Advanced Algorithmic Techniques (COMP523)

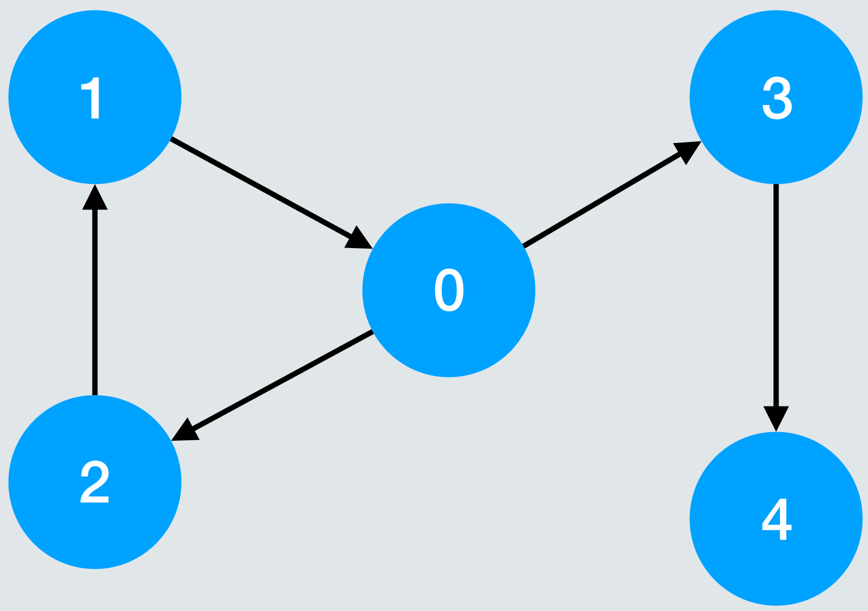
Graph Algorithms #3

Recap and plan

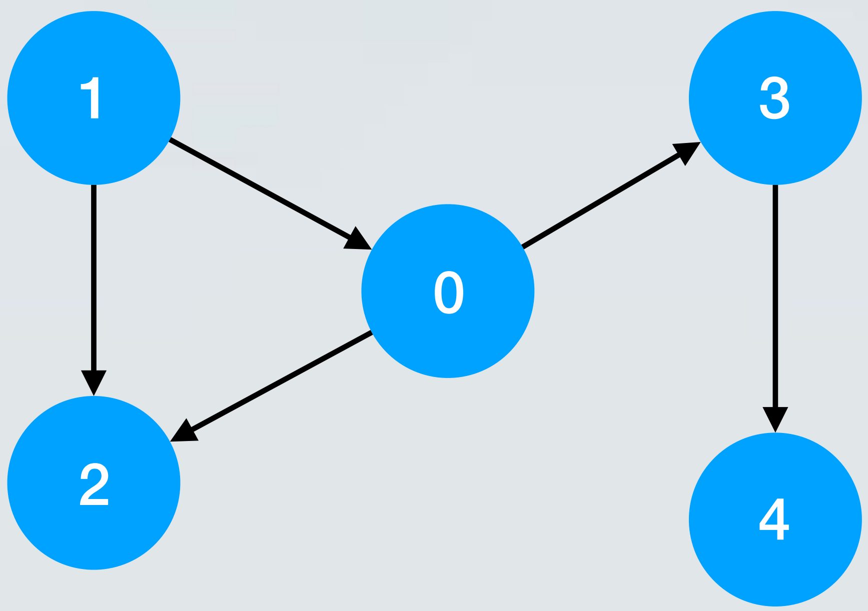
- **Last lecture:**
  - Testing bipartiteness
  - DFS and BFS on directed graphs
  - Testing connectivity
- **This lecture:**
  - Directed Acyclic Graphs (DAG)
  - Topological Ordering
  - Finding strongly connected components

Directed Acyclic Graphs

- A directed acyclic graph (DAG) G is a graph that does not have any cycles.



not a DAG



a DAG

Properties on DAGs

Properties on DAGs

- They appear quite often in many applications.

Properties on DAGs

- They appear quite often in many applications.
- Example - prerequisite modules: To take module A you need to have taken module B and module C.

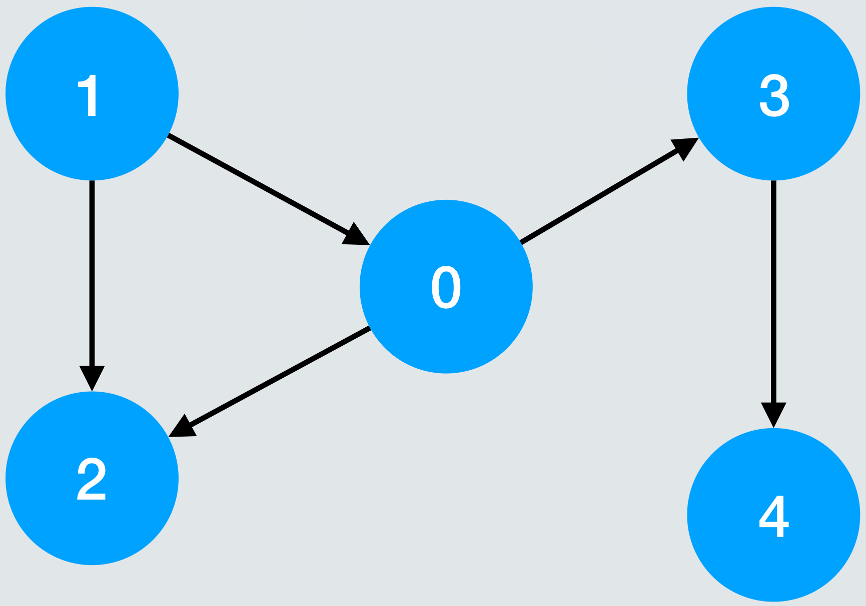
Properties on DAGs

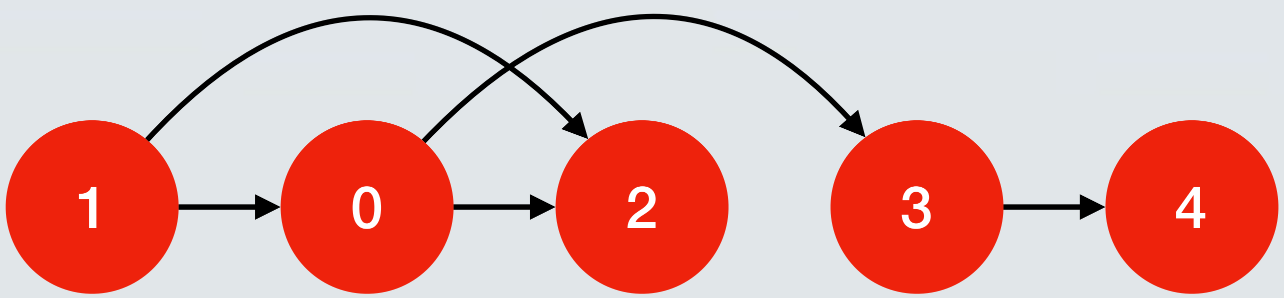
- They appear quite often in many applications.
- Example - prerequisite modules: To take module A you need to have taken module B and module C.
- If the module prerequisite relation has a cycle, then it is impossible to get a degree!

Topological Ordering

- Given a directed graph G, a topological ordering of G is an ordering of the nodes $u_1, u_2, \ldots, u_n$, such that for every edge $e=(u_i, u_j)$, it holds that $i < j$.

- Intuitively, a topological ordering orders the nodes in a way such that all edges point "forward".





Topological Ordering implies DAG

Topological Ordering implies DAG

- If graph G has a topological ordering, then G is a DAG.

Topological Ordering implies DAG

- If graph G has a topological ordering, then G is a DAG.
- Suppose by contradiction that G has a topological ordering (u_1, u_2, … , u_n) but it also has a cycle C.

Topological Ordering implies DAG

- If graph G has a topological ordering, then G is a DAG.
- Suppose by contradiction that G has a topological ordering (u_1, u_2, … , u_n) but it also has a cycle C.
- Let u_j be the smallest element of C according to the topological ordering.

Topological Ordering implies DAG

- If graph G has a topological ordering, then G is a DAG.
- Suppose by contradiction that G has a topological ordering (u_1, u_2, … , u_n) but it also has a cycle C.
- Let u_j be the smallest element of C according to the topological ordering.
- Let u_i be its *predecessor* in the cycle (i.e., there is an edge $e=(u_i, u_j)$).

Topological Ordering implies DAG

- If graph G has a topological ordering, then G is a DAG.
- Suppose by contradiction that G has a topological ordering (u_1, u_2, … , u_n) but it also has a cycle C.
- Let u_j be the smallest element of C according to the topological ordering.
- Let u_i be its *predecessor* in the cycle (i.e., there is an edge e=(u_i, u_j)).
- u_i must appear before u_i in the topological order, by the presence of this edge.

Topological Ordering implies DAG

- If graph G has a topological ordering, then G is a DAG.
- Suppose by contradiction that G has a topological ordering (u_1, u_2, … , u_n) but it also has a cycle C.
- Let u_j be the smallest element of C according to the topological ordering.
- Let u_i be its *predecessor* in the cycle (i.e., there is an edge e=(u_i, u_j)).
- u_i must appear before u_i in the topological order, by the presence of this edge.
- This **contradicts** the fact that u_j was the smallest element of C according to the topological ordering.

Does DAG imply topological ordering?

Does DAG imply topological ordering?

- TO => DAG was proved via proof-by-contradiction.

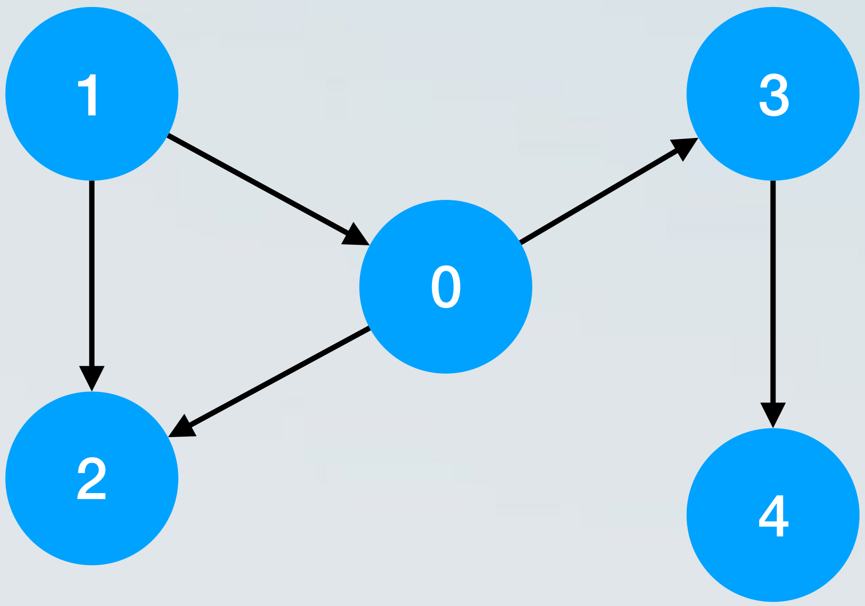
Does DAG imply topological ordering?

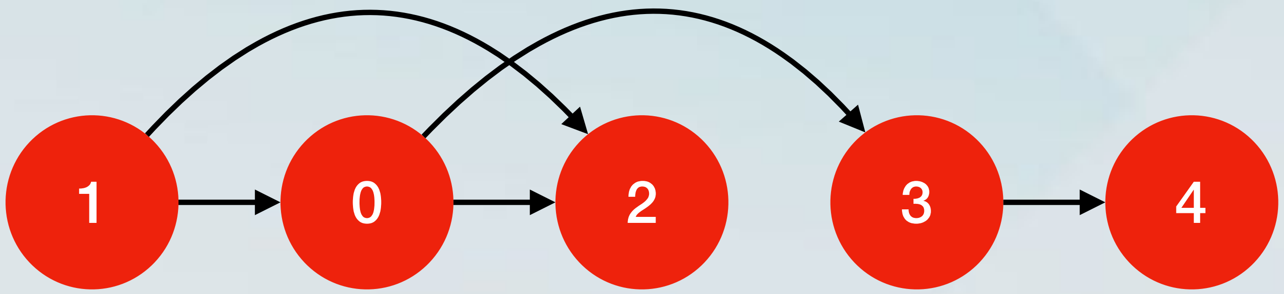
- TO => DAG was proved via proof-by-contradiction.
- DAG => TO will be proved via “proof-by-algorithm”.

Does DAG imply topological ordering?

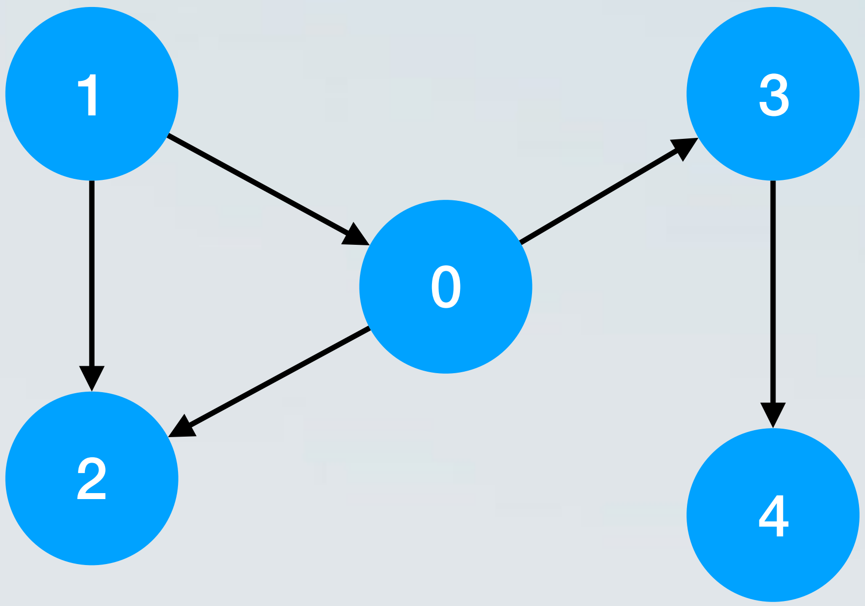
- TO => DAG was proved via proof-by-contradiction.
- DAG => TO will be proved via "proof-by-algorithm".
- We will design an *efficient* algorithm that, given a DAG G, finds a topological ordering of G.

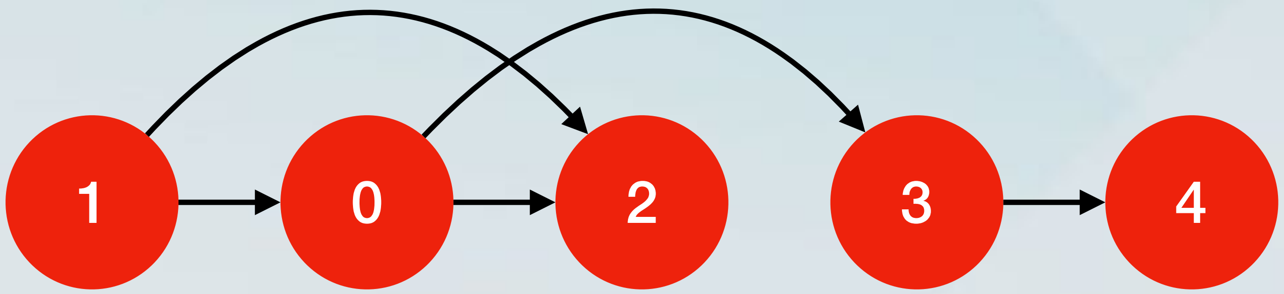
How do we start?





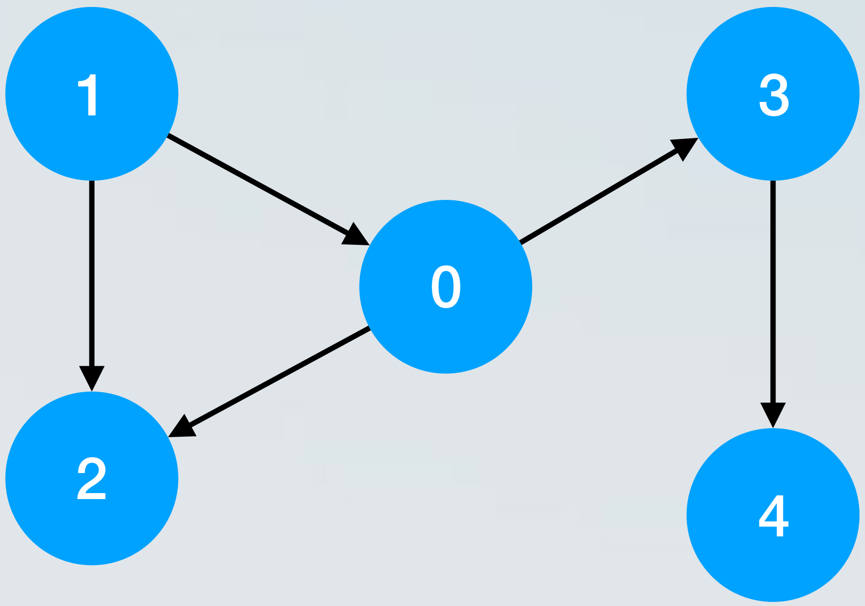
How do we start?

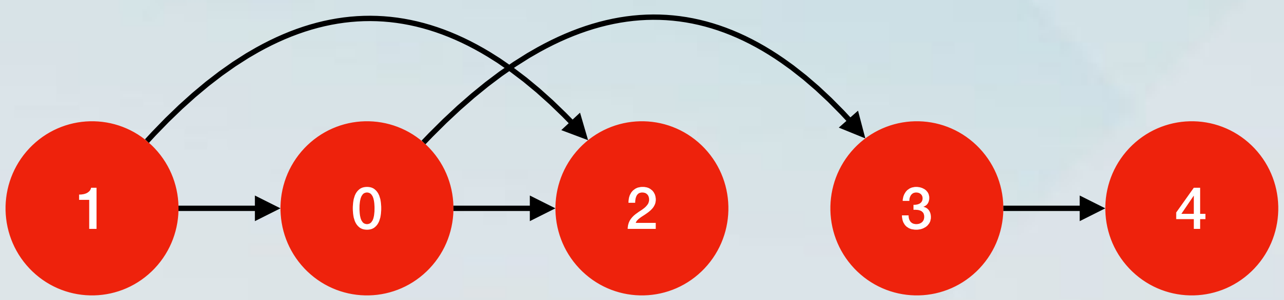




- Could we have started with anything other than node 1?

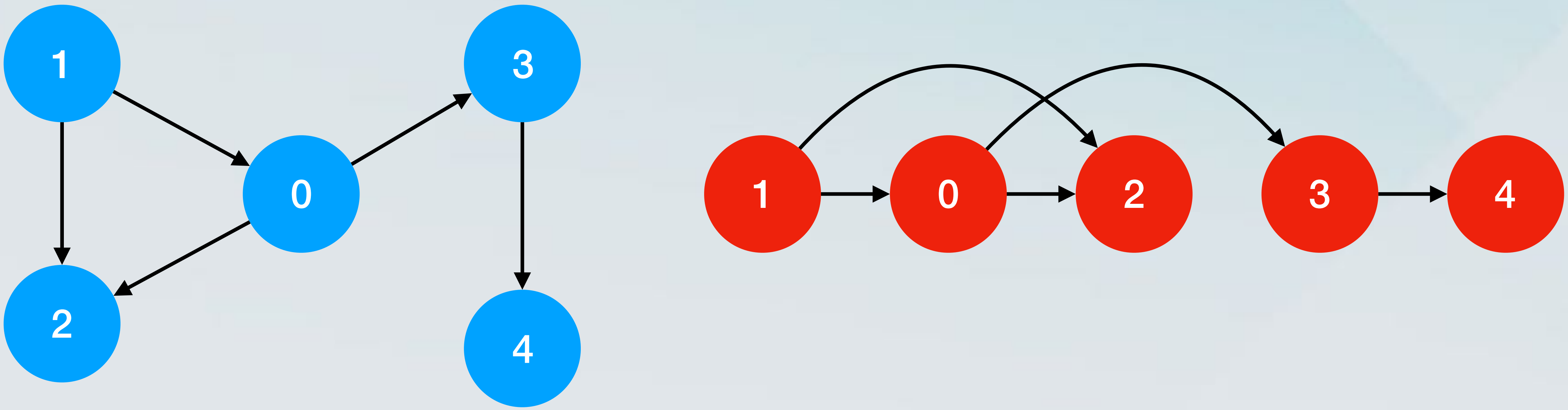
How do we start?





- Could we have started with anything other than node 1?
- The starting node must have no incoming edges!

How do we start?



- Could we have started with anything other than node 1?
- The starting node must have no incoming edges!
- Can we always find such a node?

Source node

Source node

- A source node is a node with no incoming edges.

Source node

- A source node is a node with no incoming edges.
- **Lemma:** Every DAG has at least one source node.

Source node

- A source node is a node with no incoming edges.
- **Lemma:** Every DAG has at least one source node.
- Proof by contradiction:

Source node

- A source node is a node with no incoming edges.
- **Lemma:** Every DAG has at least one source node.
- Proof by contradiction:
    - Assume that every node has at least one incoming edge.

Source node

- A source node is a node with no incoming edges.
- **Lemma:** Every DAG has at least one source node.
- Proof by contradiction:
  - Assume that every node has at least one incoming edge.
  - Start from any node u and follow edges from u *backwards*.

Source node

- A source node is a node with no incoming edges.
- **Lemma:** Every DAG has at least one source node.
- Proof by contradiction:
  - Assume that every node has at least one incoming edge.
  - Start from any node u and follow edges from u *backwards*.
    - Equivalently, we move to a neighbour of u in G^{rev}.

Source node

- A source node is a node with no incoming edges.
- **Lemma:** Every DAG has at least one source node.
- Proof by contradiction:
  - Assume that every node has at least one incoming edge.
  - Start from any node u and follow edges from u *backwards*.
    - Equivalently, we move to a neighbour of u in G^{rev}.
  - We can do that for every node, since by assumption there is no source.

Source node

- A source node is a node with no incoming edges.
- **Lemma:** Every DAG has at least one source node.
- Proof by contradiction:
  - Assume that every node has at least one incoming edge.
  - Start from any node u and follow edges from u *backwards*.
    - Equivalently, we move to a neighbour of u in G^{rev}.
  - We can do that for every node, since by assumption there is no source.
  - After at least n+1 steps, we will have visited the same node twice.

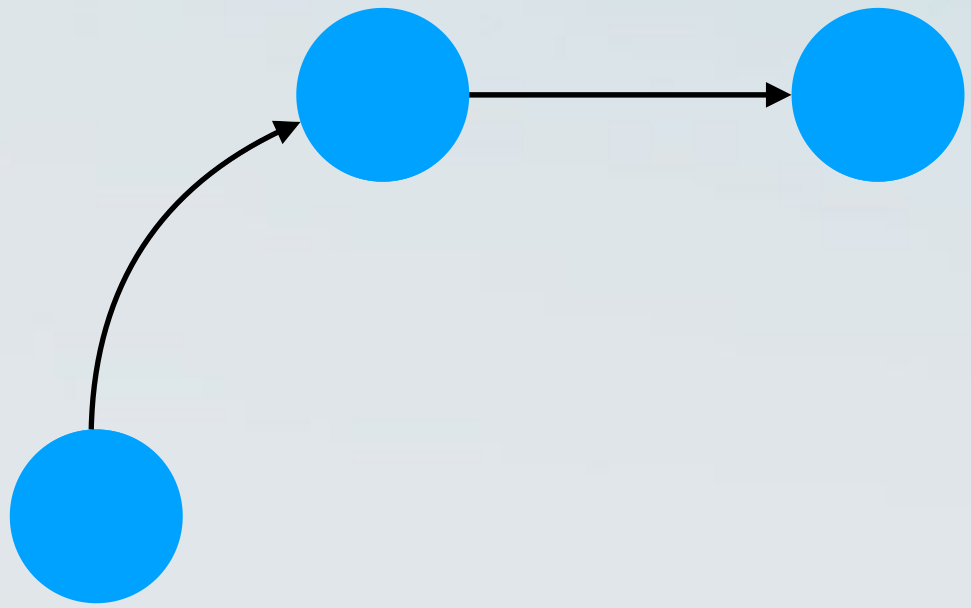
Source node

- A source node is a node with no incoming edges.
- **Lemma:** Every DAG has at least one source node.
- Proof by contradiction:
  - Assume that every node has at least one incoming edge.
  - Start from any node u and follow edges from u *backwards*.
    - Equivalently, we move to a neighbour of u in G^{rev}.
  - We can do that for every node, since by assumption there is no source.
  - After at least n+1 steps, we will have visited the same node twice.
  - The graph has a cycle, therefore it can't be a DAG. **Contradiction!**

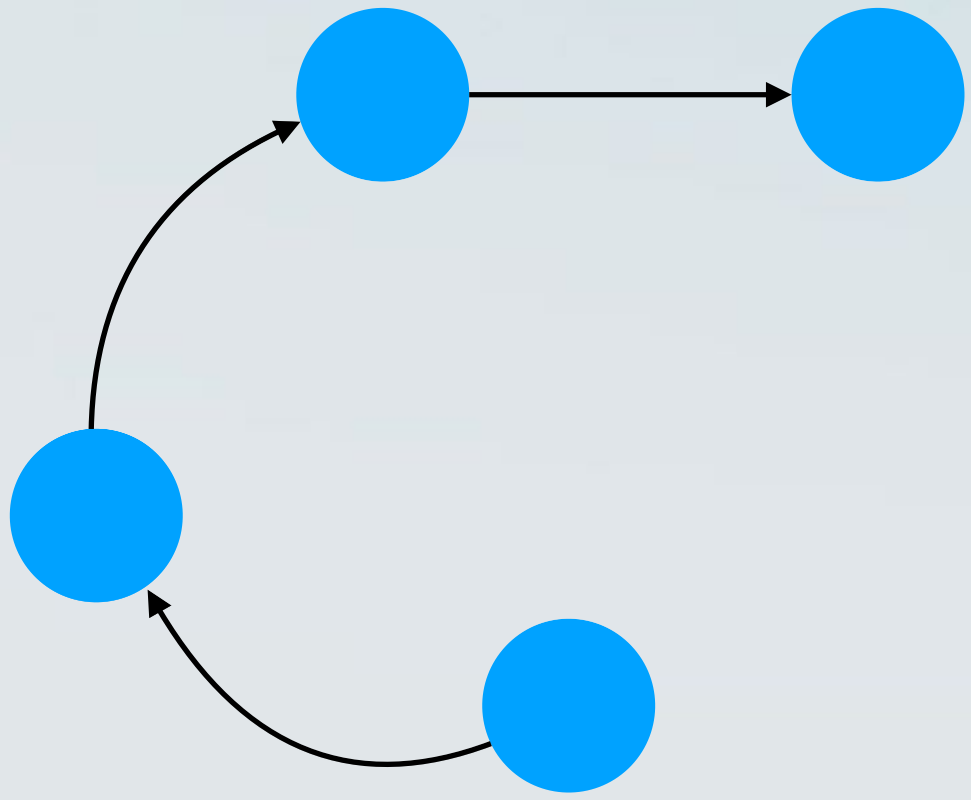
Pictorially

Pictorially

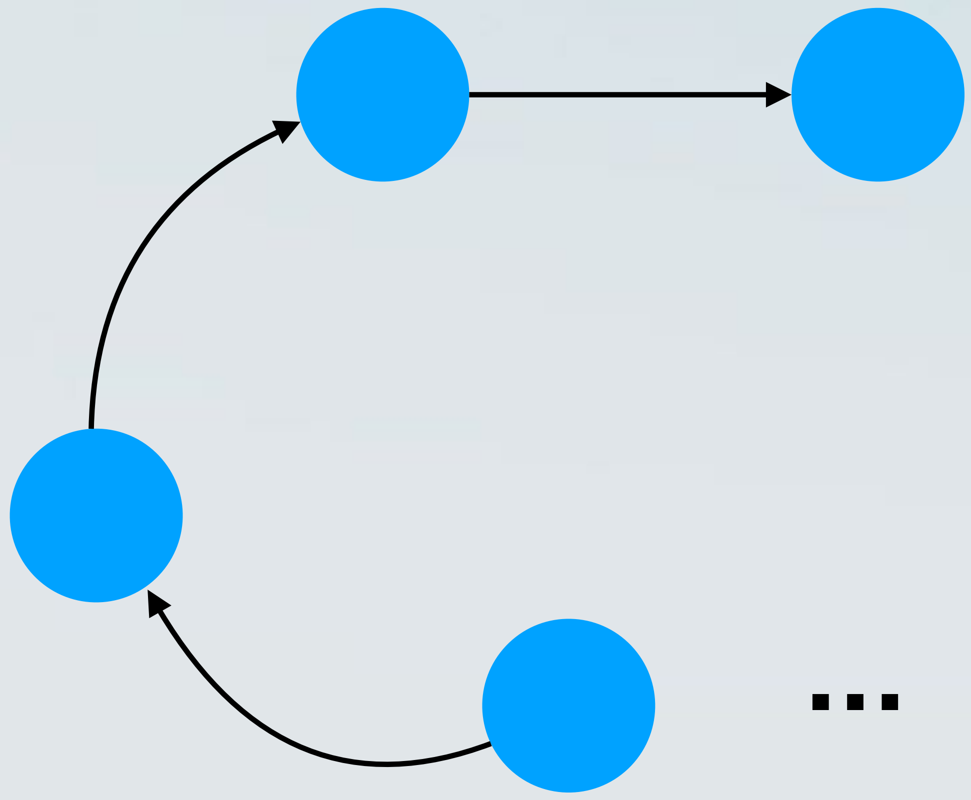
Pictorially



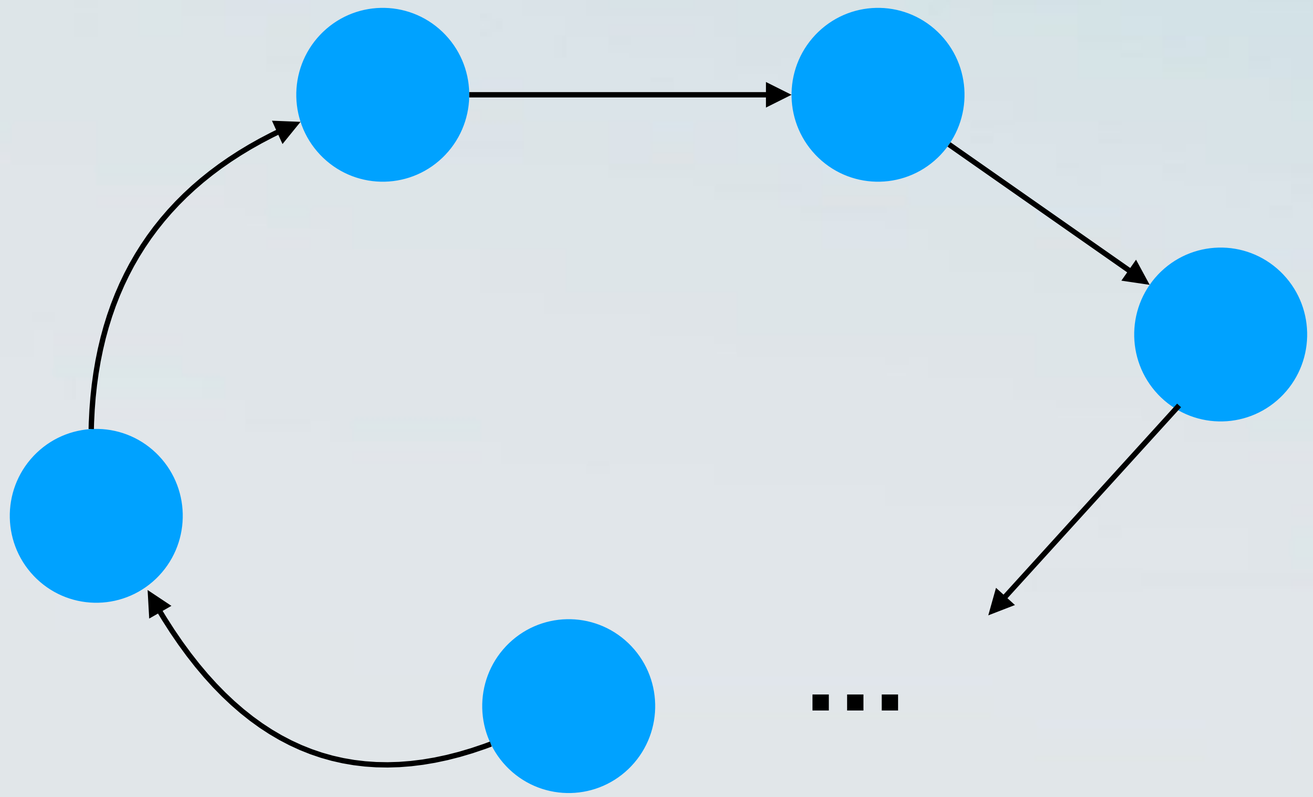
Pictorially



Pictorially



Pictorially



Another simple fact

Another simple fact

- If we remove a node u and all its incident edges from a DAG G, the resulting graph G’ is still a DAG.

Another simple fact

- If we remove a node u and all its incident edges from a DAG G, the resulting graph G’ is still a DAG.
  - If G’ had a cycle, the same cycle would be present in G.

DAG implies topological ordering

DAG implies topological ordering

- Proof-by-induction:

DAG implies topological ordering

- Proof-by-induction:
  - Base Case: If the DAG has one or two nodes, it clearly has a topological ordering.

DAG implies topological ordering

- Proof-by-induction:
  - Base Case: If the DAG has one or two nodes, it clearly has a topological ordering.
  - Inductive step: Assume that a DAG with up to k nodes has a topological ordering (Inductive Hypothesis). We will prove that a DAG with k+1 nodes has a topological ordering.

DAG implies topological ordering

- Proof-by-induction:
  - Base Case: If the DAG has one or two nodes, it clearly has a topological ordering.
  - Inductive step: Assume that a DAG with up to k nodes has a topological ordering (Inductive Hypothesis). We will prove that a DAG with k+1 nodes has a topological ordering.
    - By our lemma, there is at least one source node in G, and let u be such a node.

DAG implies topological ordering

- Proof-by-induction:
  - Base Case: If the DAG has one or two nodes, it clearly has a topological ordering.
  - Inductive step: Assume that a DAG with up to k nodes has a topological ordering (Inductive Hypothesis). We will prove that a DAG with k+1 nodes has a topological ordering.
    - By our lemma, there is at least one source node in G, and let u be such a node.
    - Put u first in the topological ordering (safe, since u is a source).

DAG implies topological ordering

- Proof-by-induction:
  - Base Case: If the DAG has one or two nodes, it clearly has a topological ordering.
  - Inductive step: Assume that a DAG with up to k nodes has a topological ordering (Inductive Hypothesis). We will prove that a DAG with k+1 nodes has a topological ordering.
    - By our lemma, there is at least one source node in G, and let u be such a node.
    - Put u first in the topological ordering (safe, since u is a source).
    - Consider the graph G’, obtained by G if we remove u and its incident edges.

DAG implies topological ordering

- Proof-by-induction:
  - Base Case: If the DAG has one or two nodes, it clearly has a topological ordering.
  - Inductive step: Assume that a DAG with up to k nodes has a topological ordering (Inductive Hypothesis). We will prove that a DAG with k+1 nodes has a topological ordering.
    - By our lemma, there is at least one source node in G, and let u be such a node.
    - Put u first in the topological ordering (safe, since u is a source).
    - Consider the graph G', obtained by G if we remove u and its incident edges.
    - G' is a DAG (by the simple fact) with k nodes.

DAG implies topological ordering

- Proof-by-induction:
  - Base Case: If the DAG has one or two nodes, it clearly has a topological ordering.
  - Inductive step: Assume that a DAG with up to k nodes has a topological ordering (Inductive Hypothesis). We will prove that a DAG with k+1 nodes has a topological ordering.
    - By our lemma, there is at least one source node in G, and let u be such a node.
    - Put u first in the topological ordering (safe, since u is a source).
    - Consider the graph G', obtained by G if we remove u and its incident edges.
    - G' is a DAG (by the simple fact) with k nodes.
      - It has a topological ordering by the induction hypothesis.

DAG implies topological ordering

- Proof-by-induction:
  - Base Case: If the DAG has one or two nodes, it clearly has a topological ordering.
  - Inductive step: Assume that a DAG with up to k nodes has a topological ordering (Inductive Hypothesis). We will prove that a DAG with k+1 nodes has a topological ordering.
    - By our lemma, there is at least one source node in G, and let u be such a node.
    - Put u first in the topological ordering (safe, since u is a source).
    - Consider the graph G', obtained by G if we remove u and its incident edges.
    - G' is a DAG (by the simple fact) with k nodes.
      - It has a topological ordering by the induction hypothesis.
    - Append this ordering to u.

Where is the “proof-by-algorithm”?

Where is the “proof-by-algorithm”?

- We can turn that induction proof into an algorithm.

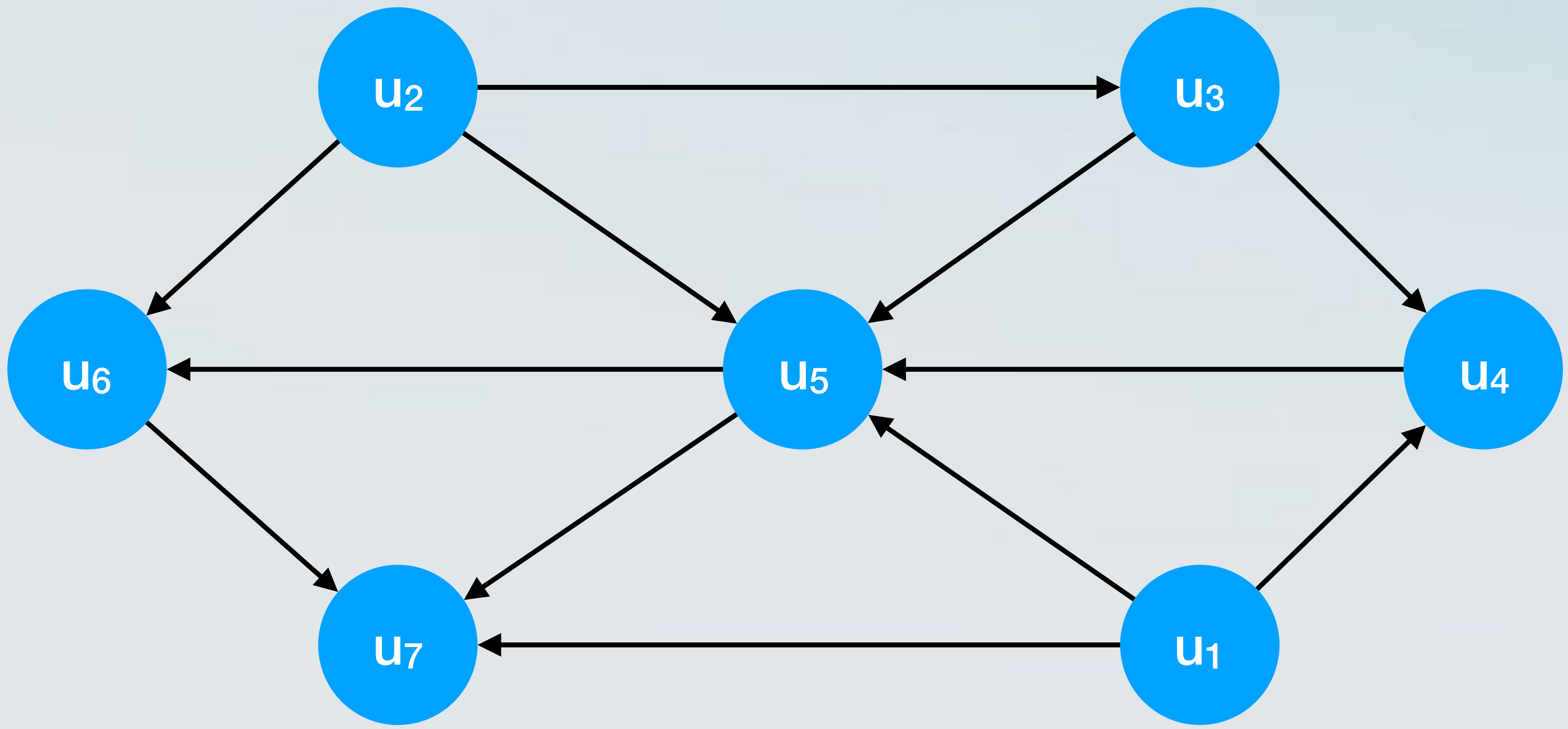
Where is the "proof-by-algorithm"?

- We can turn that induction proof into an algorithm.

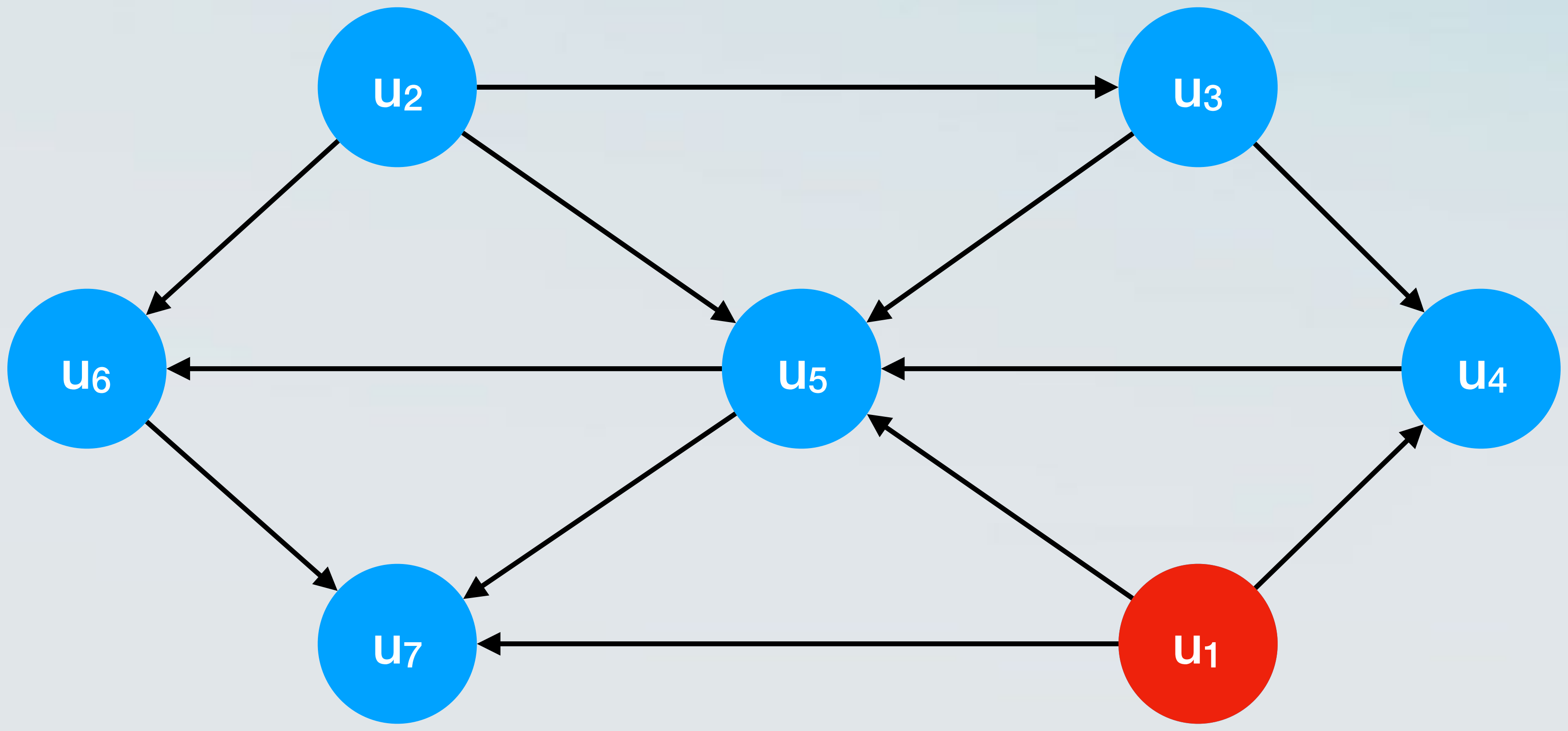
Algorithm TopologicalSort(G)

    Find a source vertex u and put it first in the order.
    Let G'=G-{u}
    TopologicalSort(G')
    Append this order after u

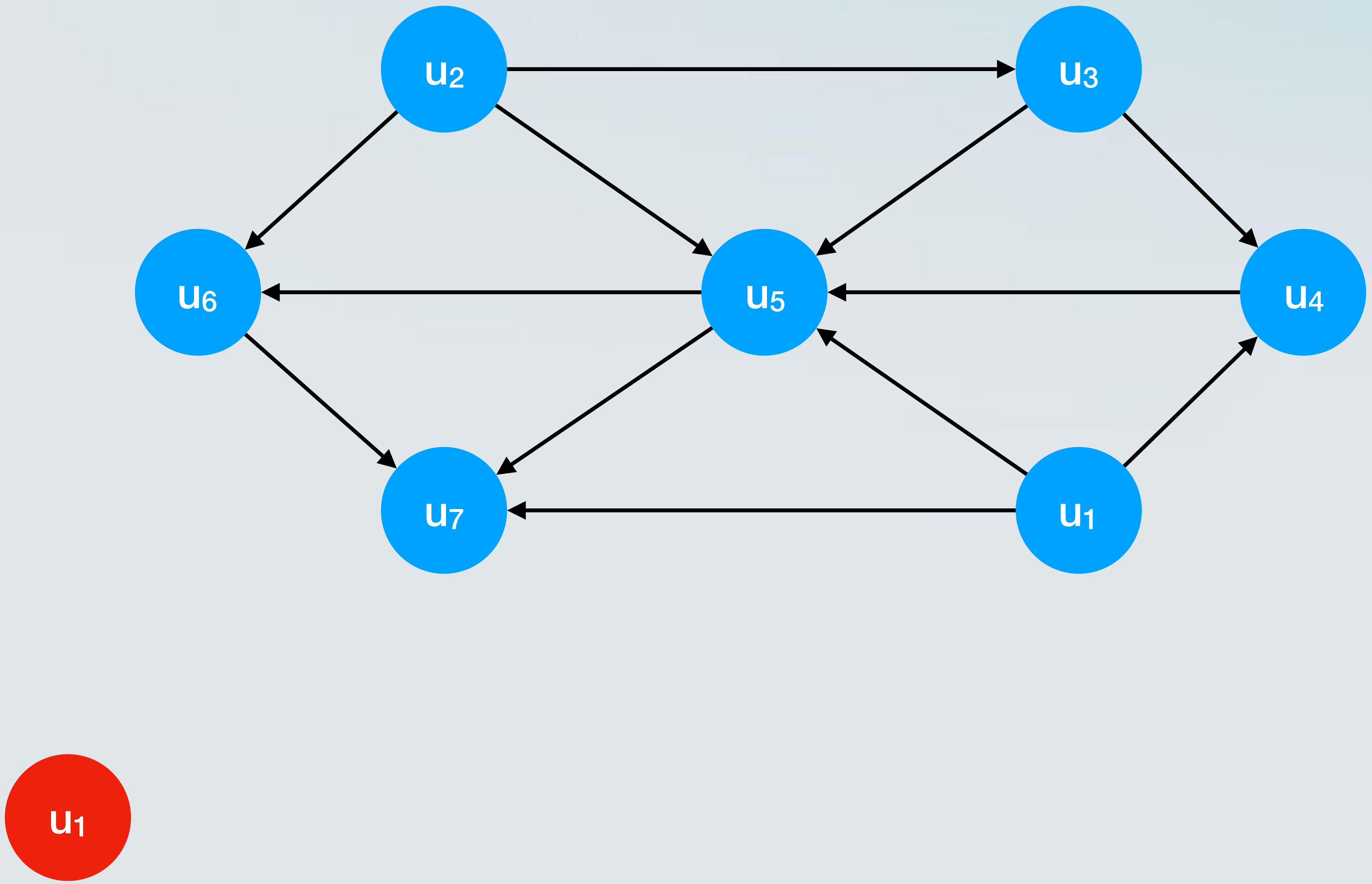
Example



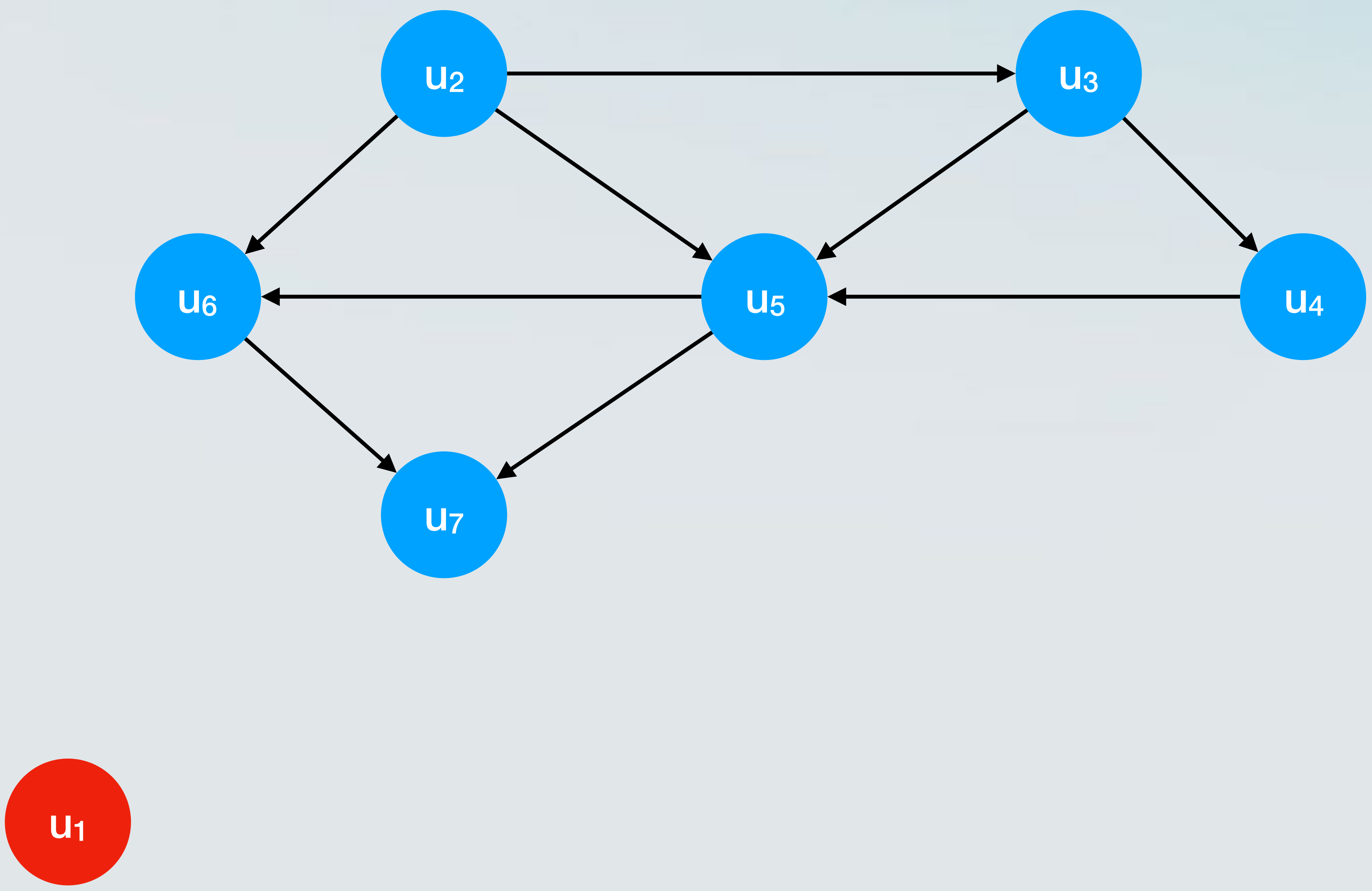
Example



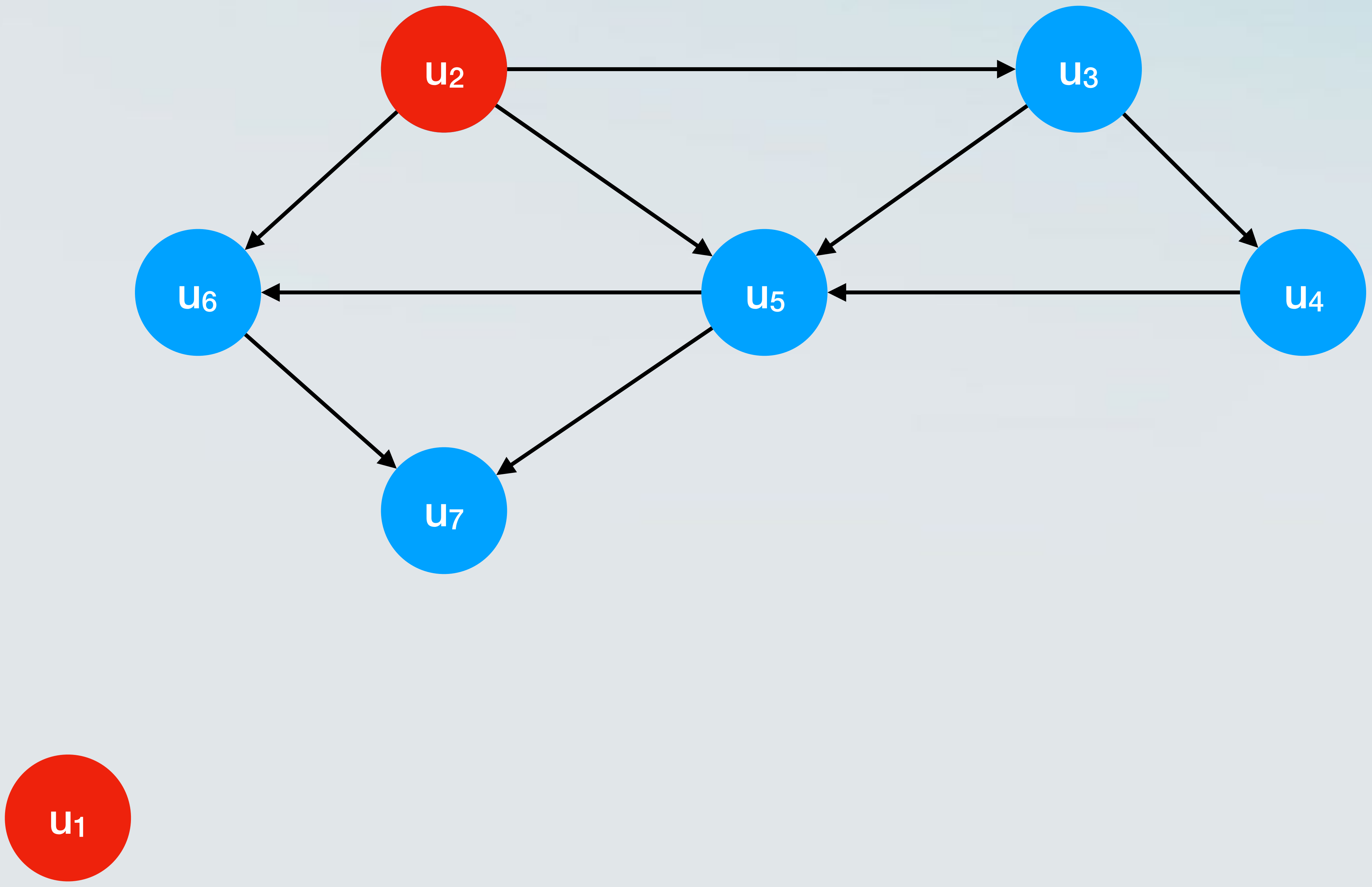
Example



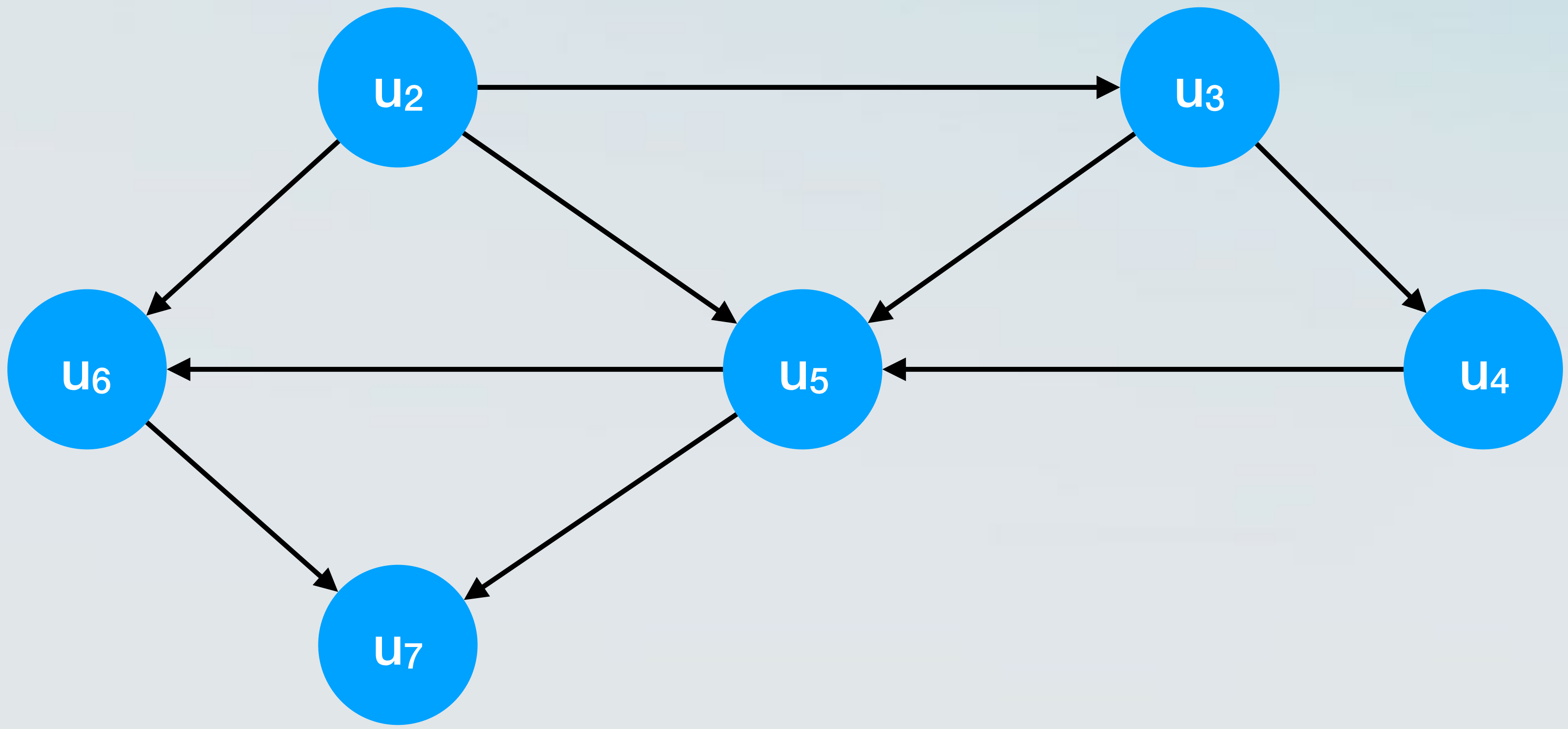
Example

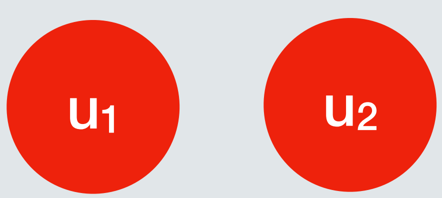


Example

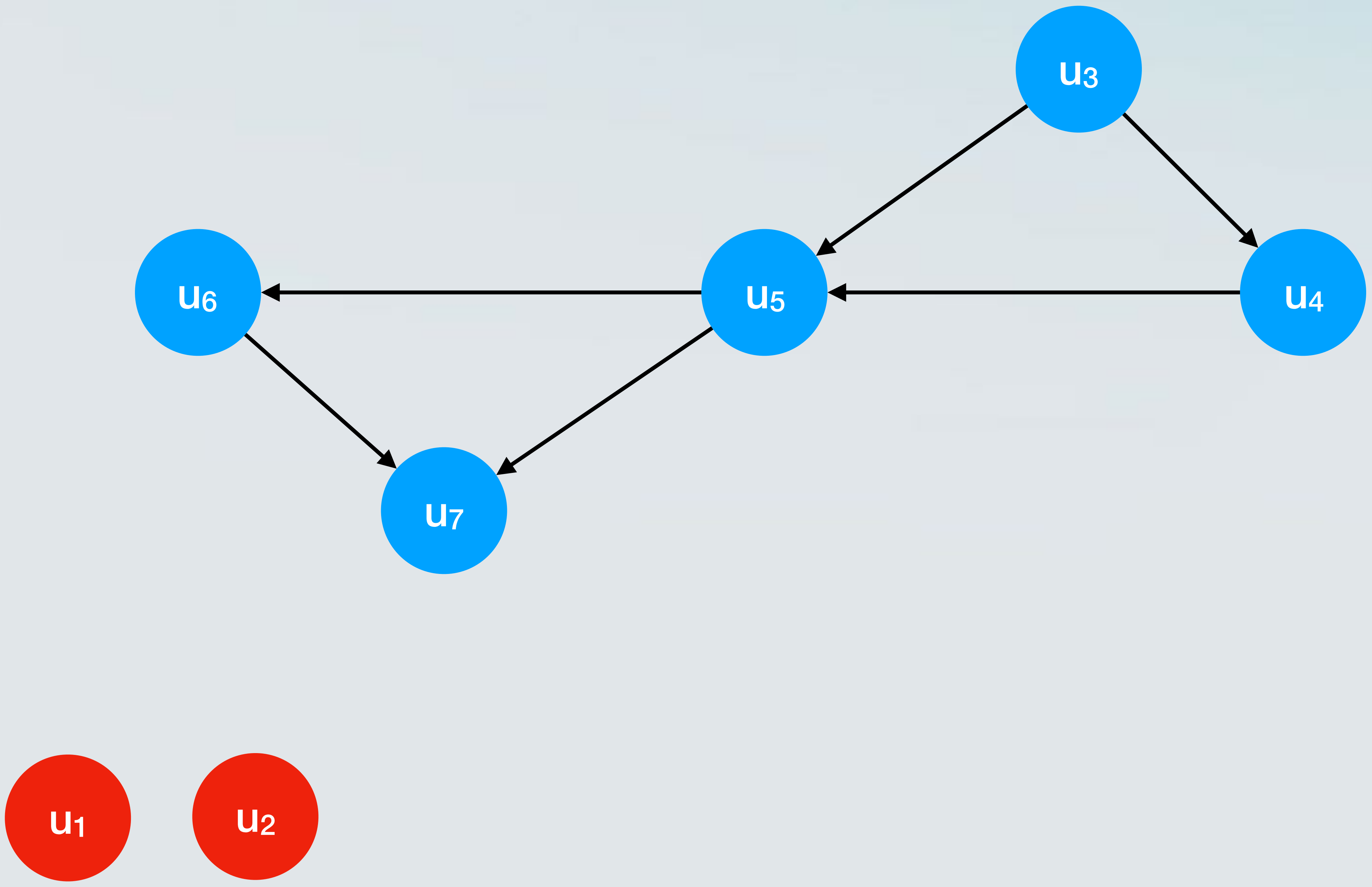


Example

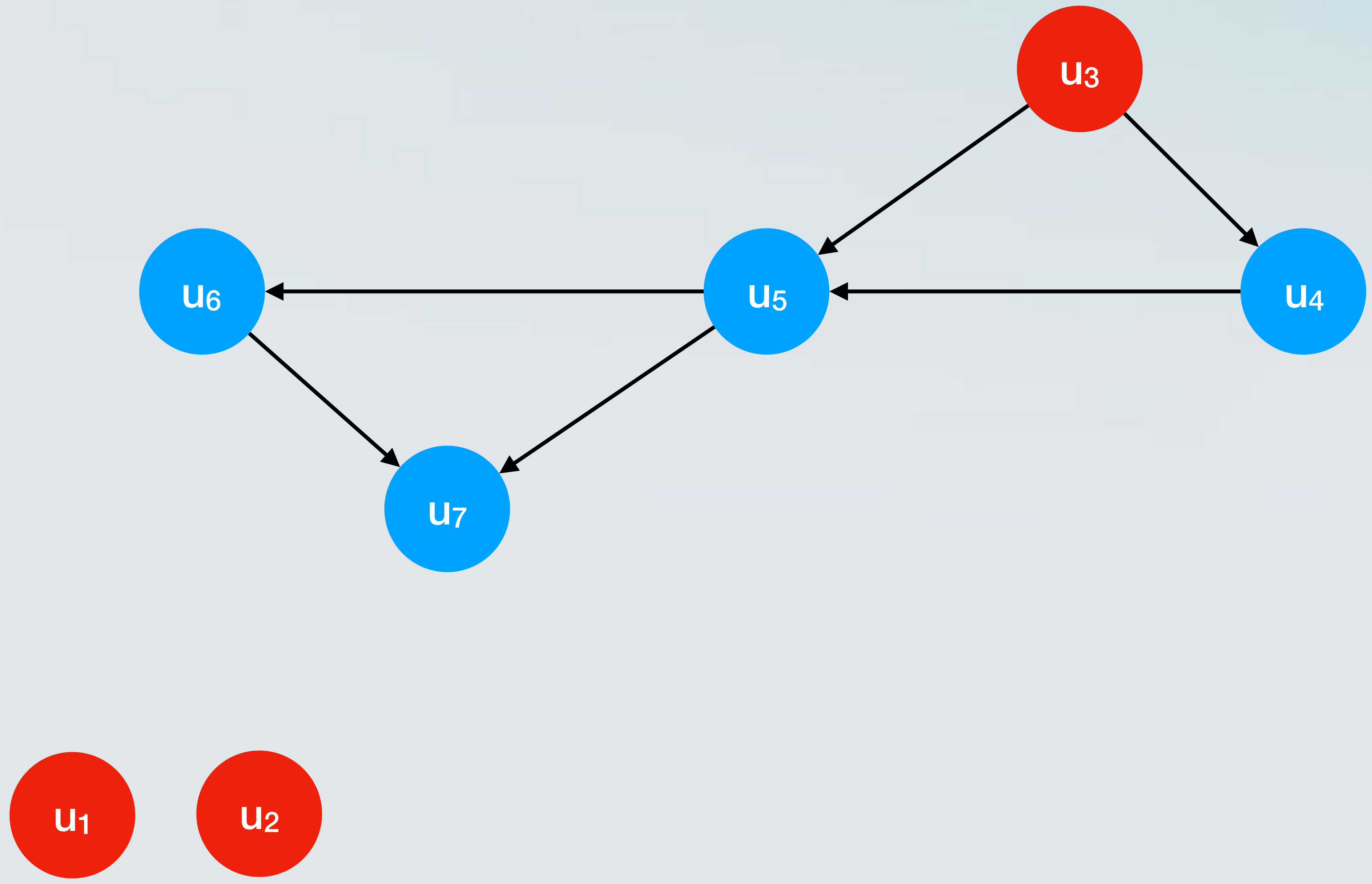




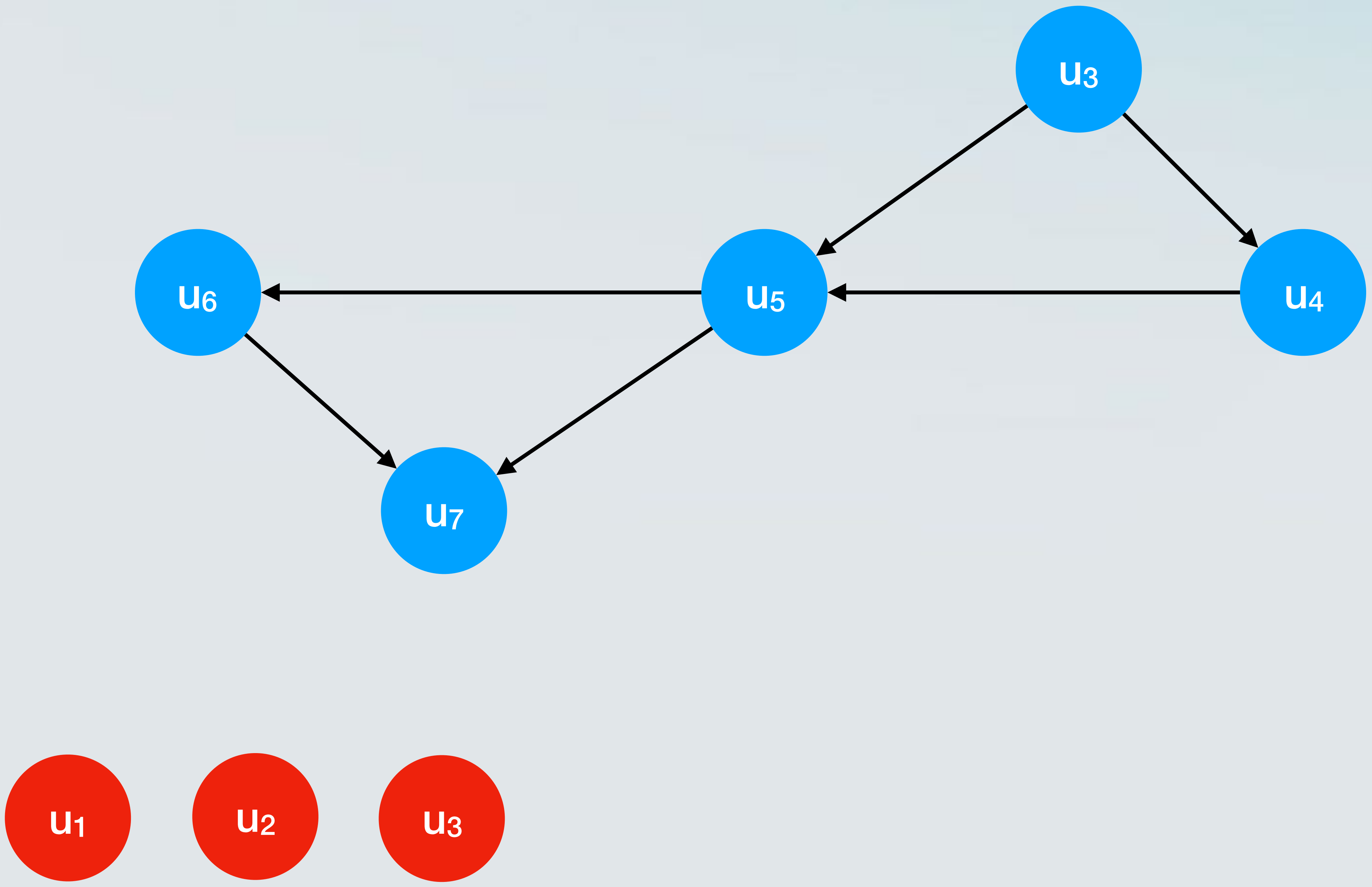
Example



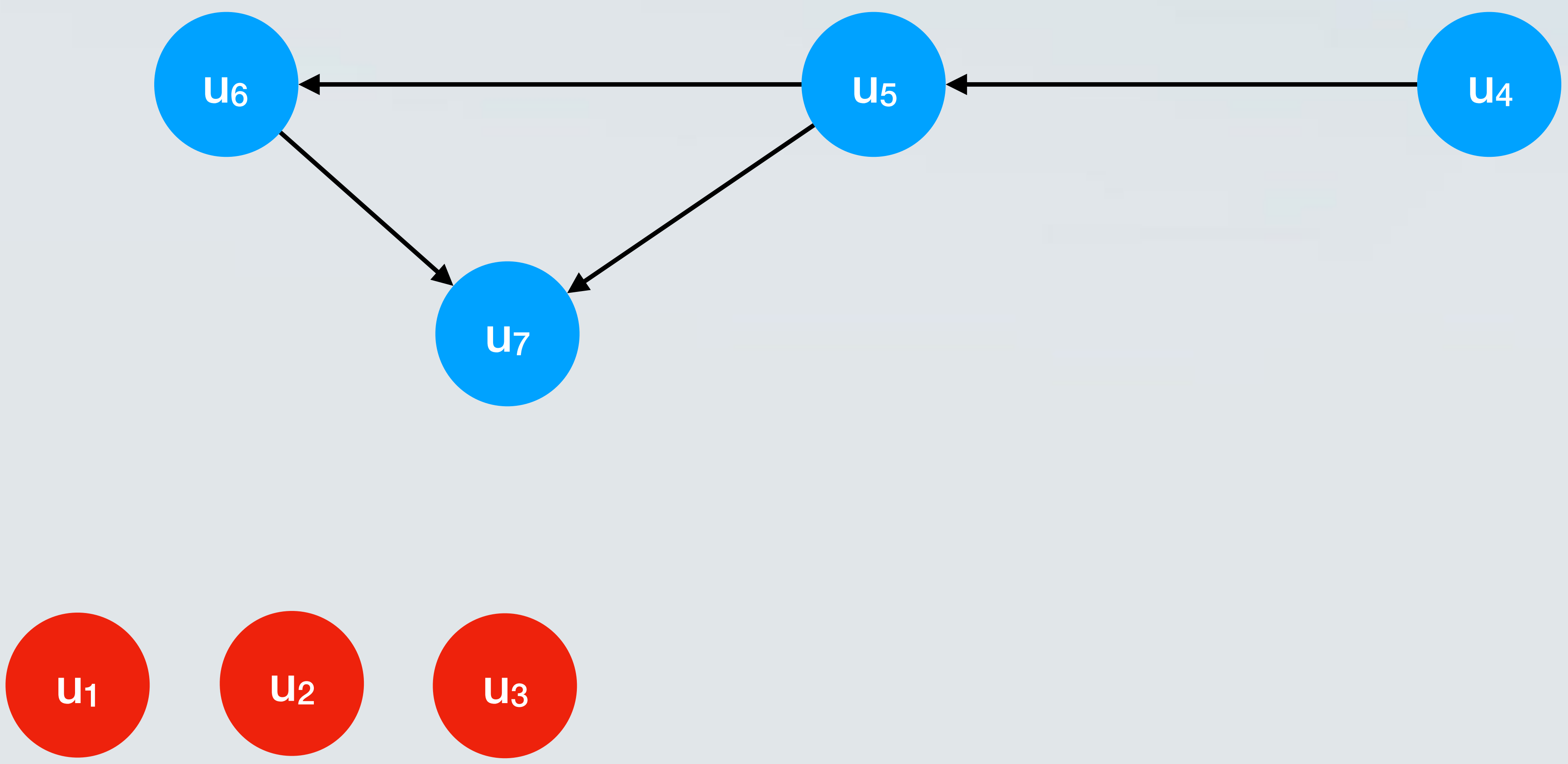
Example



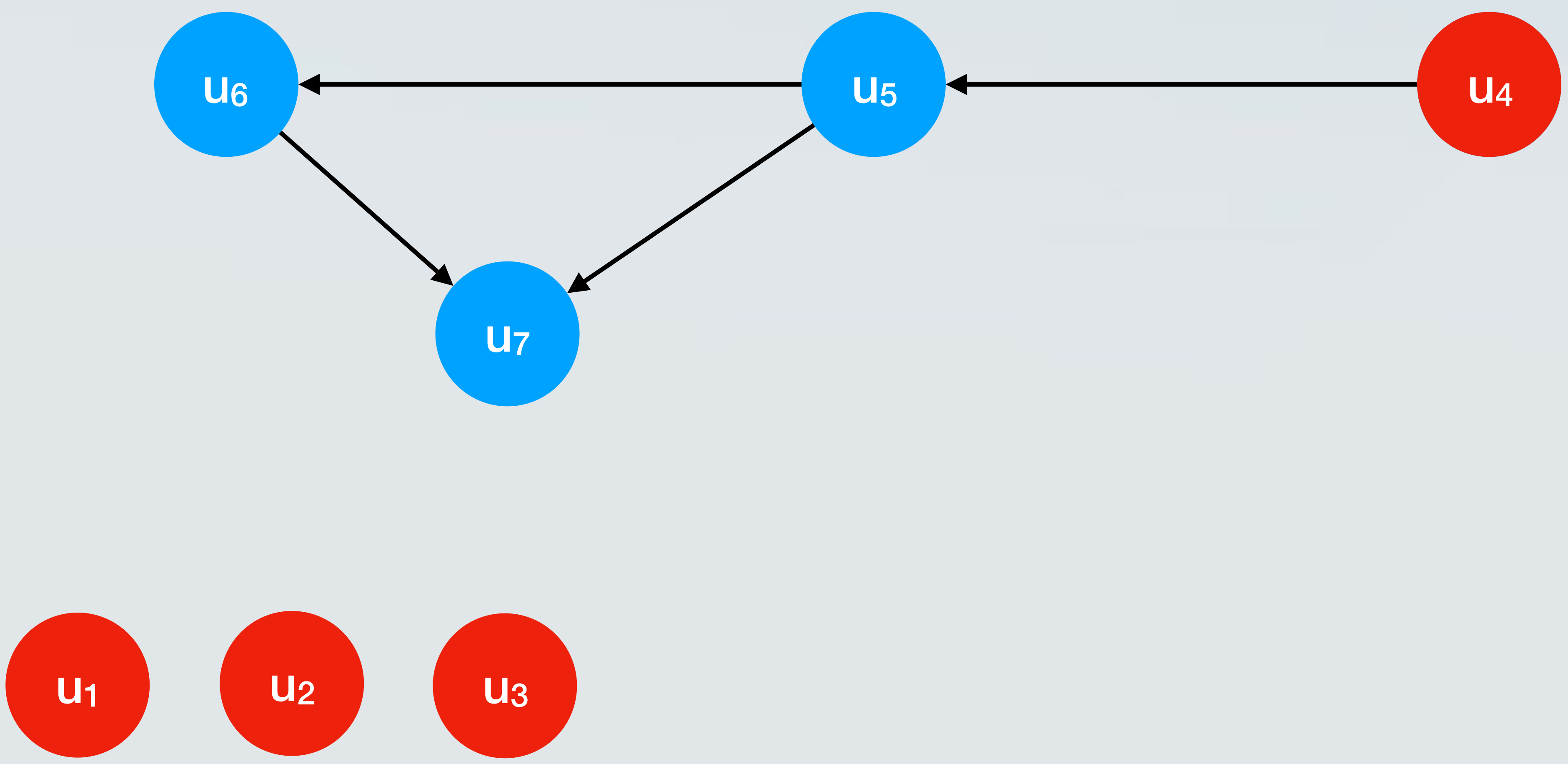
Example



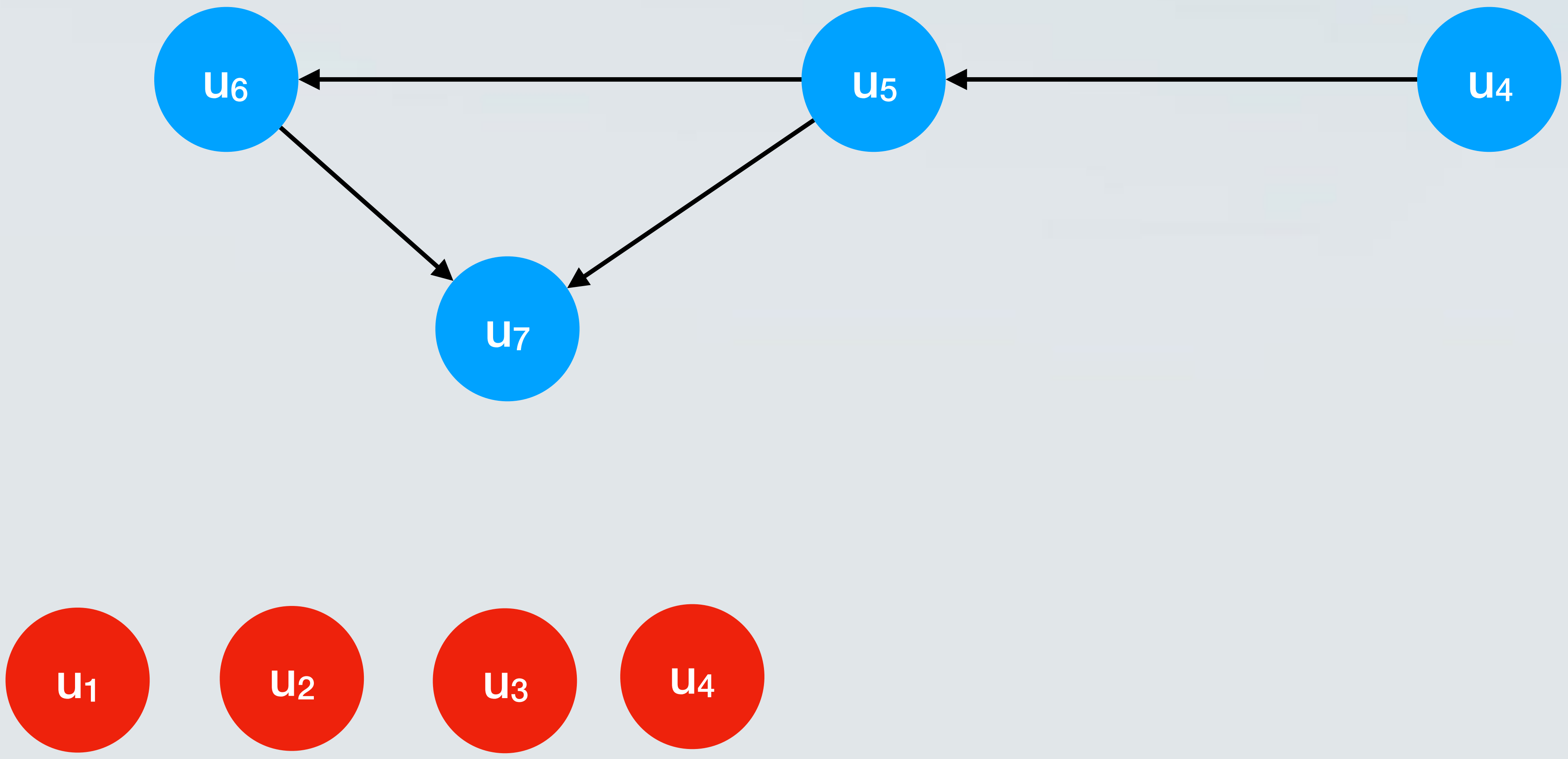
Example



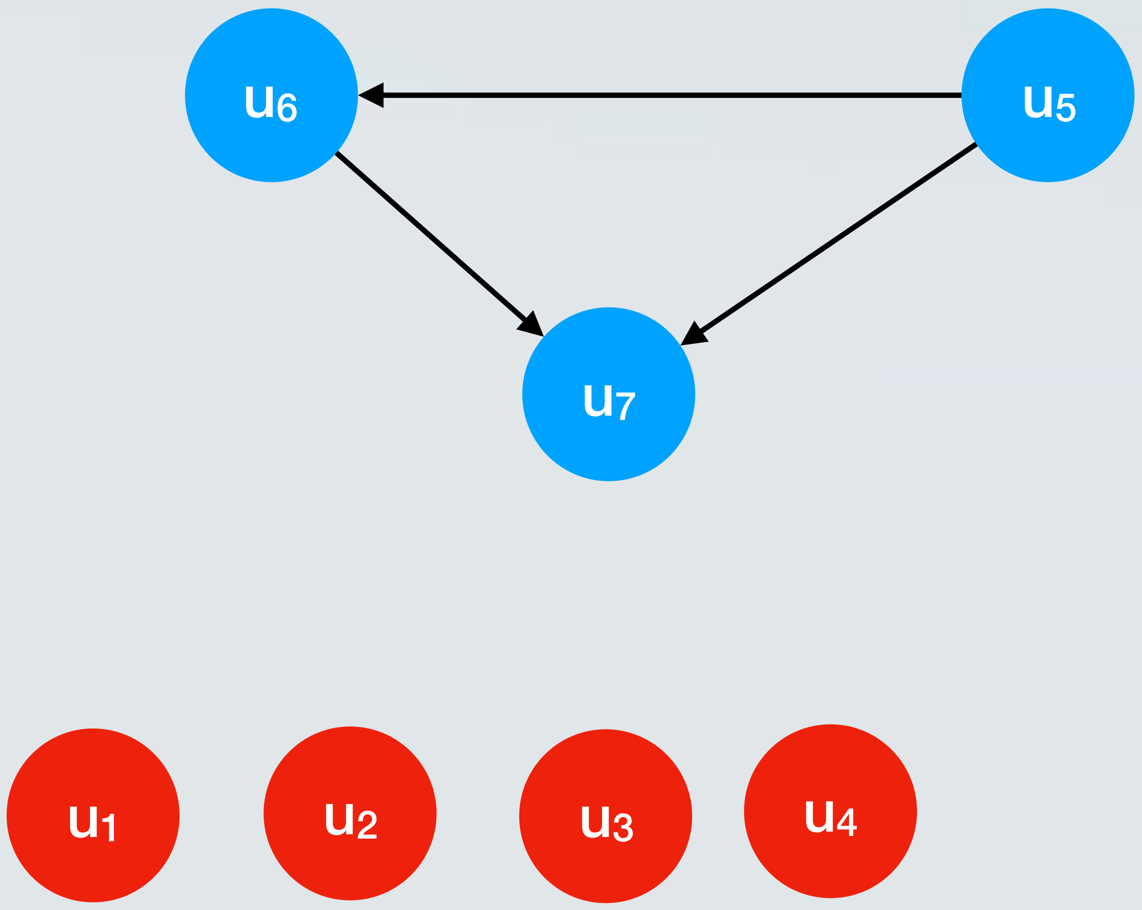
Example



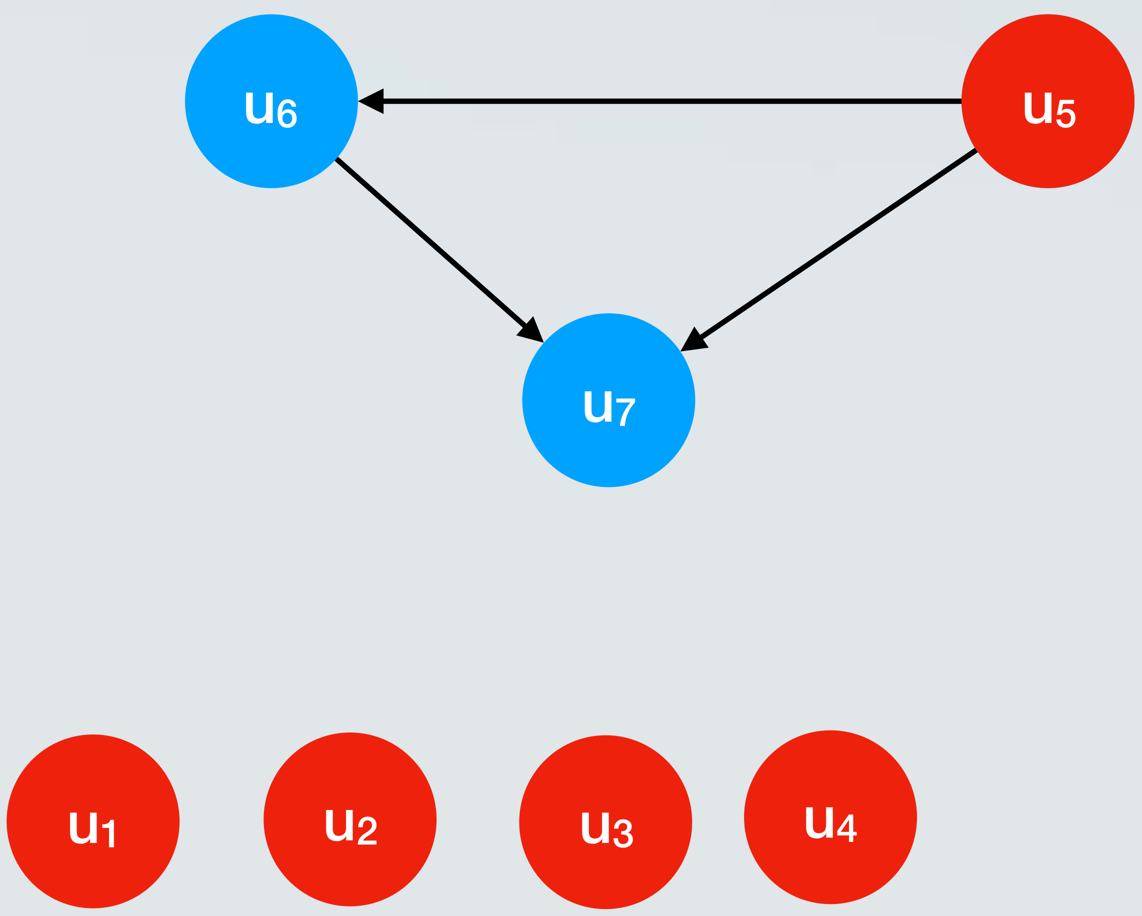
Example



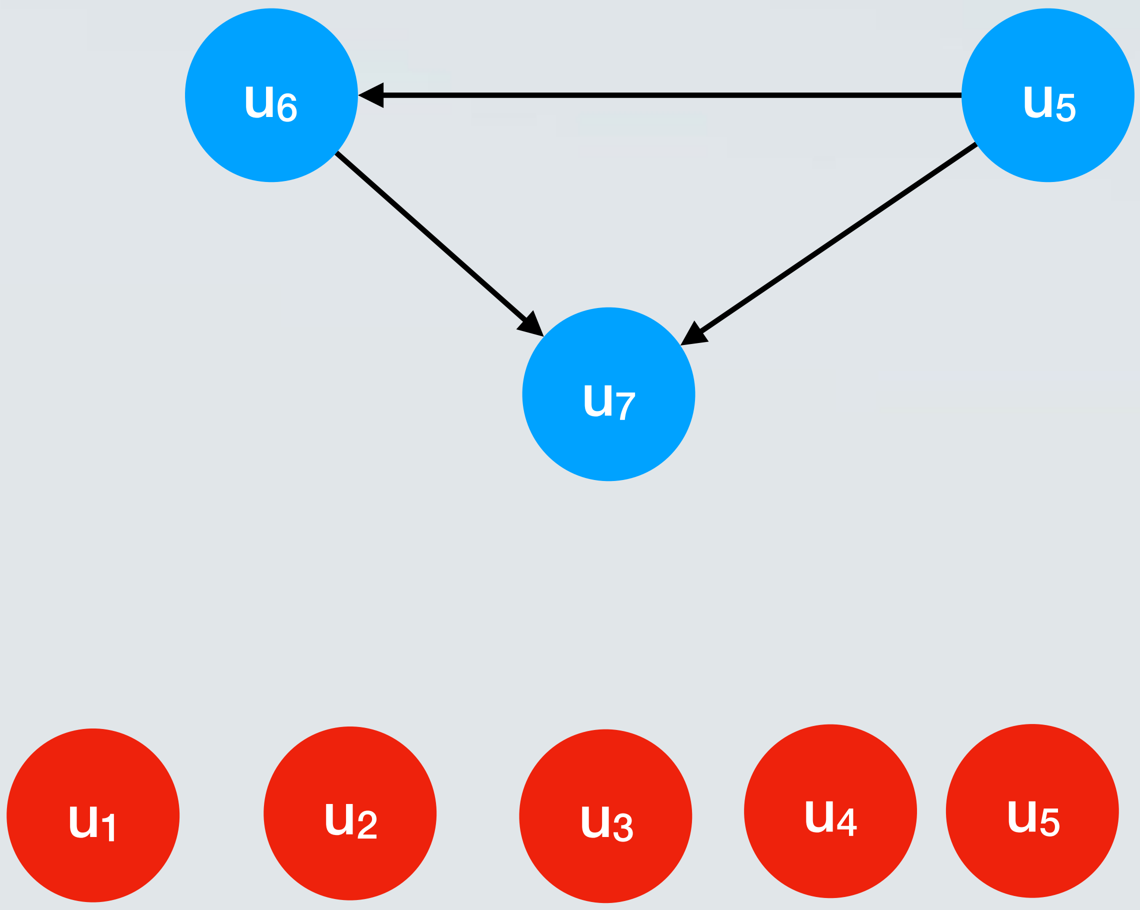
Example



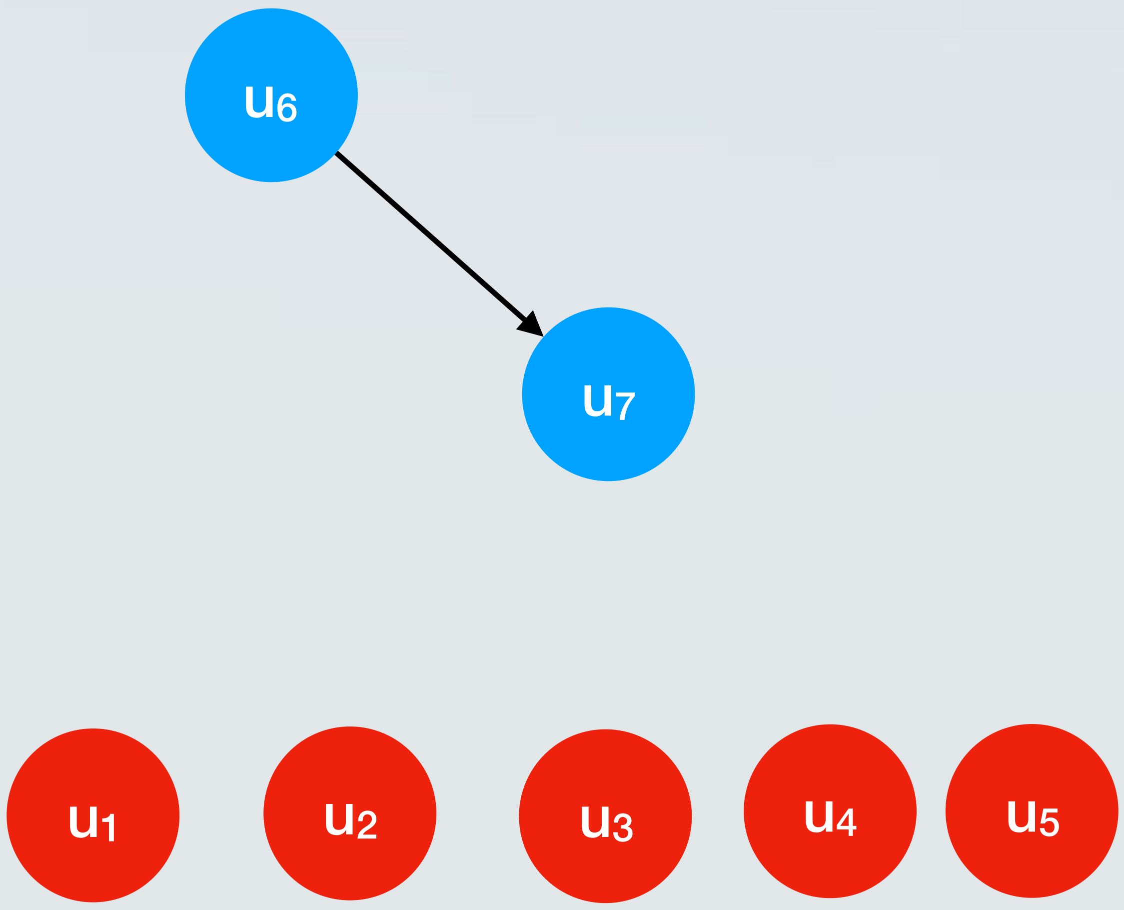
Example



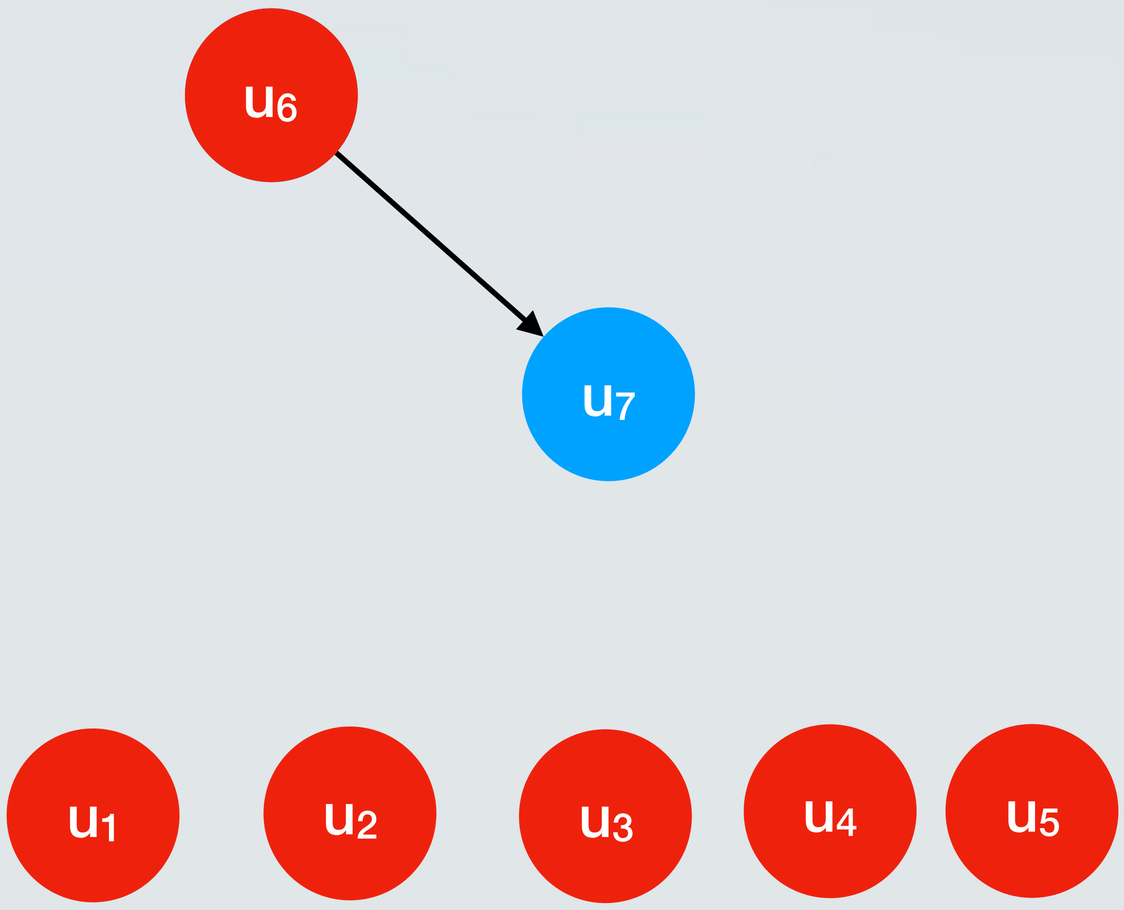
Example



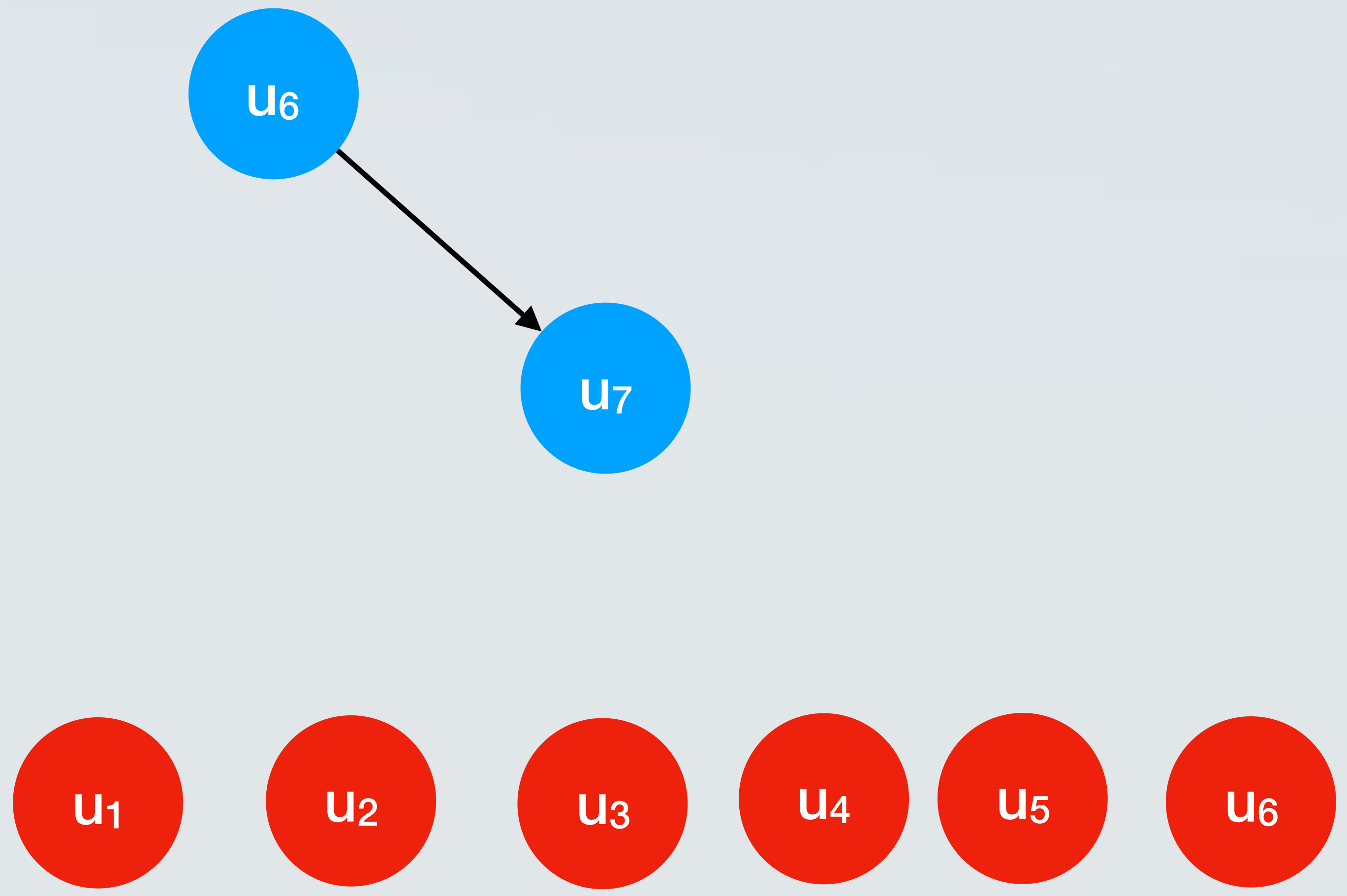
Example



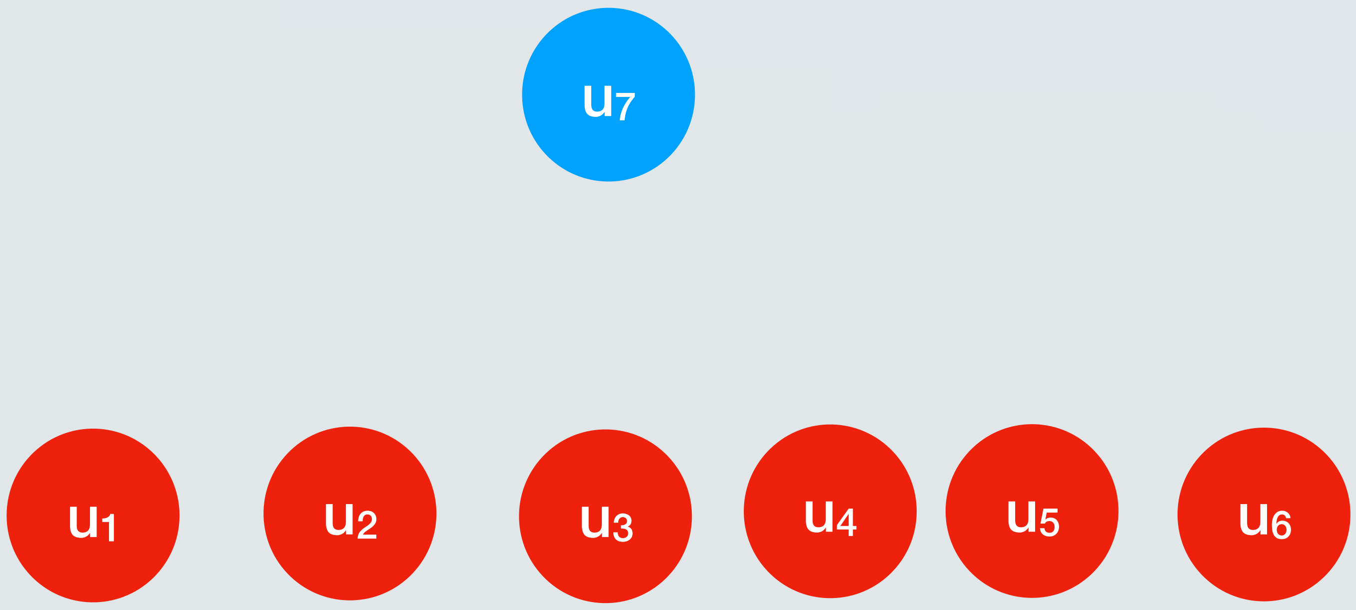
Example



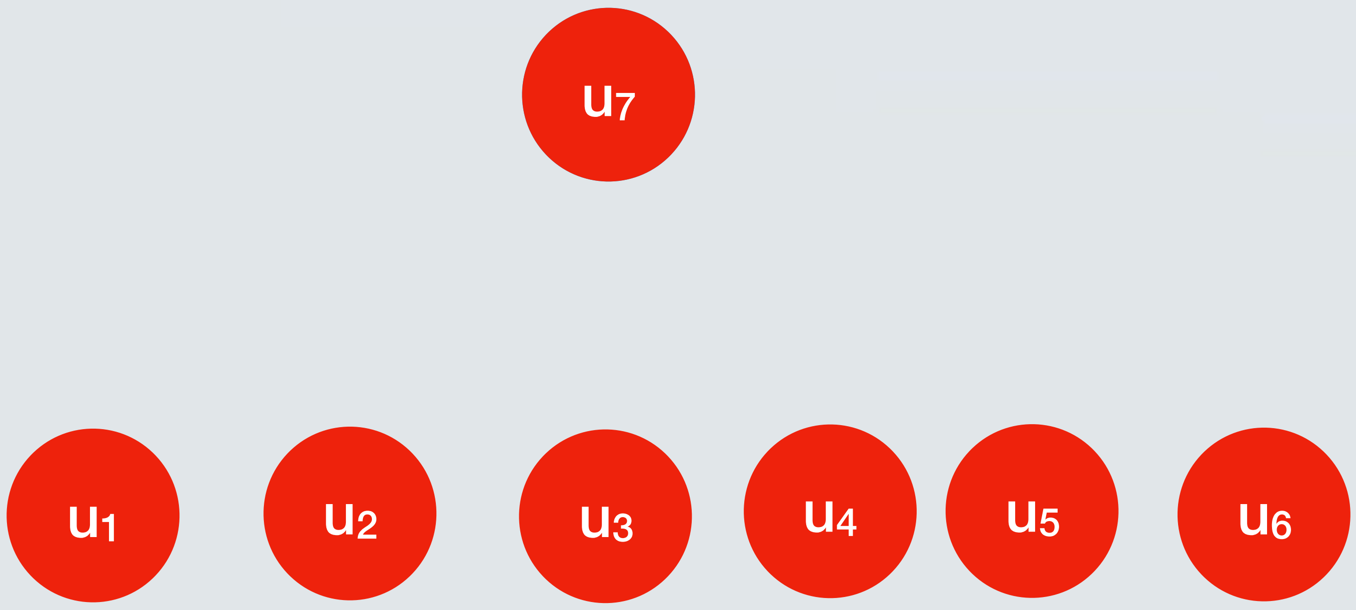
Example



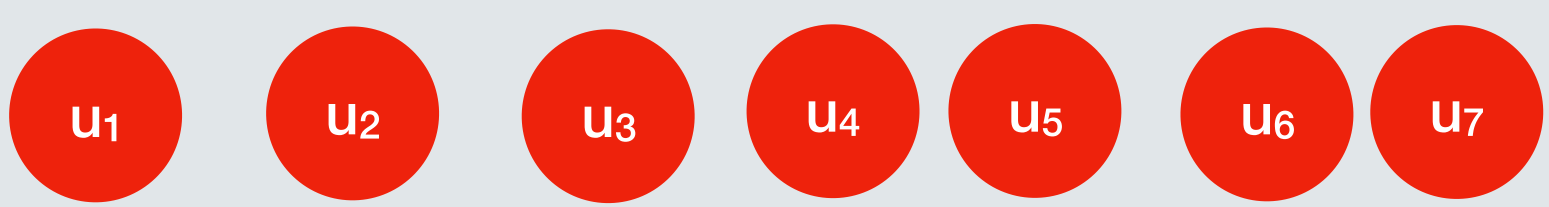
Example



Example



Example



Running time

Running time

- We need to find a source u.

Running time

- We need to find a source u.
- We could check each node of the graph.

Running time

- We need to find a source u.
- We could check each node of the graph.
- We check n nodes in the first iteration, $n-1$ nodes in the second, and so on…

Running time

- We need to find a source u.
- We could check each node of the graph.
- We check n nodes in the first iteration, $n-1$ nodes in the second, and so on…
- What is the running time of this?

Running time

- We need to find a source u.
- We could check each node of the graph.
- We check *n* nodes in the first iteration, *n-1* nodes in the second, and so on…
- What is the running time of this?
    - **$O(n^2)$**

Running time

- We need to find a source u.
- We could check each node of the graph.
- We check *n* nodes in the first iteration, *n-1* nodes in the second, and so on…
- What is the running time of this?
    - **$O(n^2)$**
- Can we do better?

A faster algorithm

- We will be more efficient in the choice of sources.
- We will say that a node is active, if it has not been selected (and therefore removed) as a source by the algorithm.
- We maintain two things:
  - (a) For each node w, the number of *incoming edges* from active nodes.
  - (b) The set S of all active nodes that have *no incoming edges* from other active nodes.

A faster algorithm

- In the beginning, all nodes are active and we can initialise (a) and (b) via a pass through the graph (time **O(m+n)**)
- In each iteration:
  - We select a node u from the set S.
  - We delete u.
  - We go through all the neighbours w of u and we reduce their value in (a) (i.e., number of incoming edges from active nodes) by 1.
  - When the value of (a) for some node w goes to 0, w is added to the set S.

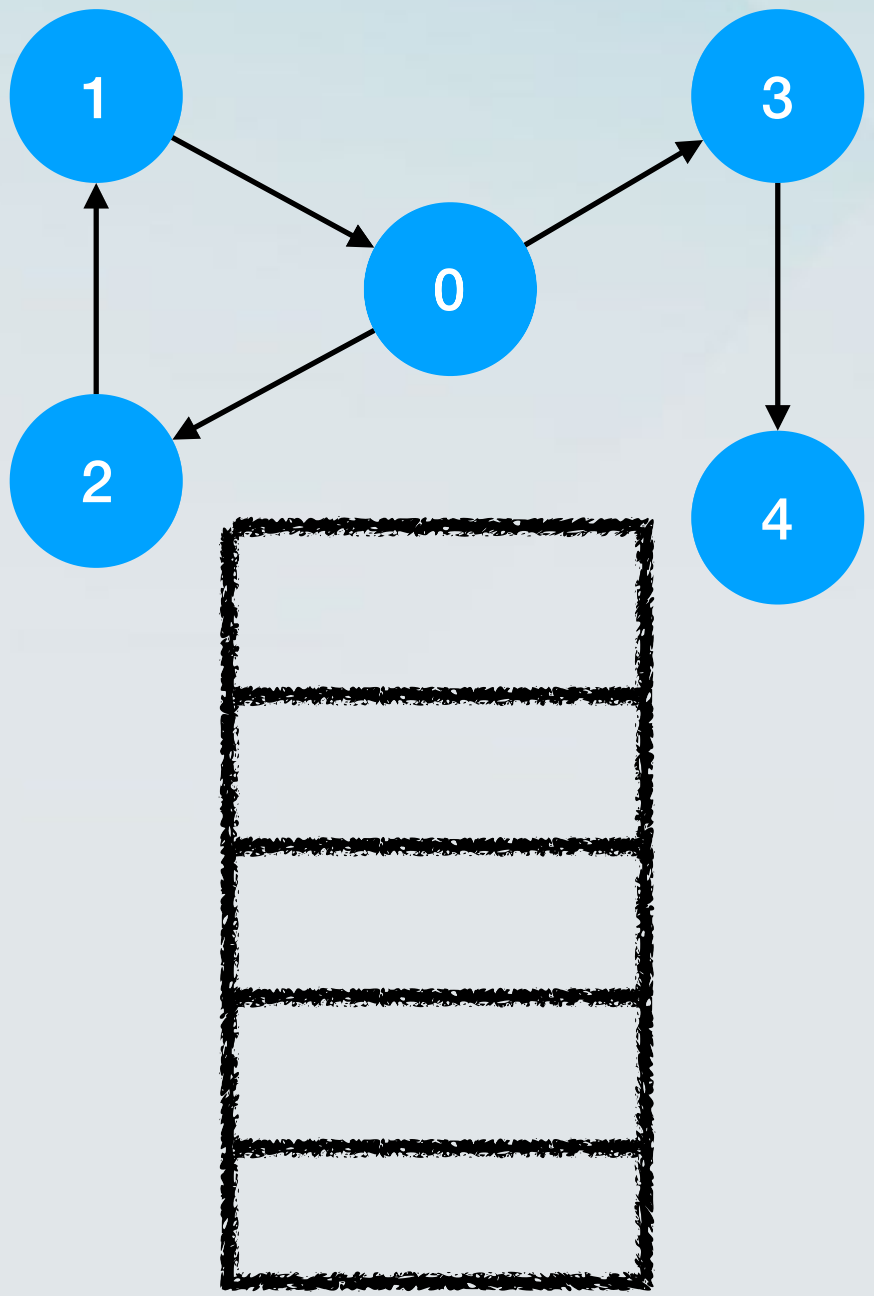
Kosaraju’s algorithm

Kosaraju's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
  - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.

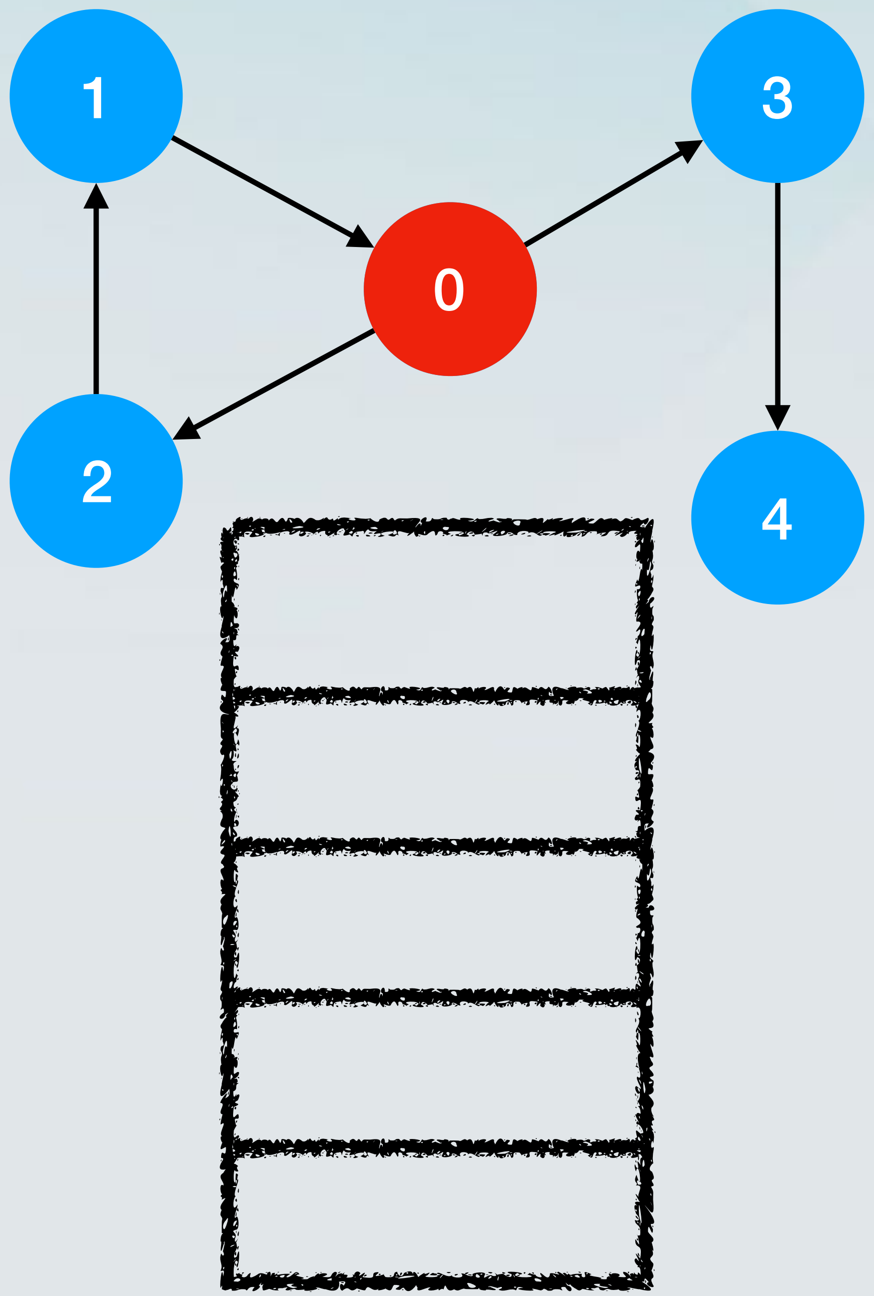
Kosaraju’s algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
  - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



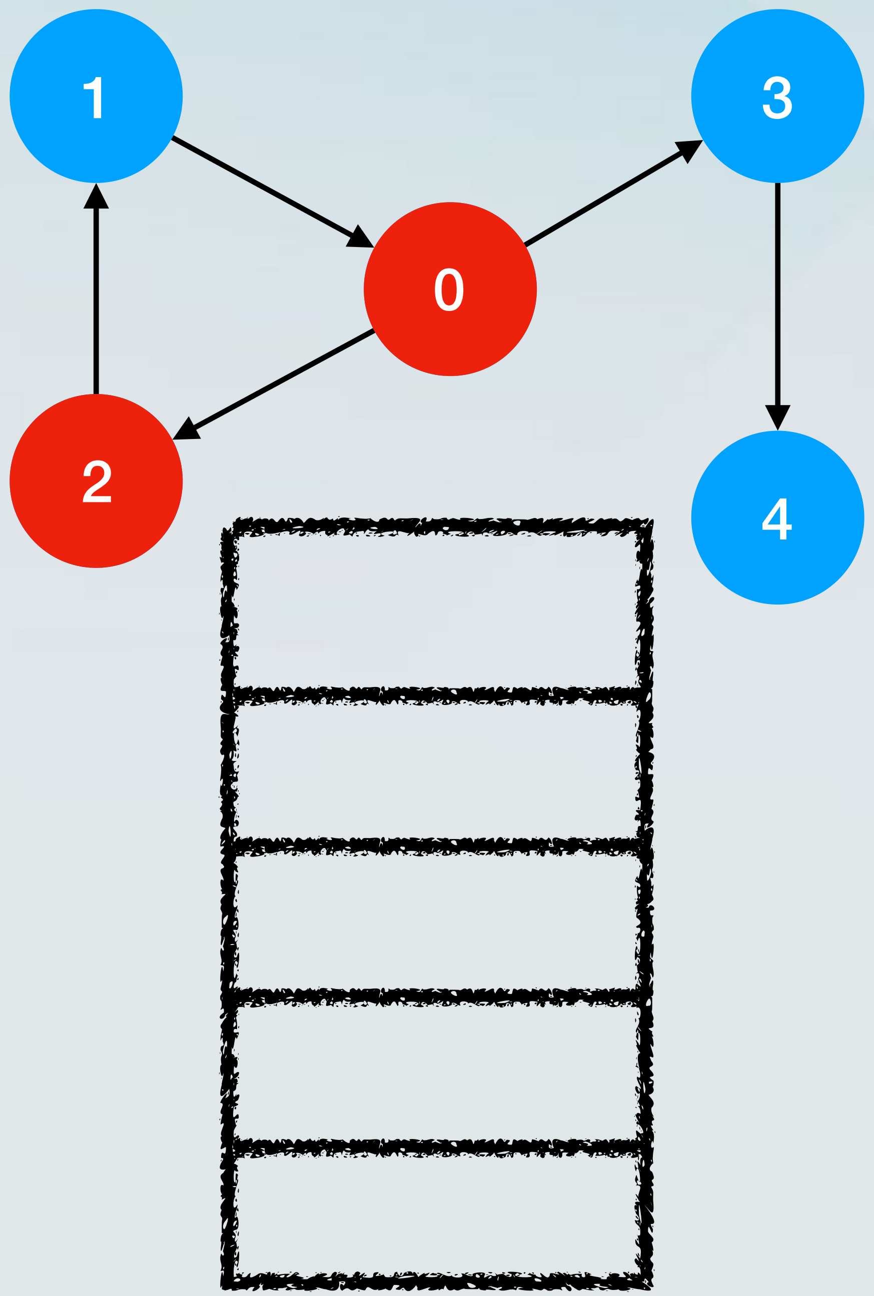
Kosaraju's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



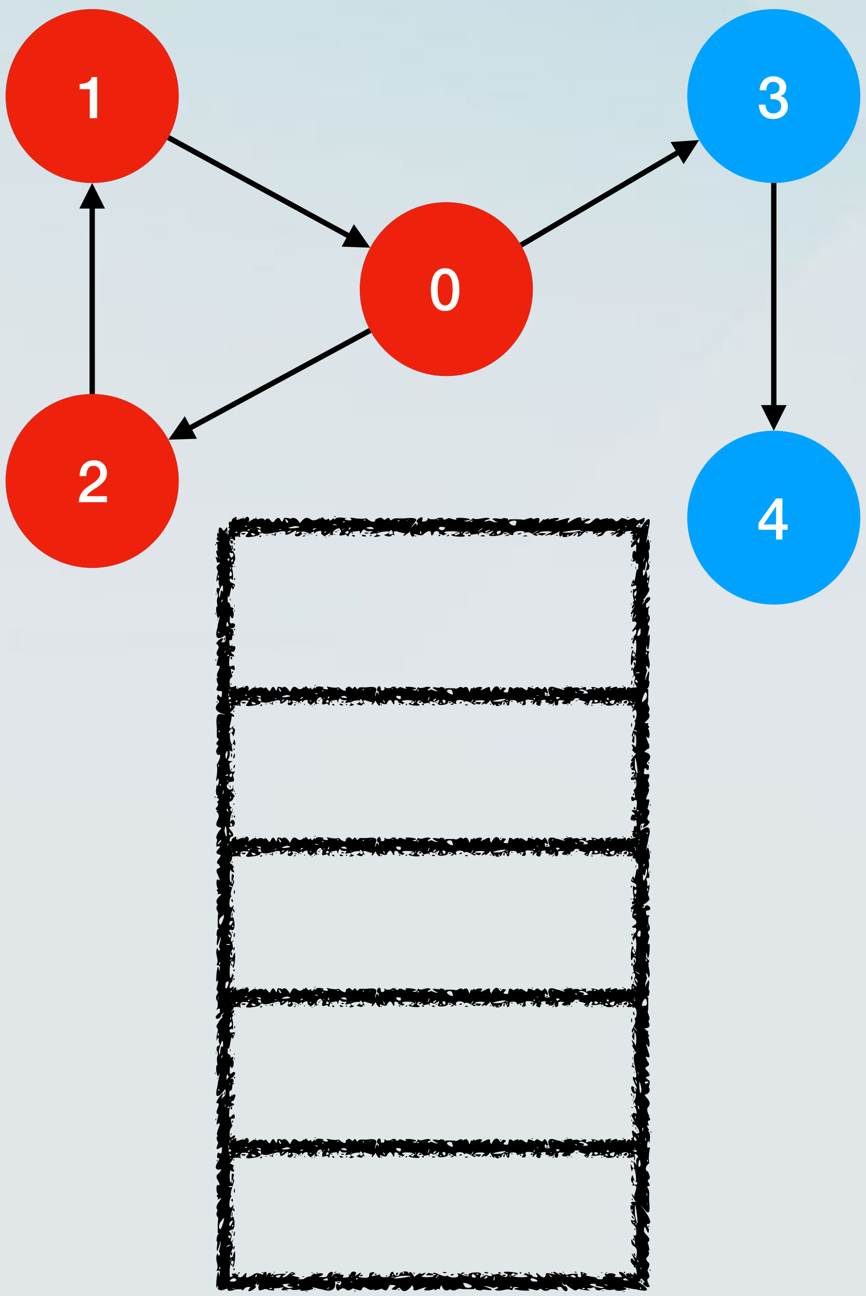
Kosaraju's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



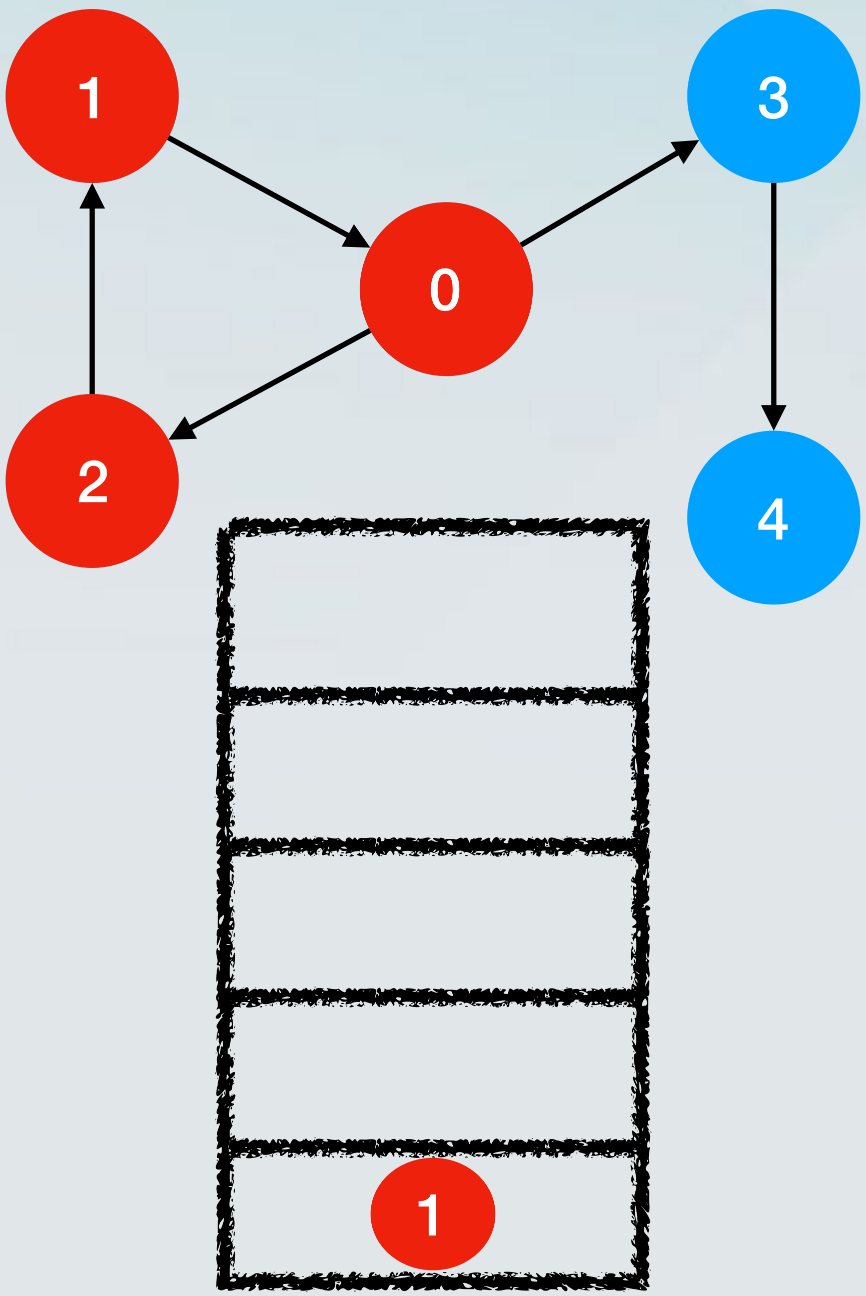
Kosaraju's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



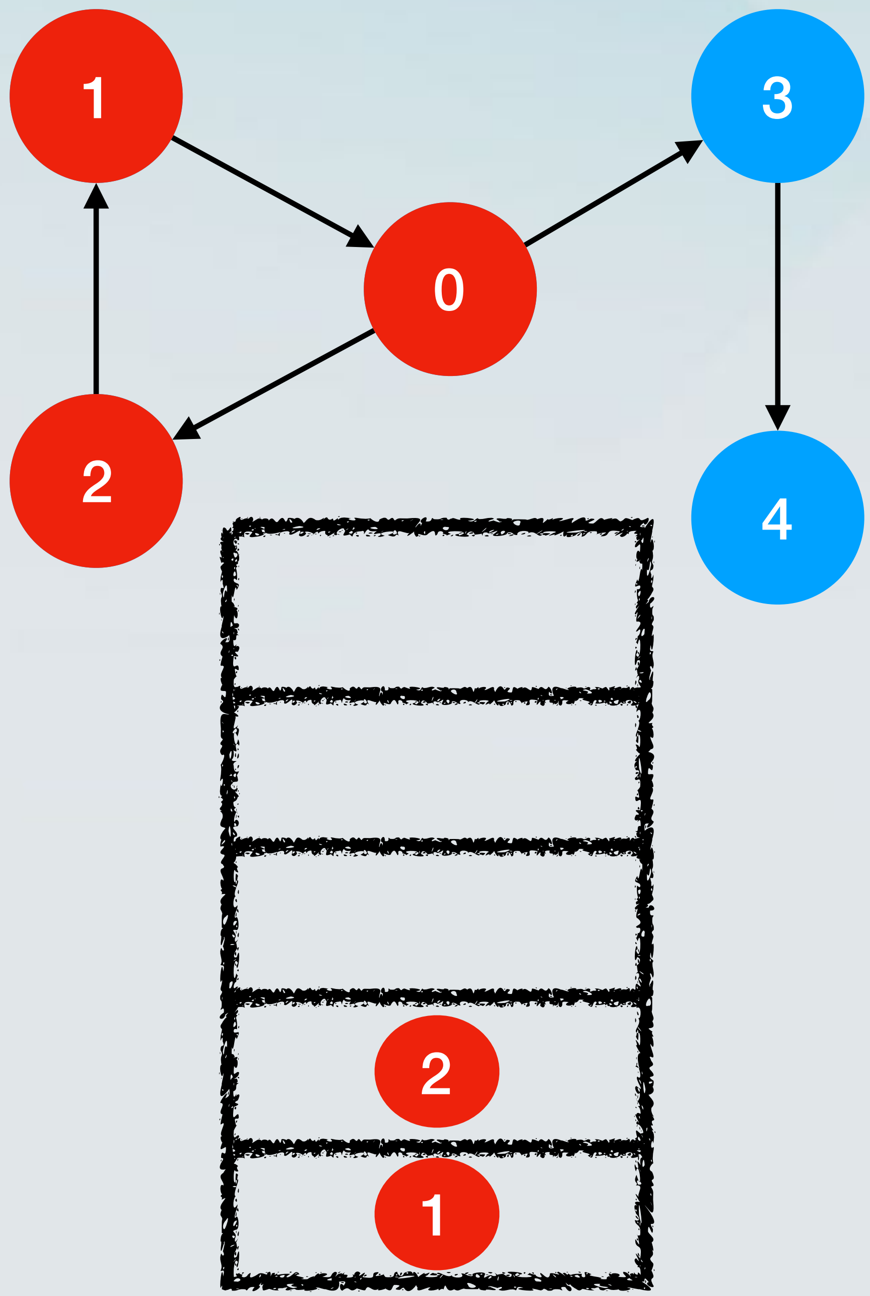
Kosaraju's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
  - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



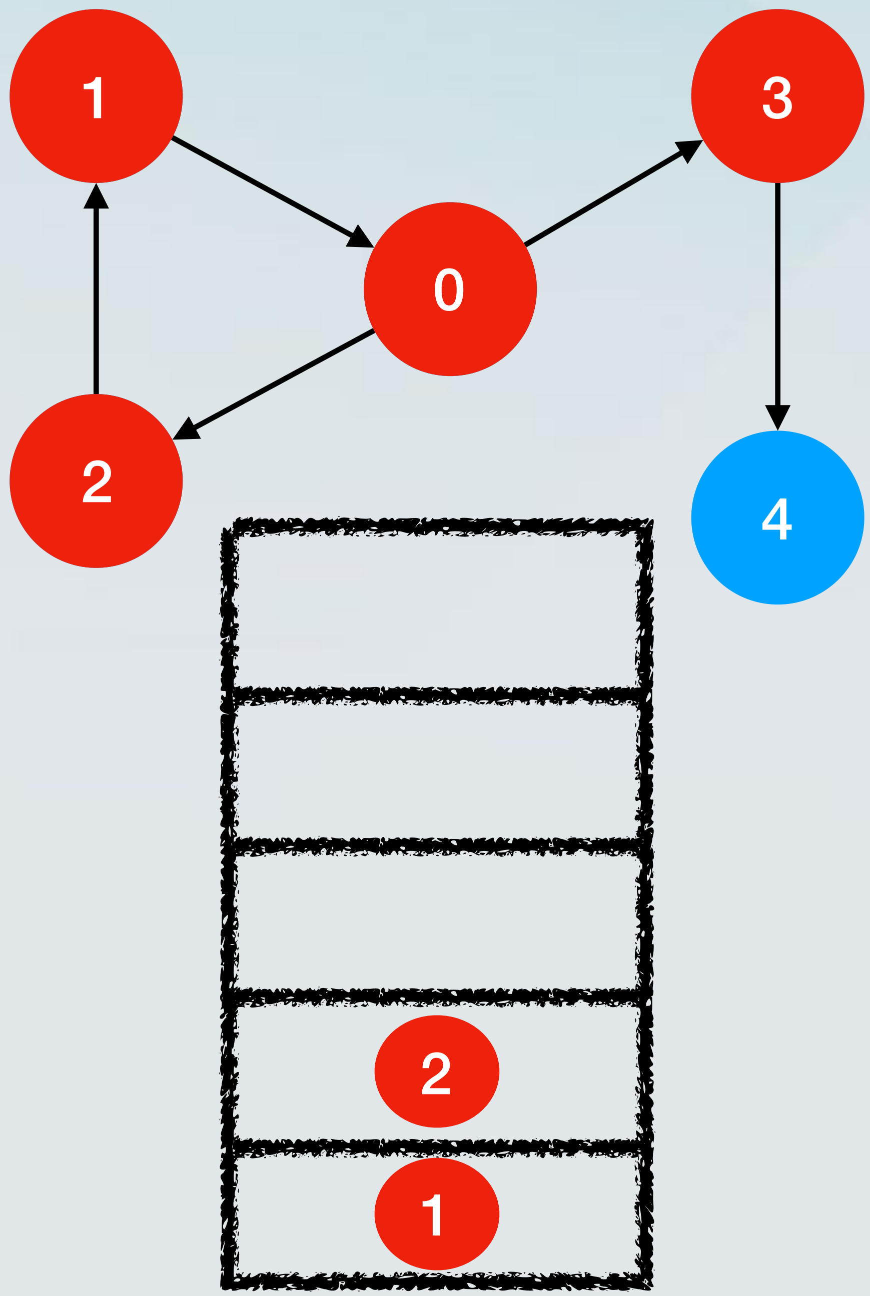
Kosaraju's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



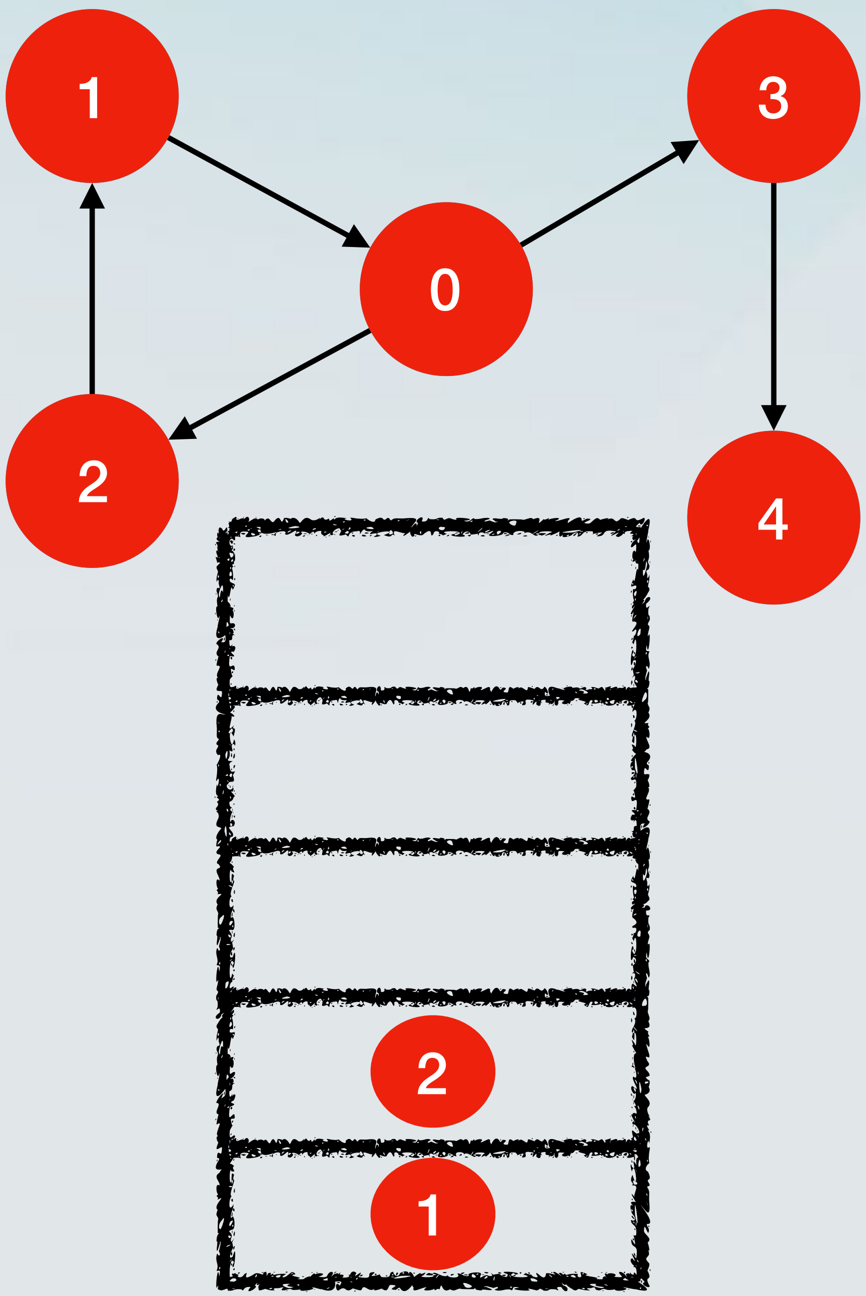
Kosaraju's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



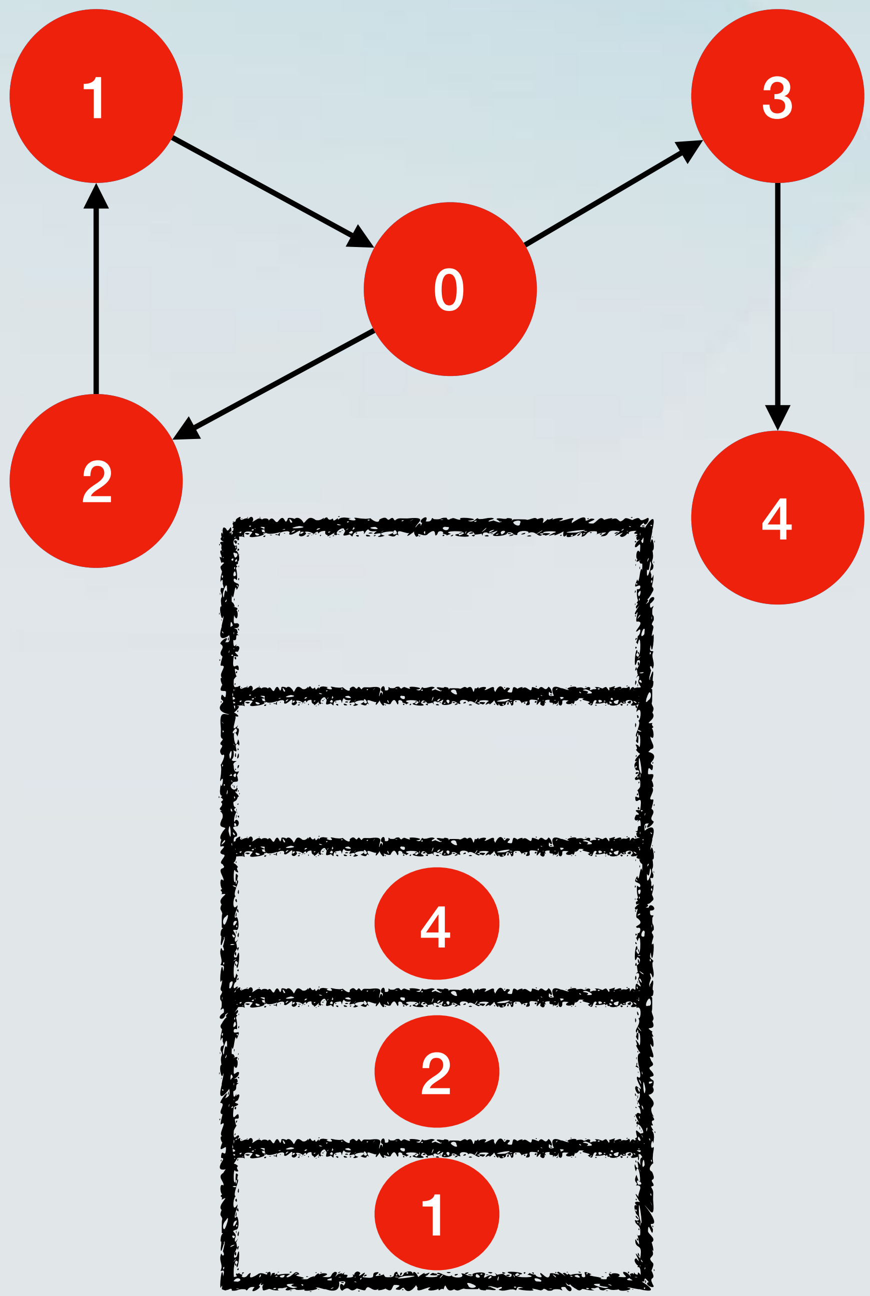
Kosaraju's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
  - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



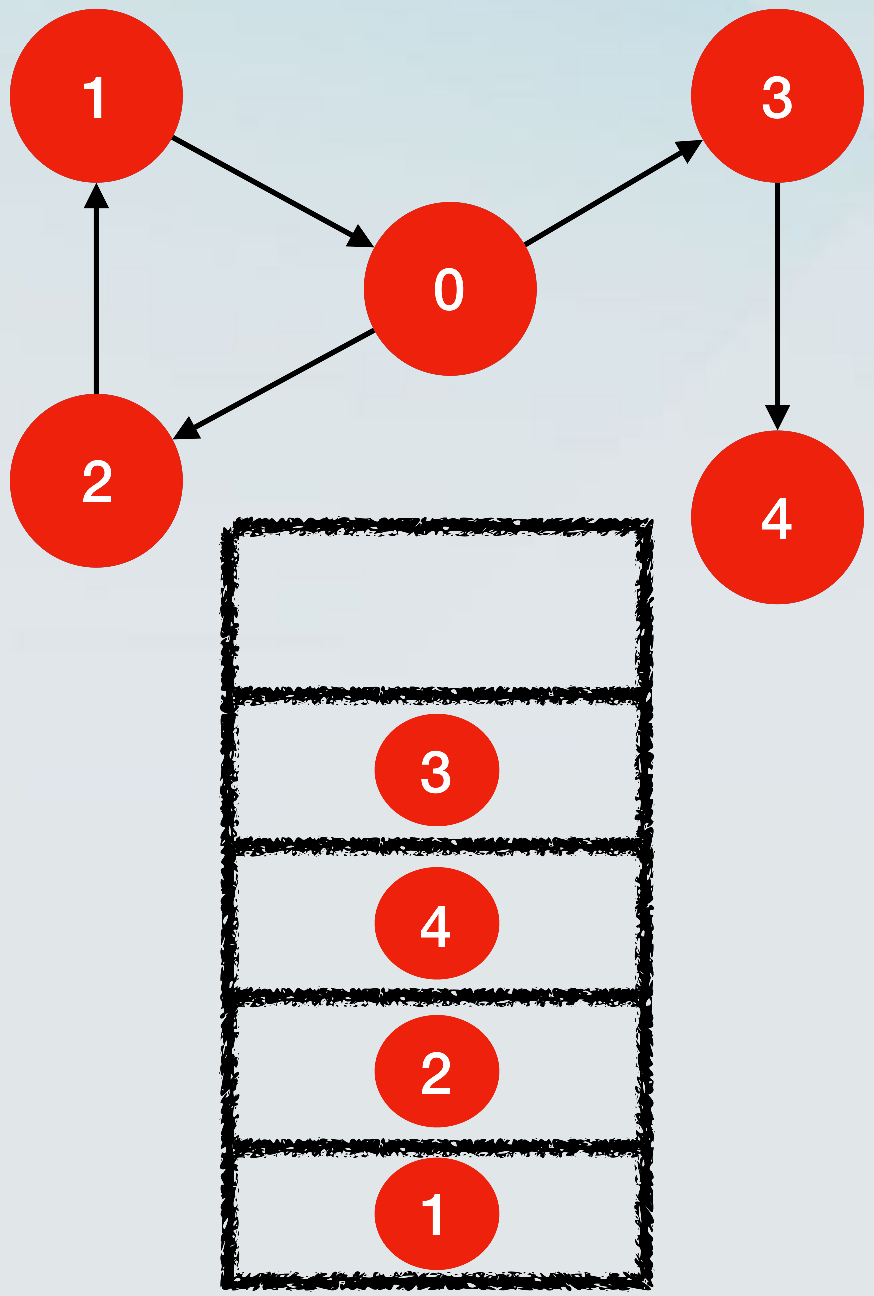
Kosaraju's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



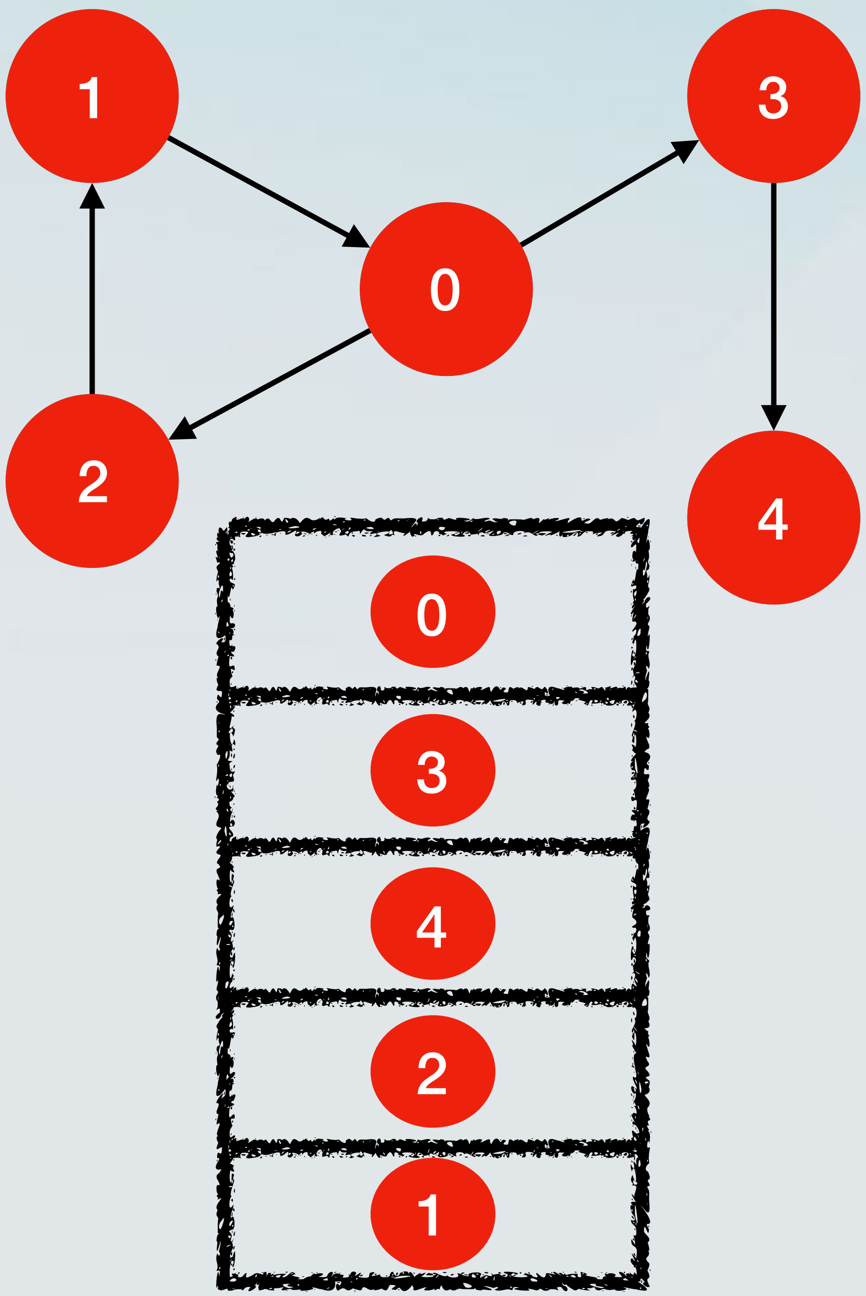
Kosaraju's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



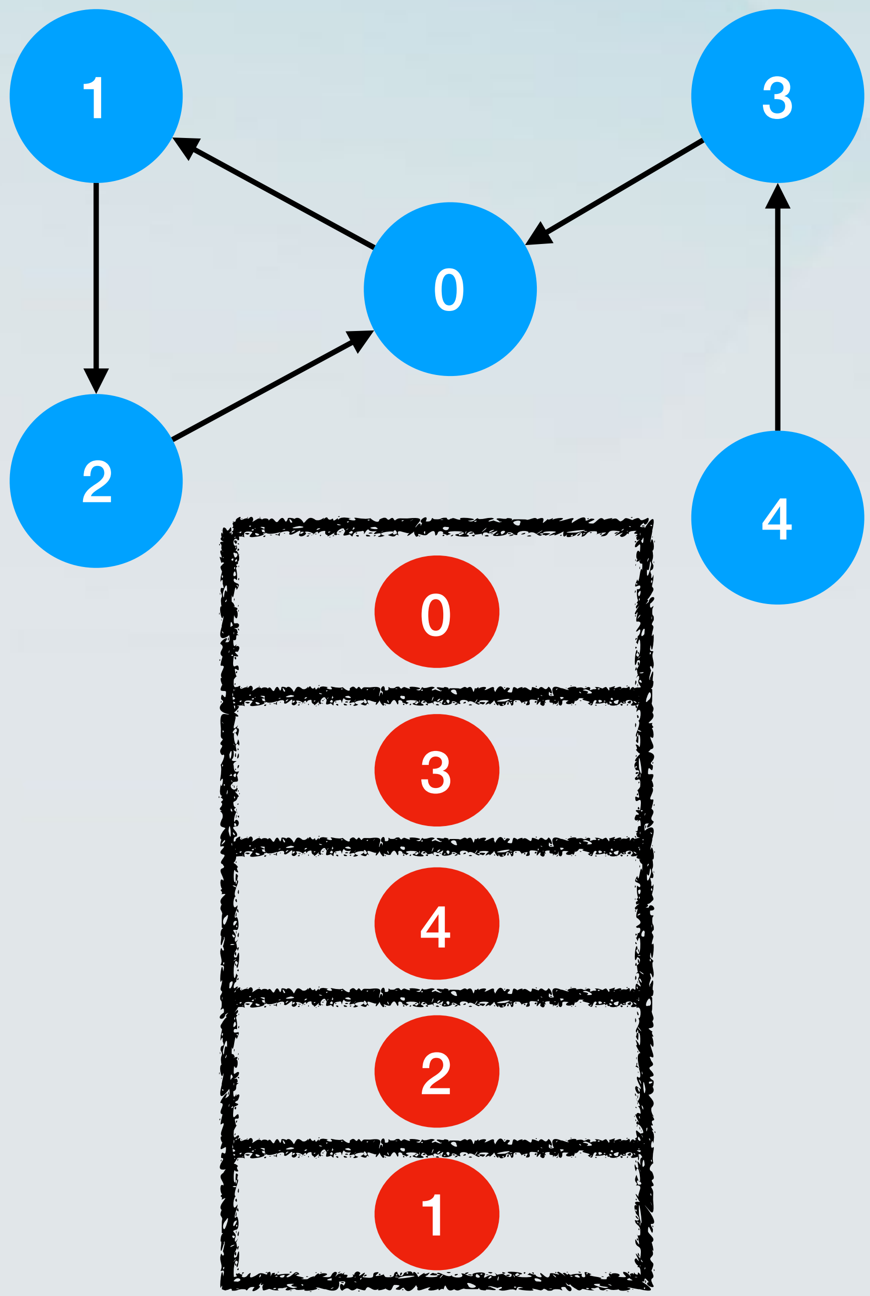
Kosaraju's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



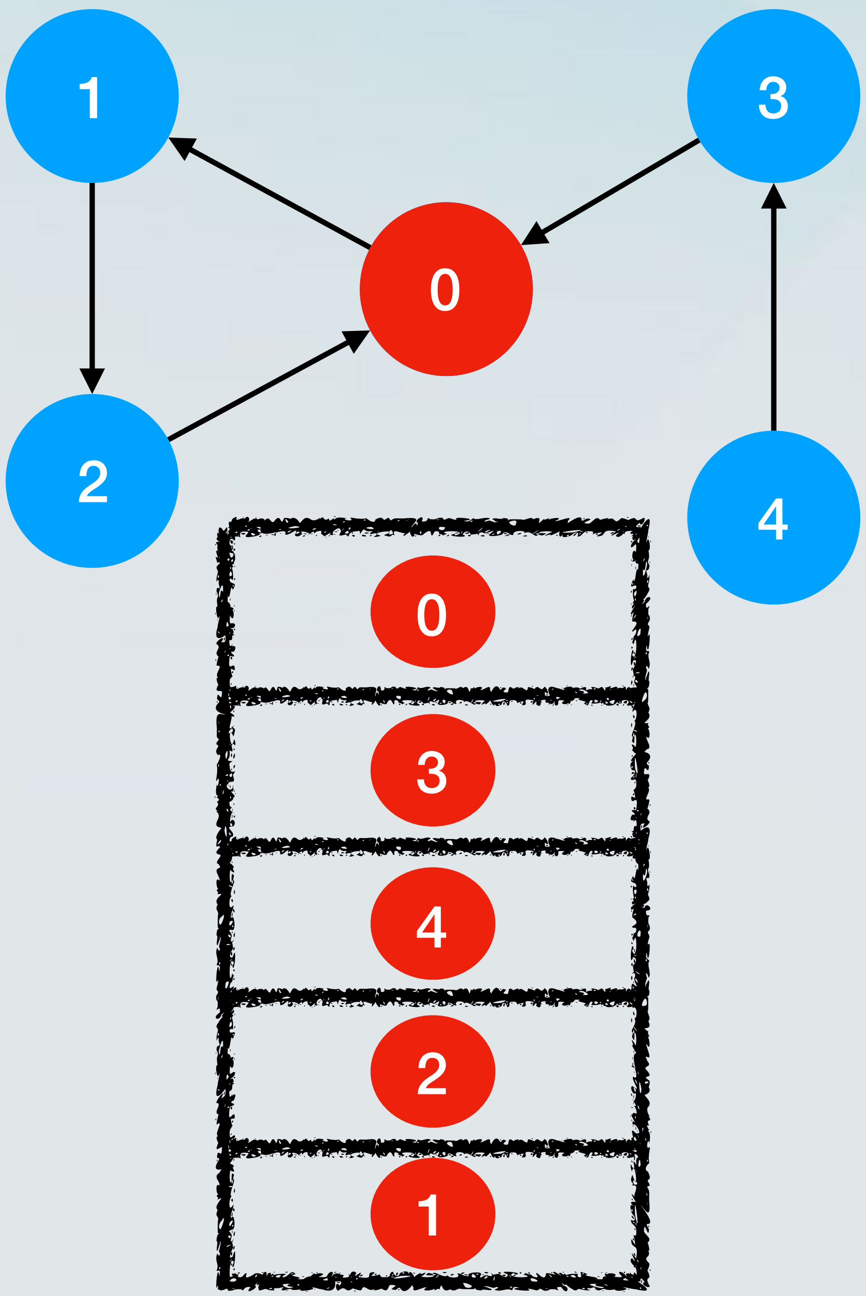
Kosaraju's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



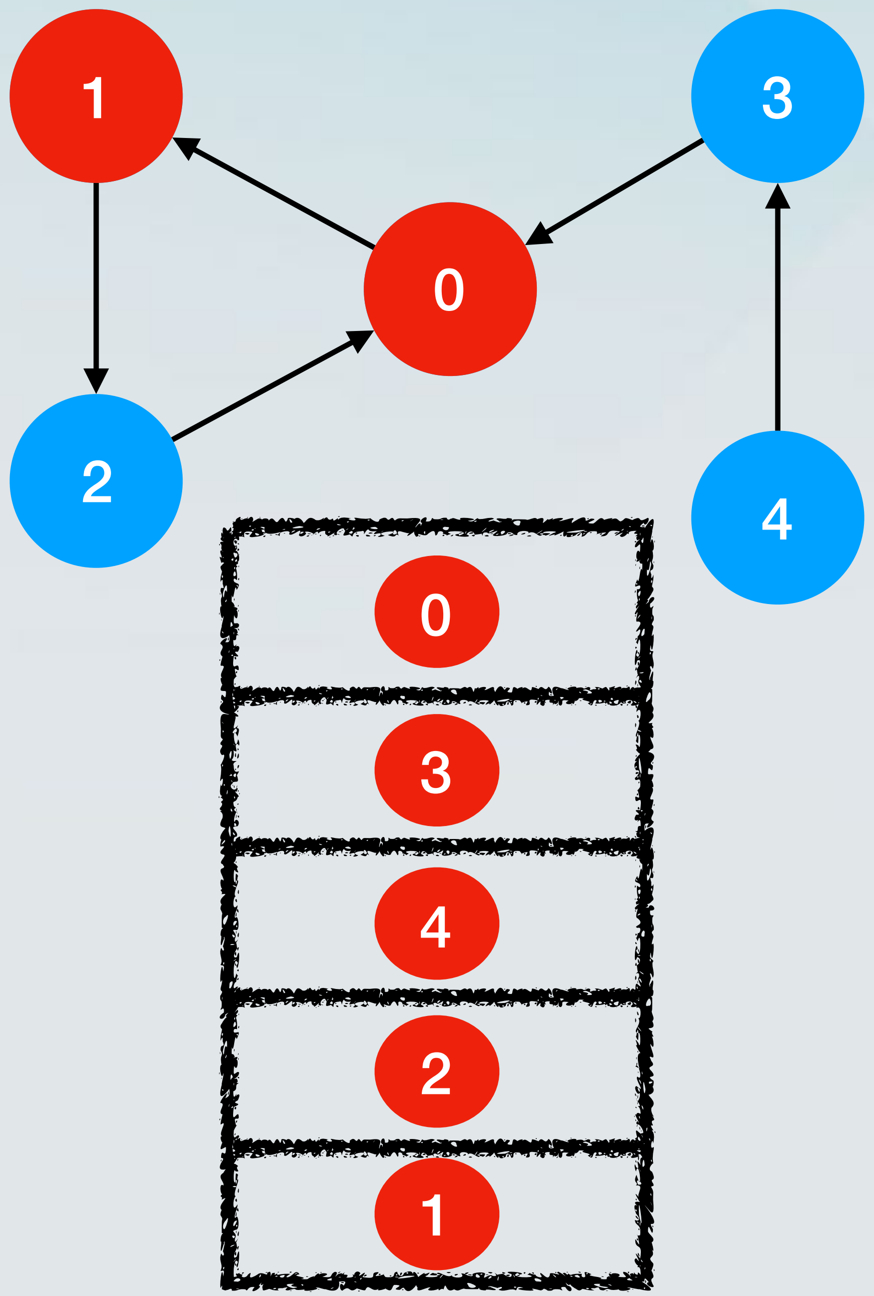
Kosaraju’s algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



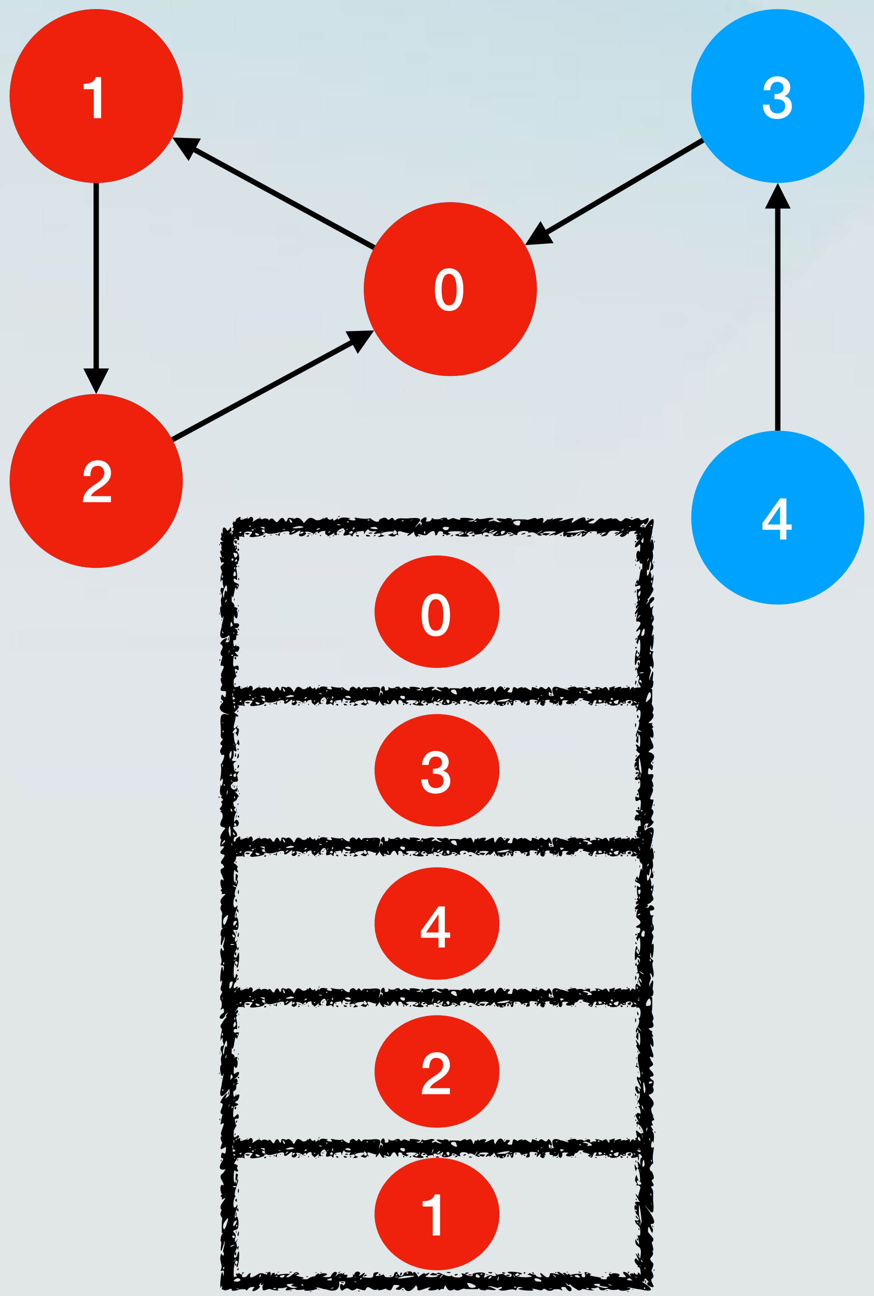
Kosaraju's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



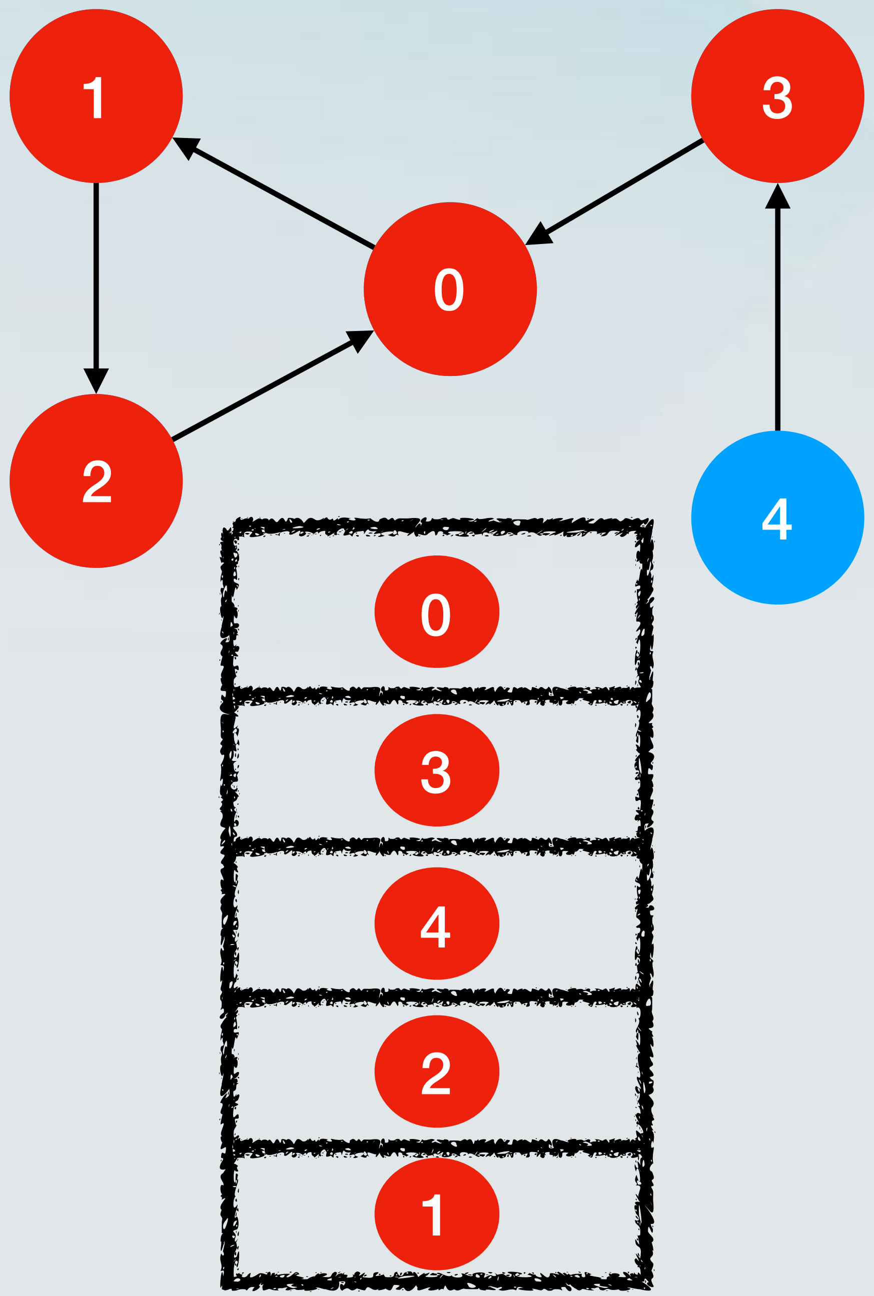
Kosaraju's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
  - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



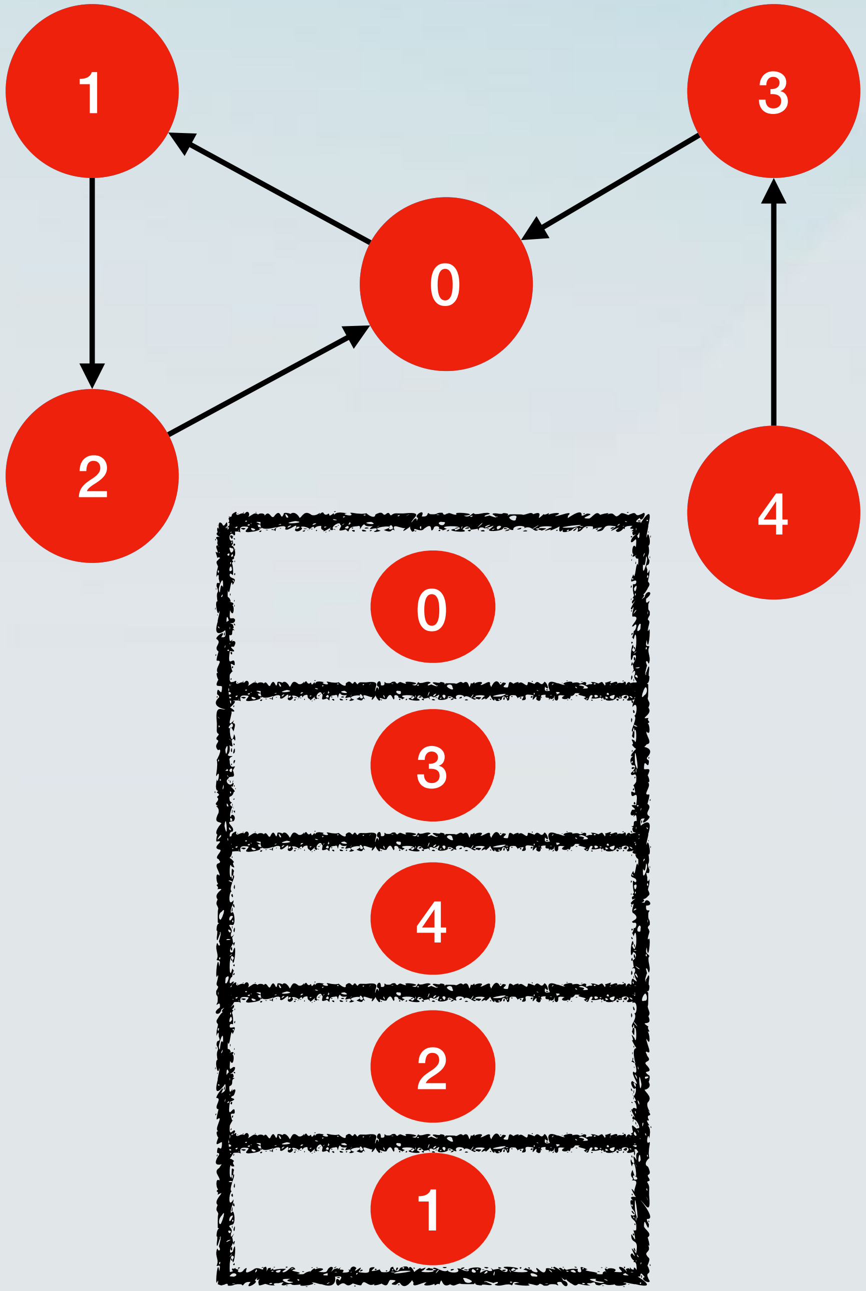
Kosaraju’s algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



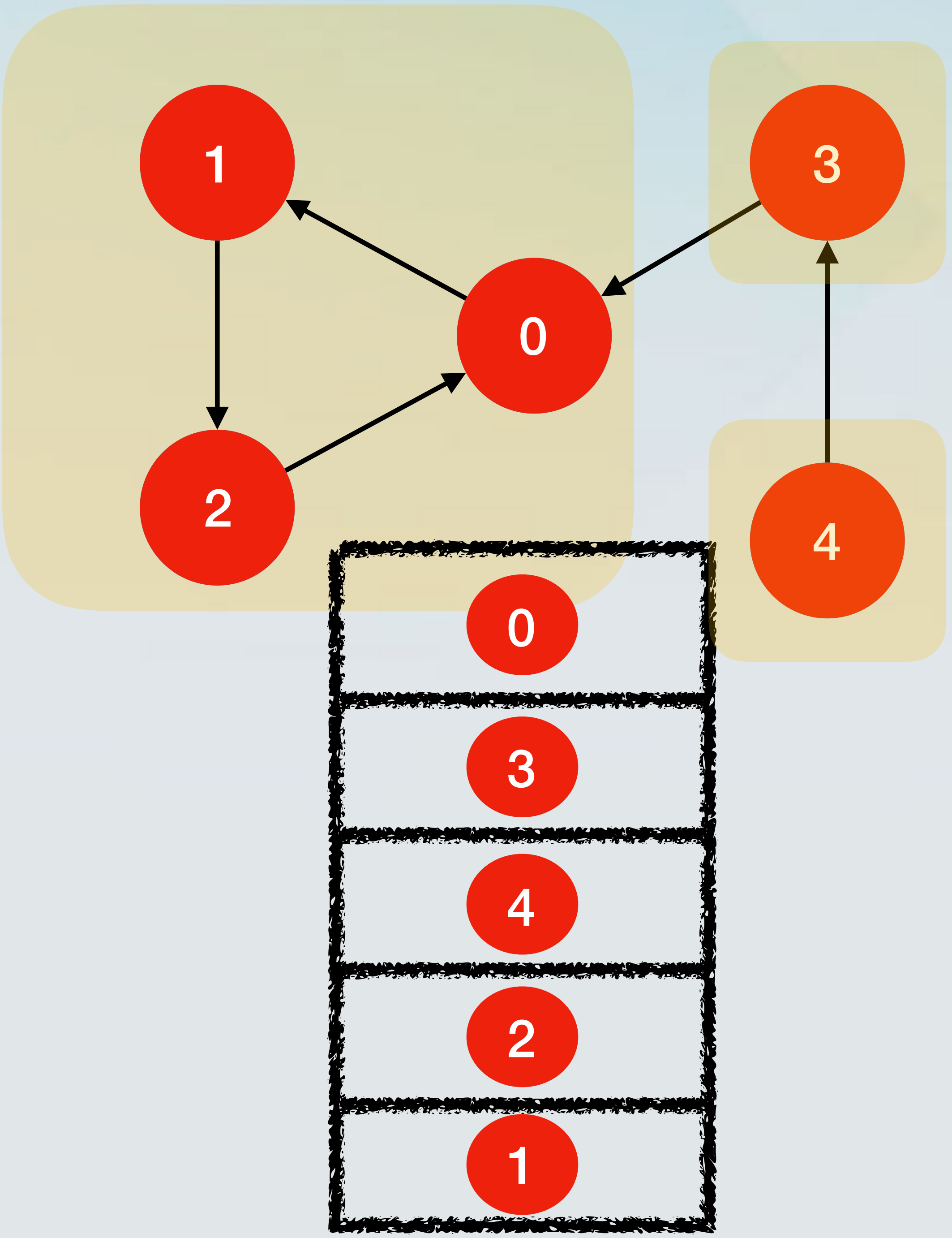
Kosaraju’s algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
    - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



Kosaraju's algorithm

- Perform a DFS on G, starting from an arbitrary nodes s.
- Add the nodes that the DFS tree reaches to a stack.
  - A node is added to the stack when the DFS for that node is completed.
- Perform a DFS on G^{rev}, visiting the nodes in the order that they are popped from the stack.
- Output the DFS trees of the second DFS as the strongly connected components.



Running time

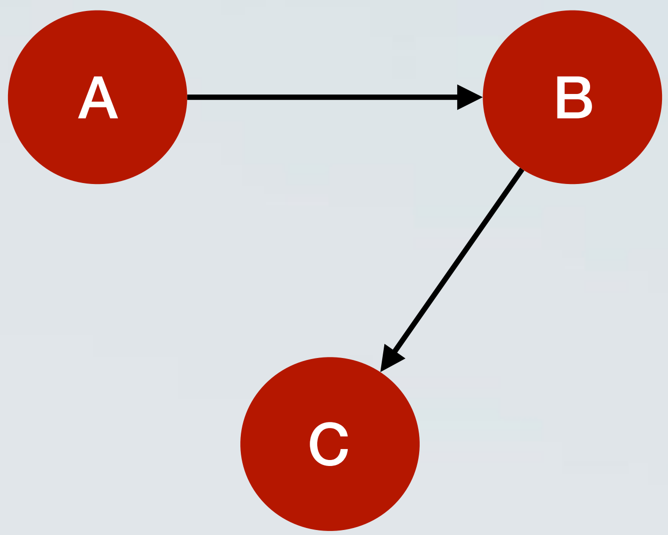
Running time

- We perform DFS twice.
- The running time is **O(m+n)**.

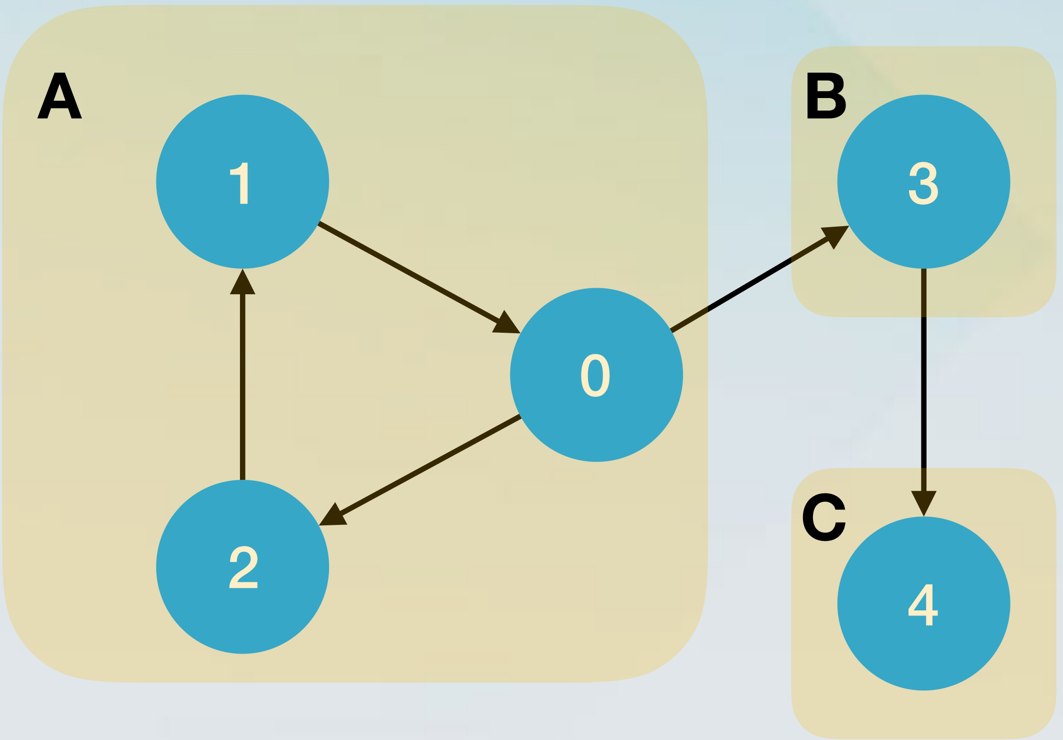
Correctness

- Define a *meta-graph* of the graph G, called the component graph $G^{SCC} = (V^{SCC}, E^{SCC})$
- Suppose that G has strongly connected components (SCCs) $C_1, C_2, \ldots, C_k$, for some k.
- $V^{SCC} = \{v_1, v_2, \ldots, v_k\}$ contains a vertex for each SCC of G.
- There is an edge (v_i, v_j) in E^{SCC} if G contains a directed edge (x,y) for some x in C_i and some y in C_j (i.e., an edge *crossing* two different components.

Example



Component Graph G^{SCC}



Graph G

Simple but key lemma

Simple but key lemma

- Let C and C' be distinct SCCs in a directed graph G.
  Let u, v in C
  Let u',v' in C'
  Suppose that G contains a path from u to u'. (1)
  Then G ***cannot*** contain a path from v' to v.

Simple but key lemma

- Let C and C' be distinct SCCs in a directed graph G.
  Let u, v in C
  Let u',v' in C'
  Suppose that G contains a path from u to u'. (1)
  Then G ***cannot*** contain a path from v' to v.

- **Proof (by contradiction)**:

Simple but key lemma

- Let C and C' be distinct SCCs in a directed graph G.
  Let u, v in C
  Let u',v' in C'
  Suppose that G contains a path from u to u'. (1)
  Then G ***cannot*** contain a path from v' to v.

- **Proof (by contradiction)**:

  - Assume there is a path from v' to v. (2)

Simple but key lemma

- Let C and C’ be distinct SCCs in a directed graph G.
  Let u, v in C
  Let u’,v’ in C’
  Suppose that G contains a path from u to u’. (1)
  Then G ***cannot*** contain a path from v’ to v.

- **Proof (by contradiction)**:
  - Assume there is a path from v’ to v. (2)
  - There is a path from u’ to v’ (same SCC).

Simple but key lemma

- Let C and C' be distinct SCCs in a directed graph G.
  Let u, v in C
  Let u',v' in C'
  Suppose that G contains a path from u to u'. (1)
  Then G ***cannot*** contain a path from v' to v.

- **Proof (by contradiction)**:
  - Assume there is a path from v' to v. (2)
  - There is a path from u' to v' (same SCC).
  - There is path from u to v' (because of (1)).

Simple but key lemma

- Let C and C' be distinct SCCs in a directed graph G.
  Let u, v in C
  Let u',v' in C'
  Suppose that G contains a path from u to u'. (1)
  Then G ***cannot*** contain a path from v' to v.

- **Proof (by contradiction)**:
  - Assume there is a path from v' to v. (2)
  - There is a path from u' to v' (same SCC).
  - There is path from u to v' (because of (1)).
  - There is a path from v to u (same SCC).

Simple but key lemma

- Let C and C' be distinct SCCs in a directed graph G.
  Let u, v in C
  Let u',v' in C'
  Suppose that G contains a path from u to u'. (1)
  Then G ***cannot*** contain a path from v' to v.

- **Proof (by contradiction)**:
  - Assume there is a path from v' to v. (2)
  - There is a path from u' to v' (same SCC).
  - There is path from u to v' (because of (1)).
  - There is a path from v to u (same SCC).
  - There is path from v' to u (because of (2)).

Simple but key lemma

- Let C and C’ be distinct SCCs in a directed graph G.
  Let u, v in C
  Let u’,v’ in C’
  Suppose that G contains a path from u to u’. (1)
  Then G ***cannot*** contain a path from v’ to v.

- **Proof (by contradiction)**:
  - Assume there is a path from v’ to v. (2)
  - There is a path from u’ to v’ (same SCC).
  - There is path from u to v’ (because of (1)).
  - There is a path from v to u (same SCC).
  - There is path from v’ to u (because of (2)).
  - This means that u and v’ are mutually reachable, hence in the same SCC.

Component graph

Component graph

- What does the simple-but-key lemma imply for the component graph?

Component graph

- What does the simple-but-key lemma imply for the component graph?
- For two distinct connected components, there can be a path from the first to the second, or vice-versa, but not both!

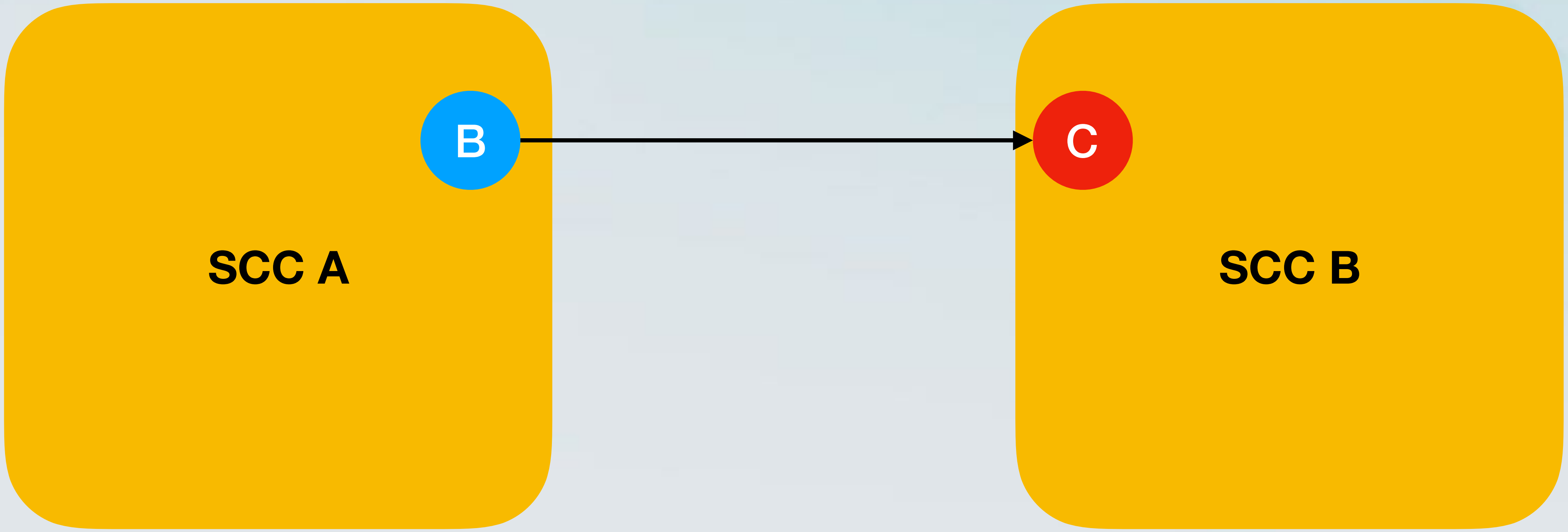
Component graph

- What does the simple-but-key lemma imply for the component graph?
- For two distinct connected components, there can be a path from the first to the second, or vice-versa, but not both!
- **The component graph is a DAG.**

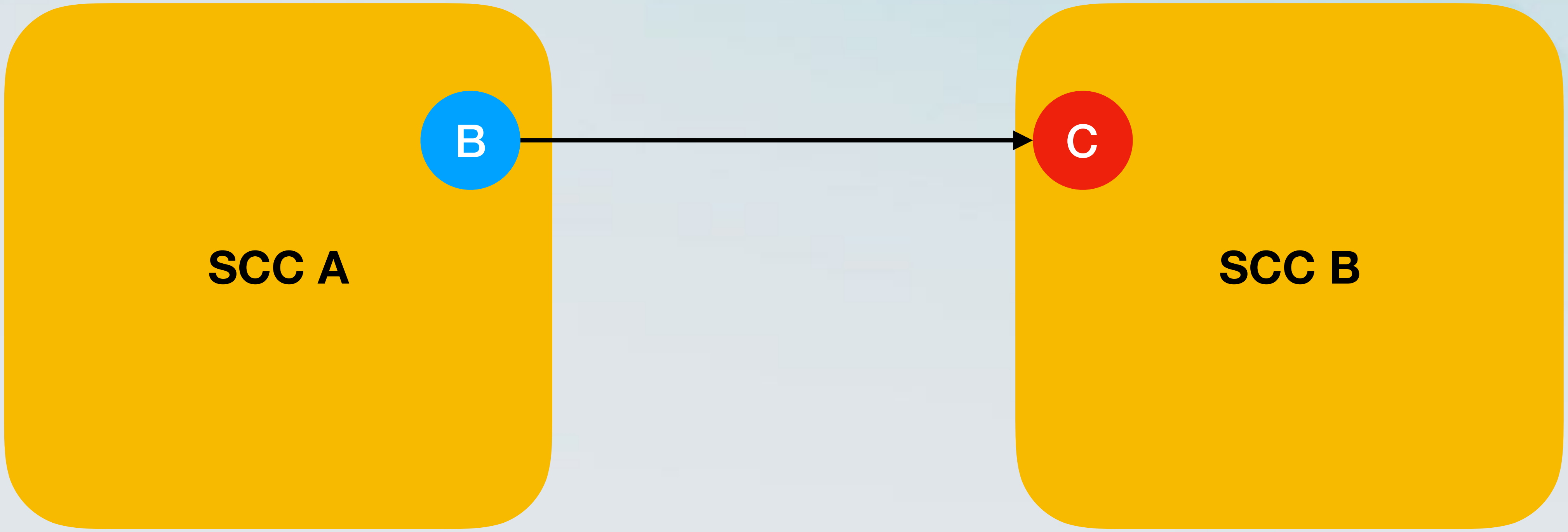
Lemma

- Let C and C’ be distinct SCCs in G. Suppose there is a directed edge ***crossing*** C and C’. Then the DFS on the nodes of C finishes later than the DFS on the nodes of C’.

Proof by picture

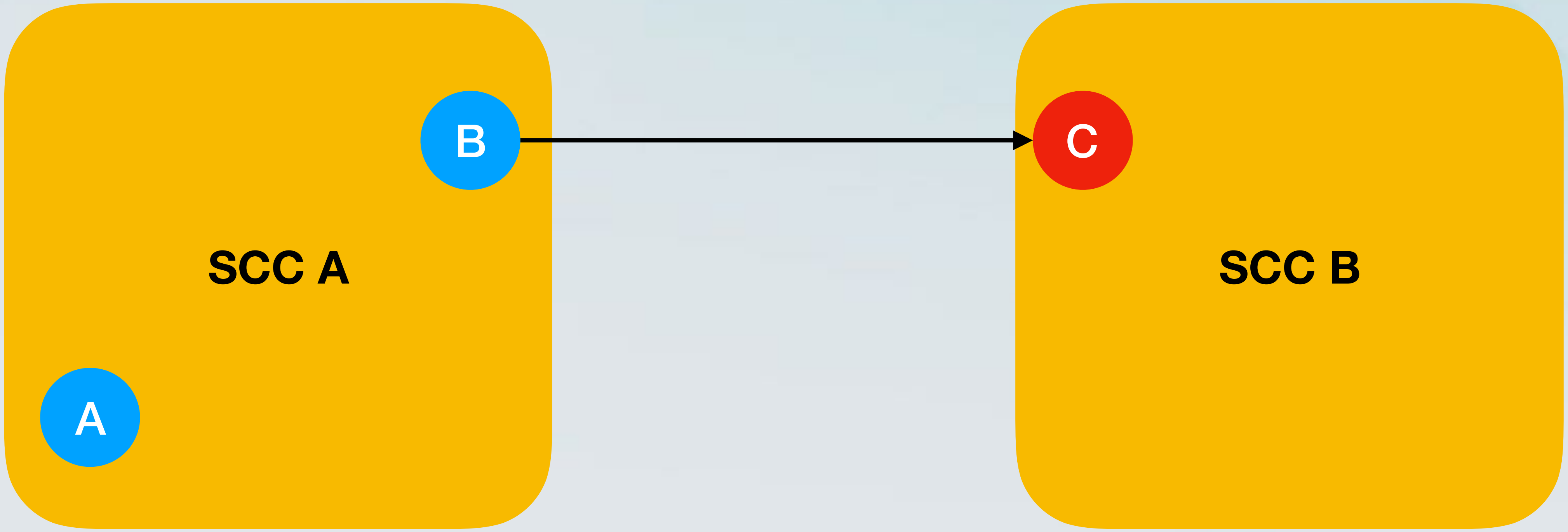


Proof by picture



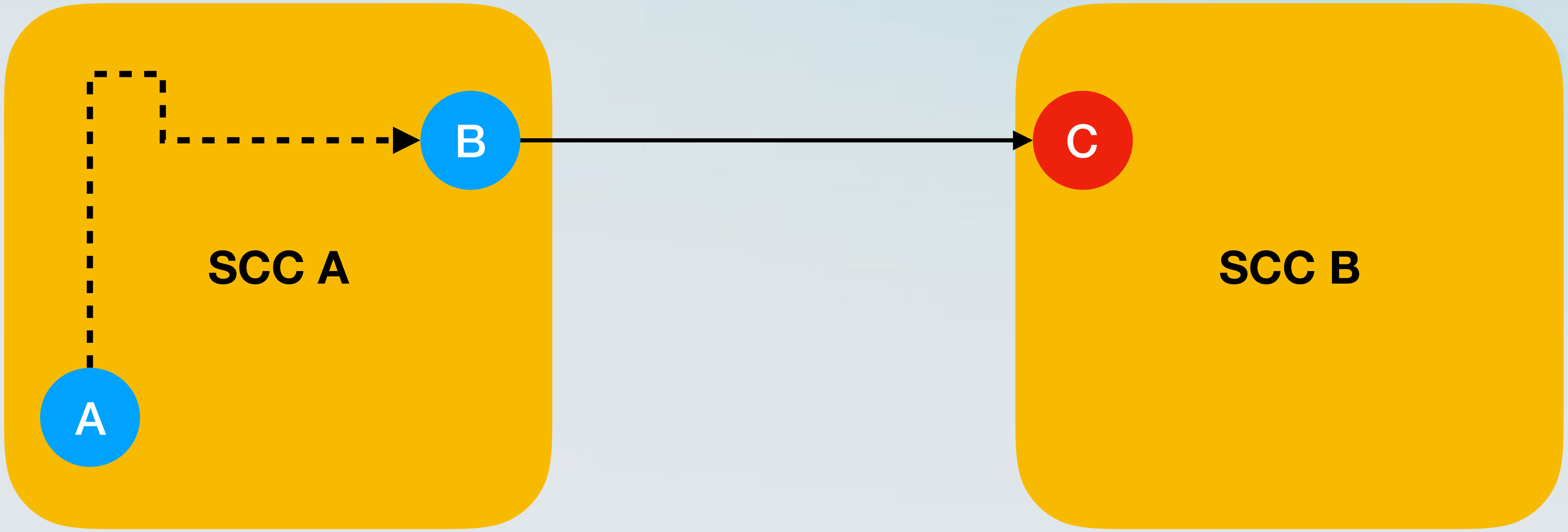
Case 1: We start here

Proof by picture



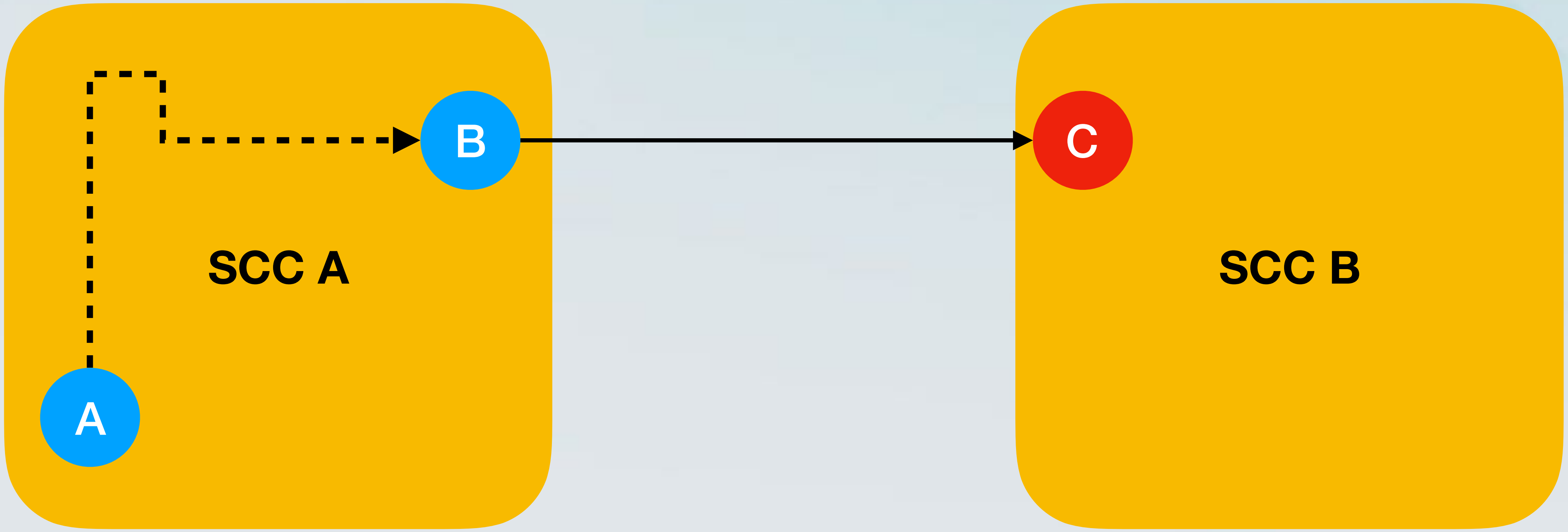
Case 1: We start here

Proof by picture



Case 1: We start here

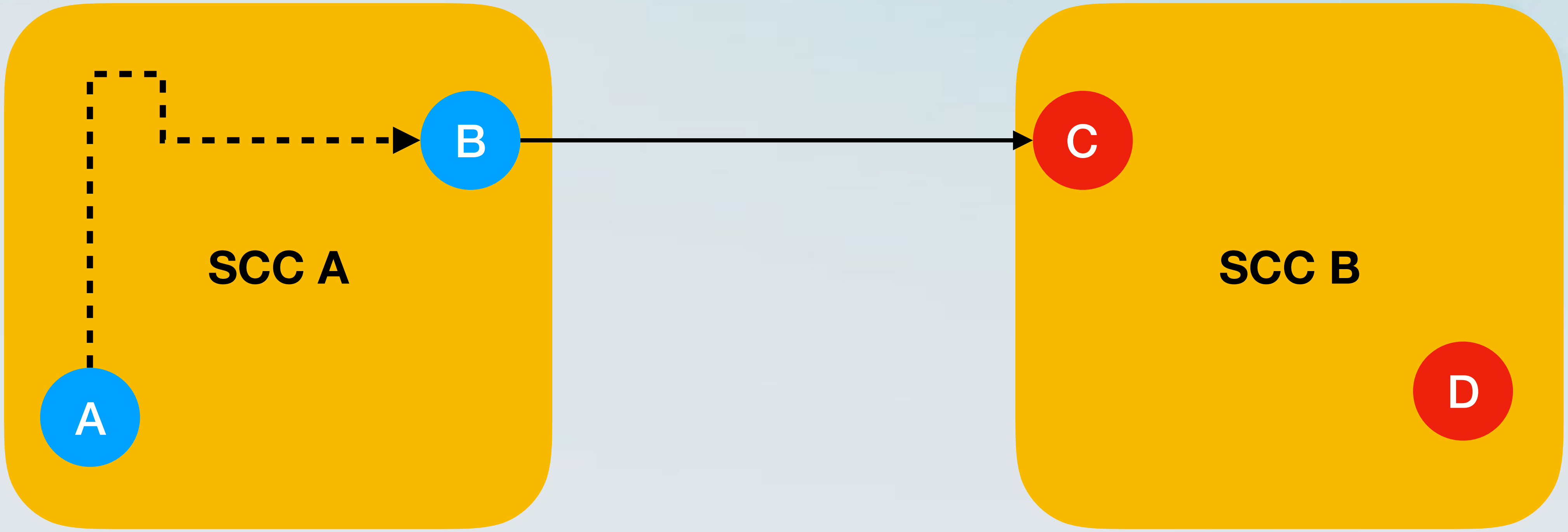
Proof by picture



Case 1: We start here

There is a path A-B
because of same SCC

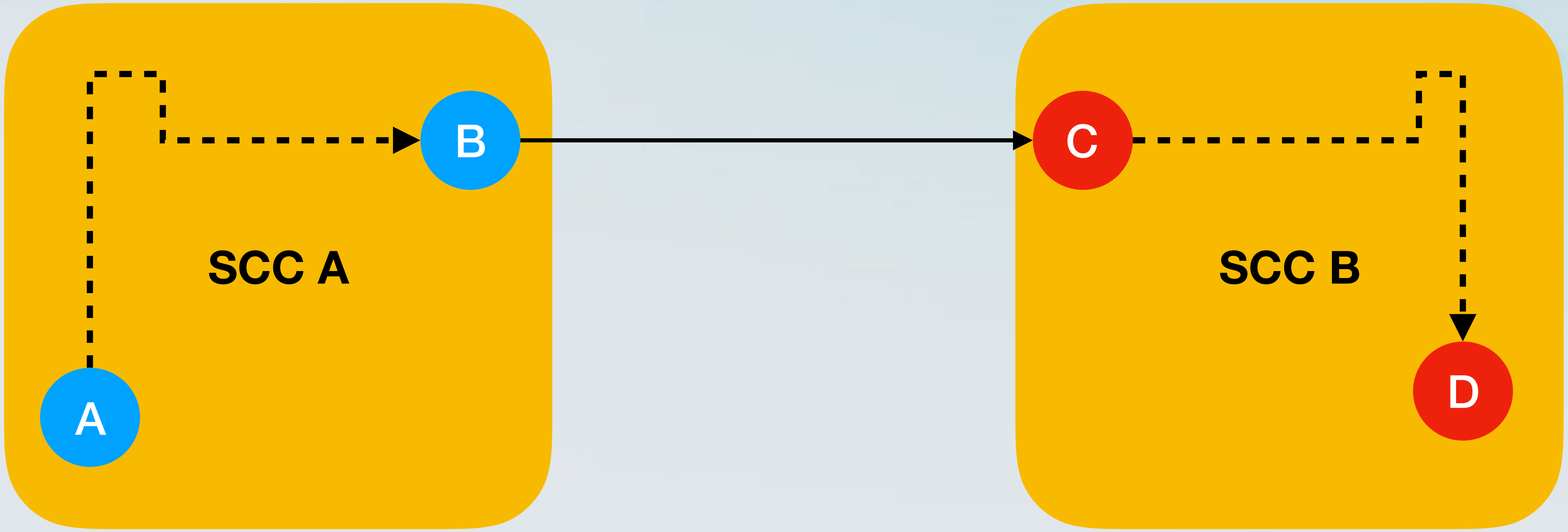
Proof by picture



Case 1: We start here

There is a path A-B because of same SCC

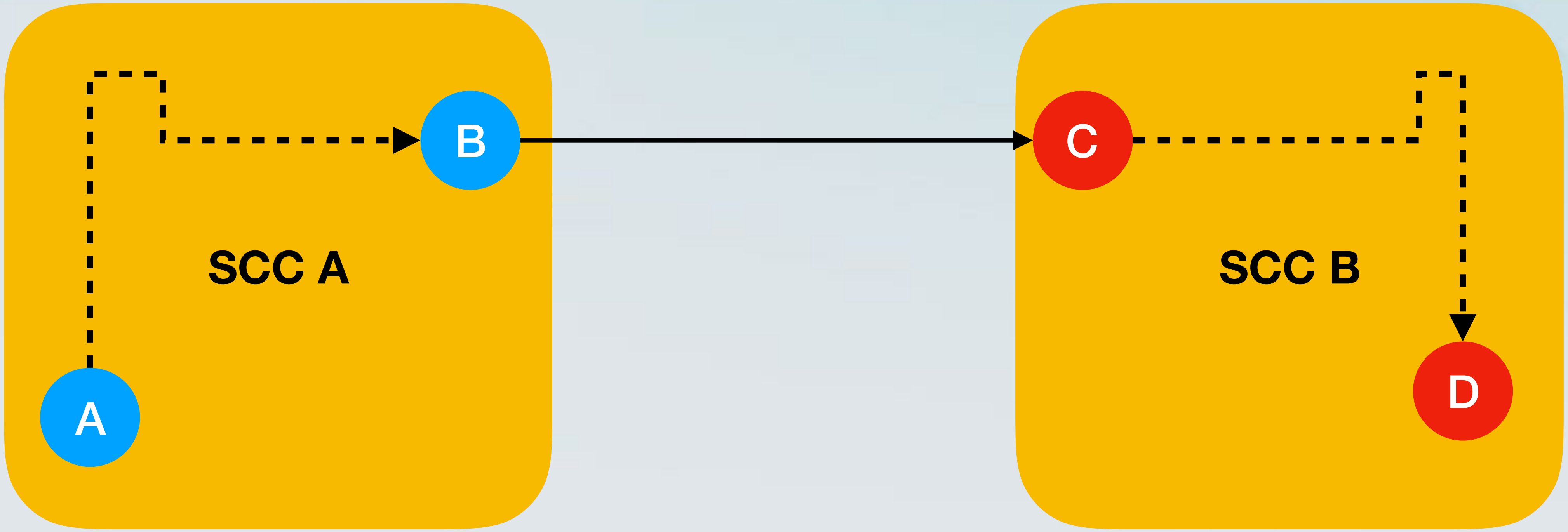
Proof by picture



Case 1: We start here

There is a path A-B
because of same SCC

Proof by picture

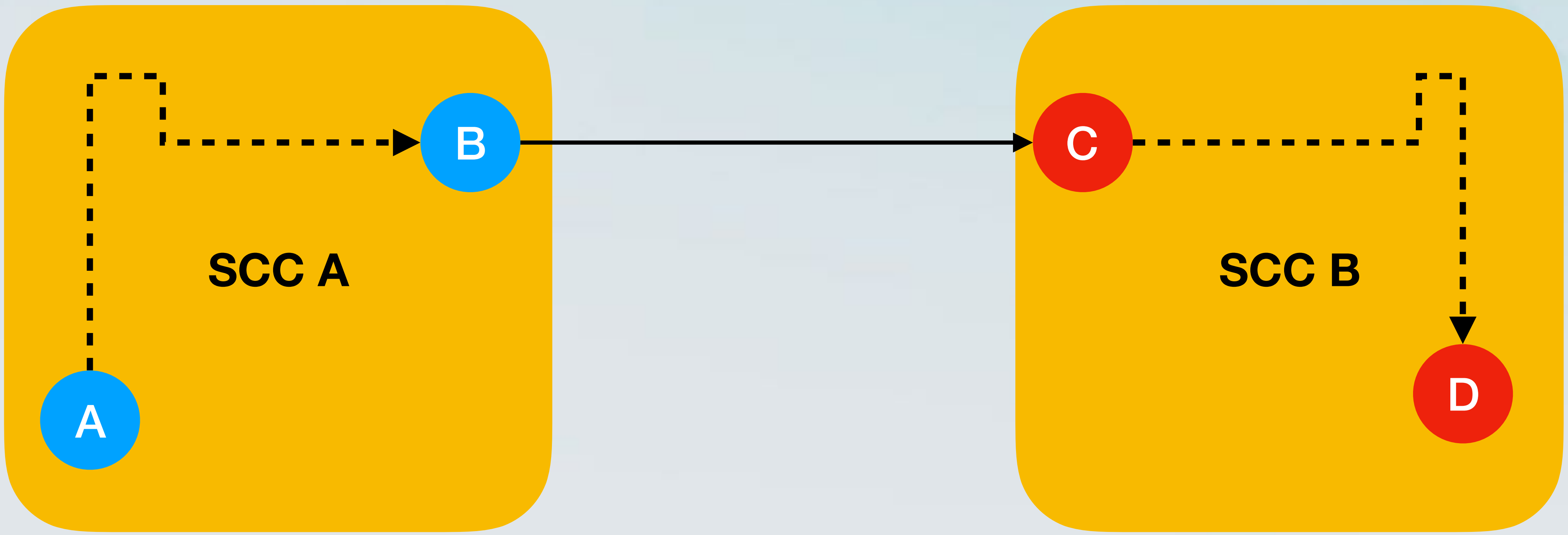


Case 1: We start here

There is a path A-B
because of same SCC

There is a path C-D
because of same SCC

Proof by picture



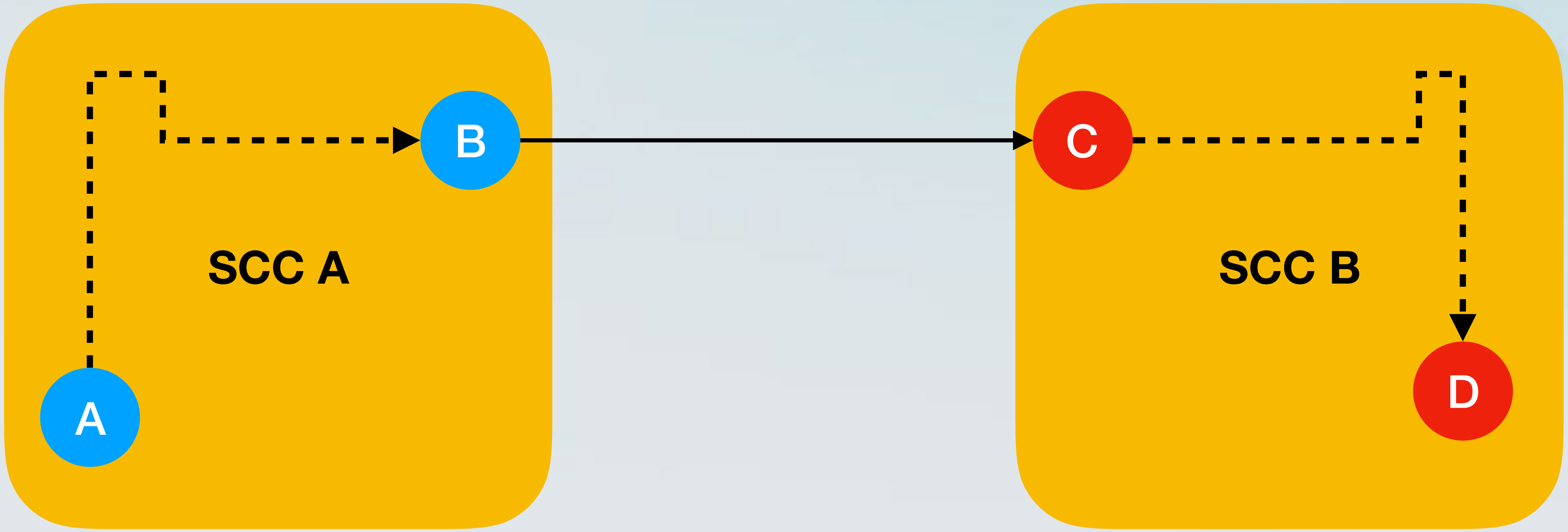
Case 1: We start here

There is a path A-B
because of same SCC

There is a path C-D
because of same SCC

There is path A-D

Proof by picture



Case 1: We start here

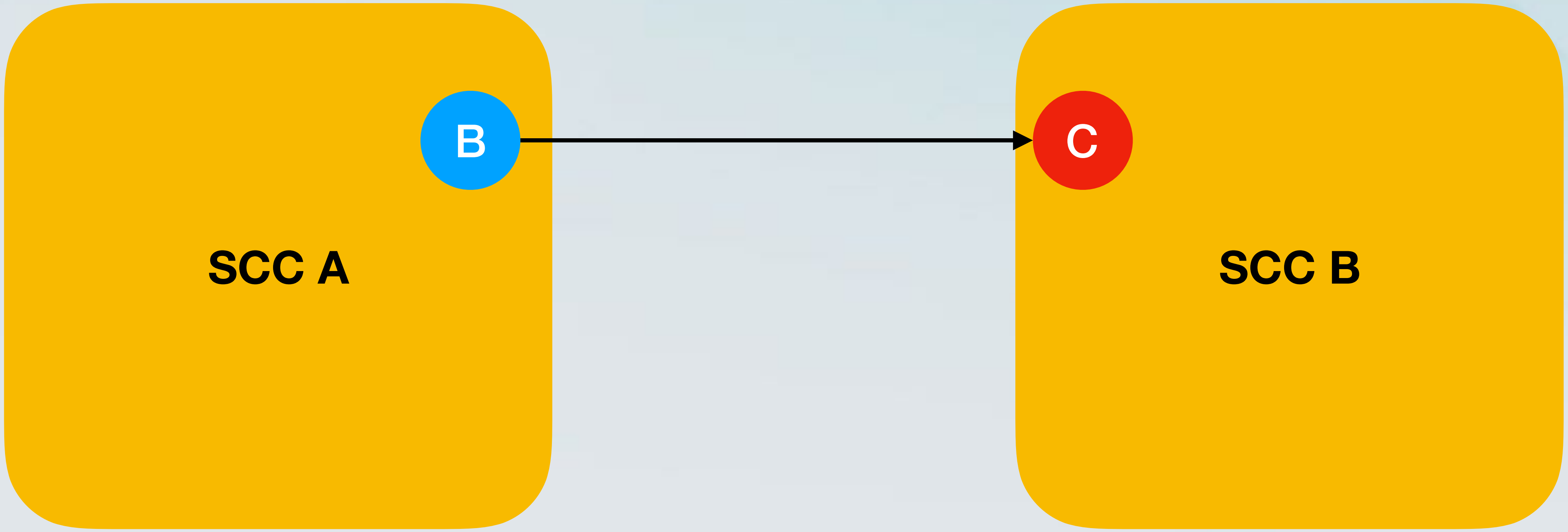
There is a path A-B
because of same SCC

There is a path C-D
because of same SCC

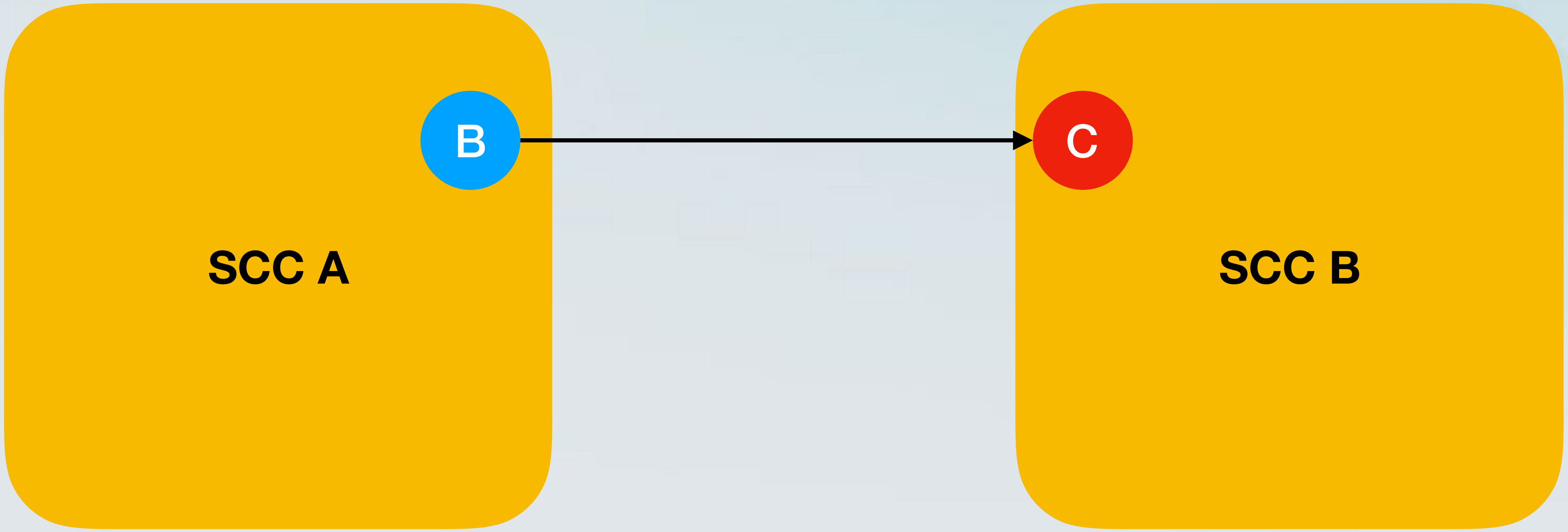
There is path A-D

DFS will explore this
path and A will finish last.

Proof by picture

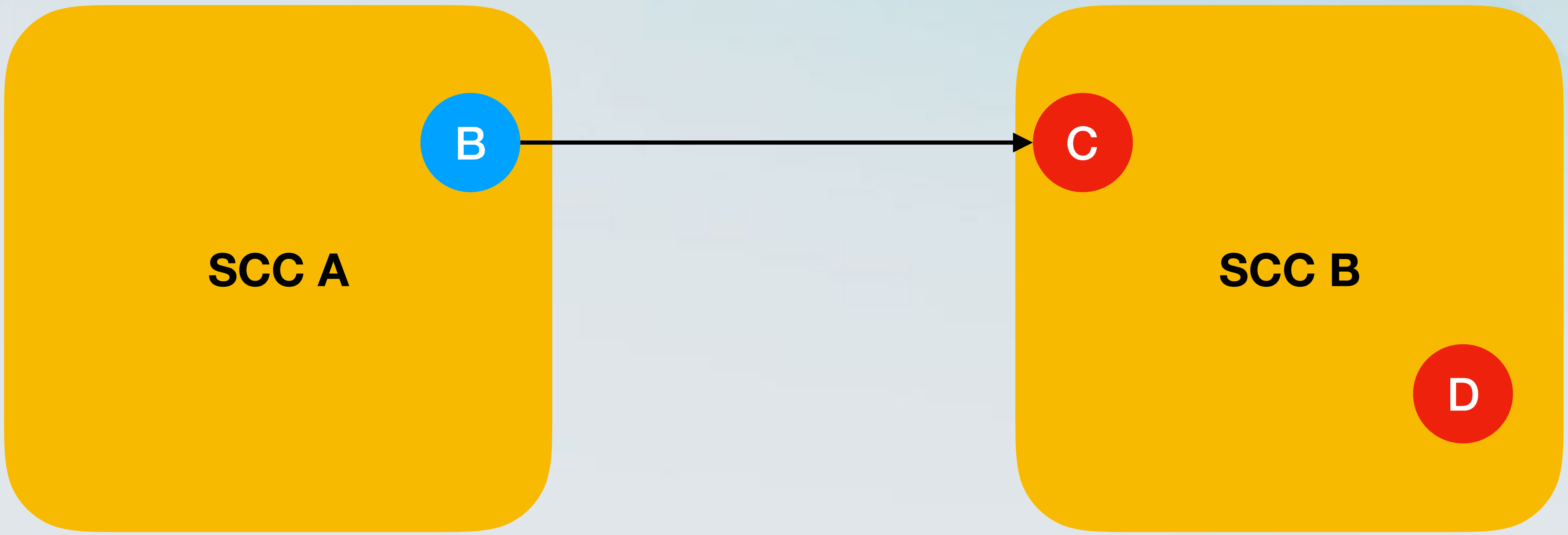


Proof by picture



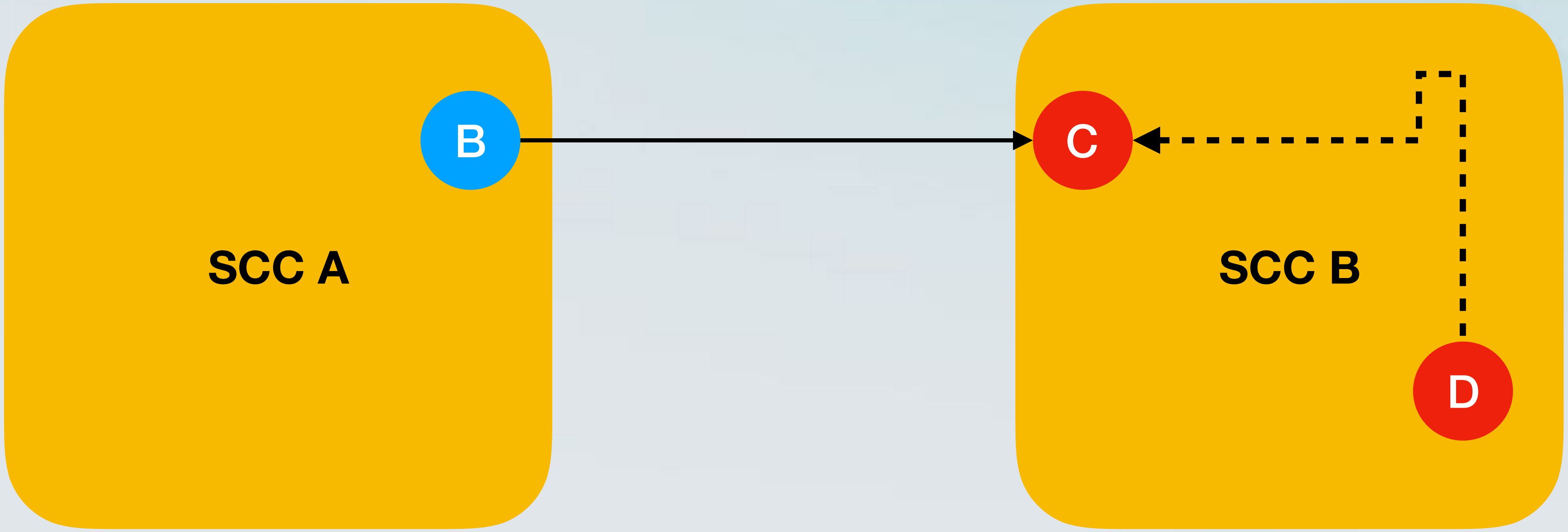
Case 2: We start here

Proof by picture



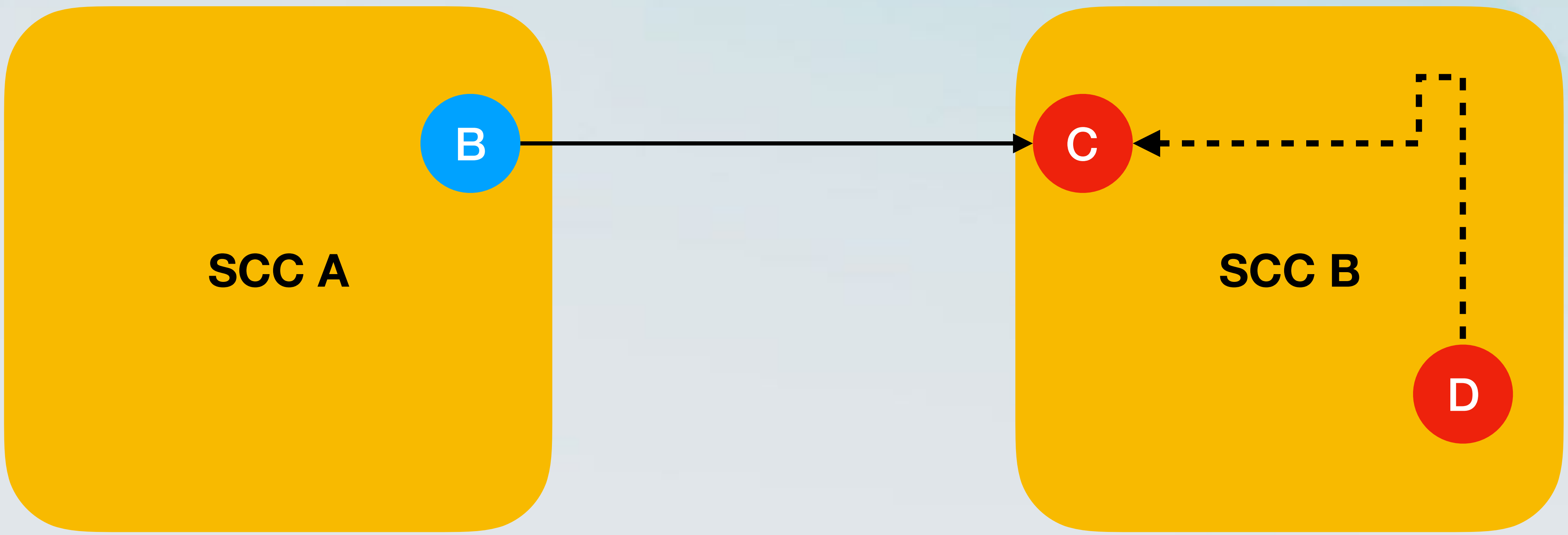
Case 2: We start here

Proof by picture



Case 2: We start here

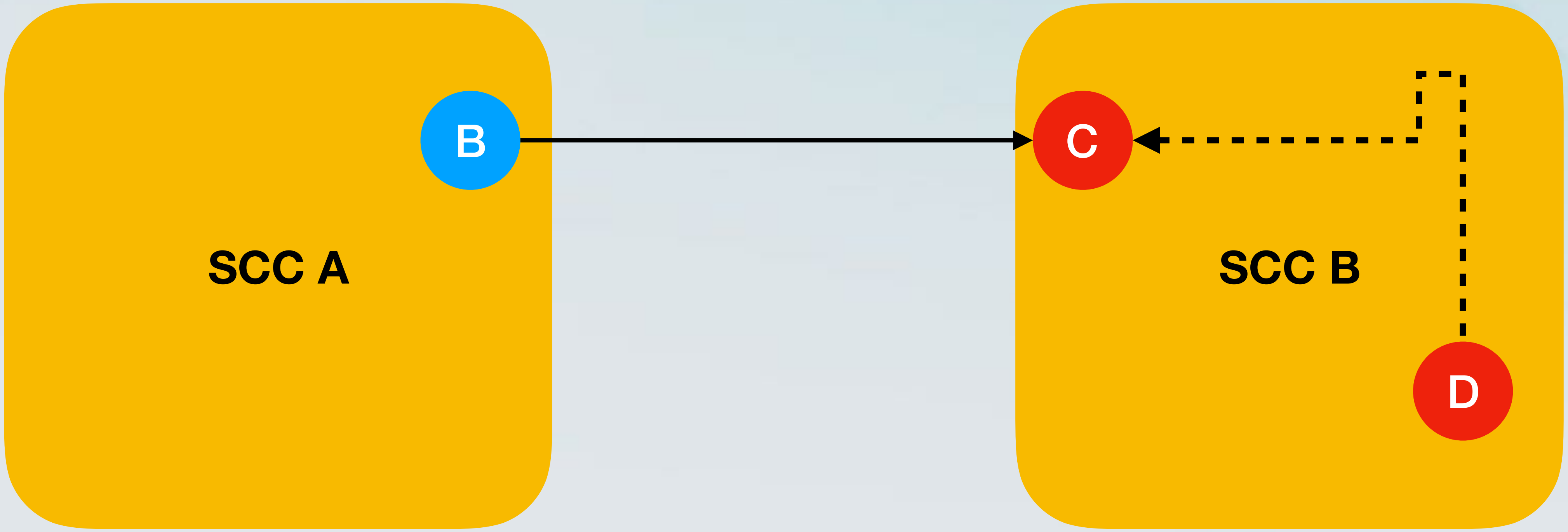
Proof by picture



Case 2: We start here

There is a path C-D because of same SCC

Proof by picture

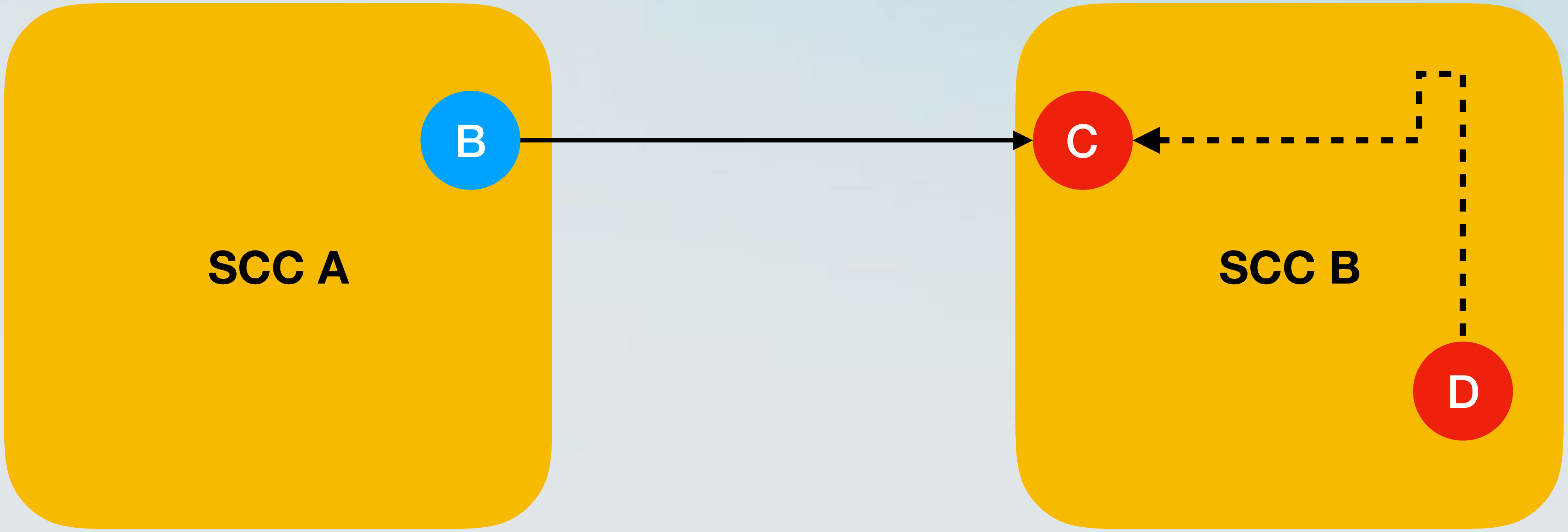


There is no path from SCC B to SCC A, by the simple-but-key lemma.

Case 2: We start here

There is a path C-D because of same SCC

Proof by picture



There is no path from SCC B to SCC A, by the simple-but-key lemma.

Case 2: We start here

There is a path C-D because of same SCC

DFS will finish with SCC B before it moves to SCC A.

Lemma and Corollary

- Lemma: Let C and C' be distinct SCCs in G. Suppose there is a directed edge ***crossing*** C and C'. Then the DFS on the nodes of C finishes later than the DFS on the nodes of C'.

- Corollary: If the forward DFS finishes on component C later than component C', then

  - there is no edge crossing from C' to C in G.

  - there is no edge crossing from C to C' in G^{rev}.

- This means that in the backward DFS on G^{rev}, if we start with the SCC that finishes last in the forward DFS of G, we will not find edges to other SCCs.

Intuition

Intuition

DFS finished here first in the forward pass.

Intuition

SCC A

DFS finished here first in the forward pass.

SCC B

This means there are no edges from SCC A to SCC B in the reverse graph.

Intuition

SCC A

DFS finished here first in the forward pass.

DFS will finish here before it moves to any other SCC.

SCC B

This means there are no edges from SCC A to SCC B in the reverse graph.

Intuition

SCC A

SCC B

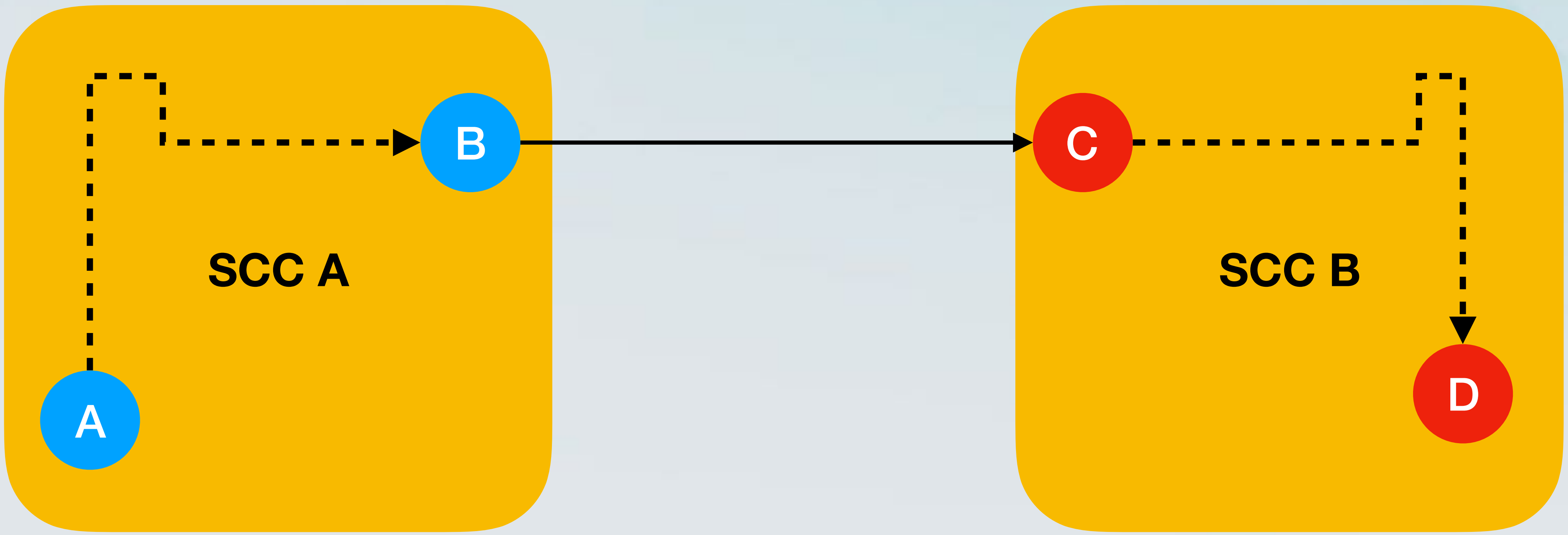
DFS finished here first in the forward pass.

This means there are no edges from SCC A to SCC B in the reverse graph.

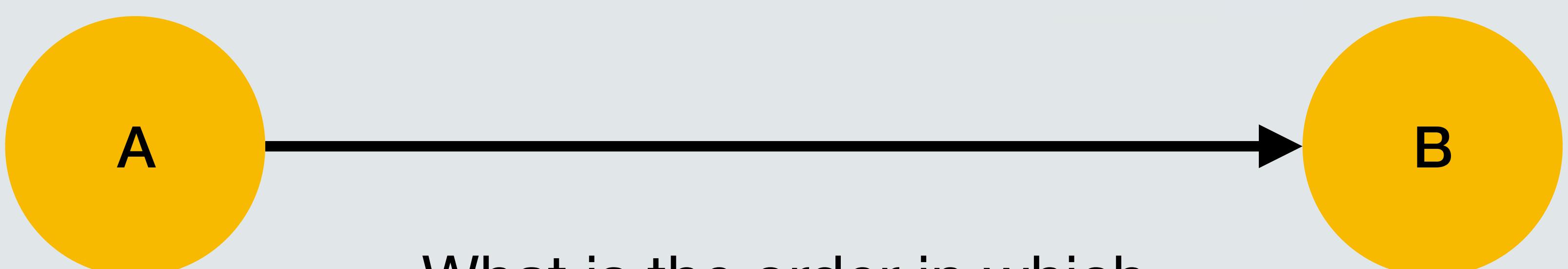
DFS will finish here before it moves to any other SCC.

SCC A will be correctly indentified.

Back to the component graph



This finishes first



What is the order in which the backward DFS visits the nodes?

Back to the component graph

Back to the component graph

- The backward DFS visits the nodes of G^{SCC} in topological order.

Back to the component graph

- The backward DFS visits the nodes of G^{SCC} in topological order.

- Alternative viewpoint:

Back to the component graph

- The backward DFS visits the nodes of G^{SCC} in topological order.
- Alternative viewpoint:
  - Produce a topological order of G^{SCC}.

Back to the component graph

- The backward DFS visits the nodes of G^{SCC} in topological order.

- Alternative viewpoint:

  - Produce a topological order of G^{SCC}.

  - Run a DFS on G^{rev} considering SCCs according to that topological order.