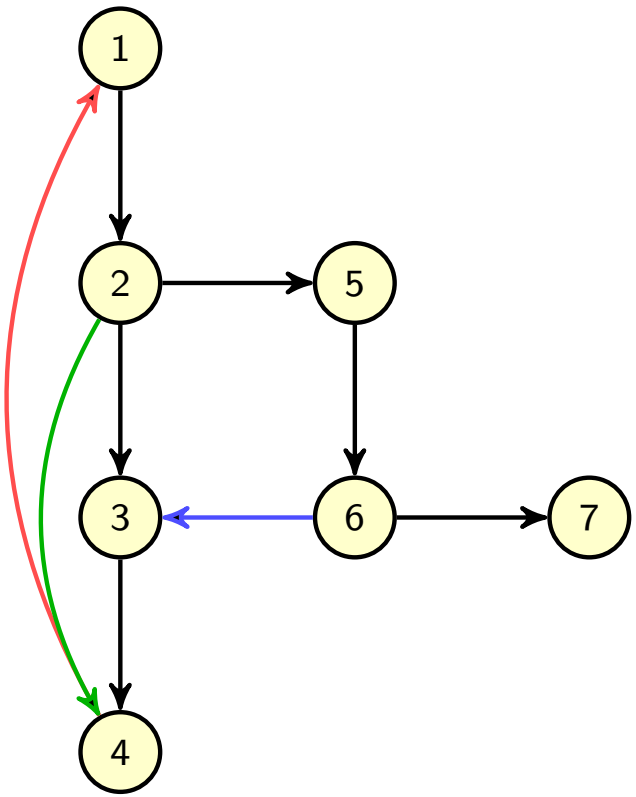
Example



Stacks after traverse(6,(6,3),3)

	6
	5
	4
	3
	2
1	1
C	S_V

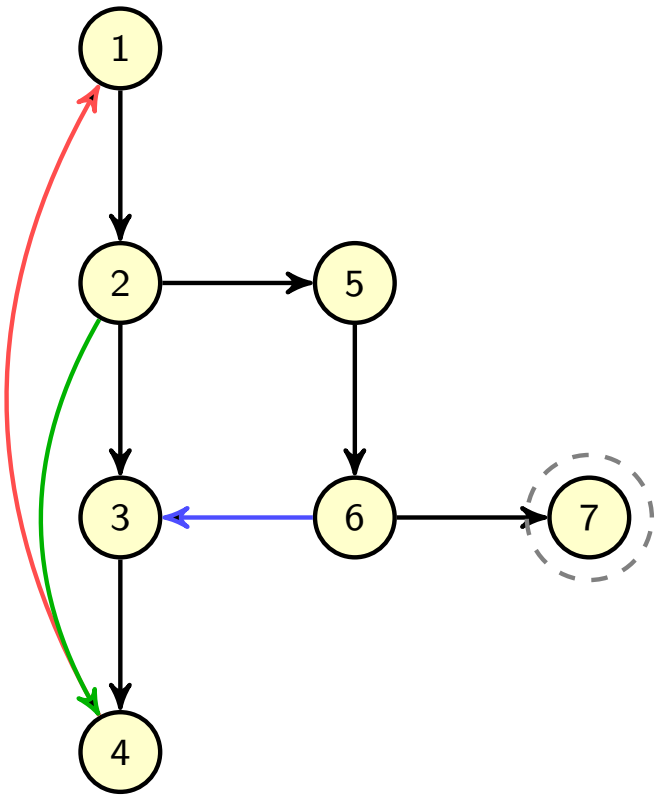
Example



Stacks after traverse(6,(6,7),7)

	7
	6
	5
	4
	3
7	2
1	1
C	S_V

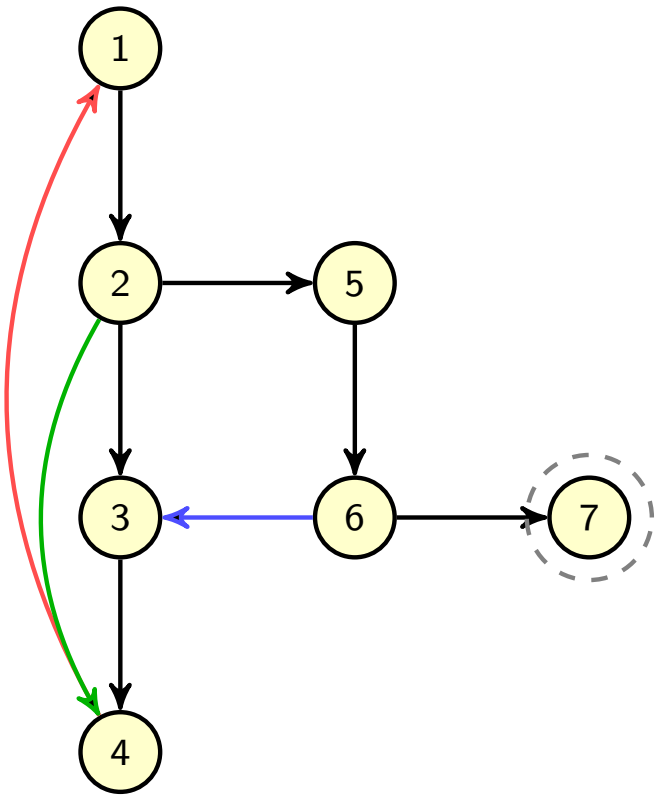
Example



Stacks after backtrack(7,(6,7),6)

	6
	5
	4
	3
	2
1	1
C	S_V

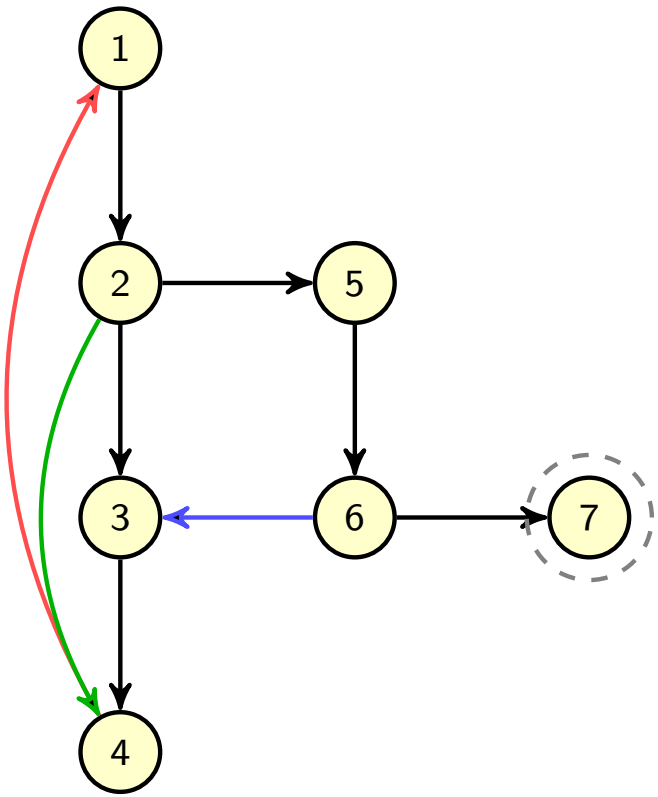
Example



Stacks after backtrack(6,(5,6),5)

	6
	5
	4
	3
	2
1	1
C	S_V

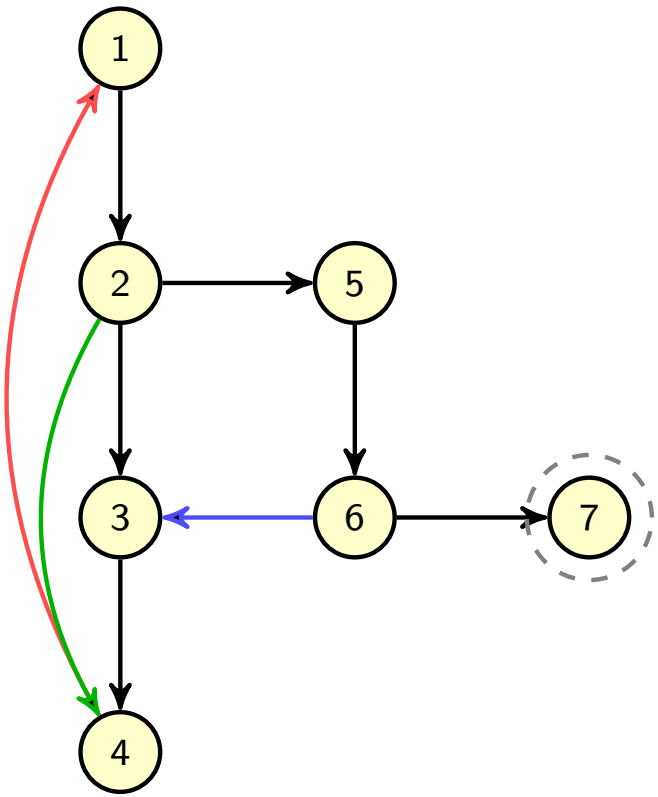
Example



Stacks after backtrack(5,(2,5),2)

	6
	5
	4
	3
	2
1	1
C	S_V

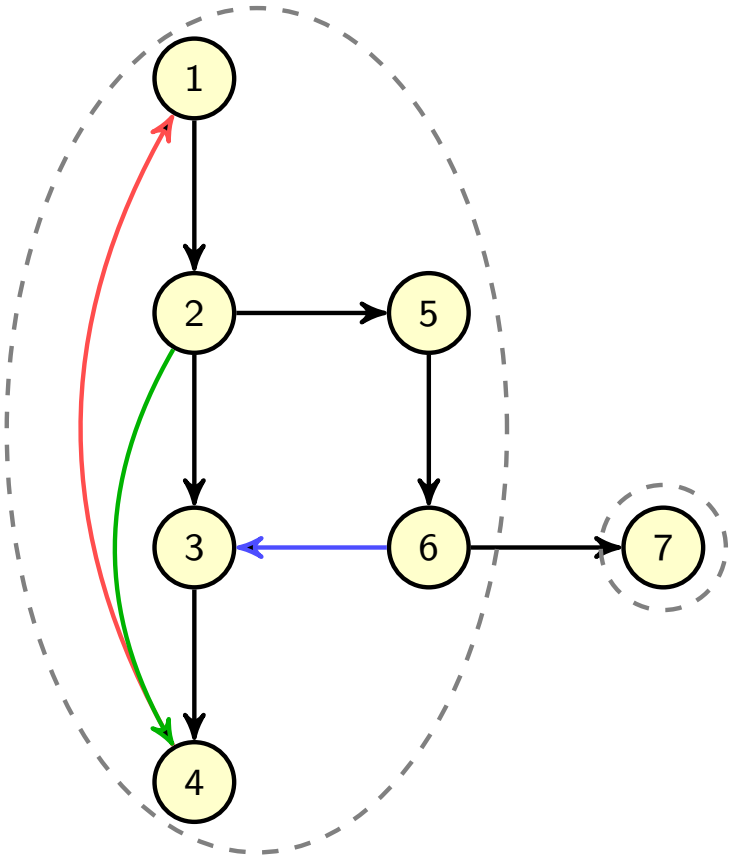
Example



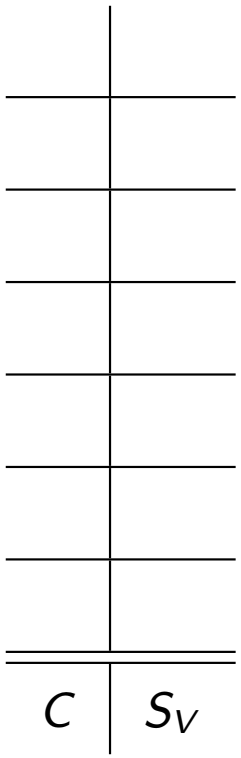
Stacks after backtrack(2,(1,2),1)

	6
	5
	4
	3
	2
1	1
C	S_V

Example



Stacks after backtrack(1,`nil`,`nil`)



Correctness

Lemma

For a directed graph G, two nodes $u, v \in V$ are in an SCC if and only if there is a closed directed sequence (cycle with repetition of vertices) of edges that contains both. The set of SCCs forms a partition of V.

Theorem

The DFS Algorithm computes the strongly connected components of a graph in time $O(n + m)$.

The proof of the theorem is left as an exercise. We have to show, in particular, that cross and forward edges are treated correctly.