

Arborescent Garbage Collection

A Dynamic Graph Approach to Immediate Cycle Collection

Frédéric Lahaie-Bertrand, **Léonard Oest O'Leary**, Olivier Melançon,
Marc Feeley and Stefan Monnier

@ International Symposium on Memory Management 2025 (ISMM'25)

Collecting cycles immediately

Most existing technique either collect cyclic garbage **or** collect garbage immediately, **not both**:

Tracing

- Can collect cycles but **not** immediately
- Ex: Mark-and-sweep, Stop-and-copy

Reference Counting

- Can collect garbage immediately, **except** cyclic structure

Here **immediate** means that the garbage is collected **before any other operation by the mutator**.

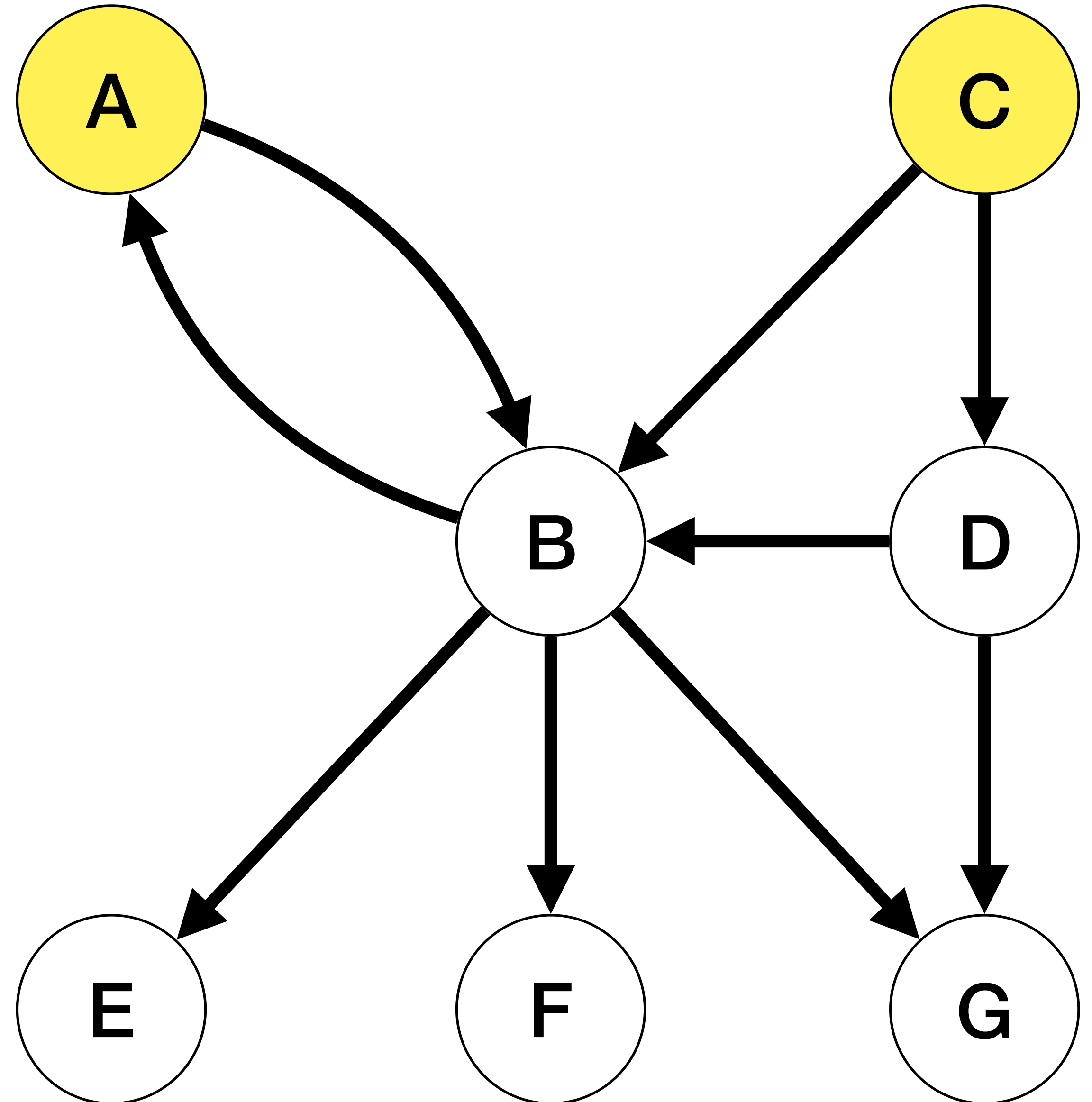
Motivations

Immediate cycle collection is useful in applications such as:

- File-systems and object stores.
- The interoperability of a garbage collector with C++ or Rust.
- Reclamation of DOM objects in a browser.
- Precise measurement of the maximum amount of live heap data during a program execution.

Our approach

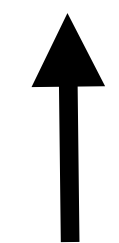
Given a **reference graph** between objects in the heap



Our approach

Given a **reference graph** between objects in the heap

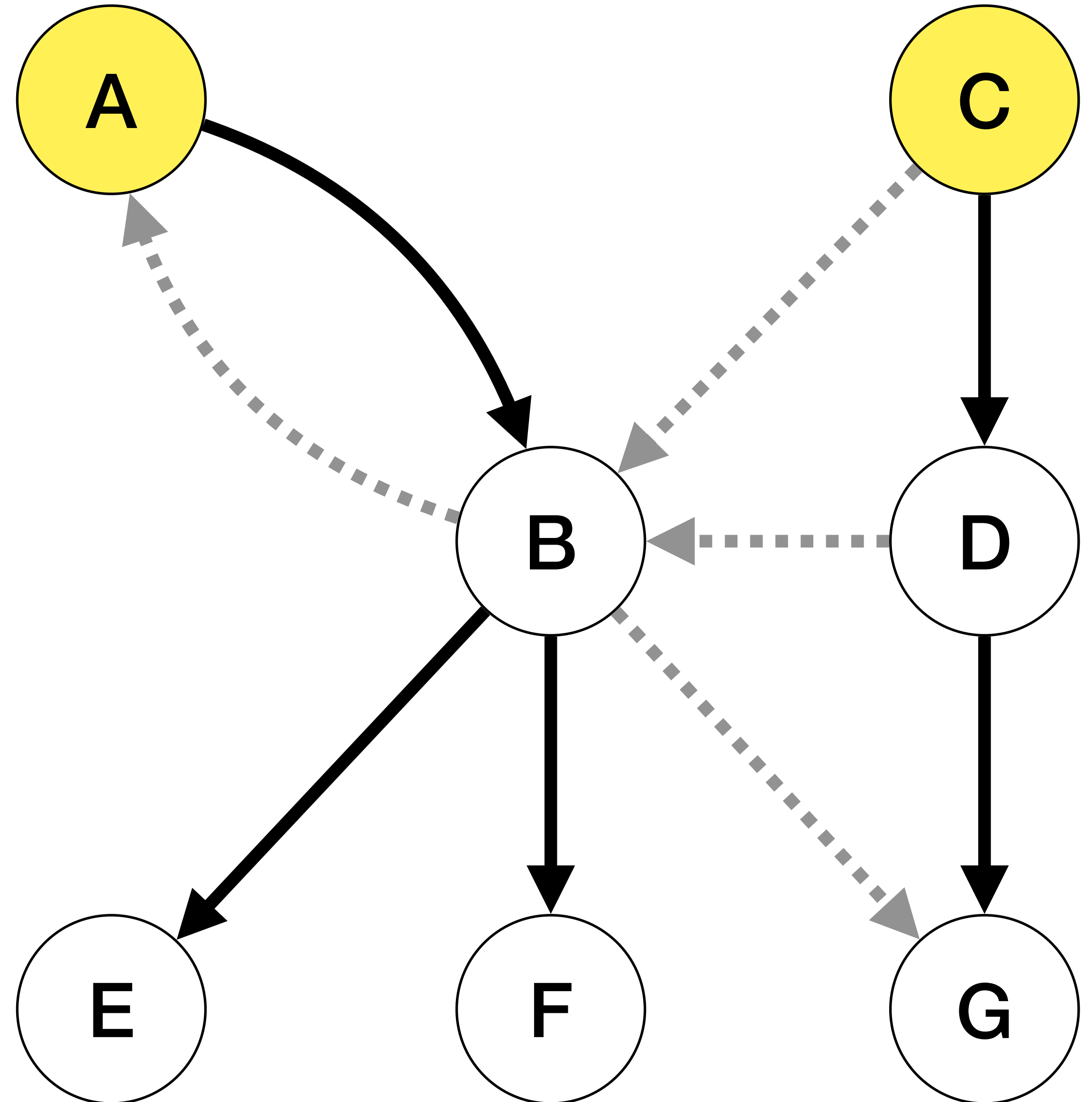
The **Garbage collection problem** can be formulated as **finding a collection of spanning trees**.



Part of the spanning tree



Part of the reference graph

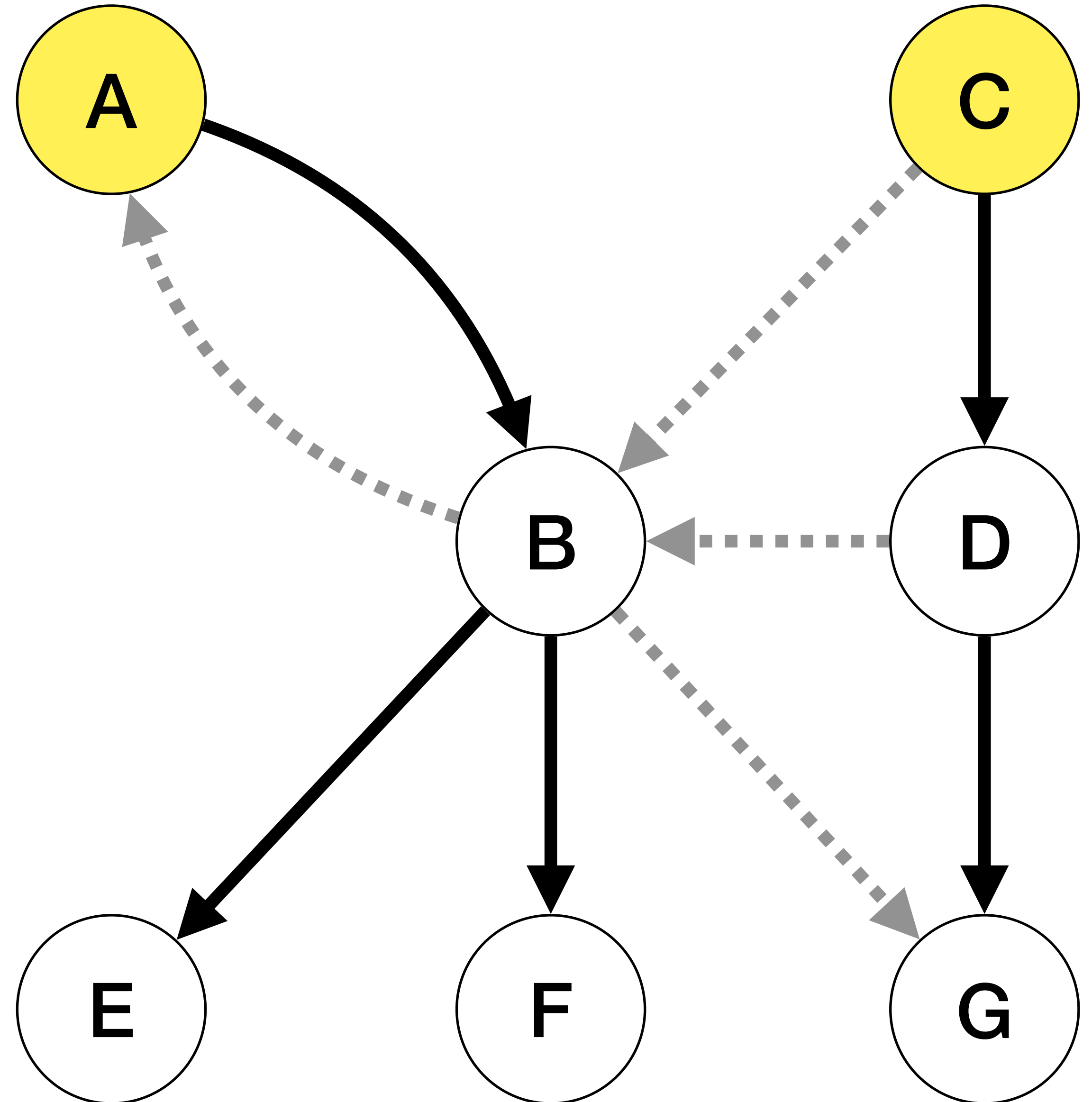


Our approach

Given a **reference graph** between objects in the heap

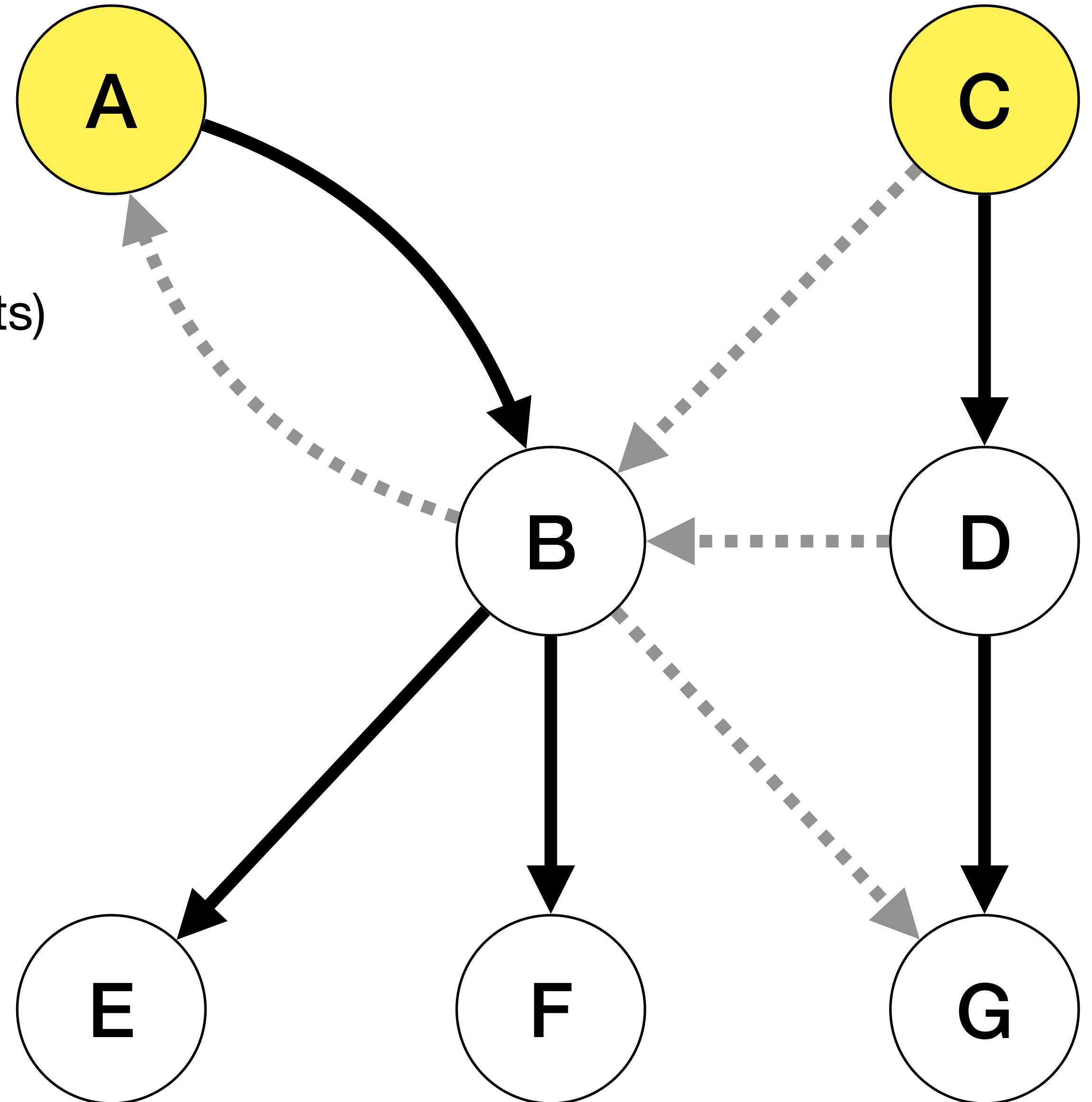
The **Garbage collection problem** can be formulated as **finding a collection of spanning trees**.

Our approach will be to **maintain** this spanning forest at all times during the execution of the program



Some definitions

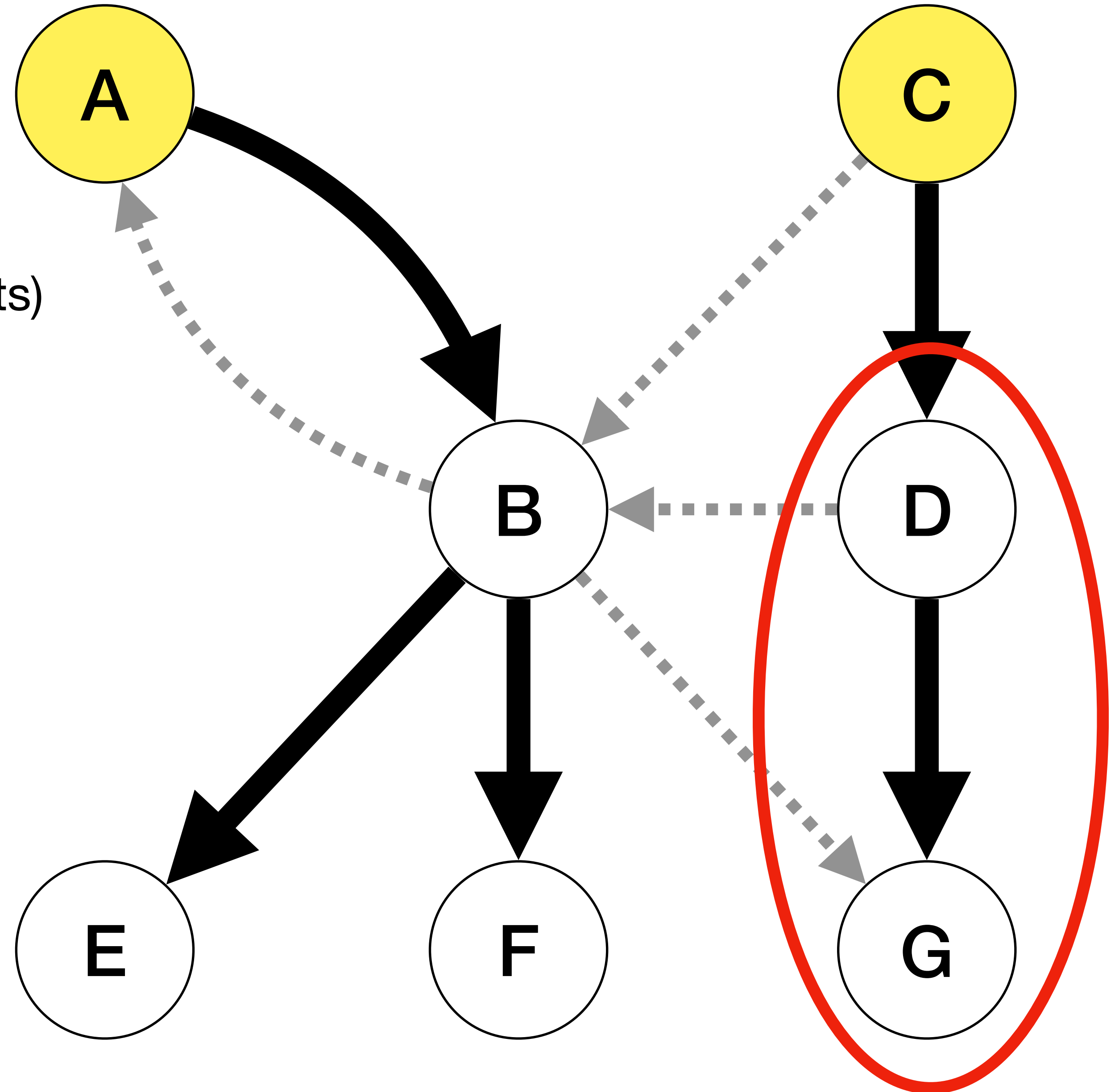
A and **C** are uncollectable nodes (or roots)



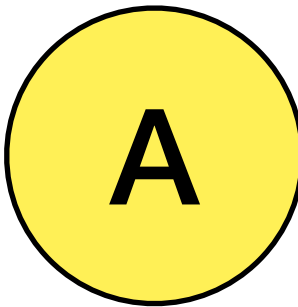
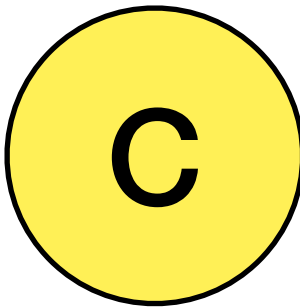
Some definitions

A and **C** are uncollectable nodes (or roots)

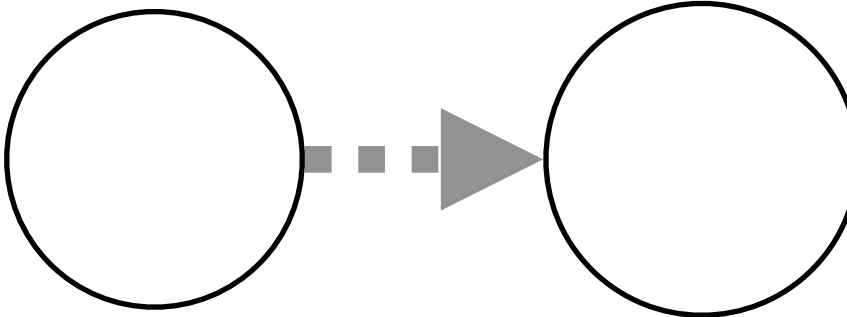
 denotes a parent/child relation

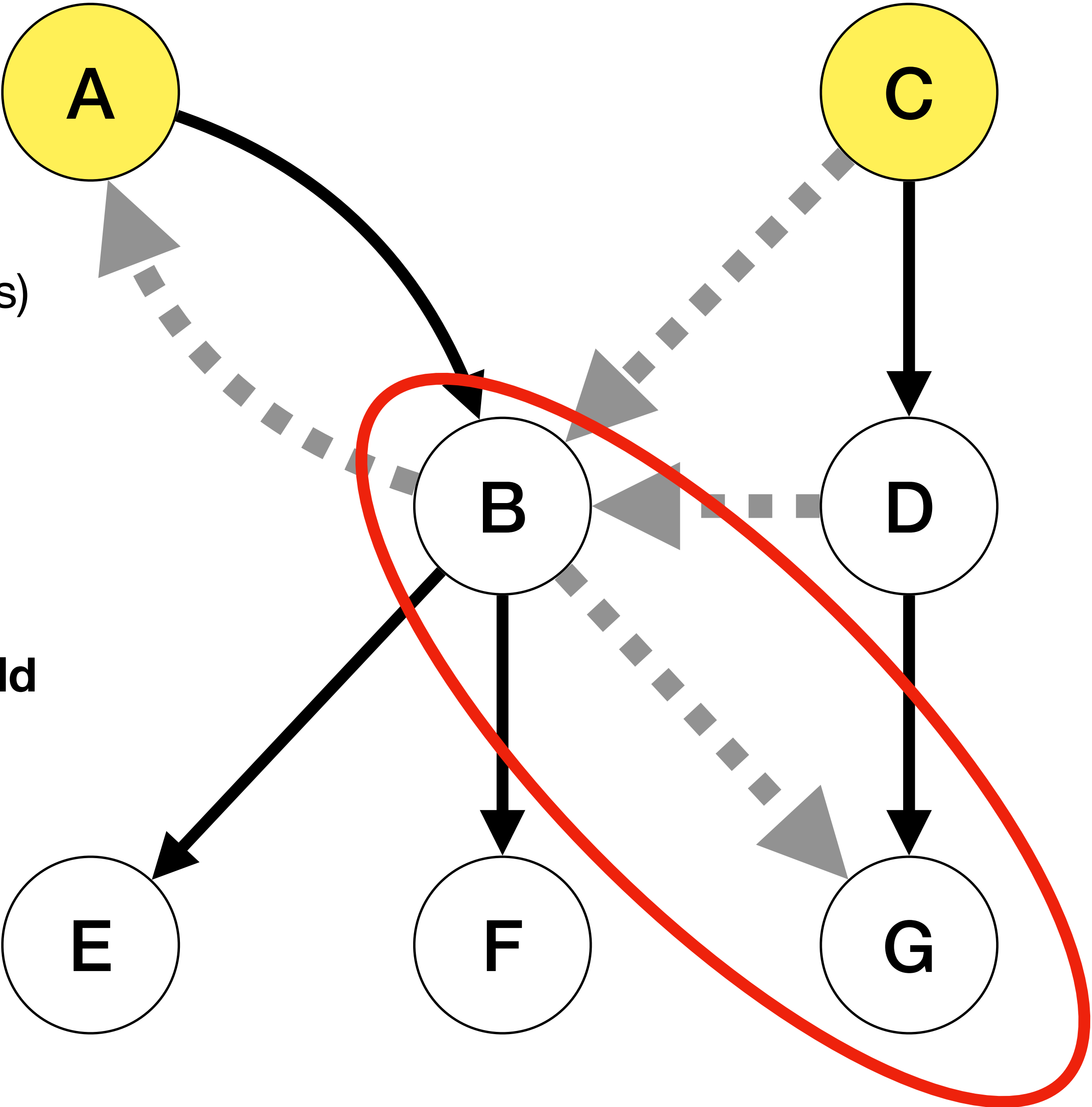


Some definitions

 and  are uncollectable nodes (or roots)

 denotes a parent/child relation

 denotes a **coparent and cochild** relationship



Even Shiloach trees

The goal of Even Shiloach trees is to maintain a **minimal** spanning forest.

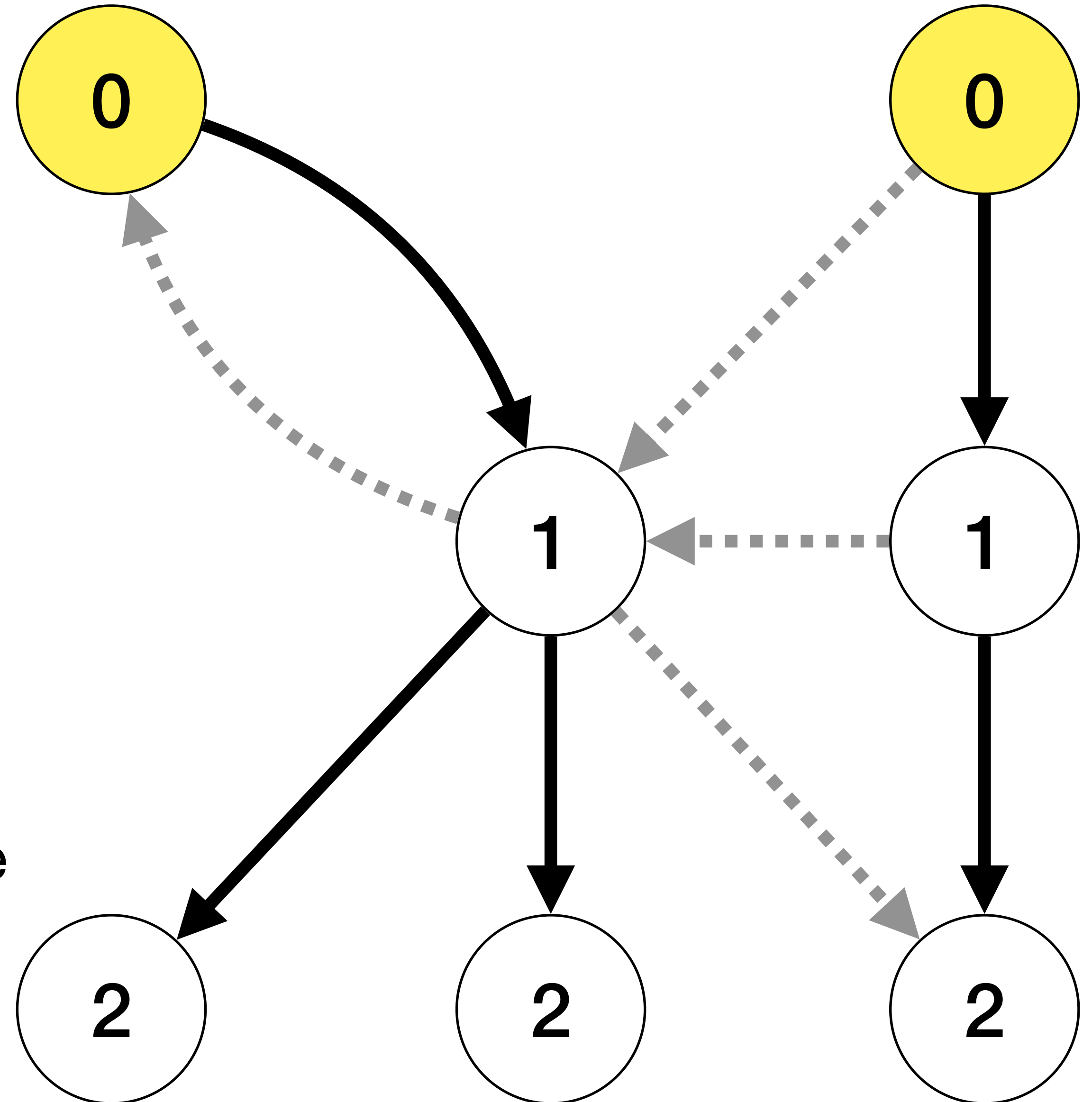
The Arborescent Garbage Collector was inspired by **Even Shiloach trees**.

Even Shiloach trees

The goal of Even Shiloach trees is to maintain a **minimal** spanning forest.

The Arborescent Garbage Collector was inspired by **Even Shiloach trees**.

Even Shiloach trees keep this distance by adding a notion of **rank**.



Even Shiloach trees

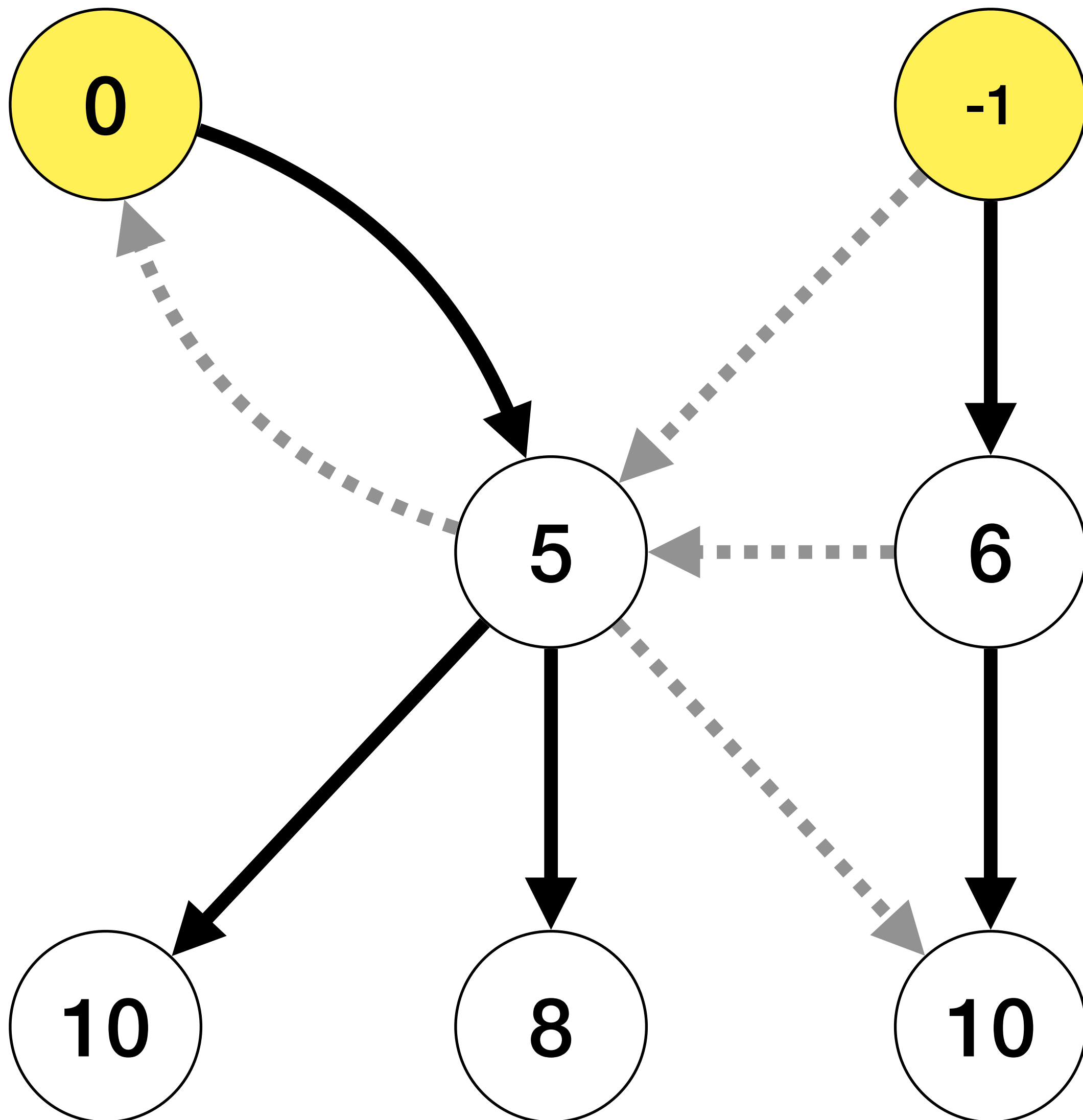
However a **minimal** spanning forest is not needed to maintain reachability.

Even Shiloach

- Maintains a **minimal** spanning forest
- Ranks are **strictly monotone**
- Ranks are needed for the algorithm to work

Arborescent Garbage Collector

- Maintains **a** spanning forest
- Ranks must be increasing
- Ranks are only needed to ensure no cycles are formed when optimizing

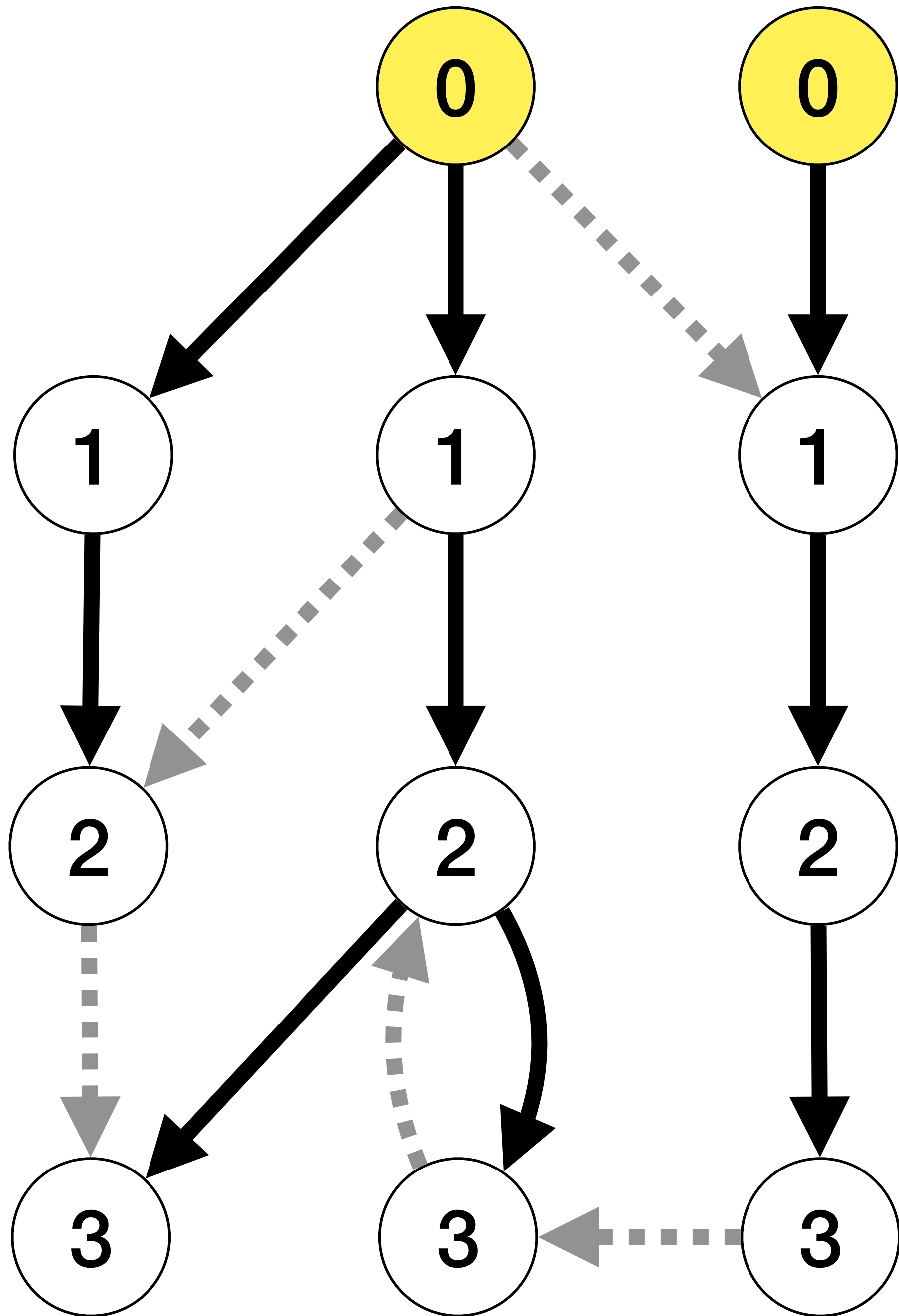


not needed to maintain reachability.

Arborescent Garbage Collector

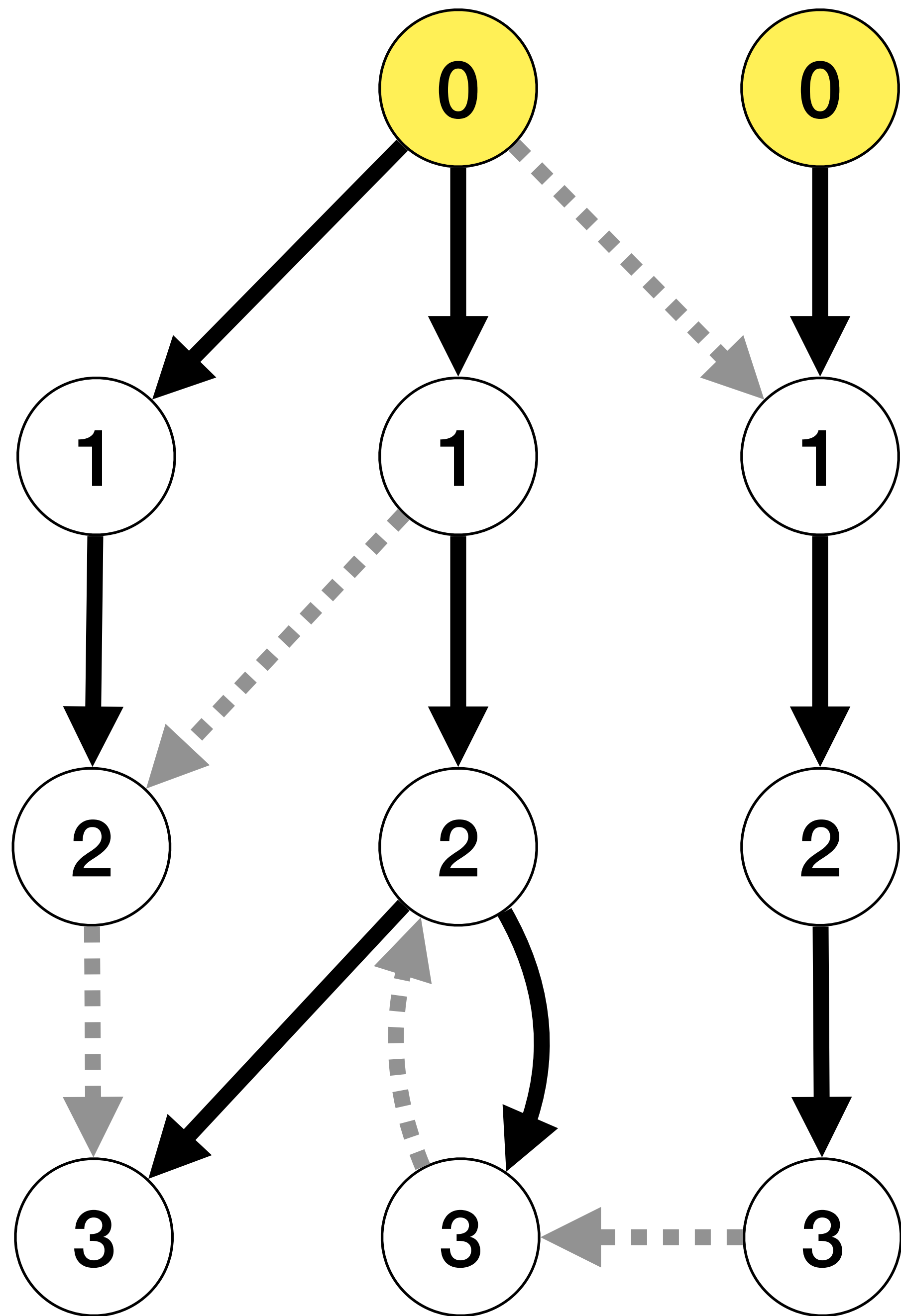
- Maintains a spanning forest
- **Ranks must be increasing**
- Ranks ensure no cycles are formed when optimizing

The Arboresecent Garbage Collection Algorithm

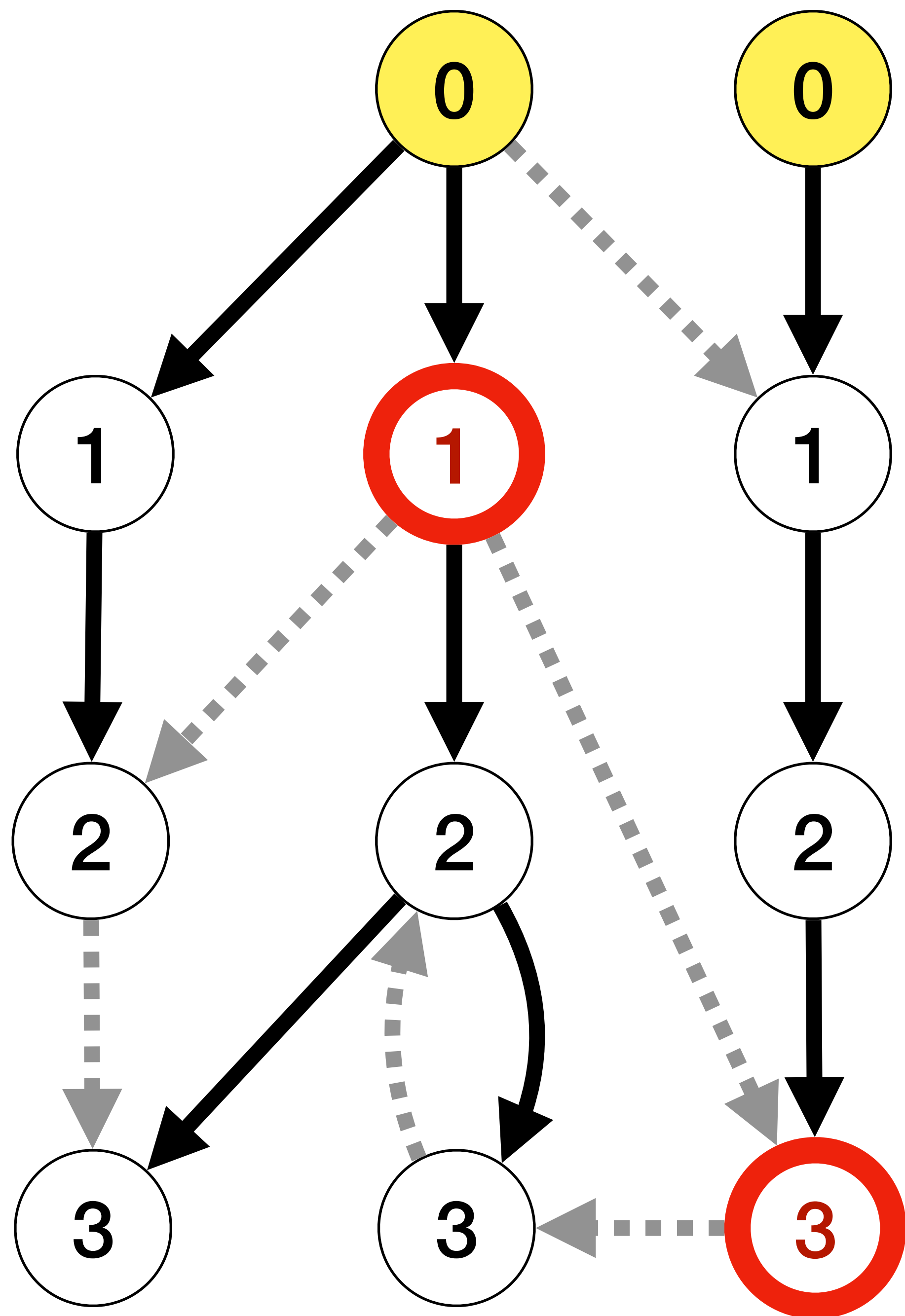


Our goal is to maintain a spanning tree over:

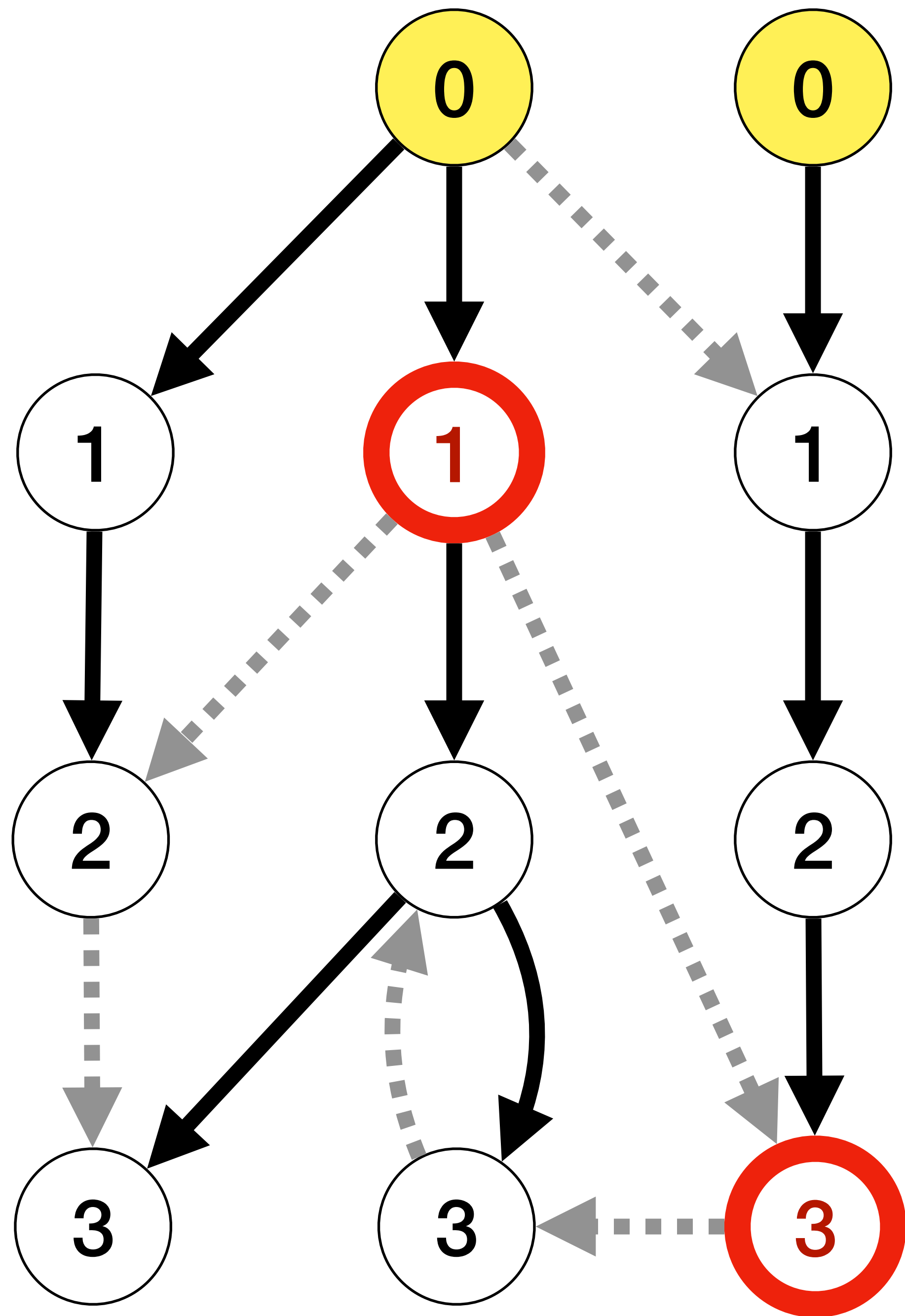
- The **addition** of an edge
- The **removal** of an edge **outside** of the spanning forest
- The **removal** of an edge **inside** of the spanning forest



Addition of an edge

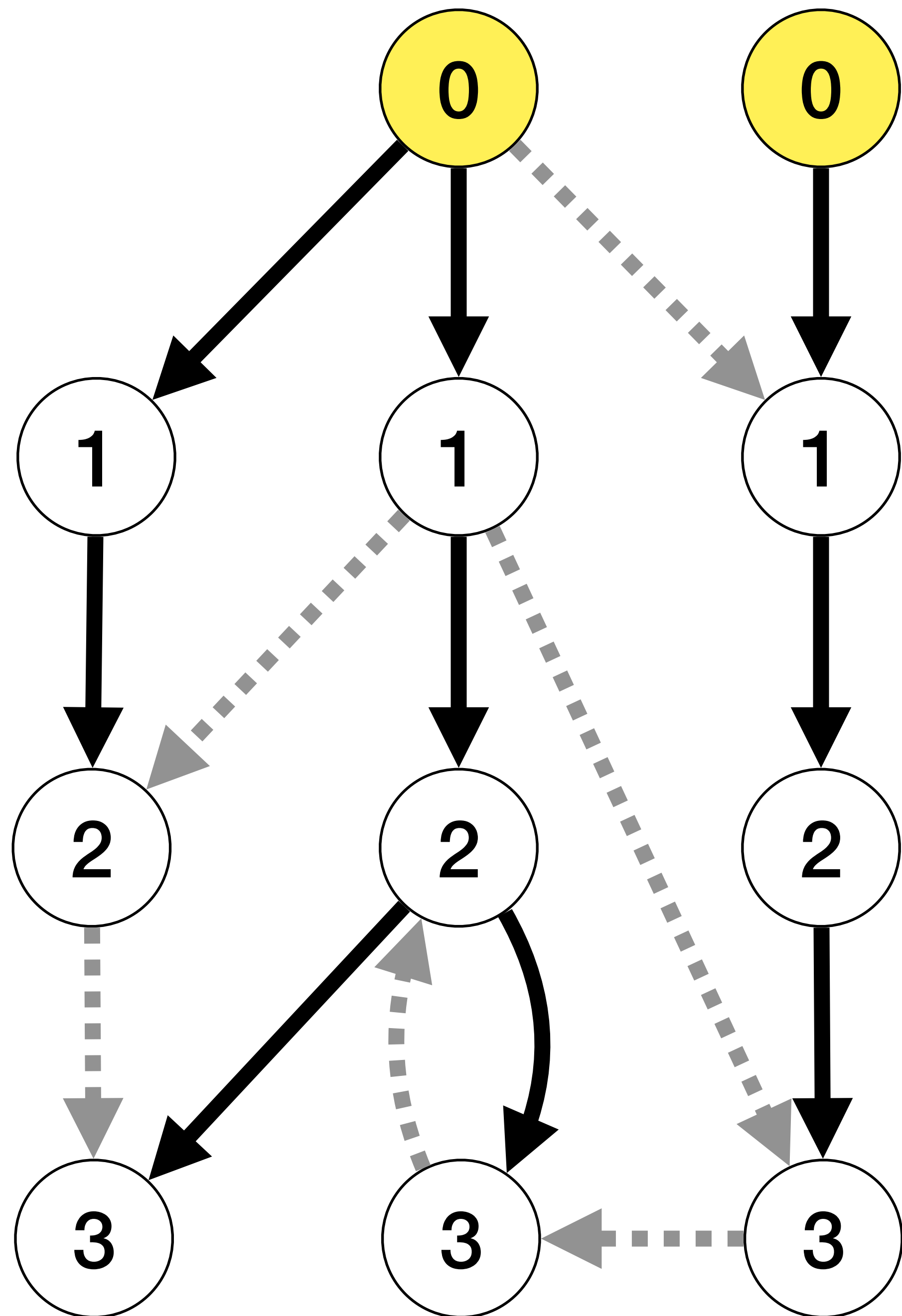


Addition of an edge



Addition of an edge

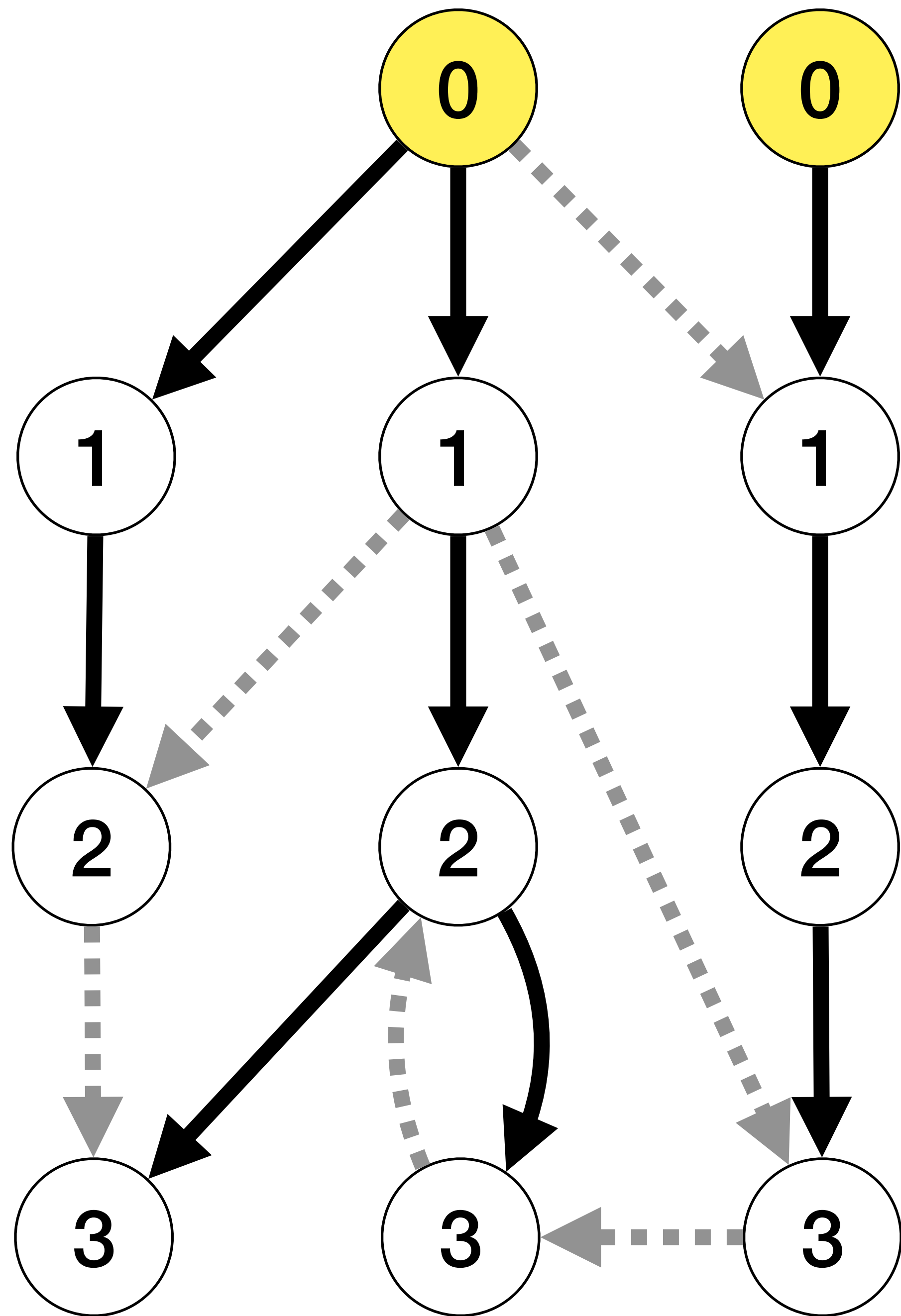
We can simply add an edge that is not part of the spanning tree.



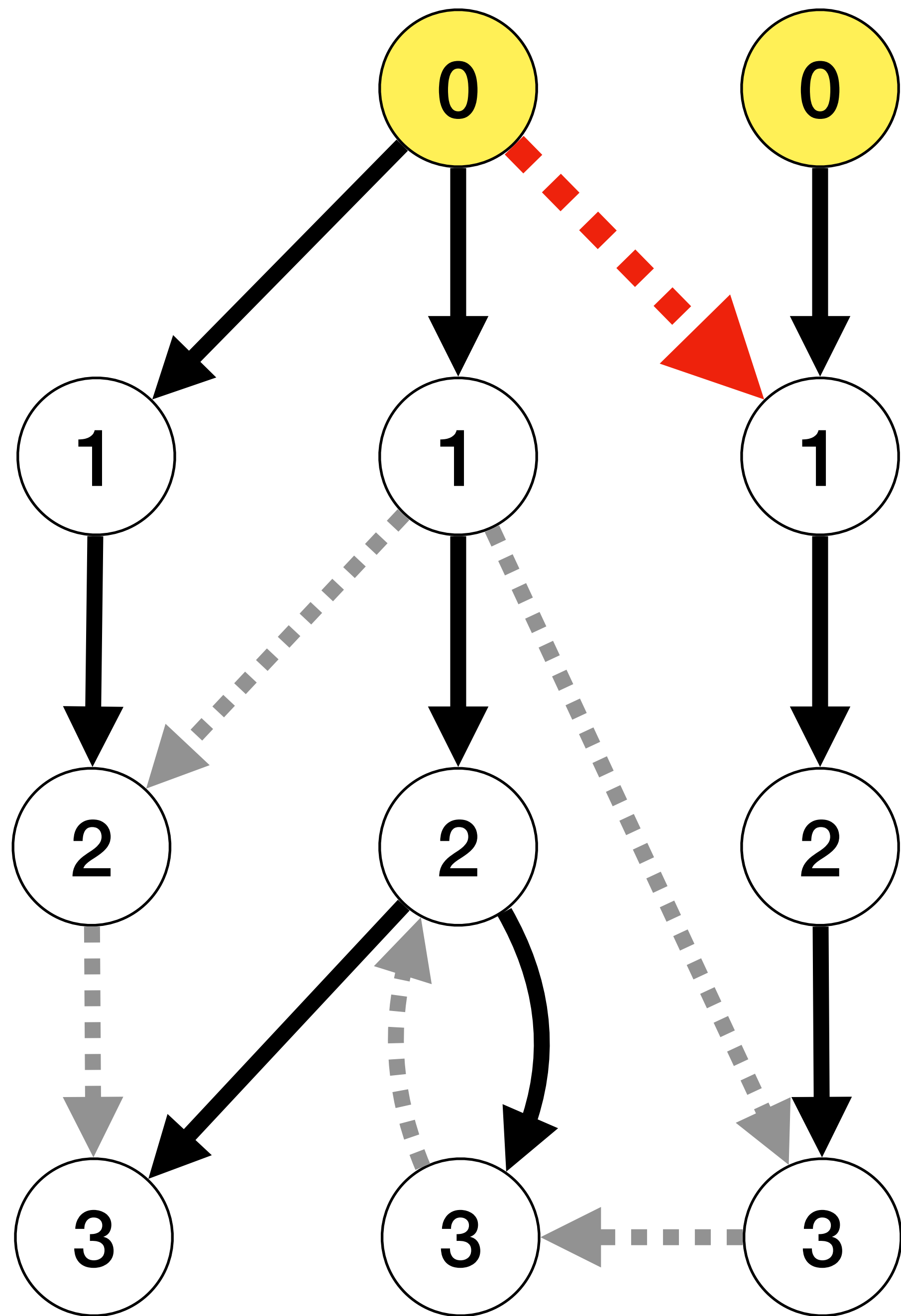
Addition of an edge

We can simply add an edge that is not part of the spanning tree.

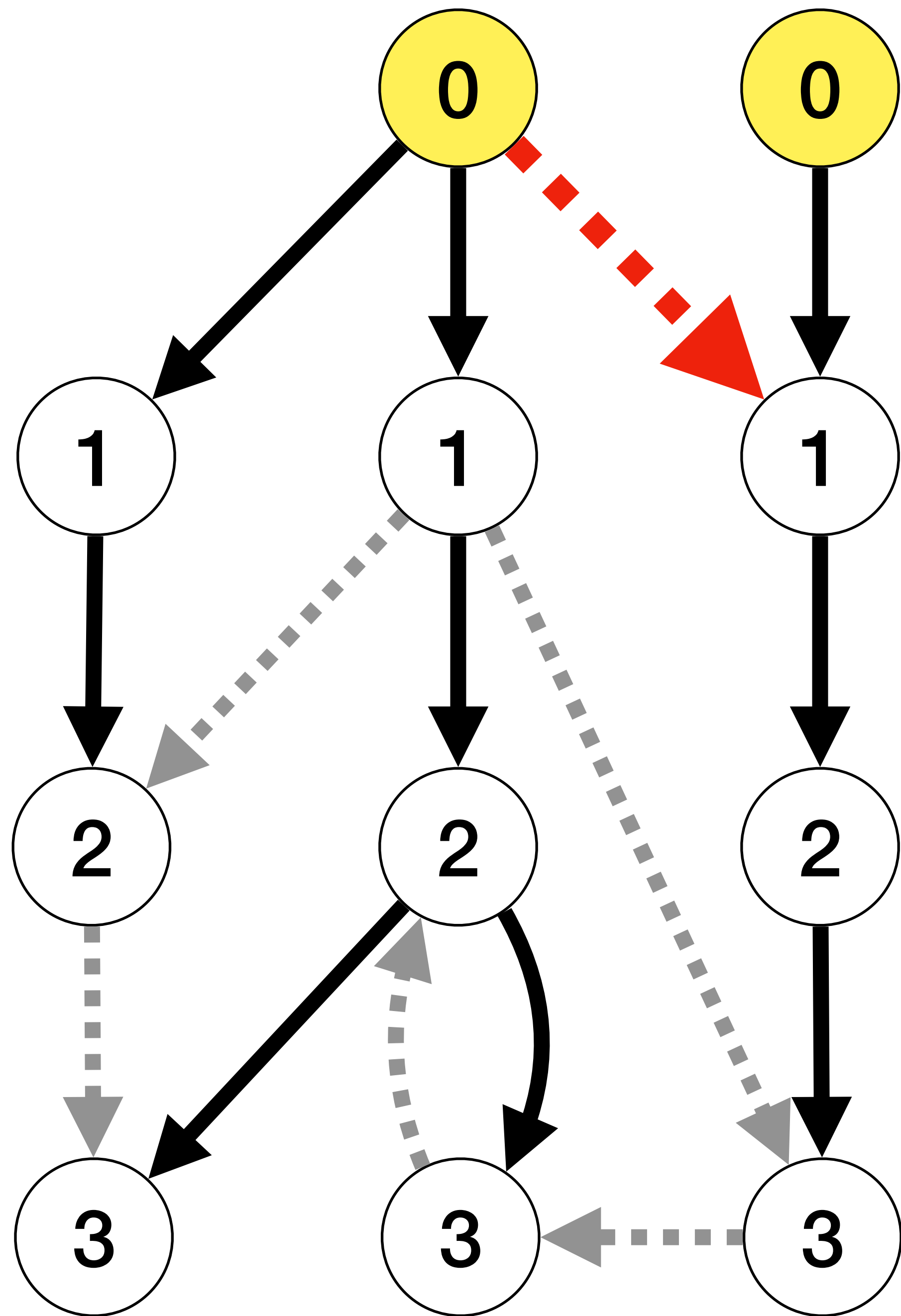
... still a spanning forest



Removal of an edge

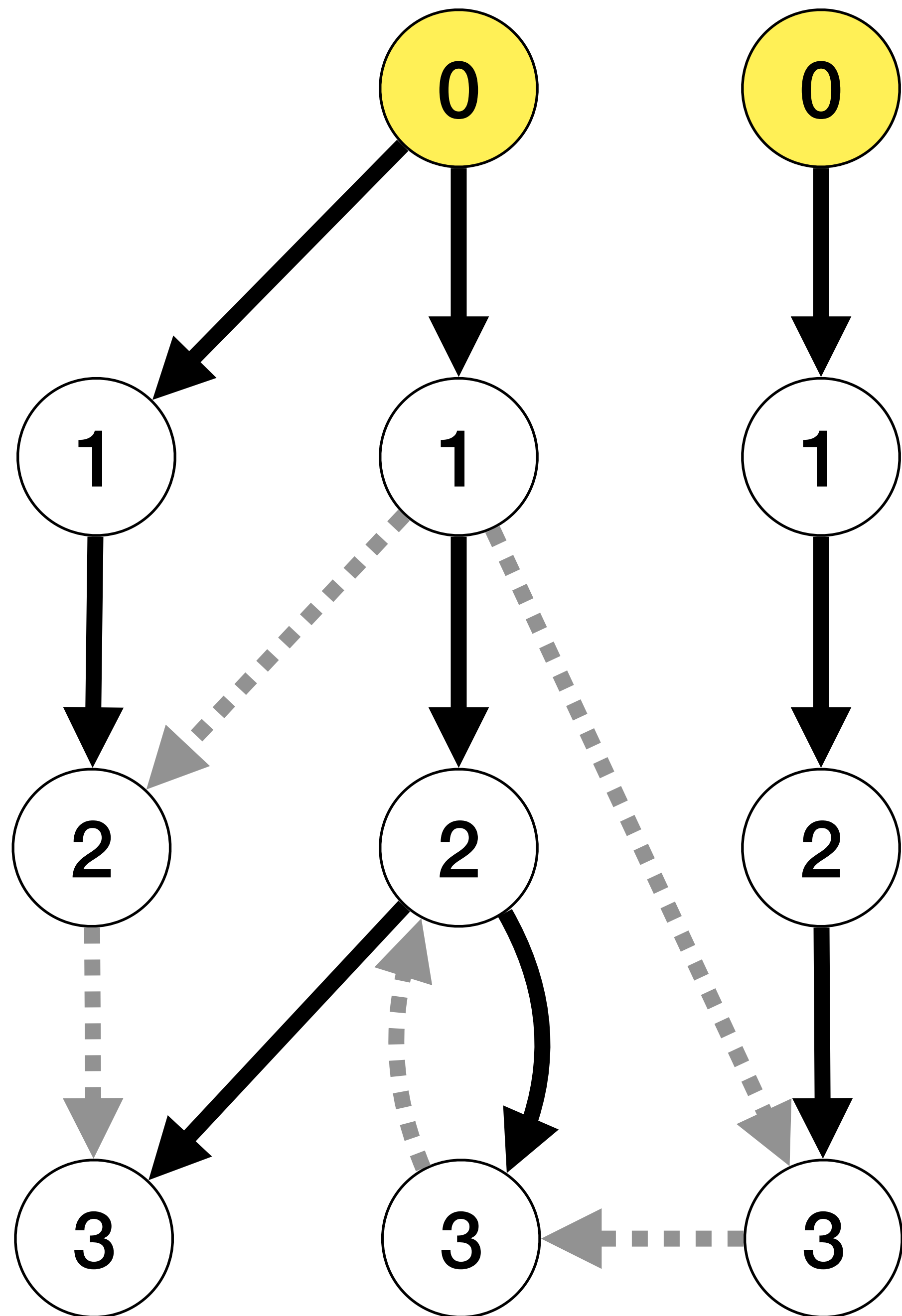


Removal of an edge



Removal of an edge

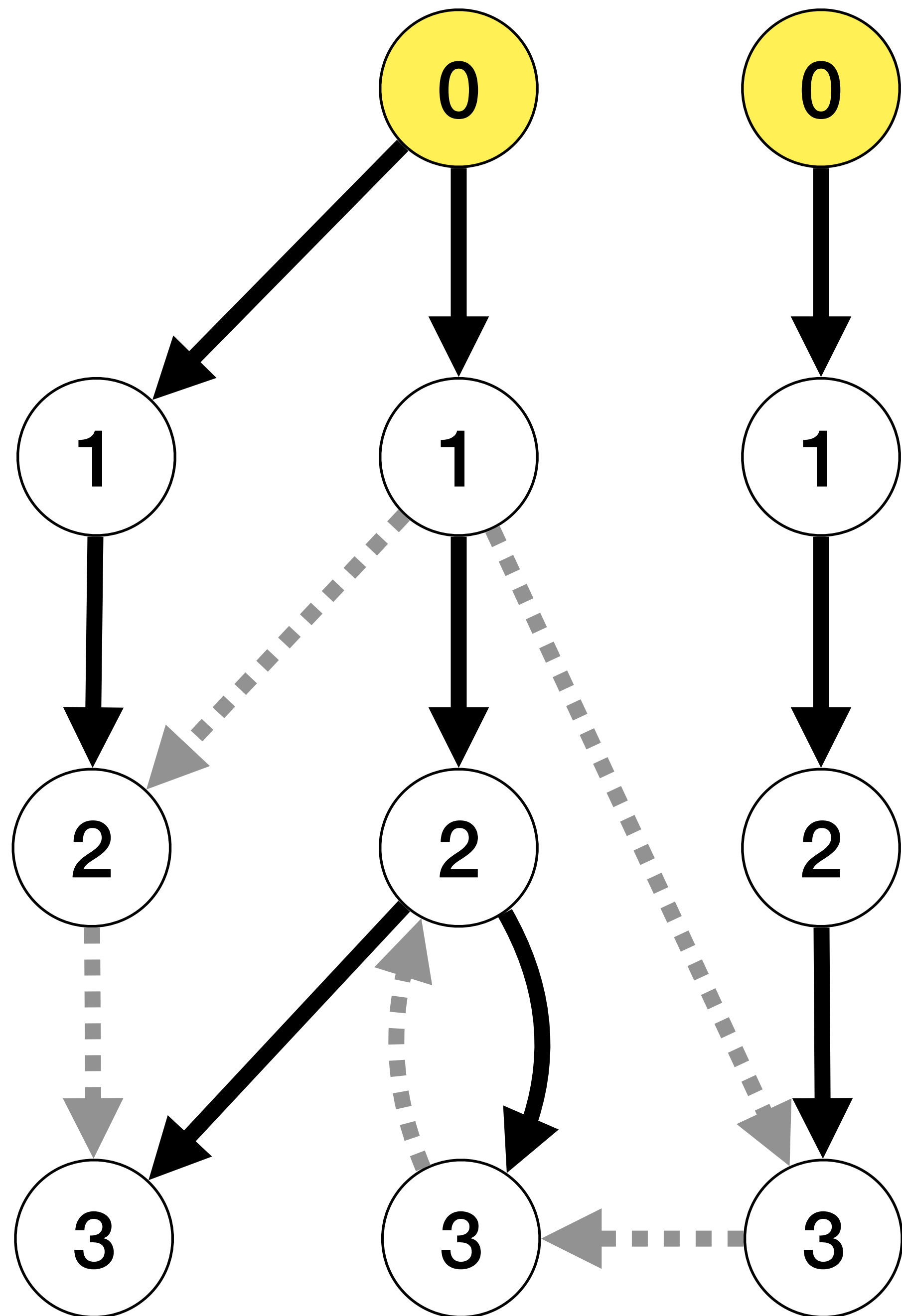
If the edge is not part of the spanning tree, we can just remove it.



Removal of an edge

If the edge is not part of the spanning tree, we can just remove it.

... still a spanning forest

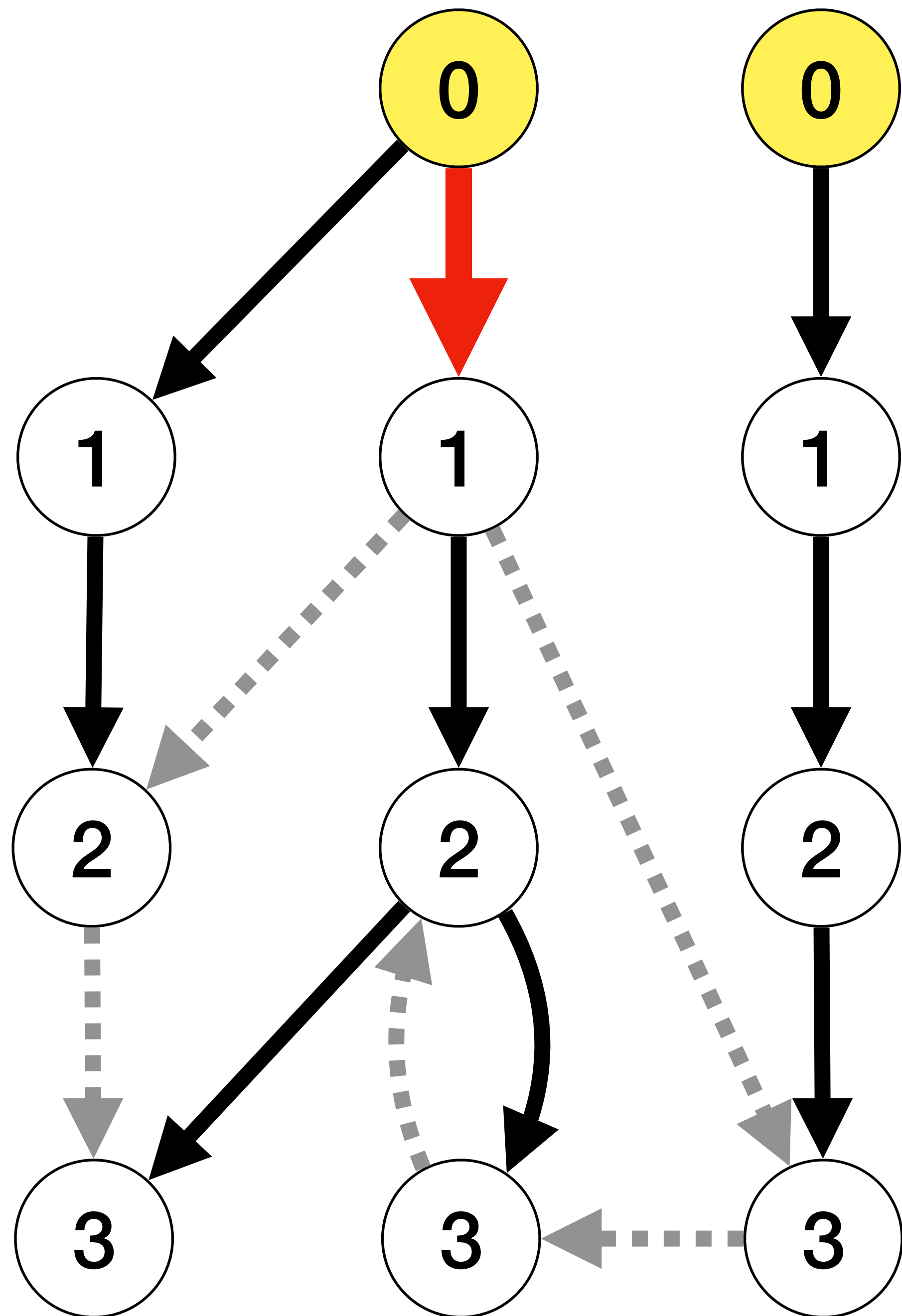


Removal of an edge

If the edge is not part of the spanning tree, we can just remove it.

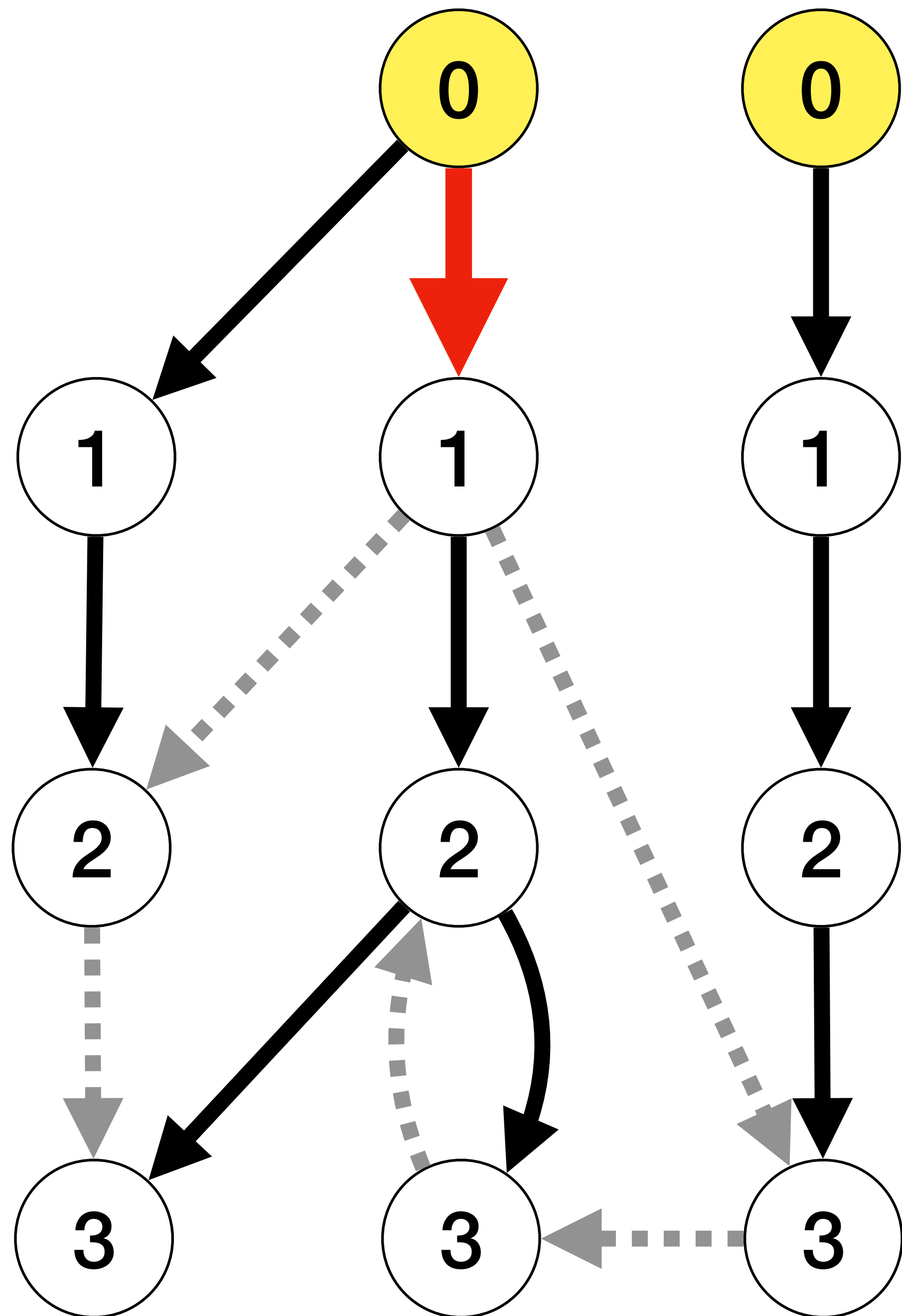
... still a spanning forest

However, if the edge is part of the spanning tree... its more complicated.



Removal of an edge

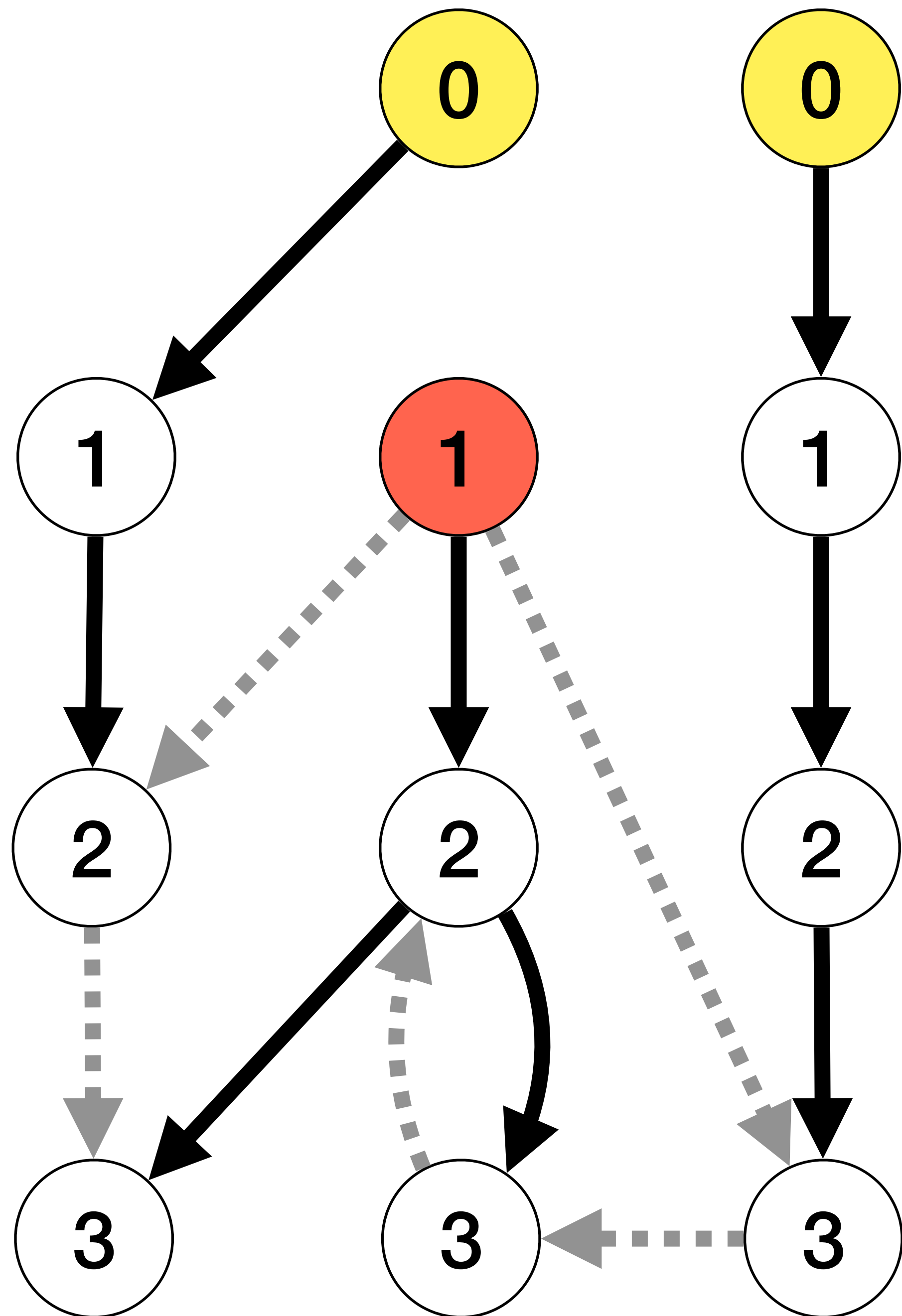
If we want to remove an edge that is part of the spanning tree, we use a 3-part algorithm



Removal of an edge

If we want to remove an edge that is part of the spanning tree, we use a **3-part algorithm**

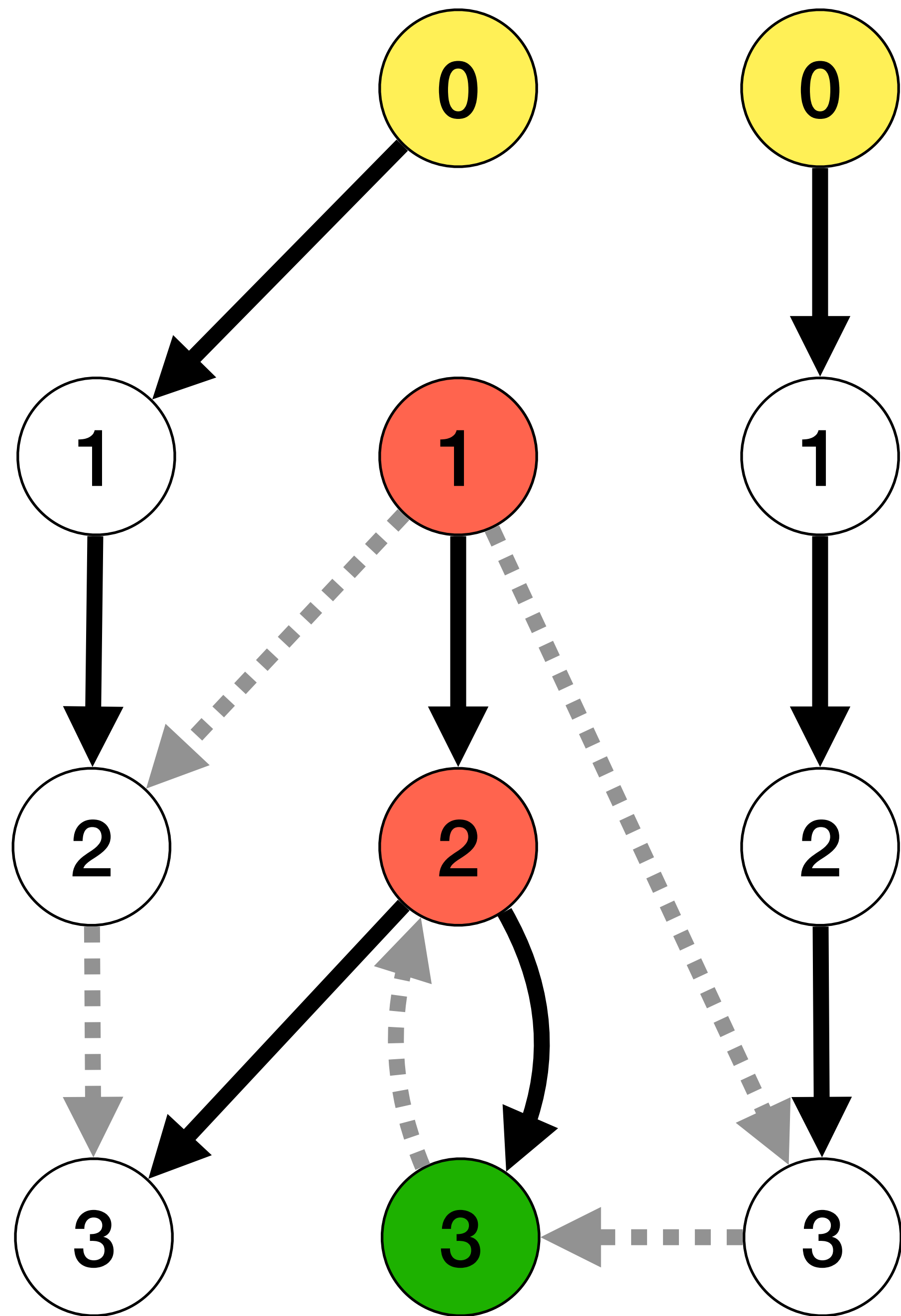
(1) **Drop:** Mark all **falling** and **anchor** nodes



Removal of an edge

If we want to remove an edge that is part of the spanning tree, we use a 3-part algorithm

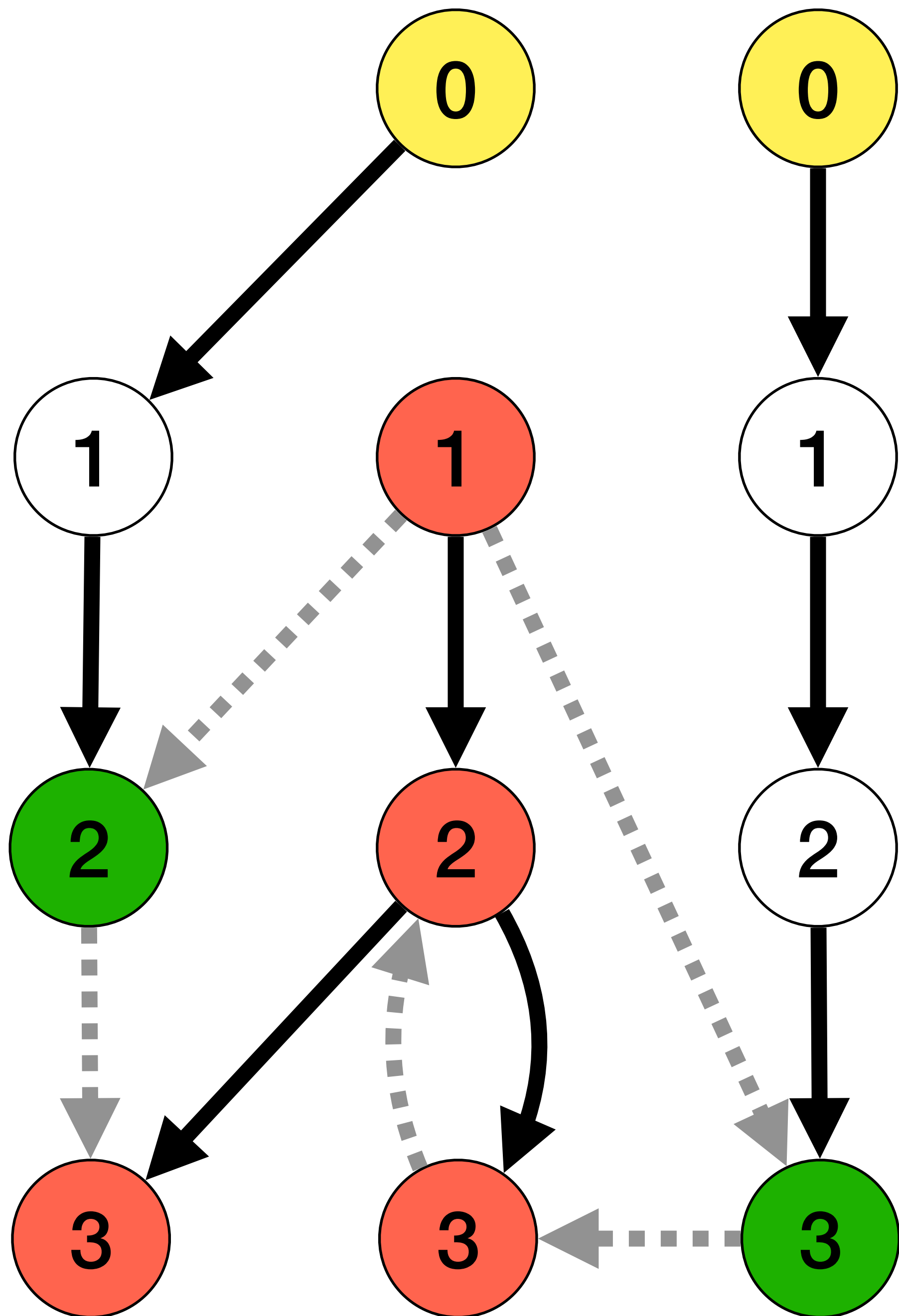
(1) **Drop:** Mark all **falling** and **anchor** nodes



Removal of an edge

If we want to remove an edge that is part of the spanning tree, we use a 3-part algorithm

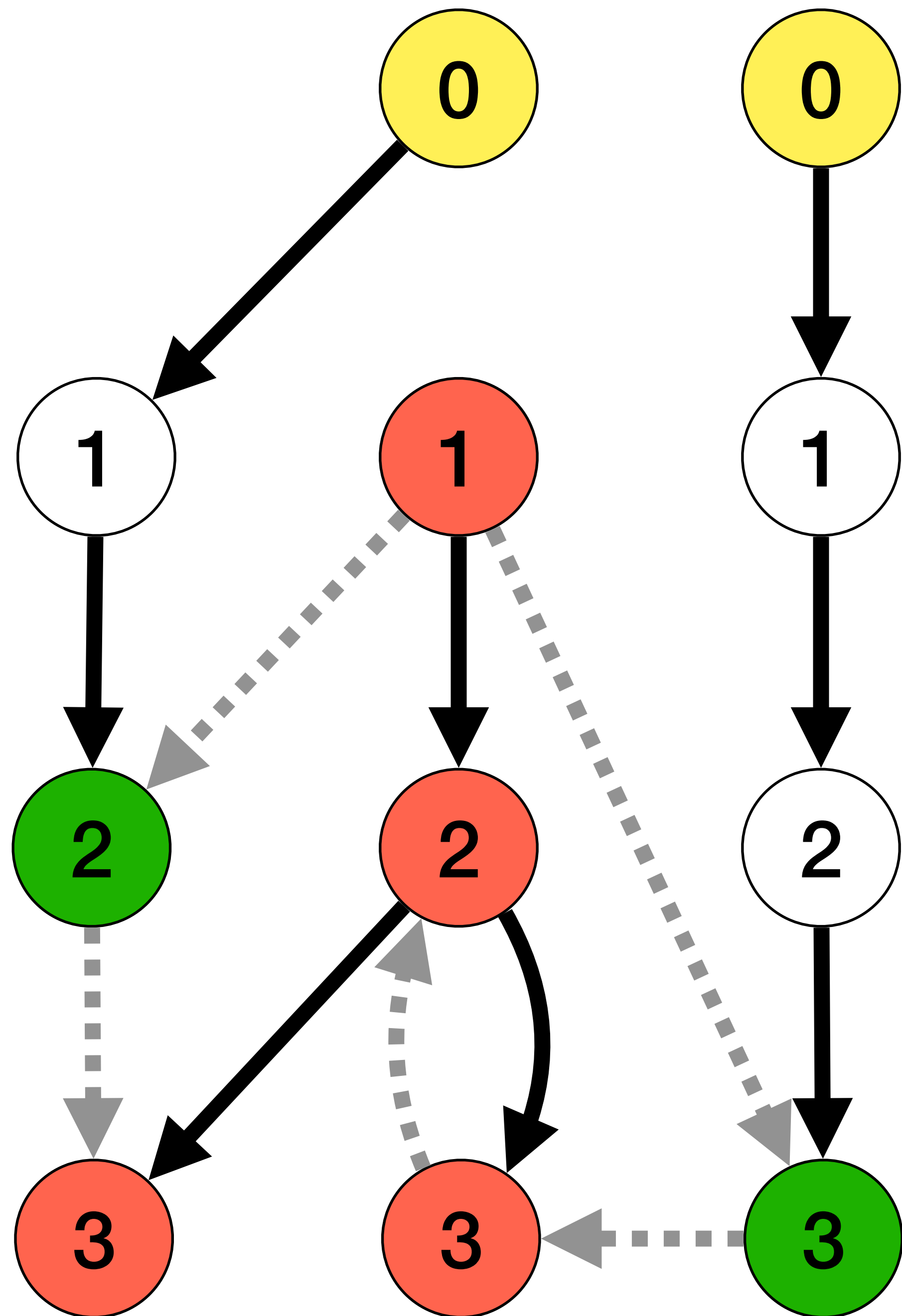
(1) **Drop:** Mark all **falling** and **anchor** nodes



Removal of an edge

If we want to remove an edge that is part of the spanning tree, we use a 3-part algorithm

(1) **Drop:** Mark all **falling** and **anchor** nodes

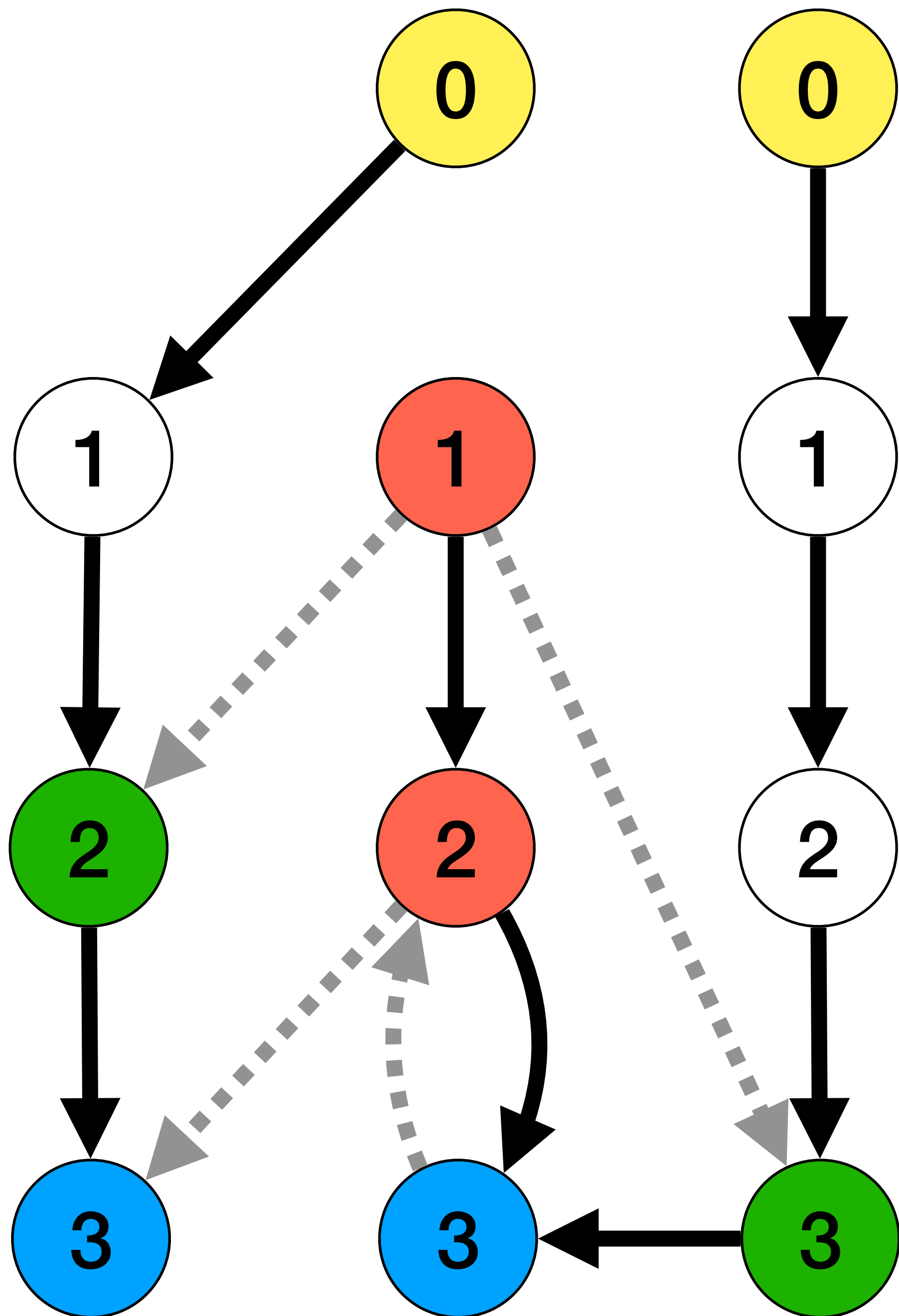


Removal of an edge

If we want to remove an edge that is part of the spanning tree, we use a 3-part algorithm

(1) **Drop:** Mark all **falling** and **anchor** nodes

(2) **Catch:** From **anchors** find all nodes that can be **caught** and **fix the spanning tree**

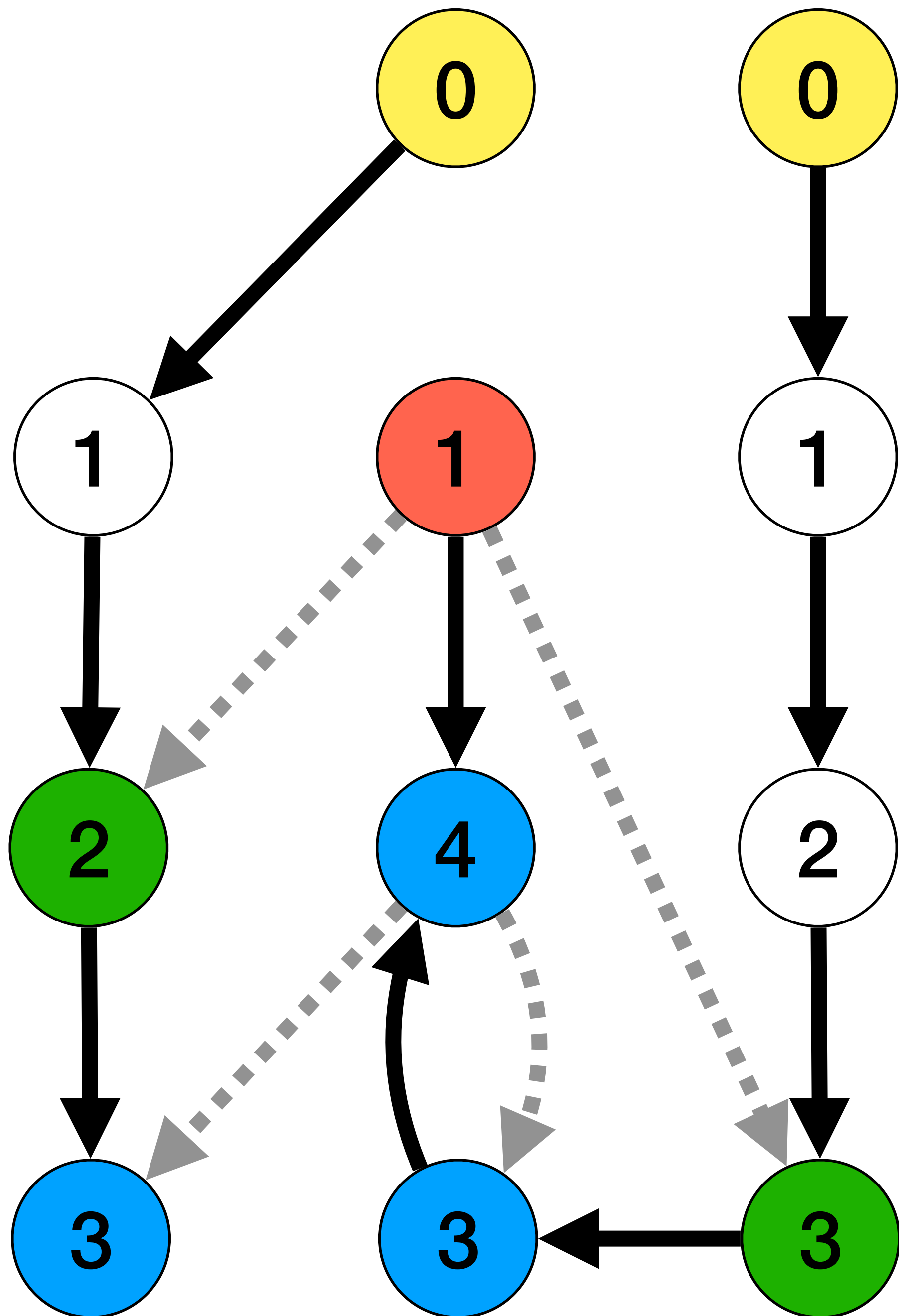


Removal of an edge

If we want to remove an edge that is part of the spanning tree, we use a 3-part algorithm

(1) **Drop:** Mark all **falling** and **anchor** nodes

(2) **Catch:** From **anchors** find all nodes that can be **caught** and **fix the spanning tree**

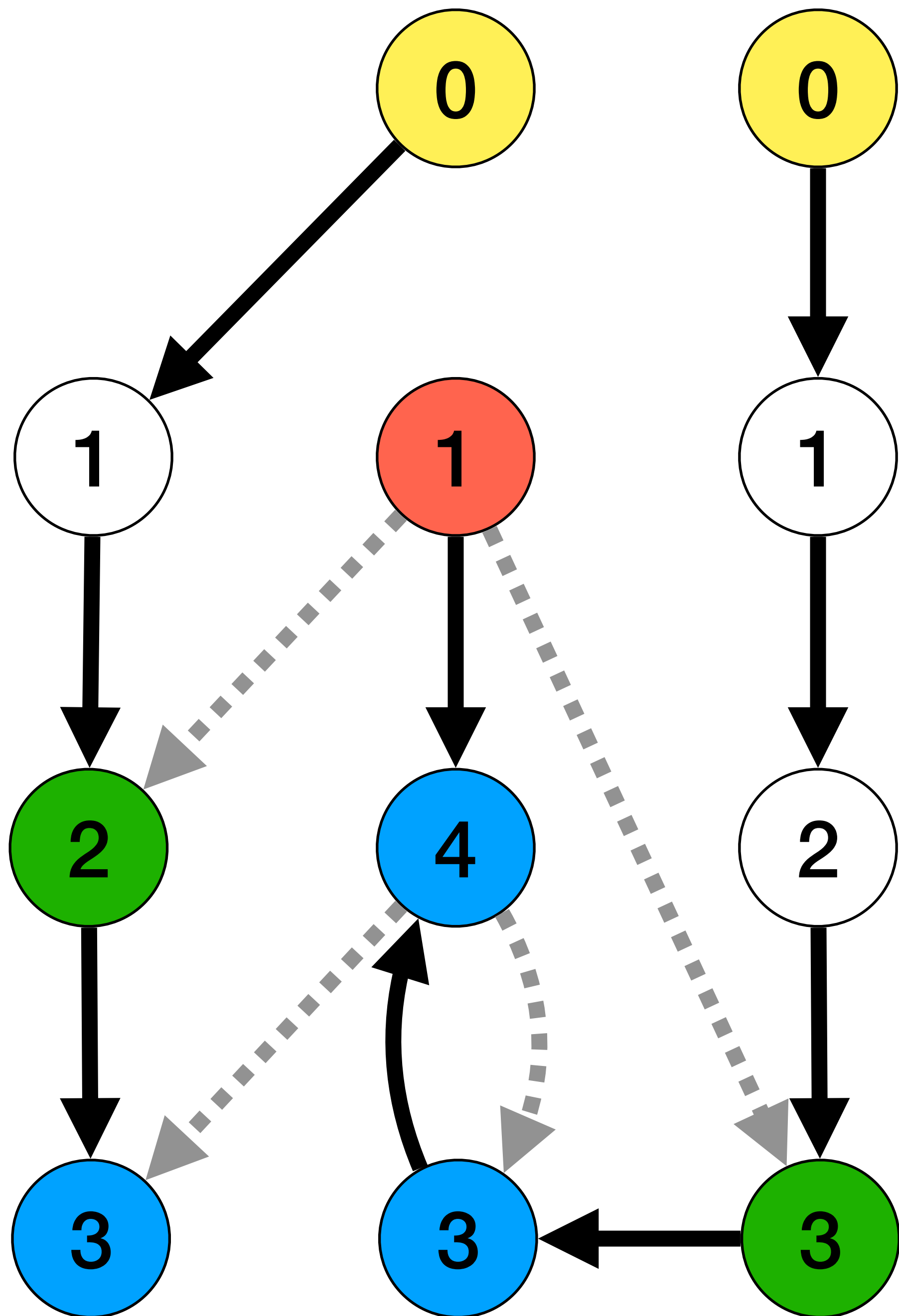


Removal of an edge

If we want to remove an edge that is part of the spanning tree, we use a 3-part algorithm

(1) **Drop:** Mark all **falling** and **anchors** nodes

(2) **Catch:** From **anchors** find all nodes that can be **caught** and **fix the spanning tree**



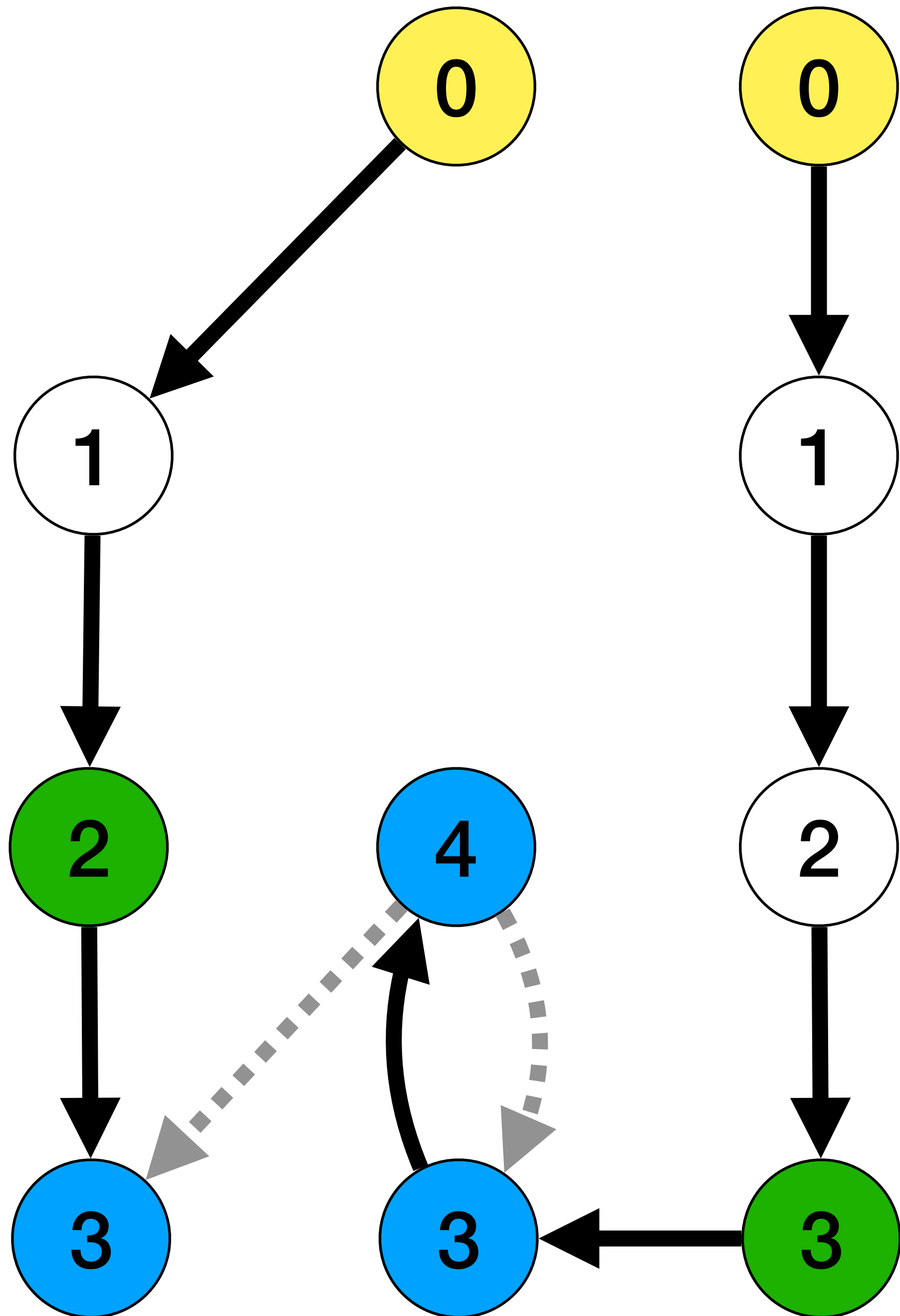
Removal of an edge

If we want to remove an edge that is part of the spanning tree, we use a 3-part algorithm

(1) **Drop:** Mark all **falling** and **anchor** nodes

(2) **Catch:** From **anchors** find all nodes that can be **caught** and **fix the spanning tree**

(3) **Collect:** Remove all **falling** nodes.



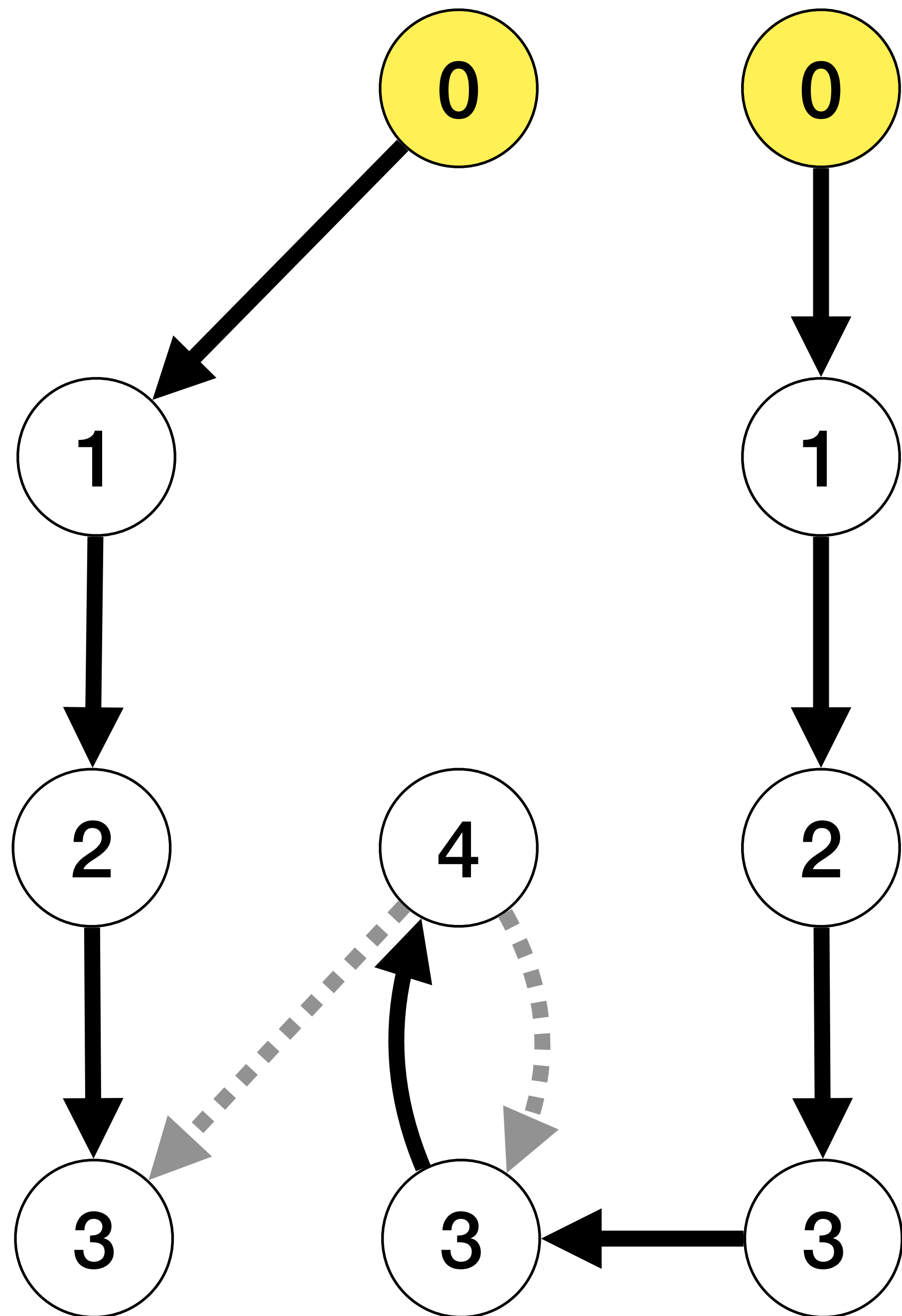
Removal of an edge

If we want to remove an edge that is part of the spanning tree, we use a 3-part algorithm

(1) **Drop:** Mark all **falling** and **anchor** nodes

(2) **Catch:** From **anchors** find all nodes that can be **caught** and **fix the spanning tree**

(3) **Collect:** Remove all **falling** nodes.



Removal of an edge

If we want to remove an edge that is part of the spanning tree, we use a 3-part algorithm

(1) **Drop:** Mark all **falling** and **anchor** nodes

(2) **Catch:** From **anchors** find all nodes that can be **caught** and **fix the spanning tree**

(3) **Collect:** Remove all **falling** nodes.

... still a spanning forest

Can we use this in practice?

Its about 3 orders of magnitudes
slower than our baseline...

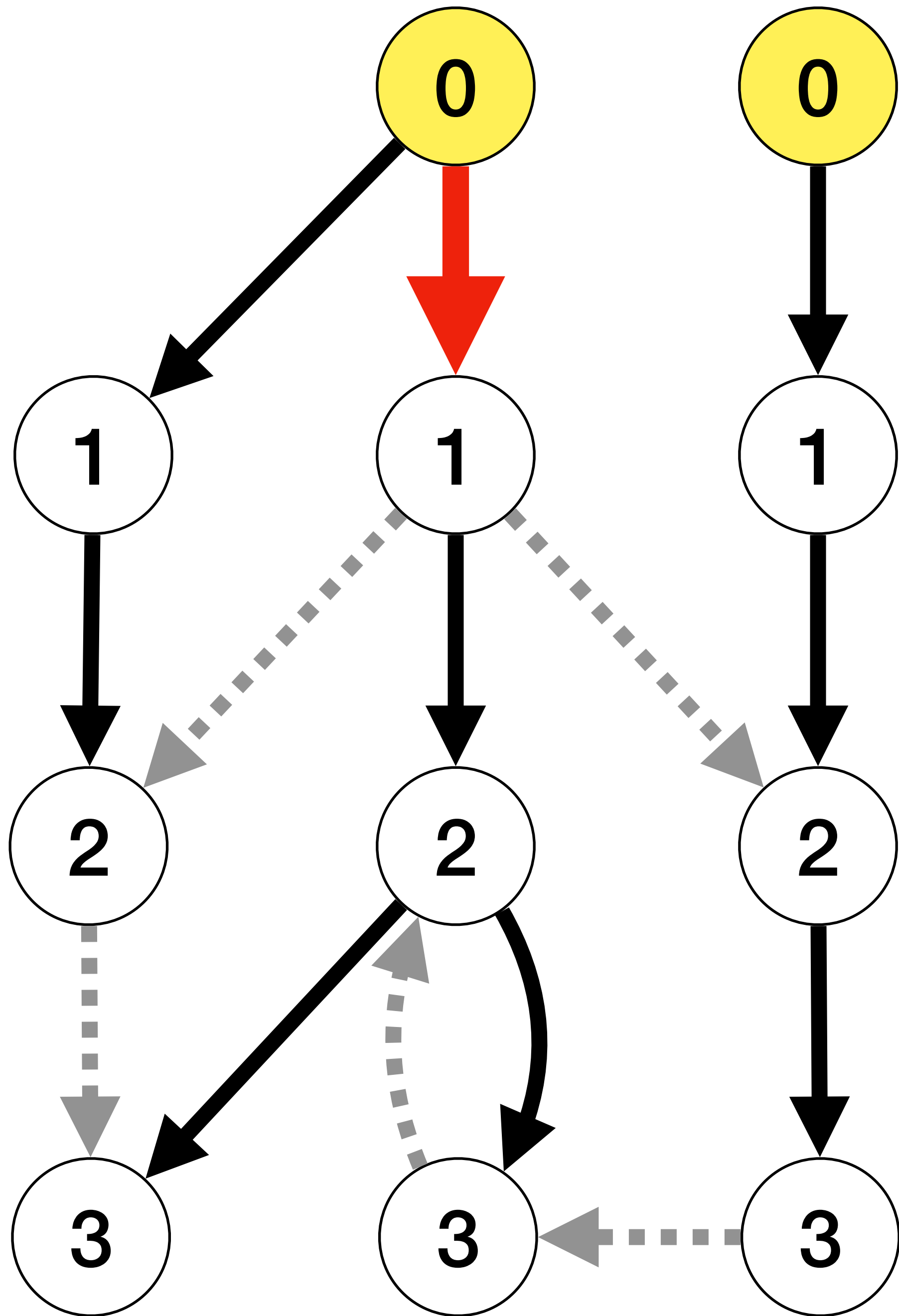
Now the question is, how to make it
faster?

First optimisation

Adoption

An adoption prevents a node from falling in the drop phase.

It occurs when the falling node has a coparent that has a smaller rank than it

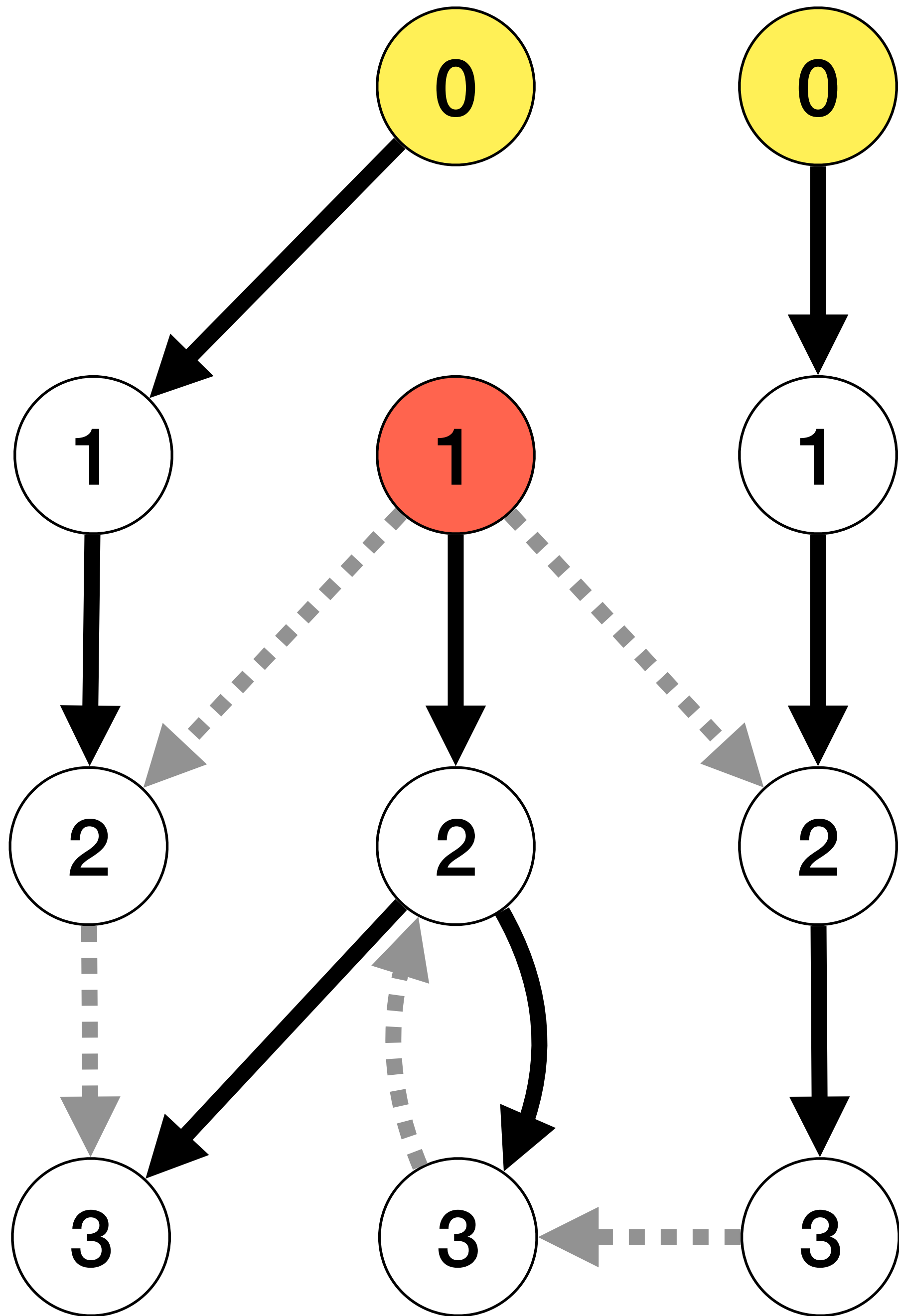


First optimisation

Adoption

An adoption prevents a node from falling in the drop phase.

It occurs when the falling node has a coparent that has a smaller rank than it

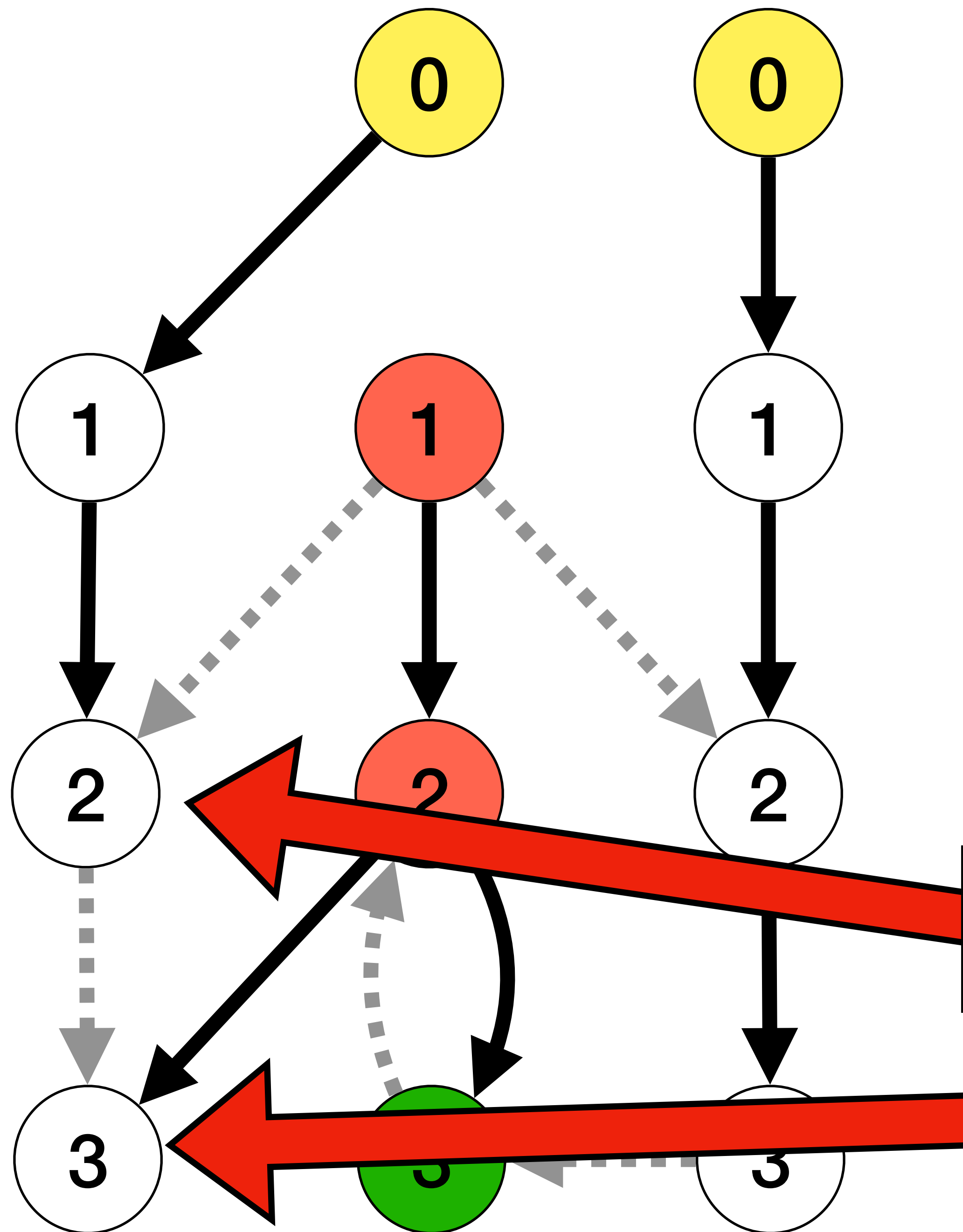


First optimisation

Adoption

An adoption prevents a node from falling in the drop phase.

It occurs when the falling node has a coparent that has a smaller rank than it



This is a coparent with a smaller rank

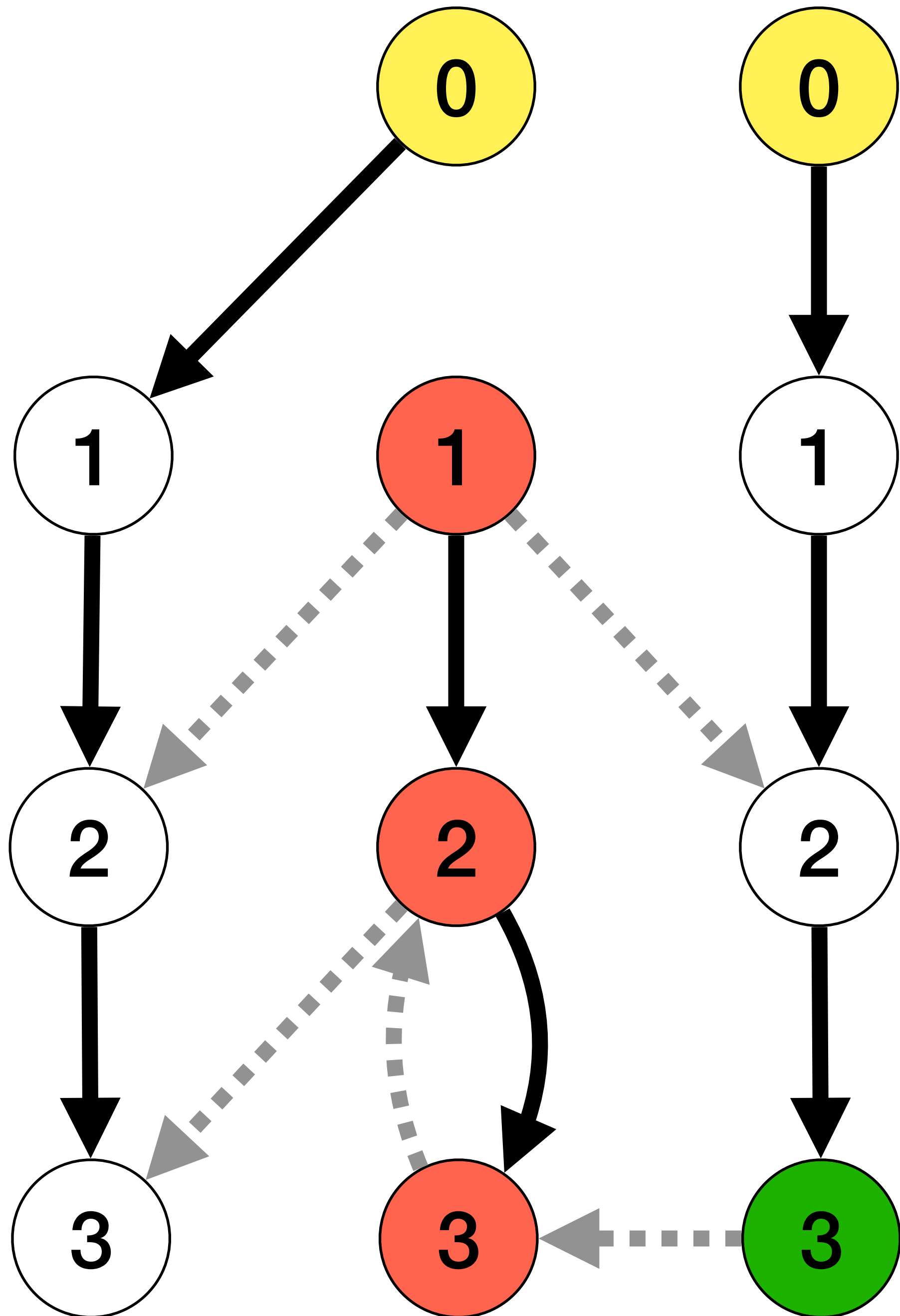
This node will fall

First optimisation

Adoption

An adoption prevents a node from falling in the drop phase.

It occurs when the falling node has a coparent that has a smaller rank than it



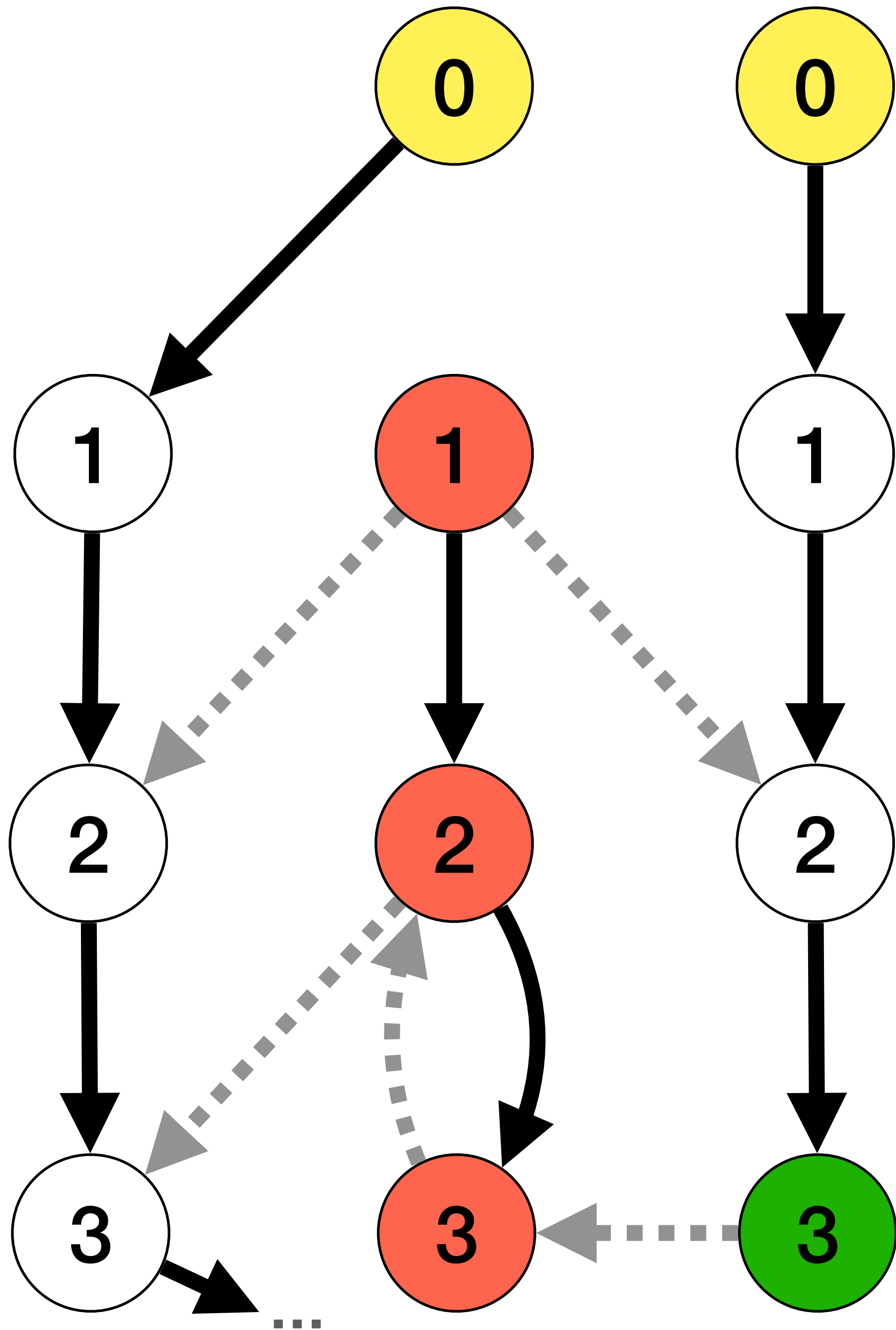
First optimisation

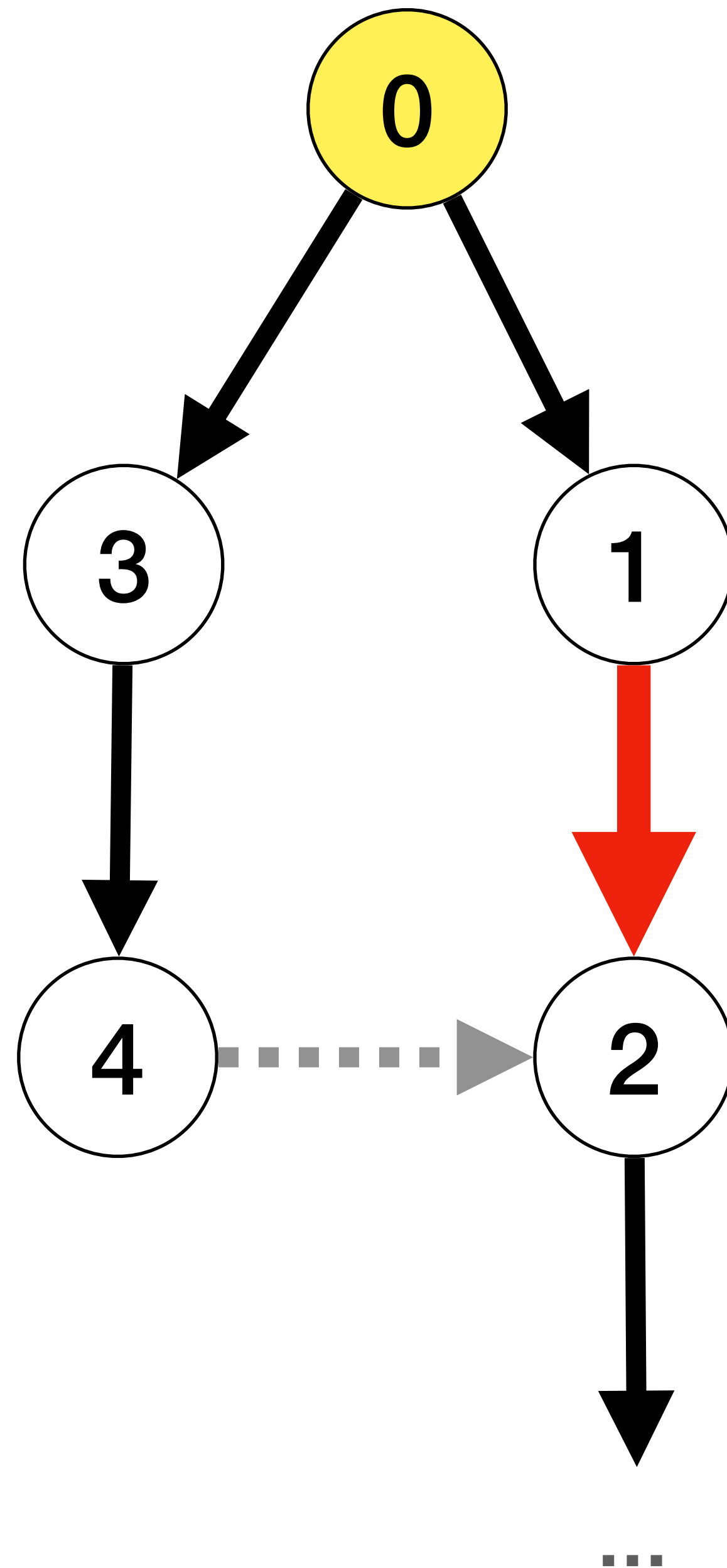
Adoption

An adoption prevents a node from falling in the drop phase.

It occurs when the falling node has a coparent that has a smaller rank than it

It can prevent a lot of nodes from falling



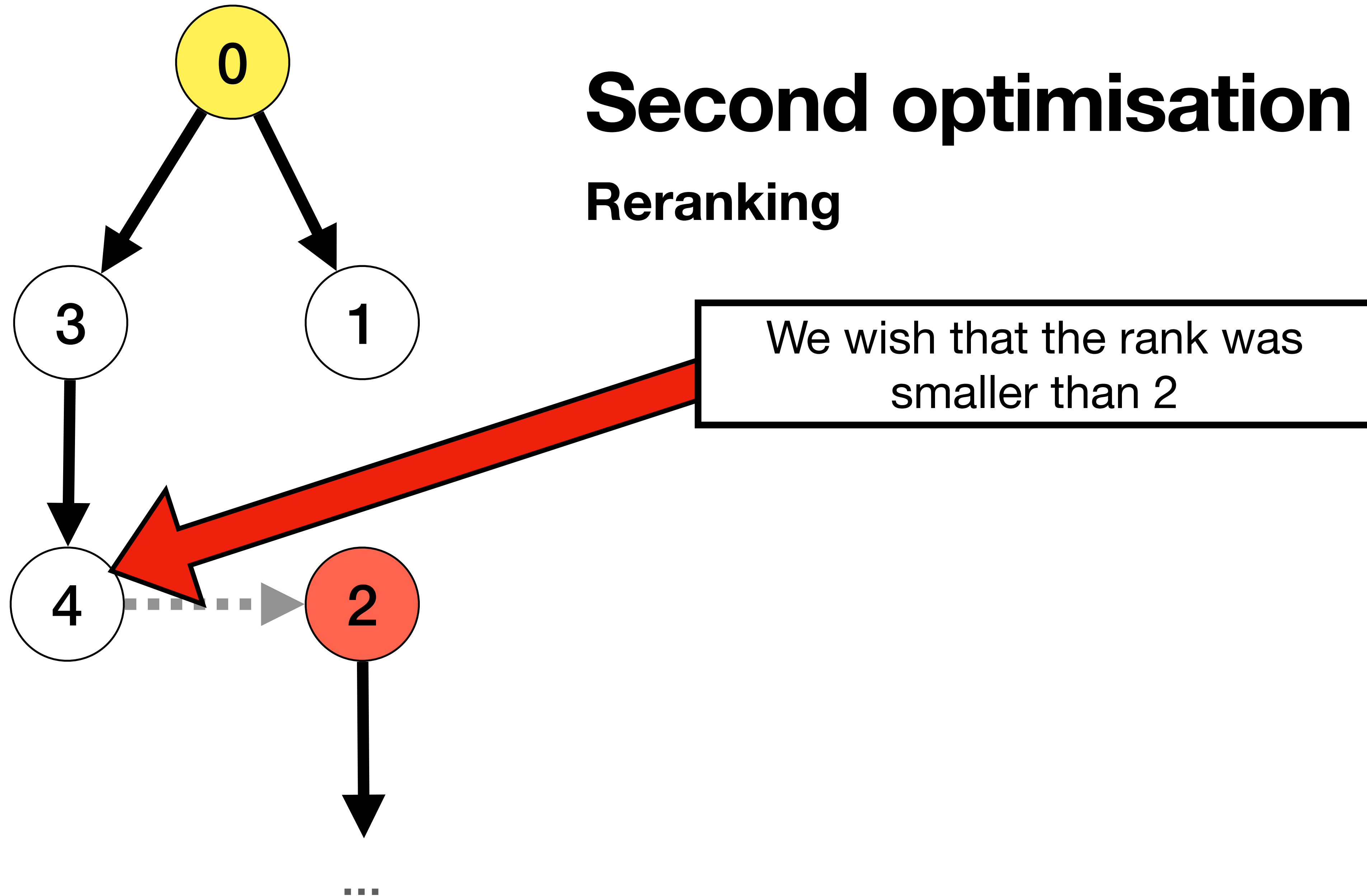


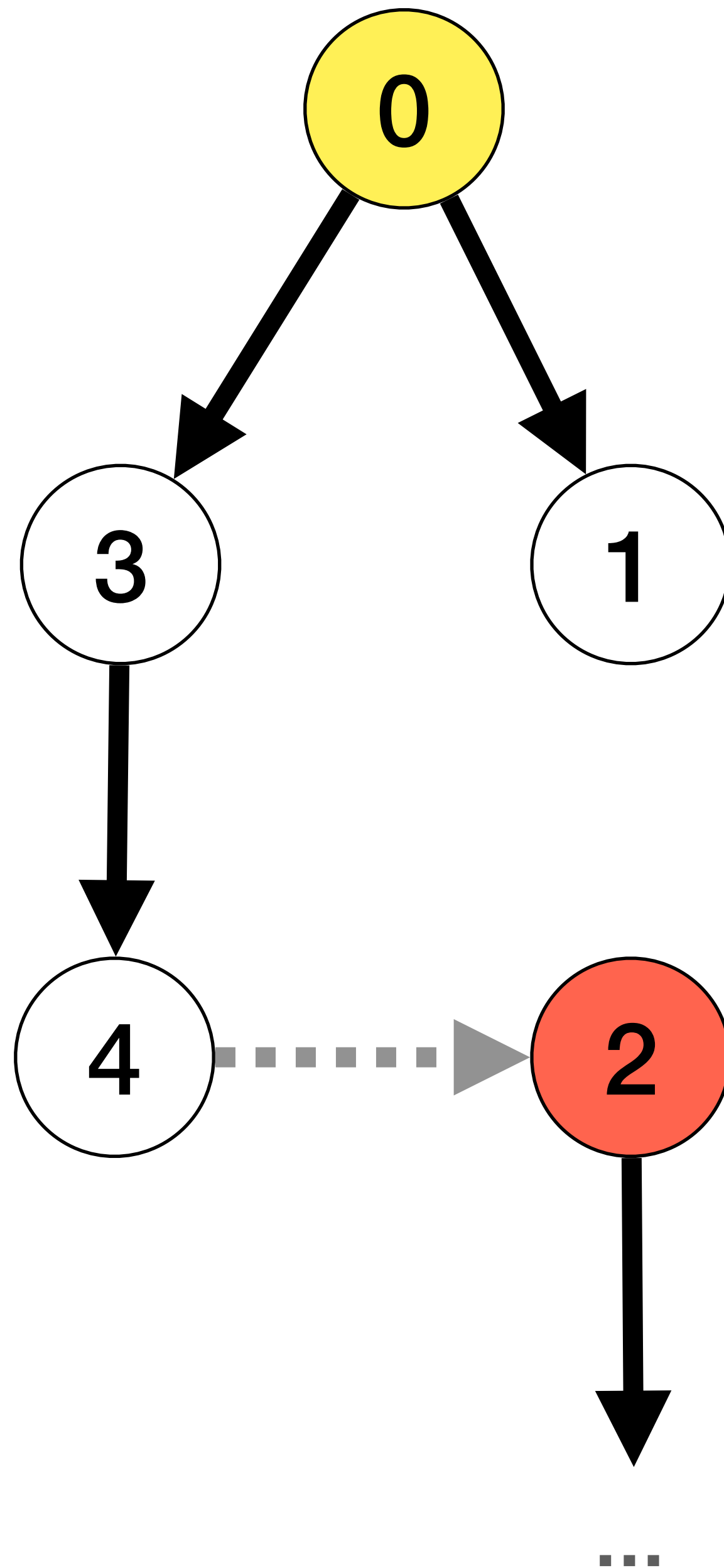
Second optimisation

Reranking

Second optimisation

Reranking

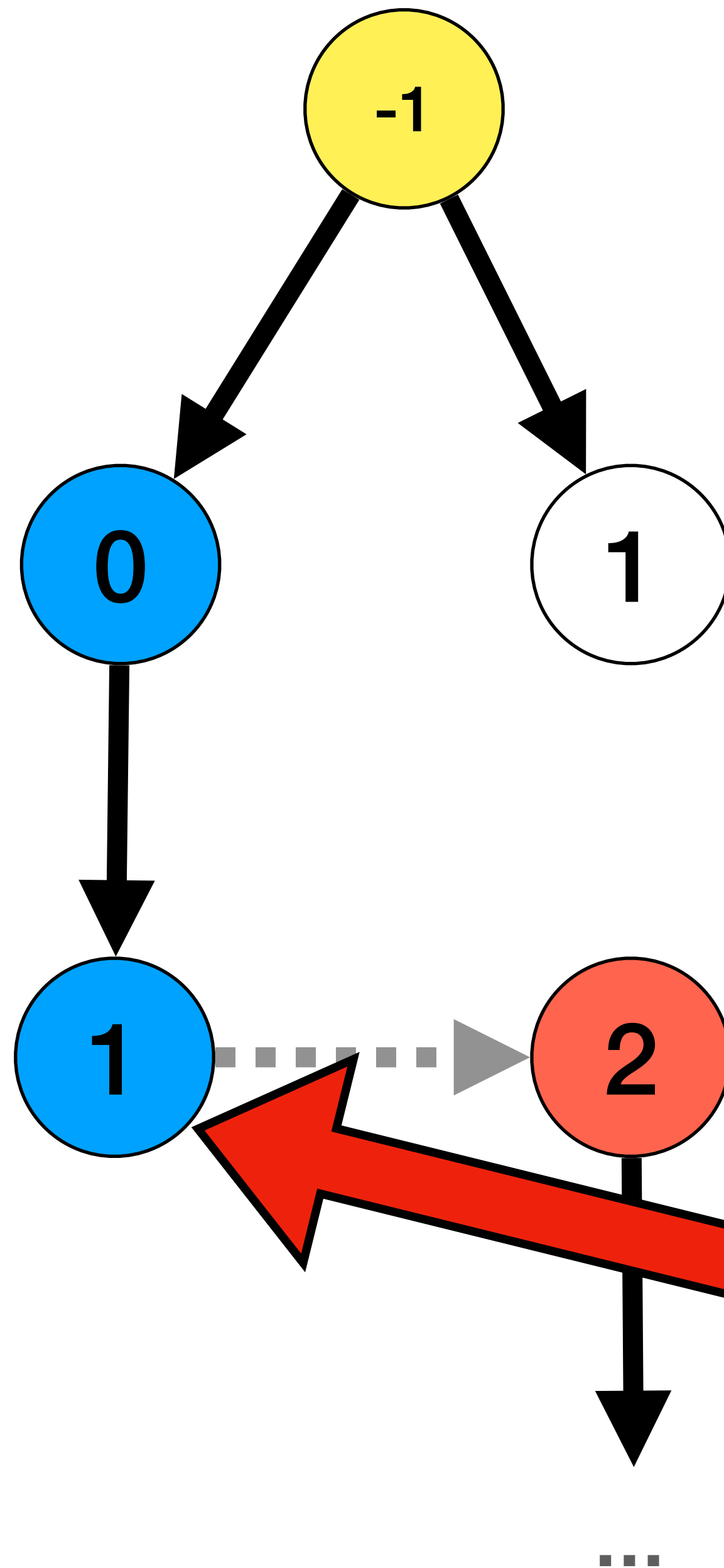




Second optimisation

Reranking

In some cases, it is more efficient to **reduce** a coparent's rank than to traverse the subgraph.

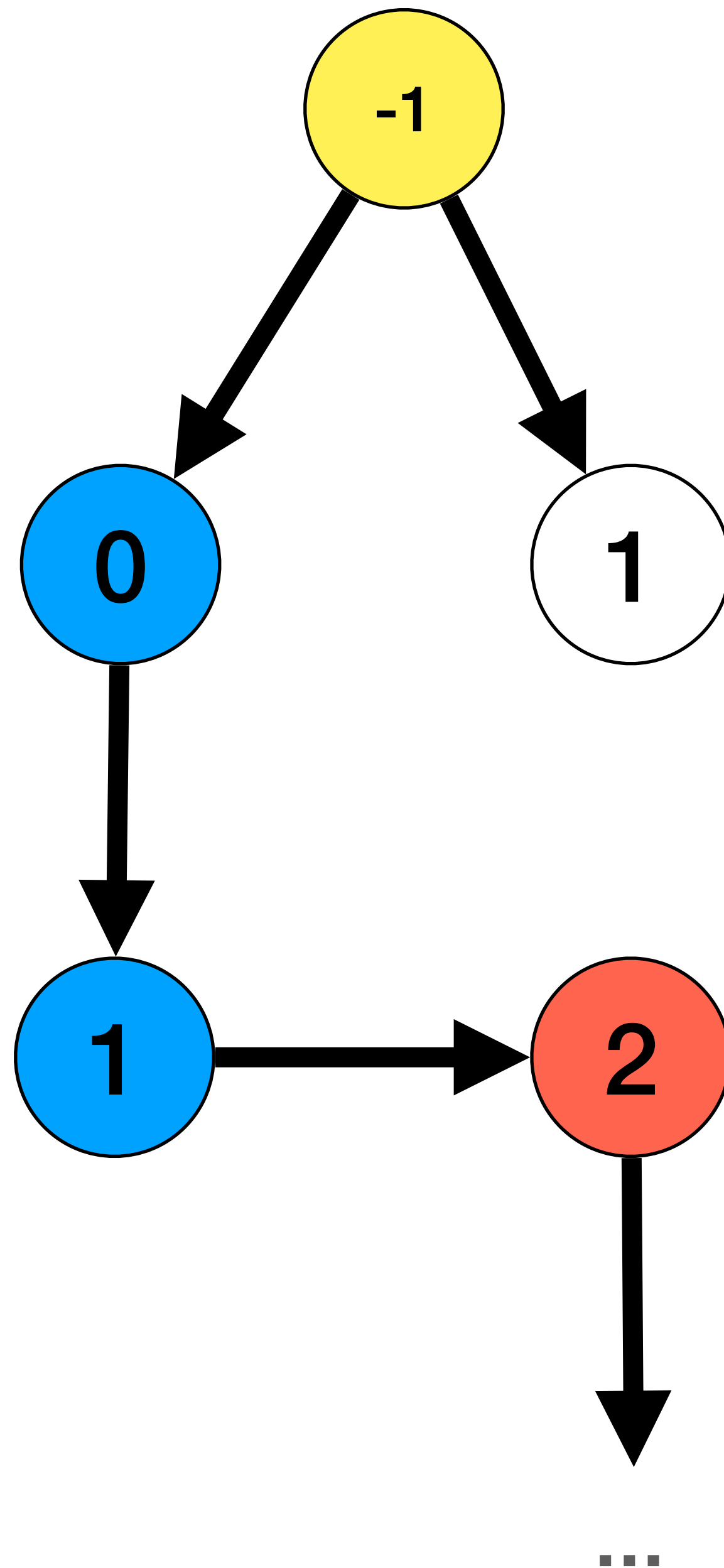


Second optimisation

Reranking

In some cases, it is more efficient to **reduce** a coparent's rank than to traverse the subgraph.

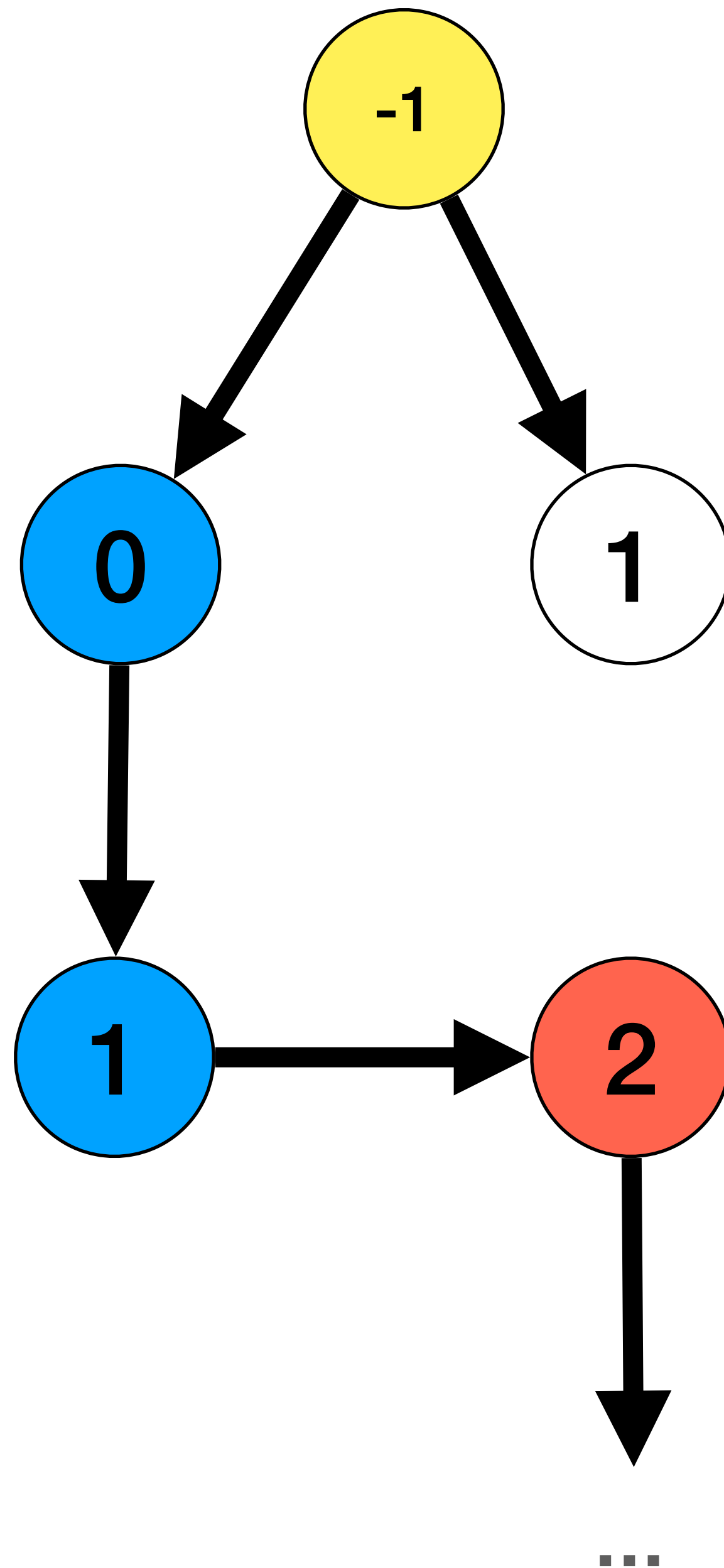
This coparent can now adopt the falling node



Second optimisation

Reranking

In some cases, it is more efficient to **reduce** a coparent's rank than to traverse the subgraph.



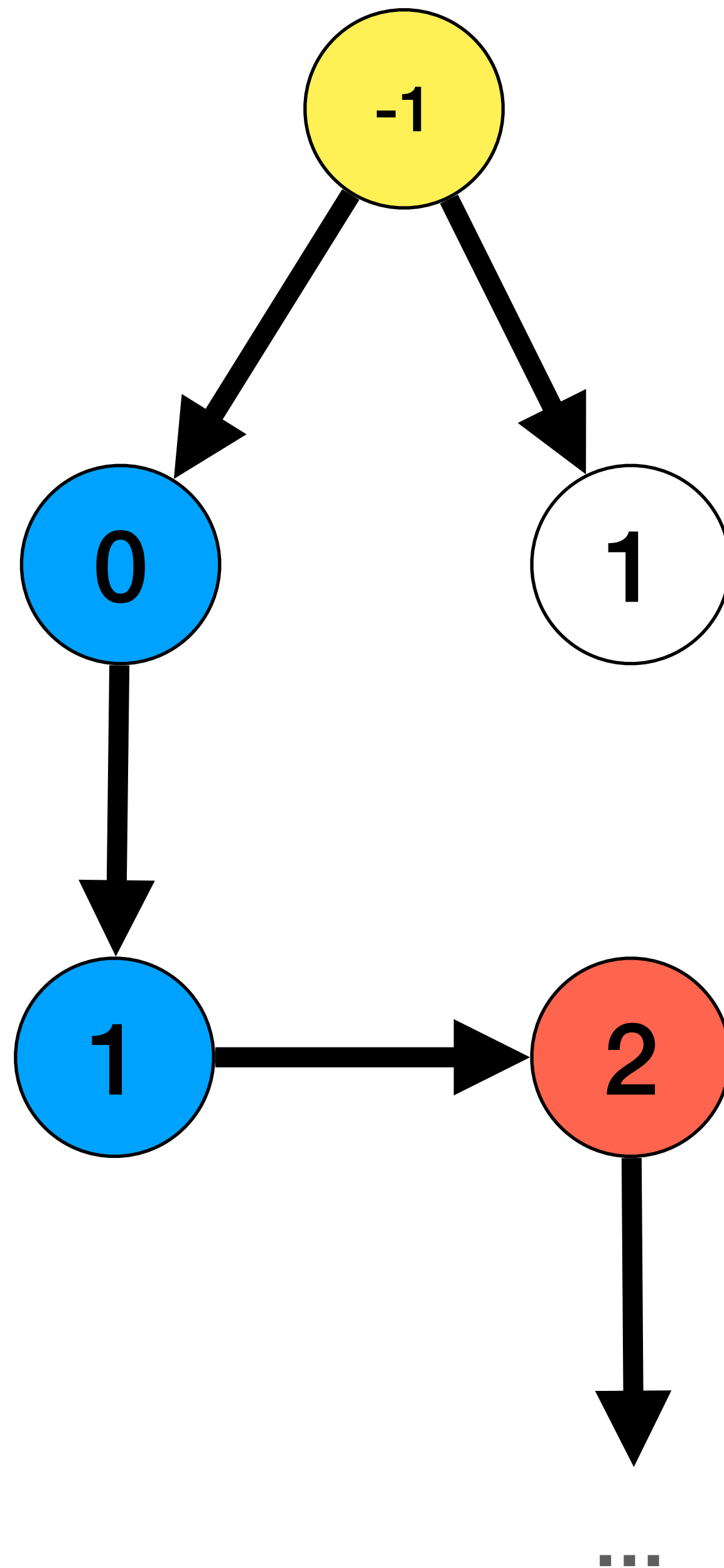
Second optimisation

Reranking

In some cases, it is more efficient to **reduce** a coparent's rank than to traverse the subgraph.

However, this can be **risky** as we need to traverse parents until:

- We find a root
- We remove enough gaps
- We find the original falling node



Second optimisation

Reranking

In some cases, it is more efficient to **reduce** a coparent's rank than to traverse the subgraph.

However, this can be **risky** as we need to traverse parents until:

- We find a root
- We remove enough gaps
- We find the original falling node

We perform the reranking on the first few nodes only.

Third optimisation

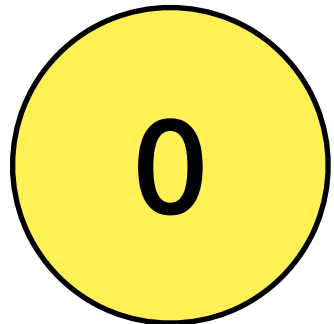
Choice of the initial rank

When creating a new object, we can choose it's rank.

Third optimisation

Choice of the initial rank

When creating a new object, we can choose it's rank.



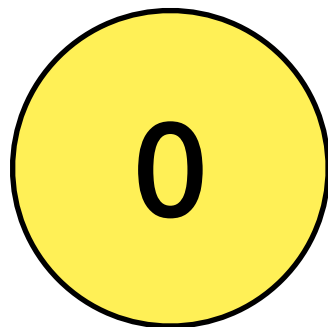
Third optimisation

Choice of the initial rank

When creating a new object, we can choose it's rank.

We choose to initialize the rank with a **decreasing global counter**.

It ensures that the newly created node can adopt any nodes.



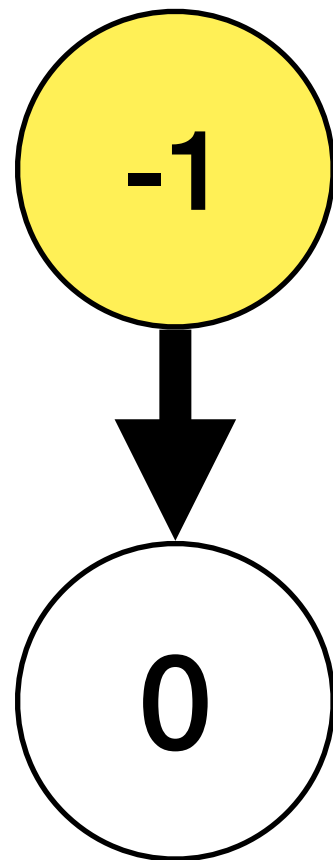
Third optimisation

Choice of the initial rank

When creating a new object, we can choose it's rank.

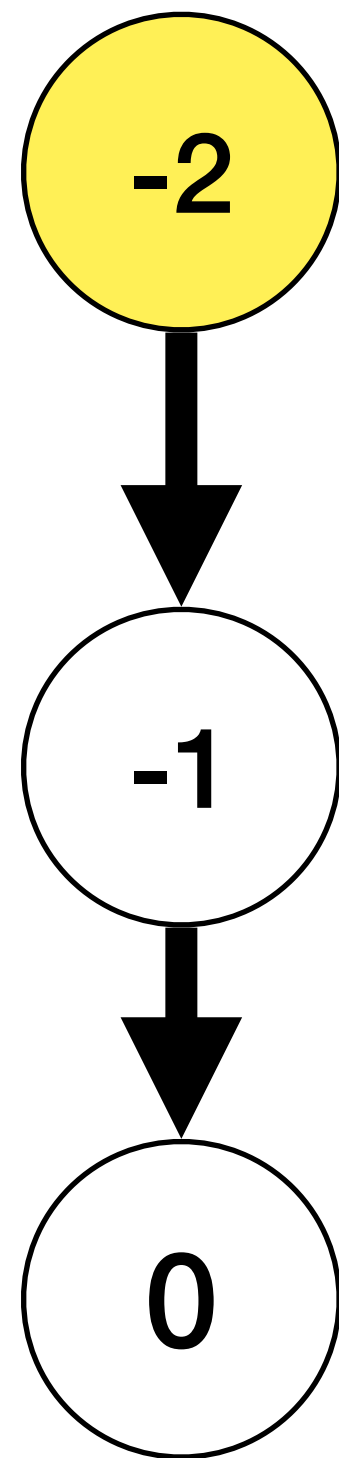
We choose to initialize the rank with a **decreasing global counter**.

It ensures that the newly created node can adopt any nodes.



Third optimisation

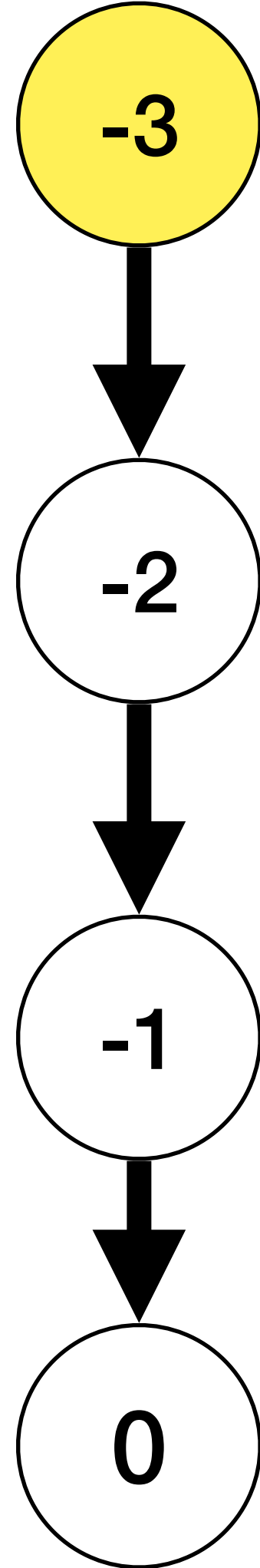
Choice of the initial rank



When creating a new object, we can choose it's rank.

We choose to initialize the rank with a **decreasing global counter**.

It ensures that the newly created node can adopt any nodes.



Third optimisation

Choice of the initial rank

When creating a new object, we can choose it's rank.

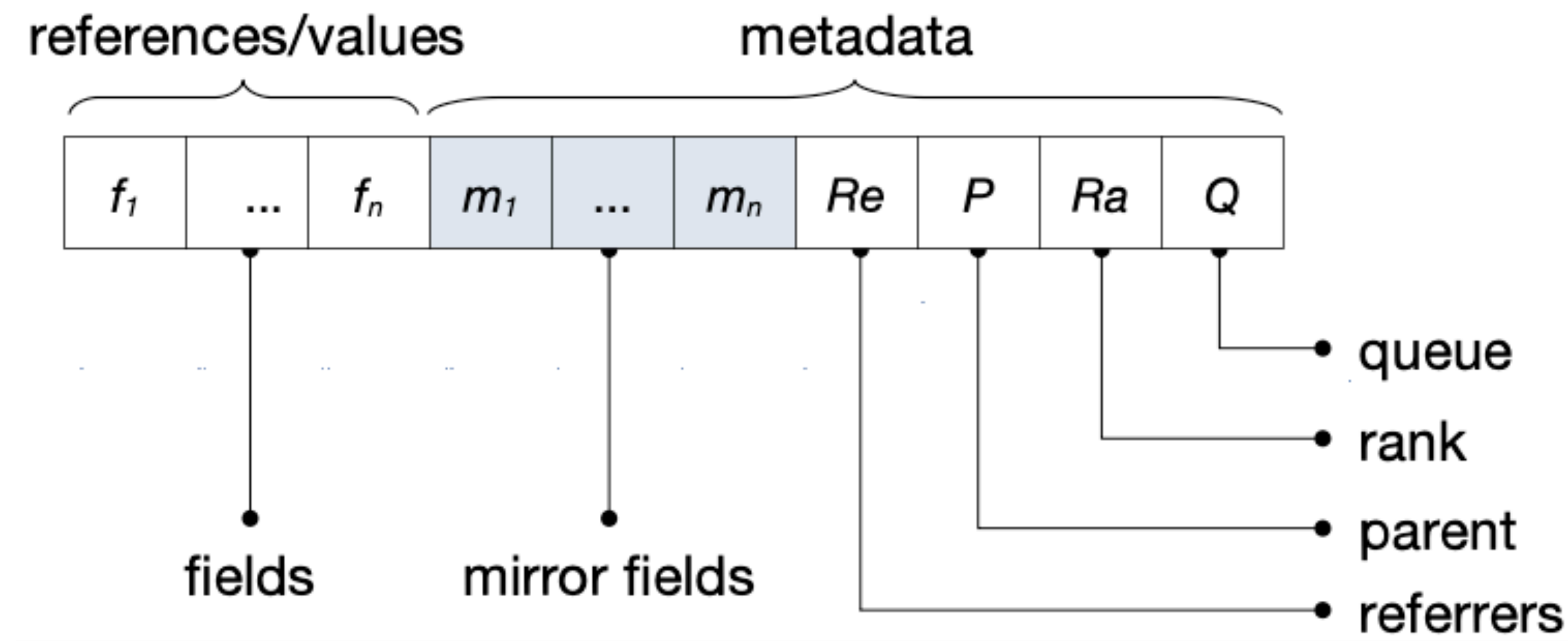
We choose to initialize the rank with a **decreasing global counter**.

It ensures that the newly created node can adopt any nodes.

The devil lies in the details

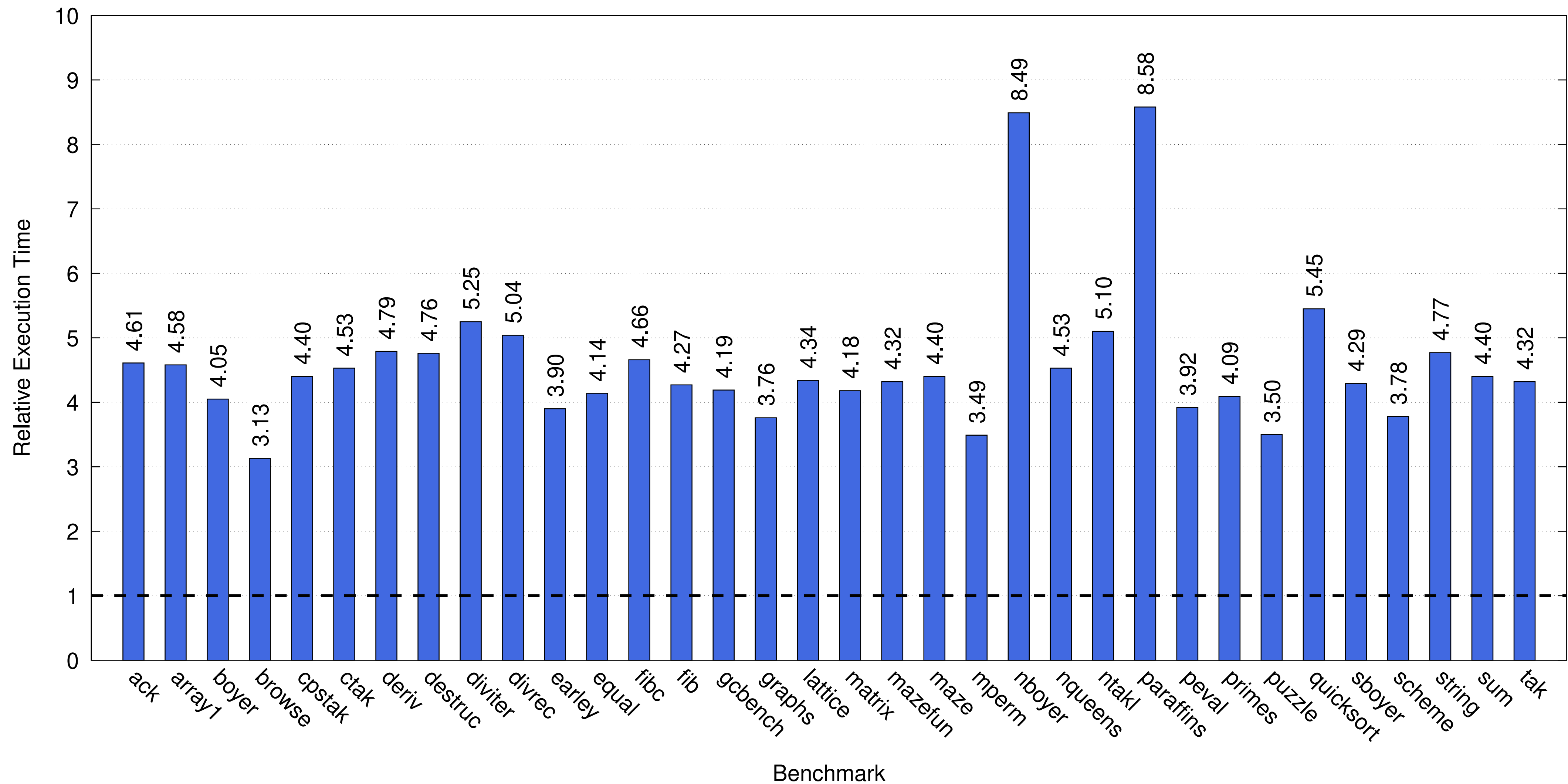
Some implementations details were omitted by lack of time.

How do we keep the coparent-cofriend relationship in memory?



There is a whole section about implementation in the paper :)

Benchmarks



Future work

More optimizations:

- better heuristics
- take advantage of specific common cases

Real world scenarios:

- interoperability with C++/Rust destructors
- object stores

More usage of other dynamic tree structures?

Conclusion

The Arborescent Garbage Collector collects **cyclic** structures **immediately** by embedding a **spanning tree** into the **reference graph**.

We add a weak notion of ranks on **objects**, allowing the **adoption** and **rerank** optimisation.

Our technique is ~4.5x slower compared to mark-and-sweep. Previous work was **prohibitive** and **unusable synchronously**.

We believe immediate collection of all garbage can enable applications requiring timely reclamation of resources.

Backup slides

Collecting cycles immediately

- **CPython:** Reference counting with occasional tracing (**not immediate**)
- **Bacon and Raejan:** Reference counting with deferred cycle collection to an asynchronous background task (**not immediate**)
- **Brownbridge and Picher:** introduced mechanisms for maintaining a spanning trees in a reference graph. (**prohibitive**)
- **Brandt et al.:** described an algorithm for repairing the spanning tree when a reference is deleted, reclaiming unreachable objects in the process. (**prohibitive**)

Contributions

- We propose a new algorithm that can collect cycles immediately without the prohibitive slowdown of previous techniques.

Benchmarks

