Graphs and Algorithms

Advanced Programming and Algorithmic Design

Alberto Casagrande
Email: `acasagrande@units.it`

a.y. 2019/2020

Is There a Unique Way to Represent. . .

- dynamic systems
- information flows
- infectious disease spread
- knowledge relations
- dependency relations
- computer network
- document network (e.g., WWW)
- money transfer tracking
- route systems

Is There a Unique Way to Represent. . .

- dynamic systems
- information flows
- infectious disease spread
- knowledge relations
- dependency relations
- computer network
- document network (e.g., WWW)
- money transfer tracking
- route systems

Yes, by using graphs

Basics

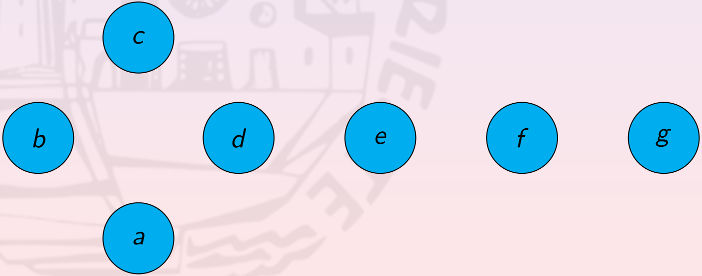
Graphs (Graph Theory)

Are pairs (V, E) where:

Graphs (Graph Theory)

Are pairs (V, E) where:

V is a set of nodes

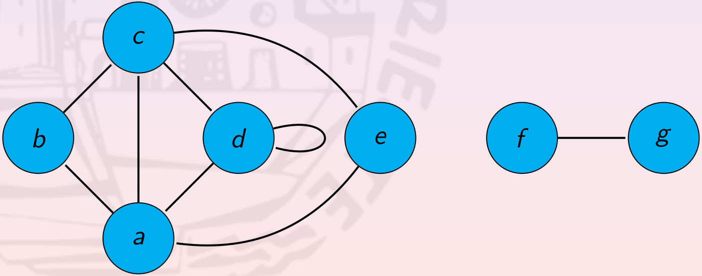


Graphs (Graph Theory)

Are pairs (V, E) where:

V is a set of nodes

E is a set of edges



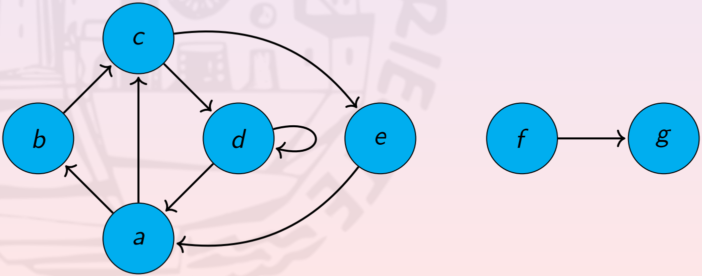
Graphs (Graph Theory)

Are pairs (V, E) where:

V is a set of nodes

E is a set of edges

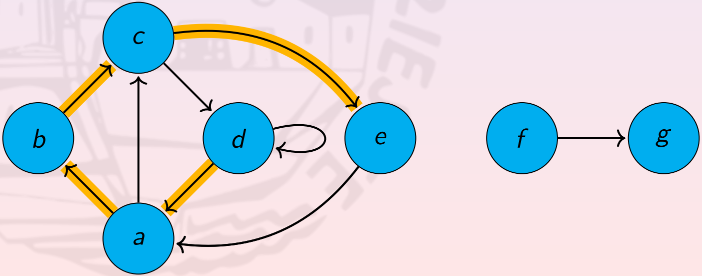
If the edges are (un)directed, the graph is (un)directed



Paths and Cycles

A path of length n between $a, b \in V$ is a sequence $e_1, \ldots, e_n$ s.t.

- e_1 involves a
- e_n involves b
- e_i and e_{i+1} involve a common node n_i

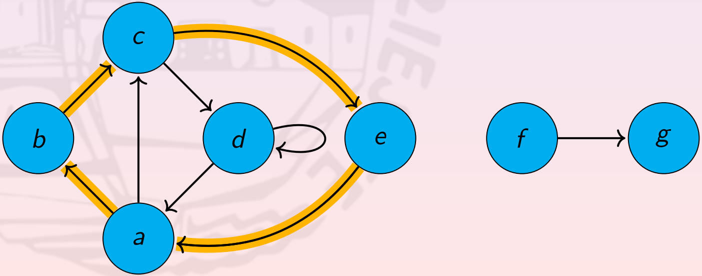


Paths and Cycles

A path of length n between $a, b \in V$ is a sequence $e_1, \ldots, e_n$ s.t.

- e_1 involves a
- e_n involves b
- e_i and e_{i+1} involve a common node n_i

A cycle is a path whose initial and final node coincide.



Connected and Acyclic Graphs (Graph Theory)

A graph is connected if there is a path between every pairs of nodes

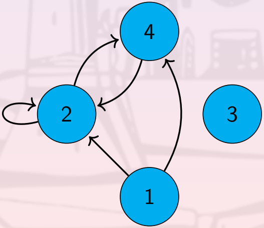
A connected component of a undirected graph G is a maximum connected sub-graph of G.

A graph is acyclic if it does not contain cycles

Directed Acyclic Graphs are also known as DAGs

Representing Graphs

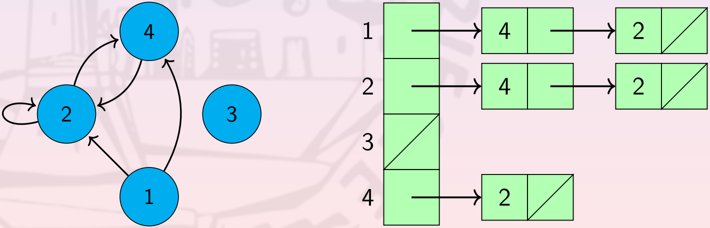
Two main ways:



Representing Graphs

Two main ways:

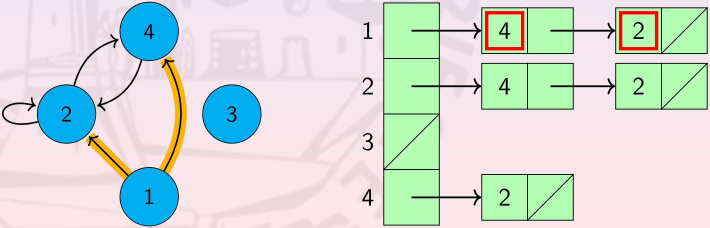
- adjacency lists (usually, for sparse graphs)



Representing Graphs

Two main ways:

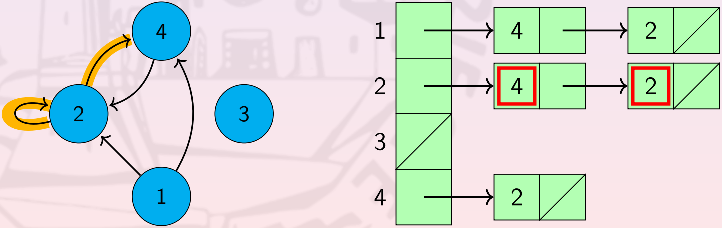
- adjacency lists (usually, for sparse graphs)



Representing Graphs

Two main ways:

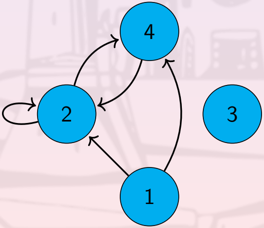
- adjacency lists (usually, for sparse graphs)

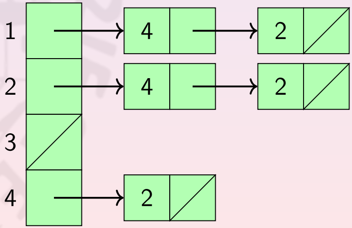


Representing Graphs

Two main ways:

- adjacency lists (usually, for sparse graphs)

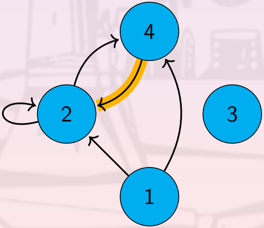


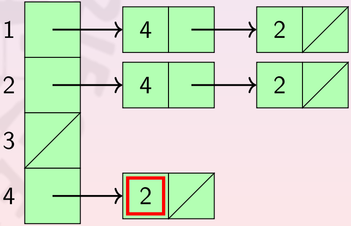


Representing Graphs

Two main ways:

- adjacency lists (usually, for sparse graphs)

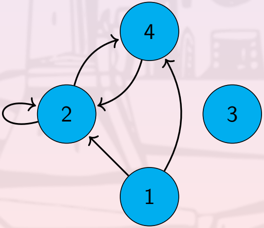




Representing Graphs

Two main ways:

- adjacency lists (usually, for sparse graphs)
- adjacency matrix (usually, for dense graphs)

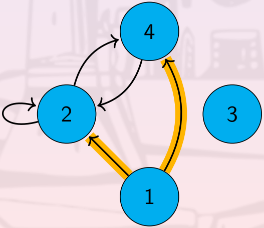


	1	2	3	4
1	0	1	0	1
2	0	1	0	1
3	0	0	0	0
4	0	1	0	0

Representing Graphs

Two main ways:

- adjacency lists (usually, for sparse graphs)
- adjacency matrix (usually, for dense graphs)

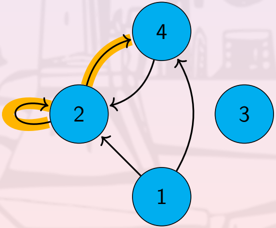


	1	2	3	4
1	0	$\boxed{1}$	0	$\boxed{1}$
2	0	1	0	1
3	0	0	0	0
4	0	1	0	0

Representing Graphs

Two main ways:

- adjacency lists (usually, for sparse graphs)
- adjacency matrix (usually, for dense graphs)

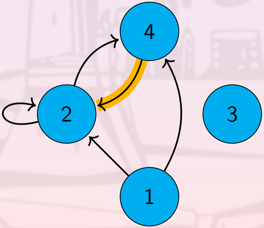


	1	2	3	4
1	0	1	0	1
2	0	1	0	1
3	0	0	0	0
4	0	1	0	0

Representing Graphs

Two main ways:

- adjacency lists (usually, for sparse graphs)
- adjacency matrix (usually, for dense graphs)



	1	2	3	4
1	0	1	0	1
2	0	1	0	1
3	0	0	0	0
4	0	$\boxed{1}$	0	0

Visiting a Graph

The most trivial task to be performed on a graph is to visit it

Two main ways for both directed and undirected graphs:

Visiting a Graph

The most trivial task to be performed on a graph is to visit it

Two main ways for both directed and undirected graphs:

Breadth-First-Search visits the graph as a "wave" from the source

Visiting a Graph

The most trivial task to be performed on a graph is to visit it

Two main ways for both directed and undirected graphs:

Breadth-First-Search visits the graph as a “wave” from the source

Deapth-First-Search search “deeper” in the graph whenever possible

Breadth-First Search

Breadth-First-Search (BFS)

Visiting order is related to the distance from a source s: the lesser the distance of a node, the sooner it will be visited

Because of this, BFS is used to compute source-node distances

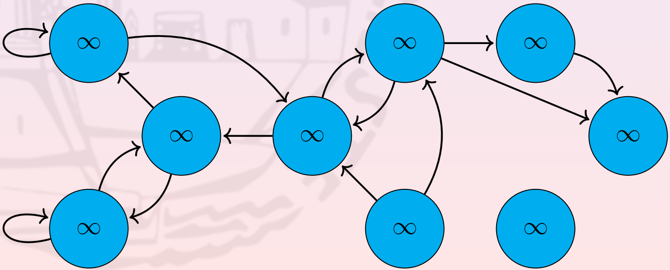
Breadth-First-Search (BFS)

Visiting order is related to the distance from a source s: the lesser the distance of a node, the sooner it will be visited

Because of this, BFS is used to compute source-node distances

It also produces the breadth-first tree i.e., the tree of shortest paths from s, and returns the shortest path from s to any reachable node

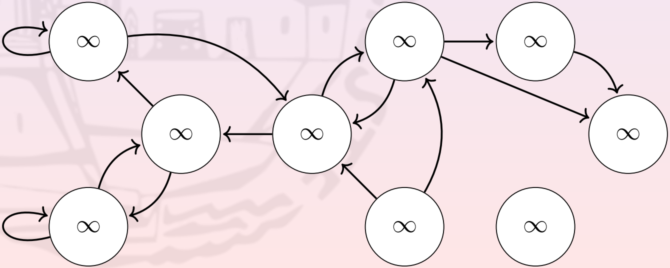
Breadth-First-Search (BFS): Coloring and Example



Breadth-First-Search (BFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

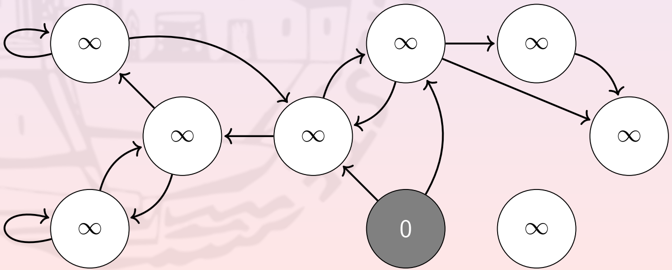
- WHITE nodes have not been discovered yet



Breadth-First-Search (BFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

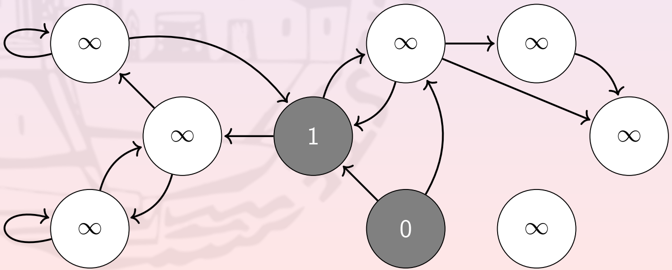
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but some of their neighbors are still undiscovered



Breadth-First-Search (BFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

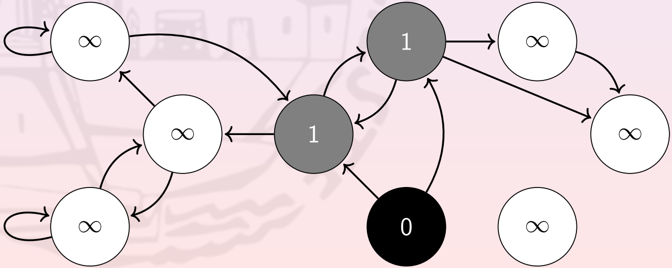
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but some of their neighbors are still undiscovered



Breadth-First-Search (BFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

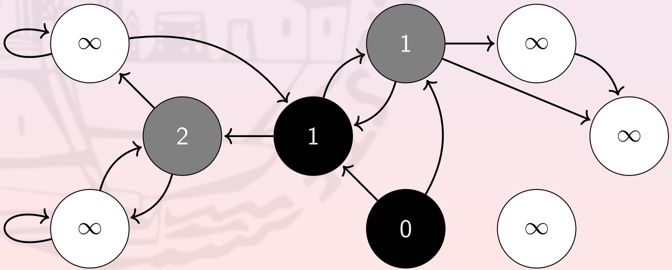
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but some of their neighbors are still undiscovered
- BLACK nodes have been discovered and all their neighbors have been discovered too



Breadth-First-Search (BFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

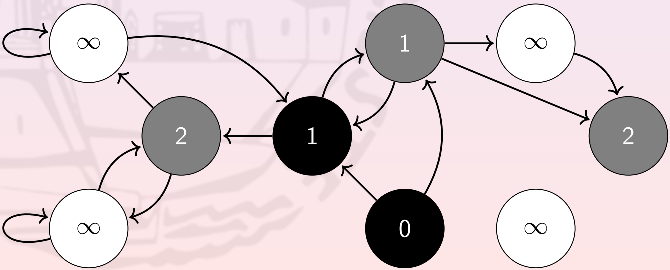
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but some of their neighbors are still undiscovered
- BLACK nodes have been discovered and all their neighbors have been discovered too



Breadth-First-Search (BFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

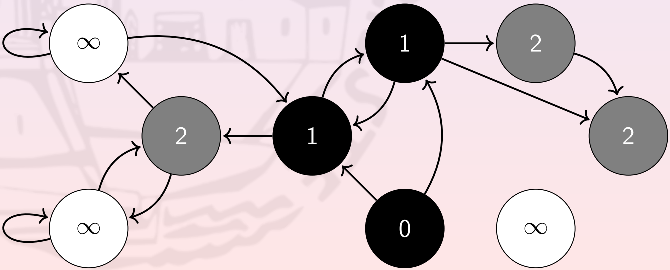
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but some of their neighbors are still undiscovered
- BLACK nodes have been discovered and all their neighbors have been discovered too



Breadth-First-Search (BFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

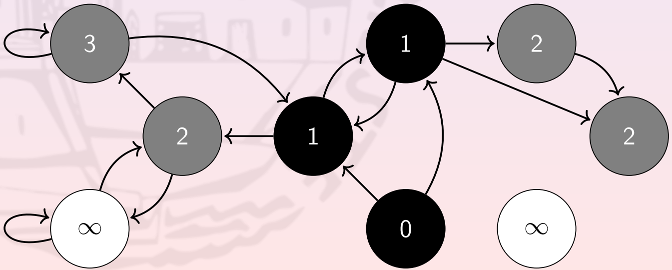
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but some of their neighbors are still undiscovered
- BLACK nodes have been discovered and all their neighbors have been discovered too



Breadth-First-Search (BFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

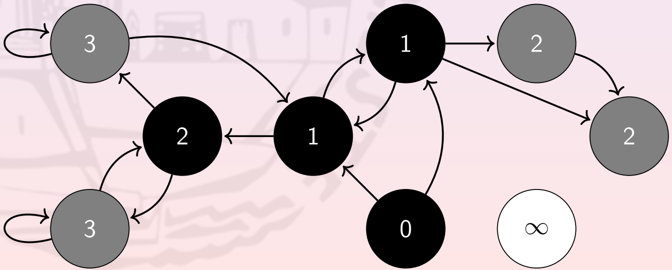
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but some of their neighbors are still undiscovered
- BLACK nodes have been discovered and all their neighbors have been discovered too



Breadth-First-Search (BFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

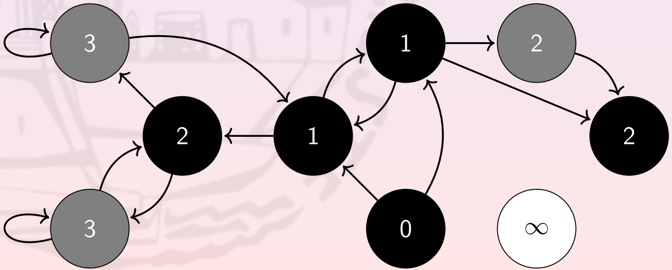
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but some of their neighbors are still undiscovered
- BLACK nodes have been discovered and all their neighbors have been discovered too



Breadth-First-Search (BFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

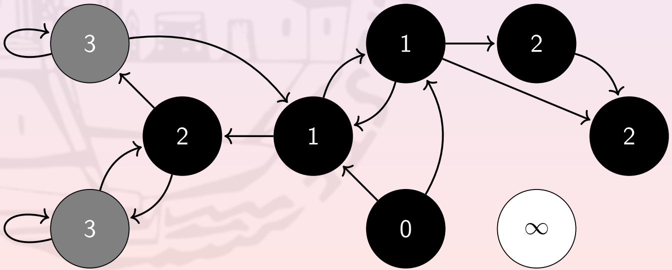
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but some of their neighbors are still undiscovered
- BLACK nodes have been discovered and all their neighbors have been discovered too



Breadth-First-Search (BFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

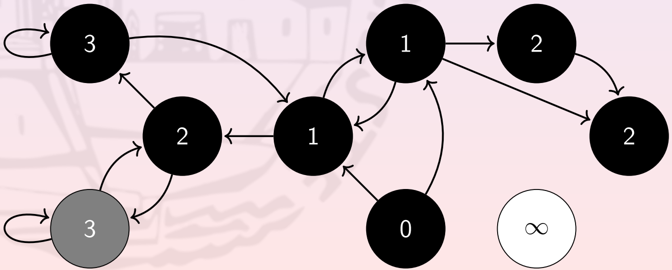
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but some of their neighbors are still undiscovered
- BLACK nodes have been discovered and all their neighbors have been discovered too



Breadth-First-Search (BFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

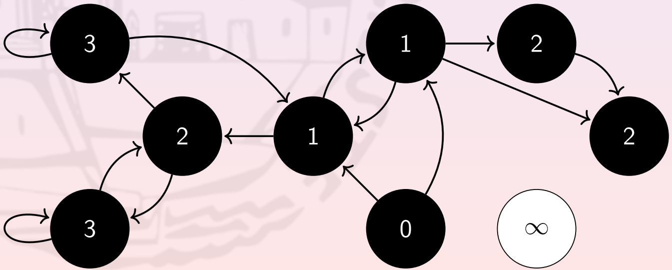
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but some of their neighbors are still undiscovered
- BLACK nodes have been discovered and all their neighbors have been discovered too



Breadth-First-Search (BFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but some of their neighbors are still undiscovered
- BLACK nodes have been discovered and all their neighbors have been discovered too



Breadth-First-Search (BFS): Pseudo-Code

```
def BFS_SET(v, color, d, pred):
  v.color ← color
  v.d ← d
  v.pred ← pred
enddef

def BFS_INIT(G, s):
  for v in G.V:
    BFS_SET(v, WHITE, ∞, NIL)
  endfor
  BFS_SET(s, GRAY, 0, s)

  return BUILD_QUEUE([s])
enddef
```

Breadth-First-Search (BFS): Pseudo-Code (Cont'd)

```
def BFS(G, s):
  Q ← BFS_INIT(G, s)
  while Q ≠ ∅:
    u ← DEQUEUE(Q)

    for v in G.Adj[u]:
      if v.color = WHITE:
        BFS_SET(v, GRAY, u.d+1, u)
        ENQUEUE(Q, v)
      endif
    endfor
    u.color ← BLACK
  endwhile
enddef
```

Breadth-First-Search (BFS): Complexity

An iteration of the `while` extracts a u from Q and GREY colors it

The `for` loop costs $\Theta(|Adj[u]|)$ per `while` iteration

Each iteration of the `for` enqueues $v \in Adj[u]$ only if it is WHITE

Breadth-First-Search (BFS): Complexity

An iteration of the `while` extracts a u from Q and GREY colors it

The `for` loop costs $\Theta(|Adj[u]|)$ per `while` iteration

Each iteration of the `for` enqueues $v \in Adj[u]$ only if it is WHITE

Every node can be inserted in Q at most once

Cumulatively, the `while` costs $O(|V|)$ and the `for` $O(|E|)$

Breadth-First-Search (BFS): Complexity

An iteration of the `while` extracts a u from Q and GREY colors it

The `for` loop costs $\Theta(|Adj[u]|)$ per `while` iteration

Each iteration of the `for` enqueues $v \in Adj[u]$ only if it is WHITE

Every node can be inserted in Q at most once

Cumulatively, the `while` costs $O(|V|)$ and the `for` $O(|E|)$

`BFG` has asymptotic complexity $O(|V| + |E|)$

Breadth-First-Search (BFS): Code Properties

Lemma

Let $Q = [v_1, \dots, v_n]$ be the queue during BFS. Then $v_i.d \leq v_{i+1}.d$ for all $i \in [1, n-1]$ and $v_n.d \leq v_1.d + 1$.

Theorem

Let $\delta(s, v)$ be the distance from s to v. After BFS:

- *$v.d \neq \infty$ iff v is reachable from s*
- *if $v.d \neq \infty$, then $v.d = \delta(s, v)$*
- *the shortest path from s to v ends with $(v.pred, v)$*

Depth-First Search

Depth-First-Search (DFS)

All the nodes of the graph are visited

Depth-First-Search (DFS)

All the nodes of the graph are visited

- a non-visited node s is selected as source

Depth-First-Search (DFS)

All the nodes of the graph are visited

- a non-visited node s is selected as source
- a path is extended as much as it is possible by adding non-visited nodes

Depth-First-Search (DFS)

All the nodes of the graph are visited

- a non-visited node s is selected as source
- a path is extended as much as it is possible by adding non-visited nodes
- the process is repeated on all the branches left behind by previous step

Depth-First-Search (DFS)

All the nodes of the graph are visited

- a non-visited node s is selected as source
- a path is extended as much as it is possible by adding non-visited nodes
- the process is repeated on all the branches left behind by previous step
- if some nodes remain non-visited, a node among them is selected as new source

Depth-First-Search (DFS)

All the nodes of the graph are visited

- a non-visited node s is selected as source
- a path is extended as much as it is possible by adding non-visited nodes
- the process is repeated on all the branches left behind by previous step
- if some nodes remain non-visited, a node among them is selected as new source

DFS produces a depth-first forrest of the predecessors

Depth-First-Search (DFS)

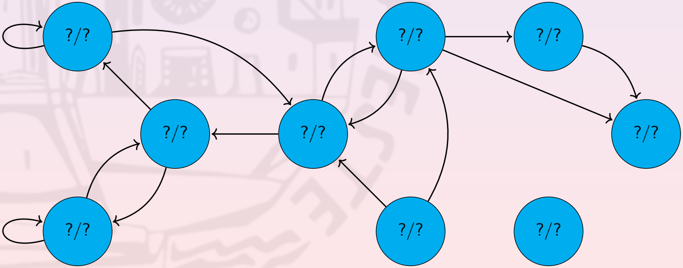
All the nodes of the graph are visited

- a non-visited node s is selected as source
- a path is extended as much as it is possible by adding non-visited nodes
- the process is repeated on all the branches left behind by previous step
- if some nodes remain non-visited, a node among them is selected as new source

DFS produces a depth-first forrest of the predecessors

DFS labels all the nodes with discovery time and finishing time

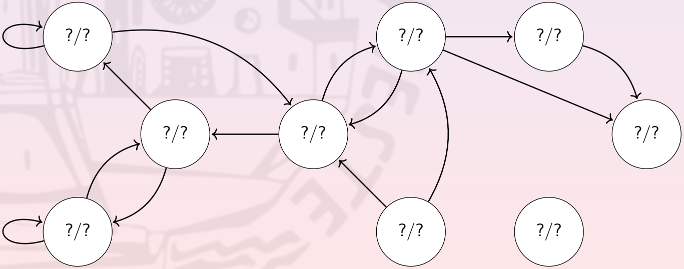
Depth-First-Search (DFS): Coloring and Example



Depth-First-Search (DFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

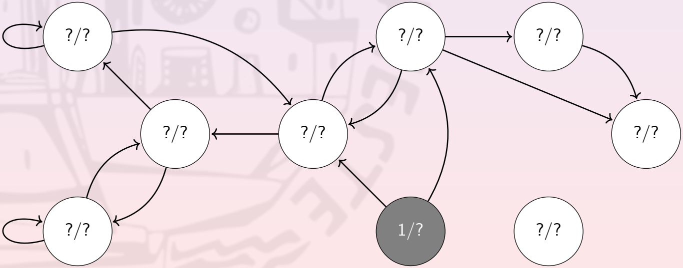
- WHITE nodes have not been discovered yet



Depth-First-Search (DFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

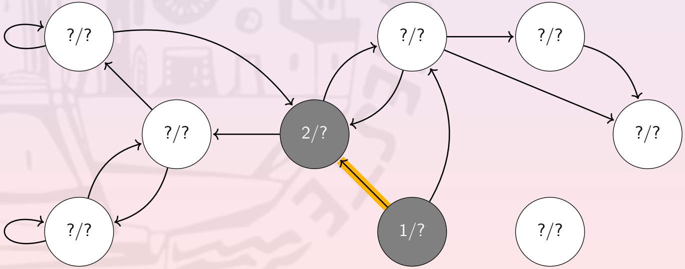
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but not finished yet



Depth-First-Search (DFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

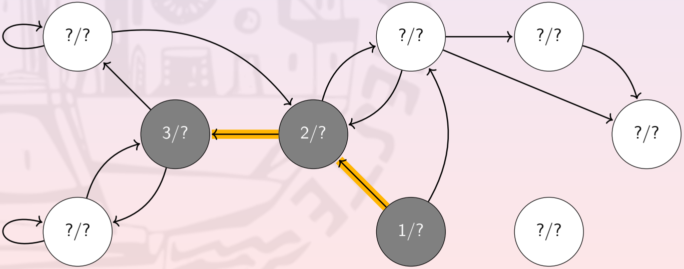
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but not finished yet



Depth-First-Search (DFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

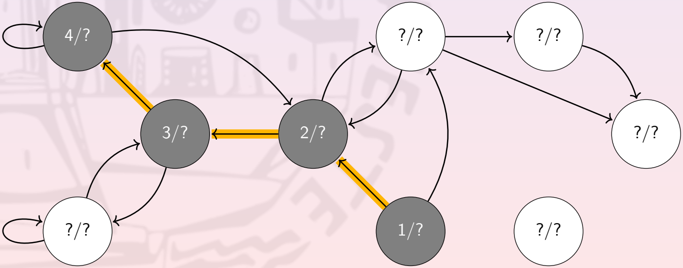
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but not finished yet
- BLACK nodes have been finished



Depth-First-Search (DFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

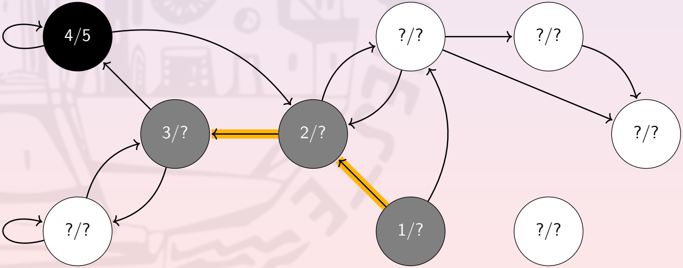
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but not finished yet
- BLACK nodes have been finished



Depth-First-Search (DFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

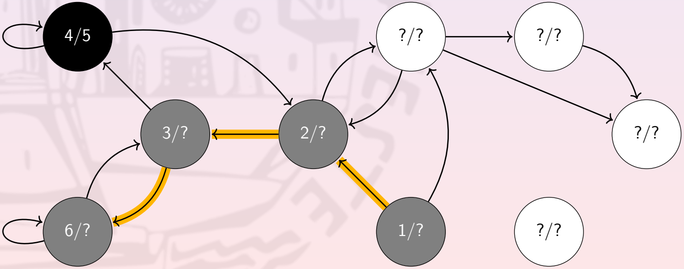
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but not finished yet
- BLACK nodes have been finished



Depth-First-Search (DFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

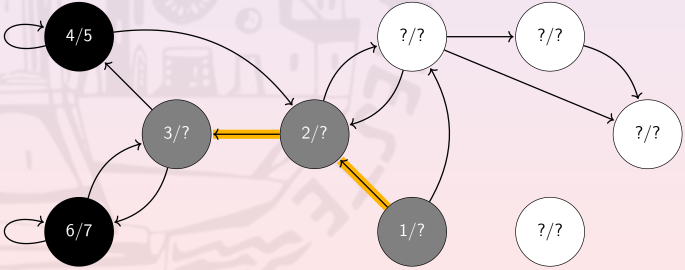
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but not finished yet
- BLACK nodes have been finished



Depth-First-Search (DFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

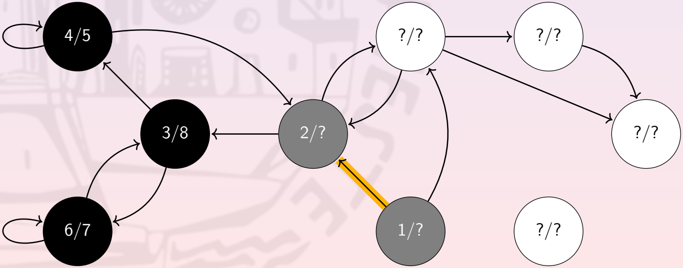
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but not finished yet
- BLACK nodes have been finished



Depth-First-Search (DFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

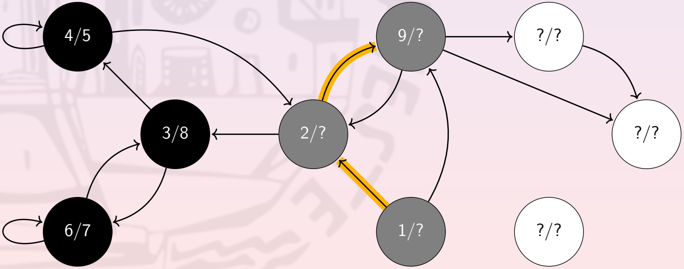
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but not finished yet
- BLACK nodes have been finished



Depth-First-Search (DFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

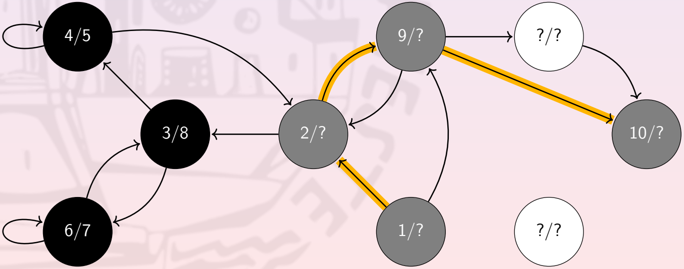
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but not finished yet
- BLACK nodes have been finished



Depth-First-Search (DFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

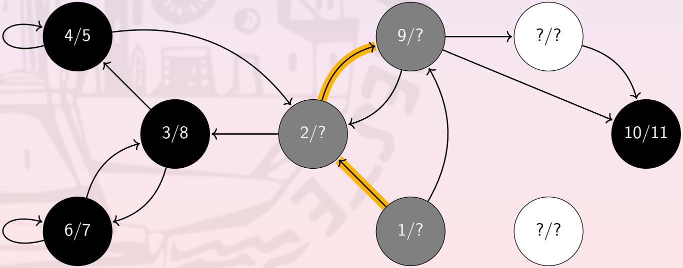
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but not finished yet
- BLACK nodes have been finished



Depth-First-Search (DFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

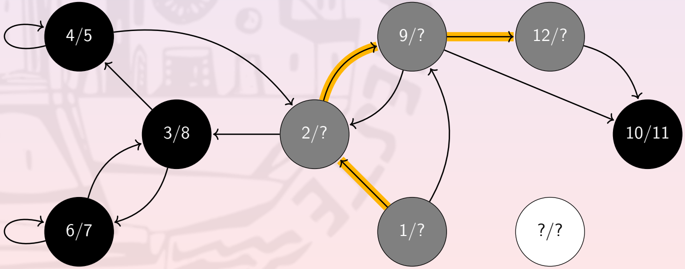
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but not finished yet
- BLACK nodes have been finished



Depth-First-Search (DFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

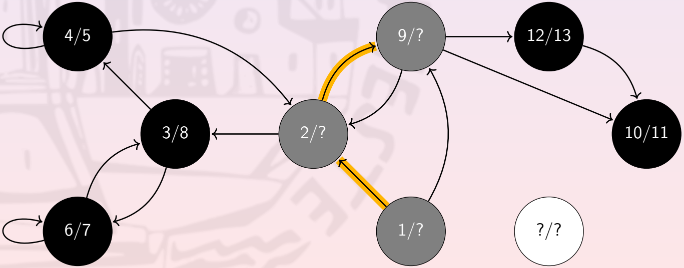
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but not finished yet
- BLACK nodes have been finished



Depth-First-Search (DFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

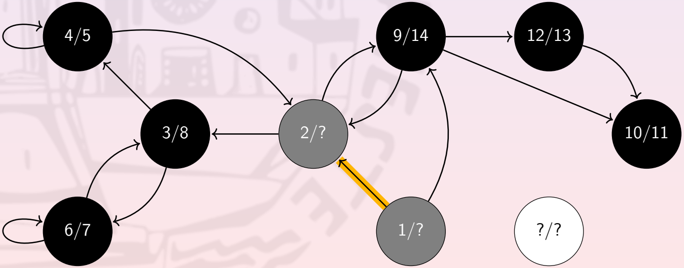
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but not finished yet
- BLACK nodes have been finished



Depth-First-Search (DFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

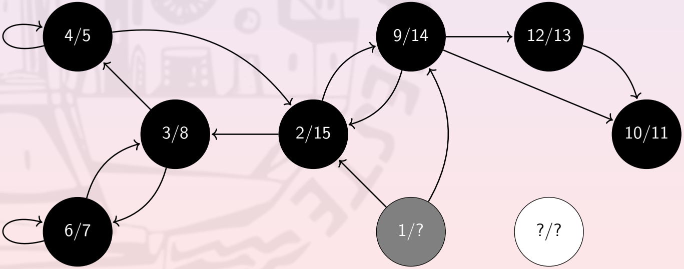
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but not finished yet
- BLACK nodes have been finished



Depth-First-Search (DFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

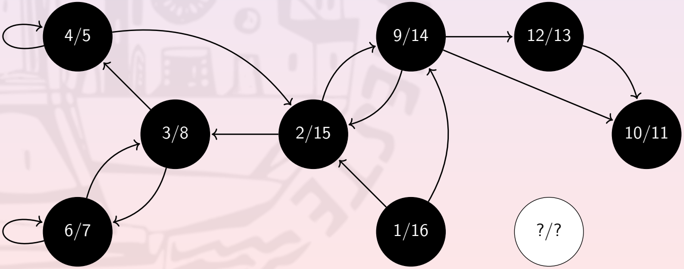
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but not finished yet
- BLACK nodes have been finished



Depth-First-Search (DFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

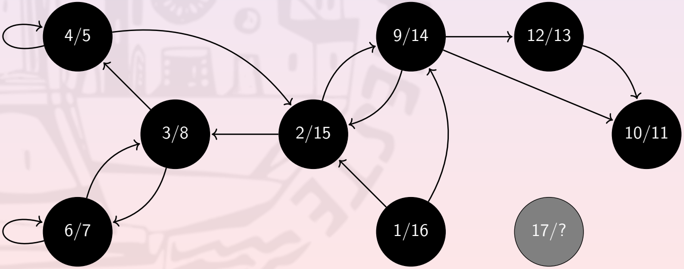
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but not finished yet
- BLACK nodes have been finished



Depth-First-Search (DFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

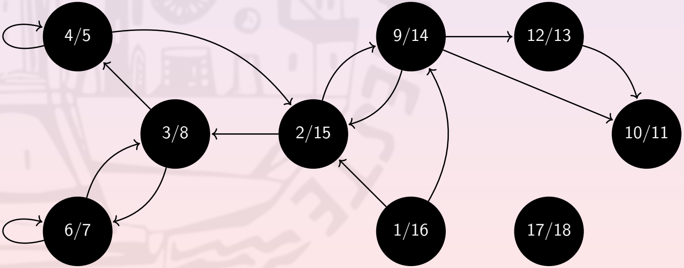
- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but not finished yet
- BLACK nodes have been finished



Depth-First-Search (DFS): Coloring and Example

Nodes are WHITE, GRAY, or BLACK colored

- WHITE nodes have not been discovered yet
- GRAY nodes have been discovered, but not finished yet
- BLACK nodes have been finished



Depth-First-Search (DFS): Pseudo-Code

```
def DFS(G):
  for v in G.V:
    v.color ← WHITE
    v.pred ← NIL
  endfor

  time ← 0
  for v in G.V:
    if v.color = WHITE:
      time ← DFS_VISIT(G,v,time)
    endif
  endfor
enddef
```

Depth-First-Search (DFS): Pseudo-Code (Cont'd 2)

```
def DFS_VISIT(G,v,time):
  # discovery
  time ← time + 1
  v.d ← time
  v.color = GRAY

  # search for WHITE neighbors
  for u in G.Adj[v]:
    if u.color = WHITE:
      u.pred ← v
      time ← DFS_VISIT(G,v,time)
    endif
  endfor
```

Depth-First-Search (DFS): Pseudo-Code (Cont'd 3)

```
    # finalization
    time ← time + 1
    v.f ← time
    v.color = BLACK

    return time
enddef
```

Depth-First-Search (DFS): Complexity

Each `DFS_VISIT` call GRAY color the node parameter

The `for` loop costs $\Theta(|Adj[u]|)$ per `DFS_VISIT` call

Each iteration of the `for` calls `DFS_VISIT` on $v \in Adj[u]$ if only if it is WHITE

Depth-First-Search (DFS): Complexity

Each `DFS_VISIT` call GRAY color the node parameter

The `for` loop costs $\Theta(|Adj[u]|)$ per `DFS_VISIT` call

Each iteration of the `for` calls `DFS_VISIT` on $v \in Adj[u]$ if only if it is WHITE

Every node is used as `DFS_VISIT` parameter exactly once

Cumulatively, the `for` loop in `DFS` is repeated $\Theta(|V|)$ time and produces $\Theta(|E|)$ iterations of the `DFS_VISIT`'s `for` loop

Depth-First-Search (DFS): Complexity

Each `DFS_VISIT` call GRAY color the node parameter

The `for` loop costs $\Theta(|Adj[u]|)$ per `DFS_VISIT` call

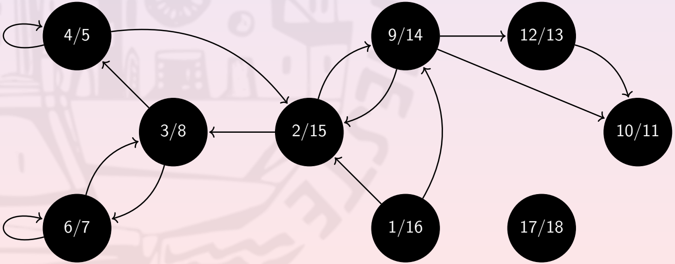
Each iteration of the `for` calls `DFS_VISIT` on $v \in Adj[u]$ if only if it is WHITE

Every node is used as `DFS_VISIT` parameter exactly once

Cumulatively, the `for` loop in `DFS` is repeated $\Theta(|V|)$ time and produces $\Theta(|E|)$ iterations of the `DFS_VISIT`'s `for` loop

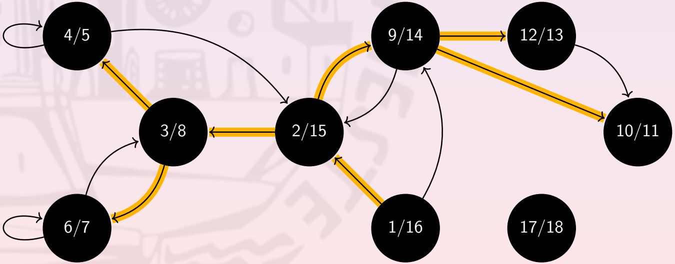
`DFS` has asymptotic complexity $\Theta(|V| + |E|)$

Depth-First-Search (DFS): Edge Classification



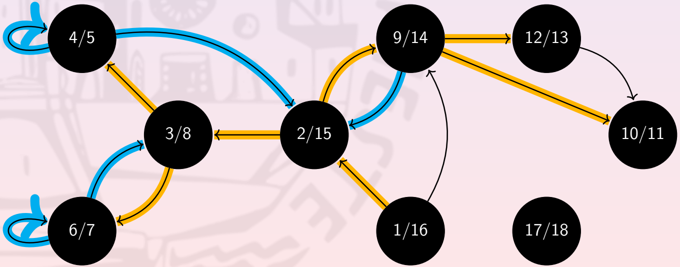
Depth-First-Search (DFS): Edge Classification

Tree Edges: belong to the depth first forest



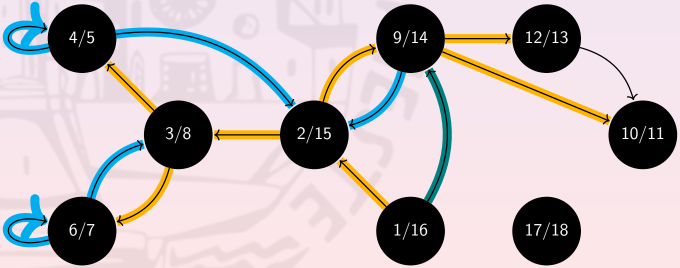
Depth-First-Search (DFS): Edge Classification

Tree Edges: belong to the depth first forest
Back Edges: connect a node to an ancestor or self-loop



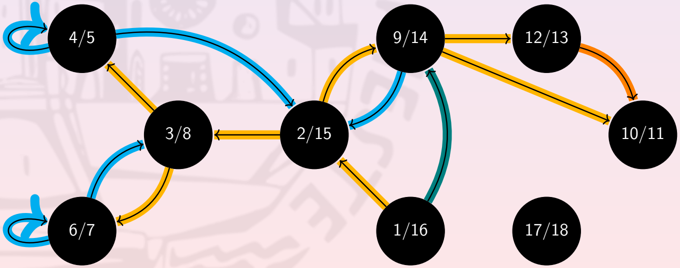
Depth-First-Search (DFS): Edge Classification

Tree Edges: belong to the depth first forest
Back Edges: connect a node to an ancestor or self-loop
Forward Edges: connect a node to a non-direct descendant



Depth-First-Search (DFS): Edge Classification

Tree Edges: belong to the depth first forest
Back Edges: connect a node to an ancestor or self-loop
Forward Edges: connect a node to a non-direct descendant
Cross Edges: the remaining edges



Depth-First-Search (DFS): Properties

Theorem (Parenthesis Theorem)

For any pair of nodes $v, u \in V$, either:

- $[u.d, u.f] \cap [v.d, v.f] = \emptyset$ *and neither u is a descendant of v nor v of u*
- $[u.d, u.f] \subsetneq [v.d, v.f]$ *and u is a descendant of v*
- $[v.d, v.f] \subsetneq [u.d, u.f]$ *and v is a descendant of u*

Depth-First-Search (DFS): Properties

Theorem (Parenthesis Theorem)

For any pair of nodes $v, u \in V$, either:

- $[u.d, u.f] \cap [v.d, v.f] = \emptyset$ *and neither u is a descendant of v nor v of u*
- $[u.d, u.f] \subsetneq [v.d, v.f]$ *and u is a descendant of v*
- $[v.d, v.f] \subsetneq [u.d, u.f]$ *and v is a descendant of u*

Theorem (White-Path Theorem)

For any pair of nodes $v, u \in V$, u is a descendant of v iff at time $v.d - 1$ there exists a WHITE-only path from v to u.

Topological Sort

How to Prepare. . . Tiramisù

It is quite simple

We have to:

How to Prepare. . . Tiramisù

It is quite simple

We have to:

- beating the egg whites until still
- adding sugar
- preparing coffee
- soaking the Savoiardi (NOT Pavesini, please!) cookies in coffee
- incorporating mascarpone cheese
- beating the egg yolks
- adding a drop of marsala
- incorporating whites and yolks

How to Prepare... Tiramisù

It is quite simple

We have to:

- beating the egg whites until still
- adding sugar
- preparing coffee
- soaking the Savoiardi (NOT Pavesini, please!) cookies in coffee
- incorporating mascarpone cheese
- beating the egg yolks
- adding a drop of marsala
- incorporating whites and yolks

Timing and sub-task ordering is fundamental

Installing Software

In a similar way, trying to install a software can be a "sorting" problem

Installing Software

In a similar way, trying to install a software can be a “sorting” problem

E.g., To install software A solve following dependencies

- Software A needs software B, C, D
- Software B depends on libraries E and F and on software G
- Software G depends on library E and on D
- ...

How to figure out what is needed and in which order?

Dependency Relations and Topological Sort

Dependency relations can be modeled by using directed graphs

Nodes represent sub-tasks, edges the needs, e.g., $(v, u) \in E$ iff $v=$"soaking cookies" needs $u=$"preparing coffee"

Dependency Relations and Topological Sort

Dependency relations can be modeled by using directed graphs

Nodes represent sub-tasks, edges the needs, e.g., $(v, u) \in E$ iff $v =$ "soaking cookies" needs $u =$ "preparing coffee"

On DAGs, there exists a topological order, $\preceq_T$, on nodes satisfying

$$\text{If } (u, v) \in E, \text{ then } u \preceq_T v$$

Computing Topological Sort and Complexity

```
def TOPOLOGICAL_SORT(G)
    call DFS(G)


enddef
```

Computing Topological Sort and Complexity

def TOPOLOGICAL_SORT(G)
call DFS(G)
as a node is finalized, push it into a stack S

enddef

Computing Topological Sort and Complexity

def TOPOLOGICAL_SORT(G)
  call DFS(G)
  as a node is finalized, push it into a stack *S*
  return S
enddef

Computing Topological Sort and Complexity

def TOPOLOGICAL_SORT(G)
call DFS(G)
as a node is finalized, push it into a stack S
return S
enddef

Asymptotic complexity $\Theta(|V| + |E|)$

Topological Sort: Correctness

Lemma

Let $[v_1, \ldots, v_n]$ be the output of TOPOLOGICAL_SORT(G). Then $v_i.f > v_{i+1}.f$ for all $i \in [1, n-1]$

Topological Sort: Correctness

Lemma

Let $[v_1, \dots, v_n]$ be the output of TOPOLOGICAL_SORT(G). Then $v_i.f > v_{i+1}.f$ for all $i \in [1, n-1]$

Lemma

G contains cycles iff DFS(G) yields back edges

Topological Sort: Correctness

Lemma

Let $[v_1, \ldots, v_n]$ be the output of TOPOLOGICAL_SORT(G). Then $v_i.f > v_{i+1}.f$ for all $i \in [1, n-1]$

Lemma

G contains cycles iff DFS(G) yields back edges

Theorem

If G is a DAG, then TOPOLOGICAL_SORT(G) produces G's topological sort

Strongly Connected Components

What Comes Before Egg or Hen?

When a graph is not a DAG, establishing a topological ordering is not possible

From the reachability point of view, all the nodes in a loop behave in the same way

If one of them is reachable from a node, so are all the others

If one of them can reach a node, so do all the others

What Comes Before Egg or Hen?

When a graph is not a DAG, establishing a topological ordering is not possible

From the reachability point of view, all the nodes in a loop behave in the same way

If one of them is reachable from a node, so are all the others

If one of them can reach a node, so do all the others

Discovering equivalent nodes is a useful task

Strongly Connected Components (SCCs)

Are maximum sub-graphs (of a directed graph) s.t., for every pair of their nodes, there is a path from one to the other and vice versa

Strongly Connected Components (SCCs)

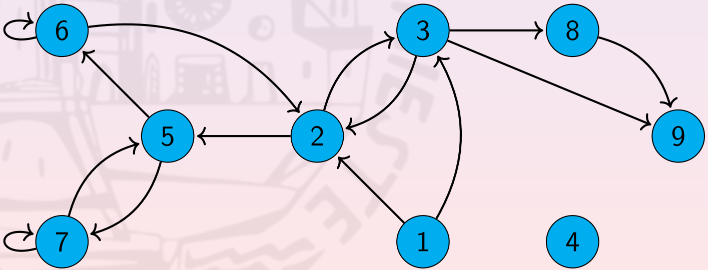
Are maximum sub-graphs (of a directed graph) s.t., for every pair of their nodes, there is a path from one to the other and vice versa

SCCs partition the nodes of any graph

Strongly Connected Components (SCCs)

Are maximum sub-graphs (of a directed graph) s.t., for every pair of their nodes, there is a path from one to the other and vice versa

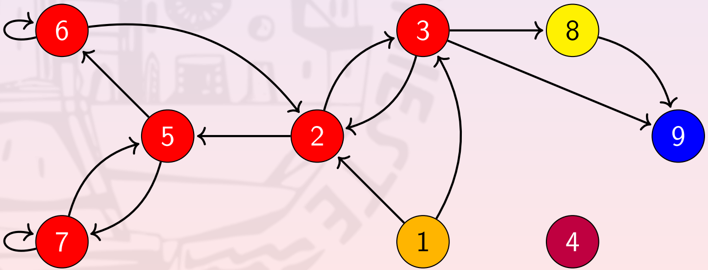
SCCs partition the nodes of any graph



Strongly Connected Components (SCCs)

Are maximum sub-graphs (of a directed graph) s.t., for every pair of their nodes, there is a path from one to the other and vice versa

SCCs partition the nodes of any graph



How to Identify SCCs?

Any idea?

How to Identify SCCs?

Any idea? What about DFS to identify SCC loops?

Lemma

G contains cycles iff DFS(G) yields back edges

How to Identify SCCs?

Any idea? What about DFS to identify SCC loops?

Lemma

G contains cycles iff $DFS(G)$ yields back edges

Fine, but how to "announce" to a node that it is in a loop?

How to Identify SCCs?

Any idea? What about DFS to identify SCC loops?

Lemma

G contains cycles iff DFS(G) yields back edges

Fine, but how to "announce" to a node that it is in a loop?

Minimum Discovery Time from the Sub-Tree (lowlink): if it is smaller than the node discovery time, then a back edge must be reachable from it

One DFS_VISIT Call to Rule Them All

One DFS_VISIT call discovers all the nodes of the SCCs it "touches", i.e., no half visited SCC

DFS_VISITs do not interleave nodes of two distinct SCCs.

One DFS_VISIT Call to Rule Them All

One DFS_VISIT call discovers all the nodes of the SCCs it "touches", i.e., no half visited SCC

DFS_VISITs do not interleave nodes of two distinct SCCs.

So, if we have a way to:

- perform a DFS_VISIT call and update lowlinks when possible

One DFS_VISIT Call to Rule Them All

One DFS_VISIT call discovers all the nodes of the SCCs it "touches", i.e., no half visited SCC

DFS_VISITs do not interleave nodes of two distinct SCCs.

So, if we have a way to:

- perform a DFS_VISIT call and update lowlinks when possible
- identify a SCC as soon as all its nodes have been visited

One DFS_VISIT Call to Rule Them All

One DFS_VISIT call discovers all the nodes of the SCCs it "touches", i.e., no half visited SCC

DFS_VISITs do not interleave nodes of two distinct SCCs.

So, if we have a way to:

- perform a DFS_VISIT call and update lowlinks when possible
- identify a SCC as soon as all its nodes have been visited
- label the SCC nodes as "not available for lowlink updates"

One DFS_VISIT Call to Rule Them All

One DFS_VISIT call discovers all the nodes of the SCCs it "touches", i.e., no half visited SCC

DFS_VISITs do not interleave nodes of two distinct SCCs.

So, if we have a way to:

- perform a DFS_VISIT call and update lowlinks when possible
- identify a SCC as soon as all its nodes have been visited
- label the SCC nodes as "not available for lowlink updates"

then we have an algorithm to identify all the SCCs

Two Useful Intuitions

How are related the discovery time and the lowlink of the first node S_f in a SCC S to be visited?

Two Useful Intuitions

How are related the discovery time and the lowlink of the first node S_f in a SCC S to be visited?

They are the same!!!

So, once S_f has been finished, all the nodes in S have too

Two Useful Intuitions

How are related the discovery time and the lowlink of the first node S_f in a SCC S to be visited?

They are the same!!!

So, once S_f has been finished, all the nodes in S have too

Which nodes in the visit belong to the just identified SCC if we not consider the nodes of the already discovered SCCs?

Two Useful Intuitions

How are related the discovery time and the lowlink of the first node S_f in a SCC S to be visited?

They are the same!!!

So, once S_f has been finished, all the nodes in S have too

Which nodes in the visit belong to the just identified SCC if we not consider the nodes of the already discovered SCCs?

The last ones!

We can use a stack to store finished nodes and use lowlink property to detect S_f

Tarjan's SCCs Algorithm: Pseudo-Code

```
def TARJAN_SCC(G):
  for v in G.V:
    v.color ← WHITE
  endfor

  time ← 0
  S ← BUILD_STACK()
  for v in G.V:
    if v.color = WHITE:
      time ← TARJAN_SCC_VISIT(G,v,S,time)
    endif
  endfor
enddef
```

Tarjan's SCCs Algorithm: Pseudo-Code (Cont'd)

```
def TARJAN_SCC_VISIT(G,v,S,time):
  time ← time + 1
  v.d ← time
  v.color = GRAY

  v.lowlink ← time
  S.push(v)
  v.onStack ← True

  for u in G.Adj[v]:
    if u.color = WHITE:
      time ← TARJAN_SCC_VISIT(G,u,S,time)
```

Tarjan's SCCs Algorithm: Pseudo-Code (Cont'd 2)

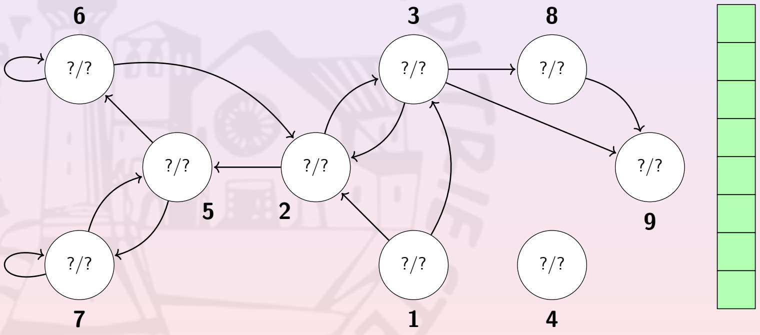
```
        v.lowlink ← min(v.lowlink, u.lowlink)
      else if u.onStack:
        v.lowlink ← min(v.lowlink, u.d)
      endif
    endfor

    if v.lowlink = v.d:
      yield EXTRACT_SCC_FROM_STACK(S,v)
    endif

    return time
enddef
```

Tarjan's SCCs Algorithm: Pseudo-Code (Cont'd 3)

```
def EXTRACT_SCC_FROM_STACK(S, v):
  L ← EMPTY_LIST()

  repeat:
    w ← S.pop()
    w.onStack ← False

    L.append(w)
  until v ≠ w

  return L
enddef
```

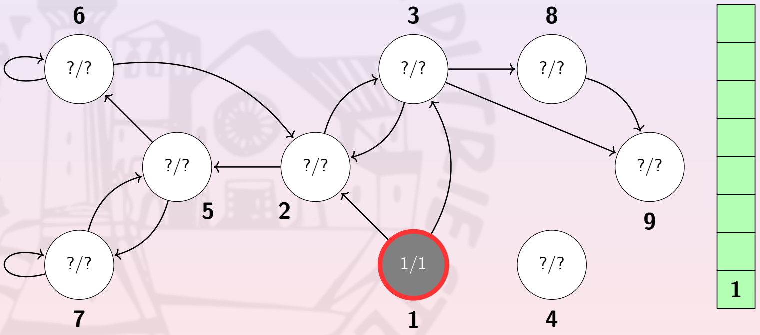
Tarjan's SCCs Algorithm: Example

Nodes are labeled by "Discovery Time" / "Lowlink"



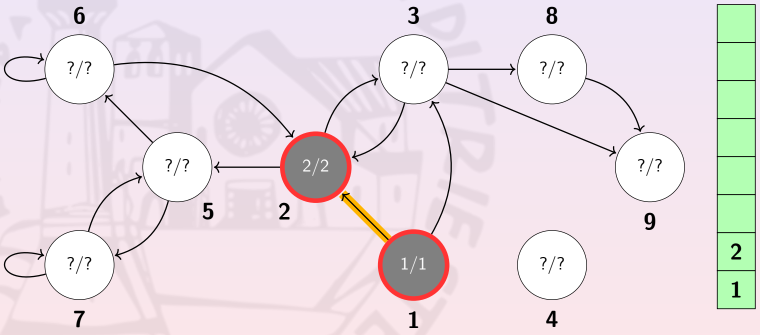
Tarjan's SCCs Algorithm: Example

Nodes are labeled by "Discovery Time" / "Lowlink"



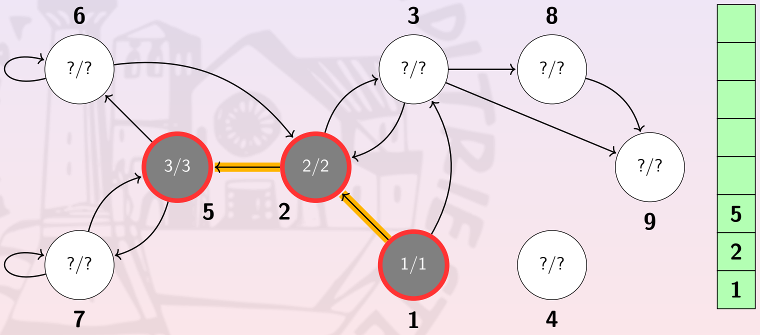
Tarjan's SCCs Algorithm: Example

Nodes are labeled by "Discovery Time" / "Lowlink"



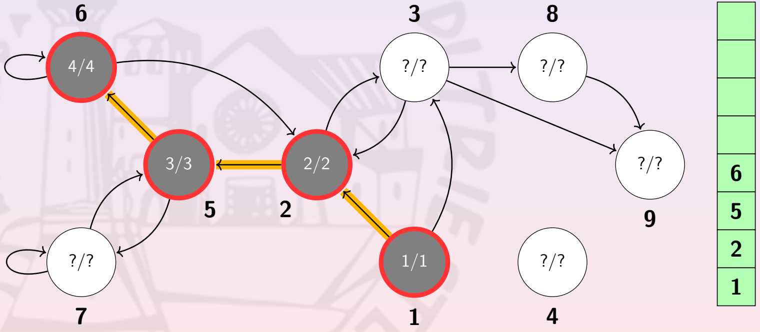
Tarjan's SCCs Algorithm: Example

Nodes are labeled by "Discovery Time" / "Lowlink"



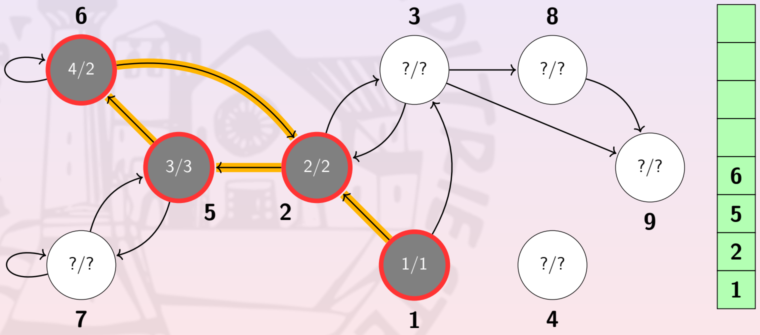
Tarjan's SCCs Algorithm: Example

Nodes are labeled by "Discovery Time" / "Lowlink"



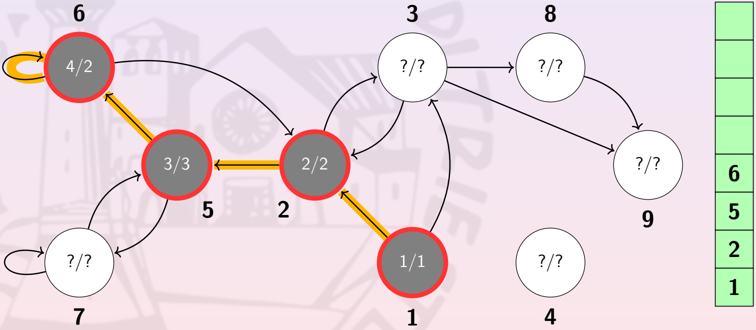
Tarjan's SCCs Algorithm: Example

Nodes are labeled by "Discovery Time" / "Lowlink"



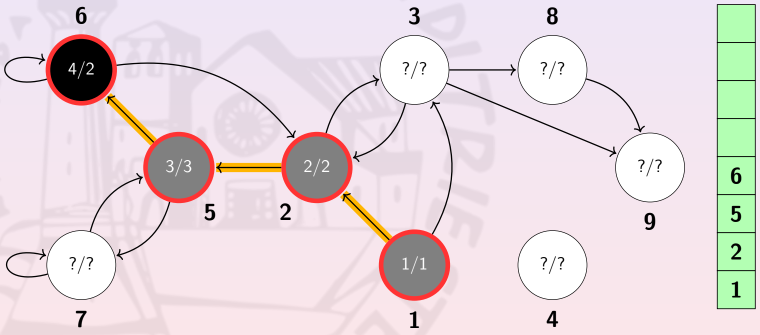
Tarjan's SCCs Algorithm: Example

Nodes are labeled by "Discovery Time" / "Lowlink"



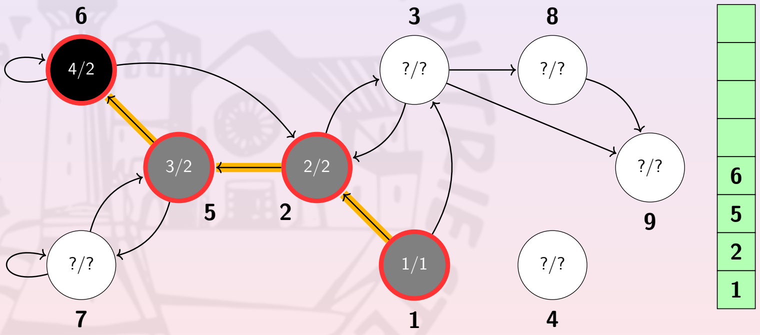
Tarjan's SCCs Algorithm: Example

Nodes are labeled by "Discovery Time" / "Lowlink"



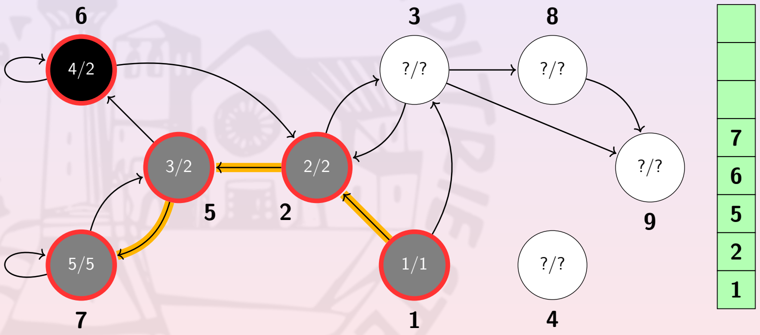
Tarjan's SCCs Algorithm: Example

Nodes are labeled by "Discovery Time" / "Lowlink"



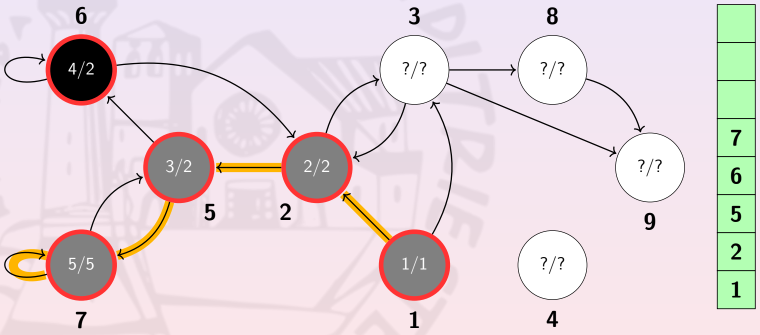
Tarjan's SCCs Algorithm: Example

Nodes are labeled by "Discovery Time" / "Lowlink"



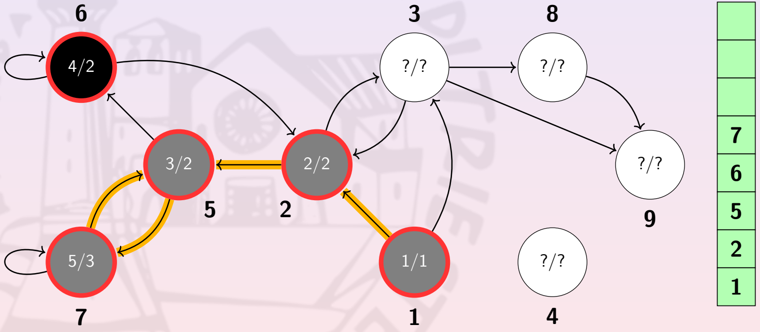
Tarjan's SCCs Algorithm: Example

Nodes are labeled by "Discovery Time" / "Lowlink"



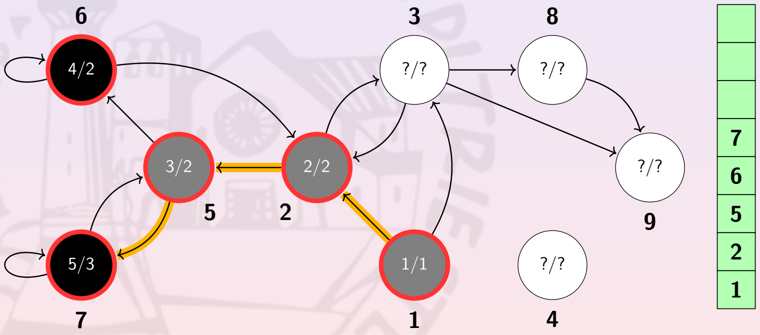
Tarjan's SCCs Algorithm: Example

Nodes are labeled by "Discovery Time" / "Lowlink"



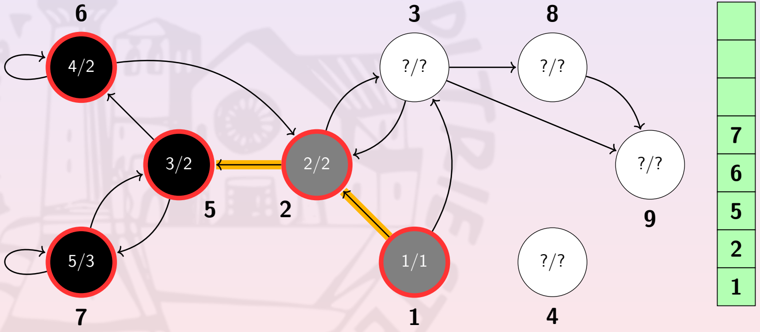
Tarjan's SCCs Algorithm: Example

Nodes are labeled by "Discovery Time" / "Lowlink"



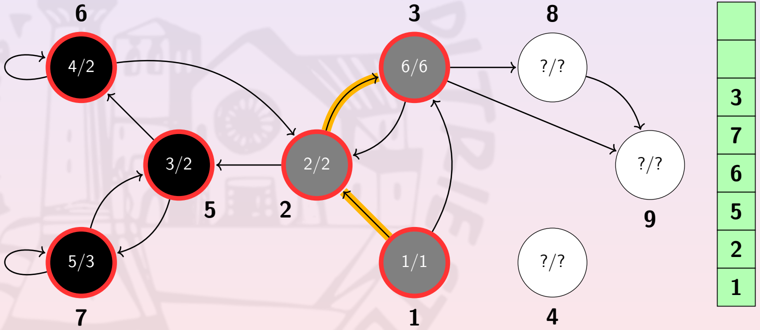
Tarjan's SCCs Algorithm: Example

Nodes are labeled by "Discovery Time" / "Lowlink"



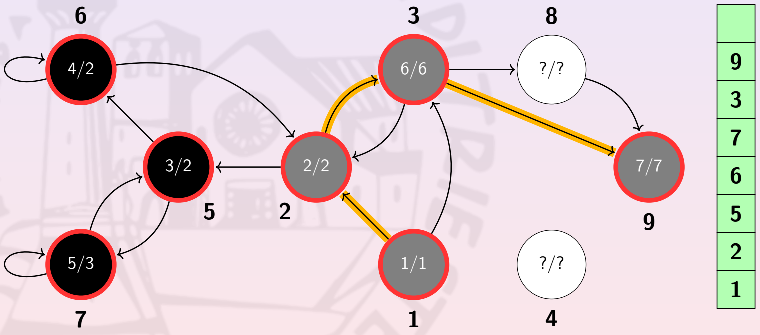
Tarjan's SCCs Algorithm: Example

Nodes are labeled by "Discovery Time" / "Lowlink"



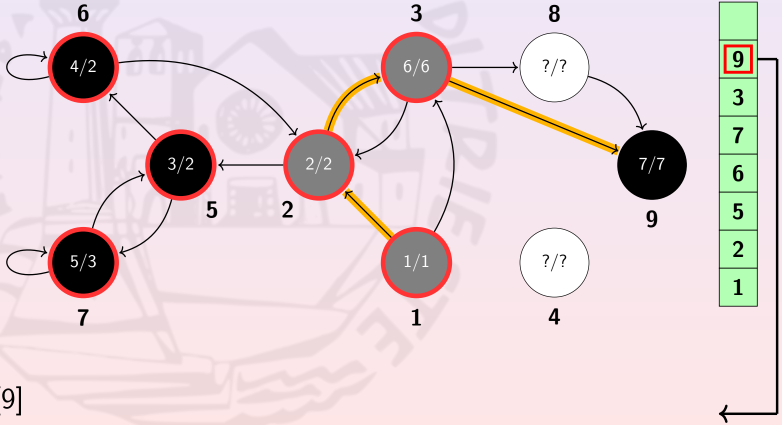
Tarjan's SCCs Algorithm: Example

Nodes are labeled by "Discovery Time" / "Lowlink"



Tarjan's SCCs Algorithm: Example

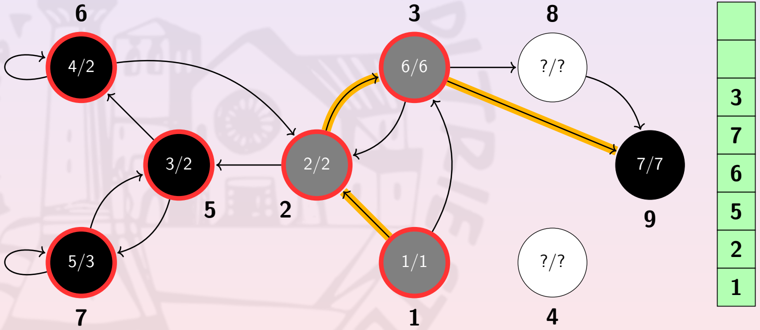
Nodes are labeled by "Discovery Time" / "Lowlink"



[9]

Tarjan's SCCs Algorithm: Example

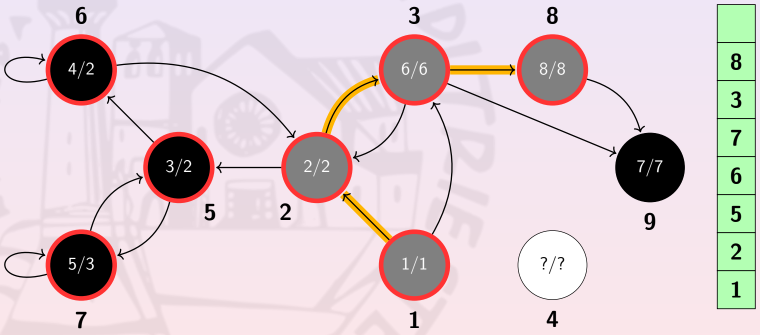
Nodes are labeled by "Discovery Time" / "Lowlink"



[9]

Tarjan's SCCs Algorithm: Example

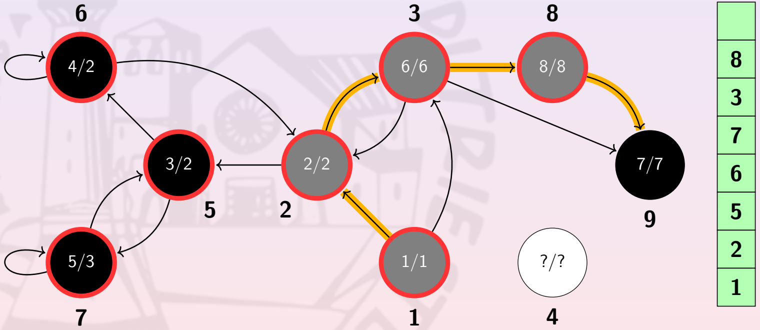
Nodes are labeled by "Discovery Time" / "Lowlink"



[9]

Tarjan's SCCs Algorithm: Example

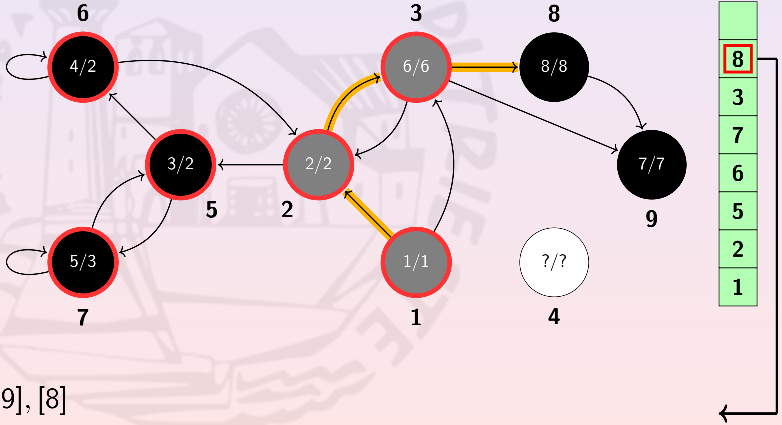
Nodes are labeled by "Discovery Time" / "Lowlink"



[9], [8]

Tarjan's SCCs Algorithm: Example

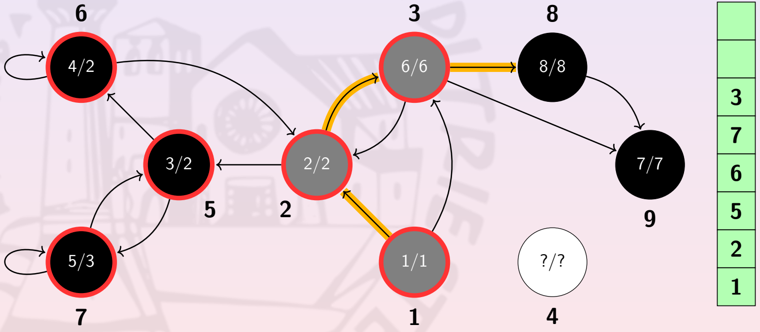
Nodes are labeled by "Discovery Time" / "Lowlink"



[9], [8]

Tarjan's SCCs Algorithm: Example

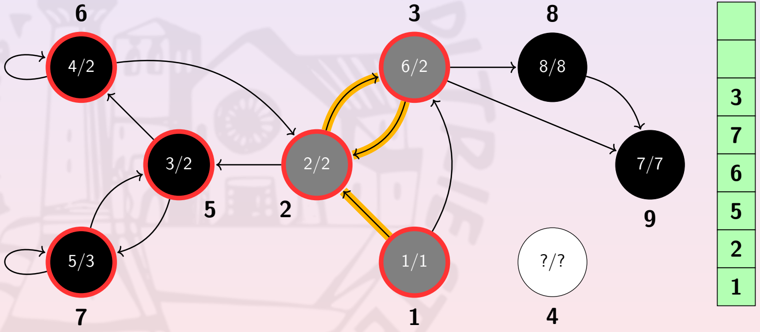
Nodes are labeled by "Discovery Time" / "Lowlink"



[9], [8]

Tarjan's SCCs Algorithm: Example

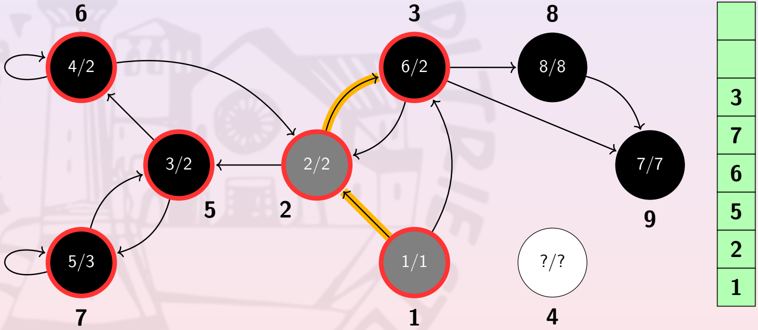
Nodes are labeled by "Discovery Time" / "Lowlink"



[9], [8]

Tarjan's SCCs Algorithm: Example

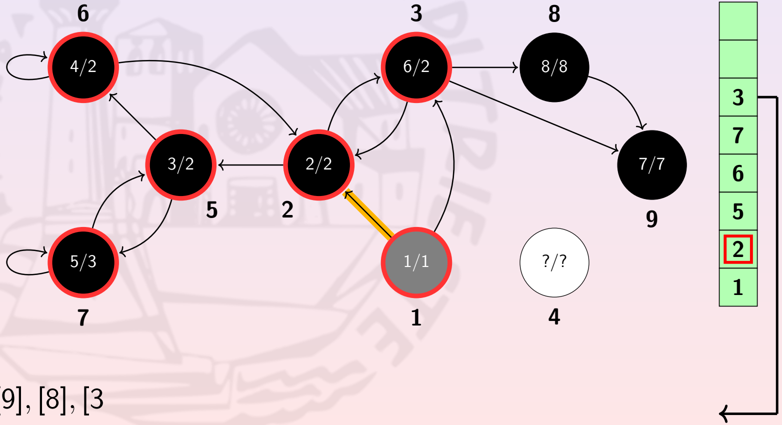
Nodes are labeled by "Discovery Time" / "Lowlink"



[9], [8]

Tarjan's SCCs Algorithm: Example

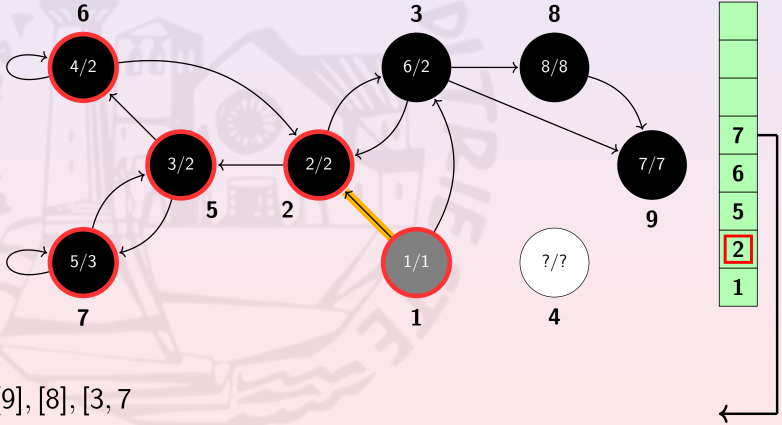
Nodes are labeled by "Discovery Time" / "Lowlink"



[9], [8], [3

Tarjan's SCCs Algorithm: Example

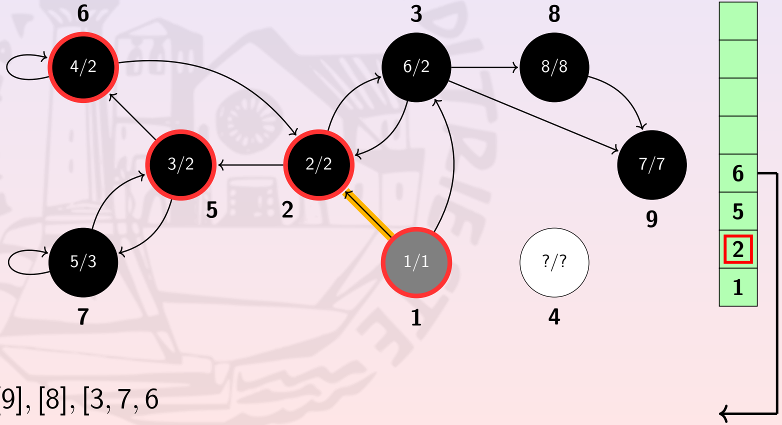
Nodes are labeled by "Discovery Time" / "Lowlink"



[9], [8], [3, 7

Tarjan's SCCs Algorithm: Example

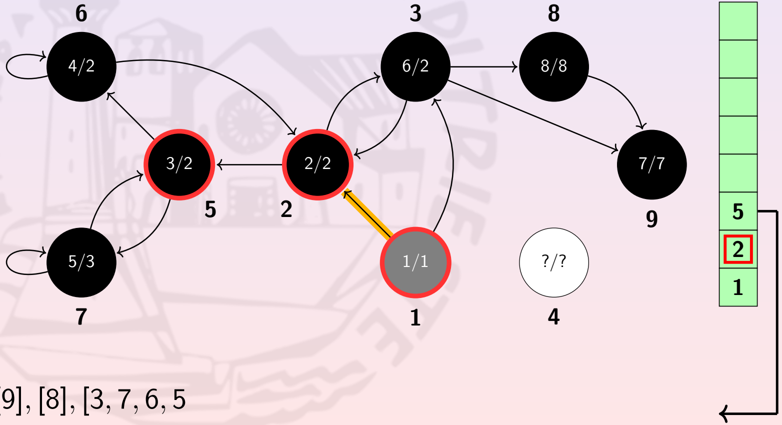
Nodes are labeled by "Discovery Time" / "Lowlink"



[9], [8], [3, 7, 6

Tarjan's SCCs Algorithm: Example

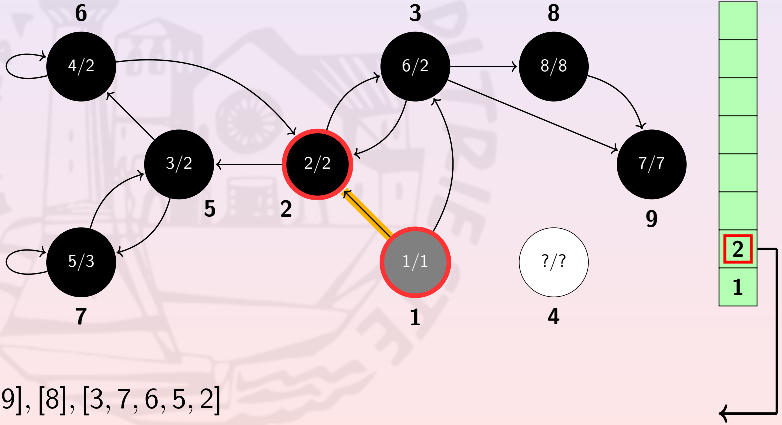
Nodes are labeled by "Discovery Time" / "Lowlink"



[9], [8], [3, 7, 6, 5

Tarjan's SCCs Algorithm: Example

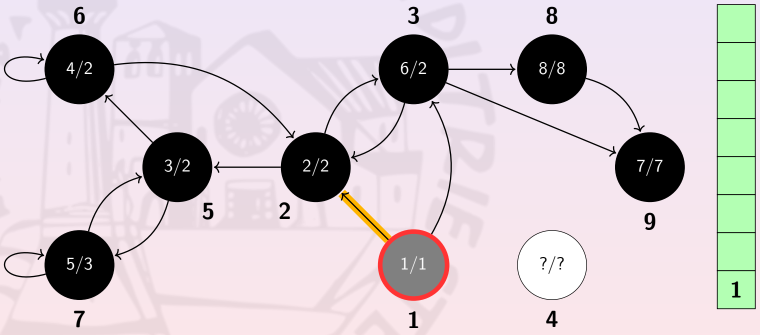
Nodes are labeled by "Discovery Time" / "Lowlink"



[9], [8], [3, 7, 6, 5, 2]

Tarjan's SCCs Algorithm: Example

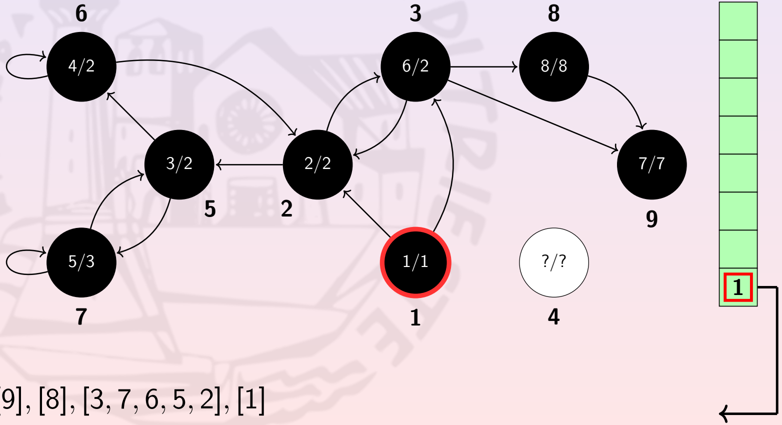
Nodes are labeled by "Discovery Time" / "Lowlink"



[9], [8], [3, 7, 6, 5, 2]

Tarjan's SCCs Algorithm: Example

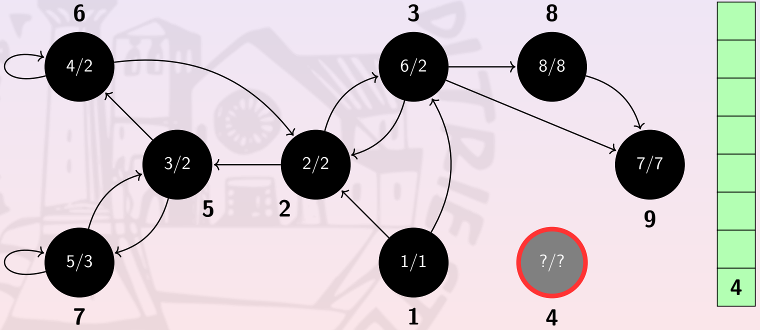
Nodes are labeled by "Discovery Time" / "Lowlink"



[9], [8], [3, 7, 6, 5, 2], [1]

Tarjan's SCCs Algorithm: Example

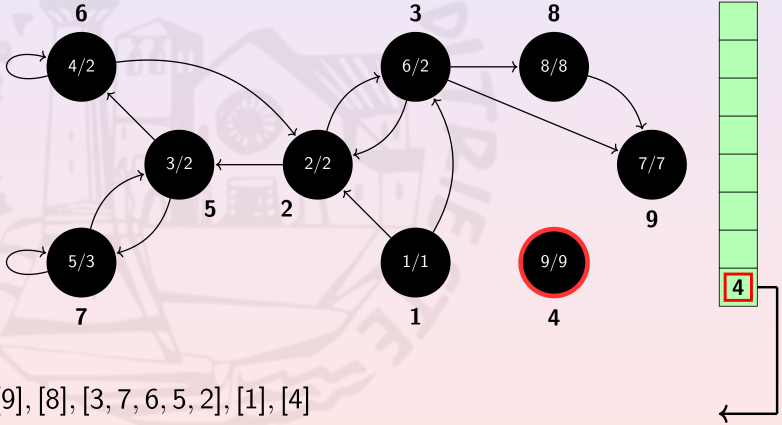
Nodes are labeled by "Discovery Time" / "Lowlink"



[9], [8], [3, 7, 6, 5, 2], [1]

Tarjan's SCCs Algorithm: Example

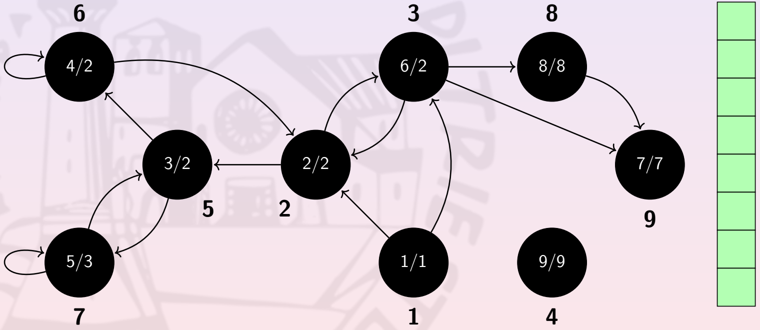
Nodes are labeled by "Discovery Time" / "Lowlink"



[9], [8], [3, 7, 6, 5, 2], [1], [4]

Tarjan's SCCs Algorithm: Example

Nodes are labeled by "Discovery Time" / "Lowlink"



[9], [8], [3, 7, 6, 5, 2], [1], [4]

Tarjan's SCCs Algorithm: Complexity

The algorithm performs a DFS-like visit + stack handling

One single node is pushed in S during each `TARJAN_SCC_VISIT` call

`TARJAN_SCC_VISIT` is called on WHITE nodes and sets them to GRAY

So, the # of node inserted into S at some point is $|V|$

All `EXTRACT_SCC_FROM_STACK` calls cumulatively cost $\Theta(|V|)$

Tarjan's algorithm costs $\Theta(|V| + |E|)$

Transitive Closure

Transitive Closure: Definition and Naïve Solution

For each pair of nodes v and w, we would like to know whether there is a path from v to w

How to solve this problem?

Transitive Closure: Definition and Naïve Solution

For each pair of nodes v and w, we would like to know whether there is a path from v to w

How to solve this problem?

Naïve solution: evaluate BFS from all the graph nodes

Complexity?

Transitive Closure: Definition and Naïve Solution

For each pair of nodes v and w, we would like to know whether there is a path from v to w

How to solve this problem?

Naïve solution: evaluate BFS from all the graph nodes

Complexity? $|V| * O(|V| + |E|) = O(|V|^2 + |V| * |E|)$

Any other ideas?

Have a Look at the Matrix Formulation of the Problem

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 1 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & 0 \\ 3 & 0 & 0 & 1 & 0 \\ 4 & 0 & \boxed{1} & 0 & 1 \end{array}$$

- w has distance 1 from v iff $A[v][w] \neq 0$

Have a Look at the Matrix Formulation of the Problem

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 1 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & 0 \\ 3 & 0 & 0 & 1 & 0 \\ 4 & 0 & \boxed{1} & 0 & 1 \end{array} \qquad \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 1 & 0 & 0 & 0 \\ 2 & \boxed{1} & 1 & 0 & 0 \\ 3 & 0 & 0 & 1 & 0 \\ 4 & 0 & 1 & 0 & 1 \end{array}$$

- w has distance 1 from v iff $A[v][w] \neq 0$
- w has distance 2 from v iff there exists z s.t. $A[v][z] \neq 0$ and $A[z][w] \neq 0$

Have a Look at the Matrix Formulation of the Problem

$$
\begin{array}{c|cccc|}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 1 & 0 & 0 & 0 \\
2 & 1 & 1 & 0 & 0 \\
3 & 0 & 0 & 1 & 0 \\
4 & 0 & \boxed{1} & 0 & 1 \\
\hline
\end{array}
\times
\begin{array}{c|cccc|}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 1 & 0 & 0 & 0 \\
2 & \boxed{1} & 1 & 0 & 0 \\
3 & 0 & 0 & 1 & 0 \\
4 & 0 & 1 & 0 & 1 \\
\hline
\end{array}
$$

- w has distance 1 from v iff $A[v][w] \neq 0$
- w has distance 2 from v iff there exists z s.t. $A[v][z] \neq 0$ and $A[z][w] \neq 0 \iff (A \times A)[v][w] > 0$

Have a Look at the Matrix Formulation of the Problem

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 1 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & 0 \\ 3 & 0 & 0 & 1 & 0 \\ 4 & 0 & \boxed{1} & 0 & 1 \end{array} \times \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 1 & 0 & 0 & 0 \\ 2 & \boxed{1} & 1 & 0 & 0 \\ 3 & 0 & 0 & 1 & 0 \\ 4 & 0 & 1 & 0 & 1 \end{array}$$

- w has distance 1 from v iff $A[v][w] \neq 0$
- w has distance 2 from v iff there exists z s.t. $A[v][z] \neq 0$ and $A[z][w] \neq 0 \iff (A \times A)[v][w] > 0$
- w has distance $\leq n$ from v iff $A^n[v][w] > 0$

Transitive Closure By Matrix Multiplication

Every acyclic path has at most length $|V|$

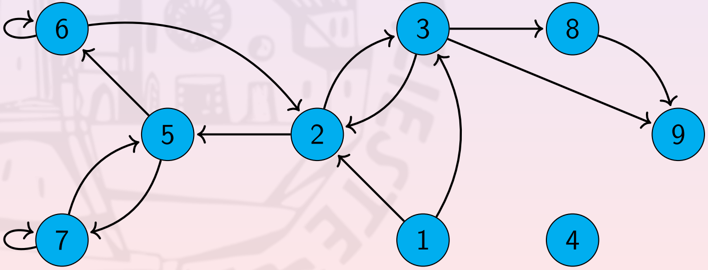
Wen can solve the problem by using Strassen's algorithm:
$(|V| - 1) * \Theta(|V|^{\log_2 7})$

Which is worst than using BFS!!!

SCCs Are Unnecessarily Cumbersome

A.f.a. reachability concerns, all the nodes in a SCC are the same

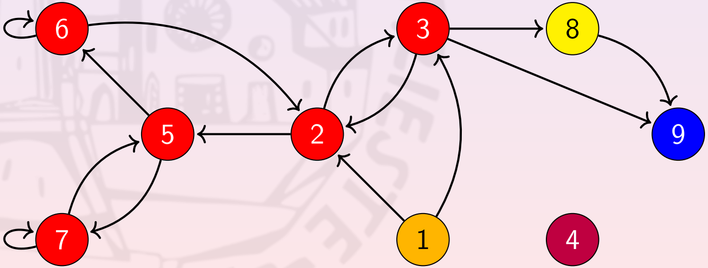
So, collapse the nodes in SCCs and build the SCCs graph $\bar{G}$



SCCs Are Unnecessarily Cumbersome

A.f.a. reachability concerns, all the nodes in a SCC are the same

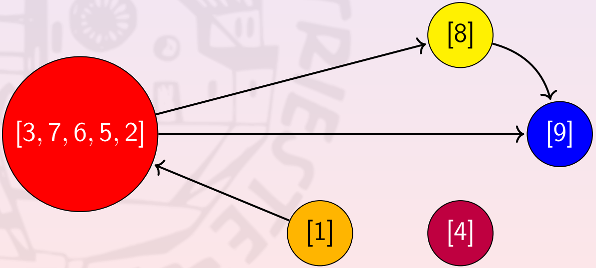
So, collapse the nodes in SCCs and build the SCCs graph $\bar{G}$



SCCs Are Unnecessarily Cumbersome

A.f.a. reachability concerns, all the nodes in a SCC are the same

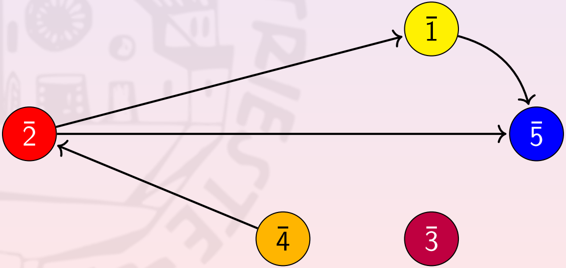
So, collapse the nodes in SCCs and build the SCCs graph $\bar{G}$



SCCs Are Unnecessarily Cumbersome

A.f.a. reachability concerns, all the nodes in a SCC are the same

So, collapse the nodes in SCCs and build the SCCs graph $\bar{G}$



And Then What?

And Then What?

Topological Sort!!!

And Then What?

Topological Sort!!!

Why?

And Then What?

Topological Sort!!!

Why? Have a look at the adjacency matrix of $\bar{G}$

	$\bar{1}$	$\bar{2}$	$\bar{3}$	$\bar{4}$	$\bar{5}$
$\bar{1}$	1	0	0	0	1
$\bar{2}$	1	1	0	0	1
$\bar{3}$	0	0	1	0	0
$\bar{4}$	0	1	0	1	0
$\bar{5}$	0	0	0	0	1

And Then What?

Topological Sort!!!

Why? Have a look at the adjacency matrix of $\bar{G}$

$$\begin{array}{c|ccccc} & \bar{1} & \bar{2} & \bar{3} & \bar{4} & \bar{5} \\ \hline \bar{1} & 1 & 0 & 0 & 0 & 1 \\ \bar{2} & 1 & 1 & 0 & 0 & 1 \\ \bar{3} & 0 & 0 & 1 & 0 & 0 \\ \bar{4} & 0 & 1 & 0 & 1 & 0 \\ \bar{5} & 0 & 0 & 0 & 0 & 1 \end{array} \Longrightarrow \begin{array}{c|ccccc} & \bar{3} & \bar{4} & \bar{2} & \bar{1} & \bar{5} \\ \hline \bar{3} & 1 & 0 & 0 & 0 & 0 \\ \bar{4} & 0 & 1 & 1 & 0 & 1 \\ \bar{2} & 0 & 0 & 1 & 1 & 1 \\ \bar{1} & 0 & 0 & 0 & 1 & 1 \\ \bar{5} & 0 & 0 & 0 & 0 & 1 \end{array}$$

The new adjacency matrix is upper-triangular

And Then What?

Topological Sort!!! Tarjan's algorithm already did it!

Why? Have a look at the adjacency matrix of $\bar{G}$

	$\bar{1}$	$\bar{2}$	$\bar{3}$	$\bar{4}$	$\bar{5}$
$\bar{1}$	1	0	0	0	1
$\bar{2}$	1	1	0	0	1
$\bar{3}$	0	0	1	0	0
$\bar{4}$	0	1	0	1	0
$\bar{5}$	0	0	0	0	1

$$\Longrightarrow$$

	$\bar{3}$	$\bar{4}$	$\bar{2}$	$\bar{1}$	$\bar{5}$
$\bar{3}$	1	0	0	0	0
$\bar{4}$	0	1	1	0	1
$\bar{2}$	0	0	1	1	1
$\bar{1}$	0	0	0	1	1
$\bar{5}$	0	0	0	0	1

The new adjacency matrix is upper-triangular

Why Upper Triangular Matrixes Are Nice?

$$\bar{G} = \left(\begin{array}{c|c} A & C \\ \hline 0 & B \end{array}\right)$$

Why Upper Triangular Matrixes Are Nice?

$$\bar{G} = \left(\begin{array}{c|c} \color{red}{\mathbf{A}} & C \\ \hline 0 & B \end{array}\right)$$

Why Upper Triangular Matrixes Are Nice?

$$\bar{G} = \left(\begin{array}{c|c} \mathbf{A} & C \\ \hline 0 & \mathbf{B} \end{array} \right)$$

Why Upper Triangular Matrixes Are Nice?

$$\bar{G} = \left(\begin{array}{c|c} \mathbf{A} & \mathbf{C} \\ \hline 0 & \mathbf{B} \end{array}\right)$$

Why Upper Triangular Matrixes Are Nice?

$$\bar{G} = \left(\begin{array}{c|c} \mathbf{A} & \mathbf{C} \\ \hline 0 & \mathbf{B} \end{array}\right)$$

Any path in $\bar{G}$ can contain at most one of the edges in C

Why Upper Triangular Matrixes Are Nice?

$$\bar{G} = \left(\begin{array}{c|c} \mathbf{A} & \mathbf{C} \\ \hline 0 & \mathbf{B} \end{array}\right)$$

Any path in $\bar{G}$ can contain at most one of the edges in C

$$\bar{G}^* = \left(\begin{array}{c|c} A^* & A^* \times C \times B^* \\ \hline 0 & B^* \end{array}\right)$$

The Complete Algorithm

- Tarjan's SCCs Algorithm on the input gran G: $\Theta(|V| + |E|)$
- Build the SCCs Graph $\bar{G}$ of G: $\Theta(|E|)$
- Topological sort of $\bar{G}$: $\Theta(|V| + |E|)$
- Compute $\bar{G}^*$ in time:

$$T(n) = 2 * T(n/2) + 2 * \Theta(n^{\log_2 7})$$

- Extend the transitive closure of $\bar{G}$ to G: $O(|V|^2)$

The Complete Algorithm

- Tarjan's SCCs Algorithm on the input gran G: $\Theta(|V| + |E|)$
- Build the SCCs Graph $\bar{G}$ of G: $\Theta(|E|)$
- Topological sort of $\bar{G}$: $\Theta(|V| + |E|)$
- Compute $\bar{G}^*$ in time:

  $$T(n) = 2 * T(n/2) + 2 * \Theta(n^{\log_2 7})$$

  It can be proved that $T(|V|) \in \Theta(|V|^{\log_2 7})$
- Extend the transitive closure of $\bar{G}$ to G: $O(|V|^2)$

The Complete Algorithm

- Tarjan's SCCs Algorithm on the input gran G: $\Theta(|V| + |E|)$
- Build the SCCs Graph $\bar{G}$ of G: $\Theta(|E|)$
- Topological sort of $\bar{G}$: $\Theta(|V| + |E|)$
- Compute $\bar{G}^*$ in time:

$$T(n) = 2 * T(n/2) + 2 * \Theta(n^{\log_2 7})$$

  It can be proved that $T(|V|) \in \Theta(|V|^{\log_2 7})$
- Extend the transitive closure of $\bar{G}$ to G: $O(|V|^2)$

The overall asymptotic complexity is $\Theta(|E| + |V|^{\log_2 7})$