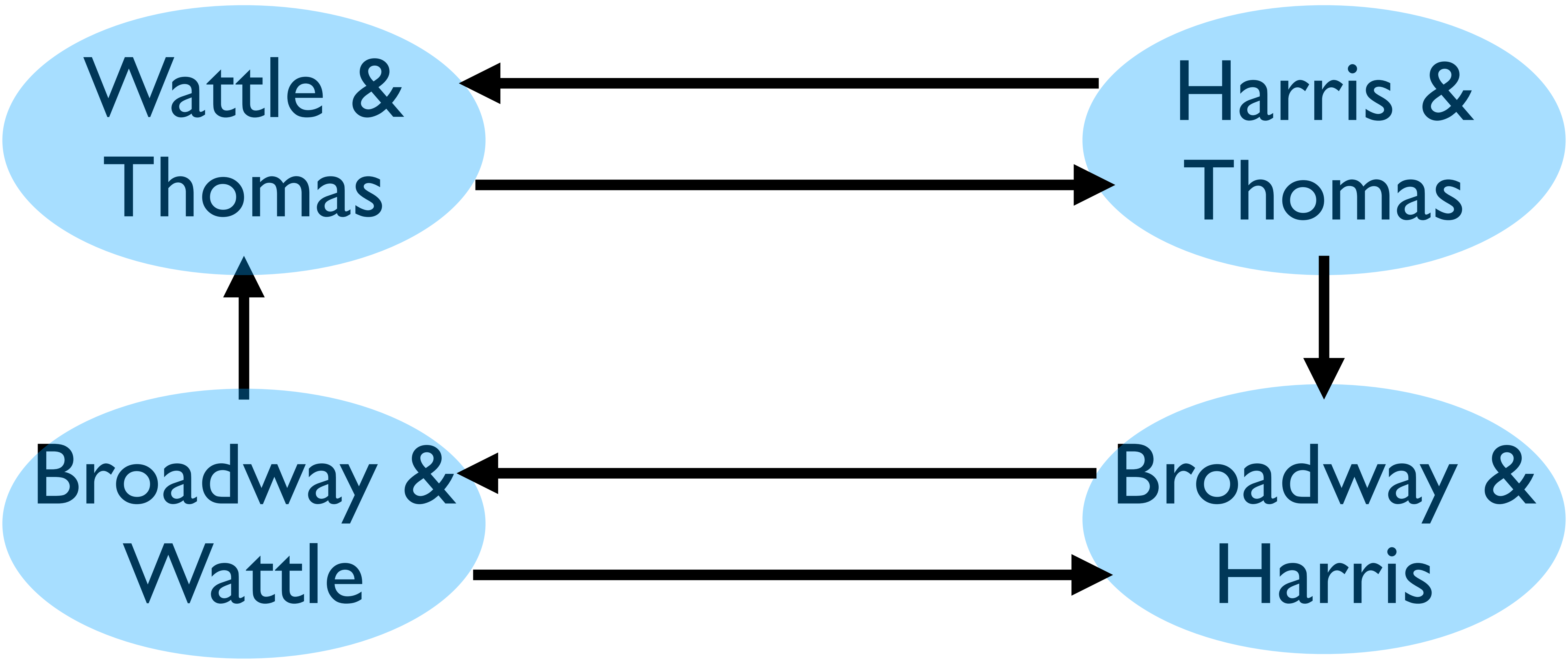
Topological Sort and DAGs

Directed Graphs

We have mostly talked about undirected graphs so far. Today will be all about directed graphs.

Directed edges naturally arise in many applications:

Graph of a road network: one way streets mean we need directed edges

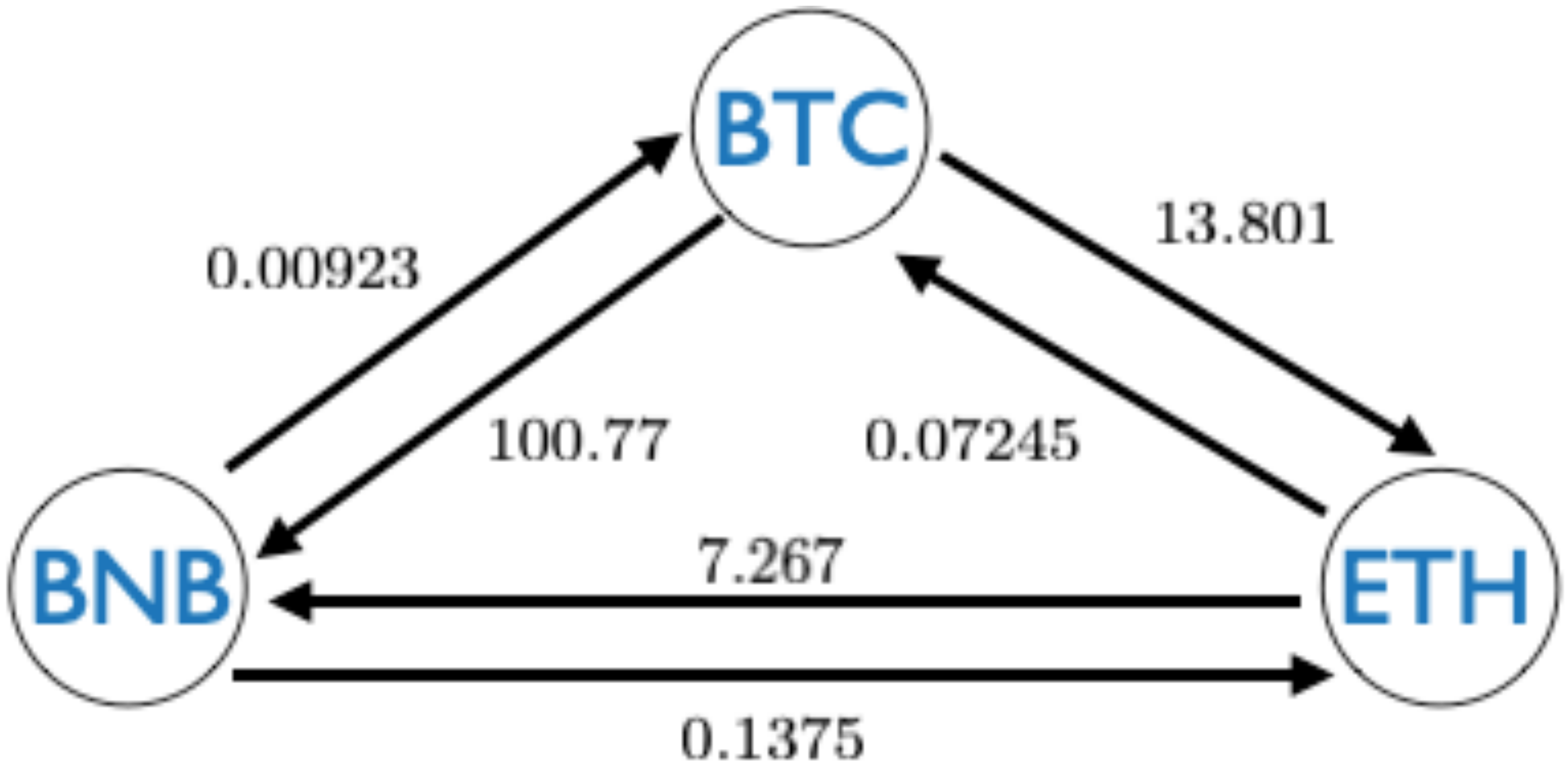


Directed Graphs

We have mostly talked about undirected graphs so far. Today will be all about directed graphs.

Directed edges naturally arise in many applications:

Graph of exchange rates:

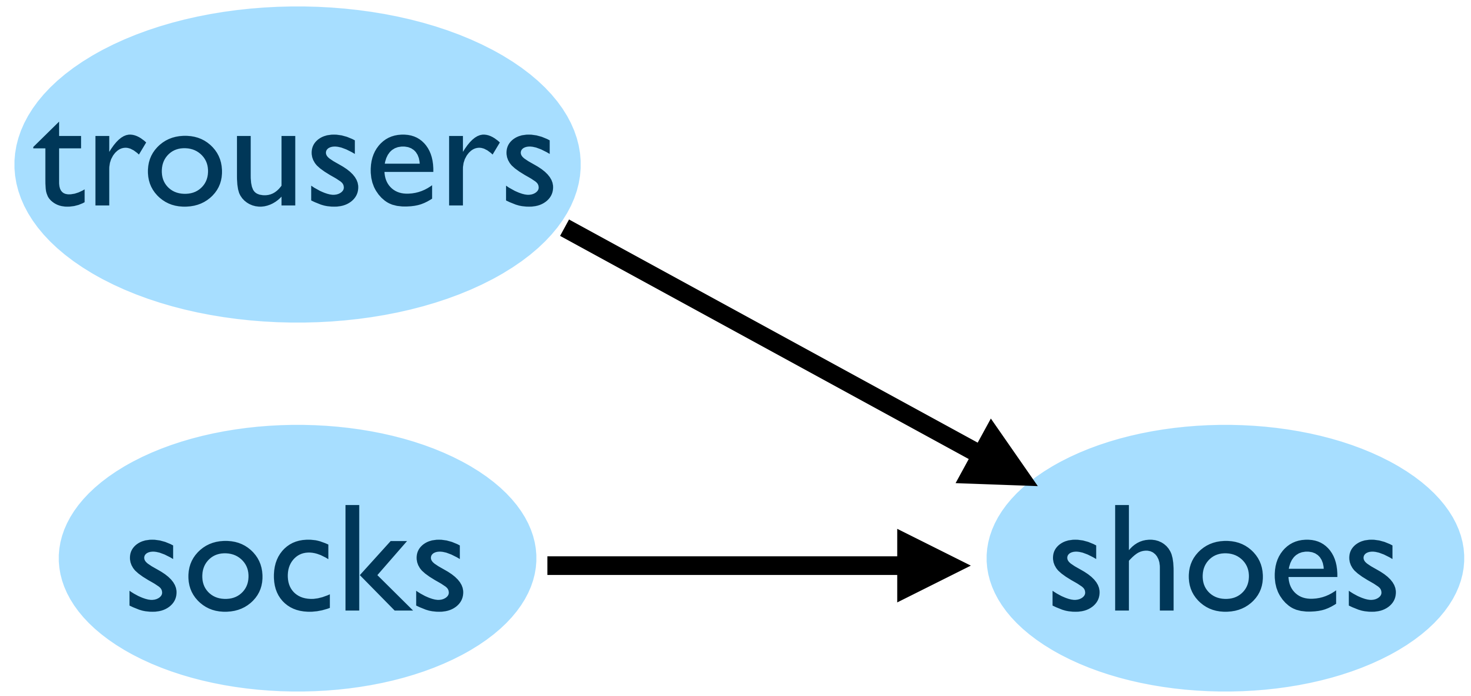


Directed Graphs

The first application we discuss today is job scheduling.

Imagine a graph where the vertices are labeled by tasks that need to be done.

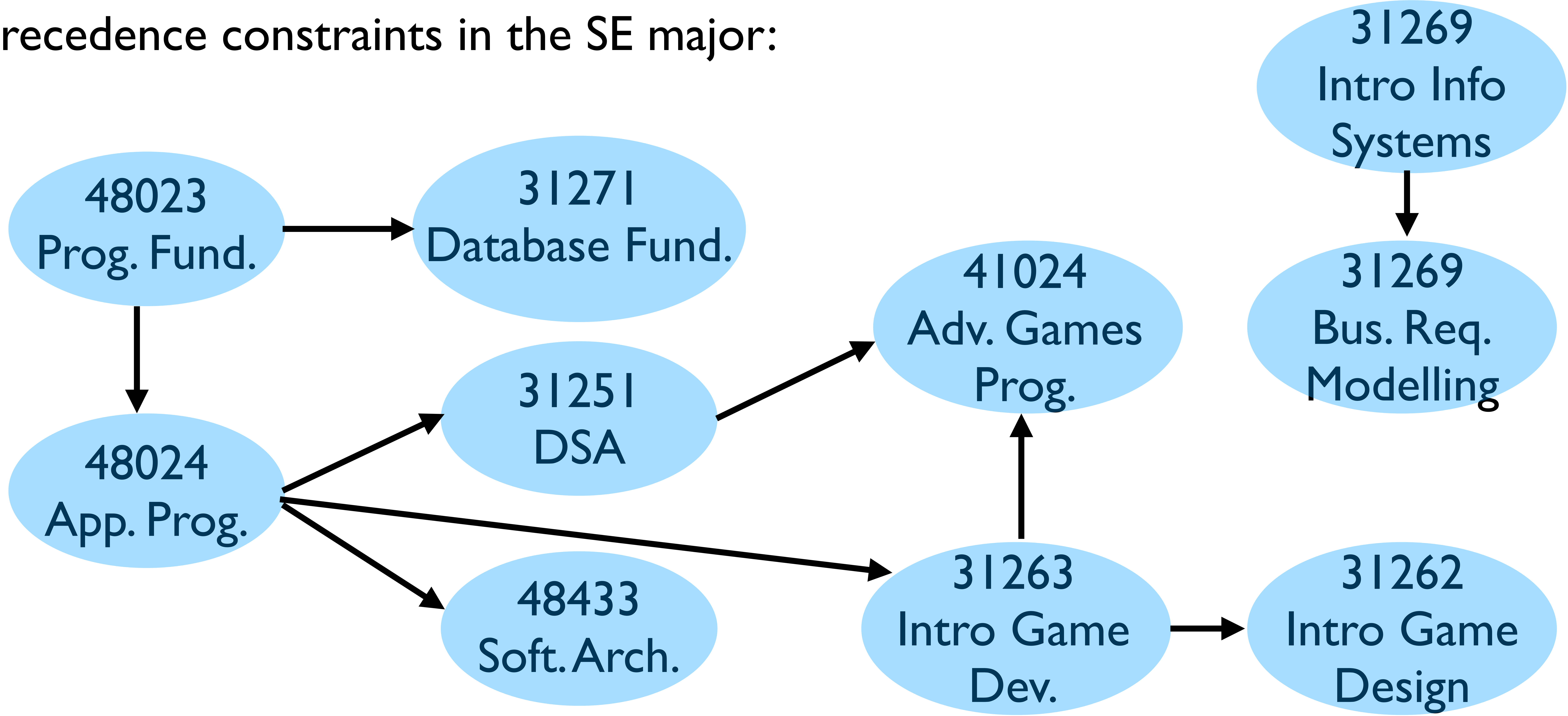
If task A must be performed before task B then we can put a directed edge from A to B.



Such a graph represents precedence constraints.

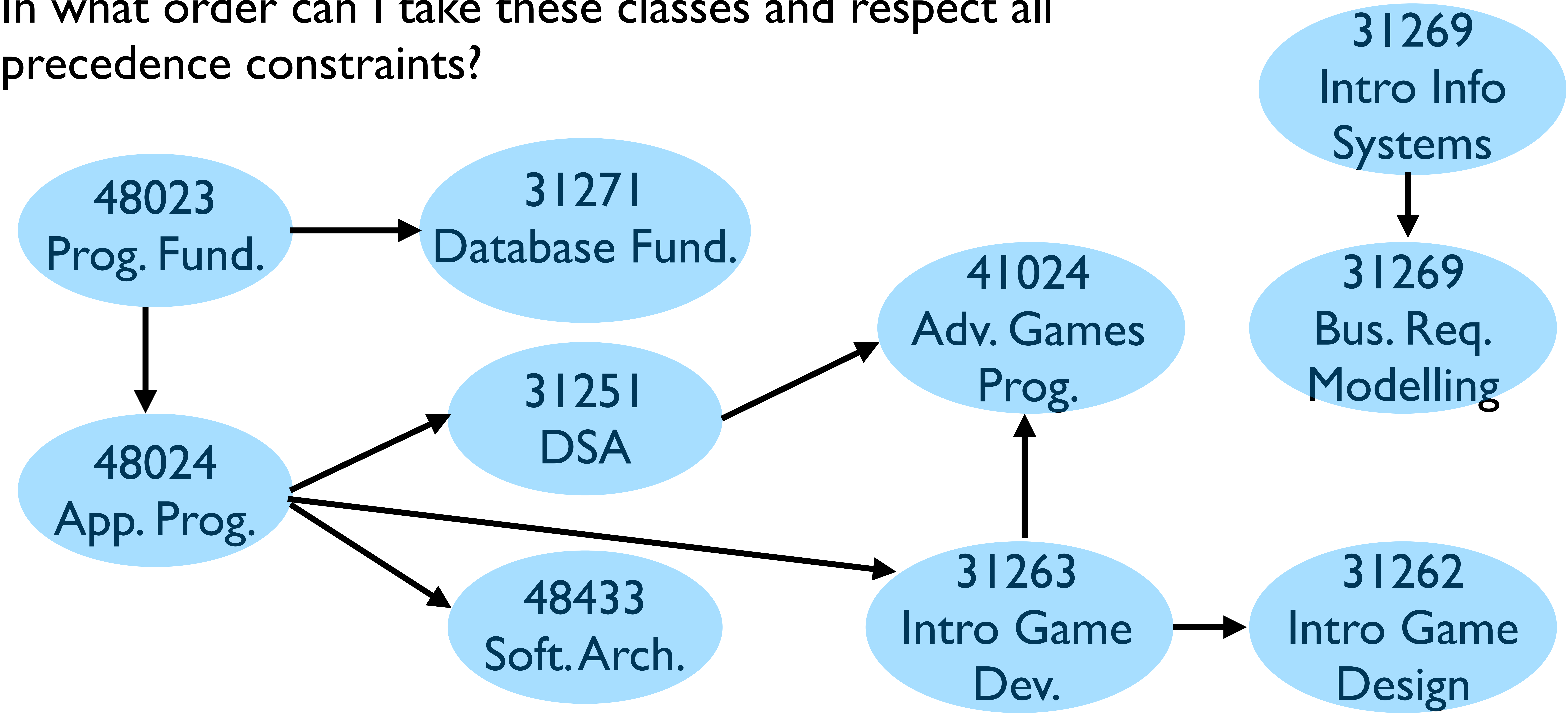
Software Engineering

Precedence constraints in the SE major:



What is a path to graduation?

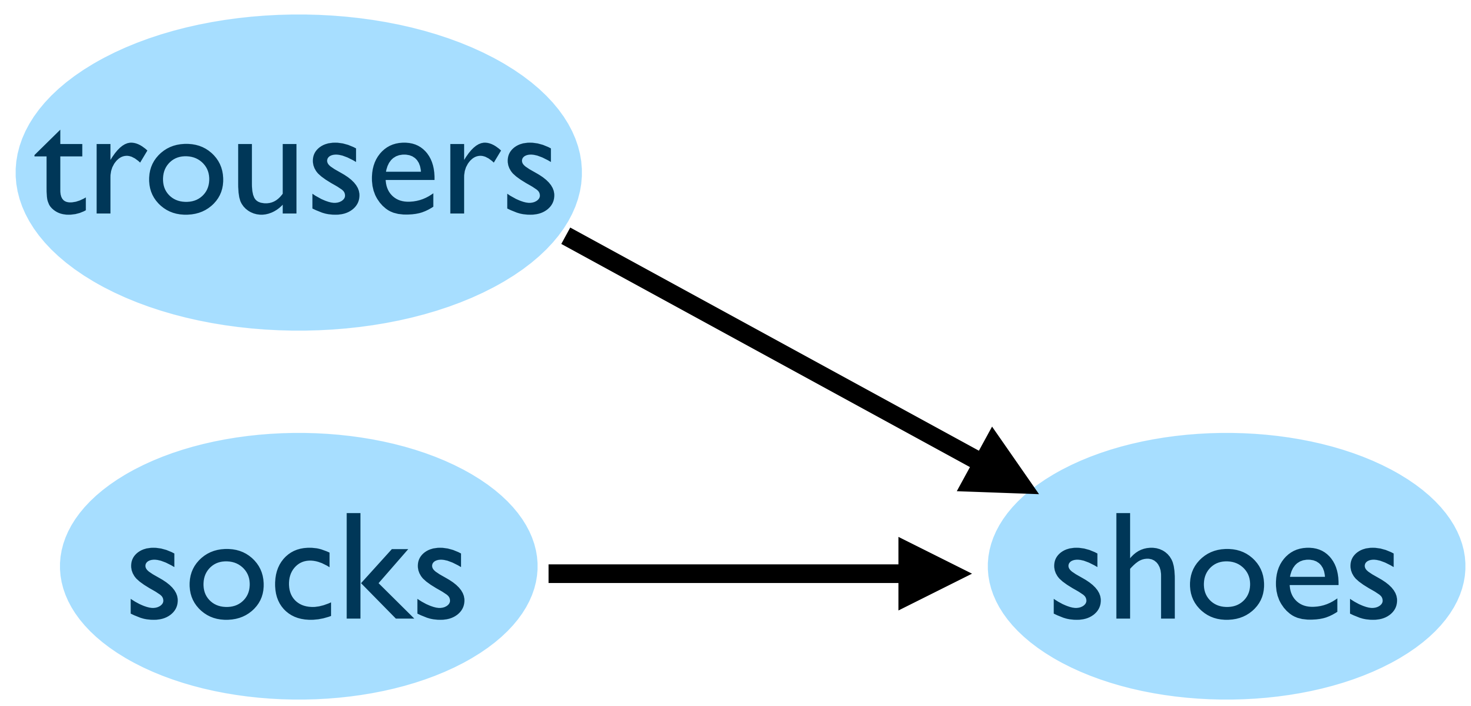
In what order can I take these classes and respect all precedence constraints?

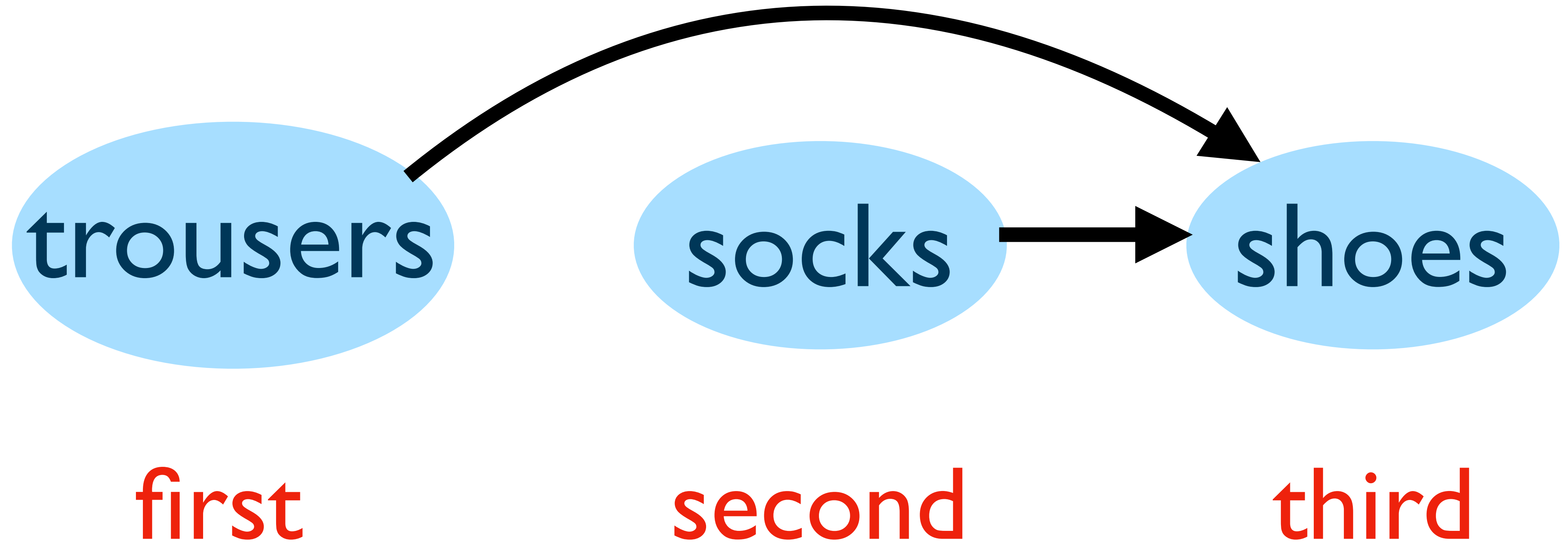


Topological Order

A path to graduation is given by a topological order.

A topological order of the vertices is a ordering of all the vertices such that if there is an edge (u, v) in the graph then u must come before v in the ordering.

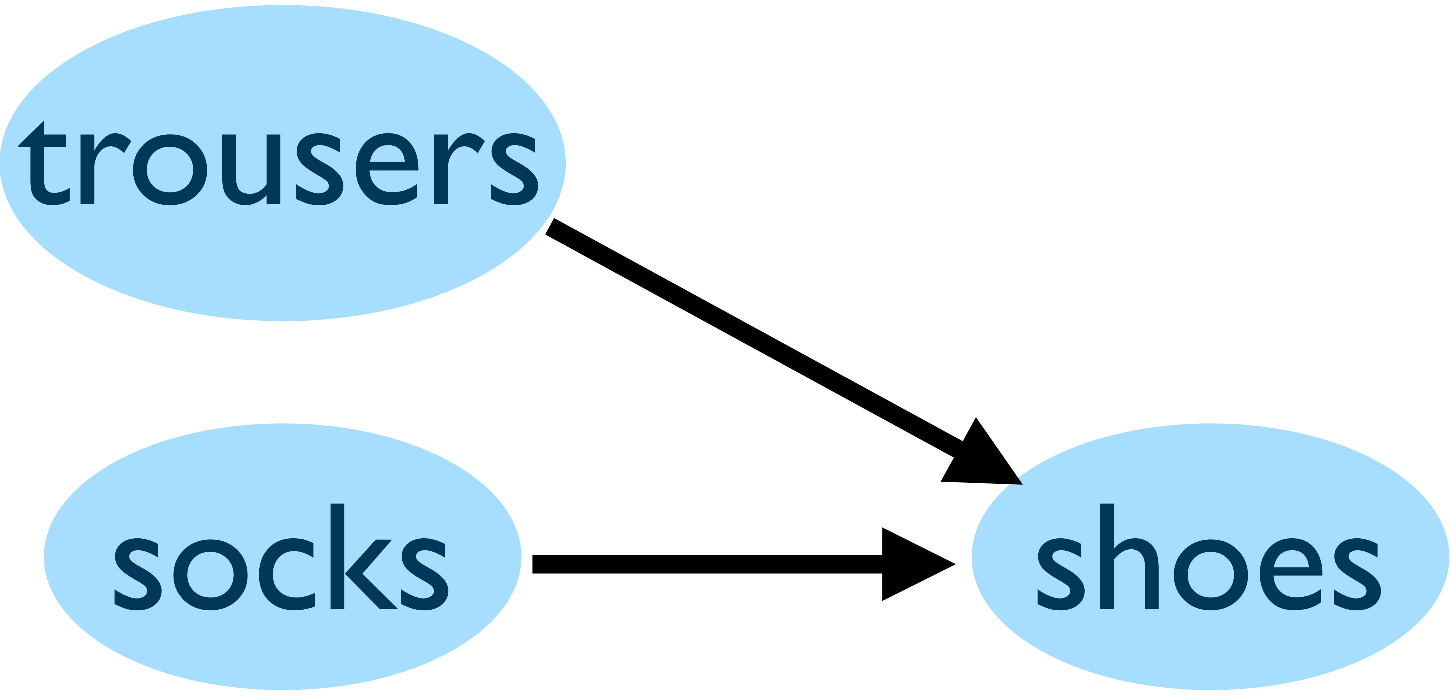


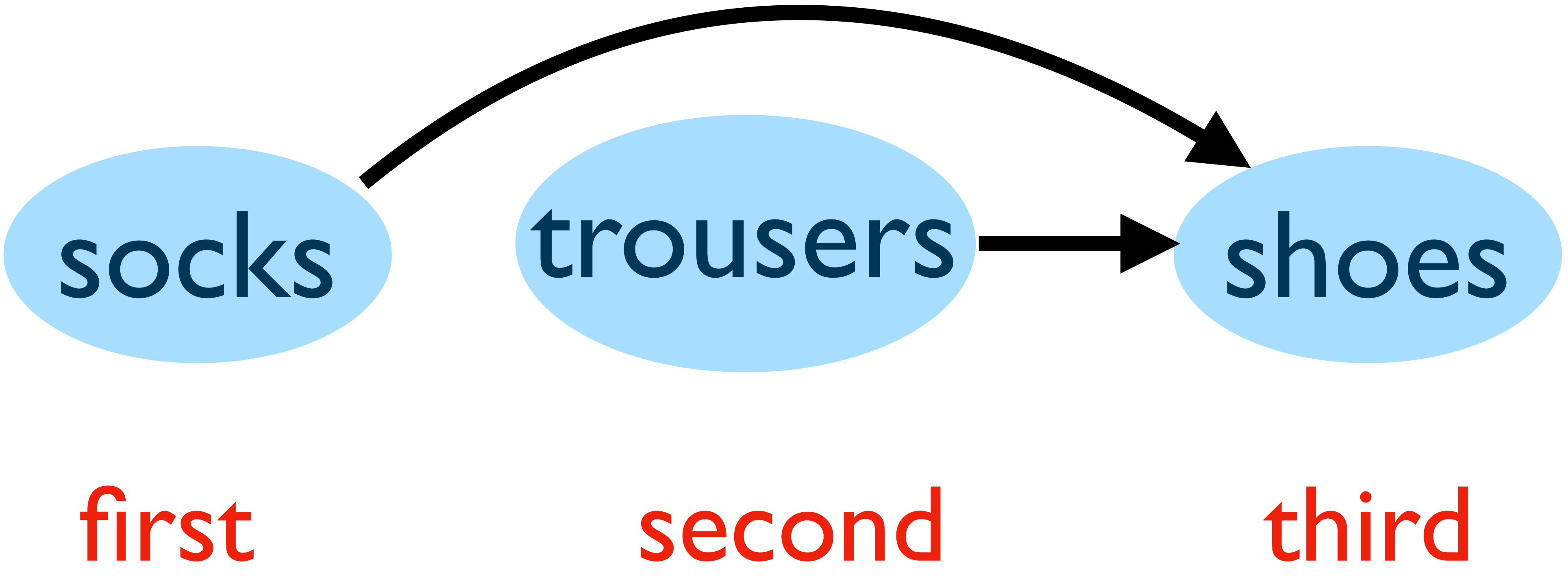


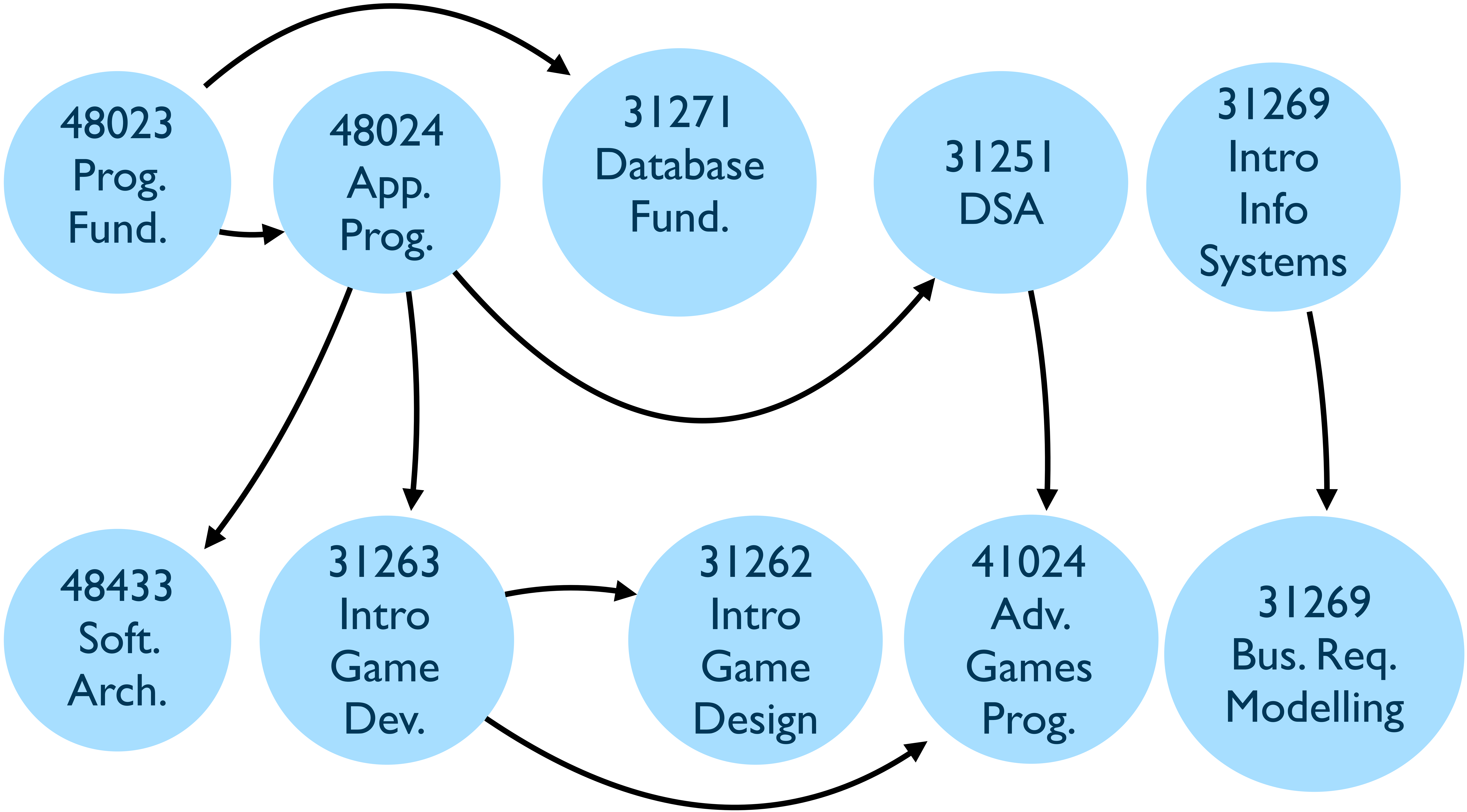
Topological Order

A topological order of the vertices is a linear order where if there is an edge (u, v) in the graph then u must come before v in the ordering.

There can be more than one topological ordering of the vertices.






48023 Prog. Fund.
48024 App. Prog.
31271 Database Fund.
31251 DSA
31269 Intro Info Systems
48433 Soft. Arch.
31263 Intro Game Dev.
31262 Intro Game Design
41024 Adv. Games Prog.
31269 Bus. Req. Modelling

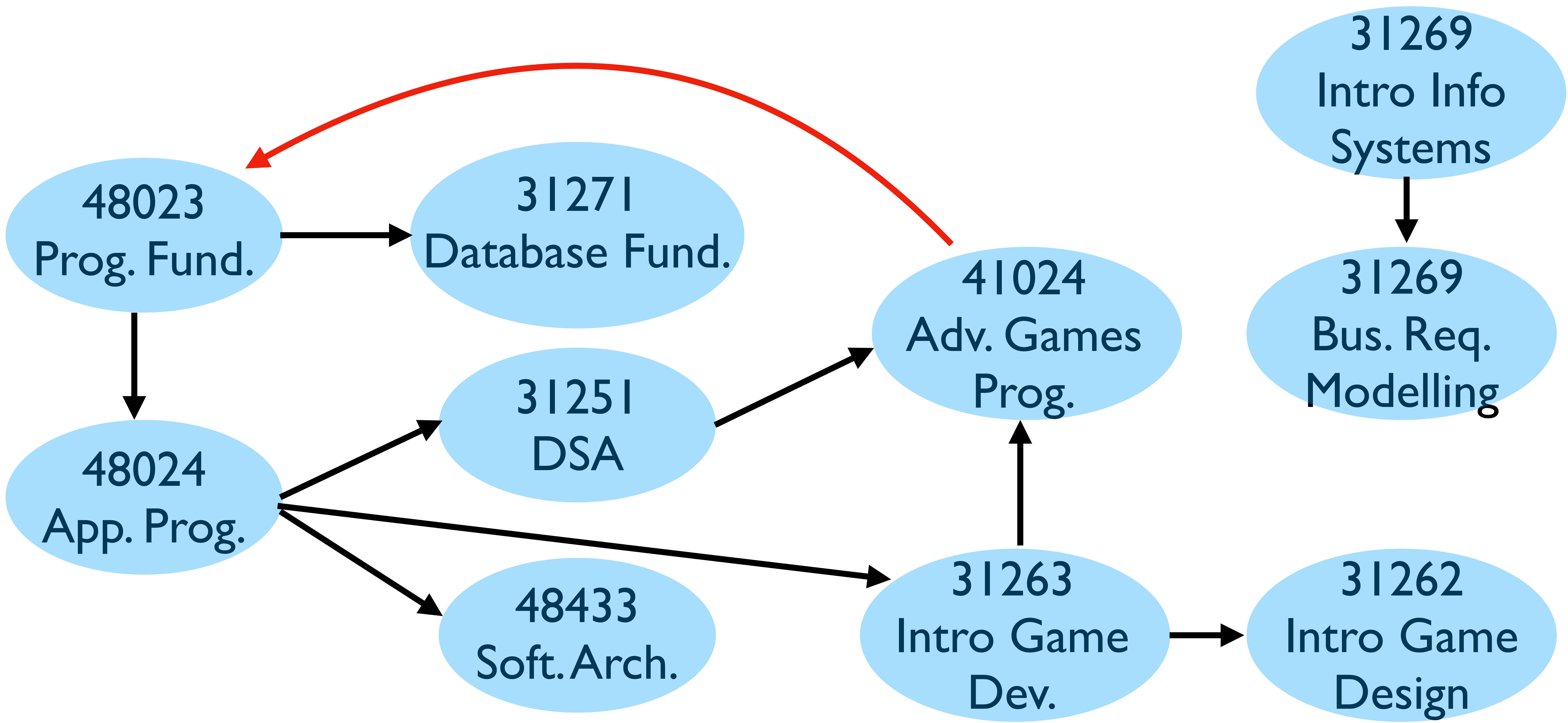
Topological Order

When does a topological order exist?

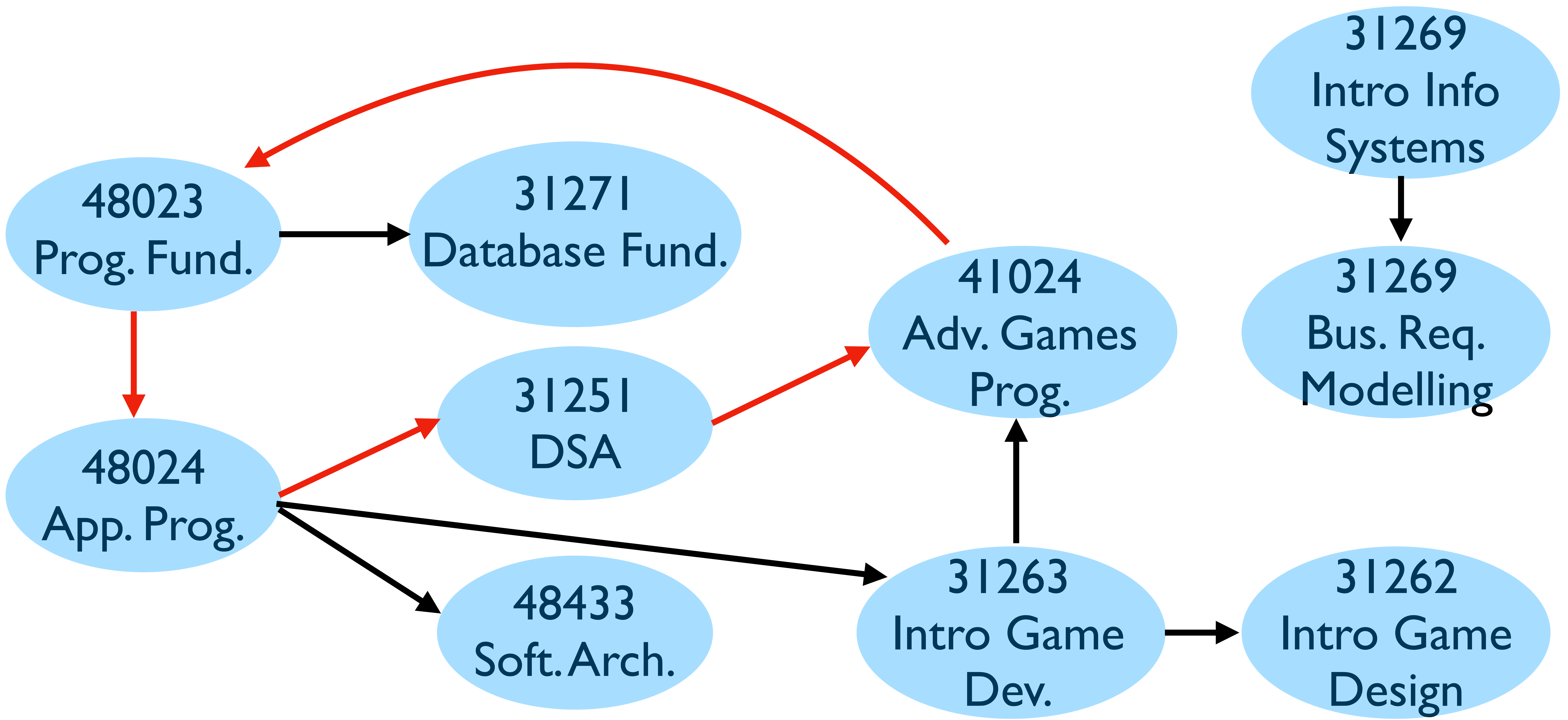
Can it be given for any directed graph?

What could be a possible obstruction to giving a topological order?

Imagine that Advanced Game Programming was a prerequisite for Programming Fundamentals



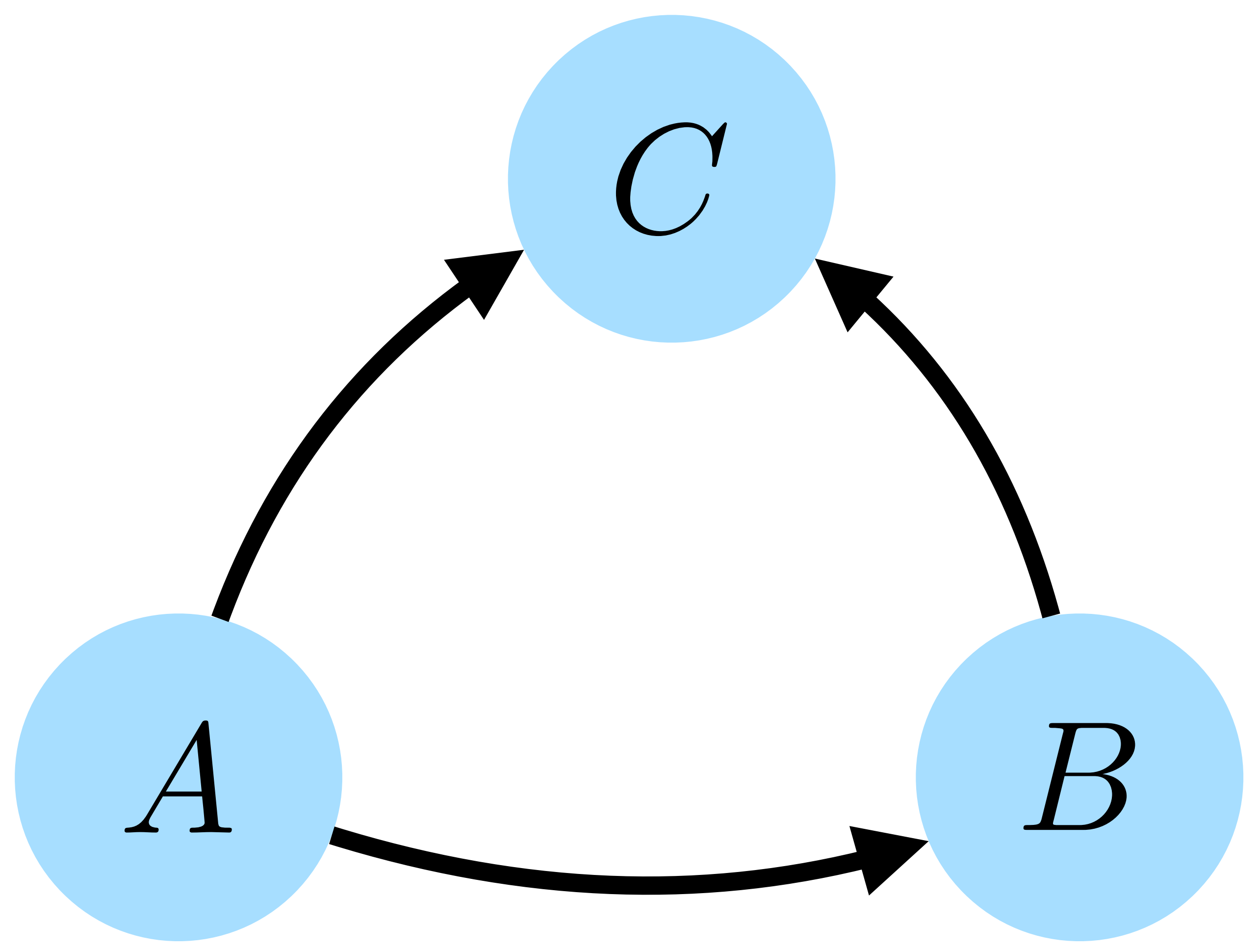
Now there is no way to graduate!



Directed Acyclic Graph (DAG)

A graph with a directed cycle cannot have a topological order.

A directed graph without any directed cycles is called a directed acyclic graph (DAG).



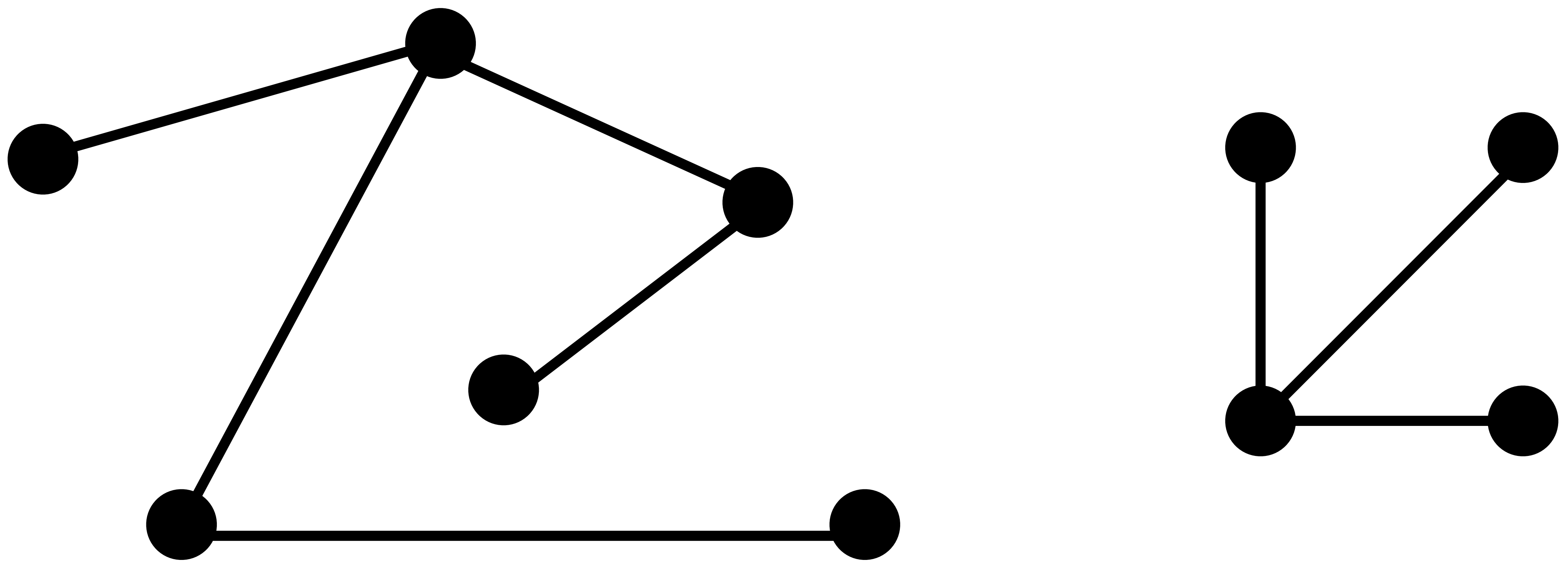
This is a DAG.

A directed cycle has to respect the orientation of edges.

Undirected vs. Directed

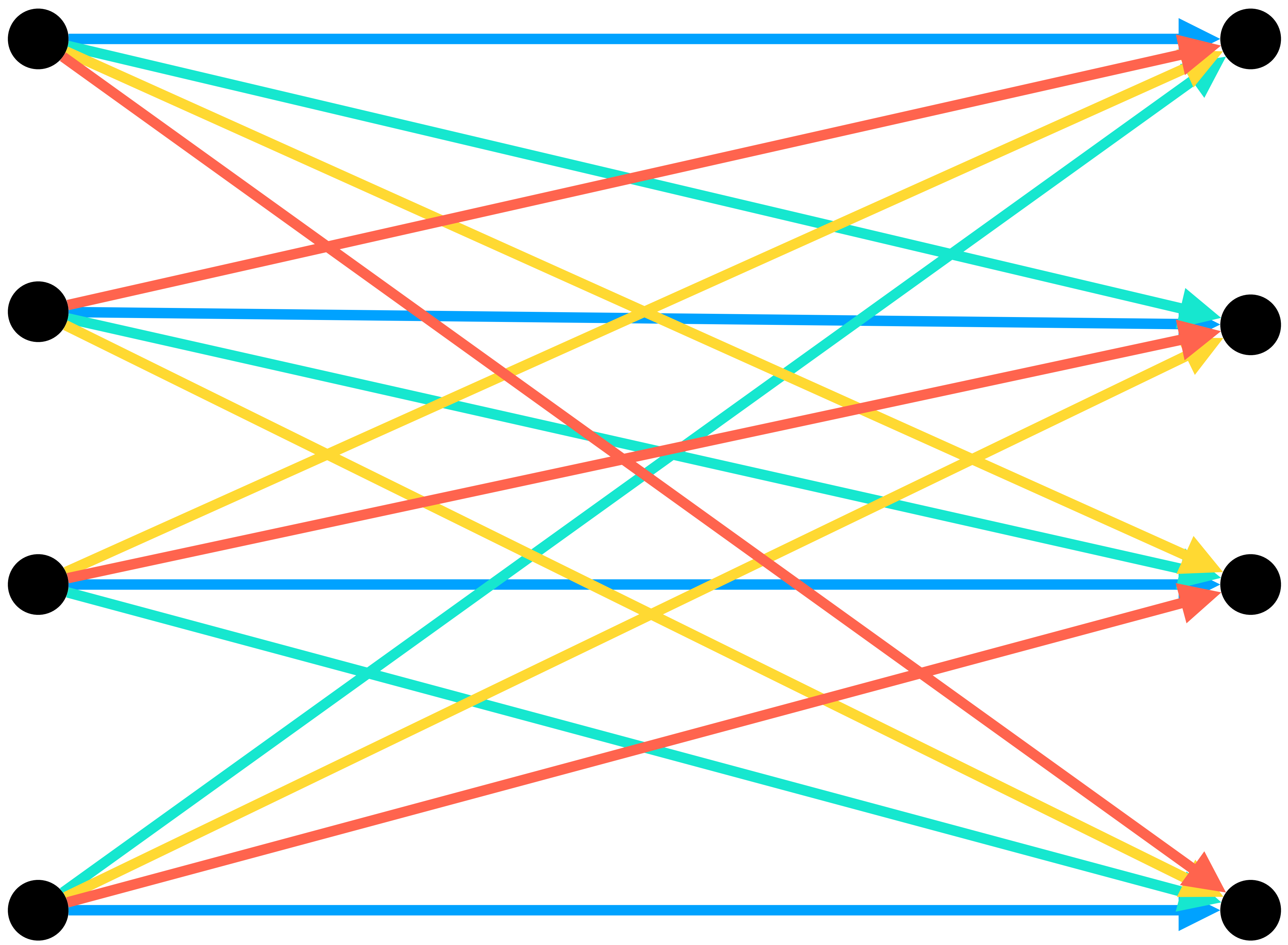
An undirected graph without a cycle is a forest, a disjoint union of trees.

It will always have at most $n-1$ edges.



Undirected vs. Directed

A DAG can have lots of edges.



There is a directed edge from every vertex on the left to every vertex on the right.

This kind of graph on n vertices has $n^2/4$ edges.

A DAG has a topological order

It turns out a directed cycle is the only obstruction to a graph having a topological order.

Fact: A directed graph has a topological order if and only if it is a DAG.

We have already seen that a graph with a directed cycle has no topological order.

We are going to see that a DAG has a topological order by giving an algorithm to find it!

Is a graph a DAG?

Fact: A directed graph has a topological order if and only if it is a DAG.

We begin with an algorithm to determine whether or not an input graph has a directed cycle.

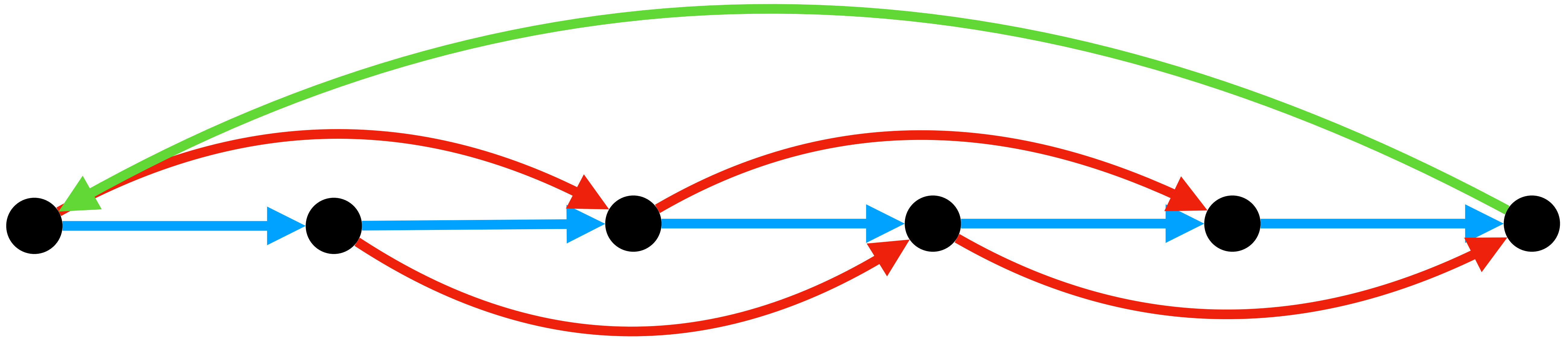
That is, we determine whether or not the graph is a DAG.

This problem is intimately related to finding a topological order.

Is a graph a DAG?

In case the graph is not a DAG, the algorithm will certify this by outputting a directed cycle.

We cannot efficiently output all the directed cycles in a graph, as there can be exponentially many.



The number of cycles in this graph is related to the Fibonacci numbers.

Detecting a Directed Cycle

Depth-First Search

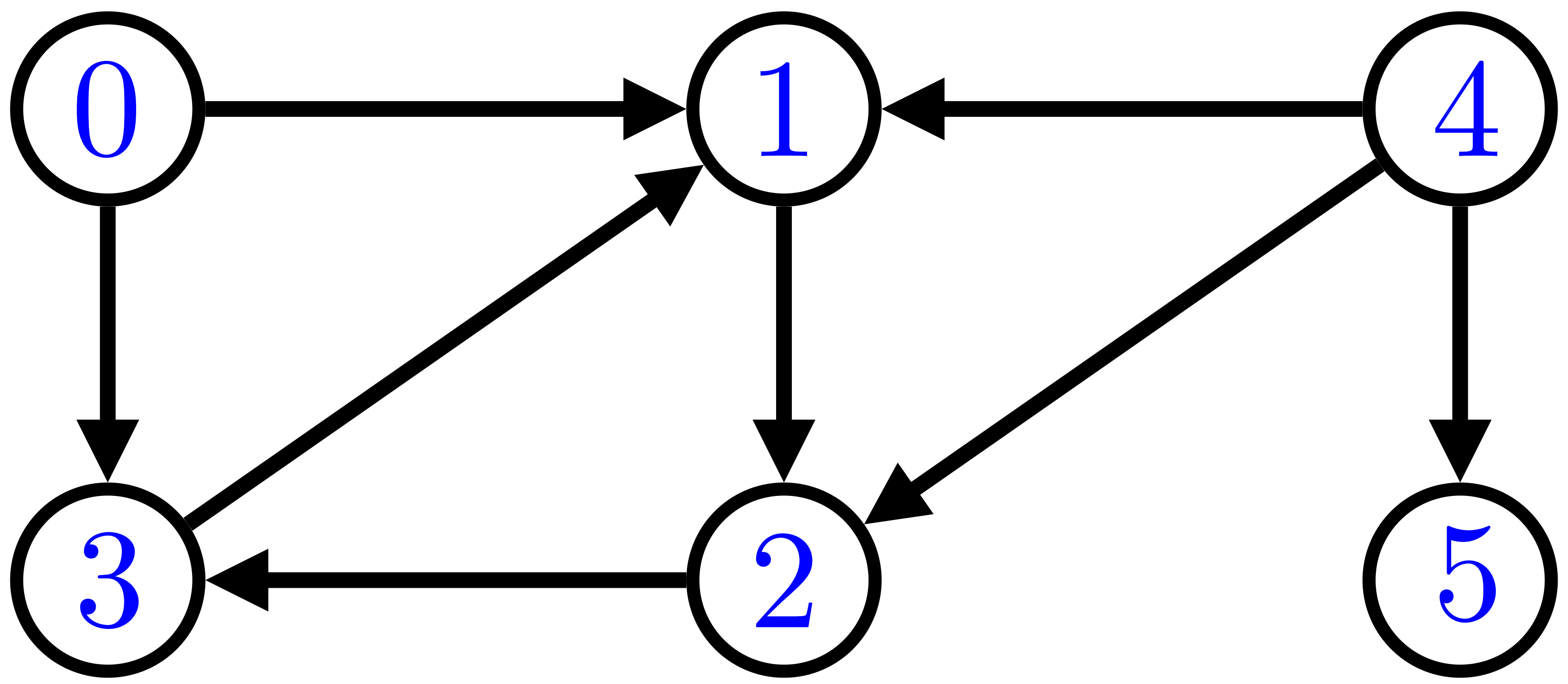
The algorithm we will use to detect a directed cycle is based on depth-first search.

We talked before about DFS in the context of undirected graphs.

Let's now look at an example in a directed graph.

To solve the cycle detection problem we are going to keep track of when we start and finish exploring a vertex.

Depth-First Search



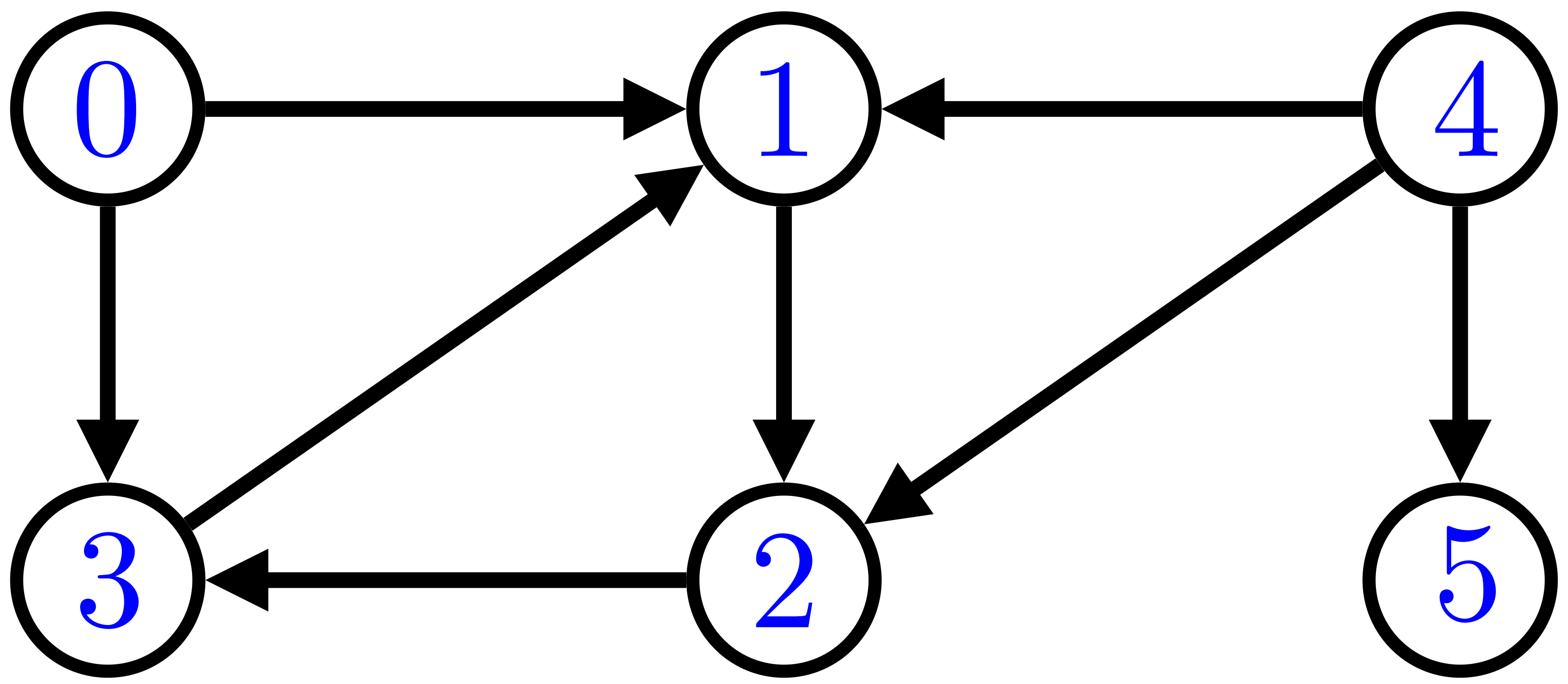
0: 1 3
1: 2
2: 3
3: 1
4: 5 2 1
5:

Adjacency List

```
bool marked[N] {};

void dfs()
{
    for(unsigned v = 0; v < N; ++v)
    {
        if(!marked[v])
        {
            dfs_visit(v);
        }
    }
}
```

Depth-First Search



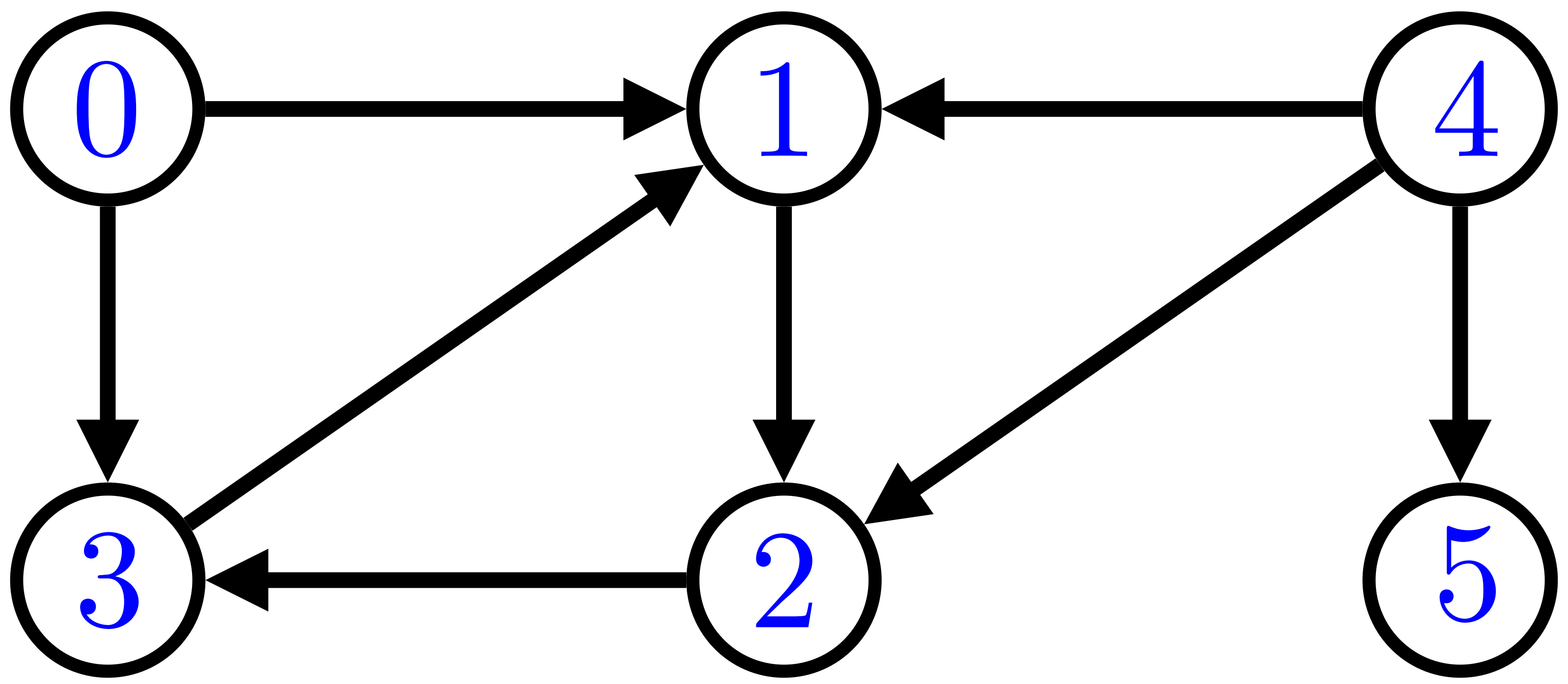
0: 1 3
1: 2
2: 3
3: 1
4: 5 2 1
5:

Adjacency List

```
bool marked[N] {};

void dfs_visit(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
          dfs_visit(u);
        }
    }
}
```

Depth-First Search



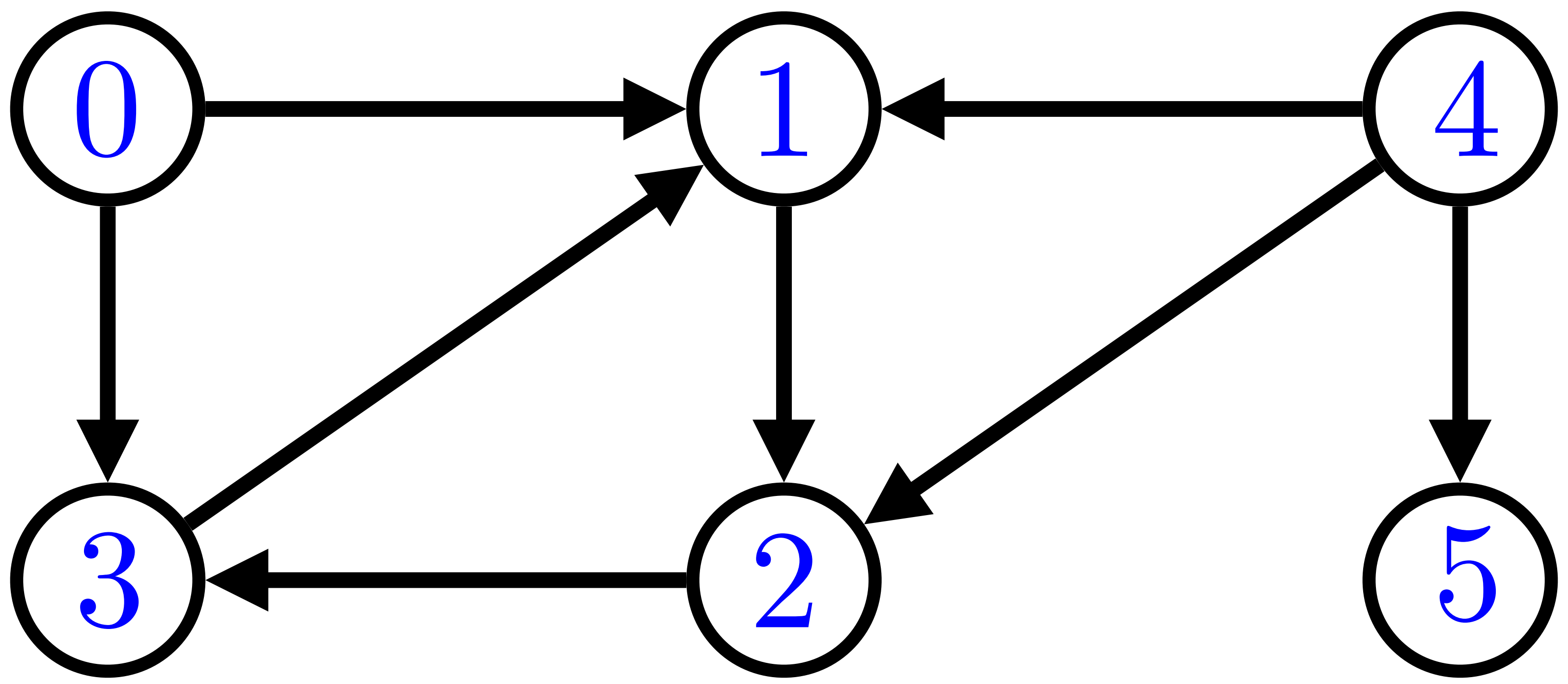

```
bool marked[N] {};

void dfs_visit(unsigned v)
{
     marked[v] = true;
     for(auto u : arr[v])
     {
          if(!marked[u])
          {
            dfs_visit(u);
          }
     }
}
```

Definition: Vertex u is reachable from vertex v if and only if there is a directed path from v to u.

Fact: $\texttt{dfs\_visit}(v)$ marks exactly those vertices u reachable from v.

Depth-First Search



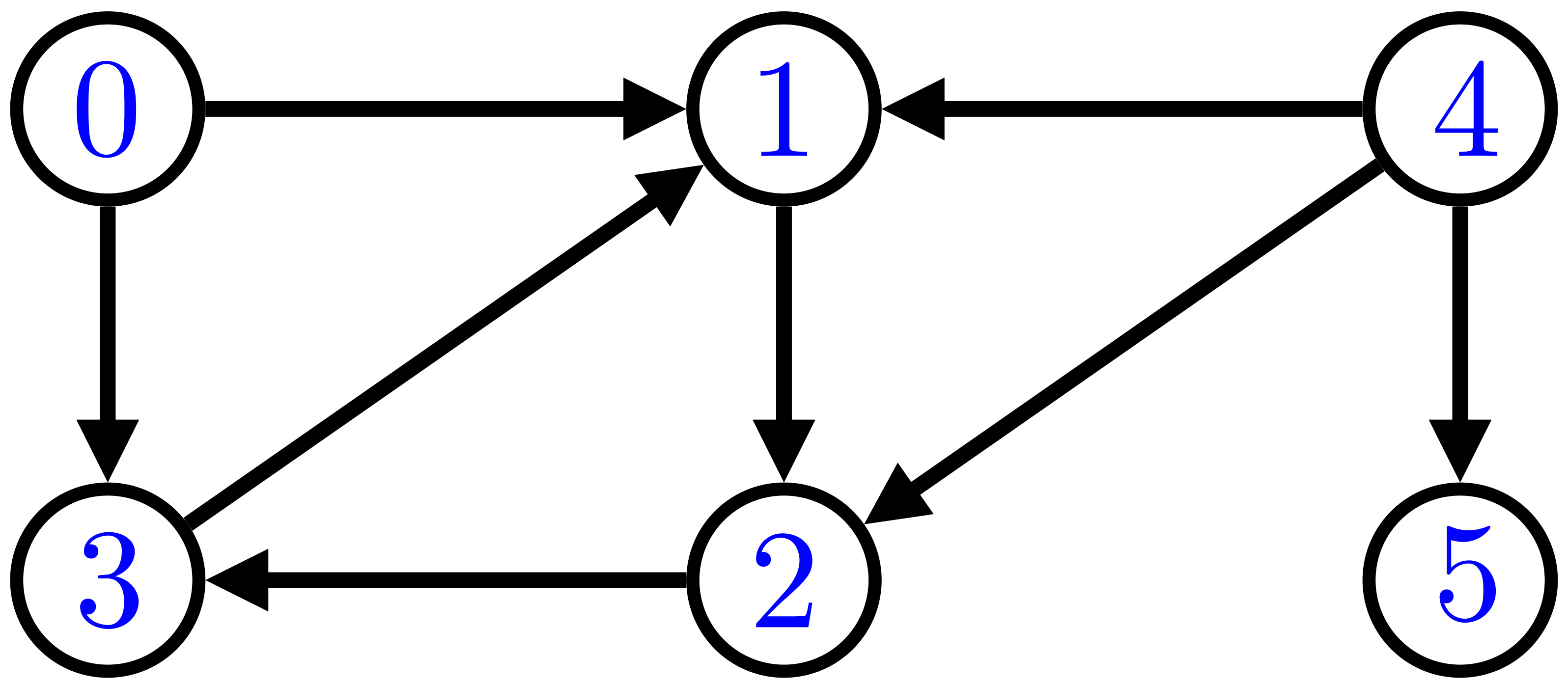
0: 1 3
1: 2
2: 3
3: 1
4: 5 2 1
5:

Adjacency List

```
bool marked[N] {};

void dfs_visit(unsigned v)
{
    marked[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
          dfs_visit(u);
        }
    }
}
```

Depth-First Search



0: 1 3
1: 2
2: 3
3: 1
4: 5 2 1
5:

Adjacency List

```
bool marked[N] {};
bool on_stack[N] {};
std::vector<int> edge_to(N,-1);

void dfs_visit(unsigned v)
{
    marked[v] = true;
    on_stack[v] = true;
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
          edge_to[u] = v;
          dfs_visit(u);
        }
    }
    on_stack[v] = false;
}
```

Depth-First Search

Here is the key claim to finding a cycle.

Claim: There is a cycle reachable from vertex v iff in $\texttt{dfs\_visit}(v)$ we find an edge to a vertex which is $\texttt{on\_stack}$.

Such an edge is called a back edge.

```
void dfs_visit(unsigned v) {
    marked[v] = true;
    on_stack[v] = true;
    for(auto u : arr[v]) {
        if(!marked[u]) {
          edge_to[u] = v;
          dfs_visit(u);
        }
        else if(on_stack[u]){
          // found a cycle!
          // process the cycle
        }
    }
    on_stack[v] = false;
}
```

https://godbolt.org/z/KrxP3Gxd8

Depth-First Search

Here is the key claim to finding a cycle.

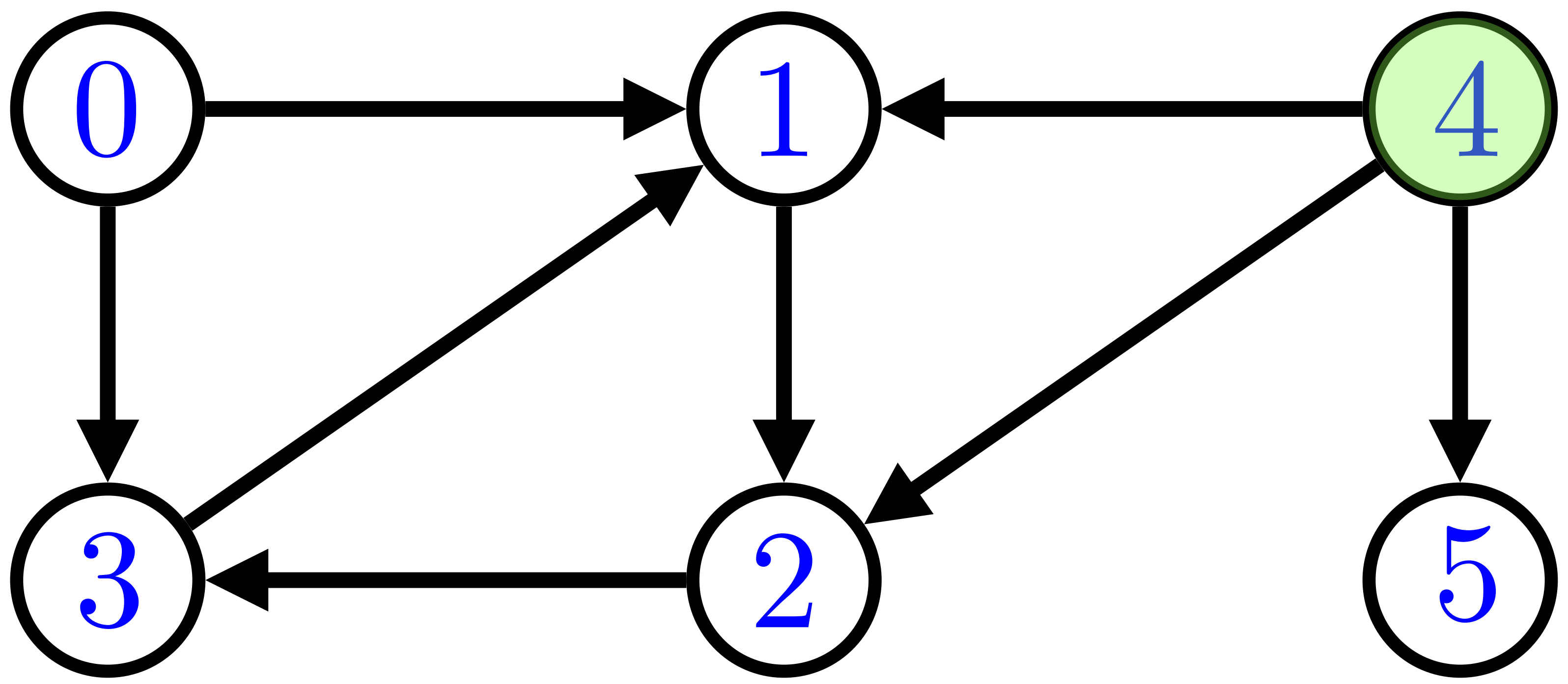
Claim: There is a cycle reachable from vertex v iff in $\texttt{dfs\_visit}(v)$ we find an edge to a vertex which is $\texttt{on\_stack}$.

If u is such a vertex, and we consider u from vertex v then the cycle is

```
void dfs_visit(unsigned v) {
    marked[v] = true;
    on_stack[v] = true;
    for(auto u : arr[v]) {
        if(!marked[u]) {
          edge_to[u] = v;
          dfs_visit(u);
        }
        else if(on_stack[u]){
          // found a cycle!
          // process the cycle
        }
    }
    on_stack[v] = false;
}
```

$$u \leftarrow v \leftarrow \texttt{edge\_to}[v] \leftarrow \texttt{edge\_to}[\texttt{edge\_to}[v]] \leftarrow \cdots \leftarrow u$$

dfs_visit(4)



0: 1 3
1: 2
2: 3
3: 1
4: 5 2 1
5:

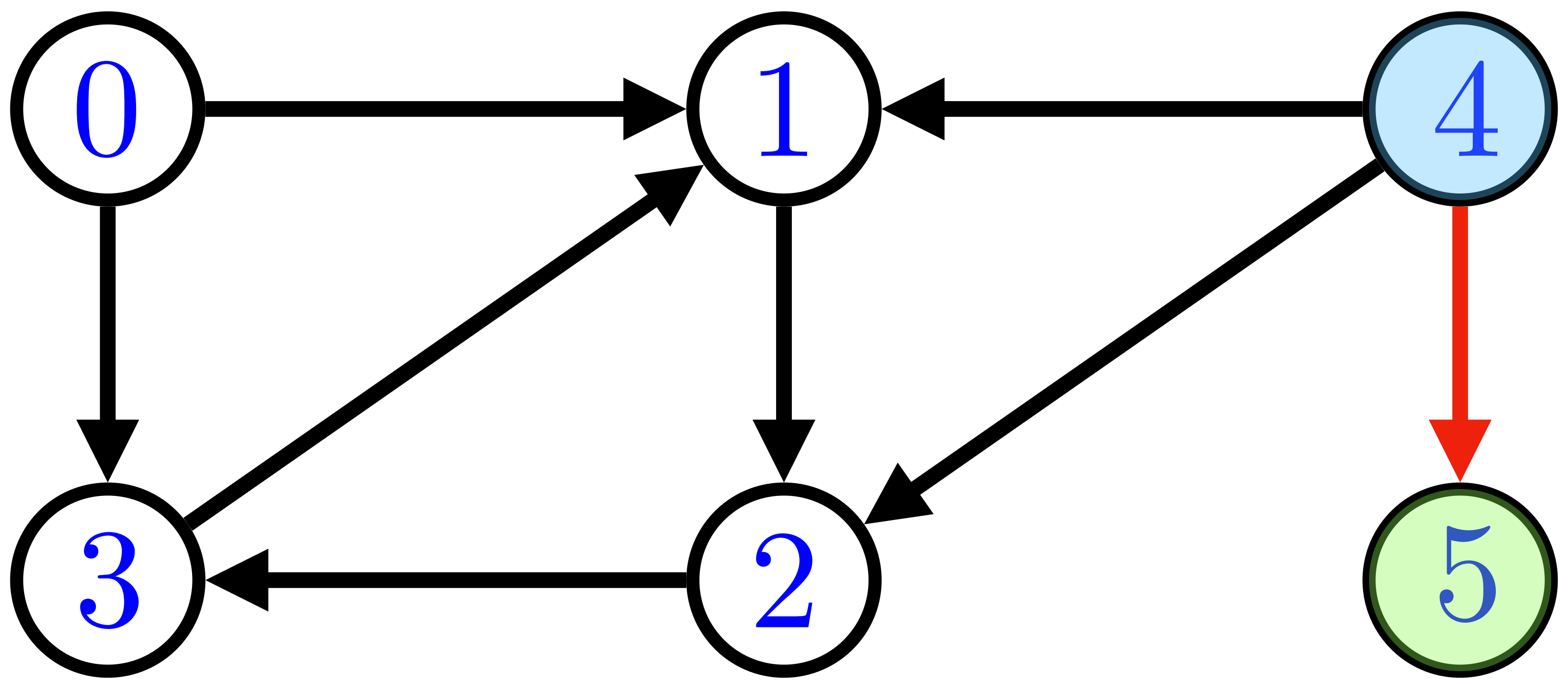
Adjacency List

on_stack

4

```
void dfs_visit(unsigned v) {
    marked[v] = true;
    on_stack[v] = true;
    for(auto u : arr[v]) {
        if(!marked[u]) {
          edge_to[u] = v;
          dfs_visit(u);
        }
        else if(on_stack[u]){
          // found a cycle!
          // process the cycle
        }
    }
    on_stack[v] = false;
}
```

dfs_visit(5)



0: 1 3
1: 2
2: 3
3: 1
4: 5 2 1
5:

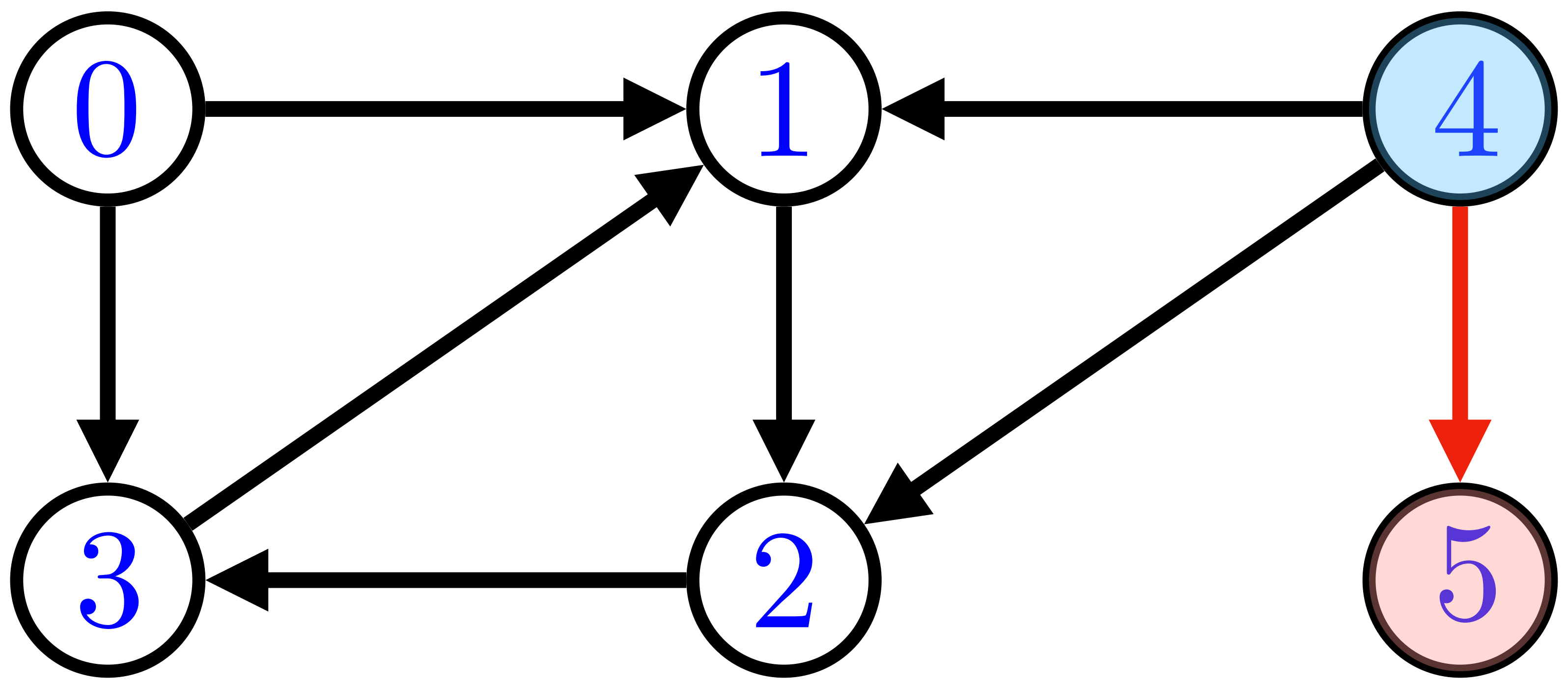
Adjacency List

on_stack

4 5 5

```
void dfs_visit(unsigned v) {
    marked[v] = true;
    on_stack[v] = true;
    for(auto u : arr[v]) {
        if(!marked[u]) {
          edge_to[u] = v;
          dfs_visit(u);
        }
        else if(on_stack[u]){
          // found a cycle!
          // process the cycle
        }
    }
    on_stack[v] = false;
}
```

dfs_visit(5)



0: 1 3
1: 2
2: 3
3: 1
4: 5 2 1
5:

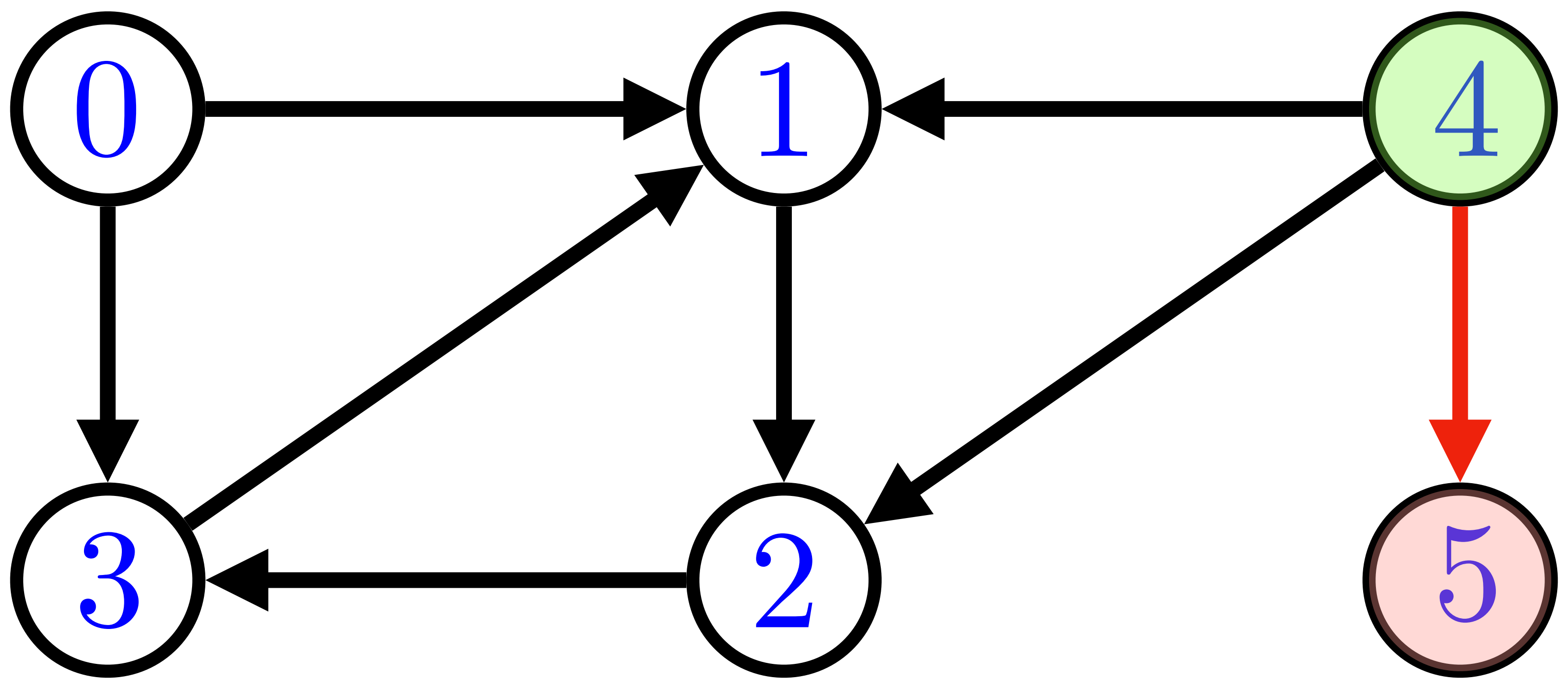
Adjacency List

on_stack

4 5 5

```
void dfs_visit(unsigned v) {
    marked[v] = true;
    on_stack[v] = true;
    for(auto u : arr[v]) {
        if(!marked[u]) {
          edge_to[u] = v;
          dfs_visit(u);
        }
        else if(on_stack[u]){
          // found a cycle!
          // process the cycle
        }
    }
    on_stack[v] = false;
}
```

dfs_visit(4)



0: 1 3
1: 2
2: 3
3: 1
4: 5 2 1
5:

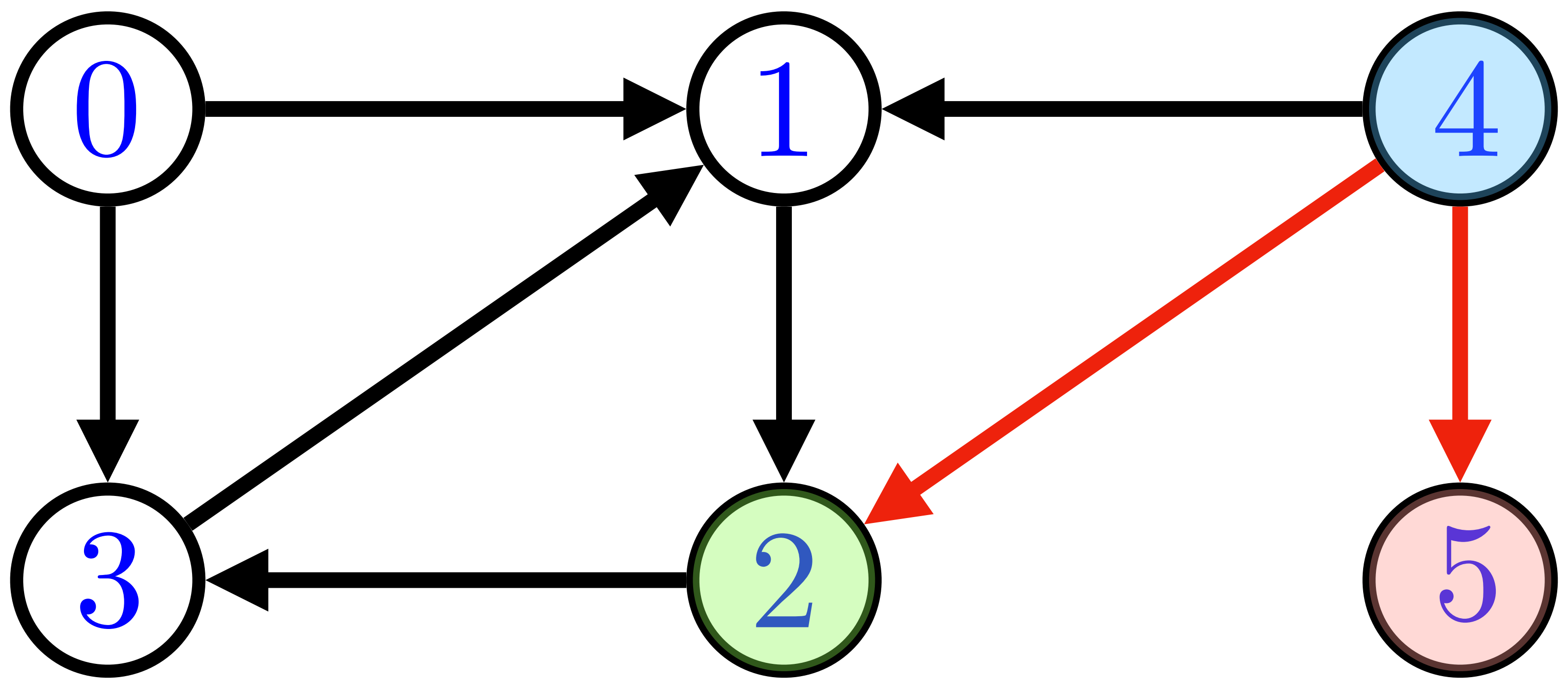
Adjacency List

on_stack

4 5 5

```
void dfs_visit(unsigned v) {
    marked[v] = true;
    on_stack[v] = true;
    for(auto u : arr[v]) {
        if(!marked[u]) {
          edge_to[u] = v;
          dfs_visit(u);
        }
        else if(on_stack[u]){
          // found a cycle!
          // process the cycle
        }
    }
    on_stack[v] = false;
}
```

dfs_visit(2)



0: 1 3
1: 2
2: 3
3: 1
4: 5 2 1
5:

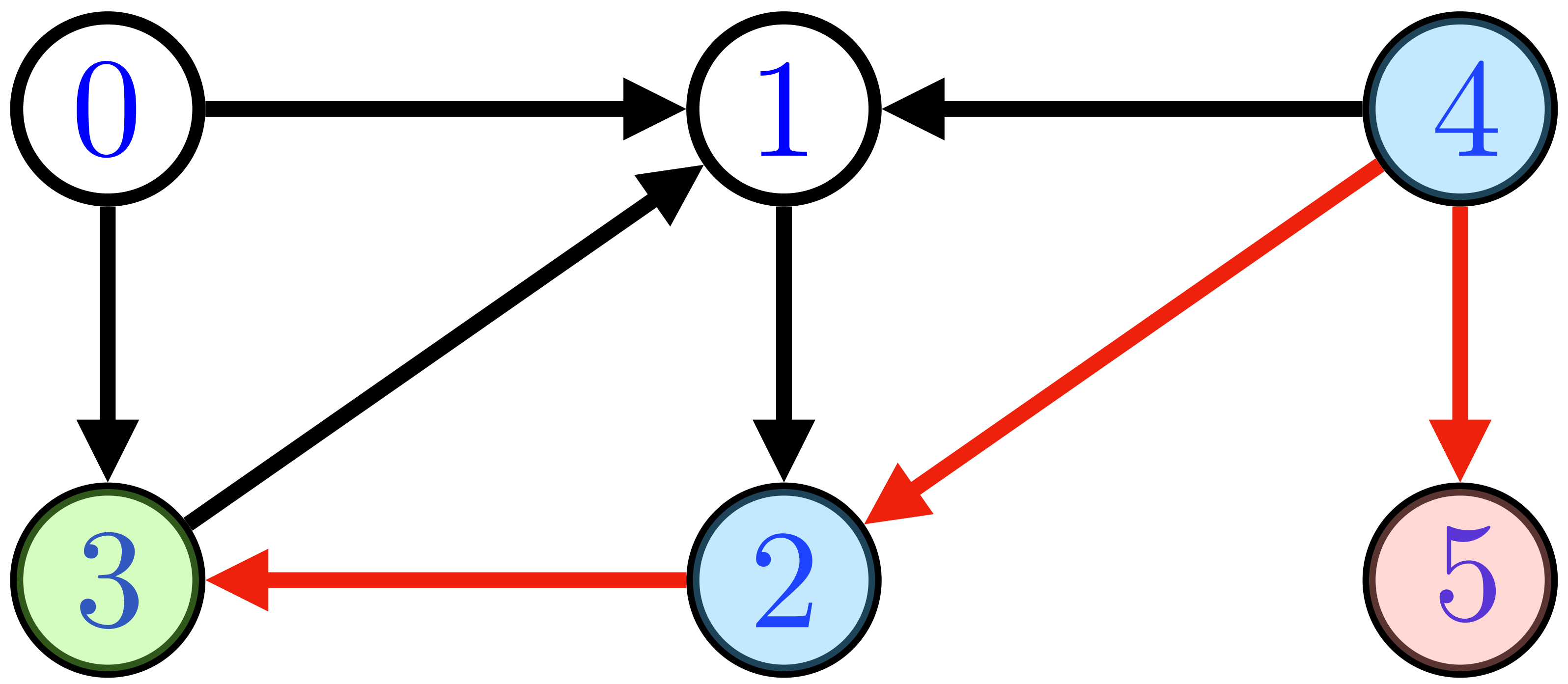
Adjacency List

on_stack

4 5 5 2

```
void dfs_visit(unsigned v) {
    marked[v] = true;
    on_stack[v] = true;
    for(auto u : arr[v]) {
        if(!marked[u]) {
          edge_to[u] = v;
          dfs_visit(u);
        }
        else if(on_stack[u]){
          // found a cycle!
          // process the cycle
        }
    }
    on_stack[v] = false;
}
```

dfs_visit(3)



0: 1 3
1: 2
2: 3
3: 1
4: 5 2 1
5:

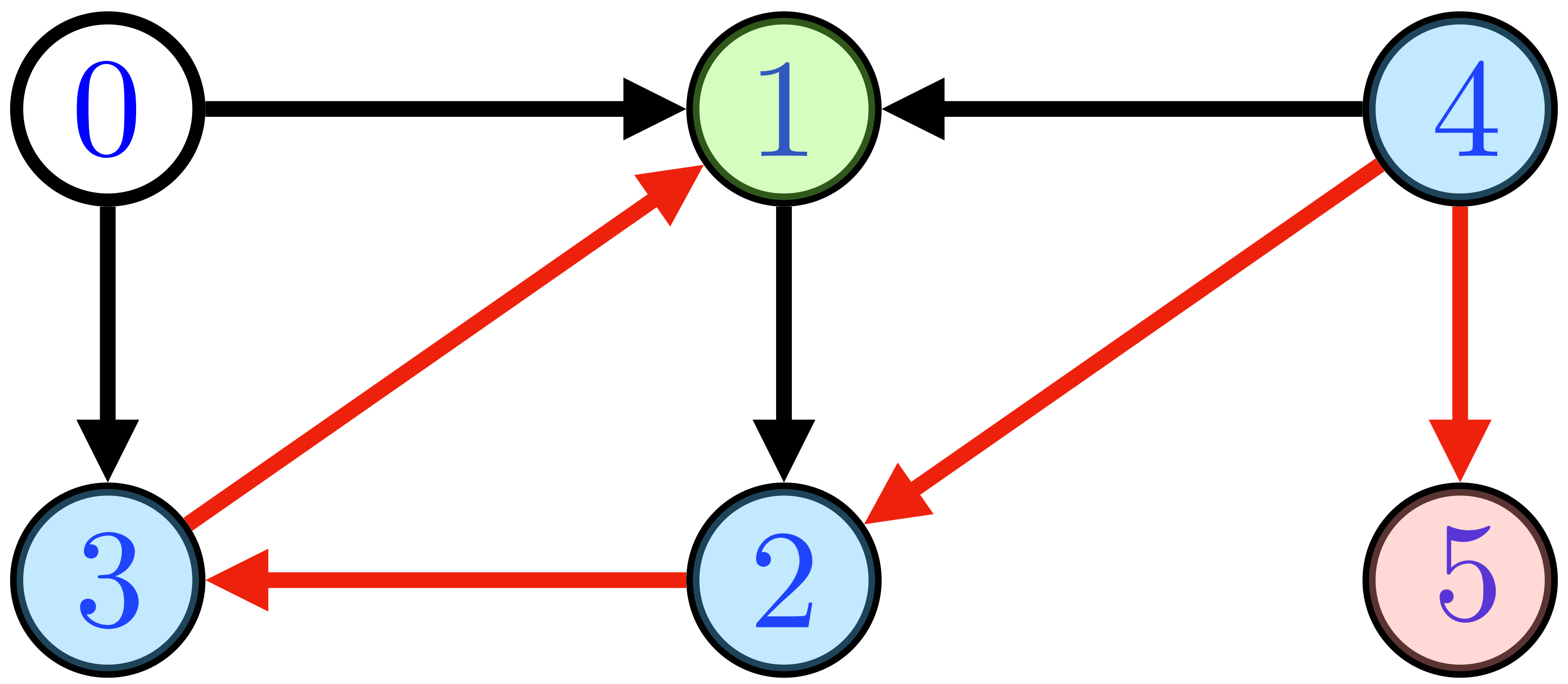
Adjacency List

on_stack

4 5 5 2 3

```
void dfs_visit(unsigned v) {
    marked[v] = true;
    on_stack[v] = true;
    for(auto u : arr[v]) {
        if(!marked[u]) {
          edge_to[u] = v;
          dfs_visit(u);
        }
        else if(on_stack[u]){
          // found a cycle!
          // process the cycle
        }
    }
    on_stack[v] = false;
}
```

$\texttt{dfs\_visit}(1)$



0: 1 3
1: 2
2: 3
3: 1
4: 5 2 1
5:

Adjacency List

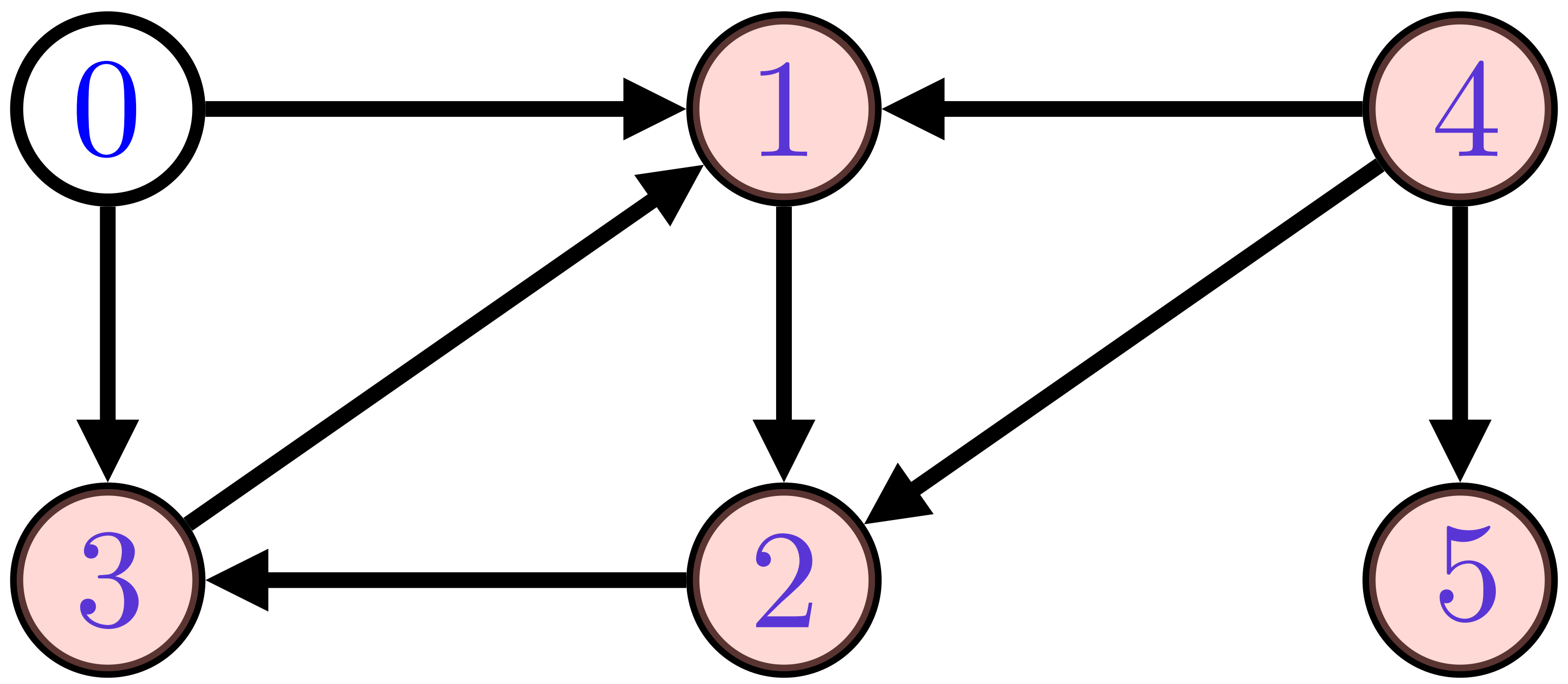
on_stack

4 5 5 2 3 1

```
void dfs_visit(unsigned v) {
    marked[v] = true;
    on_stack[v] = true;
    for(auto u : arr[v]) {
        if(!marked[u]) {
          edge_to[u] = v;
          dfs_visit(u);
        }
        else if(on_stack[u]){
          // found a cycle!
          // process the cycle
        }
    }
    on_stack[v] = false;
}
```

The edge $(1, 2)$ is a back edge. We have found a cycle!

dfs_visit(4)



0: 1 3
1: 2
2: 3
3: 1
4: 5 2 1
5:

Adjacency List

on_stack

(4(5)5(2(3(1)1)3)2)4

```
void dfs_visit(unsigned v) {
    marked[v] = true;
    on_stack[v] = true;
    for(auto u : arr[v]) {
        if(!marked[u]) {
          edge_to[u] = v;
          dfs_visit(u);
        }
        else if(on_stack[u]){
          // found a cycle!
          // process the cycle
        }
    }
    on_stack[v] = false;
}
```

Back Edges and Cycles

Claim: There is a cycle reachable from vertex v iff in $\texttt{dfs\_visit}(v)$ we find an edge to a vertex which is $\texttt{on\_stack}$.

Recall we call an edge from a vertex we are exploring to a vertex already on the stack a back edge.

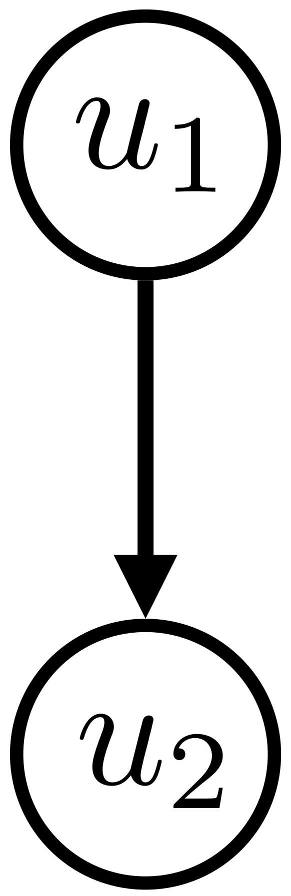
We have two show two things:

1) If we find a back edge then there is a cycle.

2) If there is a cycle then there is a back edge.

Back Edge → Cycle

Let's say that while exploring vertex u_1 we find an edge to vertex u_2 which is on the stack.

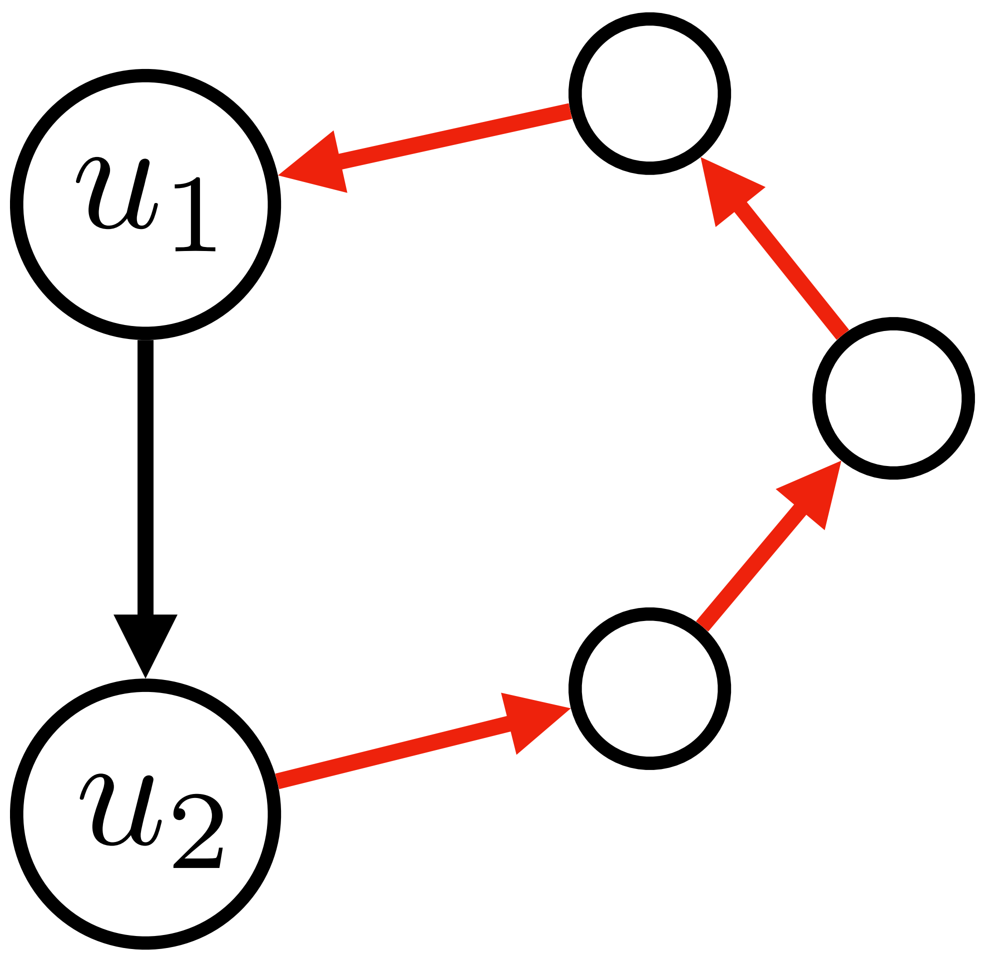


We arrived at u_1 in the call of $\texttt{dfs\_visit}(u_2)$.

$\texttt{dfs\_visit}(u_2)$ only visits vertices reachable from u_2 .

Back Edge → Cycle

Let's say that while exploring vertex u_1 we find an edge to vertex u_2 which is on the stack.

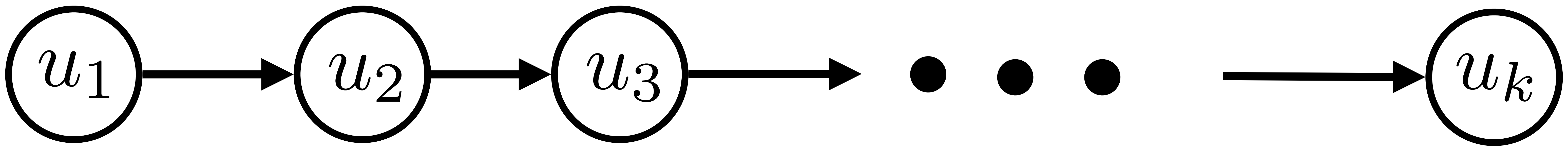


We arrived at u_1 in the call of $\texttt{dfs\_visit}(u_2)$.

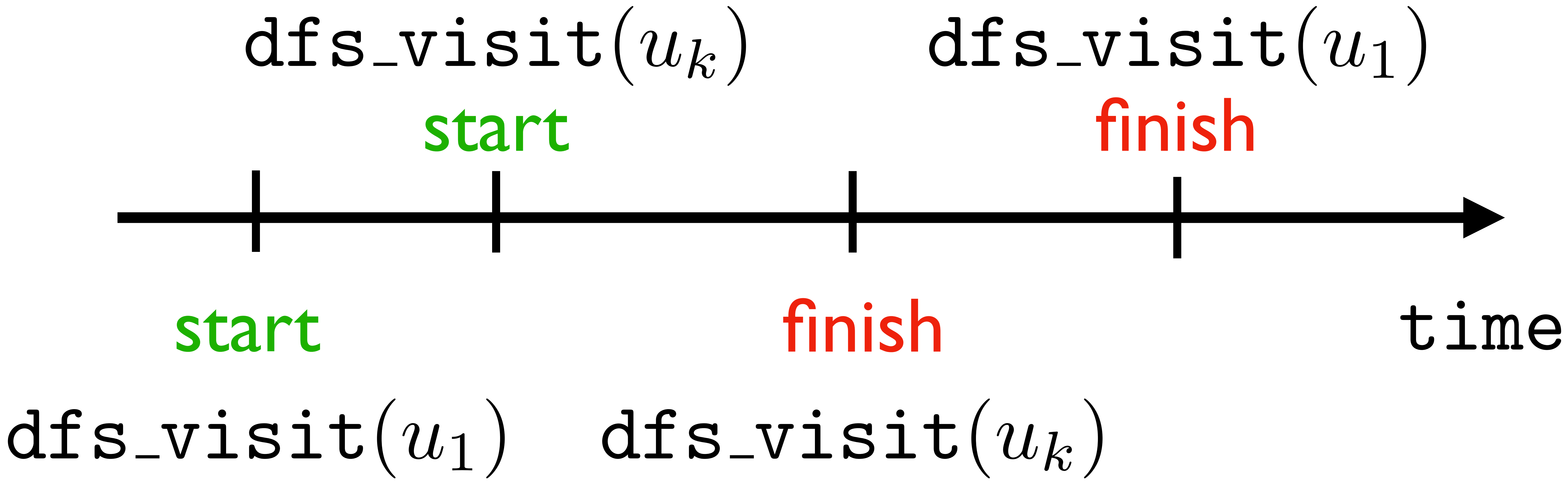
$\texttt{dfs\_visit}(u_2)$ only visits vertices reachable from u_2 .

So u_1 is reachable from u_2 , and this plus the back edge gives a cycle.

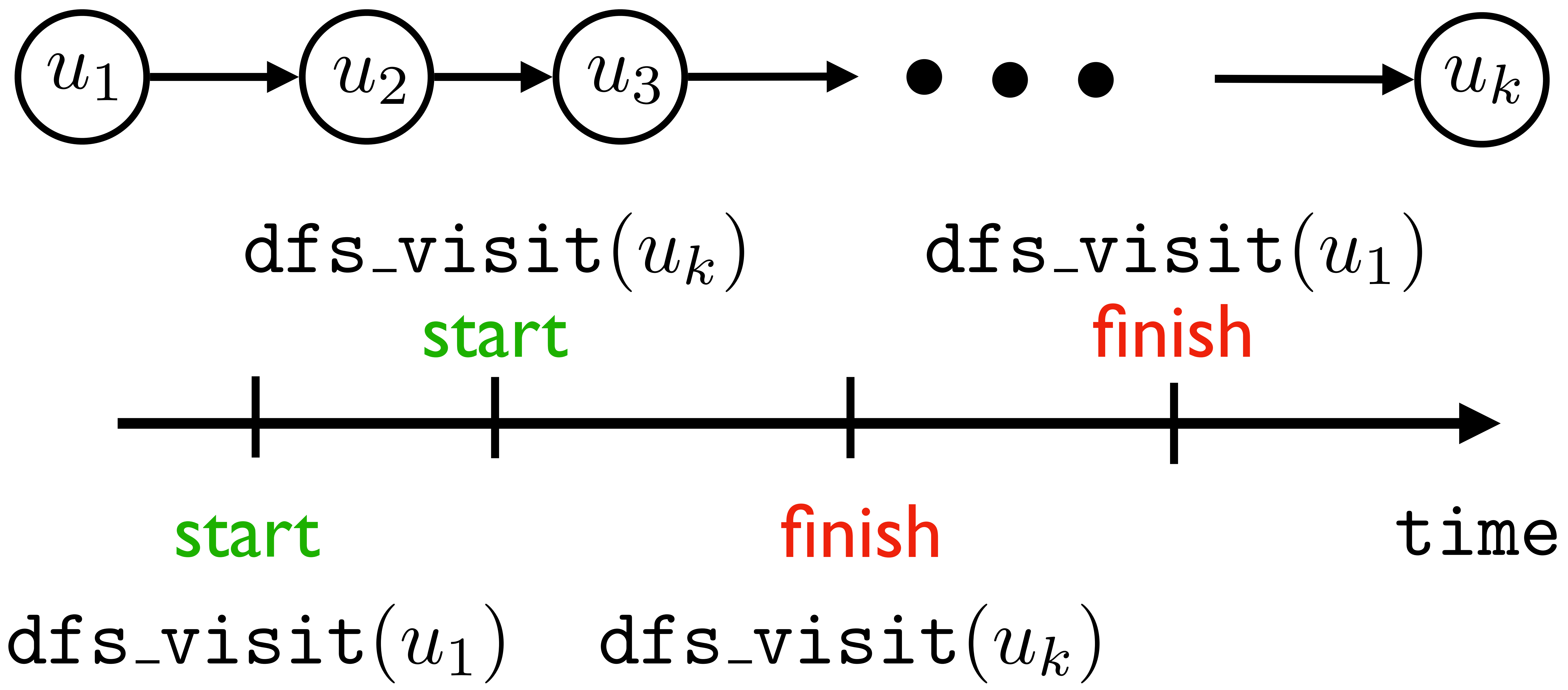
Unmarked Path Property



Unmarked path property: If there is an unmarked path from u_1 to u_k when $\texttt{dfs\_visit}(u_1)$ starts, then we will have the following ordering of start and finish times:

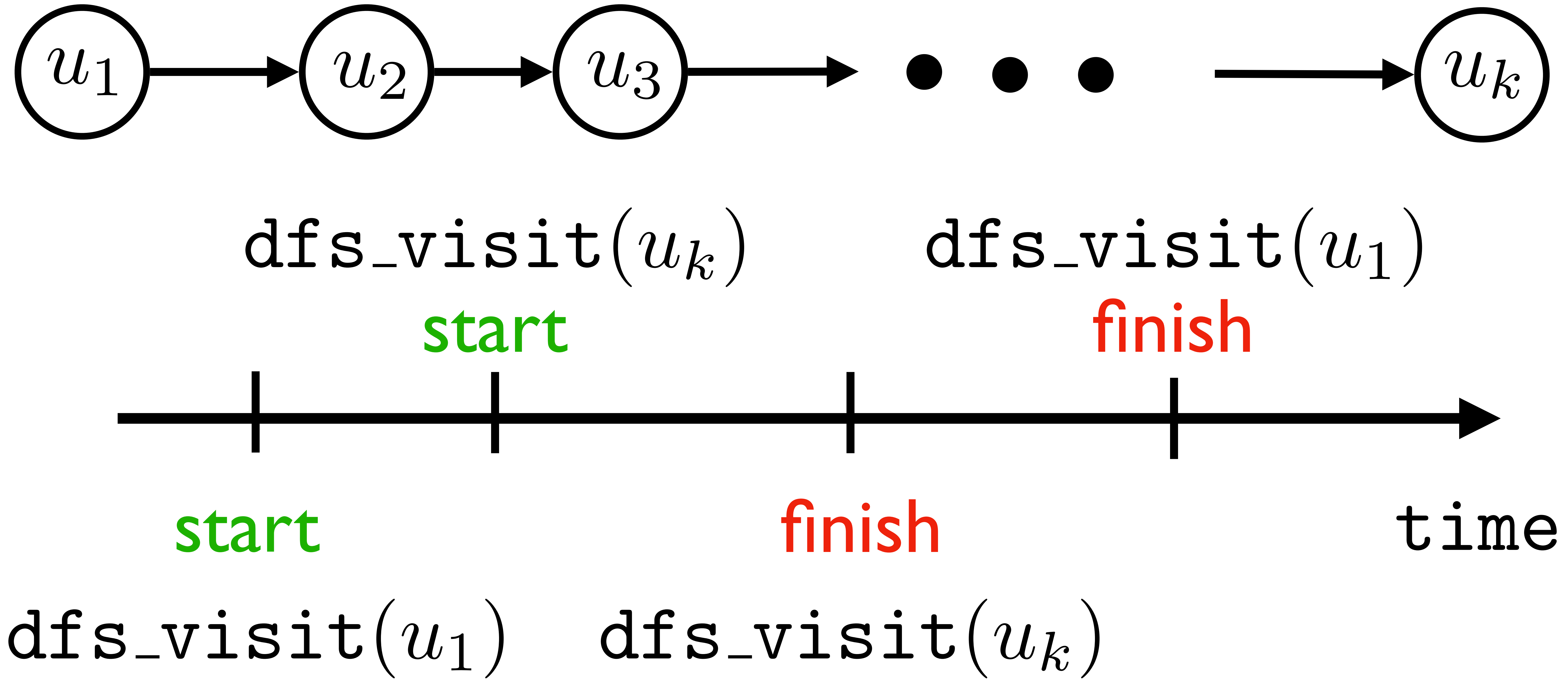


Unmarked Path Property



If dfs_visit(u_k) starts while dfs_visit(u_1) is still active, then dfs_visit(u_k) must finish before dfs_visit(u_1) can finish.

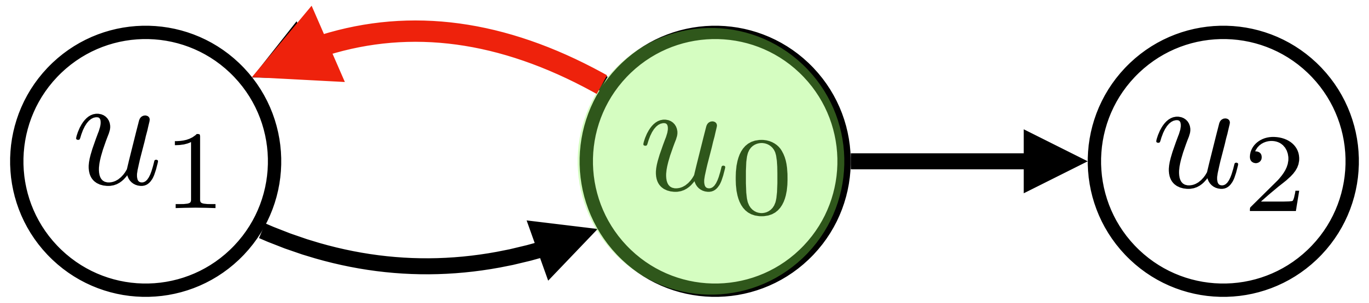
Unmarked Path Property



Show by induction that `dfs_visit` starts on all of $u_2, \ldots, u_k$ before `dfs_visit`(u_1) finishes.

We know this is true for u_2 because of the edge (u_1, u_2) .

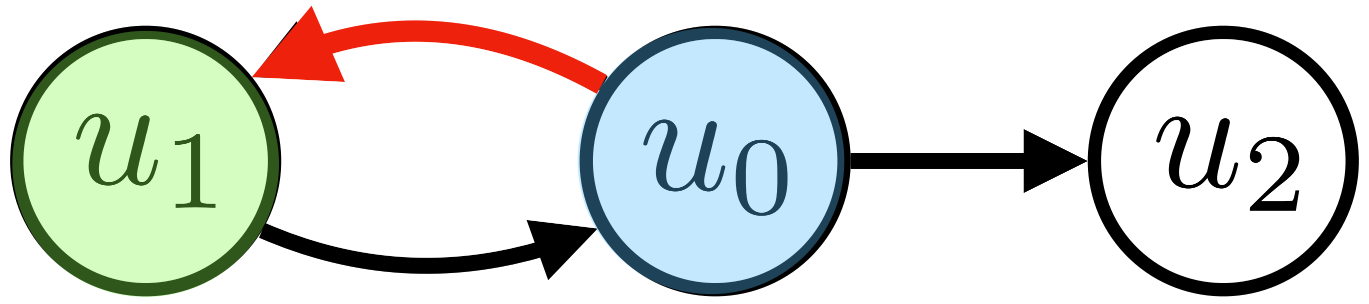
Need Path to be Unmarked



This example shows that you need the path to be unmarked for the unmarked path property to hold.

Suppose you start $\texttt{dfs\_visit}(u_0)$ and first visit u_1 .

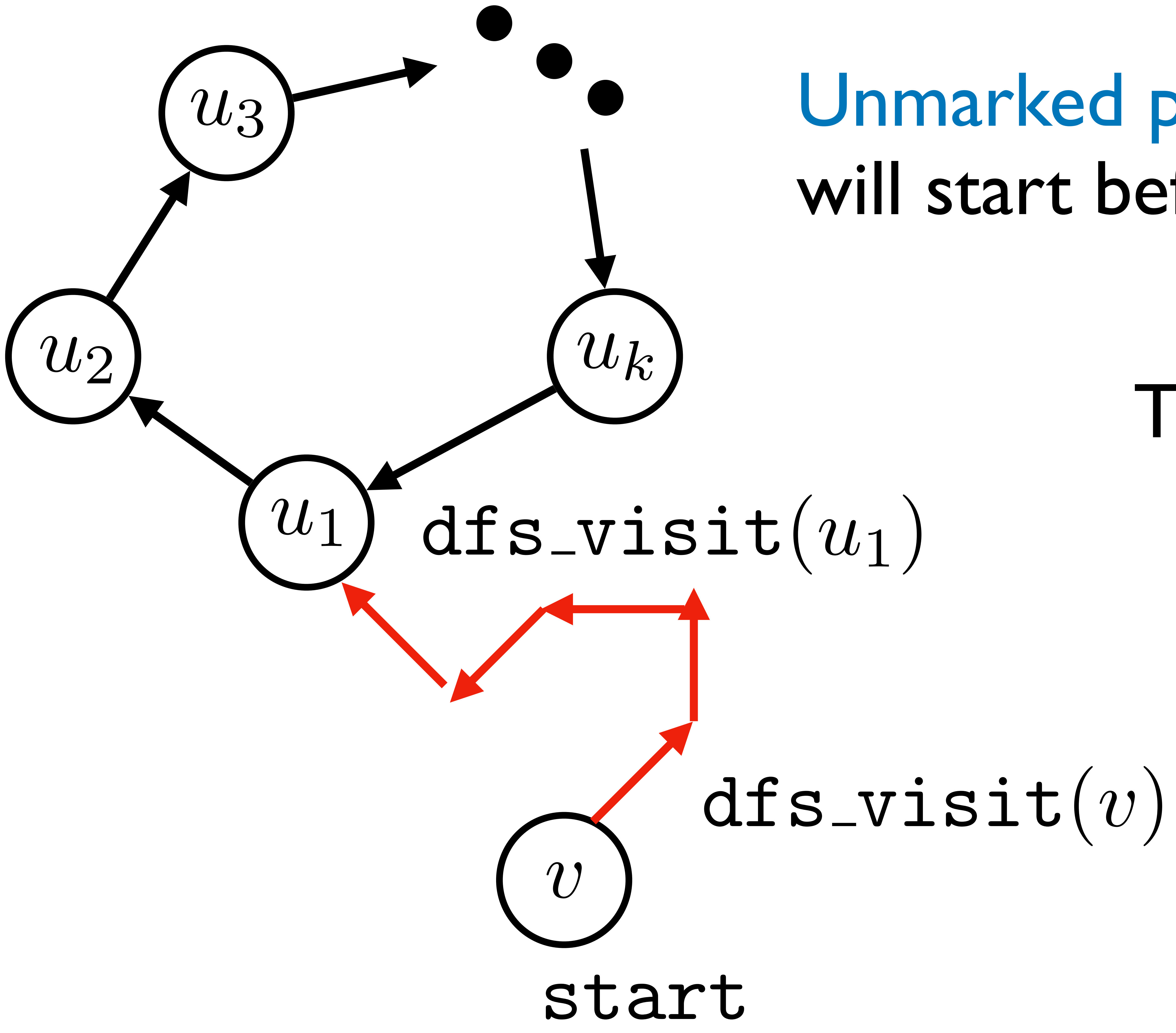
Need Path to be Unmarked



Now when we start $\texttt{dfs\_visit}(u_1)$ there is a path to u_2 but there is not an unmarked path.

And in this case $\texttt{dfs\_visit}(u_1)$ will finish before $\texttt{dfs\_visit}(u_2)$ begins.

Cycle → Back Edge

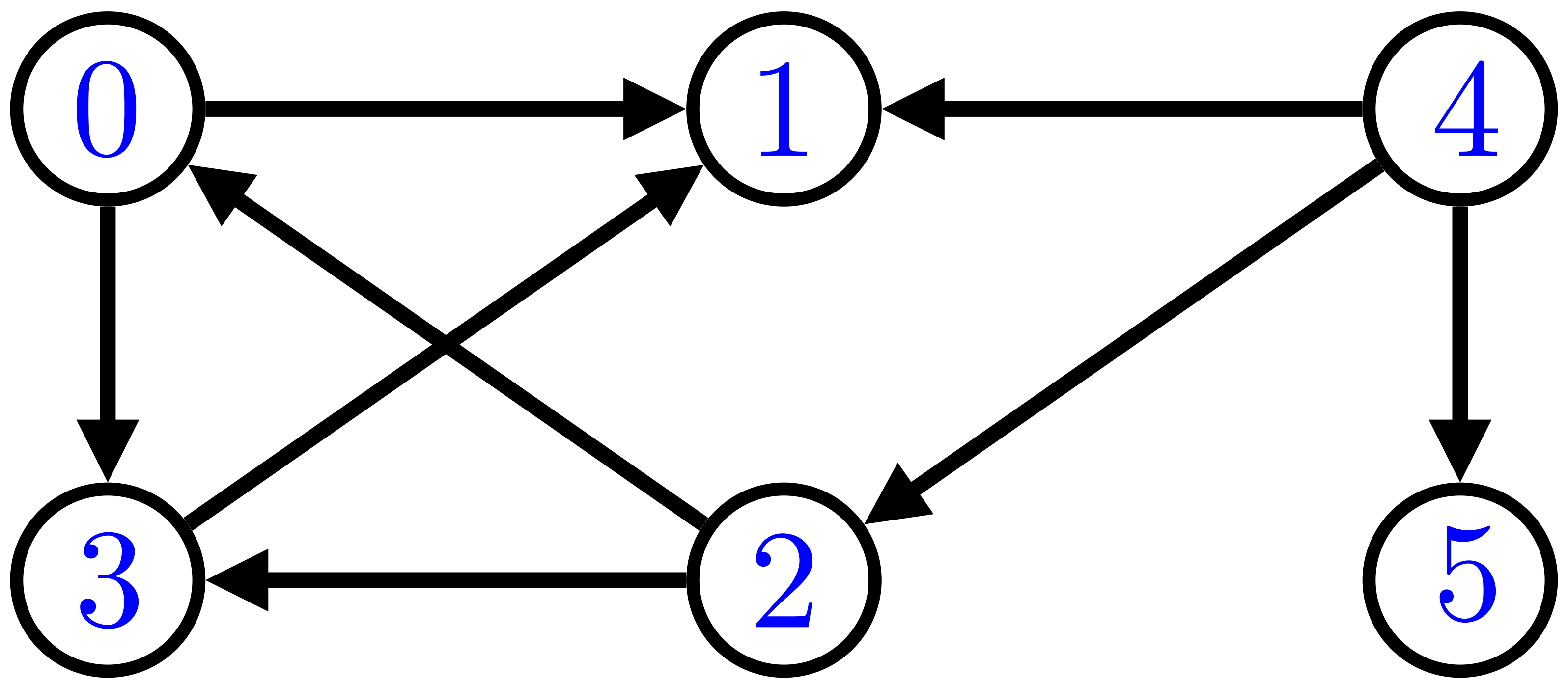


Unmarked path property implies $\texttt{dfs\_visit}(u_k)$ will start before $\texttt{dfs\_visit}(u_1)$ finishes.

The edge (u_k, u_1) will be a back edge.

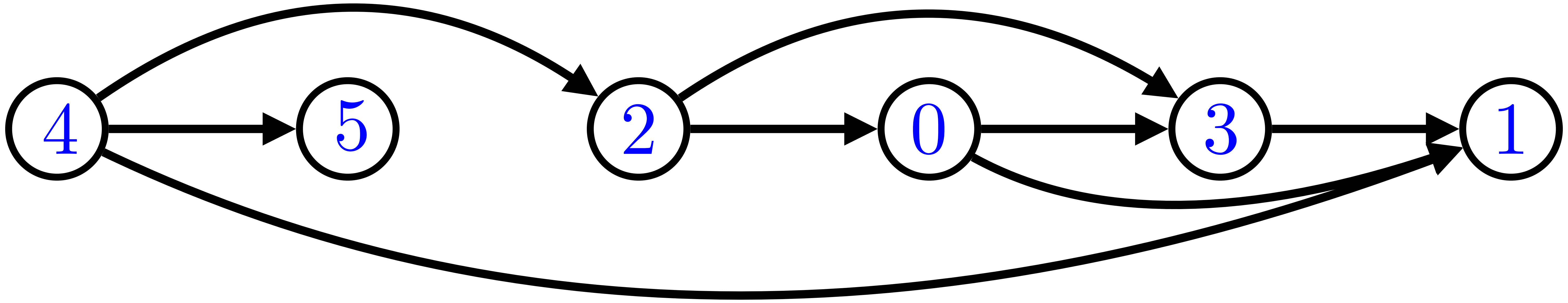
Topological Sort

Topological Sort

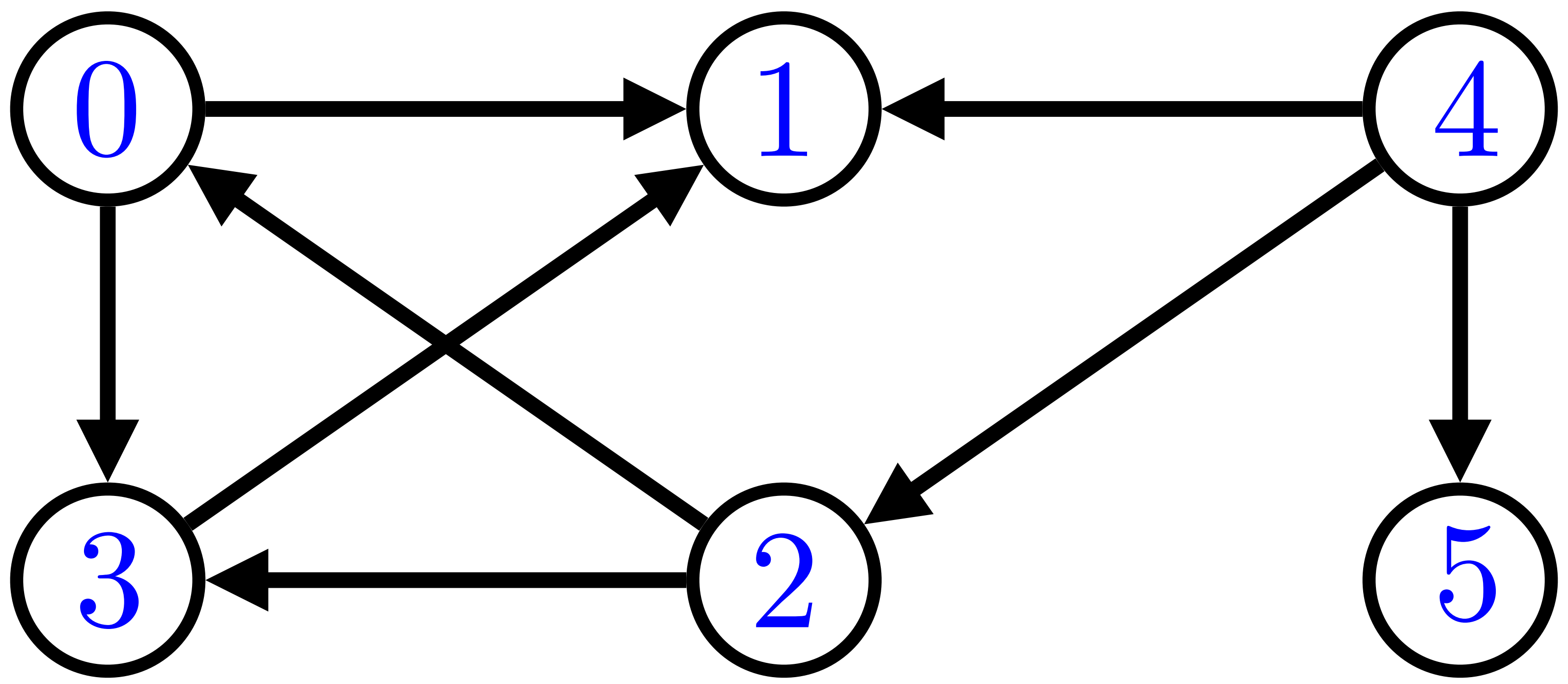


We want to order the vertices so that u comes before v in the ordering for every edge (u, v) in the graph.

Example topological sort: 4, 5, 2, 0, 3, 1



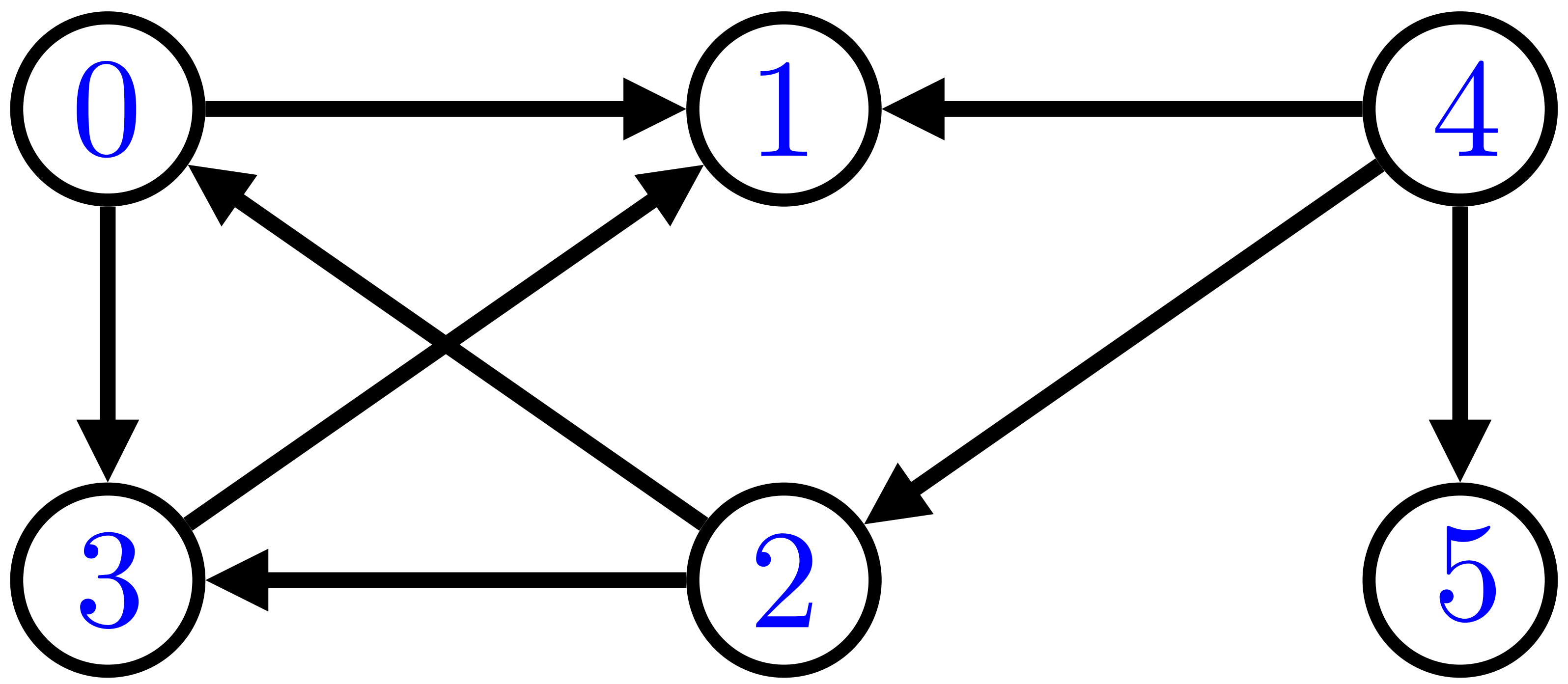
DFS Outer Loop




```
bool marked[N] {};

void dfs()
{
    for(unsigned v = 0; v < N; ++v)
    {
        if(!marked[v])
        {
            dfs_visit(v);
        }
    }
}
```

DFS Visit

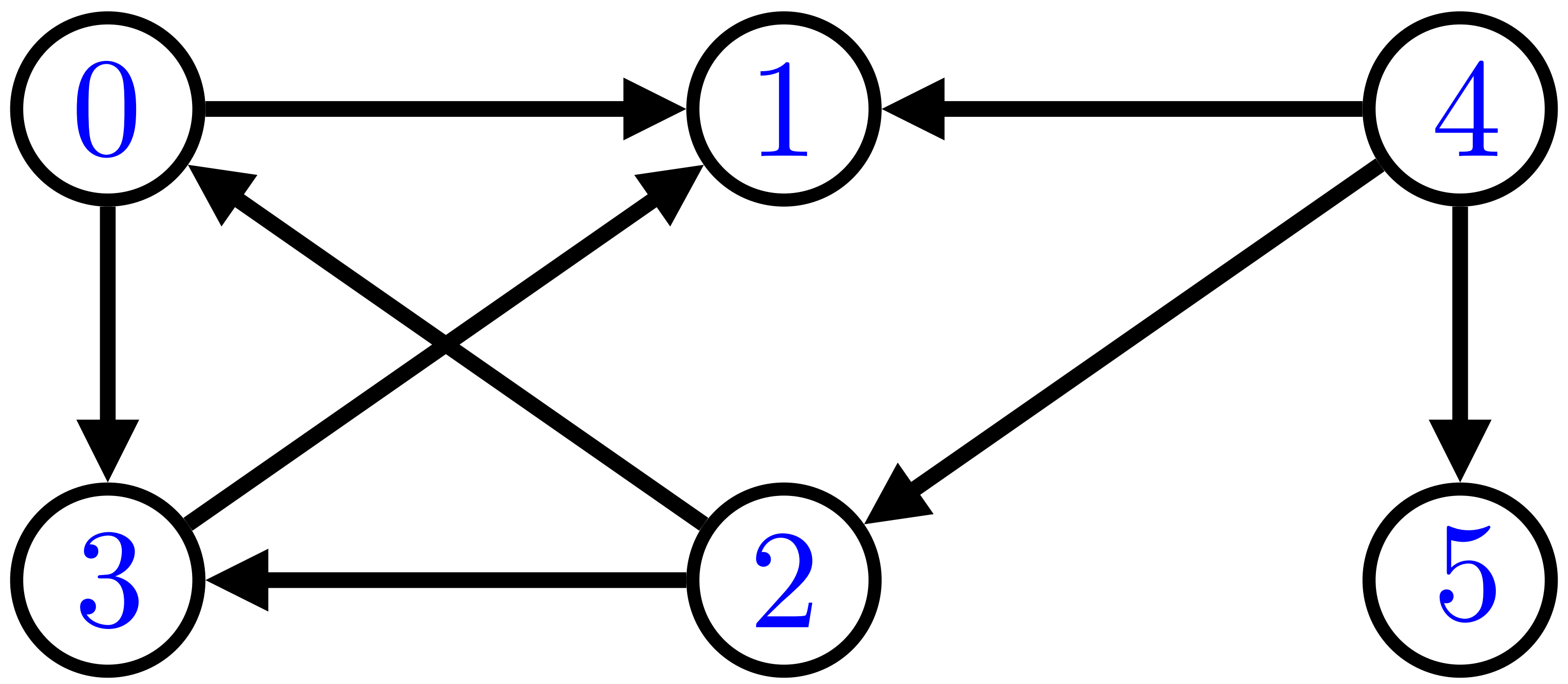


https://godbolt.org/z/Esa56YT4E

```
std::vector<unsigned> preorder {};
std::vector<unsigned> postorder {};
std::list<unsigned> reverse_postorder {};

void dfs_visit(unsigned v)
{
    marked[v] = true;
    on_stack[v] = true;
    preorder.push_back(v);
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            dfs_visit(u);
        }
    }
    postorder.push_back(v);
    reverse_postorder.push_front(v);
    on_stack[v] = false;
}
```

Preorder

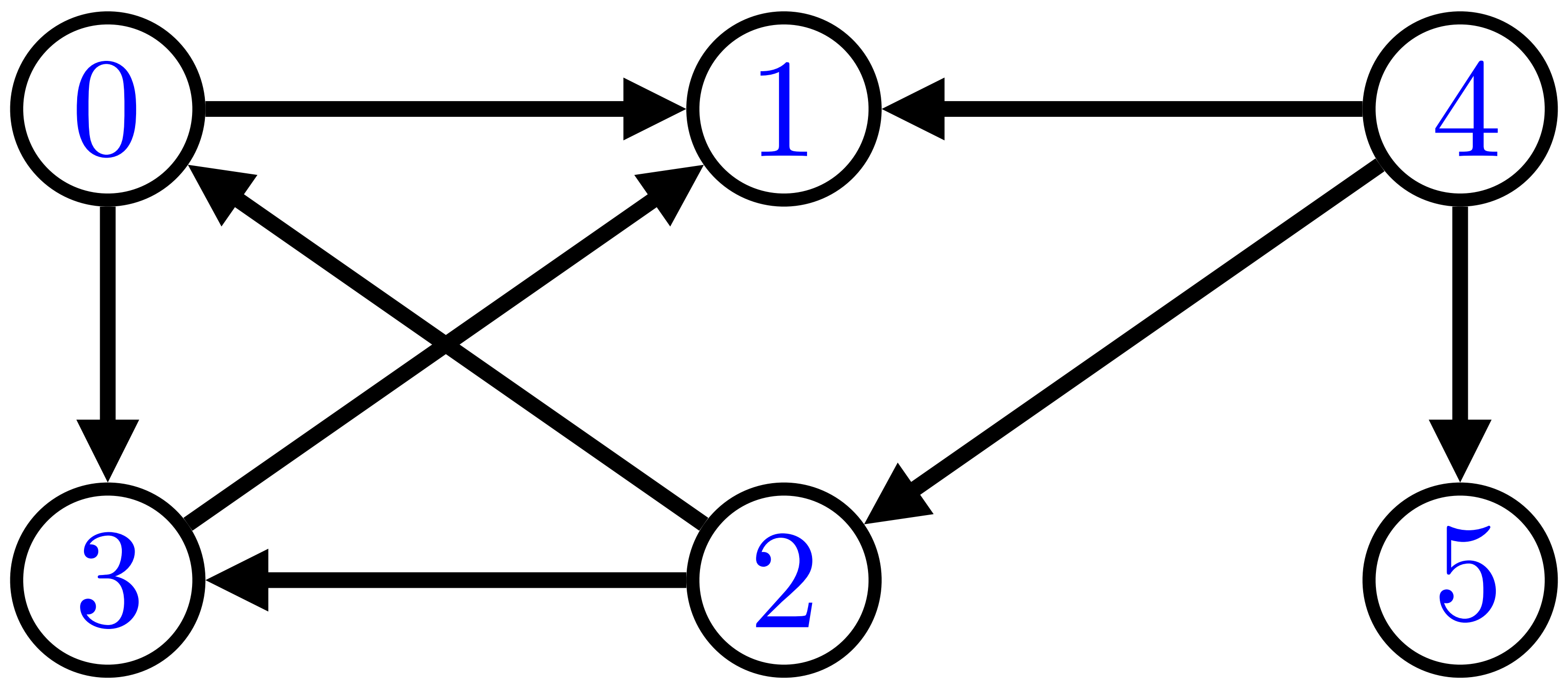


preorder: ordered by when `dfs_visit` starts on a vertex.

```
std::vector<unsigned> preorder {};

void dfs_visit(unsigned v)
{
    marked[v] = true;
    on_stack[v] = true;
    preorder.push_back(v);
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            dfs_visit(u);
        }
    }
    postorder.push_back(v);
    reverse_postorder.push_front(v);
    on_stack[v] = false;
}
```

Postorder

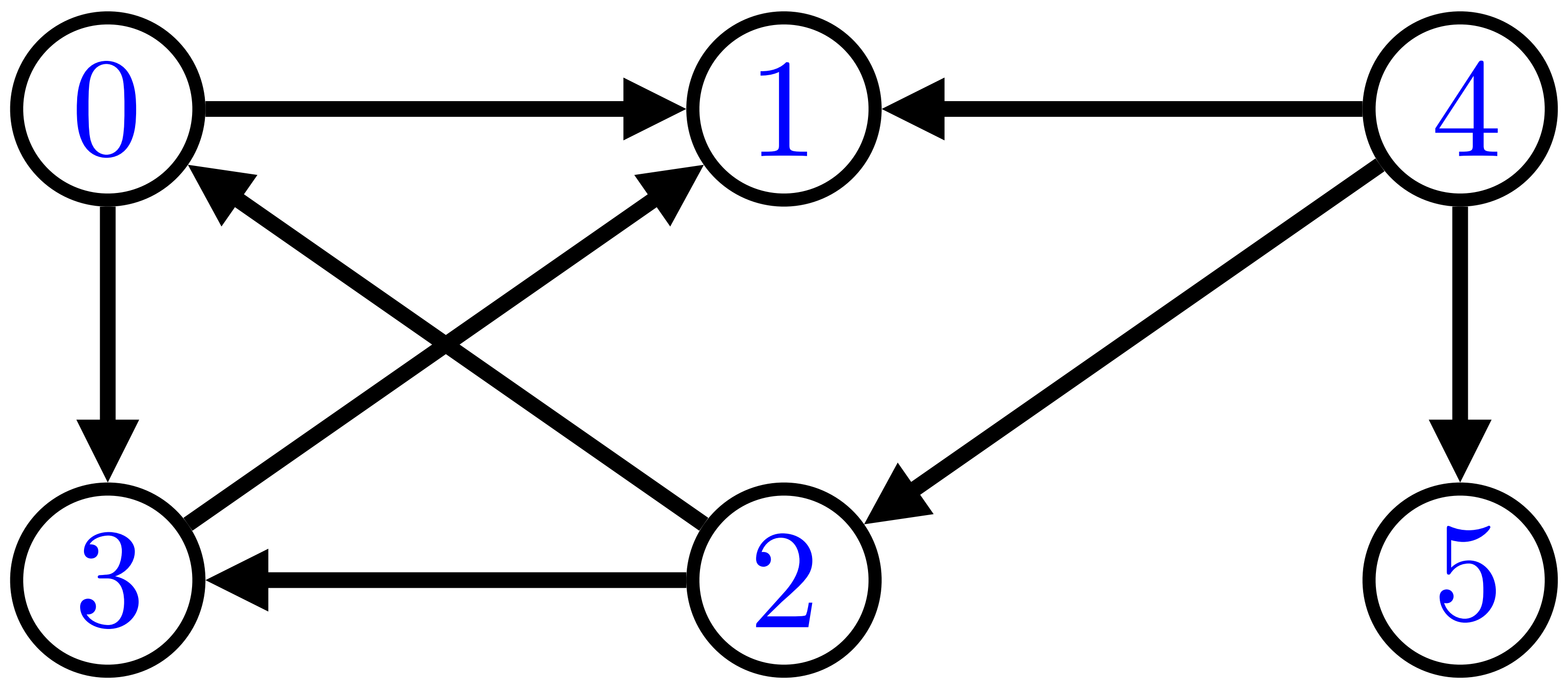



```
std::vector<unsigned> postorder {};

void dfs_visit(unsigned v)
{
    marked[v] = true;
    on_stack[v] = true;
    preorder.push_back(v);
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            dfs_visit(u);
        }
    }
    postorder.push_back(v);
    reverse_postorder.push_front(v);
    on_stack[v] = false;
}
```

postorder: ordered by when `dfs_visit` finishes on a vertex.

Reverse Postorder

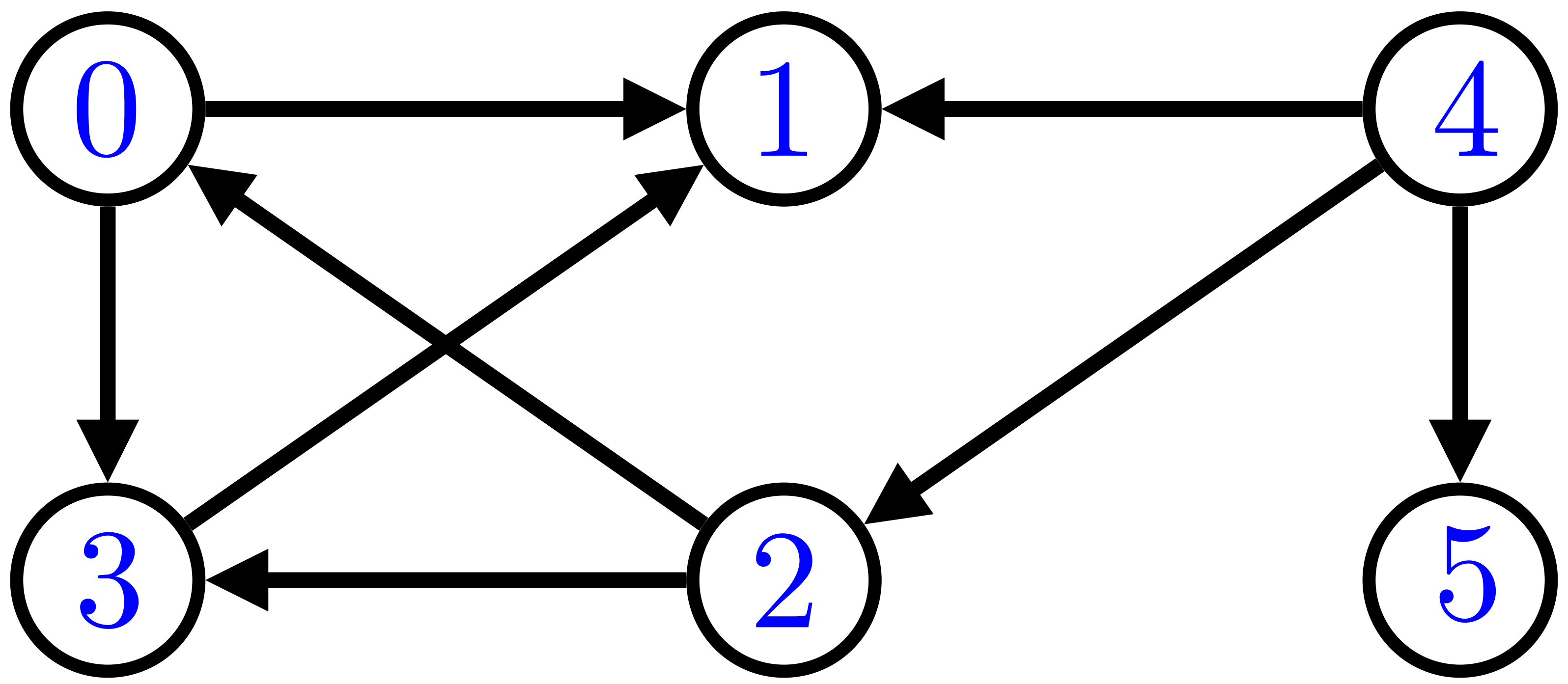



```
std::list<unsigned> reverse_postorder {};

void dfs_visit(unsigned v)
{
    marked[v] = true;
    on_stack[v] = true;
    preorder.push_back(v);
    for(auto u : arr[v])
    {
        if(!marked[u])
        {
            dfs_visit(u);
        }
    }
    postorder.push_back(v);
    reverse_postorder.push_front(v);
    on_stack[v] = false;
}
```

reverse postorder: the reverse order of when `dfs_visit` finishes on a vertex.

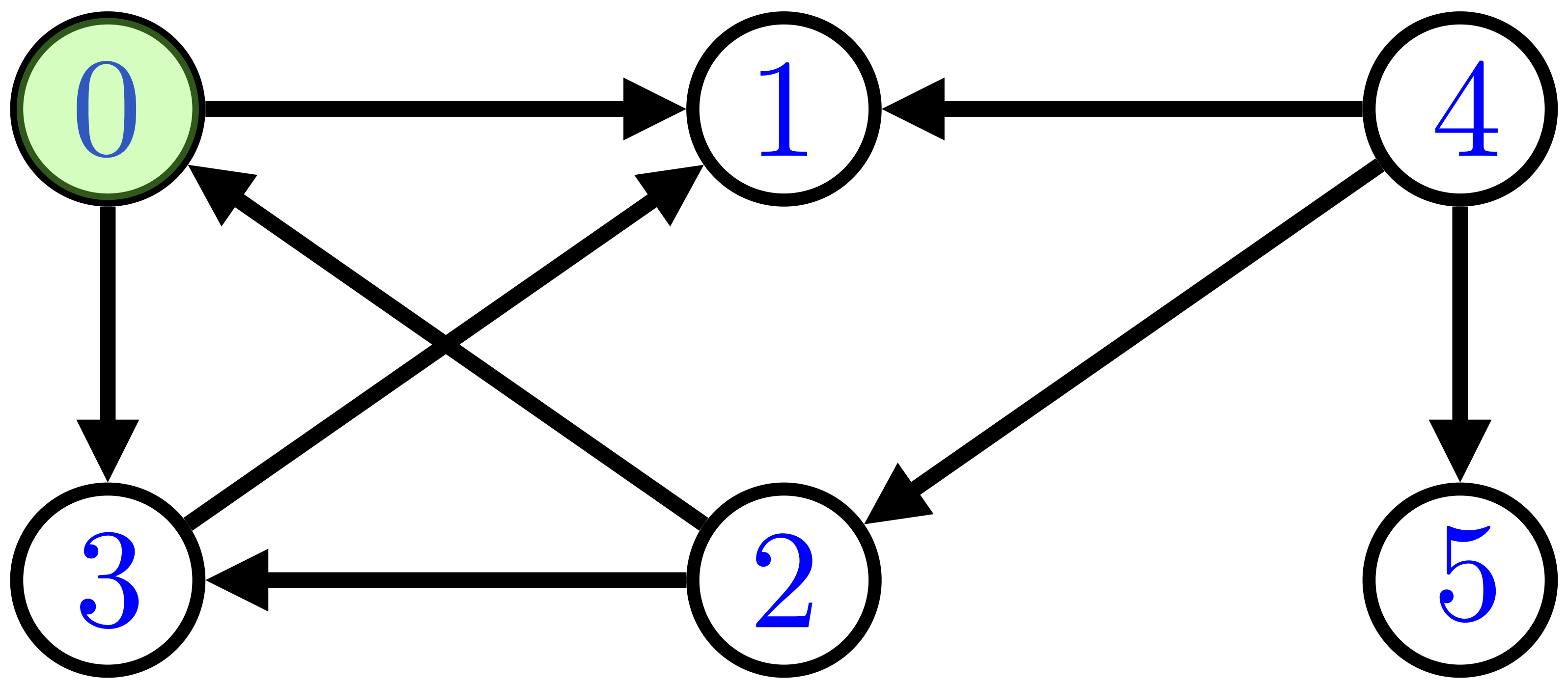
Reverse Postorder



Fact: If G is a DAG, then reverse postorder is a topological sort of the vertices.

reverse postorder: the reverse order of when `dfs_visit` finishes on a vertex.

Depth-First Search



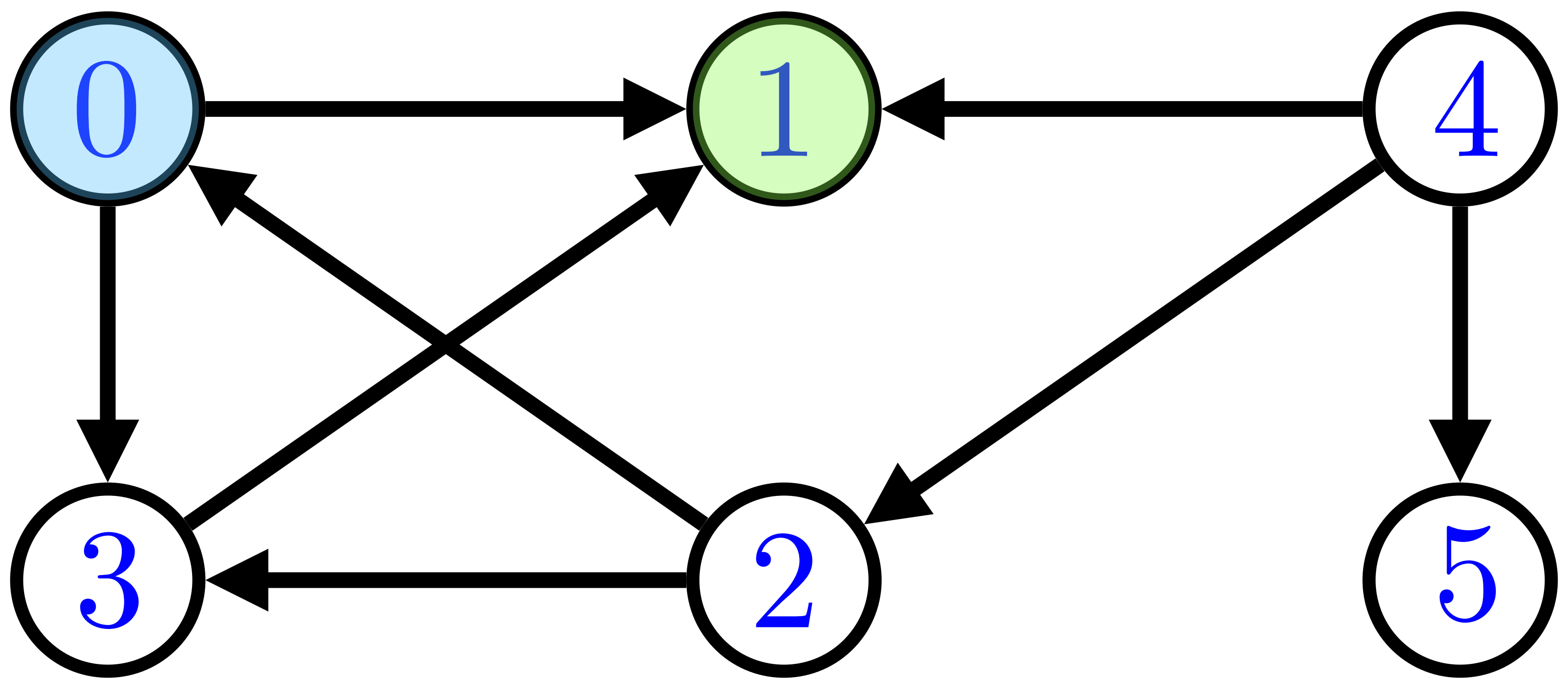
0: 1 3
1:
2: 3 0
3: 1
4: 5 2 1
5:

Adjacency List

on_stack

0

Depth-First Search



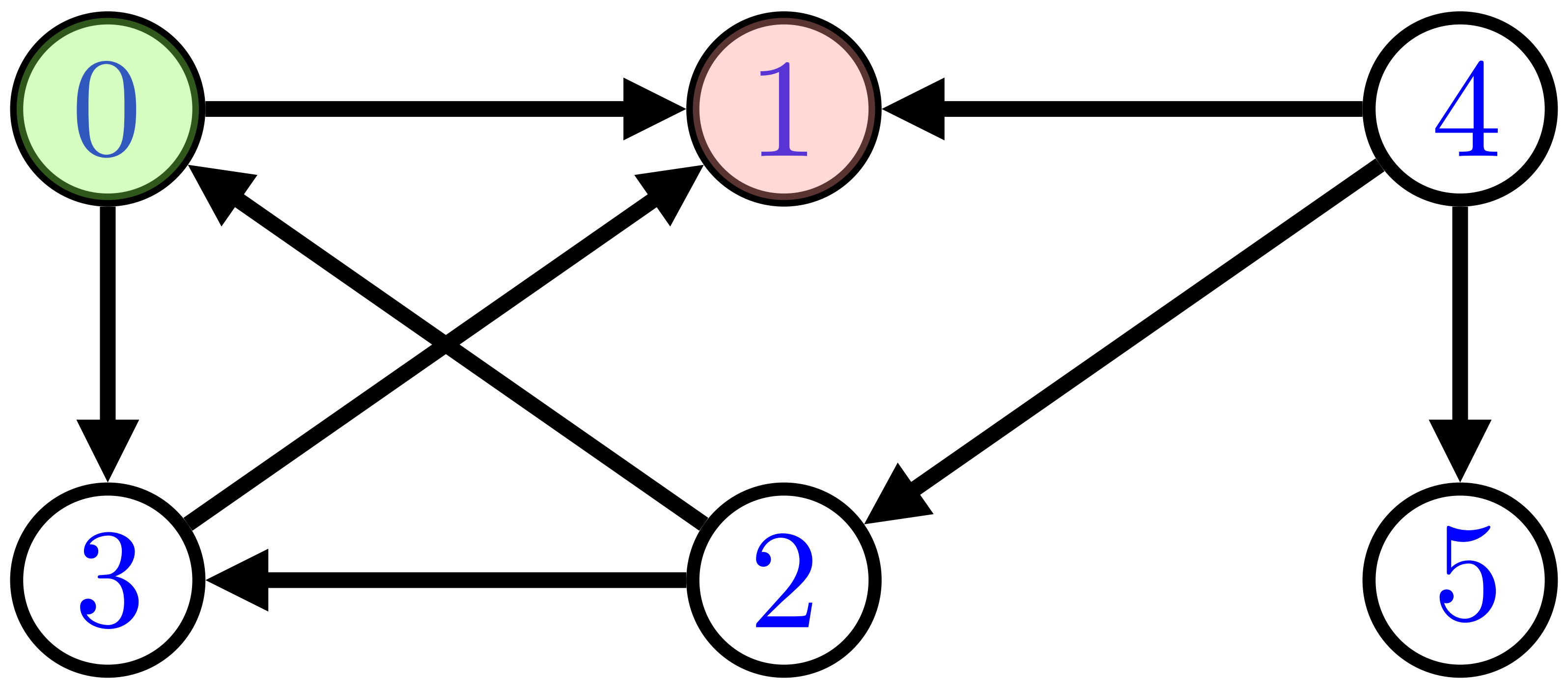
on_stack

0 1

0: 1 3
1:
2: 3 0
3: 1
4: 5 2 1
5:

Adjacency List

Depth-First Search



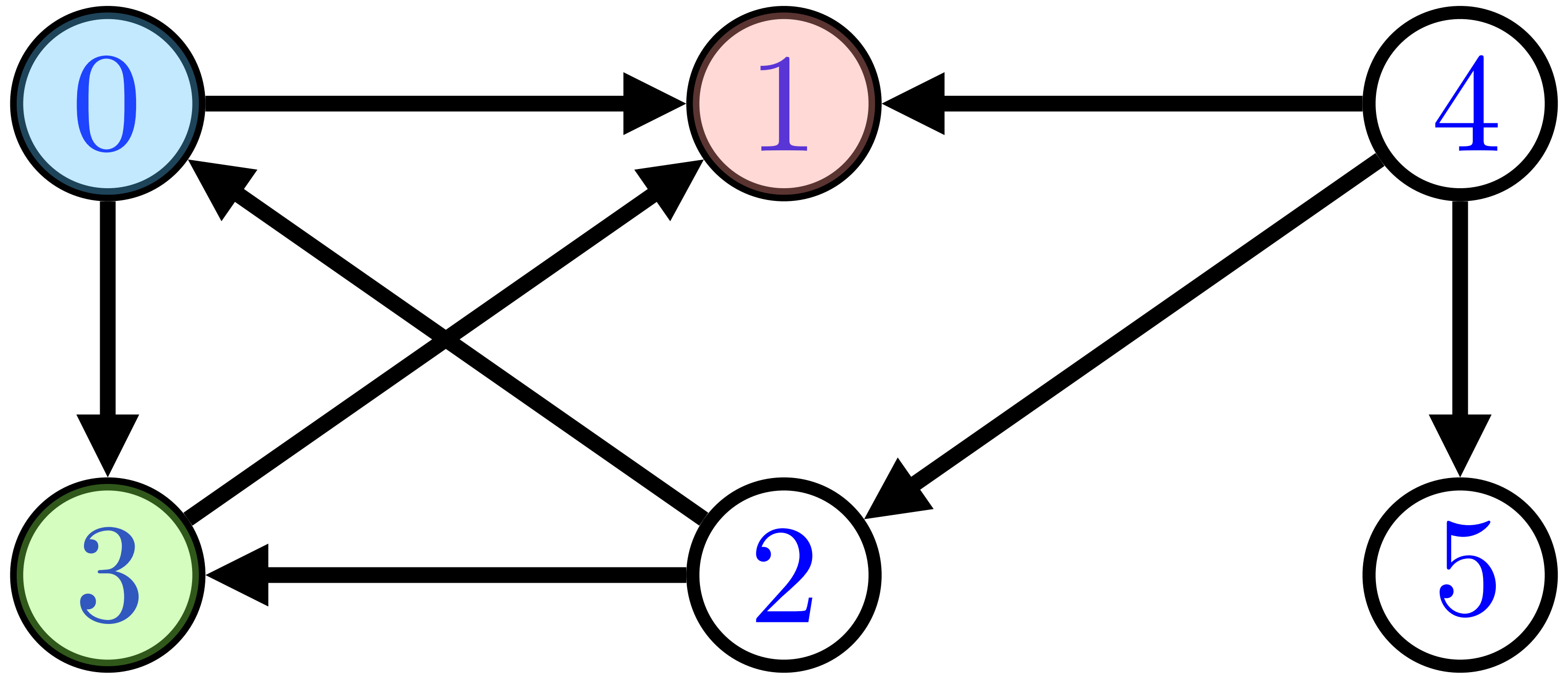
0: 1 3
1:
2: 3 0
3: 1
4: 5 2 1
5:

Adjacency List

`on_stack`

0 1 1

Depth-First Search



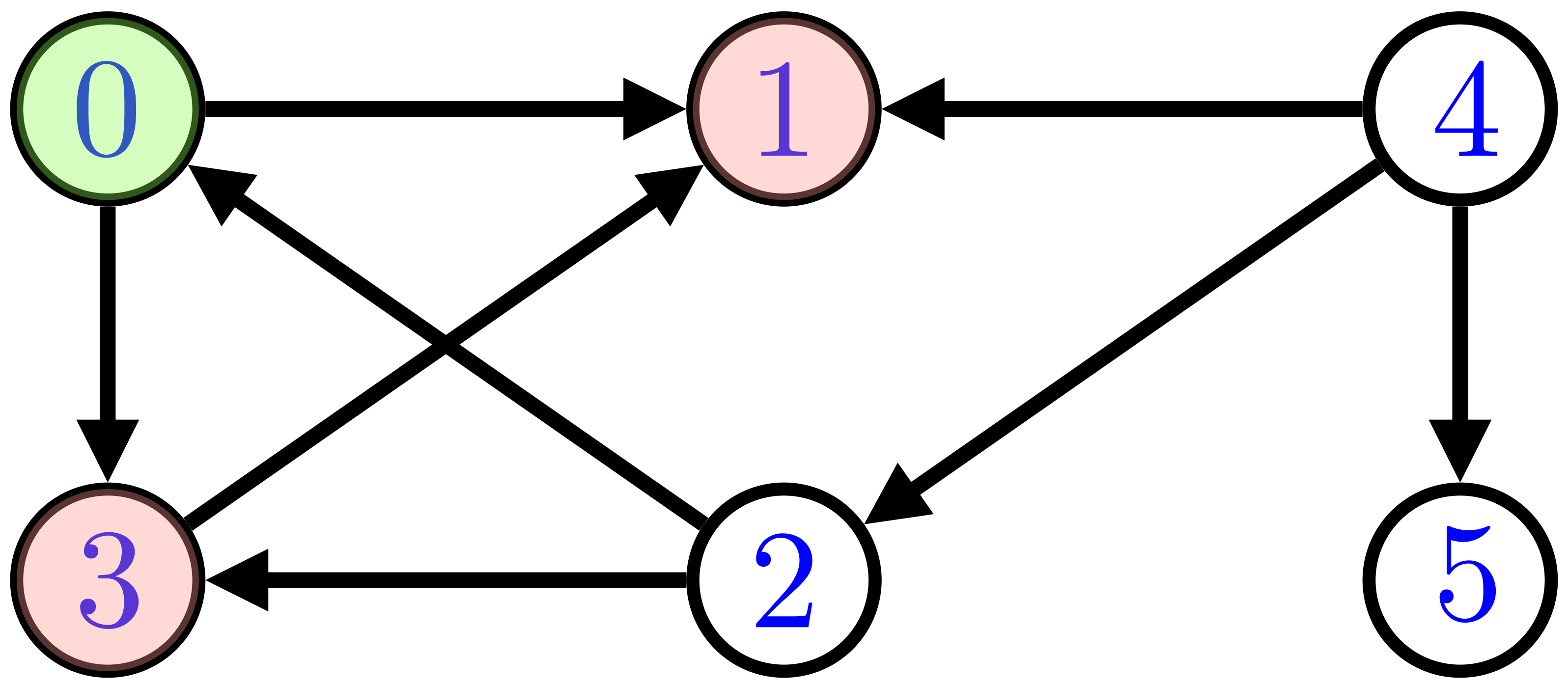
on_stack

0 1 1 3

0: 1 3
1:
2: 3 0
3: 1
4: 5 2 1
5:

Adjacency List

Depth-First Search



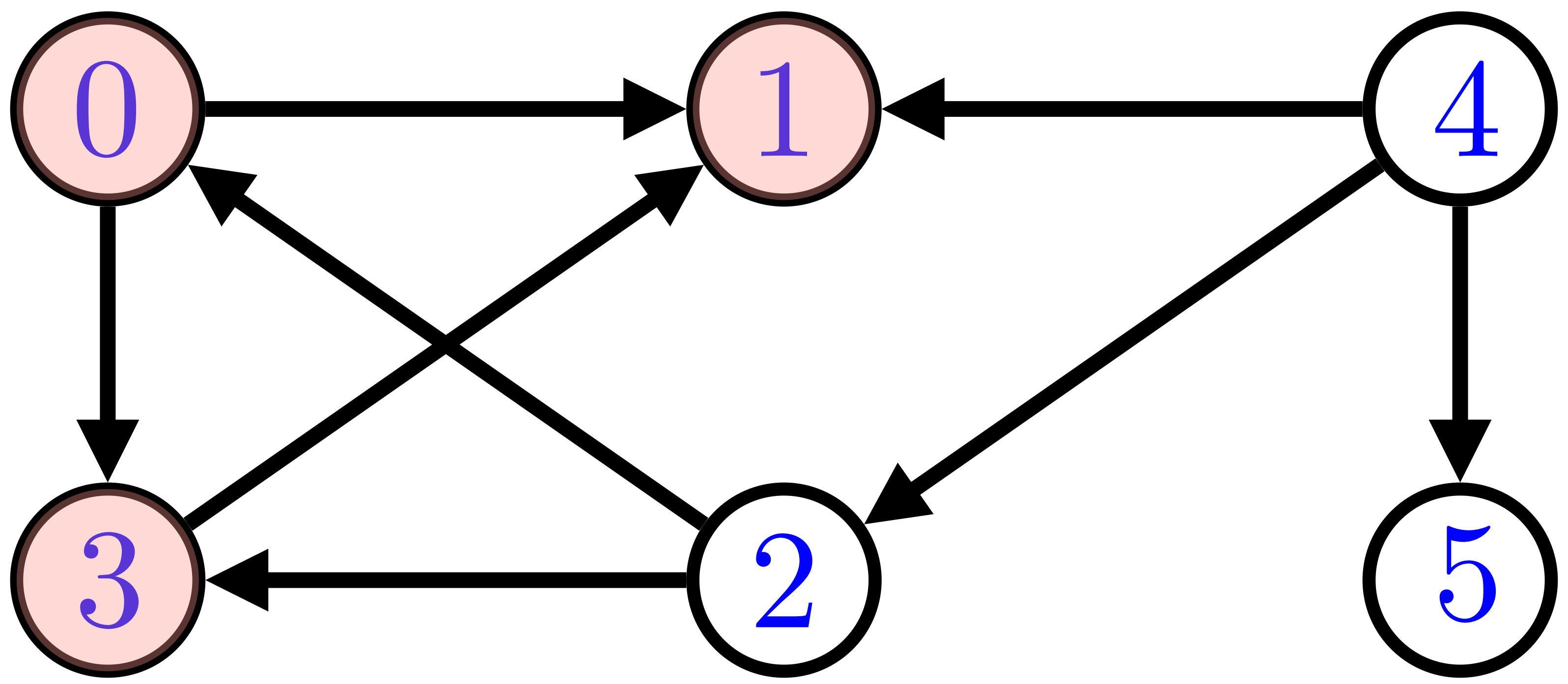
on_stack

0 1 1 3 3

0: 1 3
1:
2: 3 0
3: 1
4: 5 2 1
5:

Adjacency List

Depth-First Search



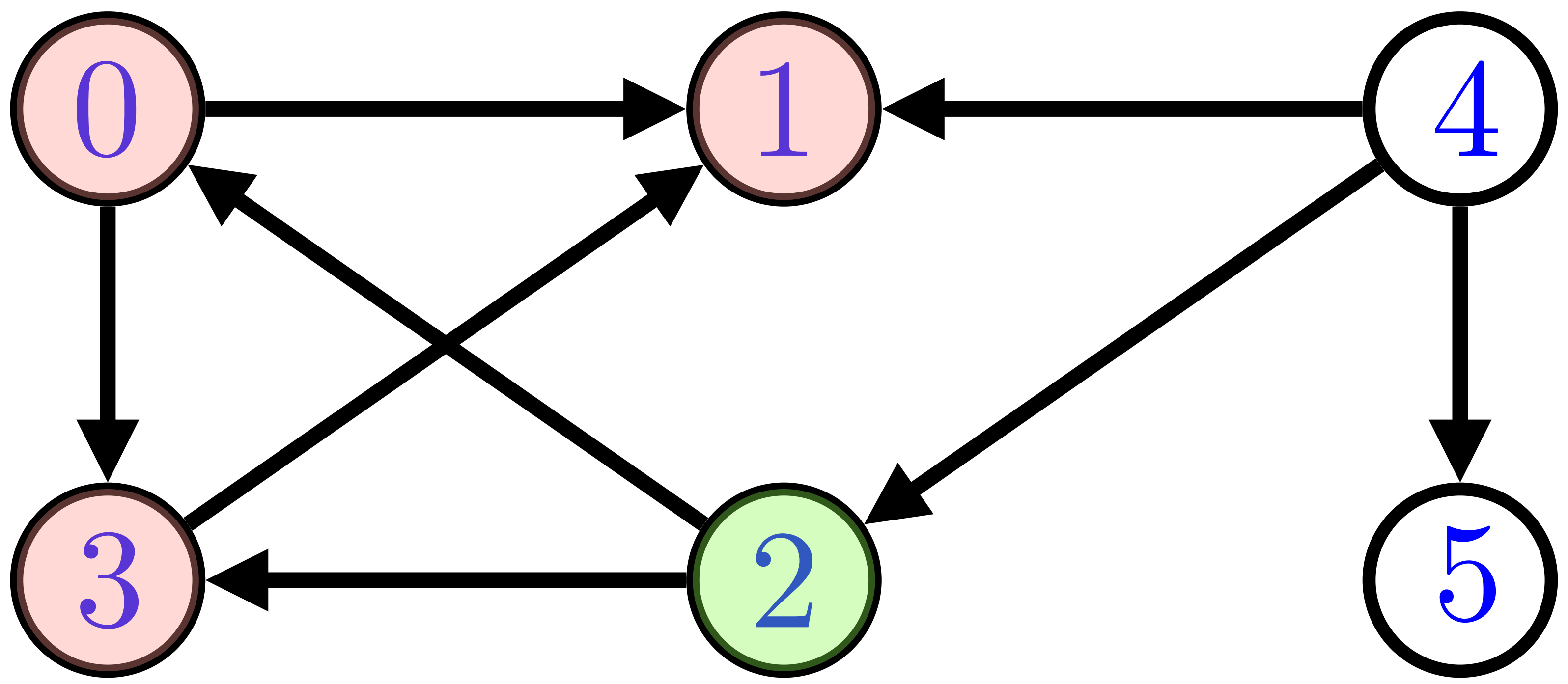
`on_stack`

0 1 1 3 3 0

0: 1 3
1:
2: 3 0
3: 1
4: 5 2 1
5:

Adjacency List

Depth-First Search



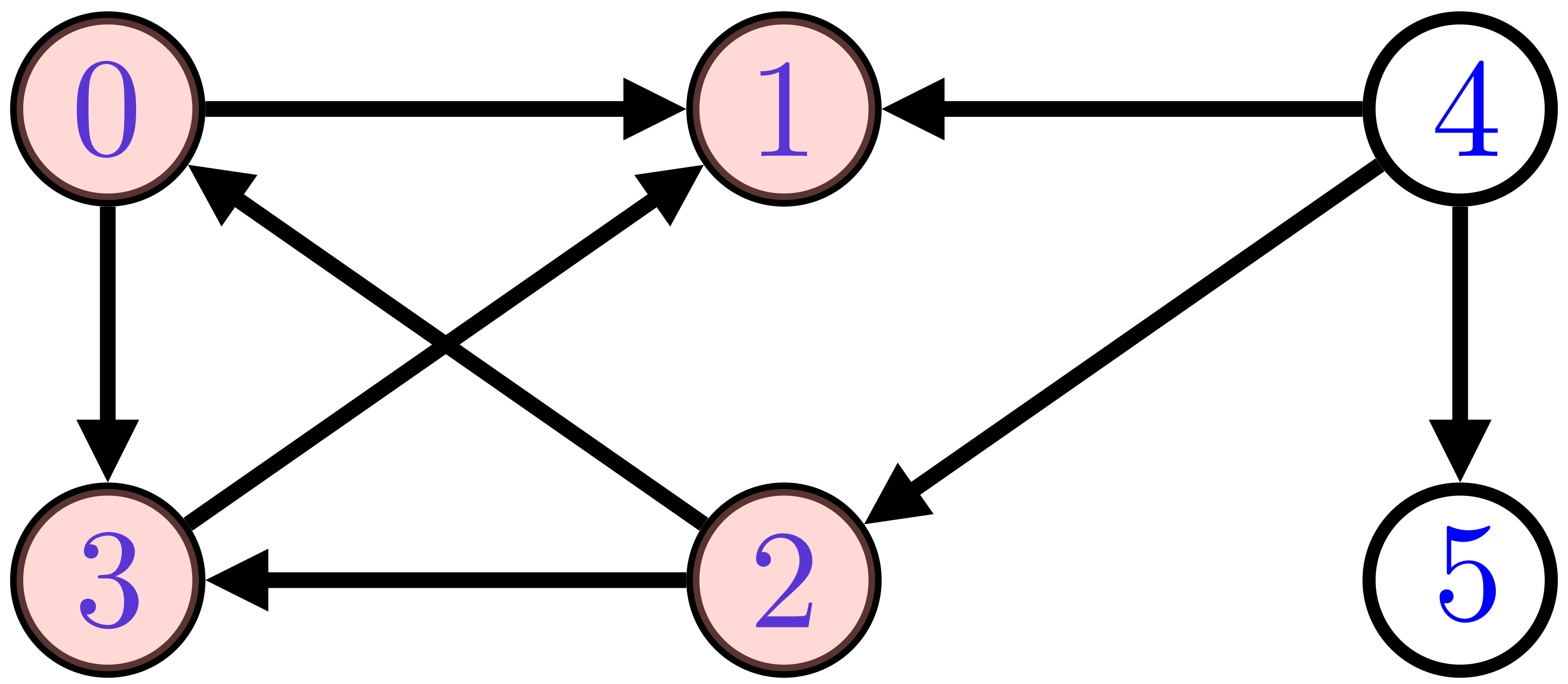
on_stack

0 1 1 3 3 0 2

0: 1 3
1:
2: 3 0
3: 1
4: 5 2 1
5:

Adjacency List

Depth-First Search



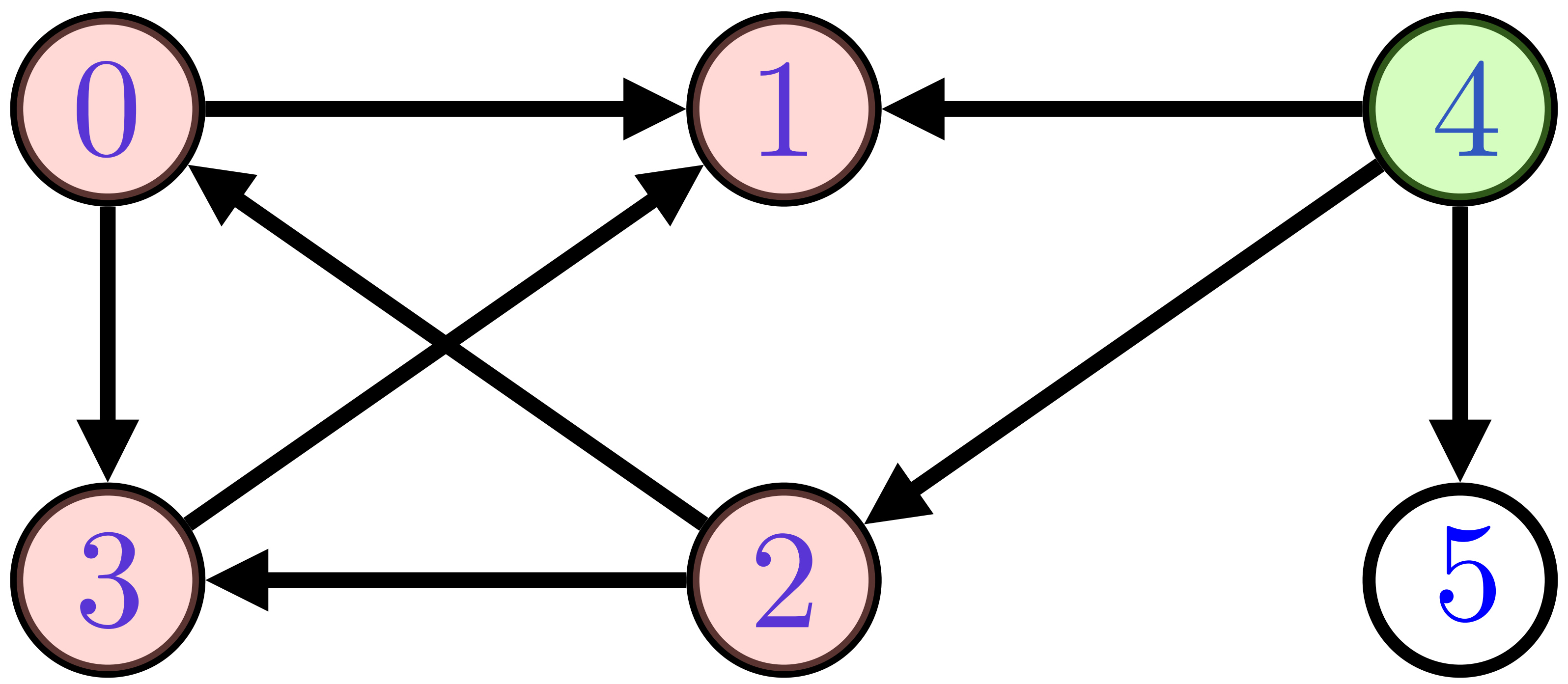
on_stack

0 1 1 3 3 0 2 2

0: 1 3
1:
2: 3 0
3: 1
4: 5 2 1
5:

Adjacency List

Depth-First Search



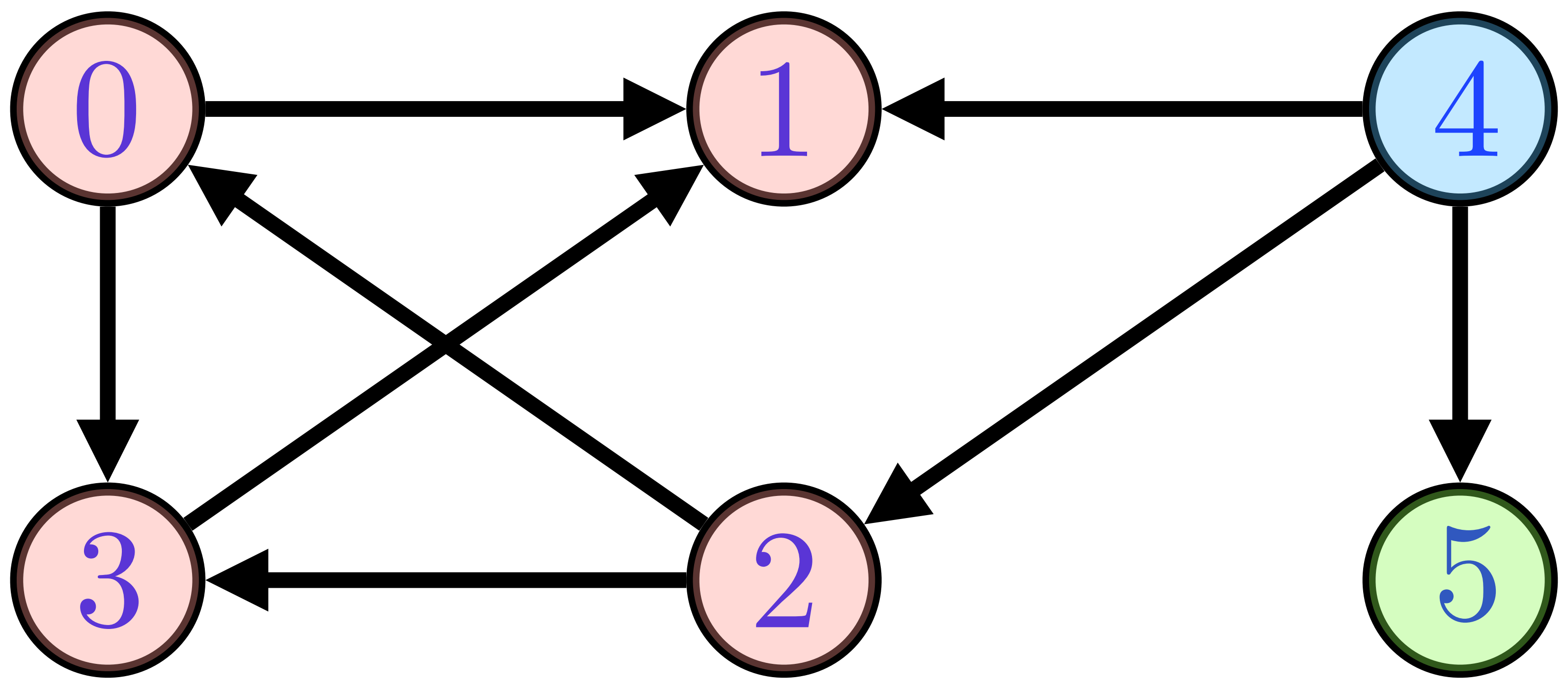
on_stack

0 1 1 3 3 0 2 2 4

0: 1 3
1:
2: 3 0
3: 1
4: 5 2 1
5:

Adjacency List

Depth-First Search



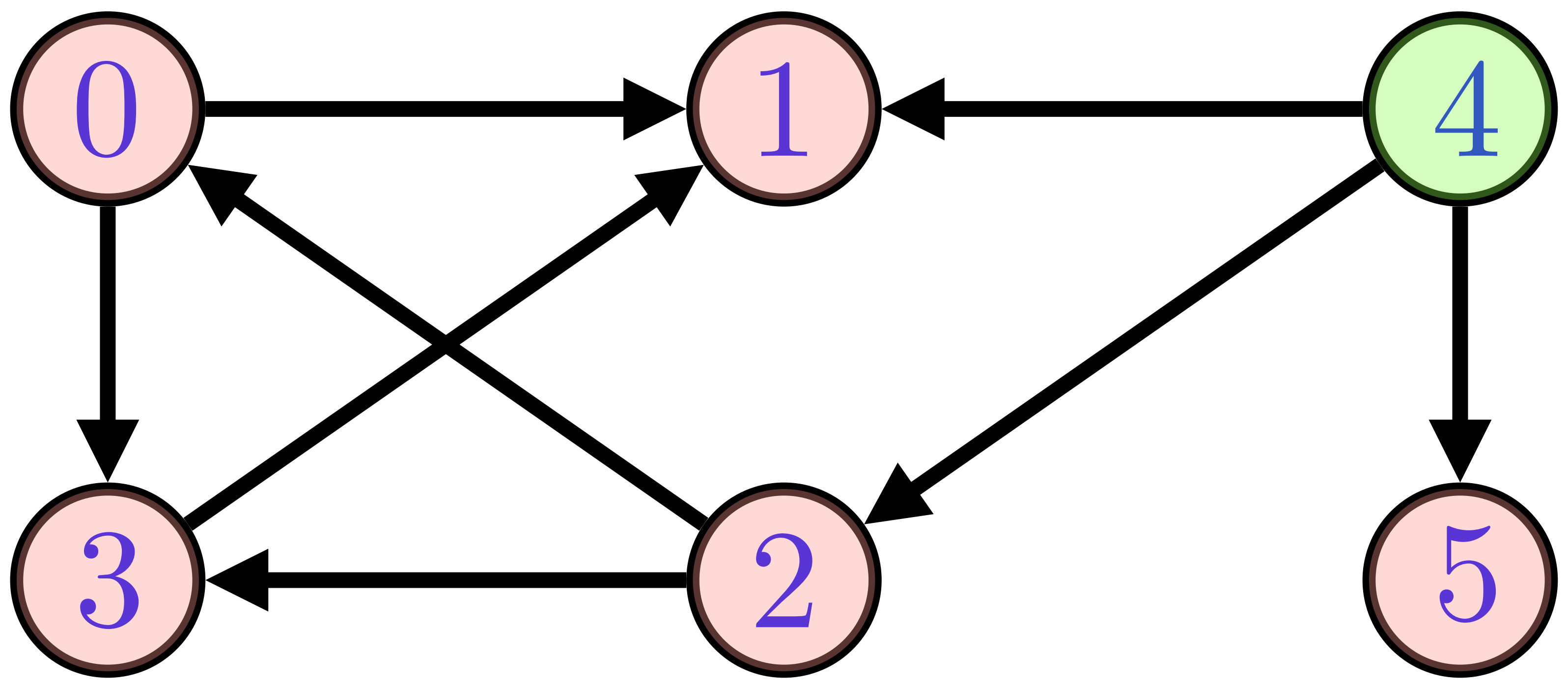
on_stack

0 1 1 3 3 0 2 2 4 5

0: 1 3
1:
2: 3 0
3: 1
4: 5 2 1
5:

Adjacency List

Depth-First Search



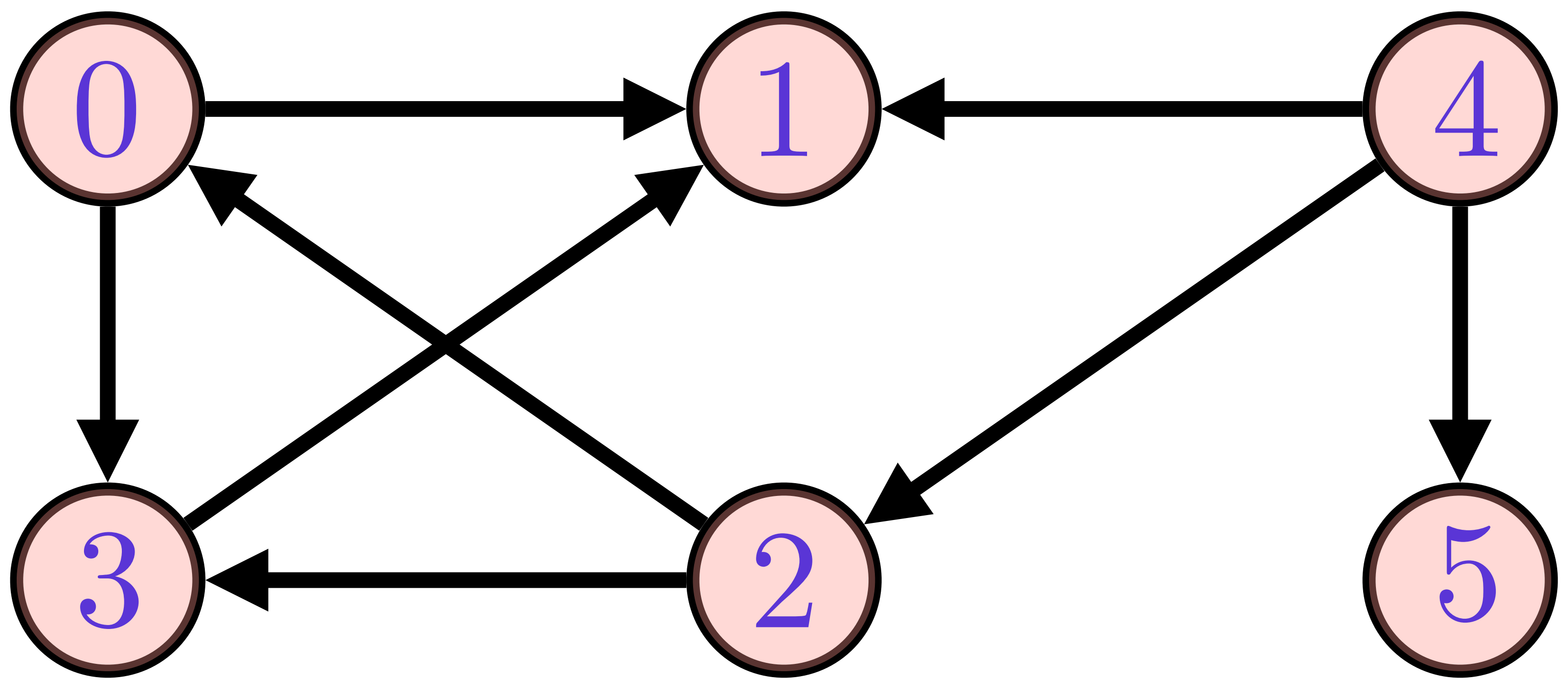
`on_stack`

0 1 1 3 3 0 2 2 4 5 5

0: 1 3
1:
2: 3 0
3: 1
4: 5 2 1
5:

Adjacency List

Final Orderings



0: 1 3
1:
2: 3 0
3: 1
4: 5 2 1
5:

Adjacency List

`on_stack`

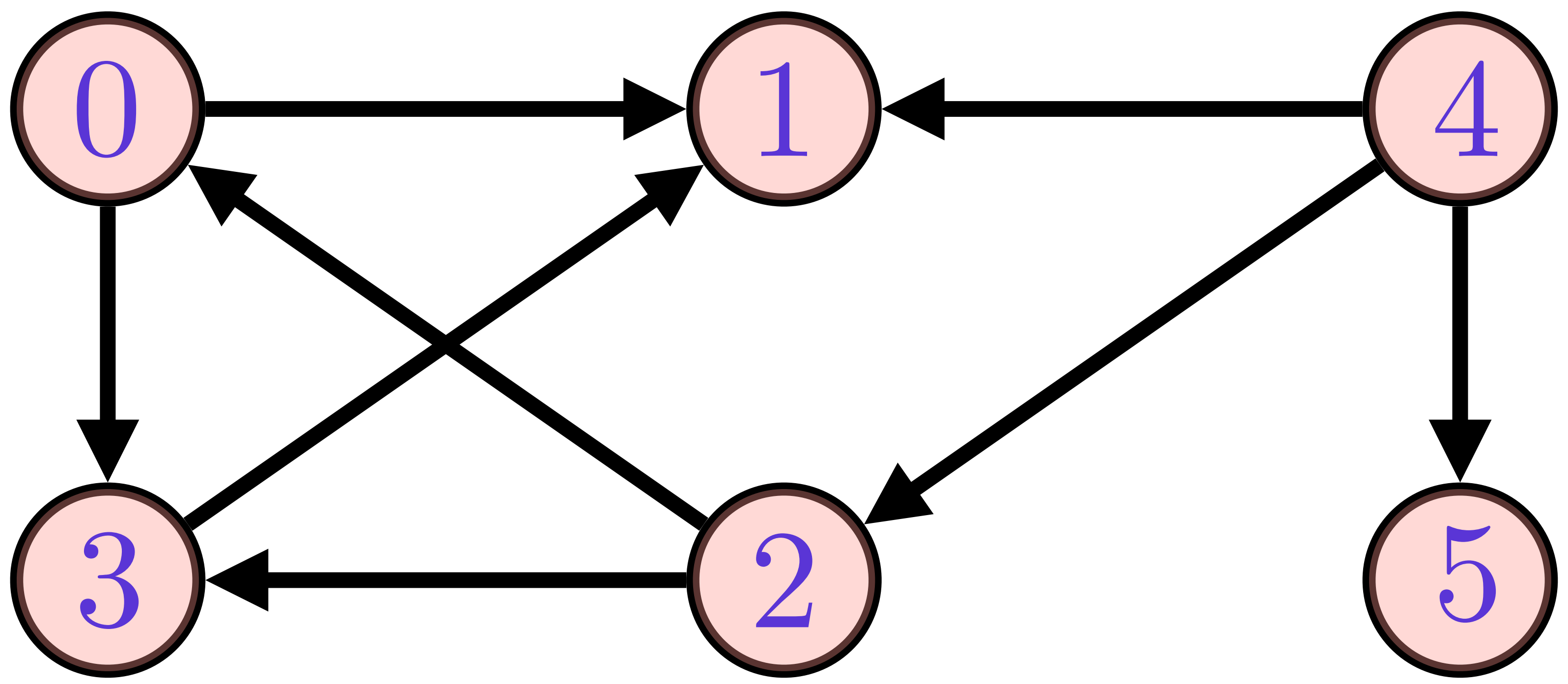
0 1 1 3 3 0 2 2 4 5 5 4

preorder: 0 1 3 2 4 5

postorder: 1 3 0 2 5 4

reverse postorder: 4 5 2 0 3 1

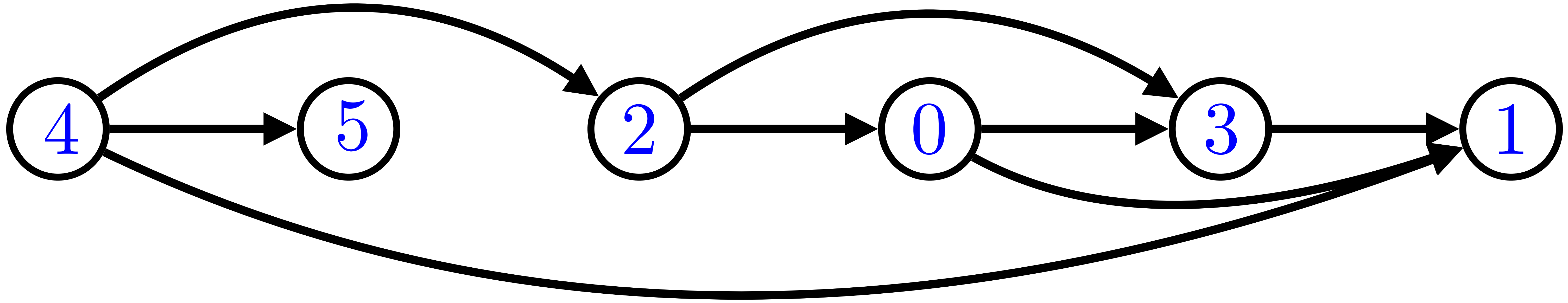
Topological Sort



`on_stack`

0 1 1 3 3 0 2 2 4 5 5 4

reverse postorder: 4 5 2 0 3 1



Reverse Postorder is a Topological Sort in a DAG

Fact: If G is a DAG, then reverse postorder is a topological sort of the vertices.

We need to show that if (u, v) is an edge in a DAG then $\texttt{dfs\_visit}(u)$ finishes after $\texttt{dfs\_visit}(v)$.

This means u comes before v in the reverse postorder and so the topological sort constraint is satisfied.

Fact: If G is a DAG, then reverse postorder is a topological sort of the vertices.

We need to show that if (u, v) is an edge in a DAG then $\texttt{dfs\_visit}(u)$ finishes **after** $\texttt{dfs\_visit}(v)$.

Case 1: $\texttt{dfs\_visit}(u)$ starts **before** $\texttt{dfs\_visit}(v)$.

v is unmarked when $\texttt{dfs\_visit}(u)$ starts.

$\texttt{dfs\_visit}(v)$ will be called during $\texttt{dfs\_visit}(u)$.

$\texttt{dfs\_visit}(v)$ has to terminate before the recursion returns back to $\texttt{dfs\_visit}(u)$.

Fact: If G is a DAG, then reverse postorder is a topological sort of the vertices.

We need to show that if (u, v) is an edge in a DAG then $\texttt{dfs\_visit}(u)$ finishes **after** $\texttt{dfs\_visit}(v)$.

Case 2: $\texttt{dfs\_visit}(v)$ starts **before** $\texttt{dfs\_visit}(u)$.

Case 2a: $\texttt{dfs\_visit}(v)$ finishes before $\texttt{dfs\_visit}(u)$ starts.

Fact: If G is a DAG, then reverse postorder is a topological sort of the vertices.

We need to show that if (u, v) is an edge in a DAG then $\texttt{dfs\_visit}(u)$ finishes **after** $\texttt{dfs\_visit}(v)$.

Case 2: $\texttt{dfs\_visit}(v)$ starts **before** $\texttt{dfs\_visit}(u)$.

Case 2b: $\texttt{dfs\_visit}(u)$ starts before $\texttt{dfs\_visit}(v)$ finishes.

Then (u, v) is a back edge, which cannot happen in a DAG.

Summary

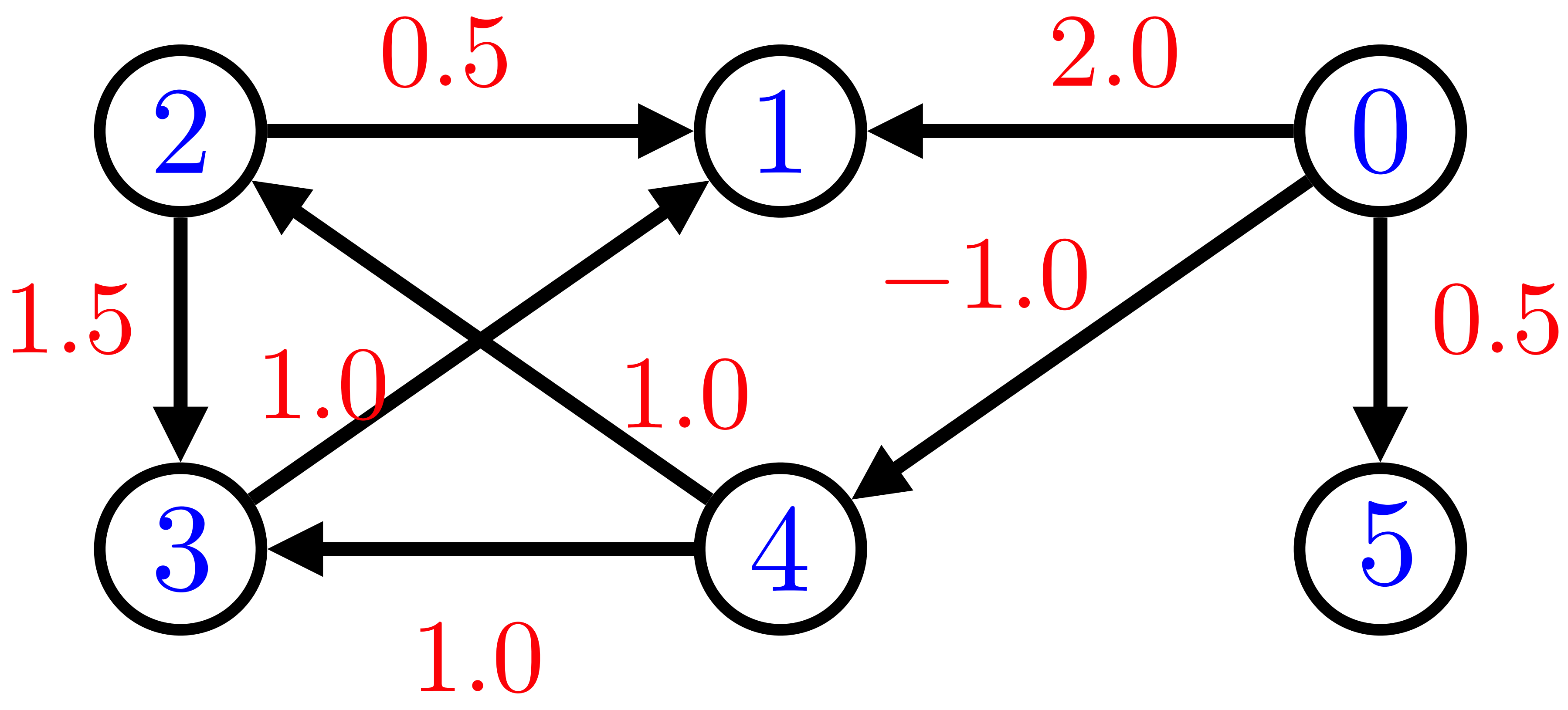
Fact: A graph G has a topological sort iff it is a DAG.

A topological sort of a DAG is given by a reverse postorder of the vertices from depth-first search.

We can find a topological sort in time $O(|V| + |E|)$ in the adjacency list model.

Shortest Paths in a DAG

Shortest Paths in a DAG

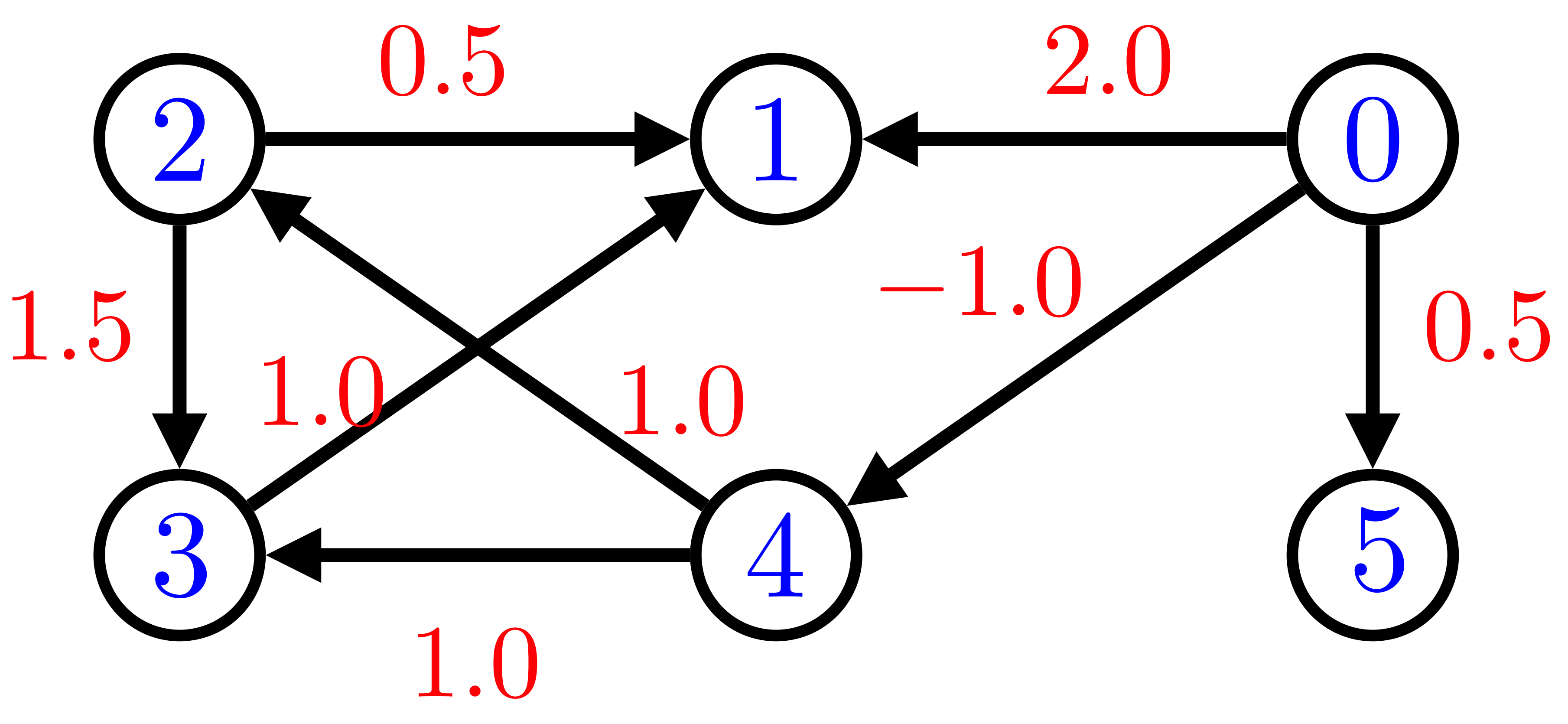


Let's say we want to solve the single-source shortest path problem in this DAG.

We can allow negative edge weights as we know there will be no negative-weight cycles as there are no cycles as all.

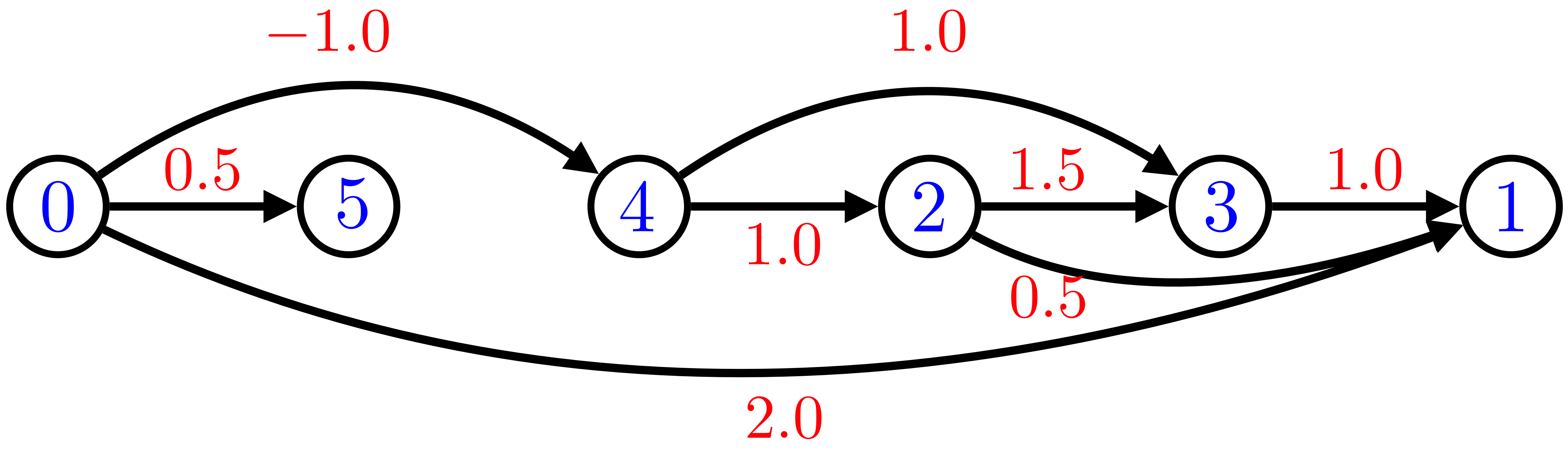
This is a particularly nice case on which to instantiate the generic shortest path algorithm.

Shortest Paths in a DAG



Step 1: Compute a topological sort of the graph.

This is an ordering of the vertices such that $u < v$ for every edge (u, v) .



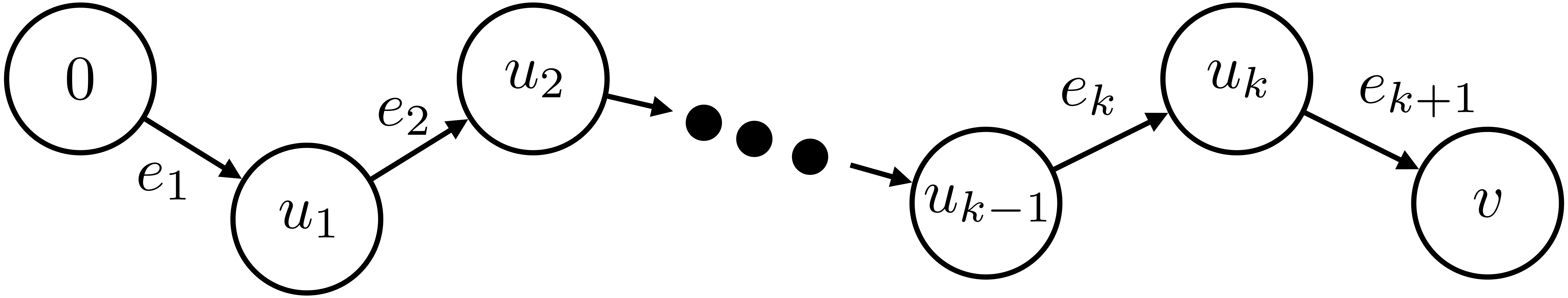
Algorithm

```
for(auto v : topo_order) {
    for(const auto& edge : adj_list[v]) {
        relax(edge);
    }
}
```

We can compute a topological order by depth-first search in time $O(|V| + |E|)$ in the adjacency list model.

The overall running time is $O(|V| + |E|)$ in the adjacency list model.

Why it Works



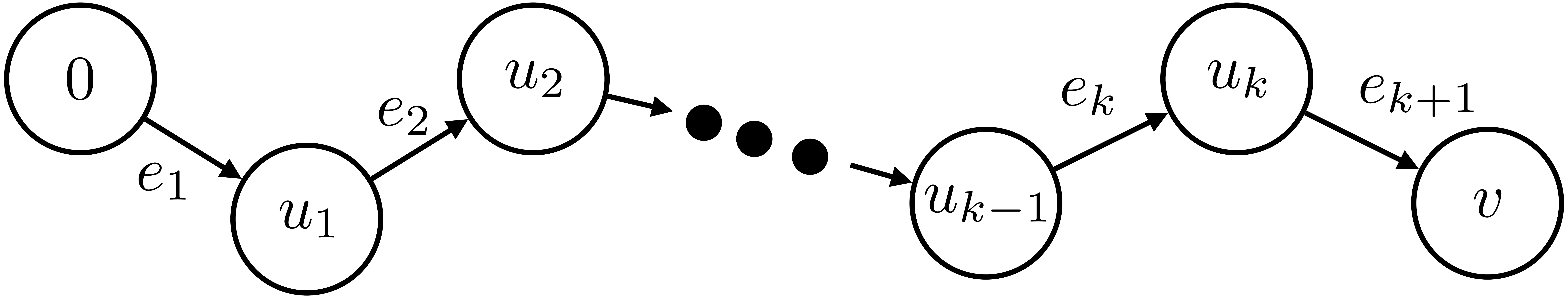
If this is a shortest path from 0 to vertex v we know that

$$0 < u_1 < u_2 < \cdots < u_k < v$$

in the topological order.

By relaxing the outgoing edges of vertices in topological order we relax e_1 before e_2 before e_3 etc. all the way to e_{k+1}.

Why it Works



By relaxing the outgoing edges of vertices in topological order we relax this path.

Relax a Path Property: If the algorithm relaxes a shortest path from 0 to then `dist_to`$[v] = d(0, v)$.