Graph Theory

Graph Traversal Algorithms

Sergio Peignier sergio.peignier@insa-lyon.fr

Associate Professor
INSA Lyon
Biosciences department

Table of contents

Breadth-First Search

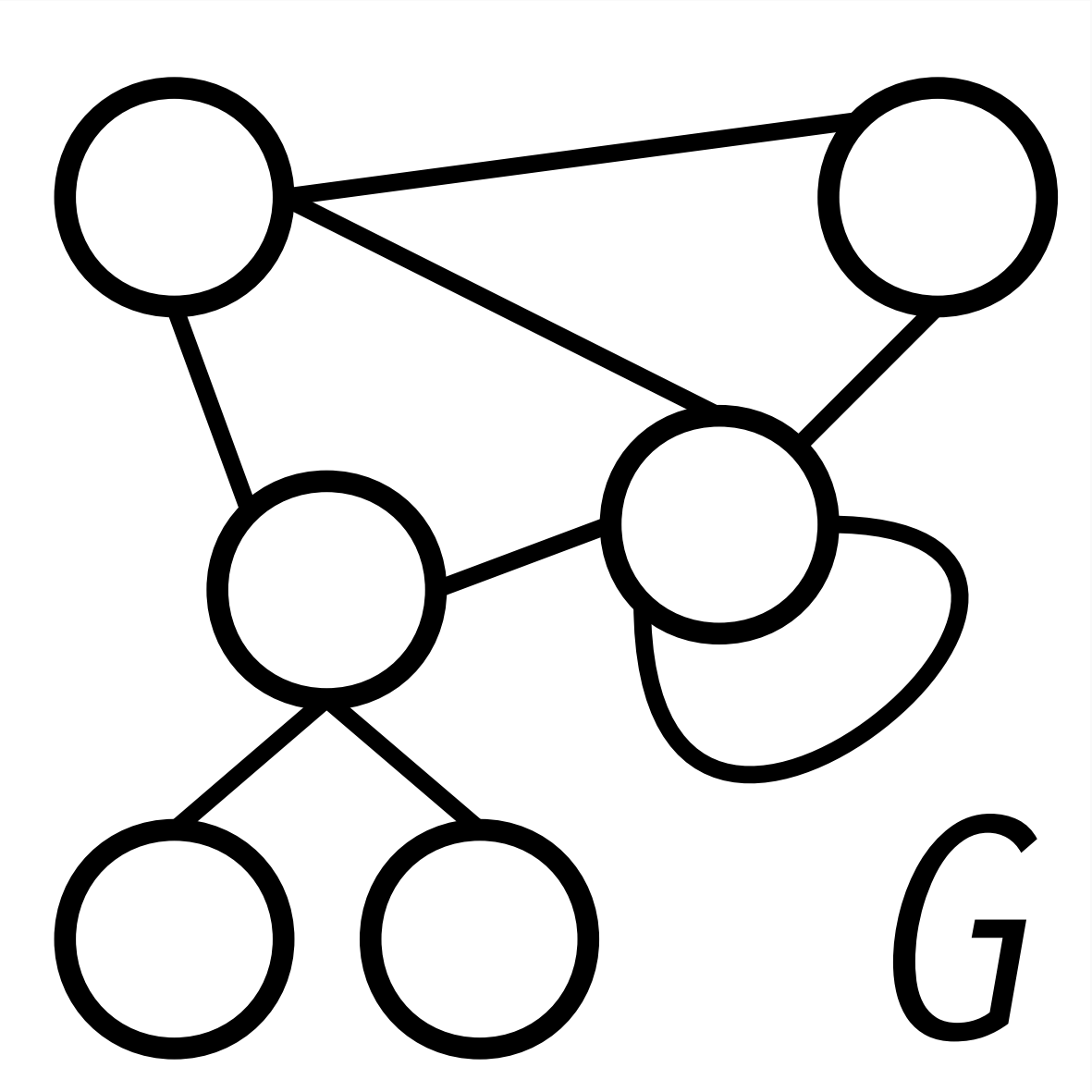
Breadth-First Search

- Algorithm to **traverse/search** in **graphs**.
- **Starts** at some **arbitrary node** *s*.
- **Visit all neighboring** nodes at the **present depth**.
- **Move** to nodes from the **next depth level**.

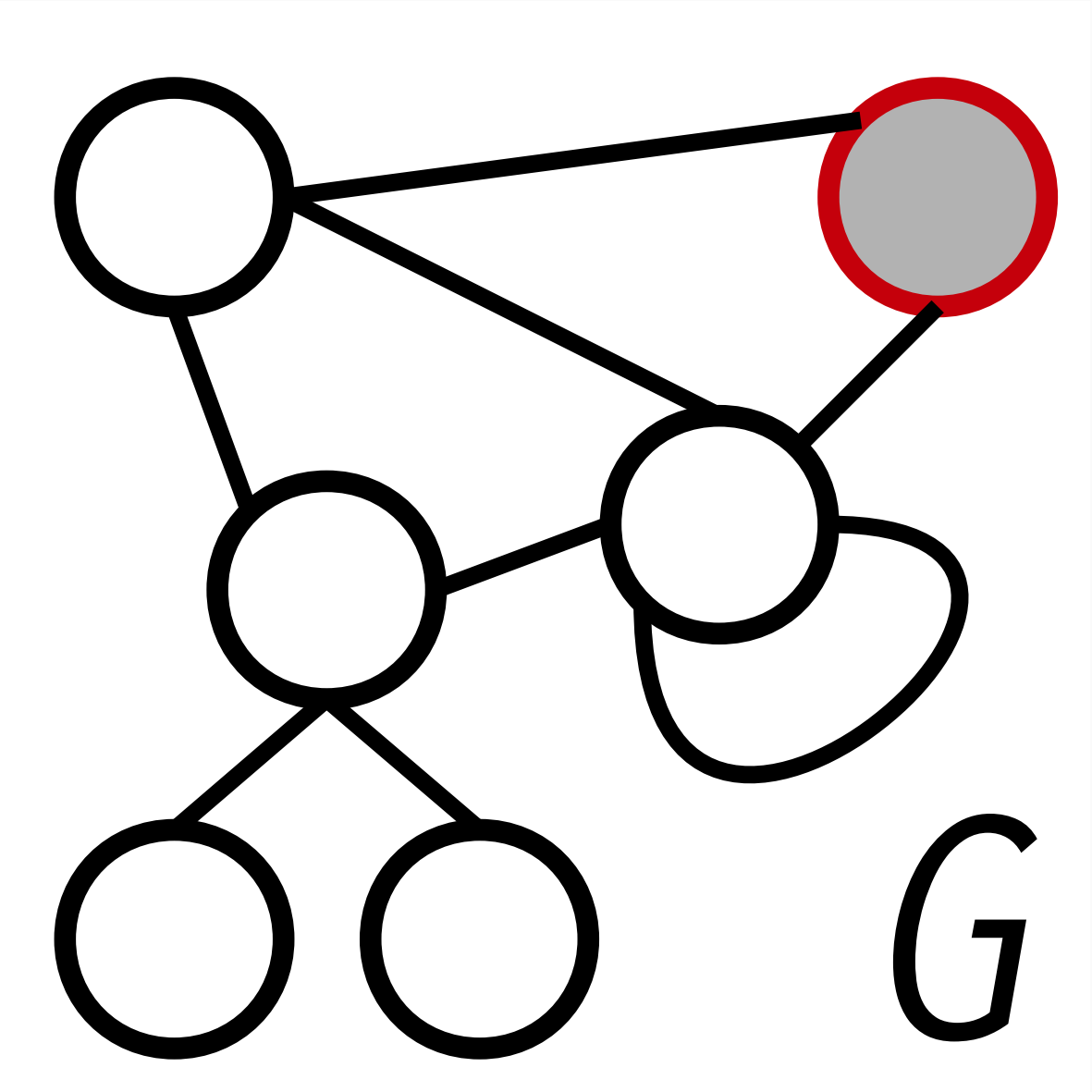
Pseudocode

```
function BFS(G = ⟨V, E⟩, s ∈ V)
    for all v ∈ V do
        state[v] ← undiscovered,    parent[v] ← ∅
    end for
    state[s] ← discovered,    parent[s] ← root
    Q ← [s] // Queue containing s
    while Size(Q) > 0 do
        u ← Q[0]
        for all v ∈ Neighbors(u) do
            if state[v] = undiscovered then
                state[v] ← discovered,    parent[v] ← u
                Q ← Q + [v] // Append new node
            end if
        end for
        Q ← Q[1 : Size(Q)] //Remove 1st node (FIFO)
    end while
    Return parent
end function
```

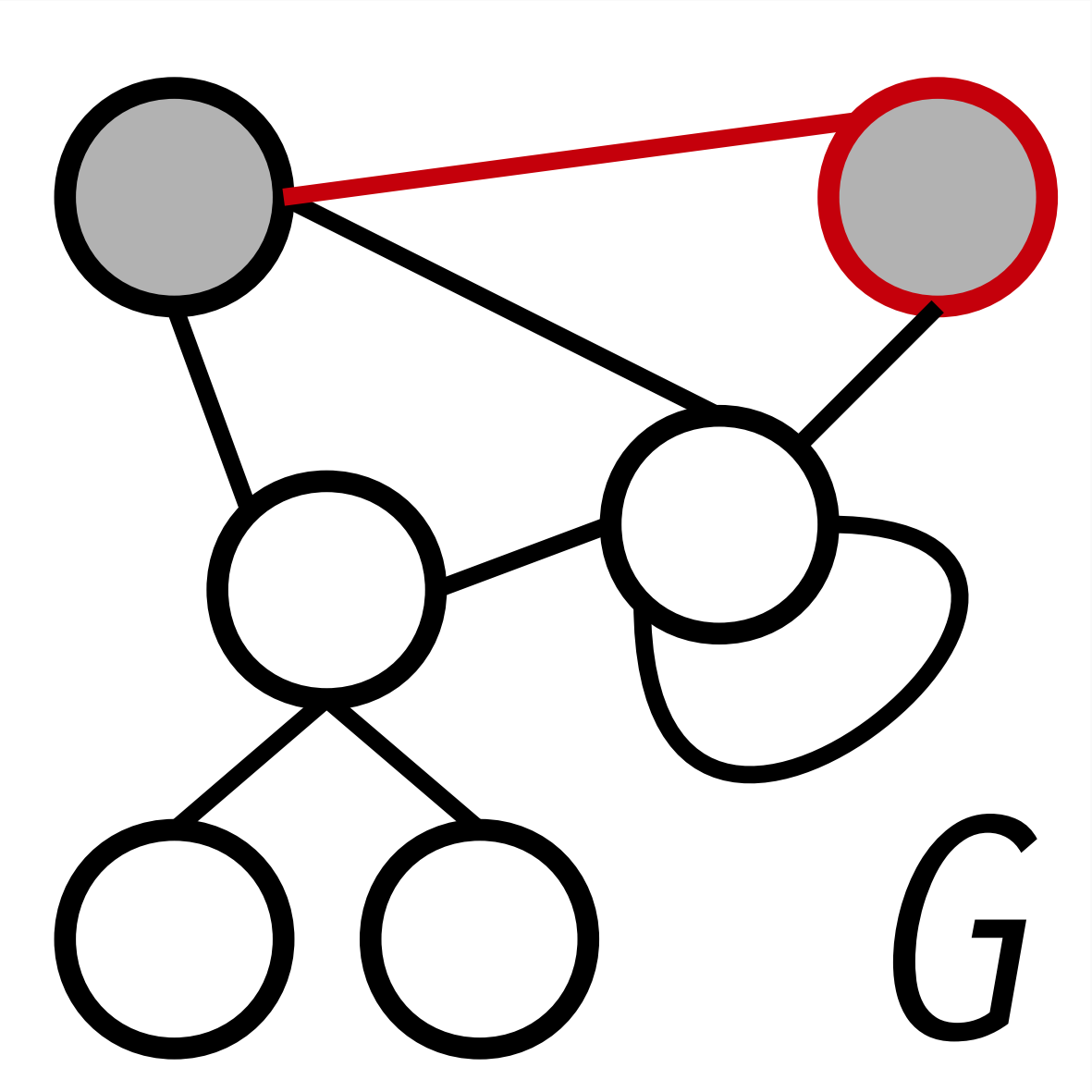
Example



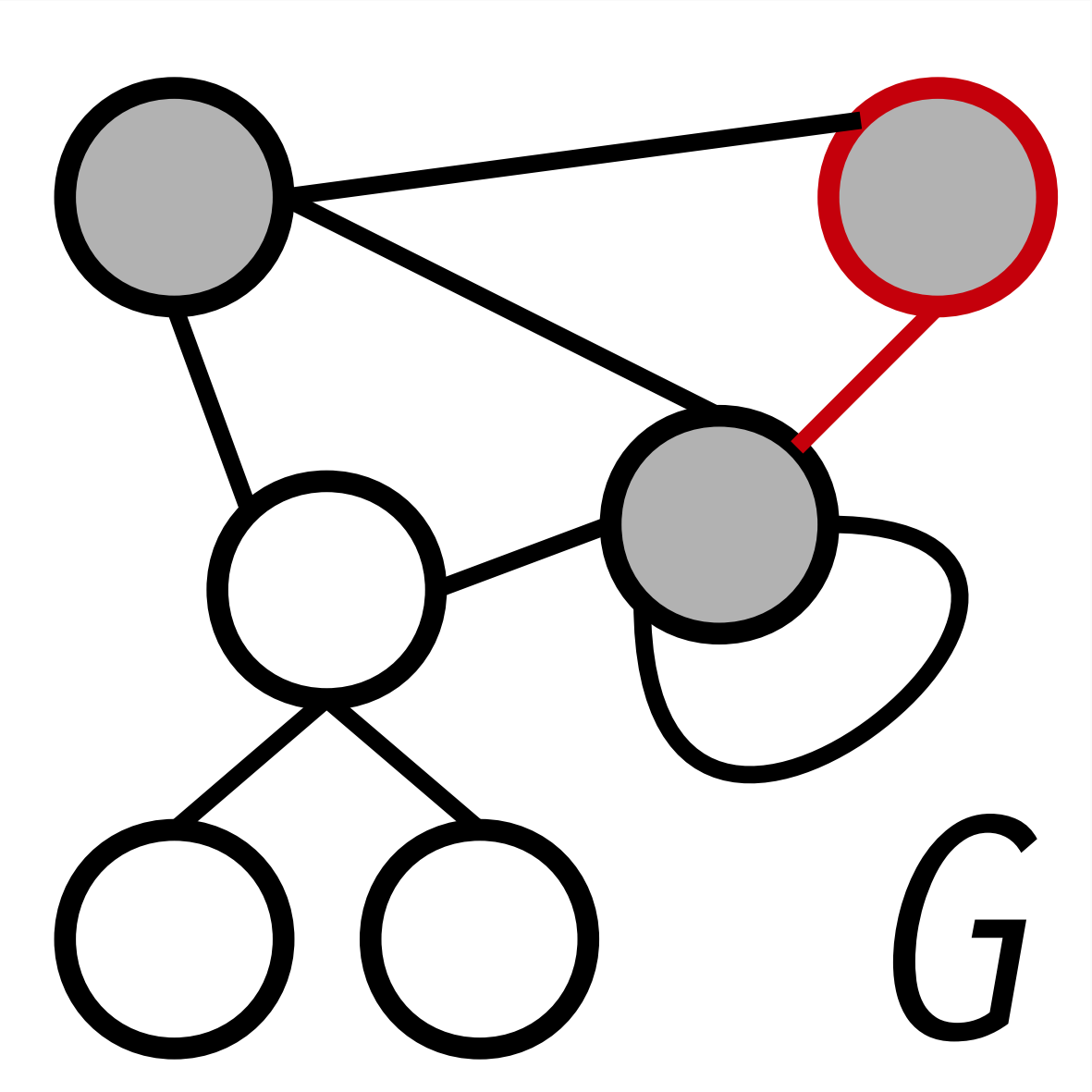
Example



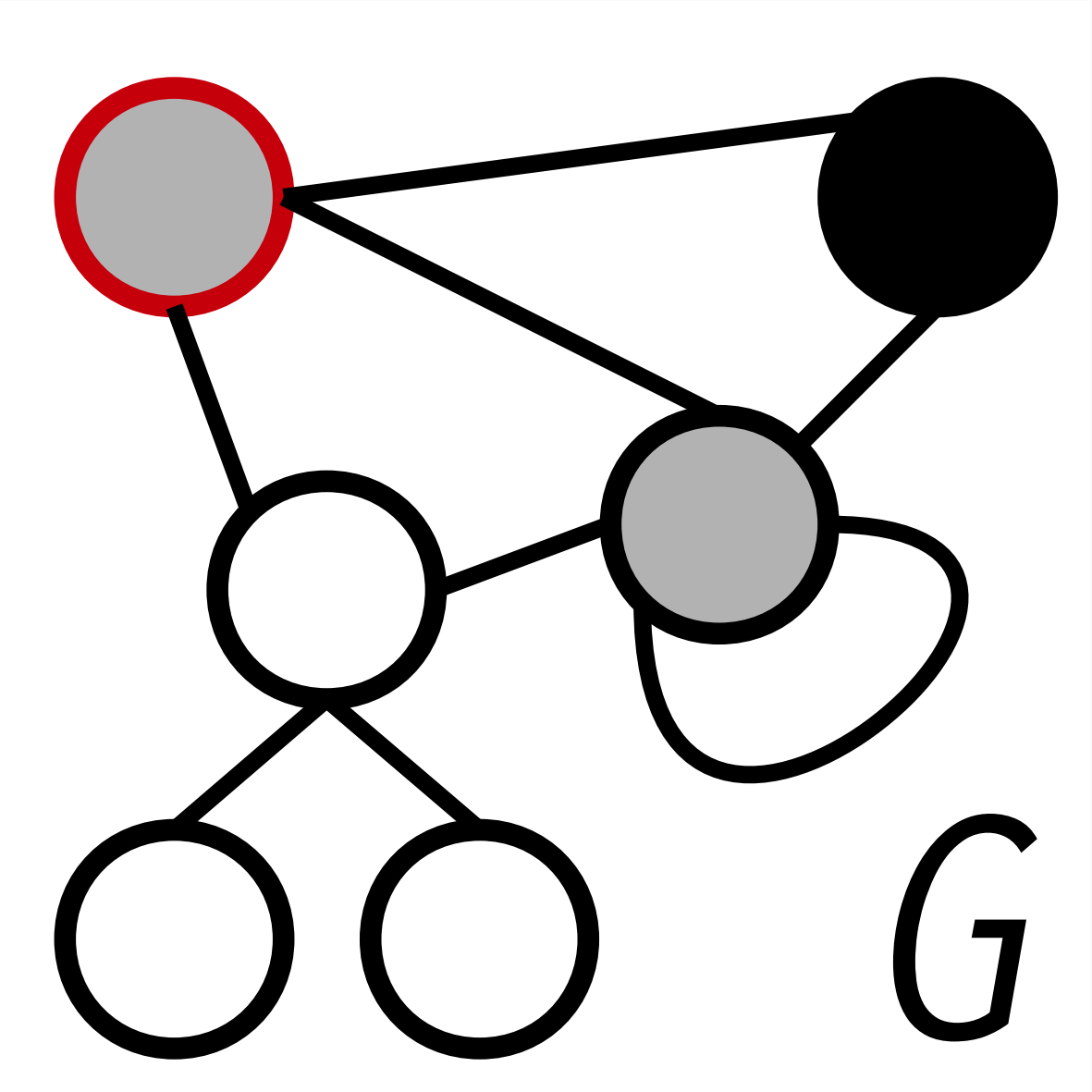
Example



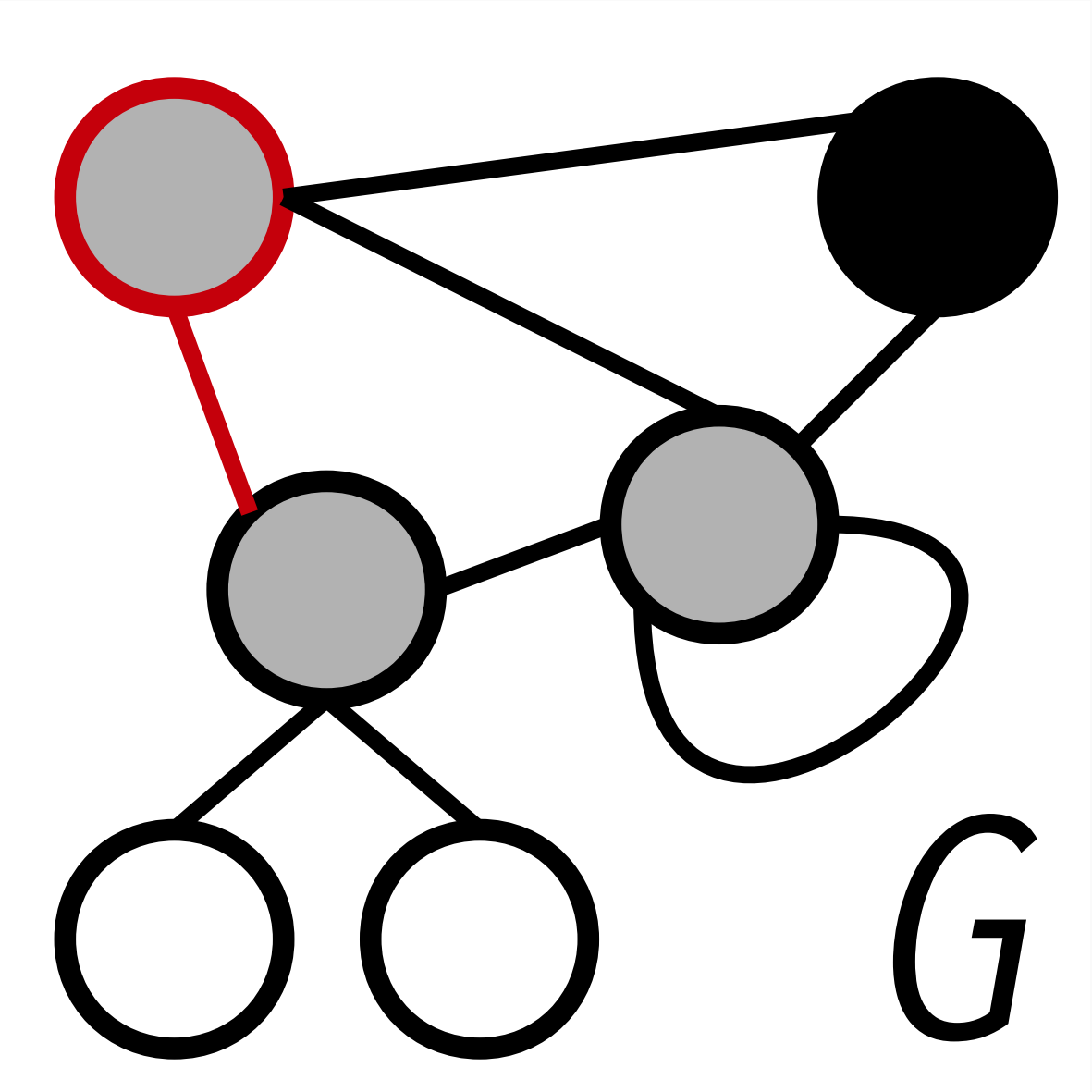
Example



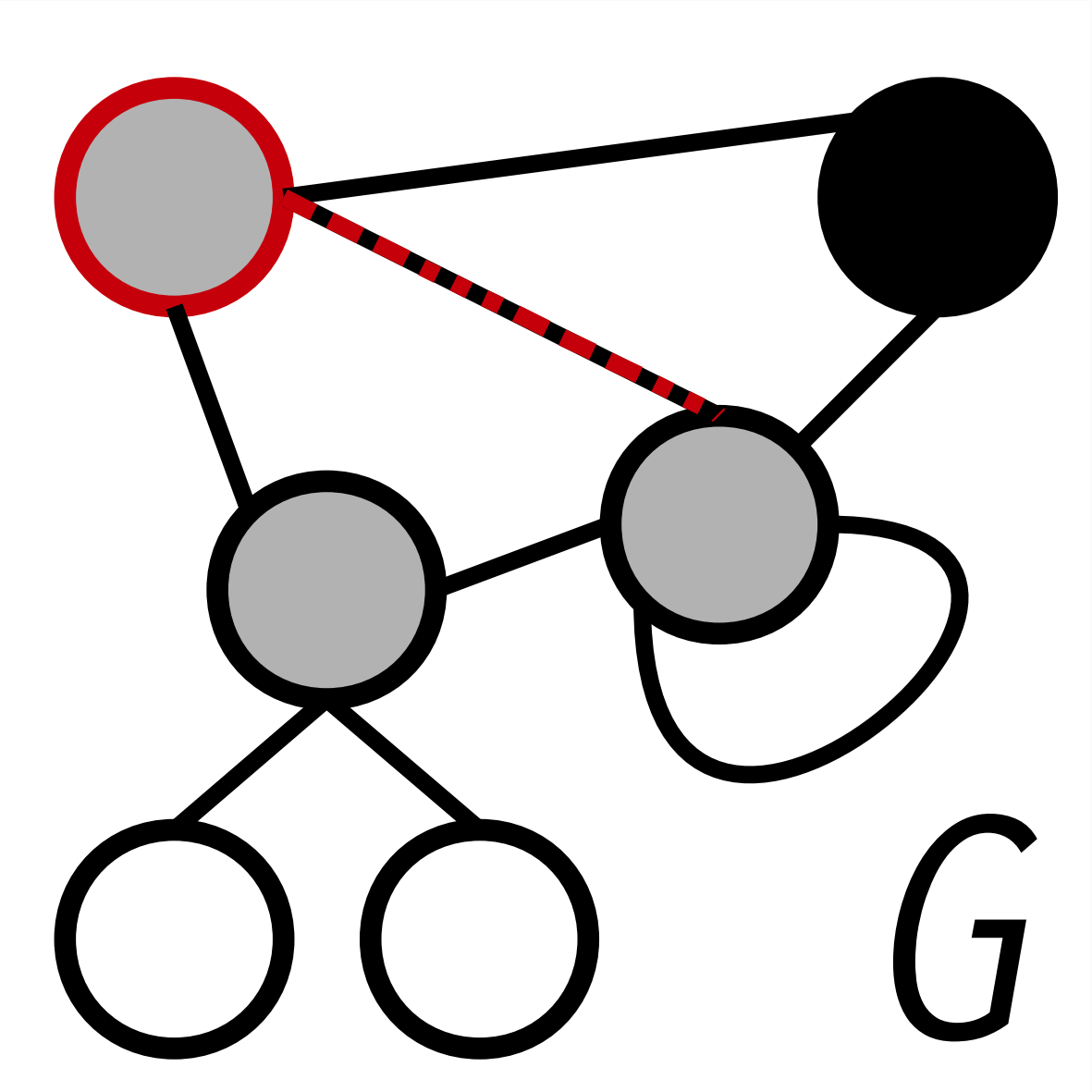
Example



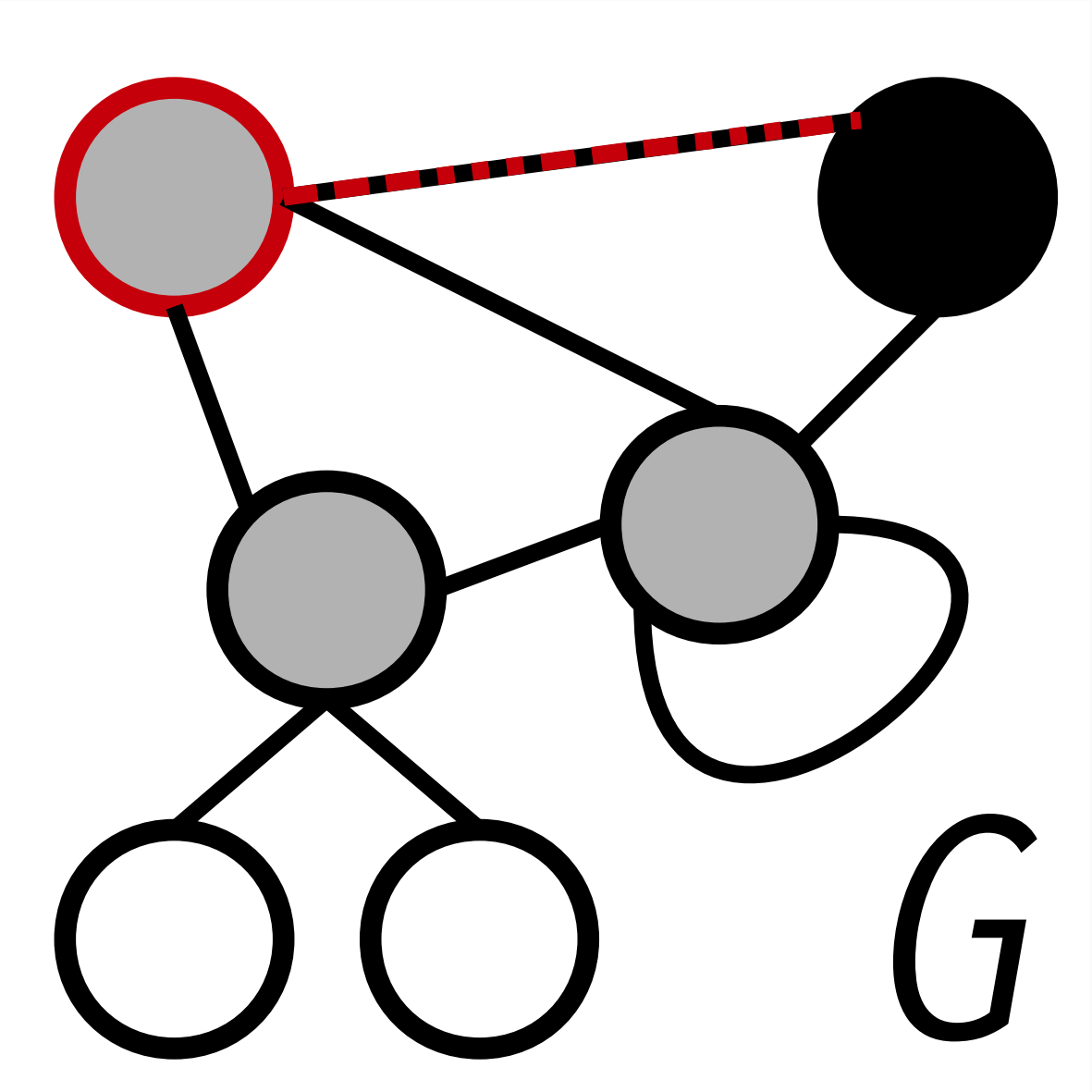
Example



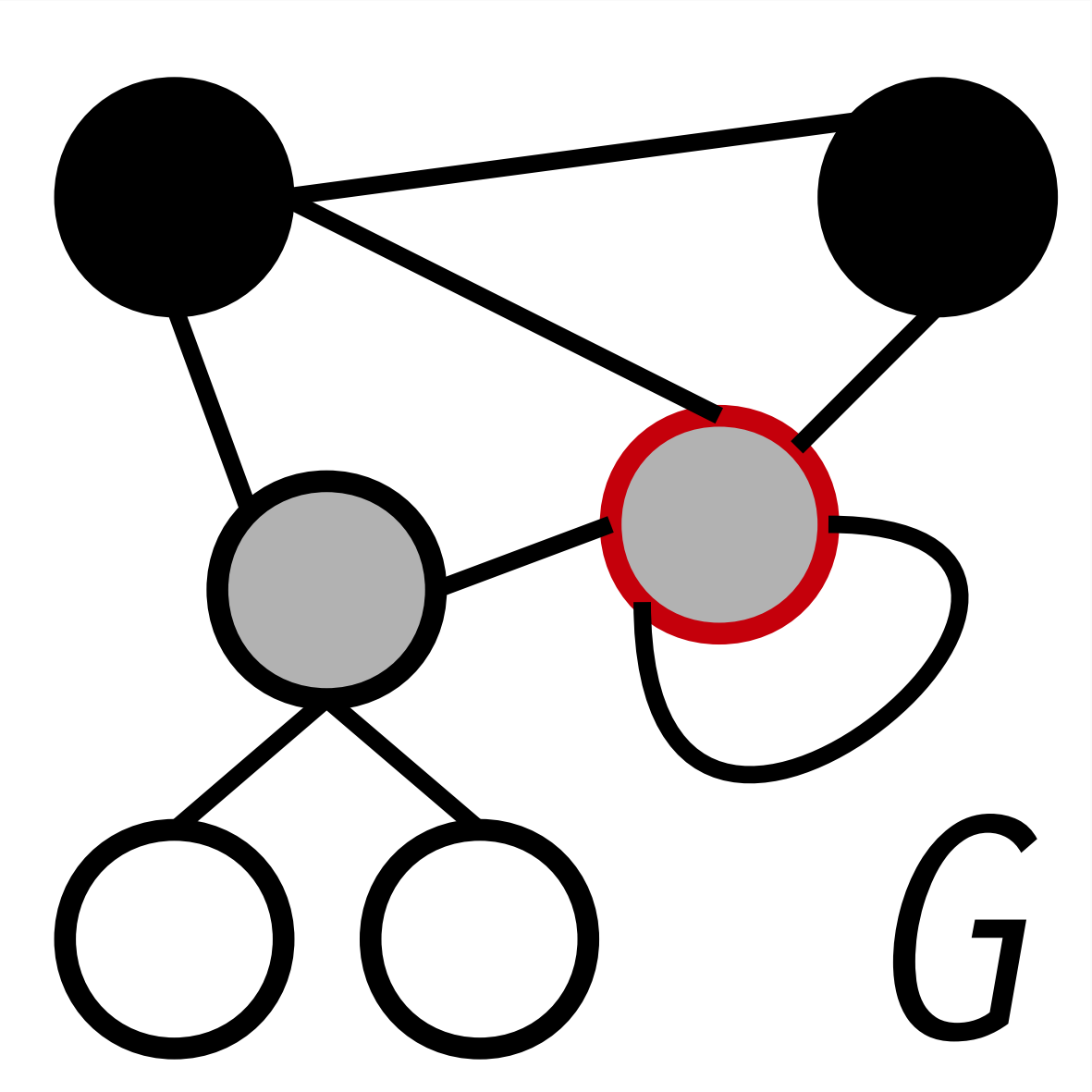
Example



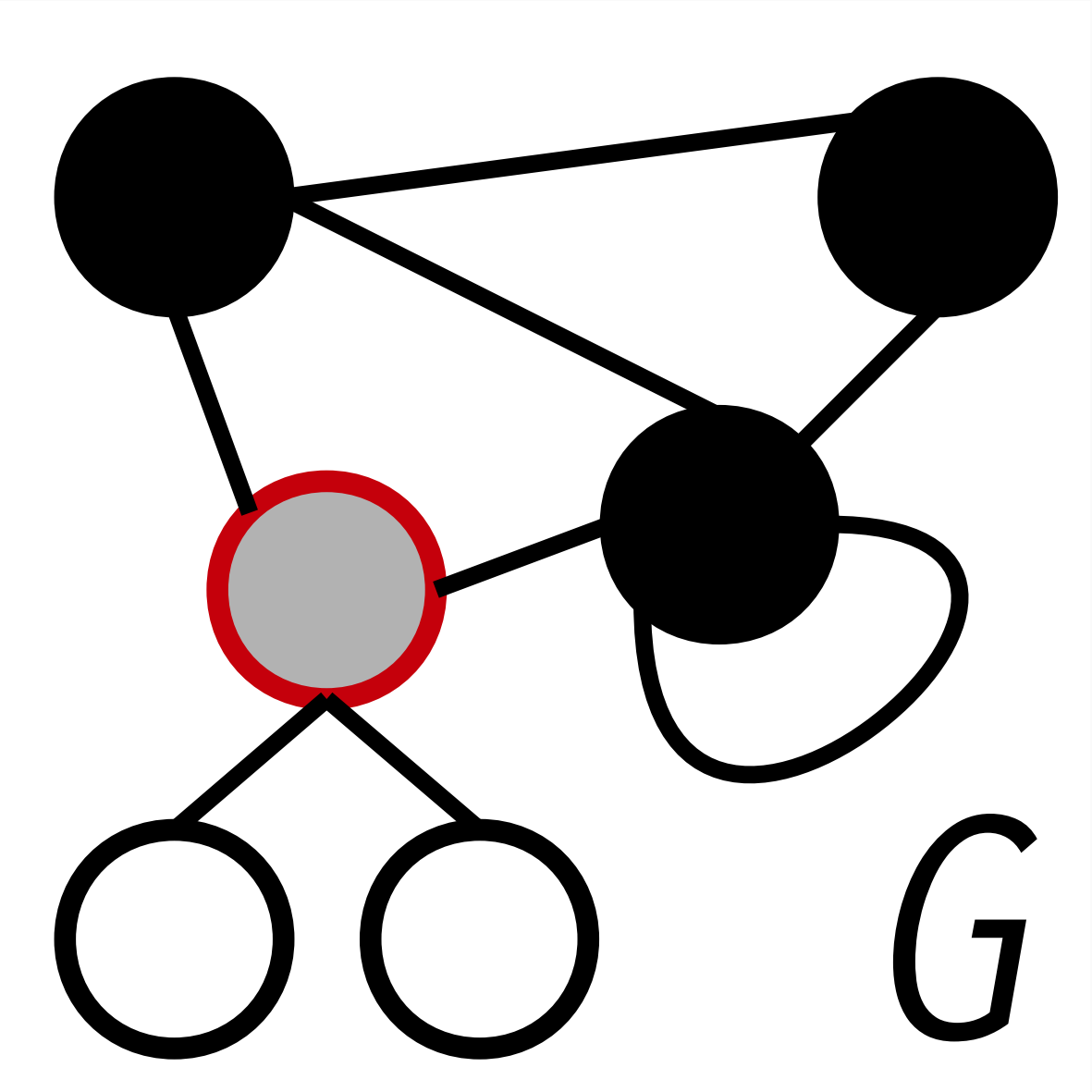
Example



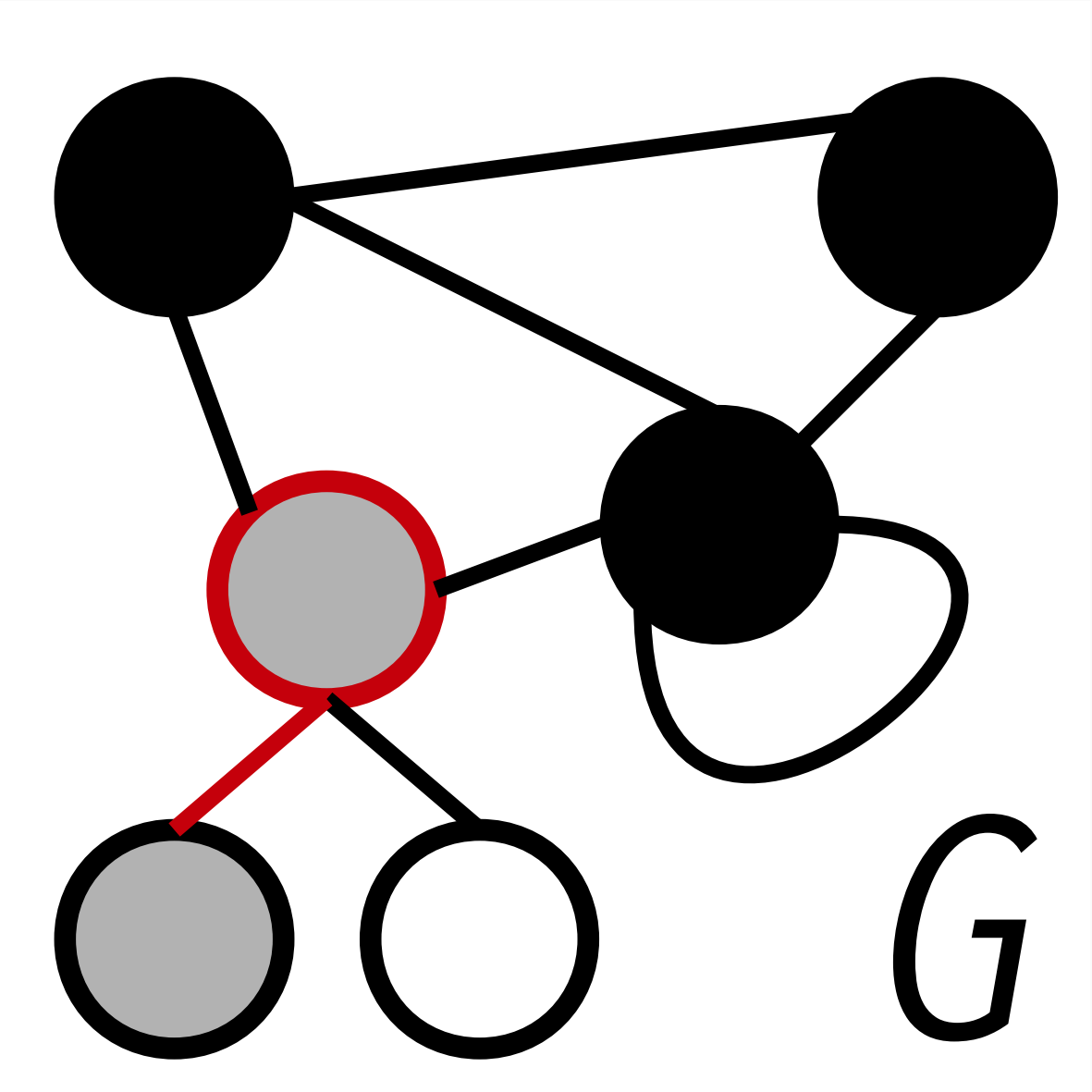
Example



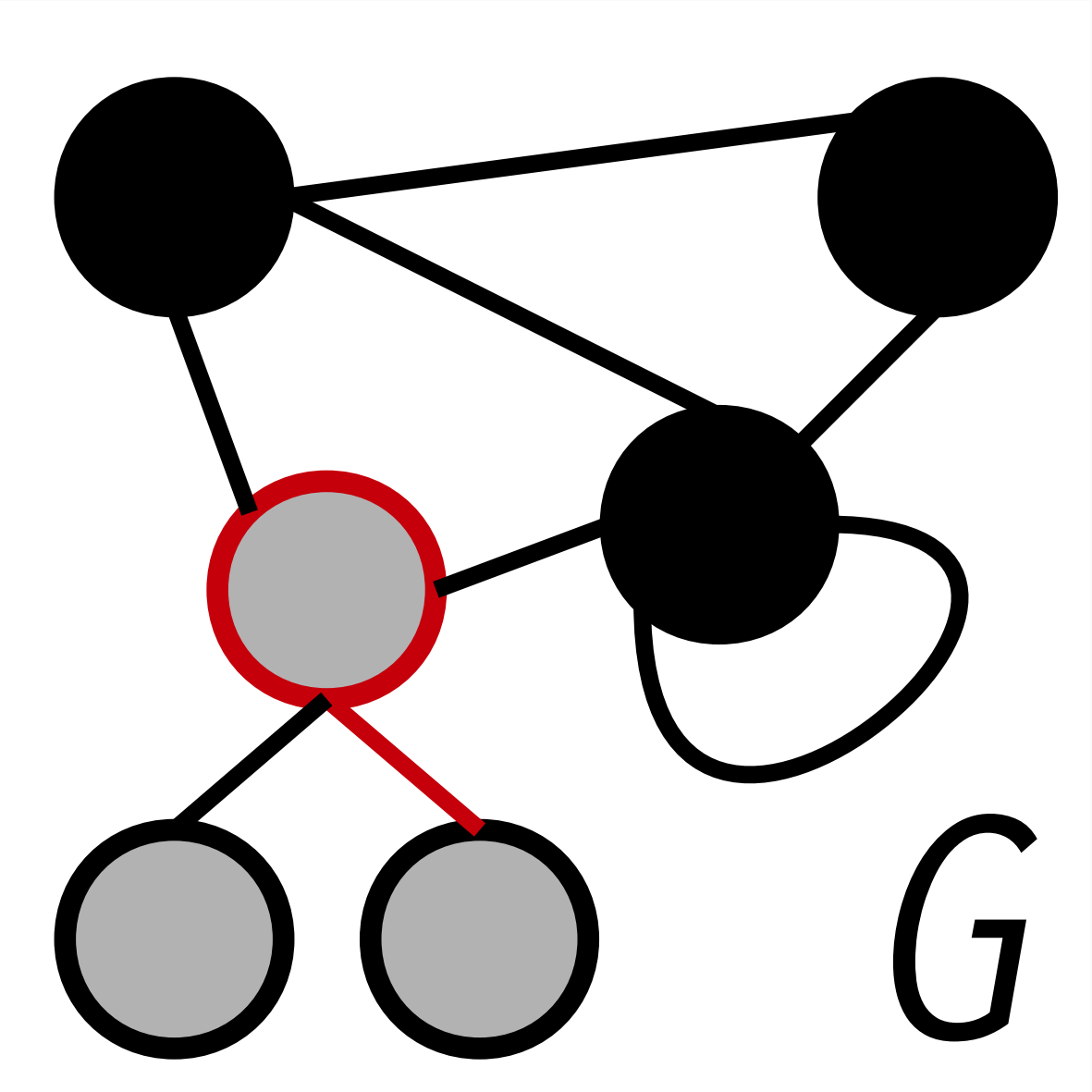
Example



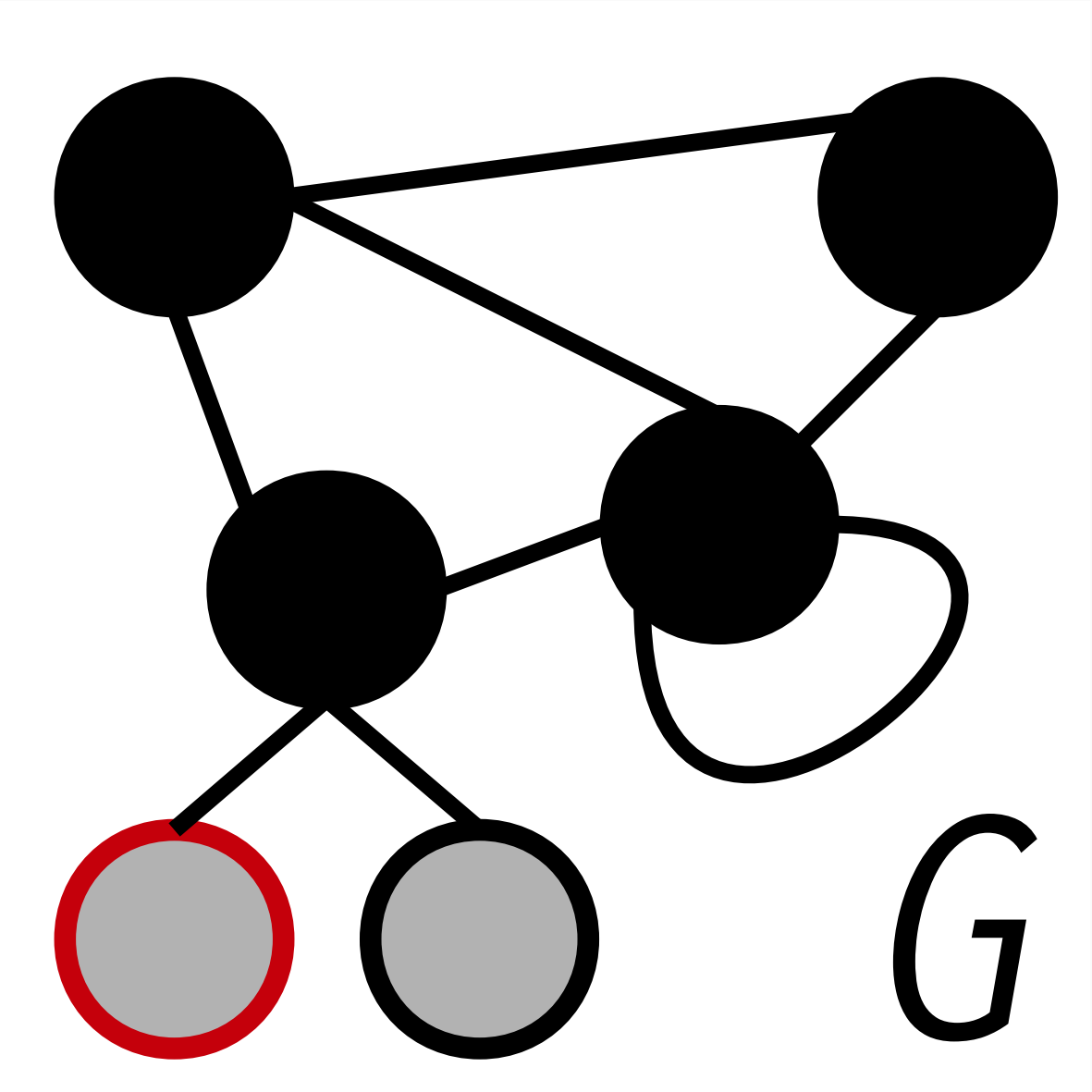
Example



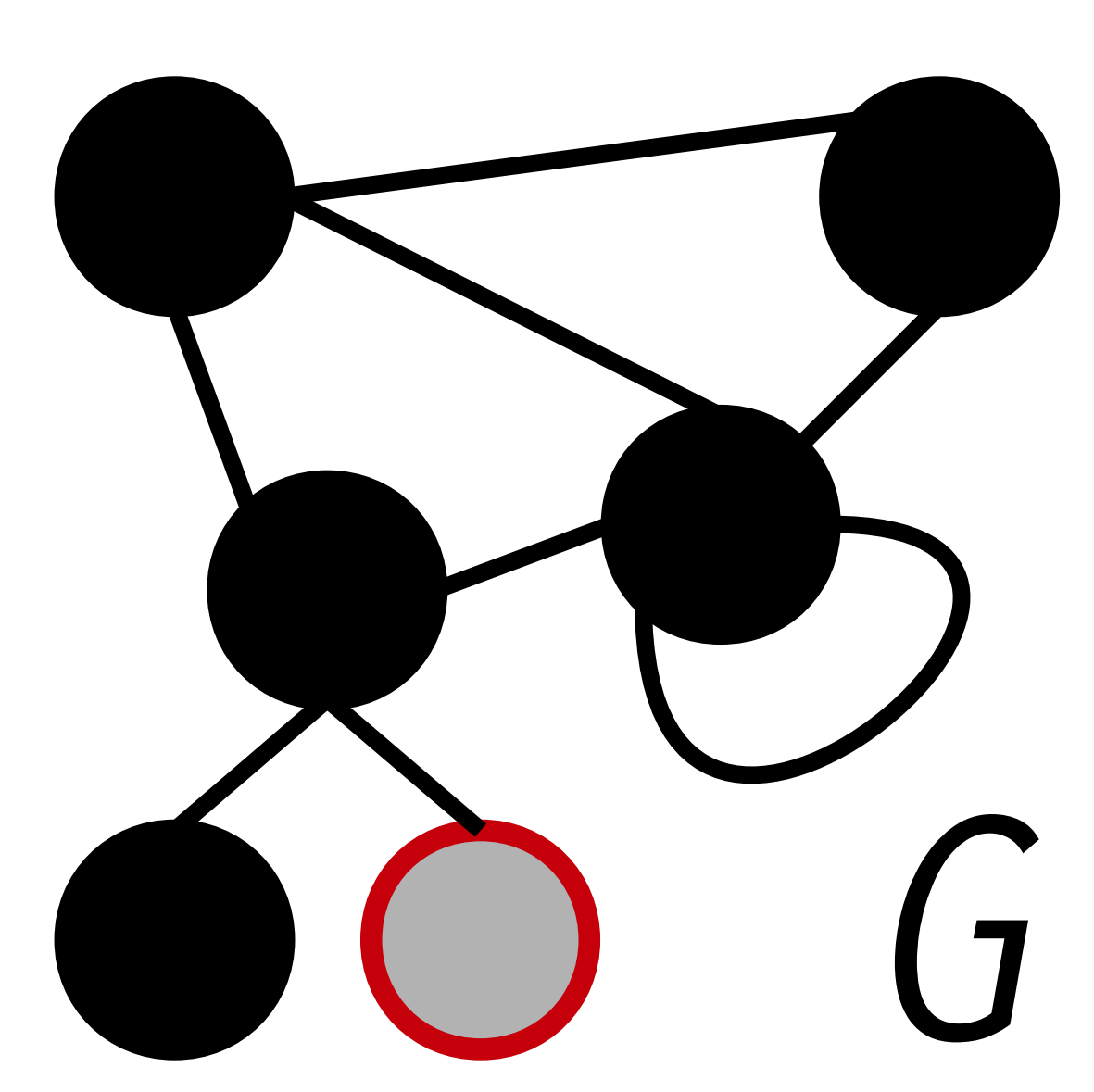
Example



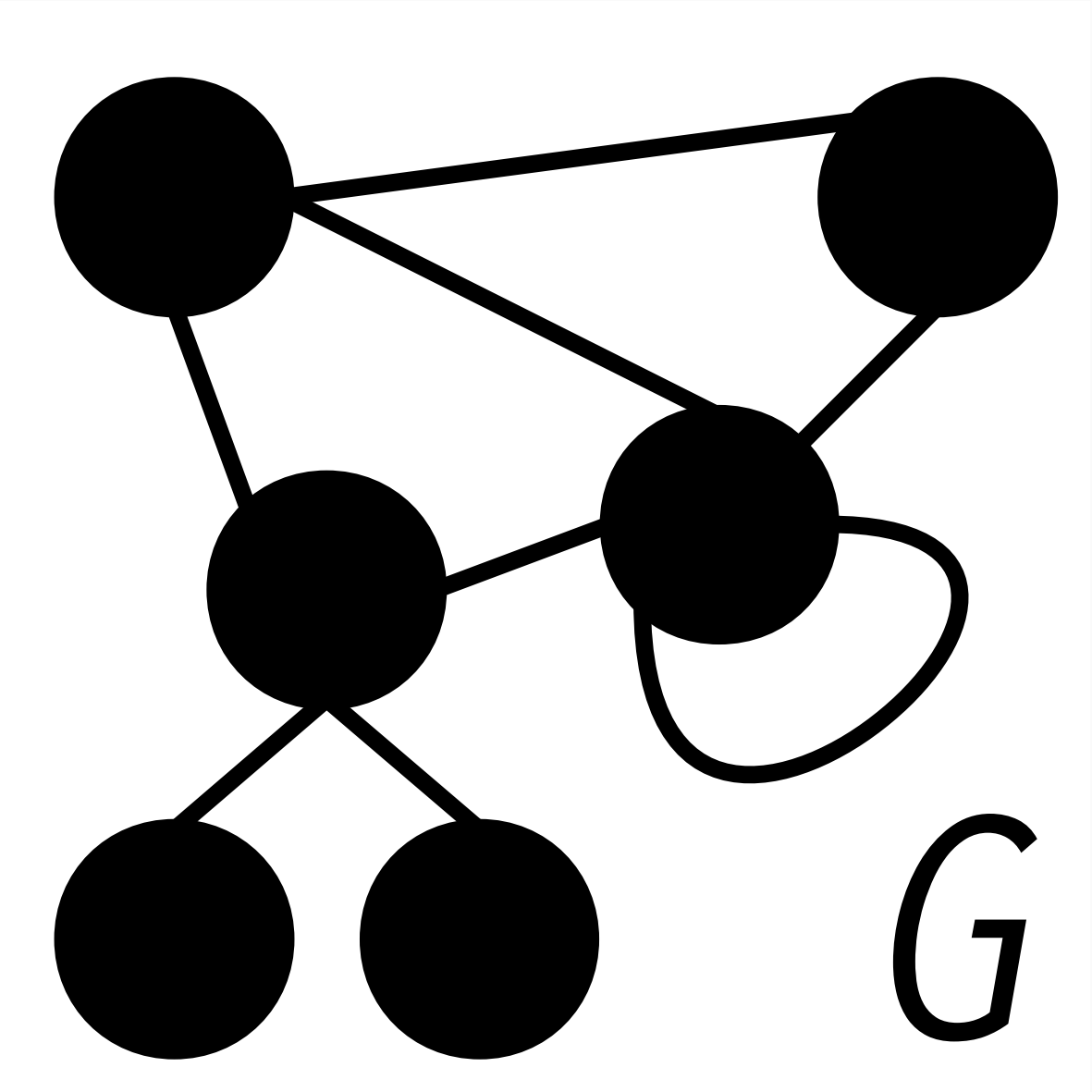
Example



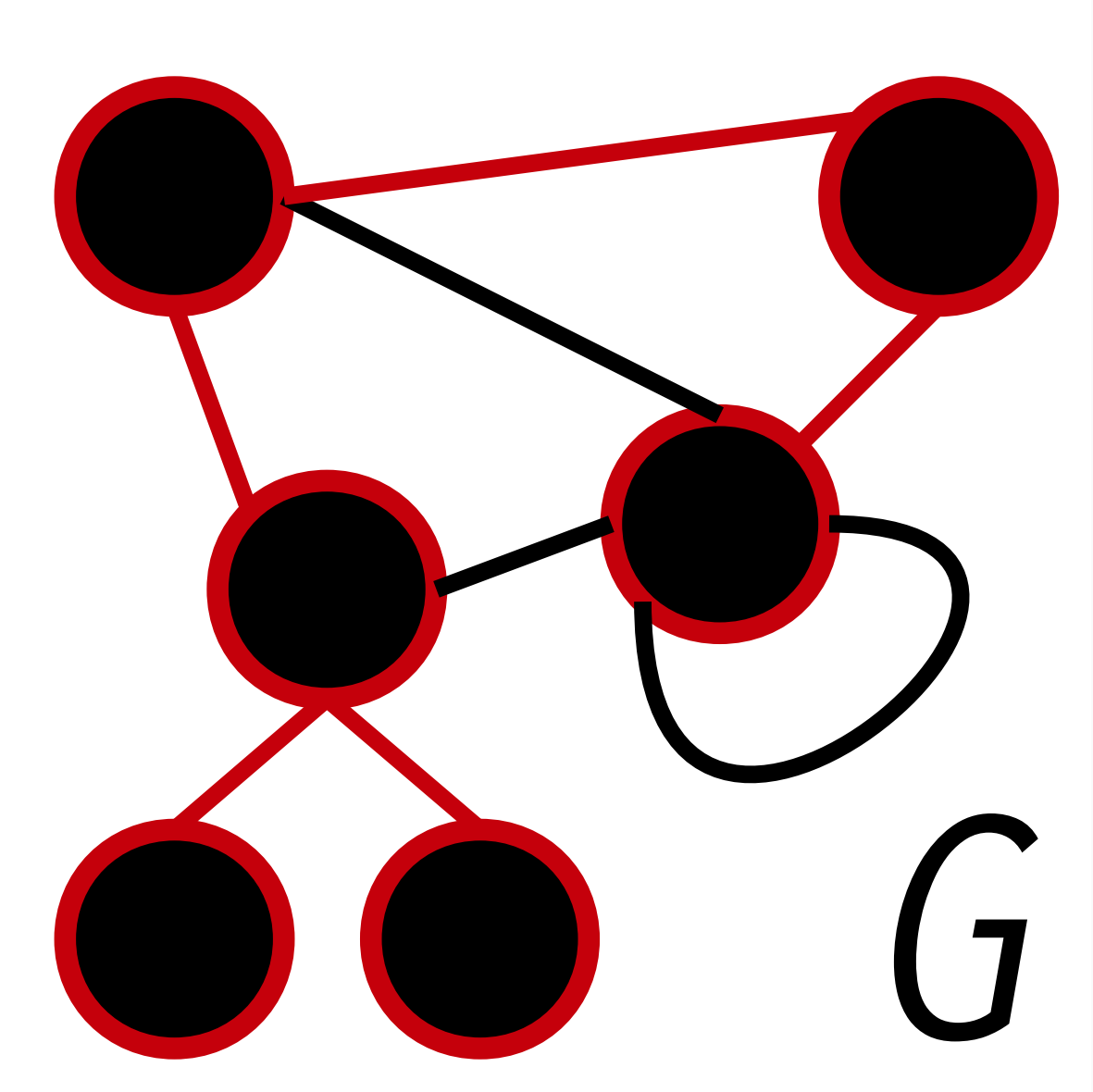
Example



Example



Example



Analysis

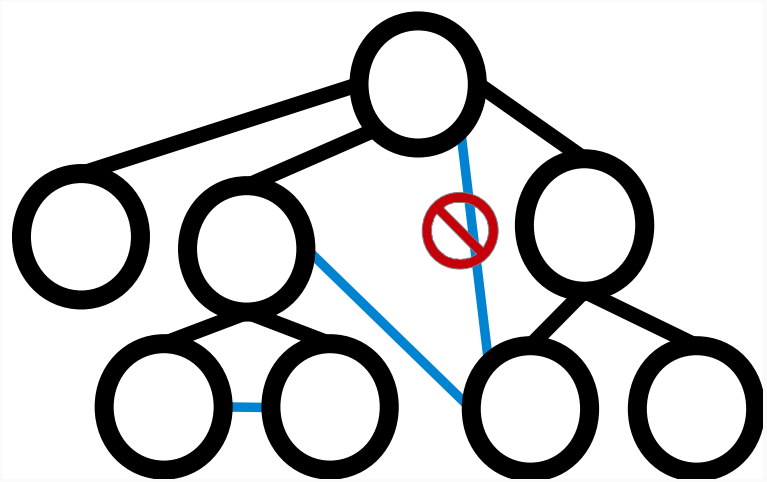
Remark: Memorize the nodes' levels using a counter.

Time Complexity: $\mathcal{O}(|E| + |V|)$ (Why?)

Property 1: BFS $\rightarrow$ spanning tree $T = \langle V, E' \rangle$ (pseudocode?).

Property 2: BFS $\rightarrow$ shortest path from s to $v \in V$ (Proof?).

Property 3: e : edge s.t. $e \in E$ and $e \notin E'$; e connects 2 nodes that are at most 1 level apart (Proof?).



Exercice

Modify the BFS Algorithm to check if G is a bipartite.

Depth-First Search

Depth-First Search

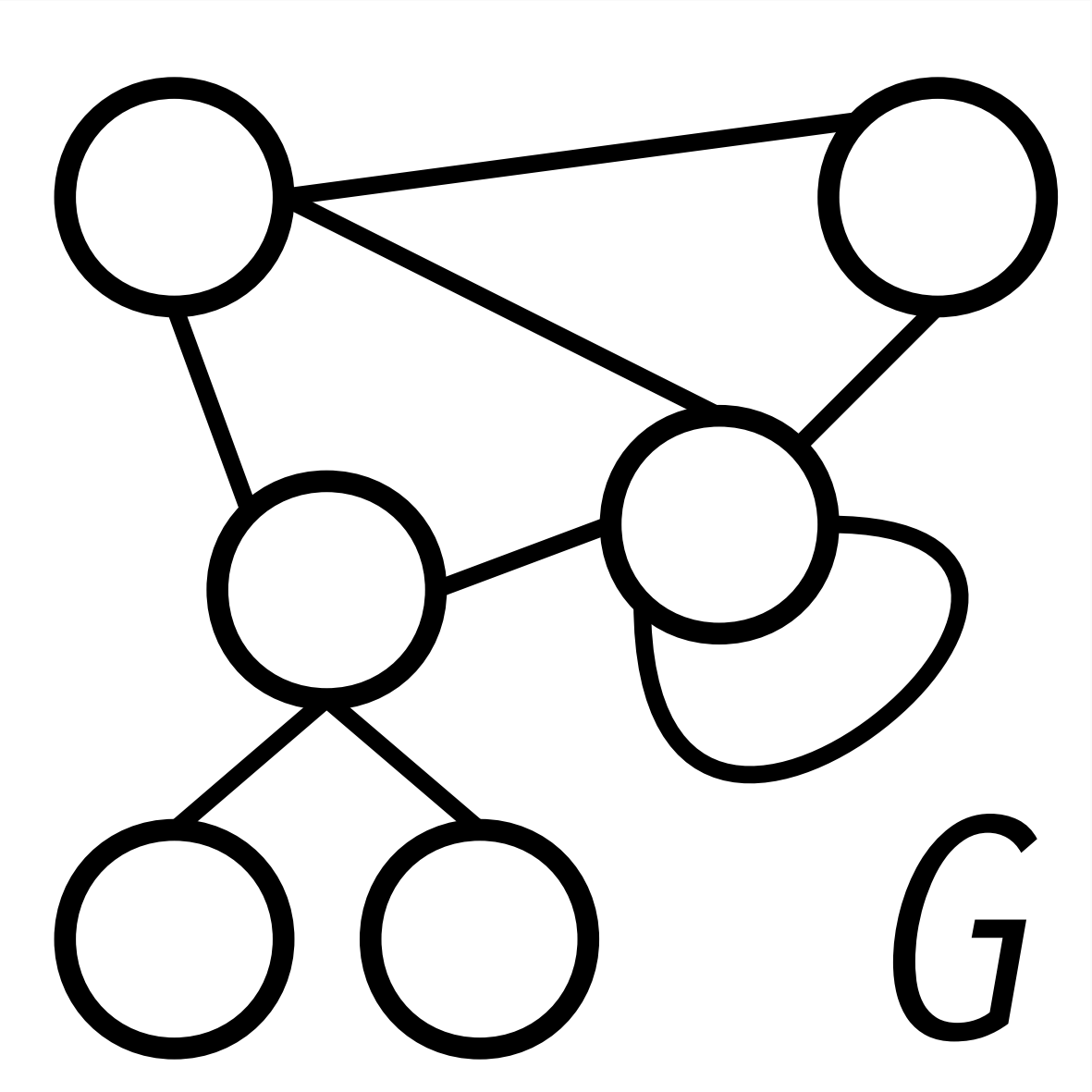
- Algorithm to **traverse/search** in **graphs**.
- **Starts** at some **arbitrary node** s.
- Explore **each branch** as **deep** as possible before **backtracking**.

Pseudocode

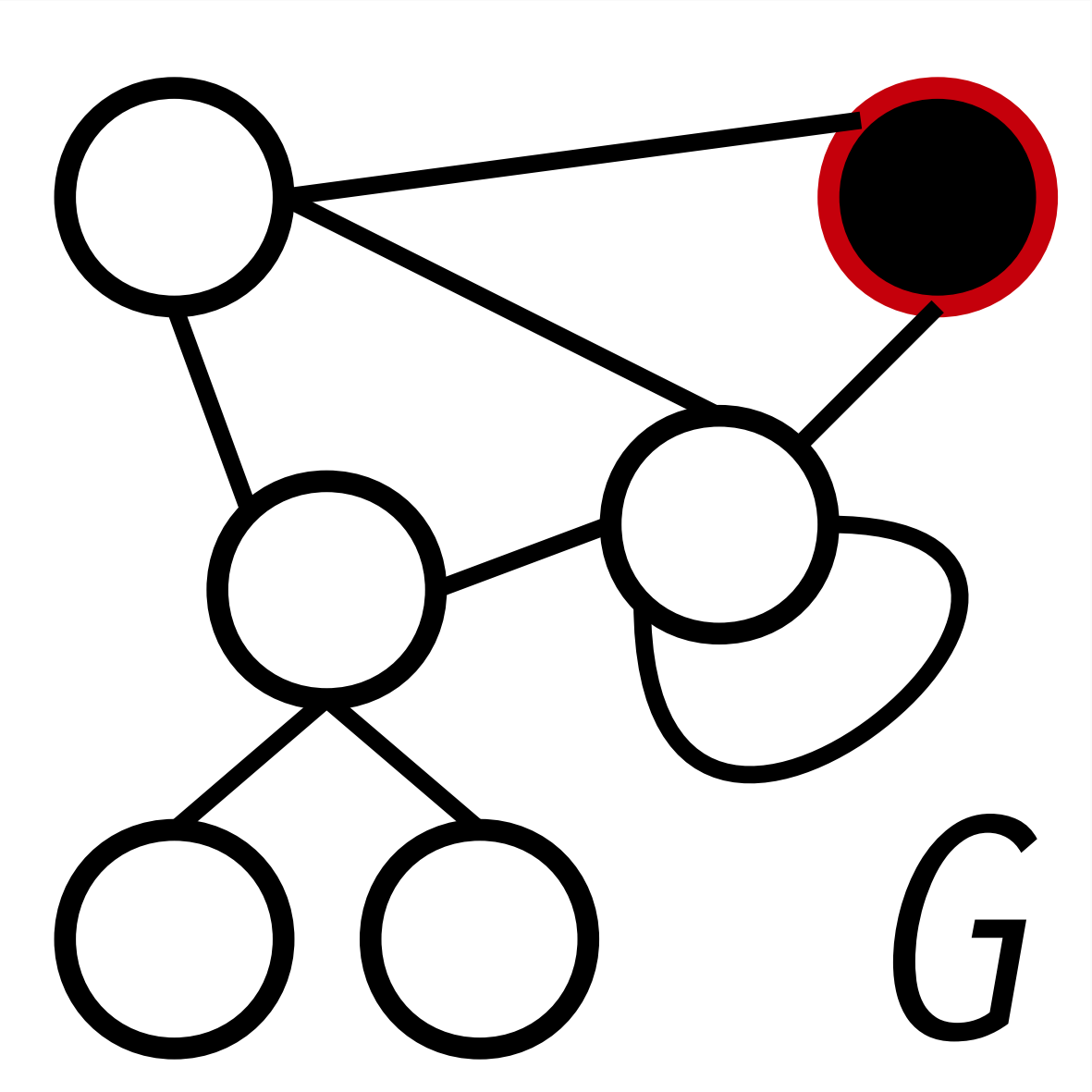
```
function DFS(G = ⟨V, E⟩, s ∈ V)
    for all v ∈ V do
        state[v] ← undiscovered,    parent[v] ← ∅
    end for
    state[s] ← root
    DFS_visit(G, s, parent, state)
    Return parent
end function

function DFS_VISIT(G = ⟨V, E⟩, u ∈ V, parent, state)
    state[u] ← discovered
    for all v ∈ Neighbors(u) do
        if state[v] = undiscovered then
            parent[v] ← u
            DFS_visit(G, v, parent, state)
        end if
    end for
end function
```

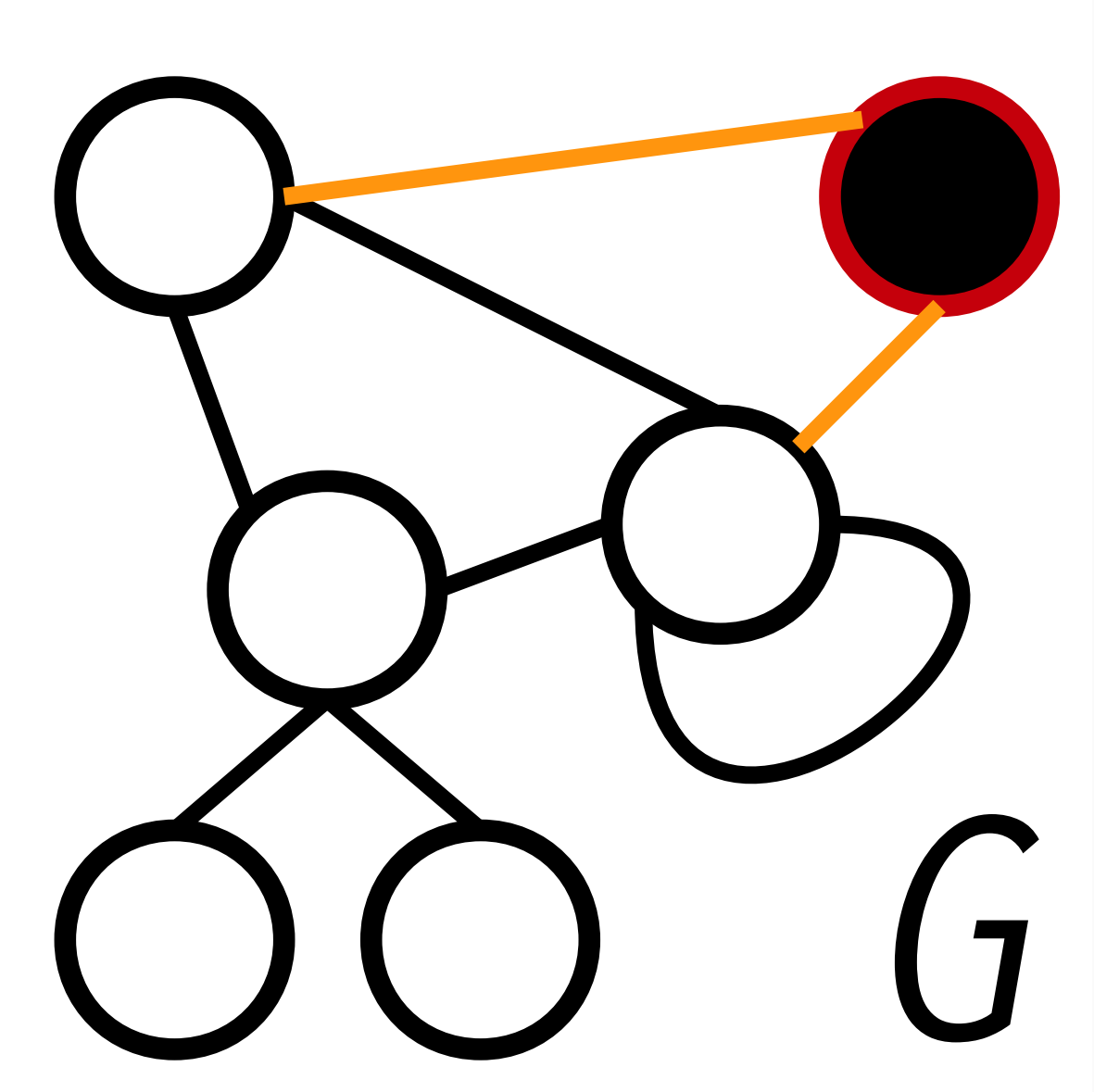
Example



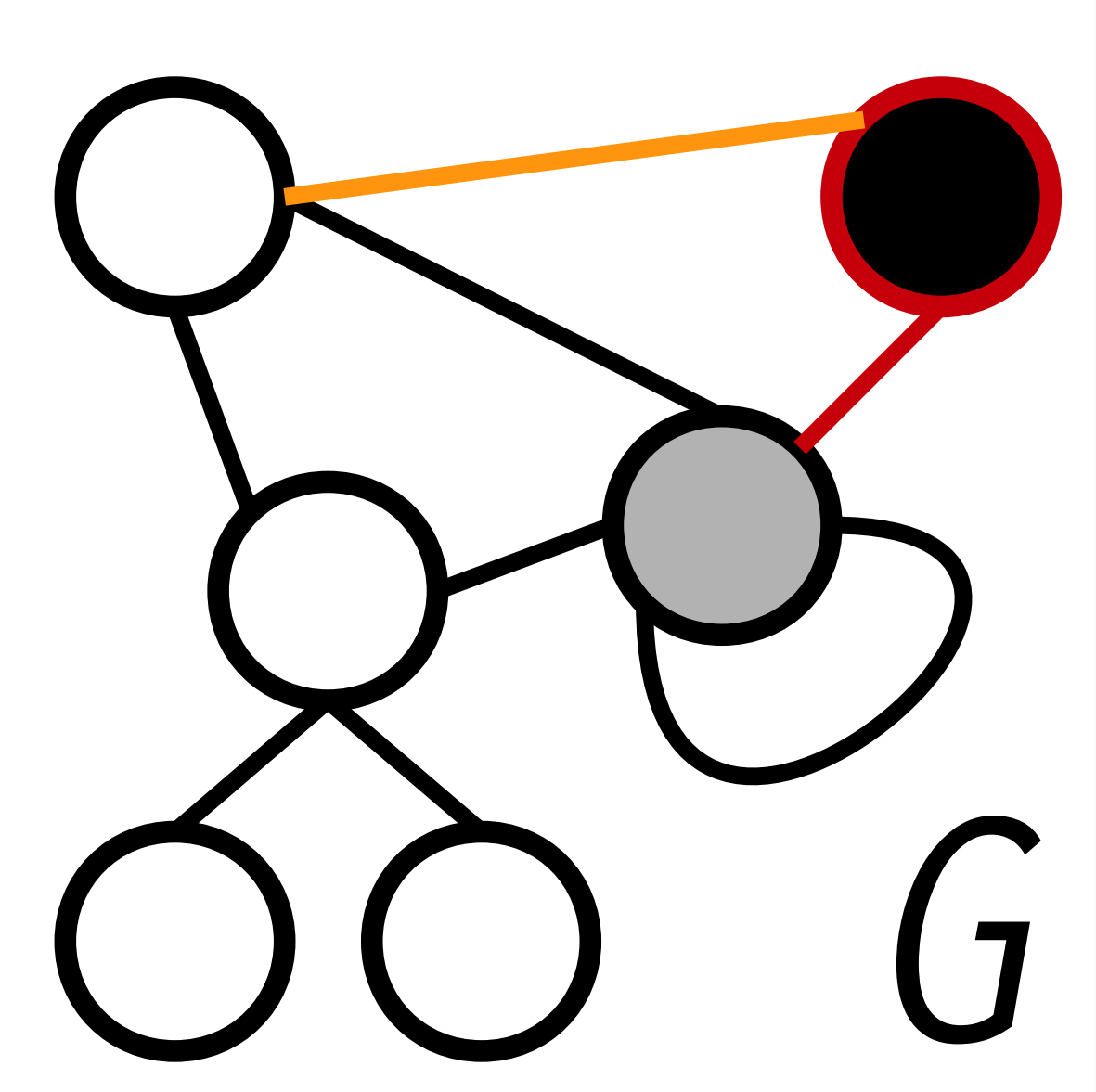
Example



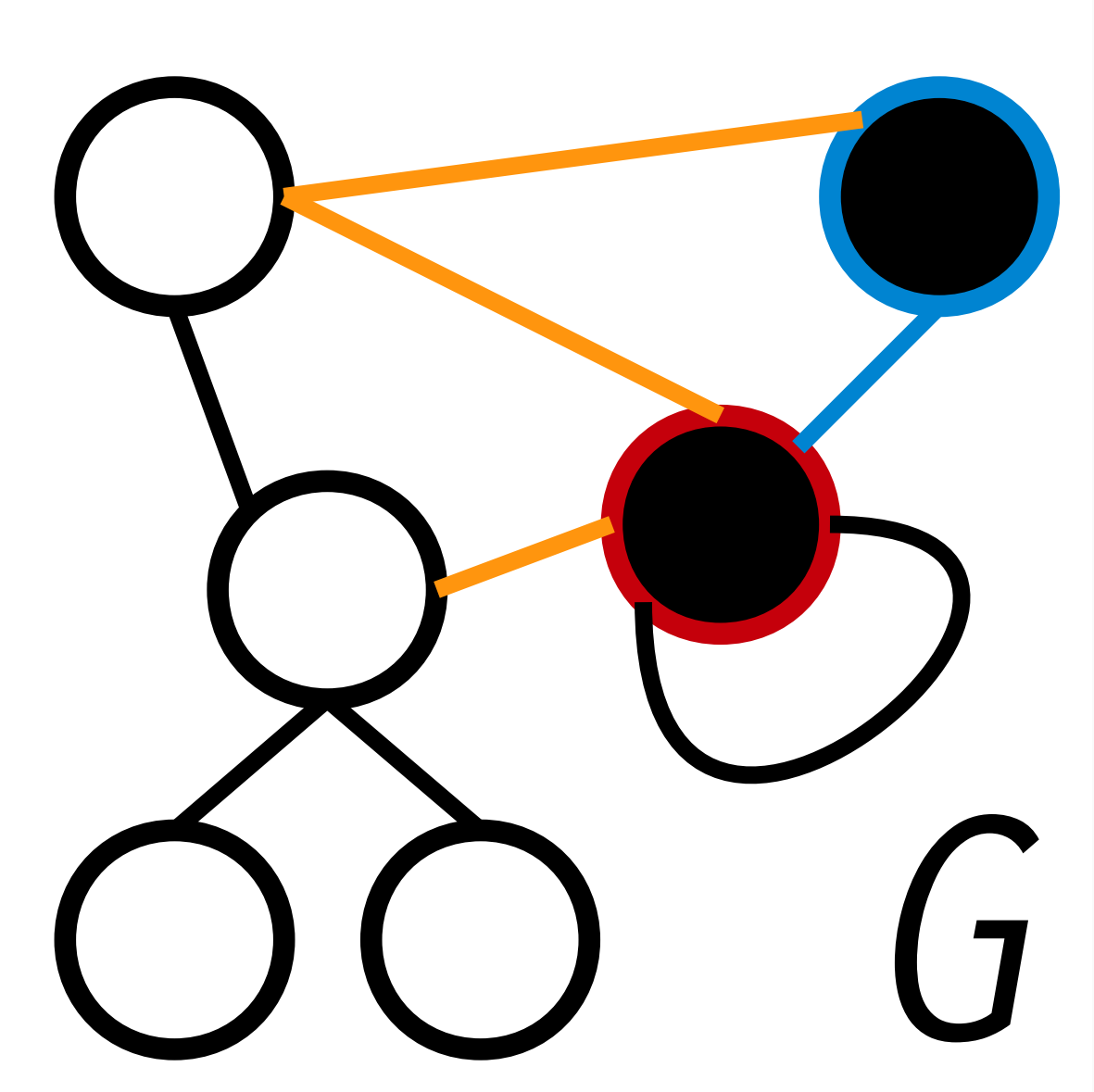
Example



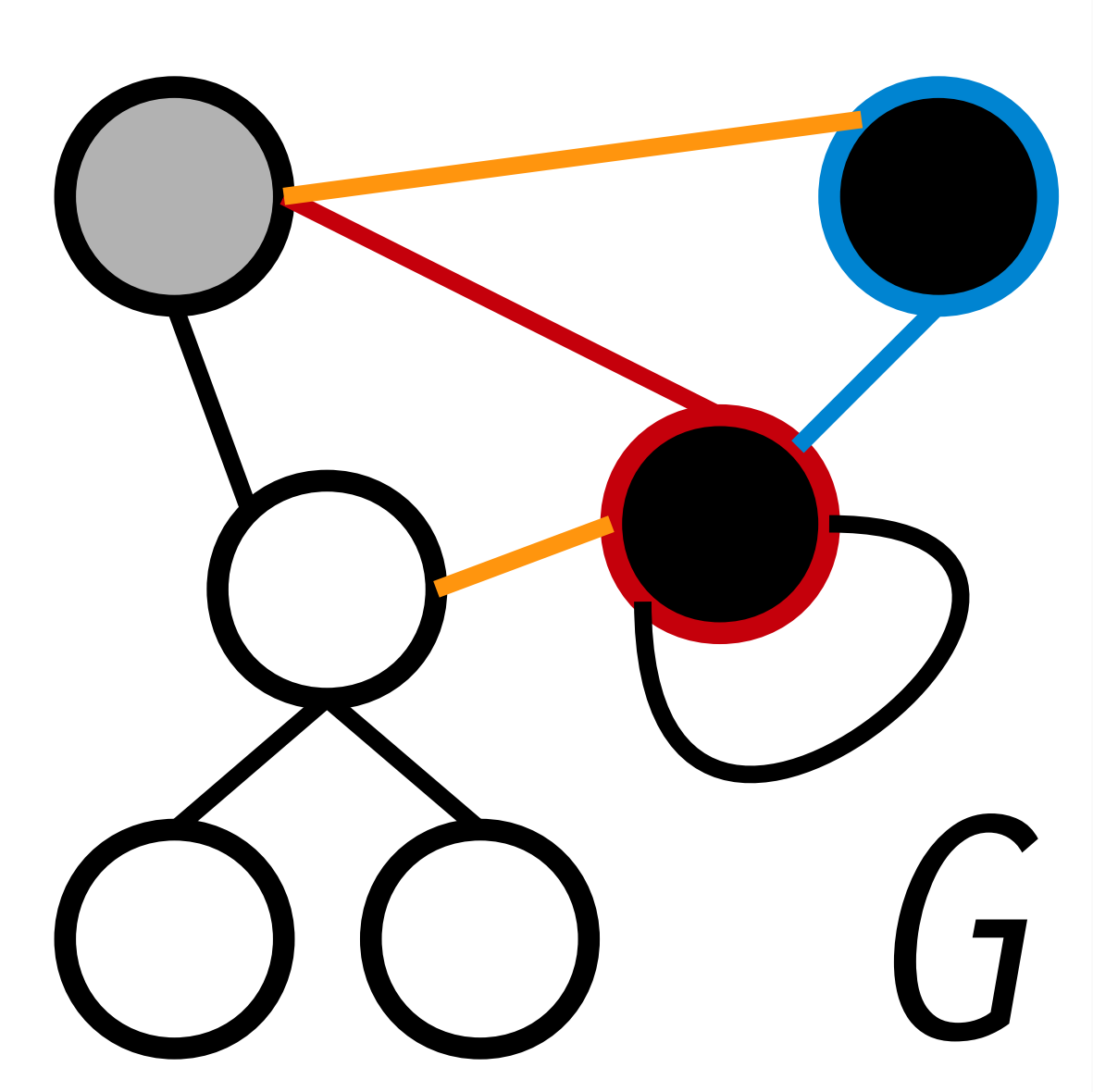
Example



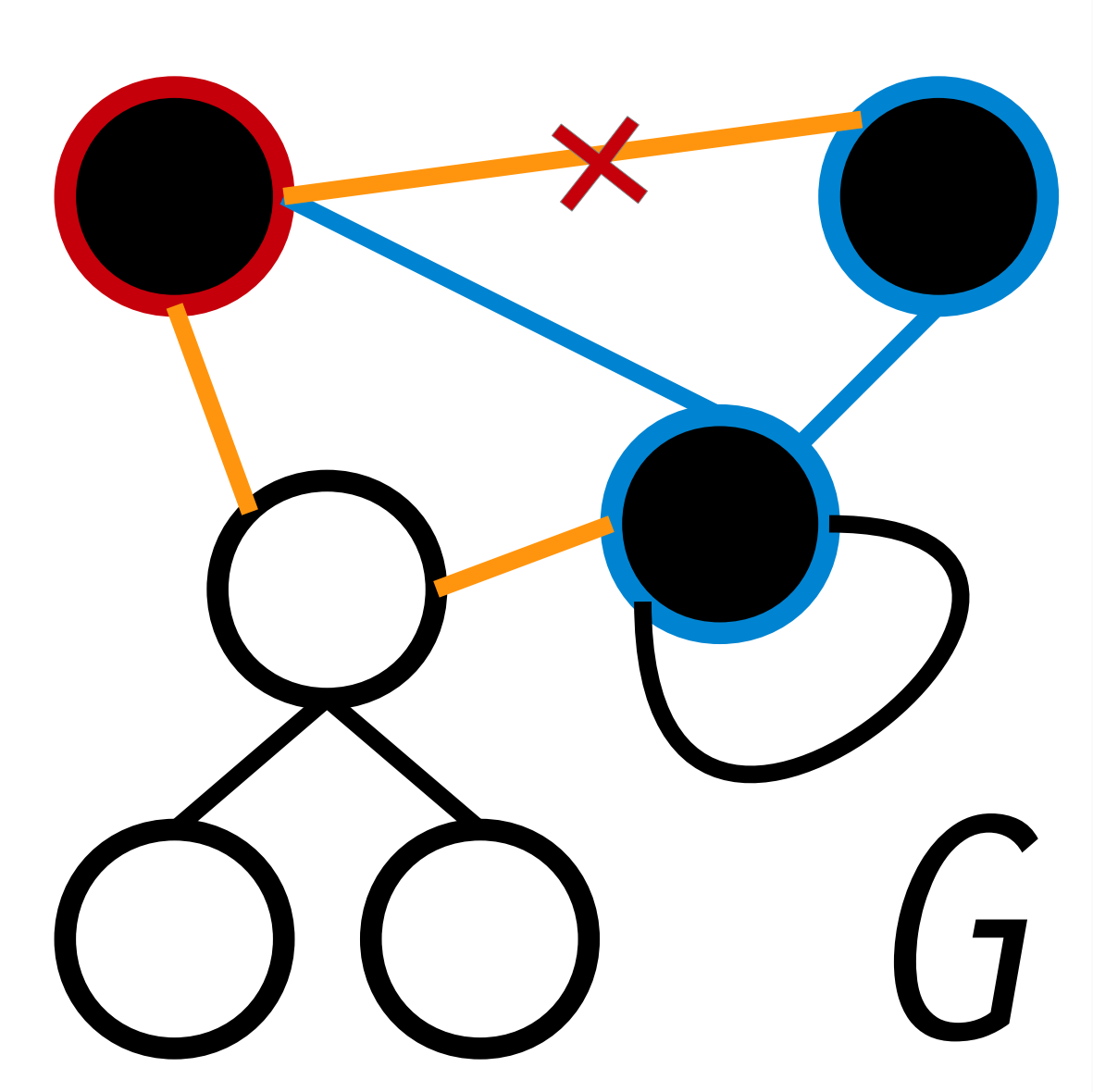
Example



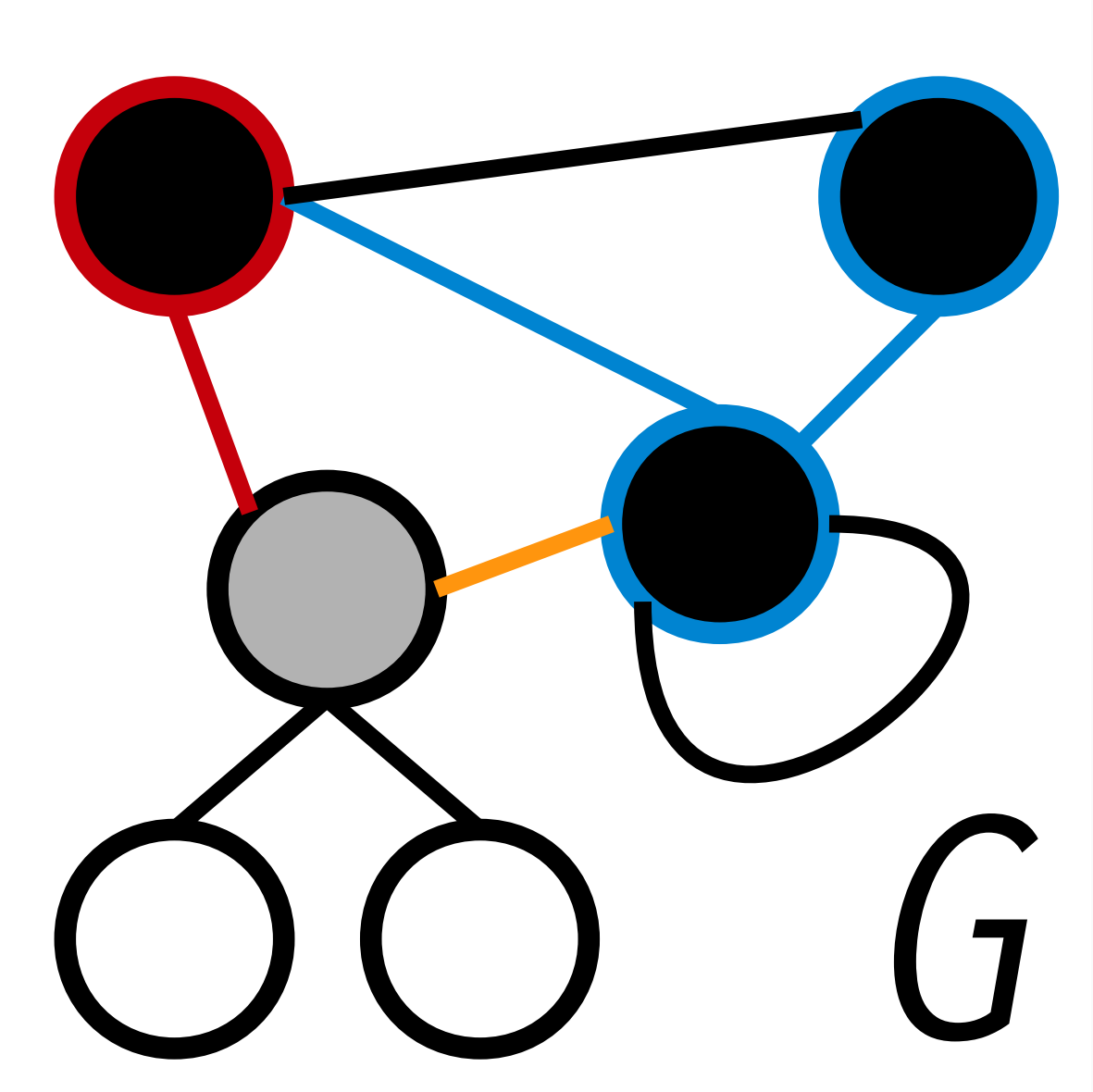
Example



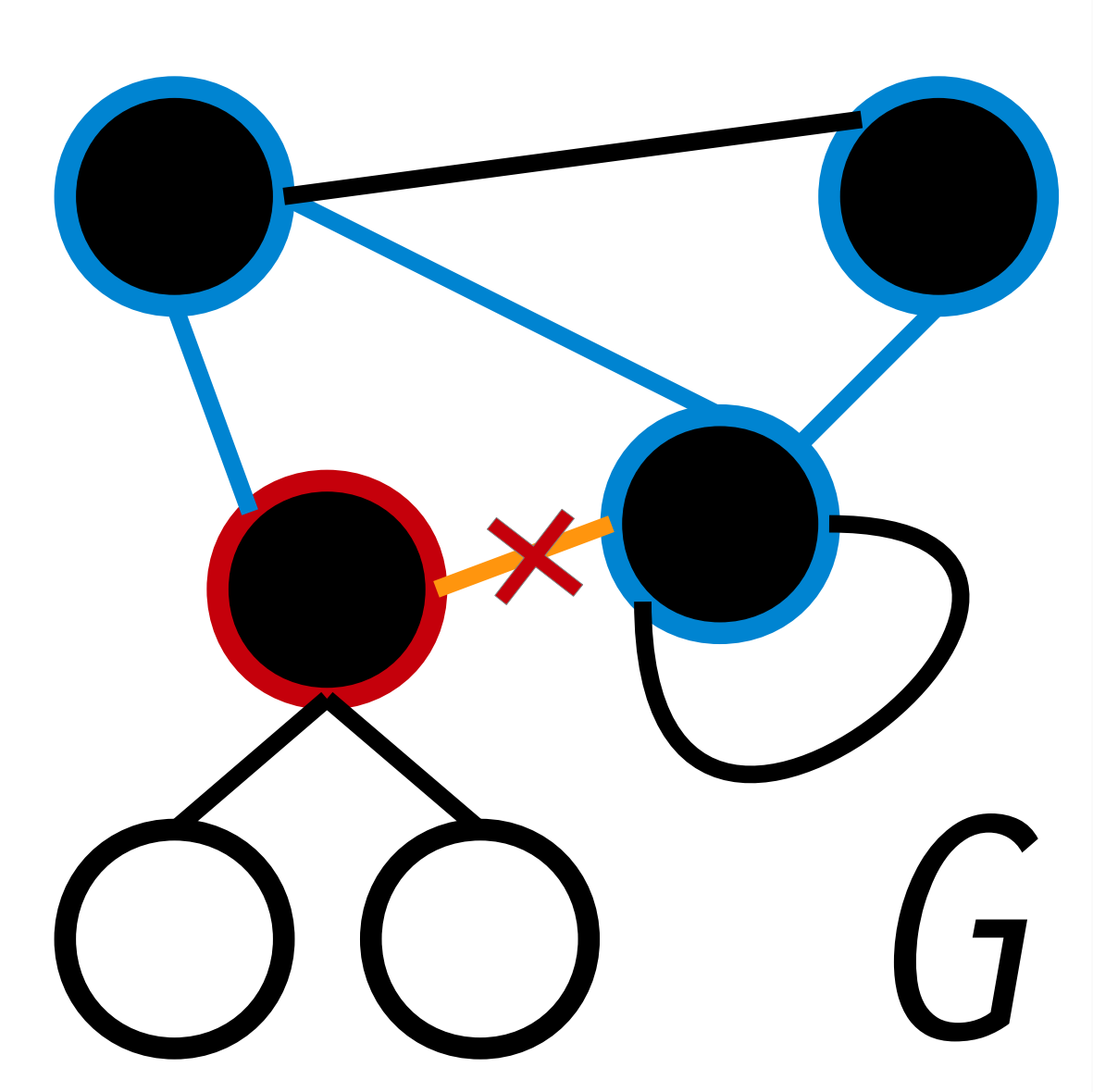
Example



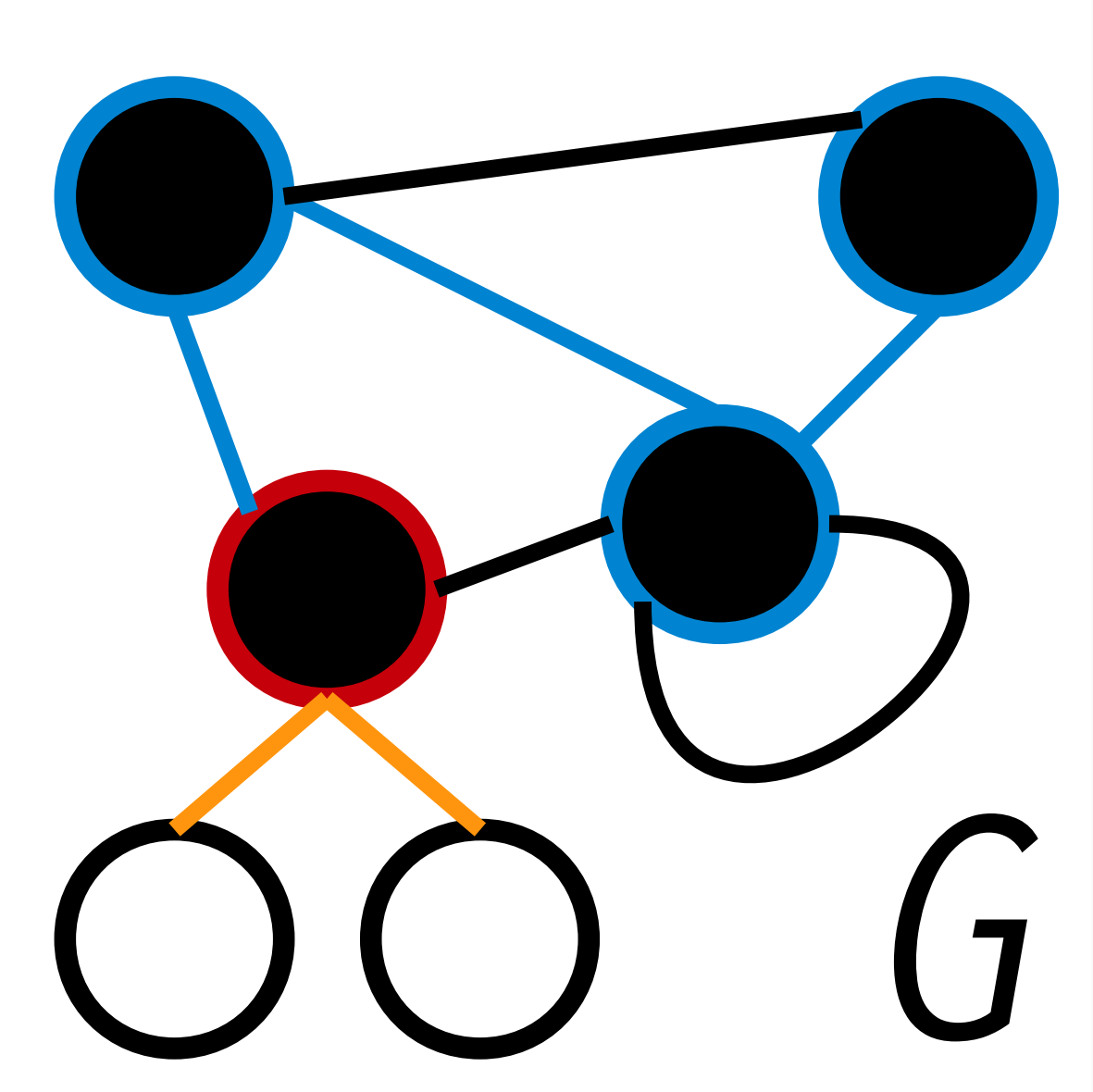
Example



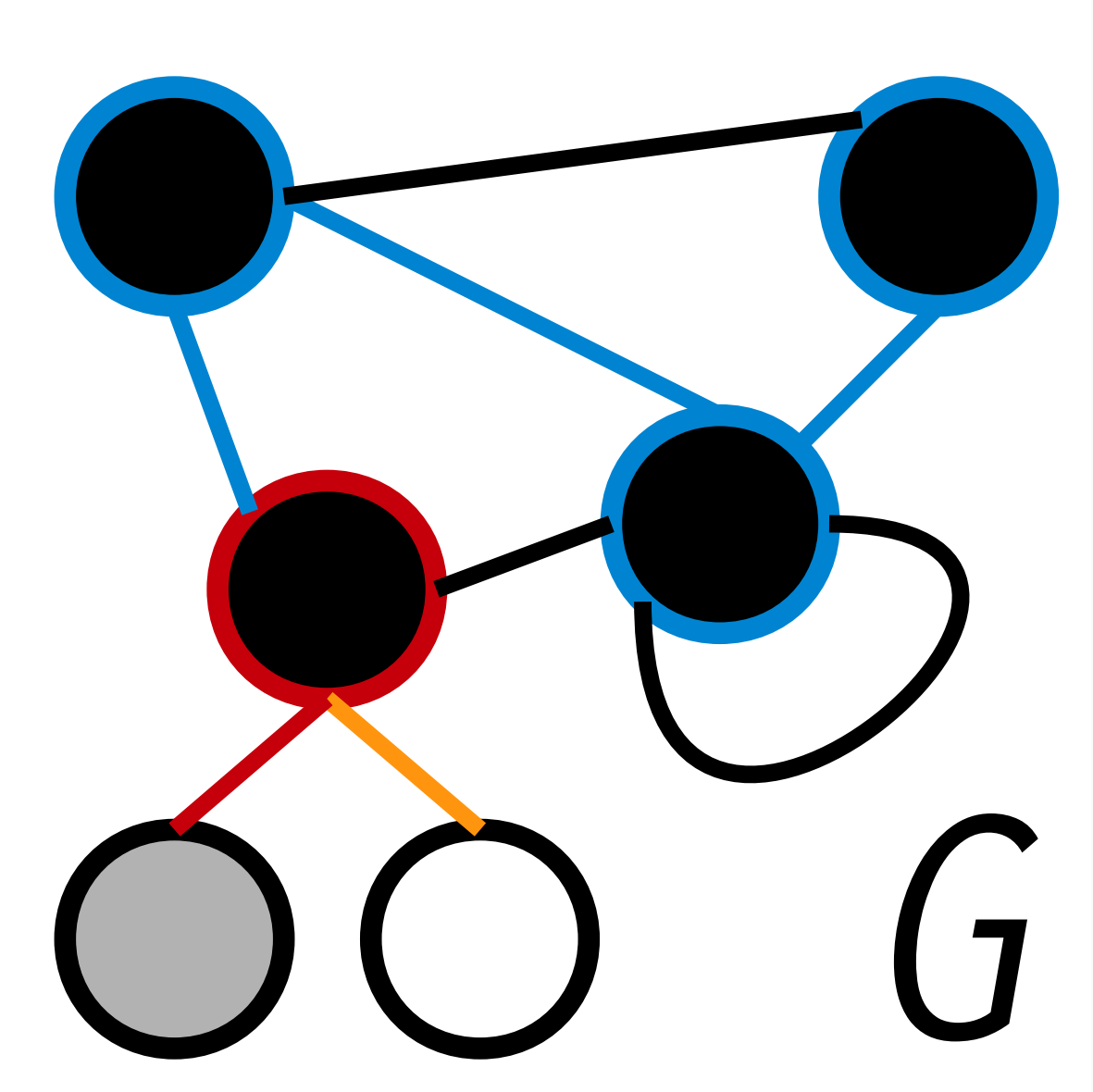
Example



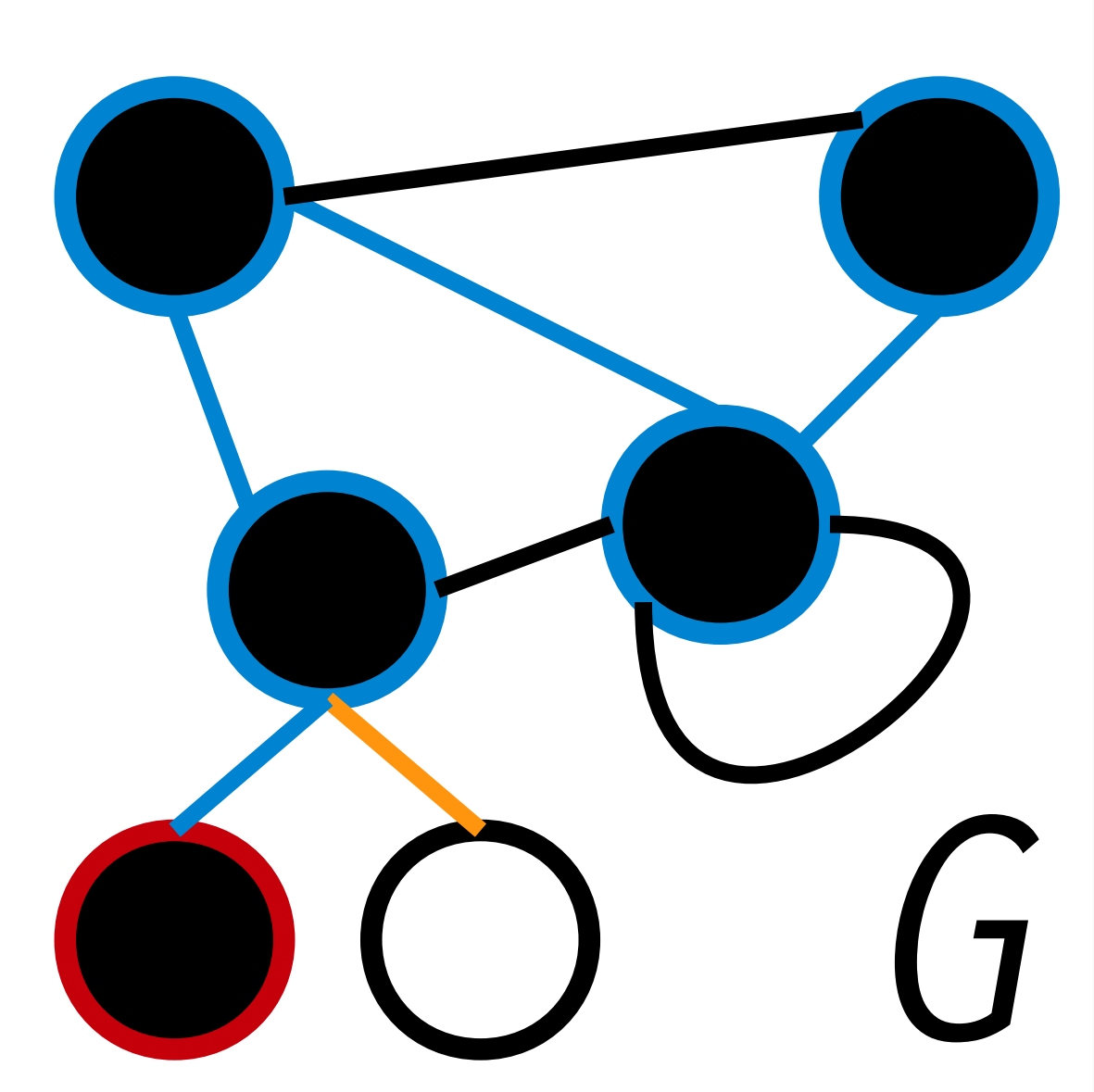
Example



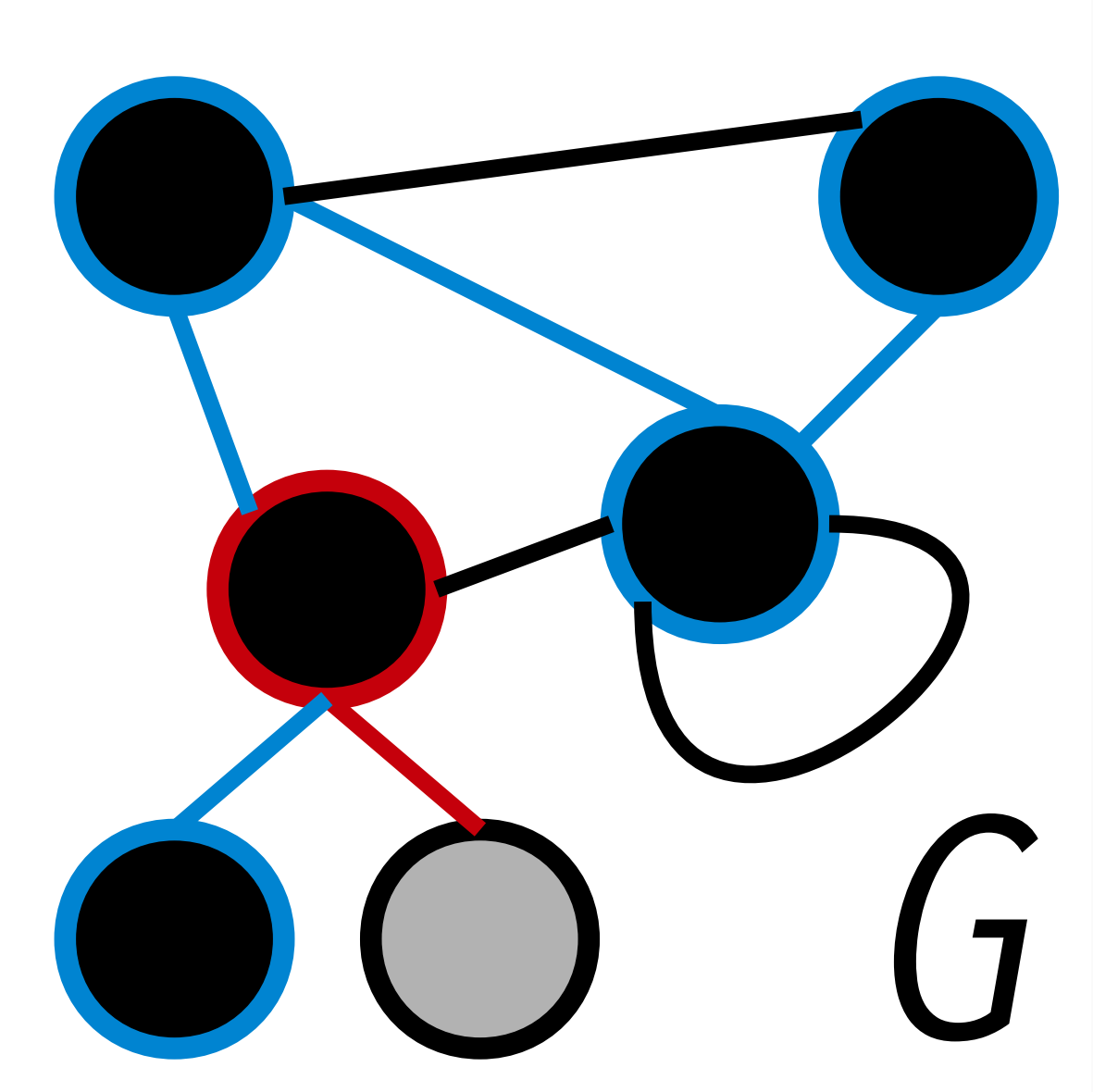
Example



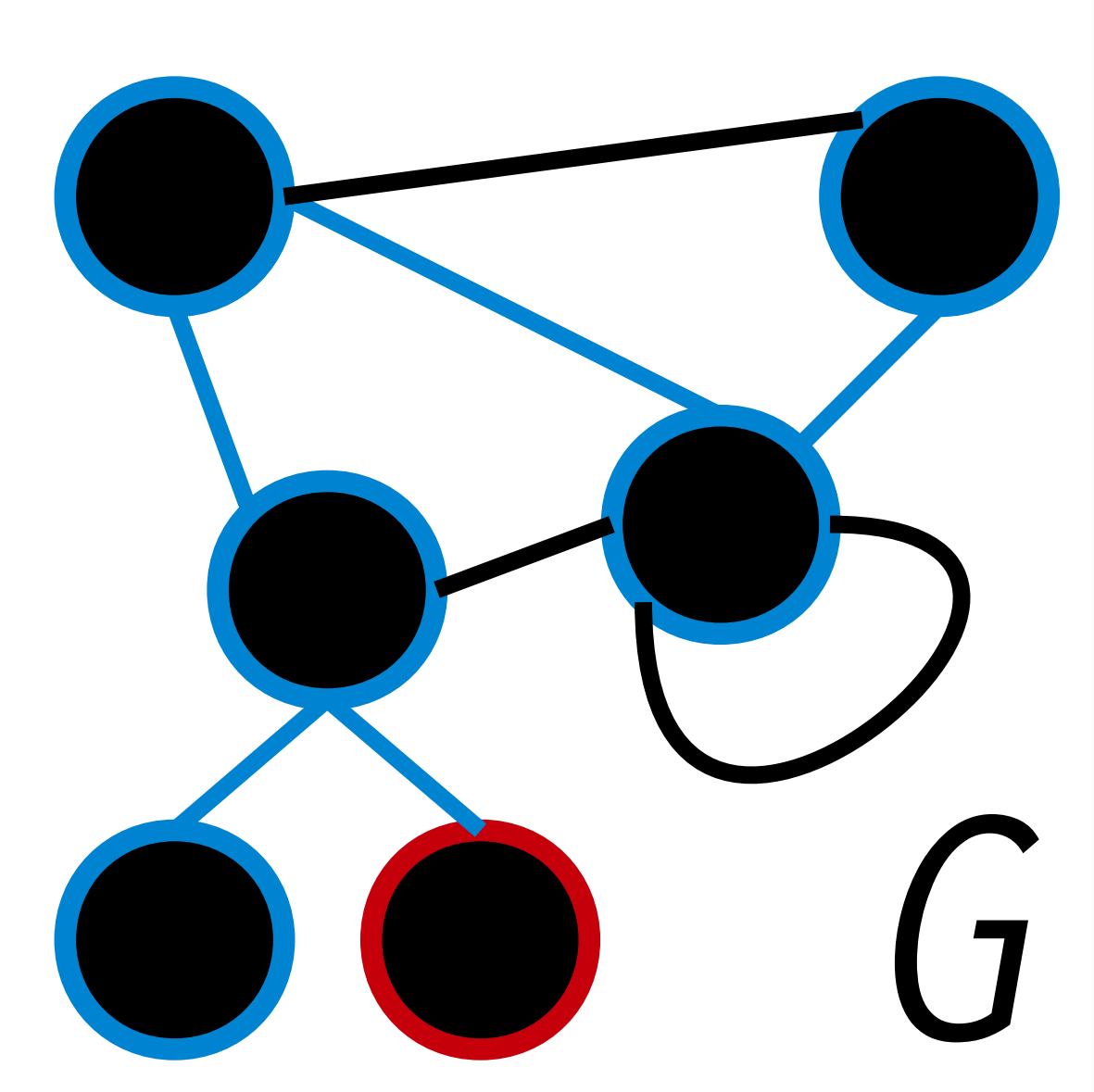
Example



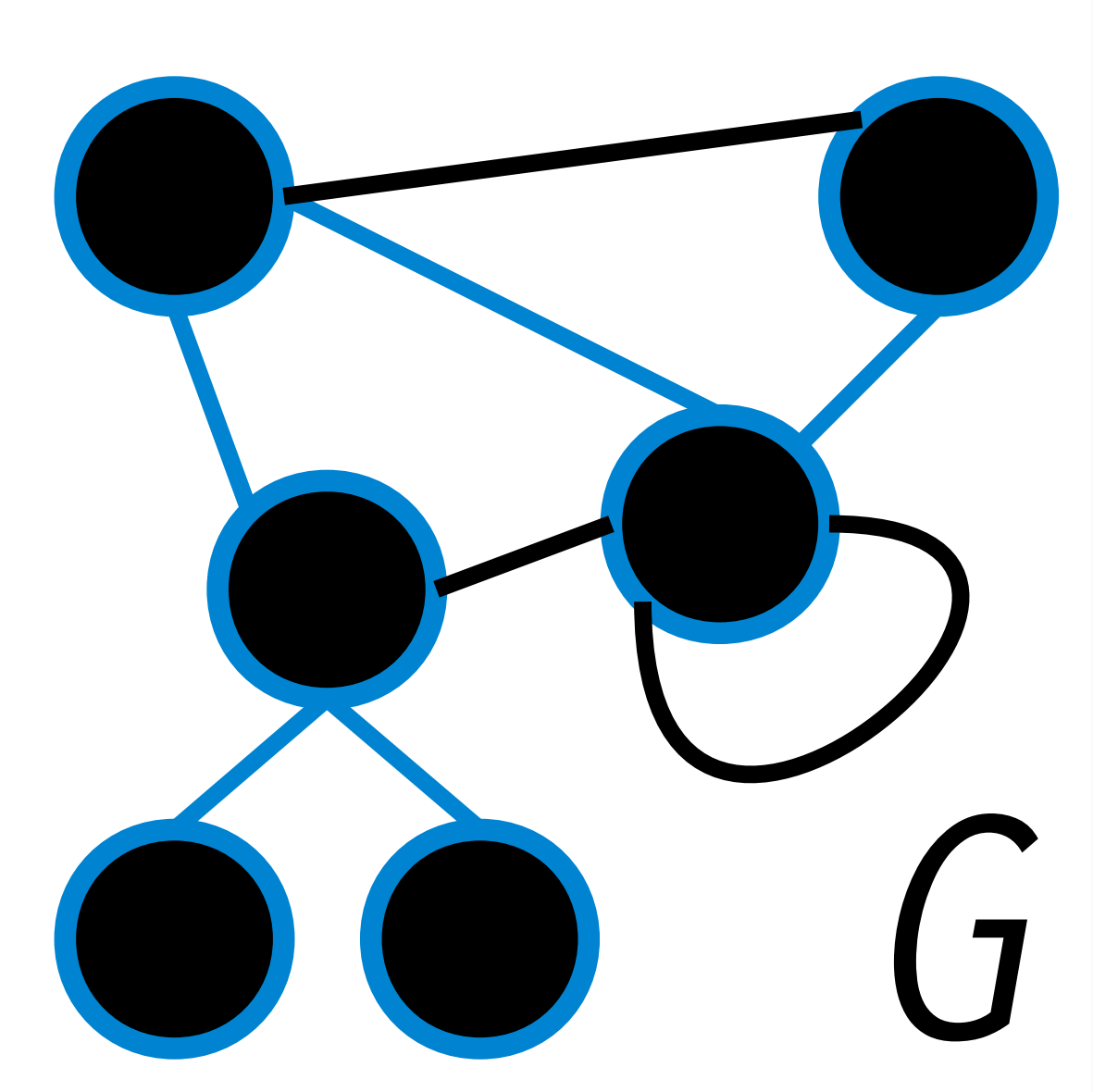
Example



Example



Example



Analysis

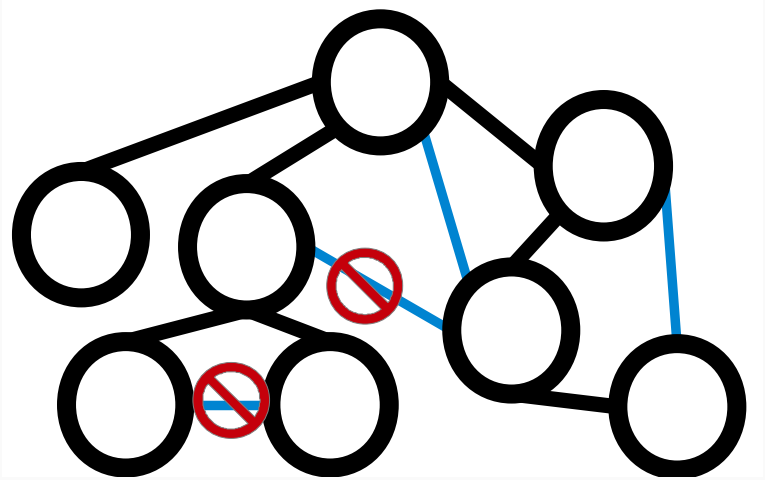
Property 1: DFS $\rightarrow$ spanning tree $T = \langle V, E' \rangle$ (pseudocode).

Time Complexity: $\mathcal{O}(|E| + |V|)$ (Why?)

Property 1: DFS $\rightarrow$ Nodes **ordering**

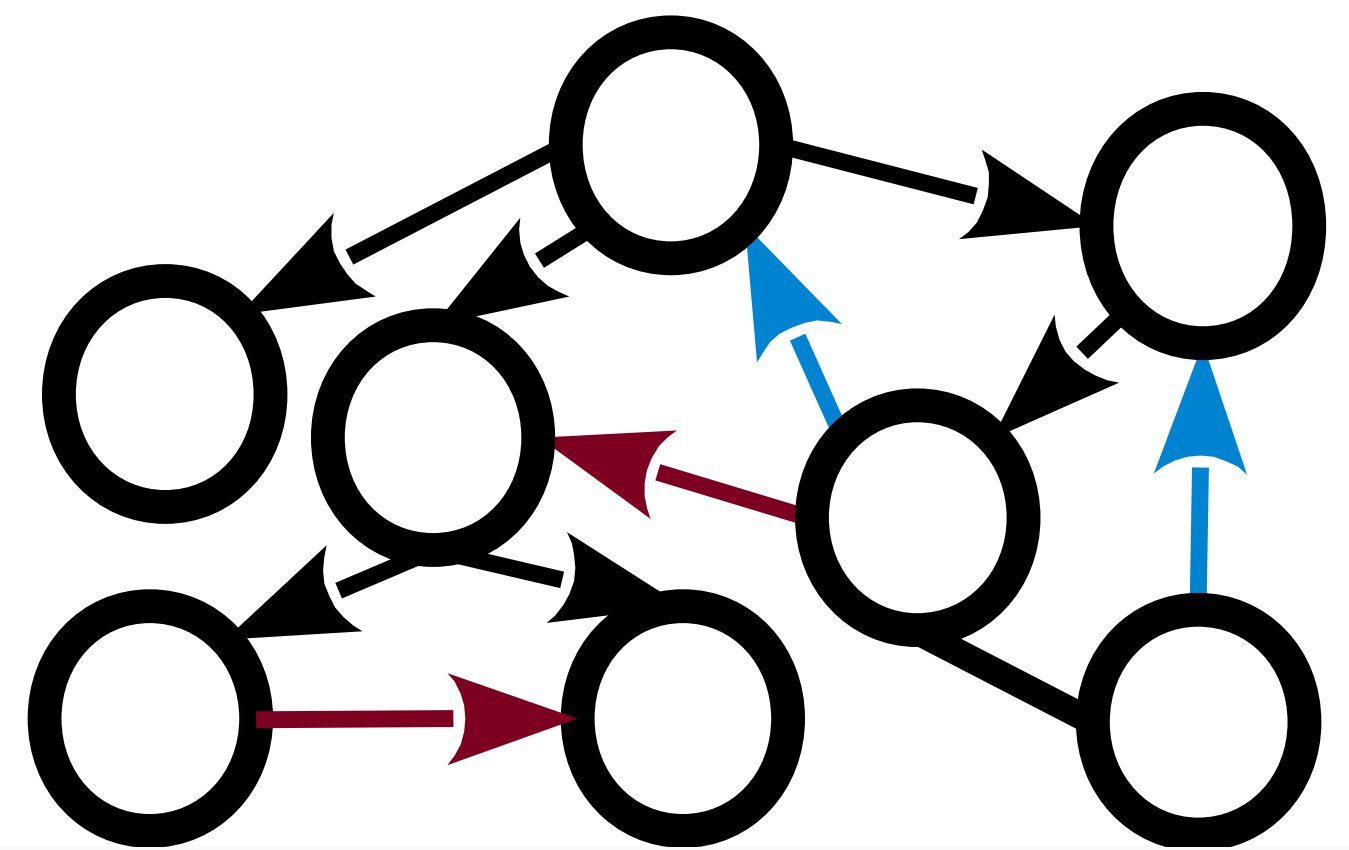
Property 2: DFS $\rightarrow$ **biconnectivity**, **bridges**, **cut nodes**.

Property 3: (Undirected graphs) edge e s.t. $e \in E$ and $e \notin E'$; e links a node with an ancestor in T; e$\neq$cross-edge (Proof?).



Backward-edges, Forward-edges, Cross-edges

- Forward-edges : black
- Backward-edges : blue
- Cross-edges : red

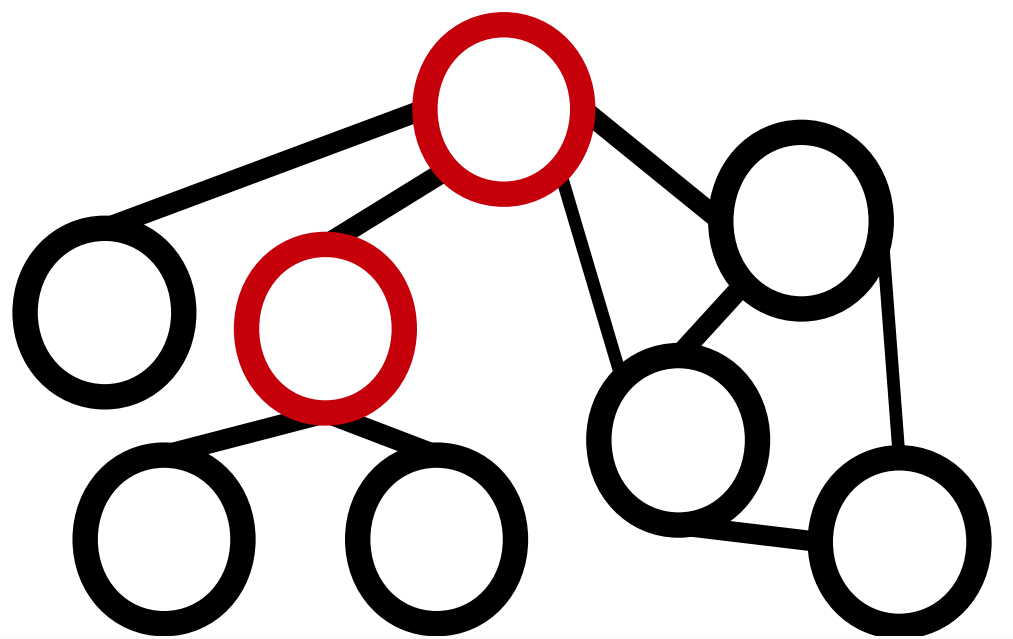


Biconnectivity using DFS

Biconnectivity

Definition 1: Graph $G = \langle V, E \rangle$ is **biconnected** if it has **no cut/articulation node**, i.e., 2 nodes or more should be removed to disconnect G.

Definition 2: A **cut/articulation node** is a node $v \in V$ s.t. removing v disconnects G.



Biconnectivity | Algorithm

Easy solution: Remove each node and test with BFS or DFS is G remains connected (Complexity?).

Analysis:

- **Root:** Cut node if its nb. of children ≥ 2
- **Leaves:** Never
- **Internal node** u: cut node if $\exists$ subtree rooted at a child of u **without** back-edges to an ancestor of u.

Articulation node detection

Methodology:

- Keep track of **back-edges going higher**; using **DFS Discovery times** (*discT*).
- $l(v)$: **Highest** node **reached** from **subtree rooted** at v

$l(v) = min\{discT[w] : u \in Descendent(v) \text{ and } \langle u, w \rangle \text{: back-edge}\}$

u is an **articulation point** if $\exists v \in Children(u)$ s.t.:

$$l(v) \geq discT[u]$$

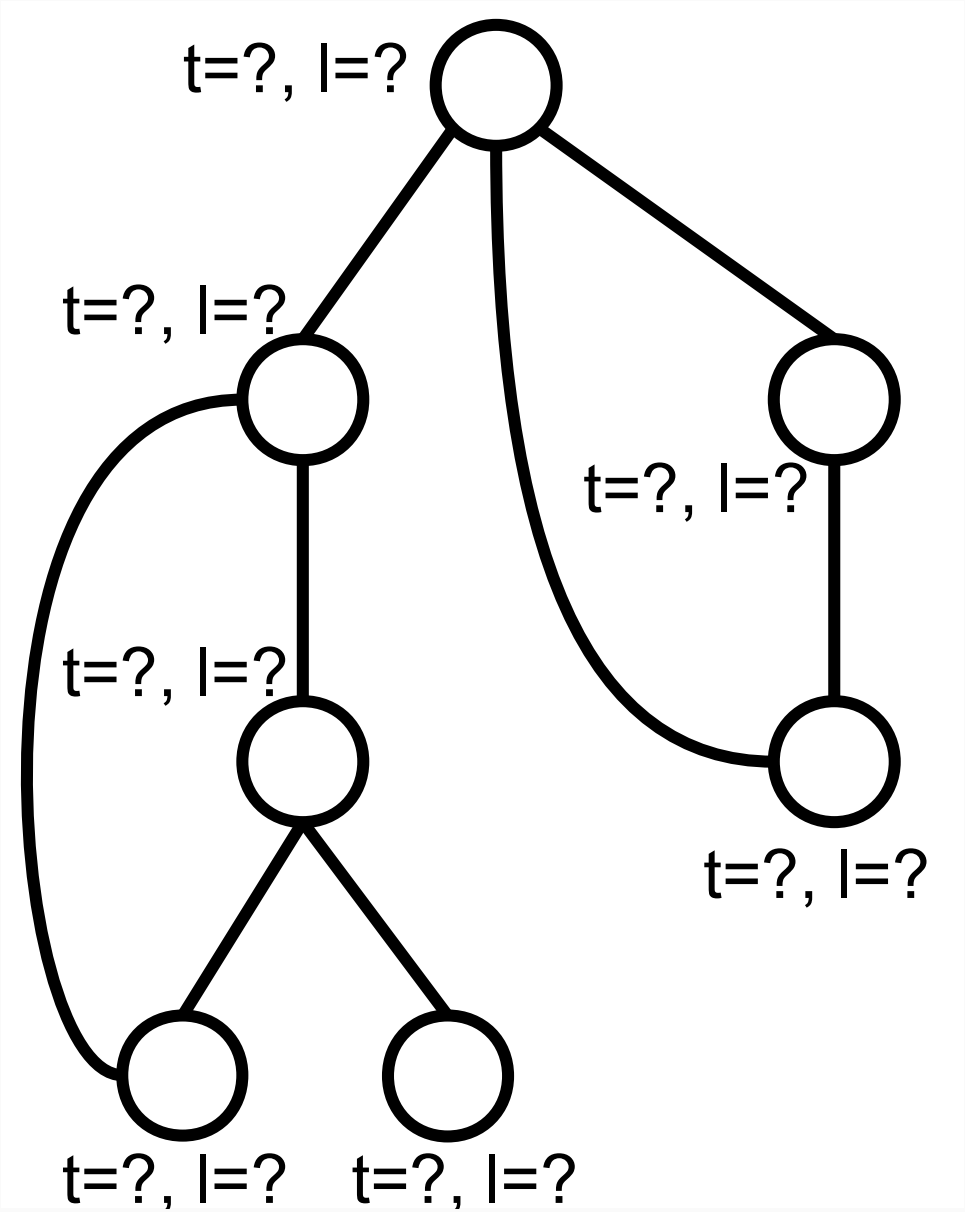
Pseudocode | Compute $l(u), \quad \forall u \in V$

```
function DFS_L(G, u, parent, state, l, t, discT)
    state[u] ← discovered
    discT[u] ← time;    l[u] ← time
    t ← t + 1
    for all v ∈ Neighbors(u, G) do
        if state[v] = undiscovered then
            parent[v] ← u
            DFS_l(G, v, parent, state, t)
            // After DFS_l l(v) is stored in l[v]
            l[u] ← min(l[u], l[v])
        else
            if v ≠ parent[u] then
                // Then ⟨u, v⟩ is a back-edge
                l[u] ← min(l[u], discT[v])
            end if
        end if
    end for
end function
```

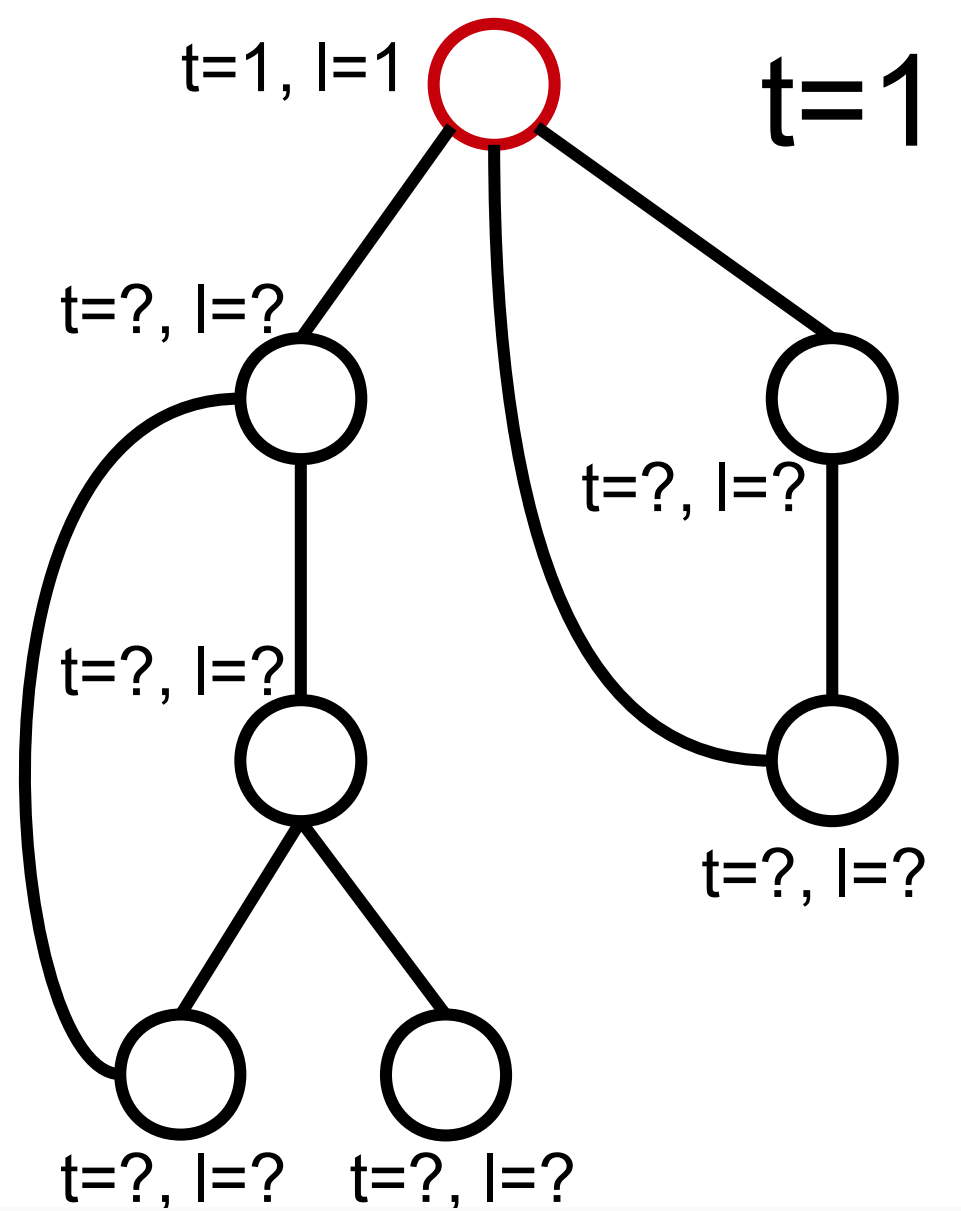
Pseudocode | Retrieve Articulation Points

```
function ARTICULATION_POINTS(G, parents, T, l, discT)
    for all v ∈ V do
        if parent[v] = root then // v is the root
            if |Neighbors(v, T)| > 1 then
                articulation[v] ← true
            end if
        else
            for all u ∈ V  s.t.  parents[u] = v do
                if l[u] ≥ discT[v] then
                    articulation[v] ← true
                end if
            end for
        end if
    end for
end function
```

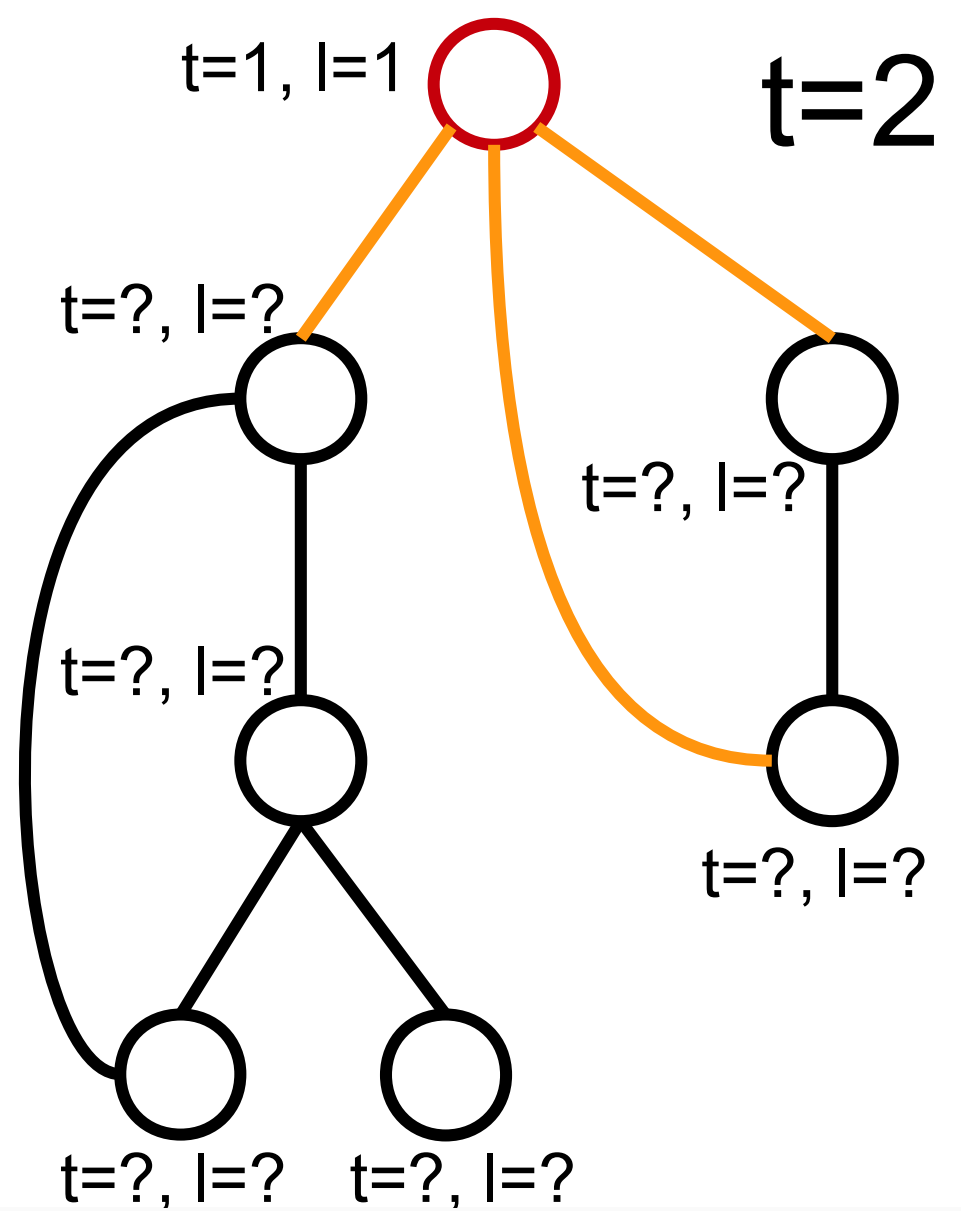
Example



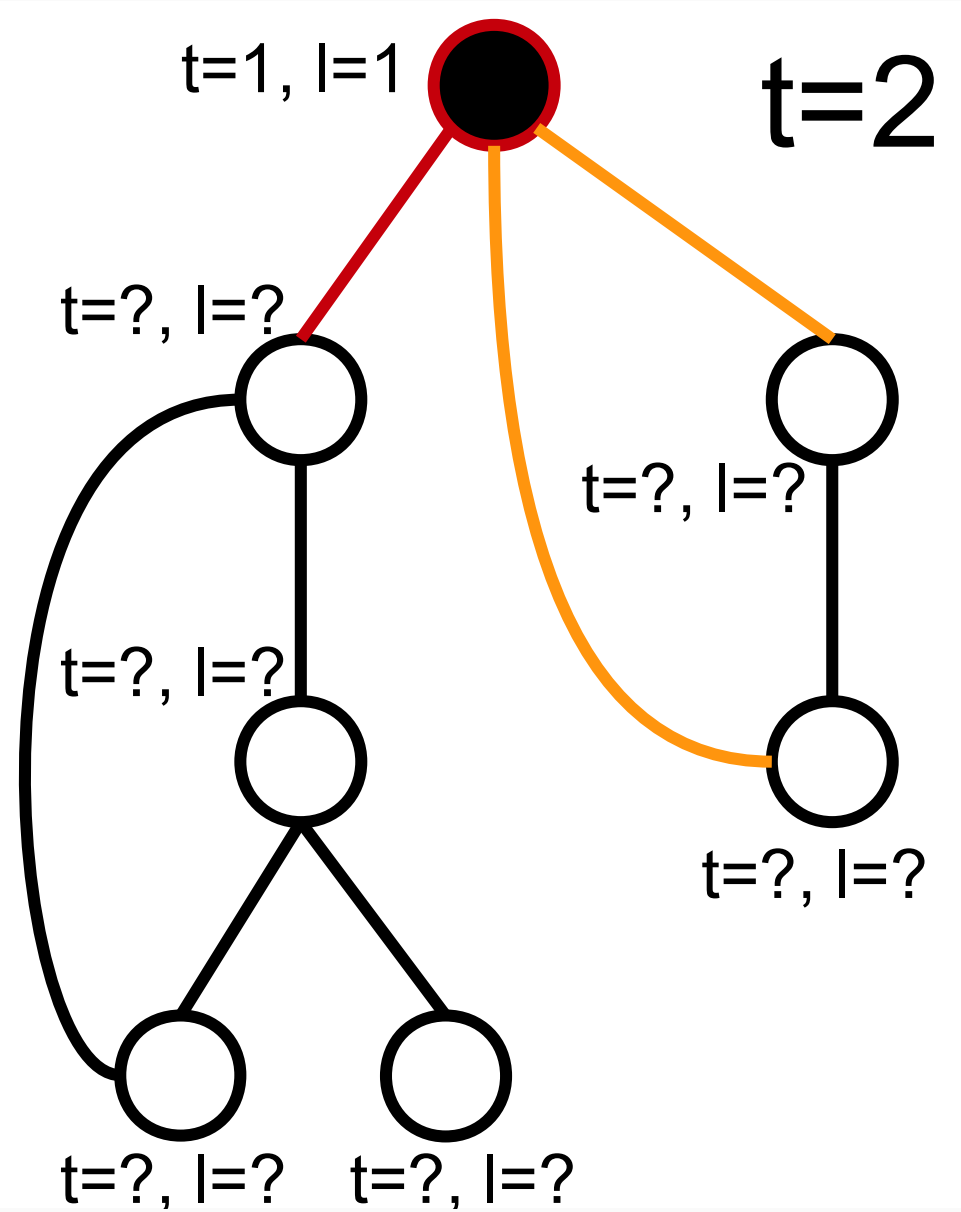
Example



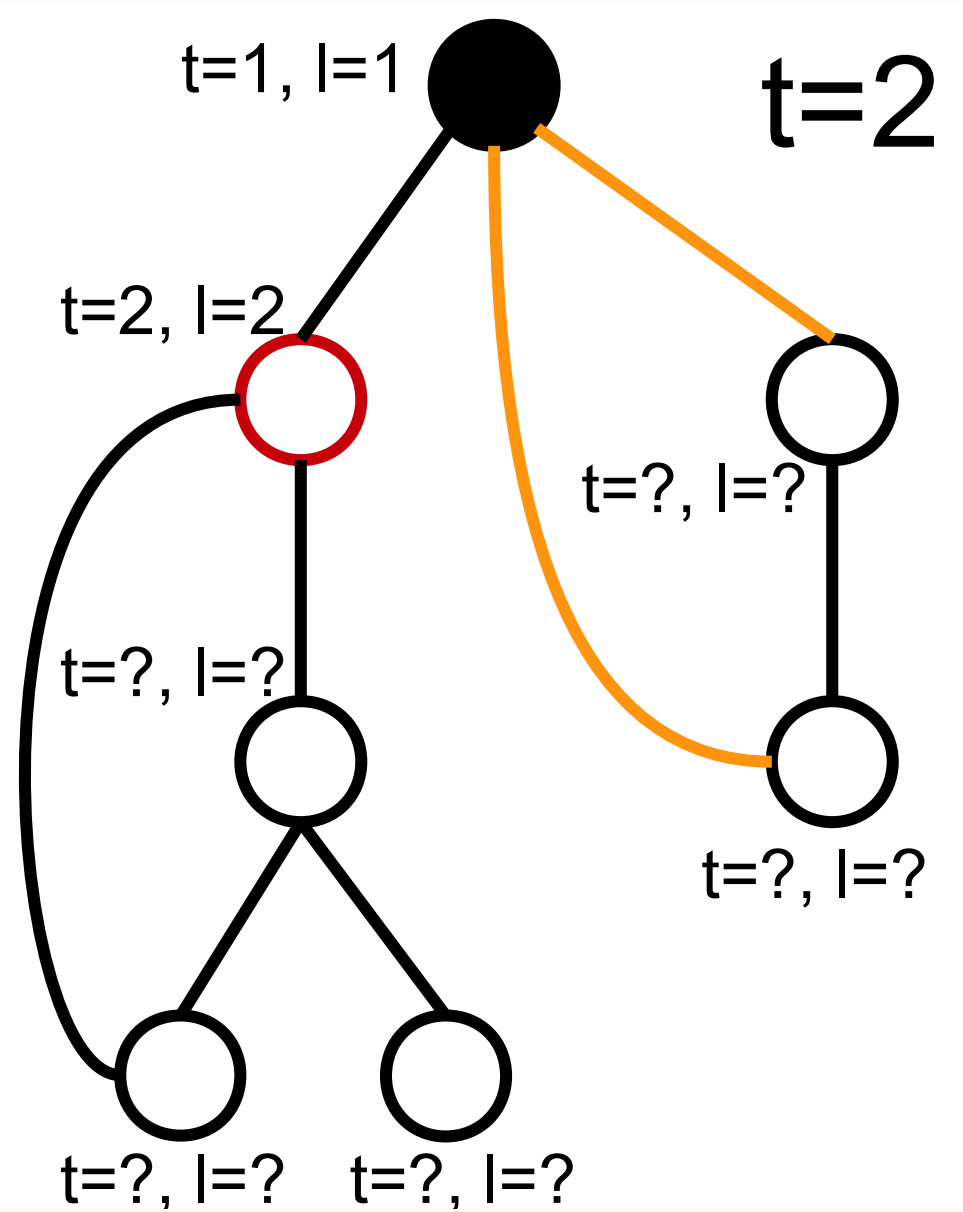
Example



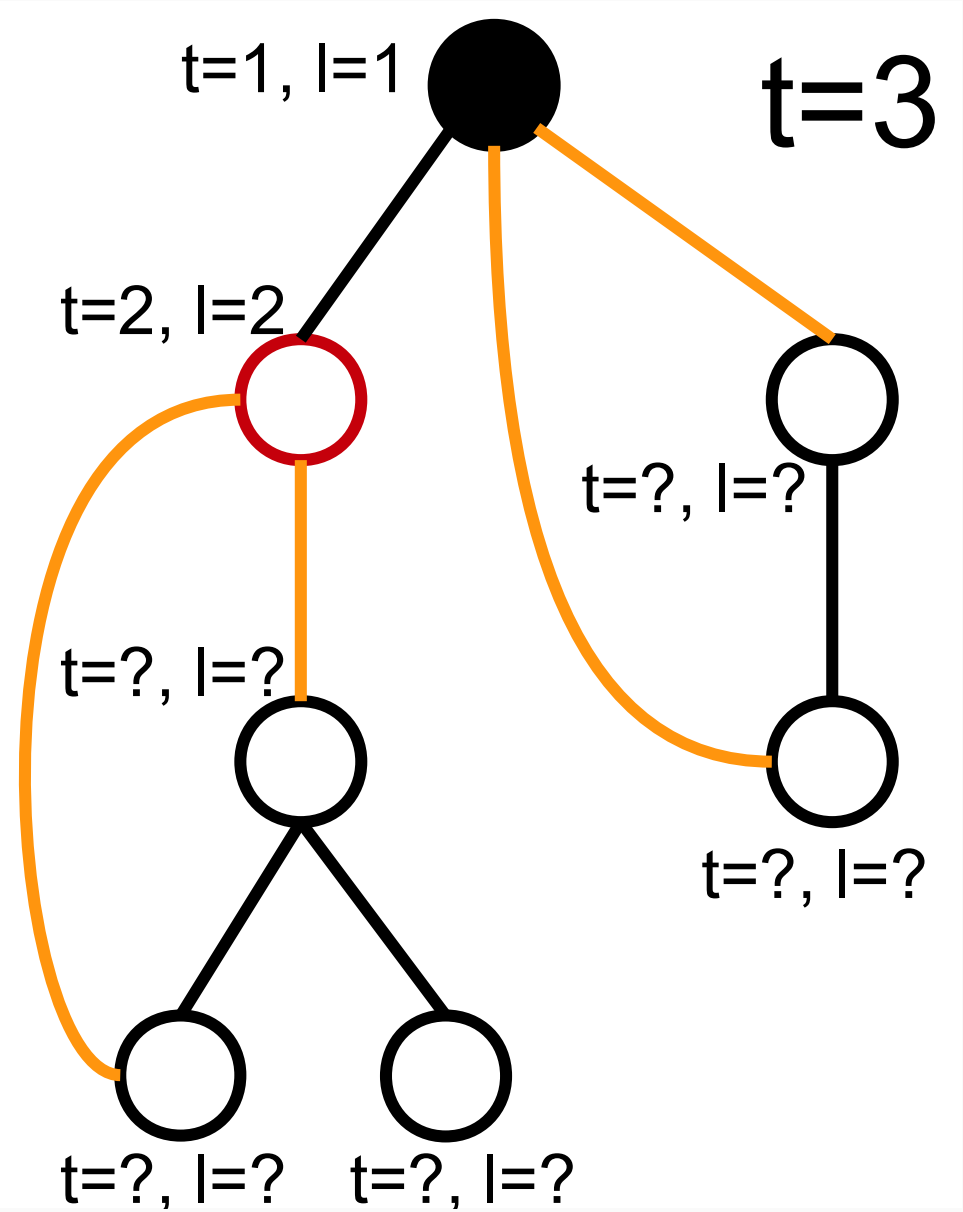
Example



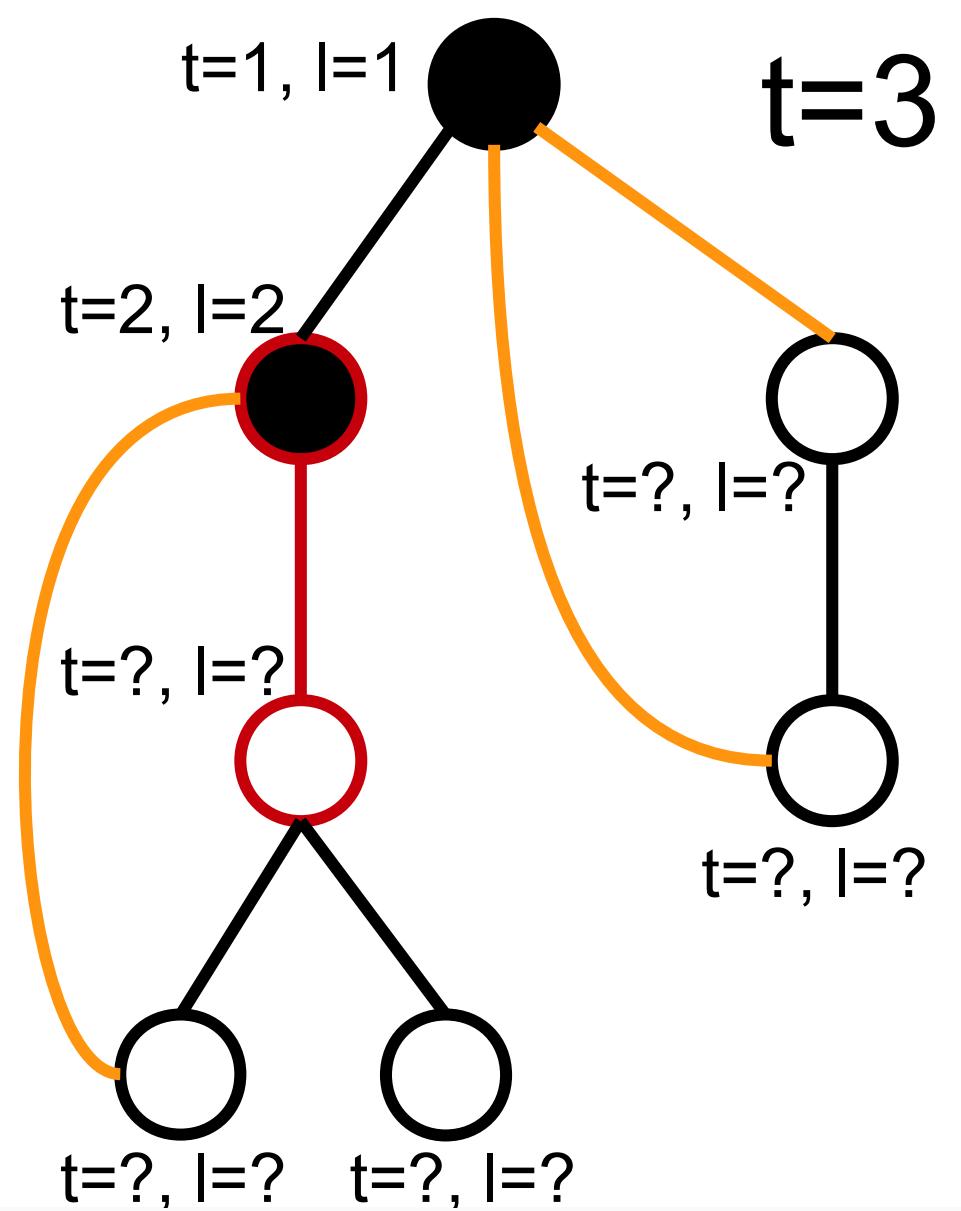
Example



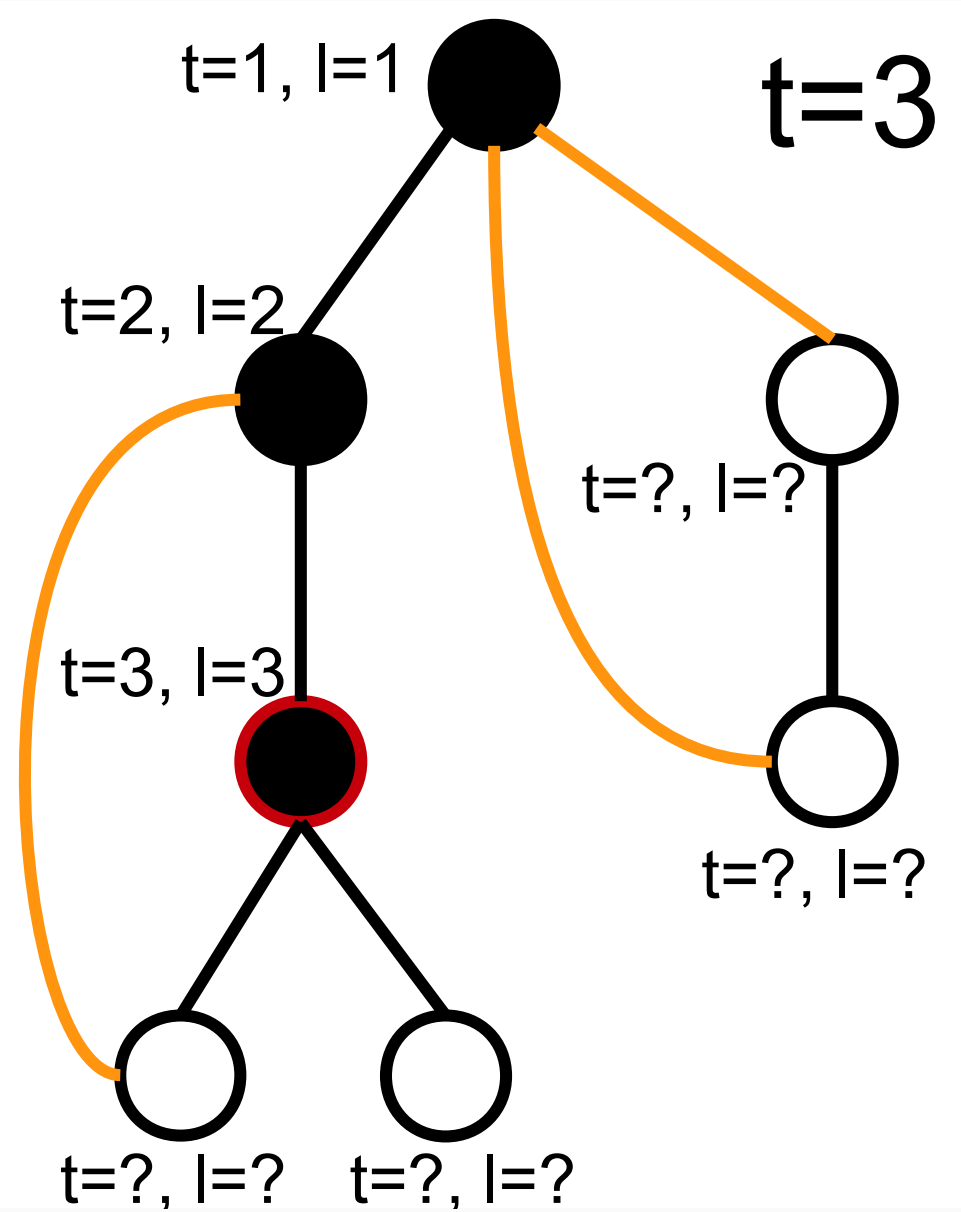
Example



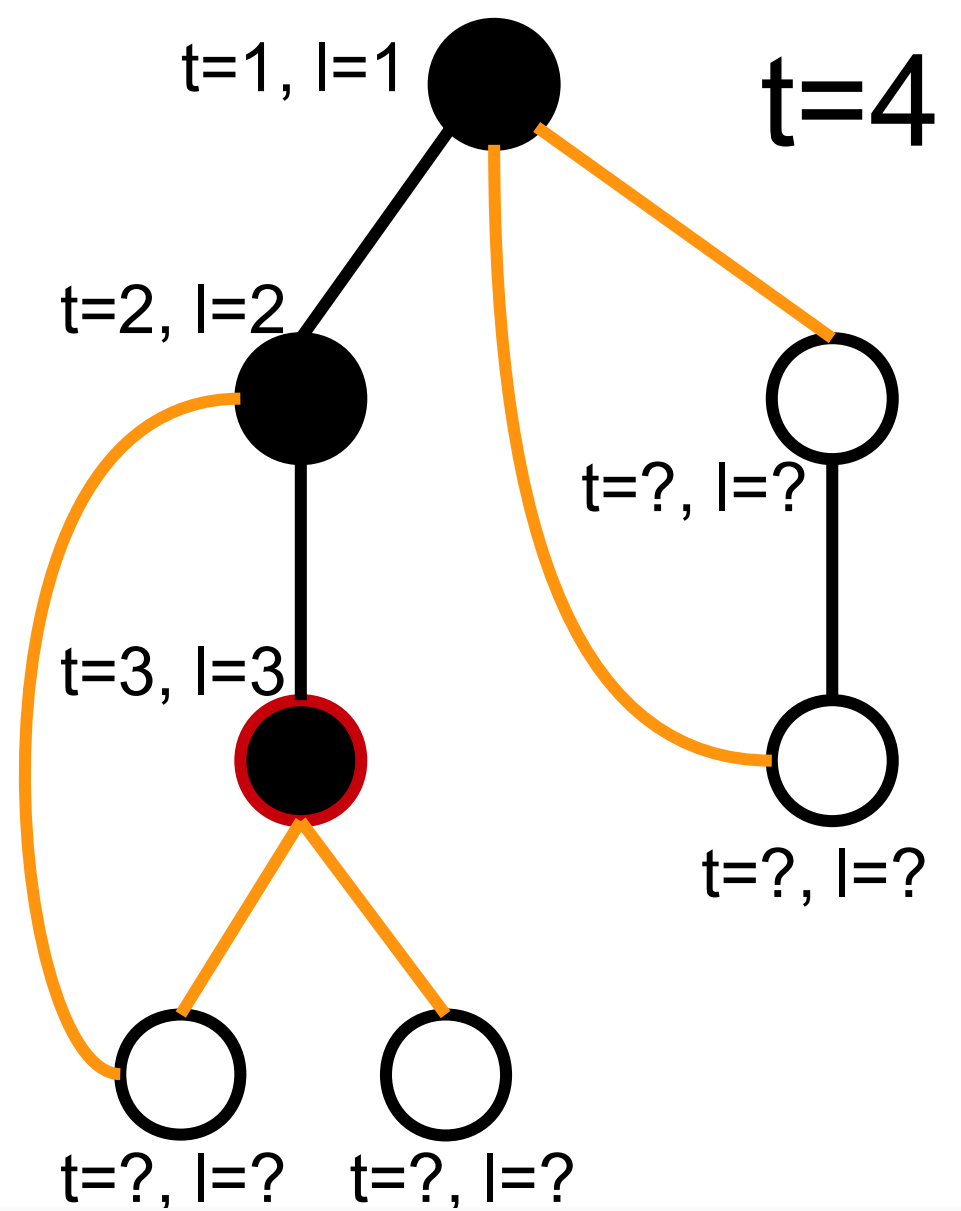
Example



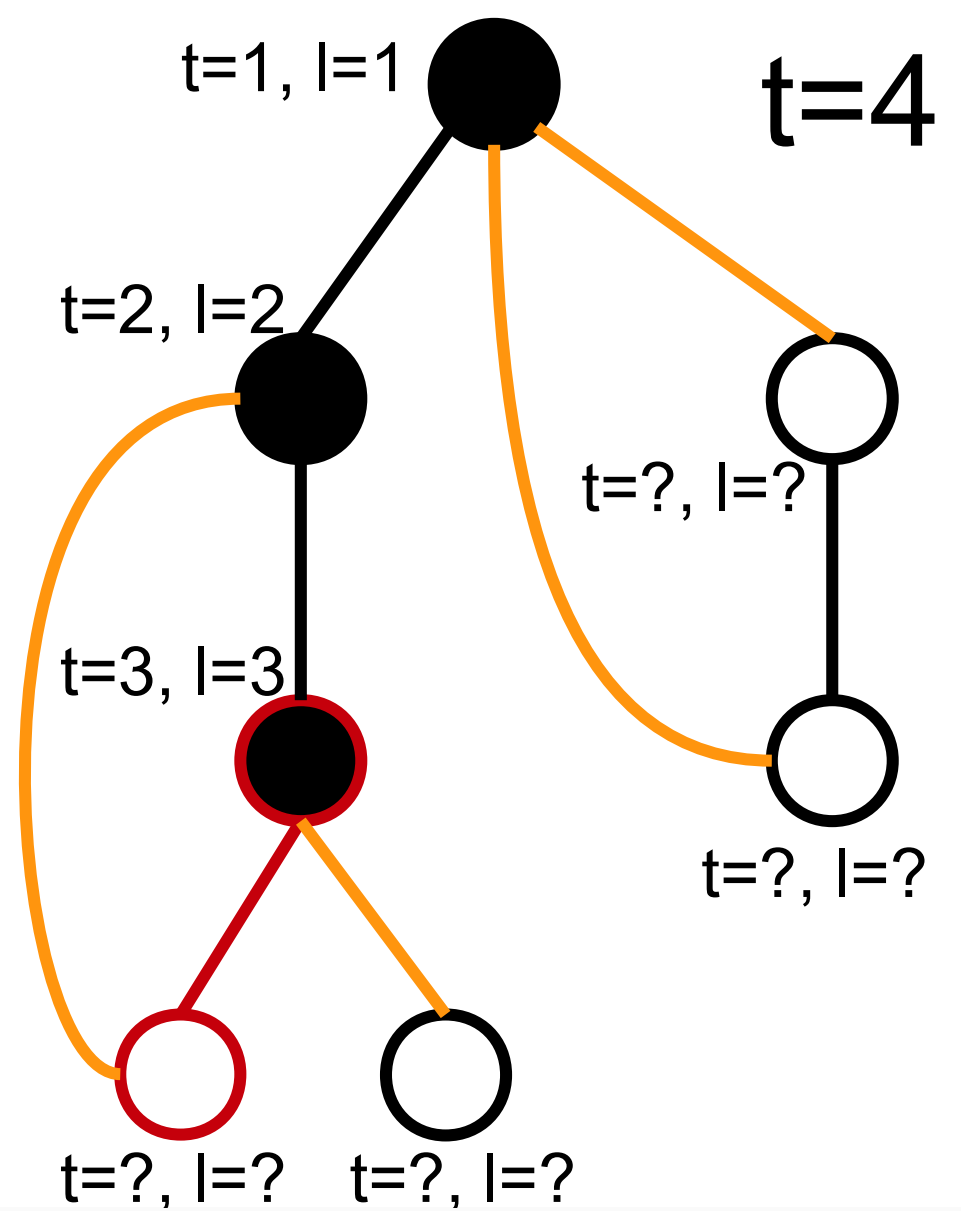
Example



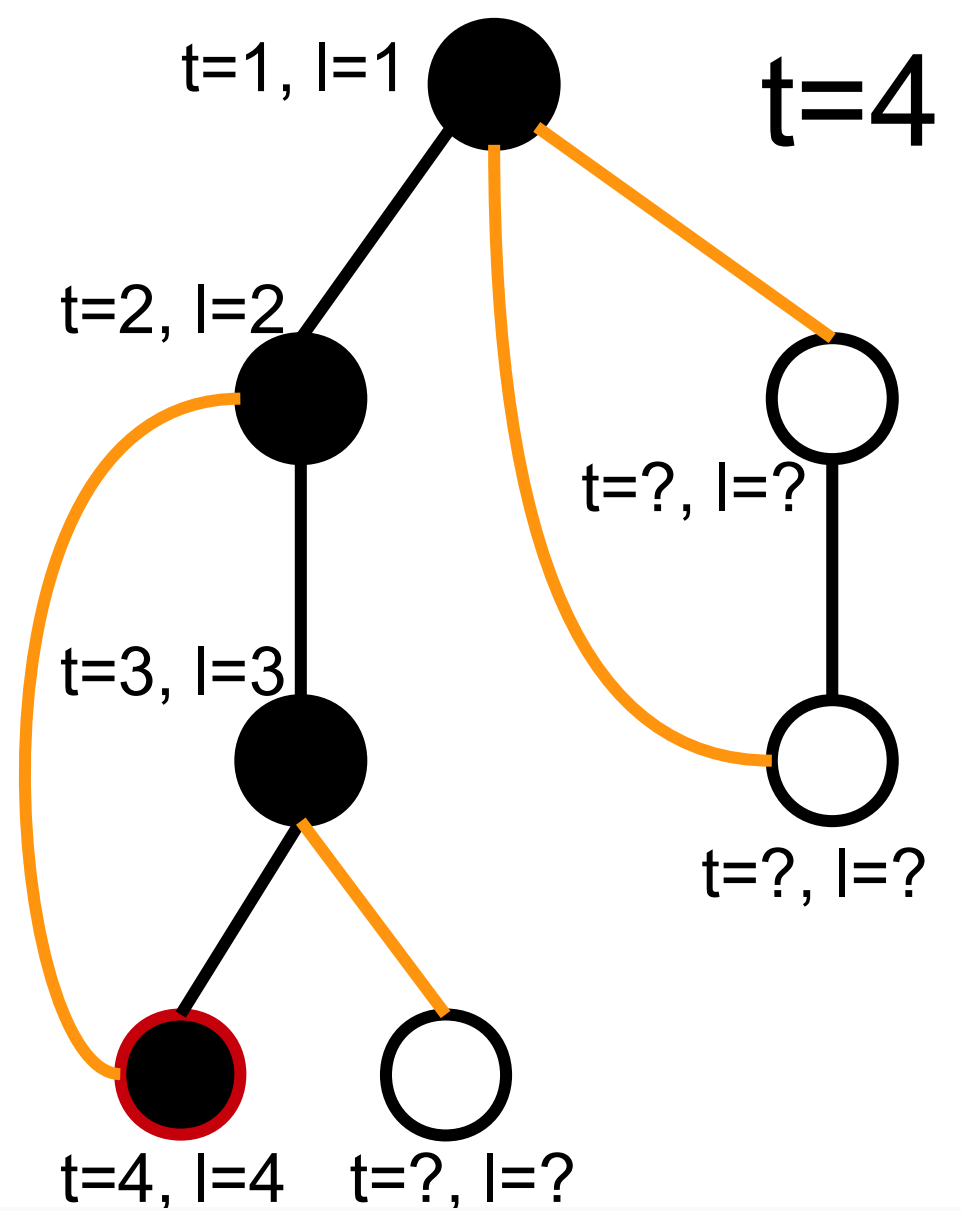
Example



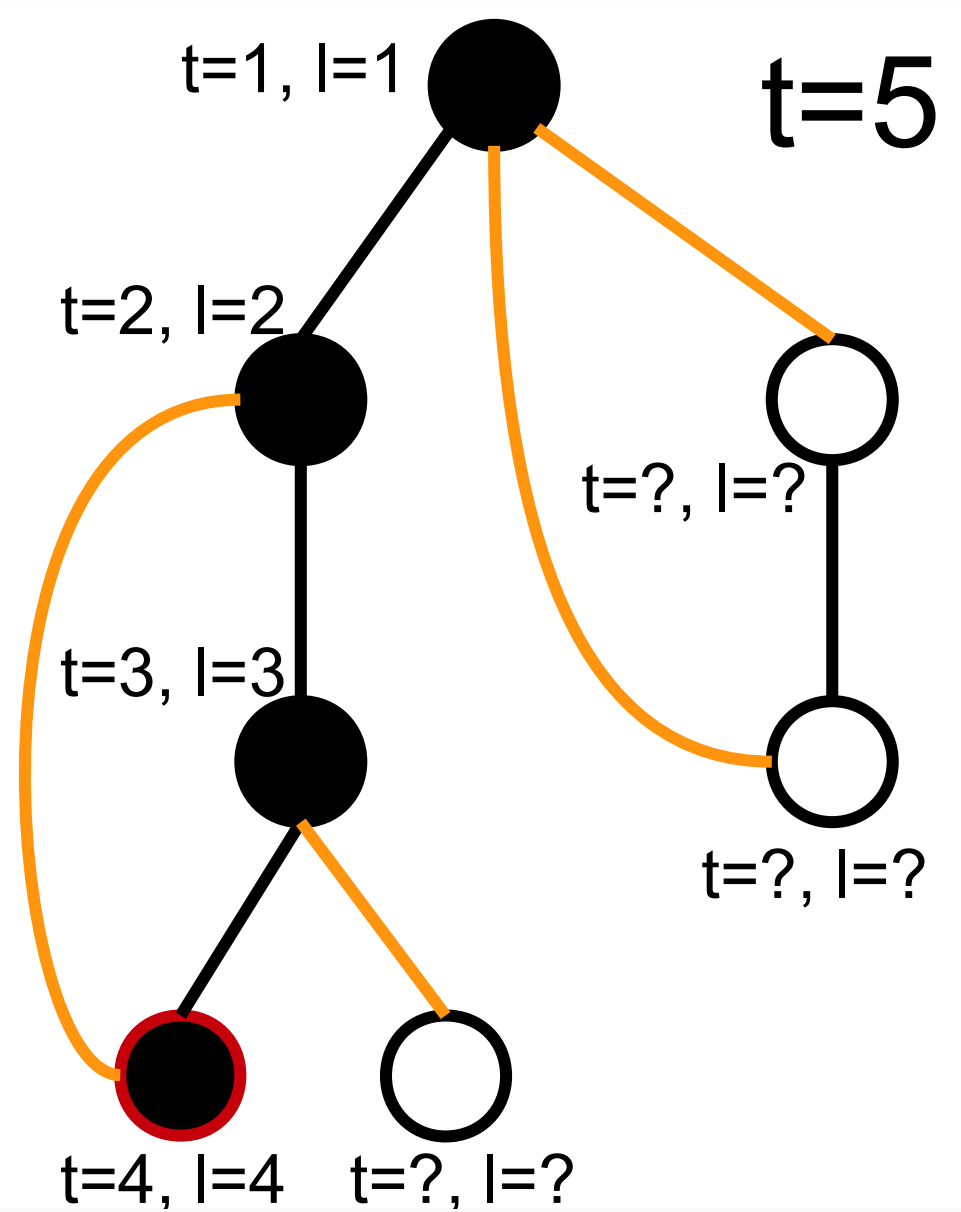
Example



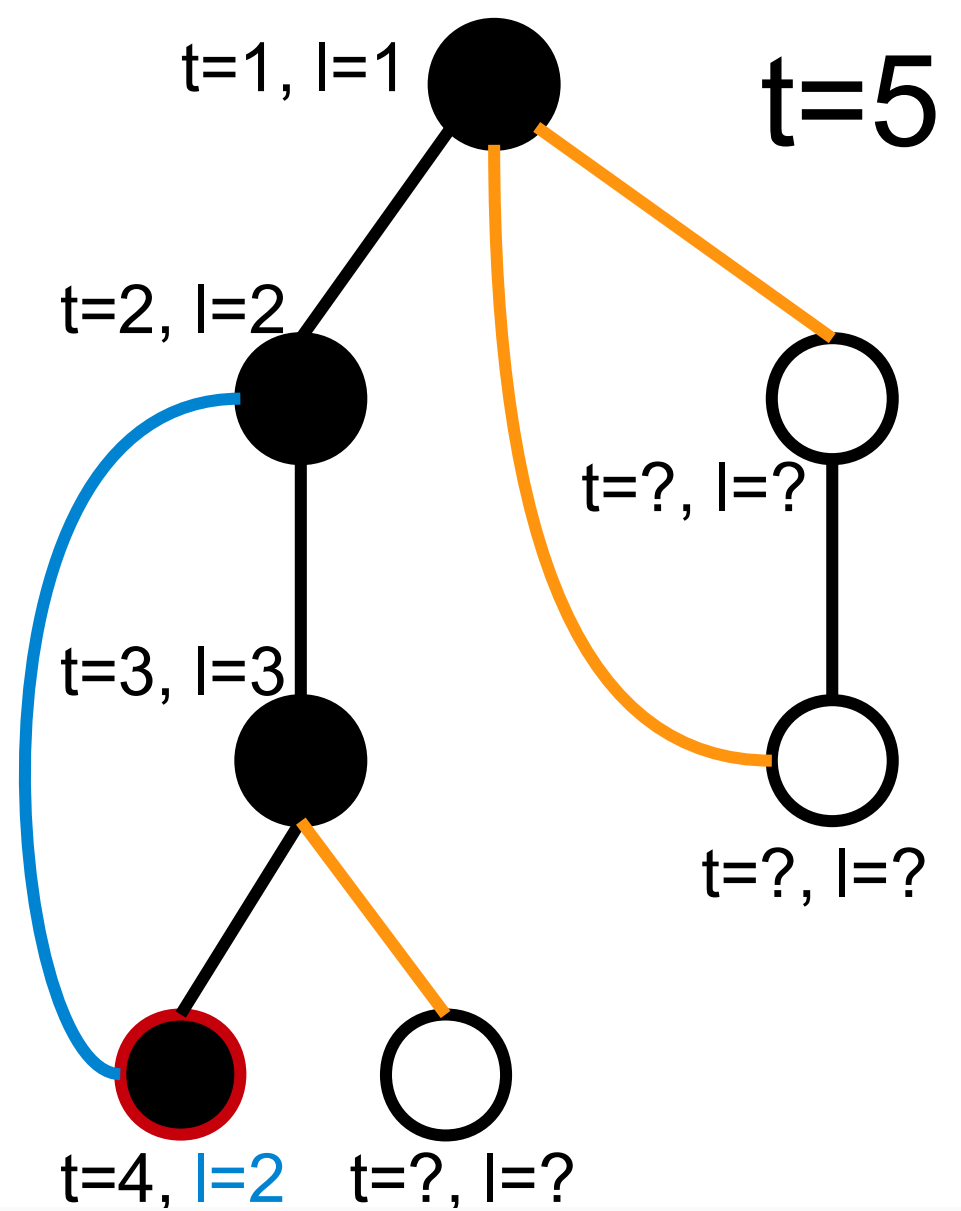
Example



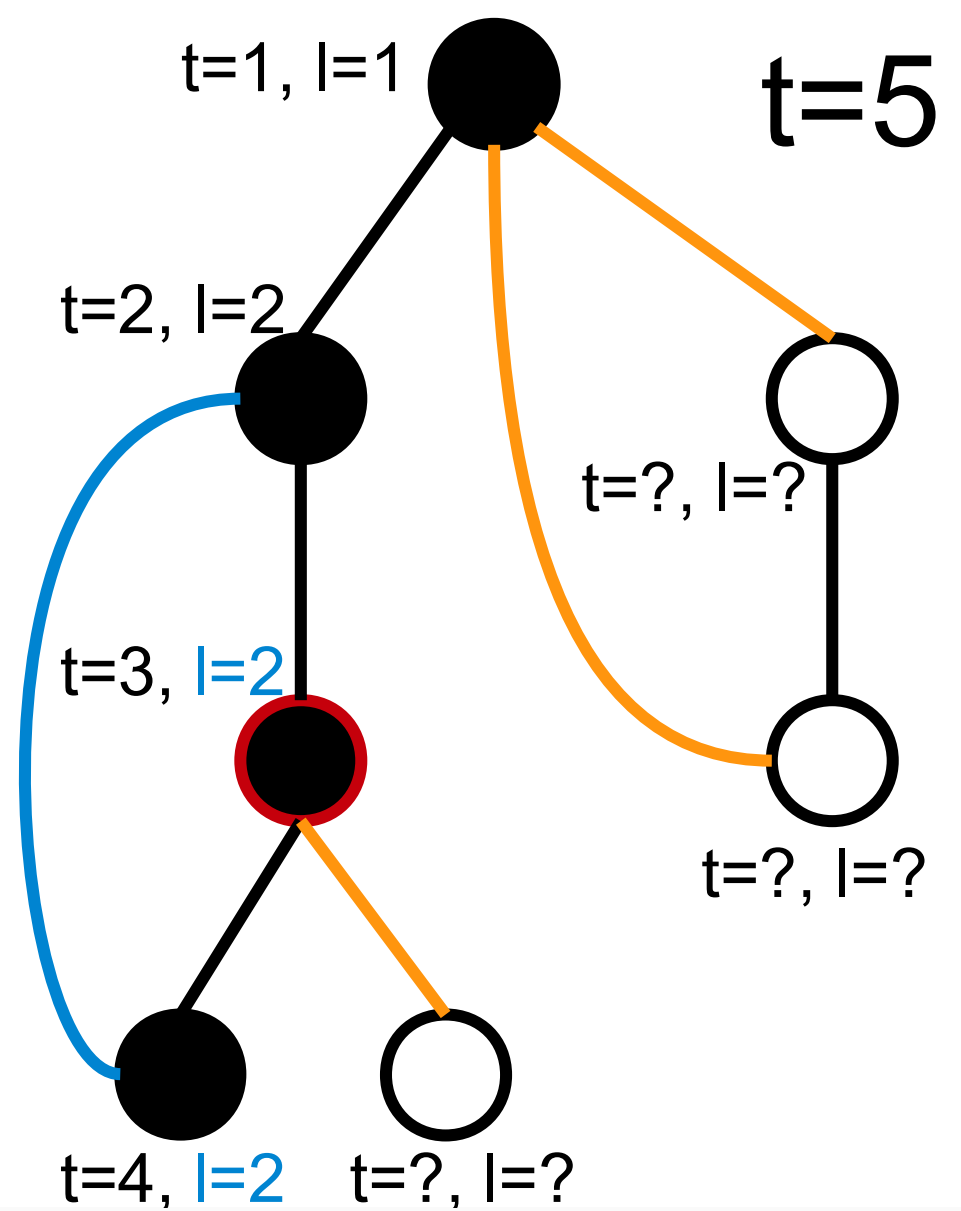
Example



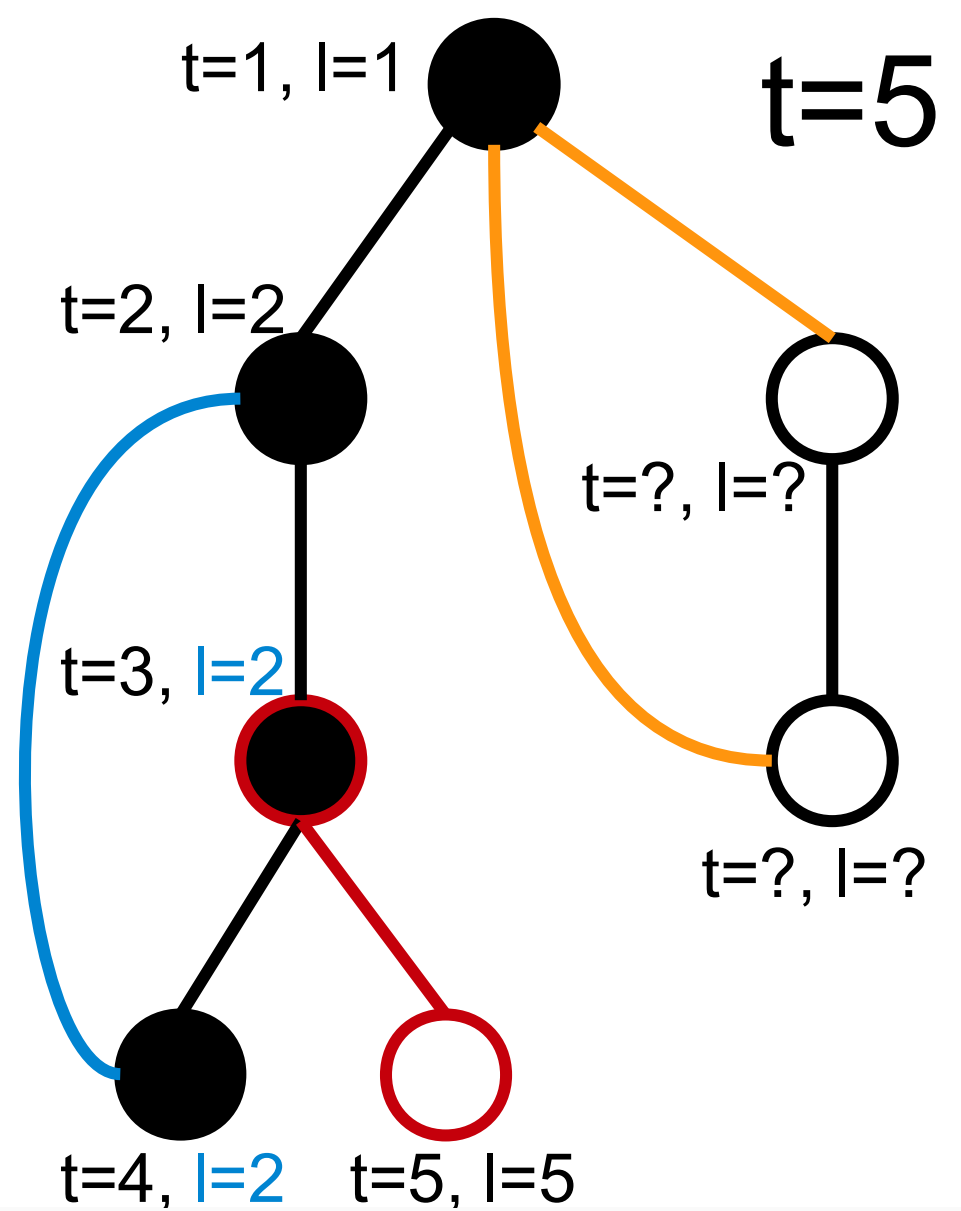
Example



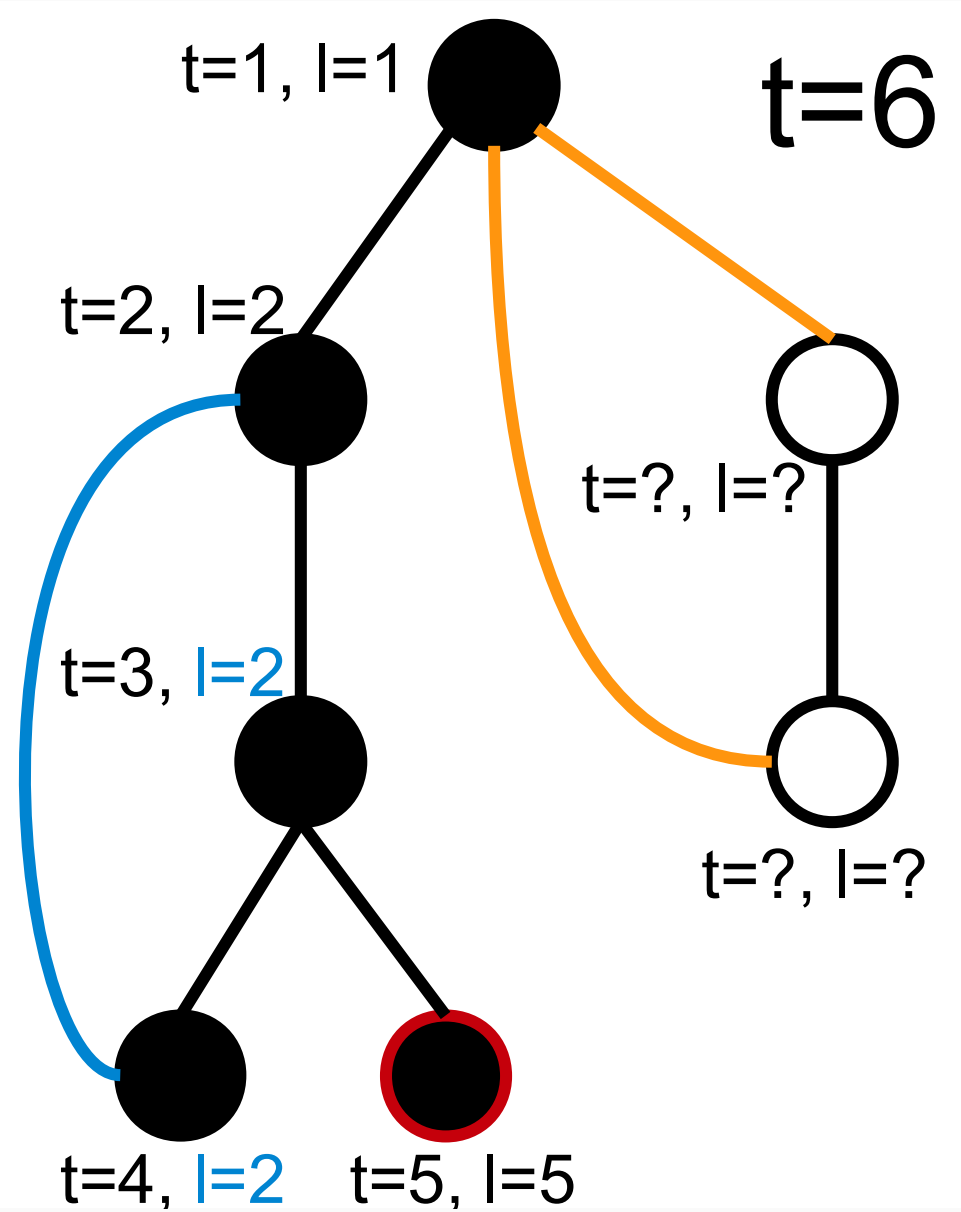
Example



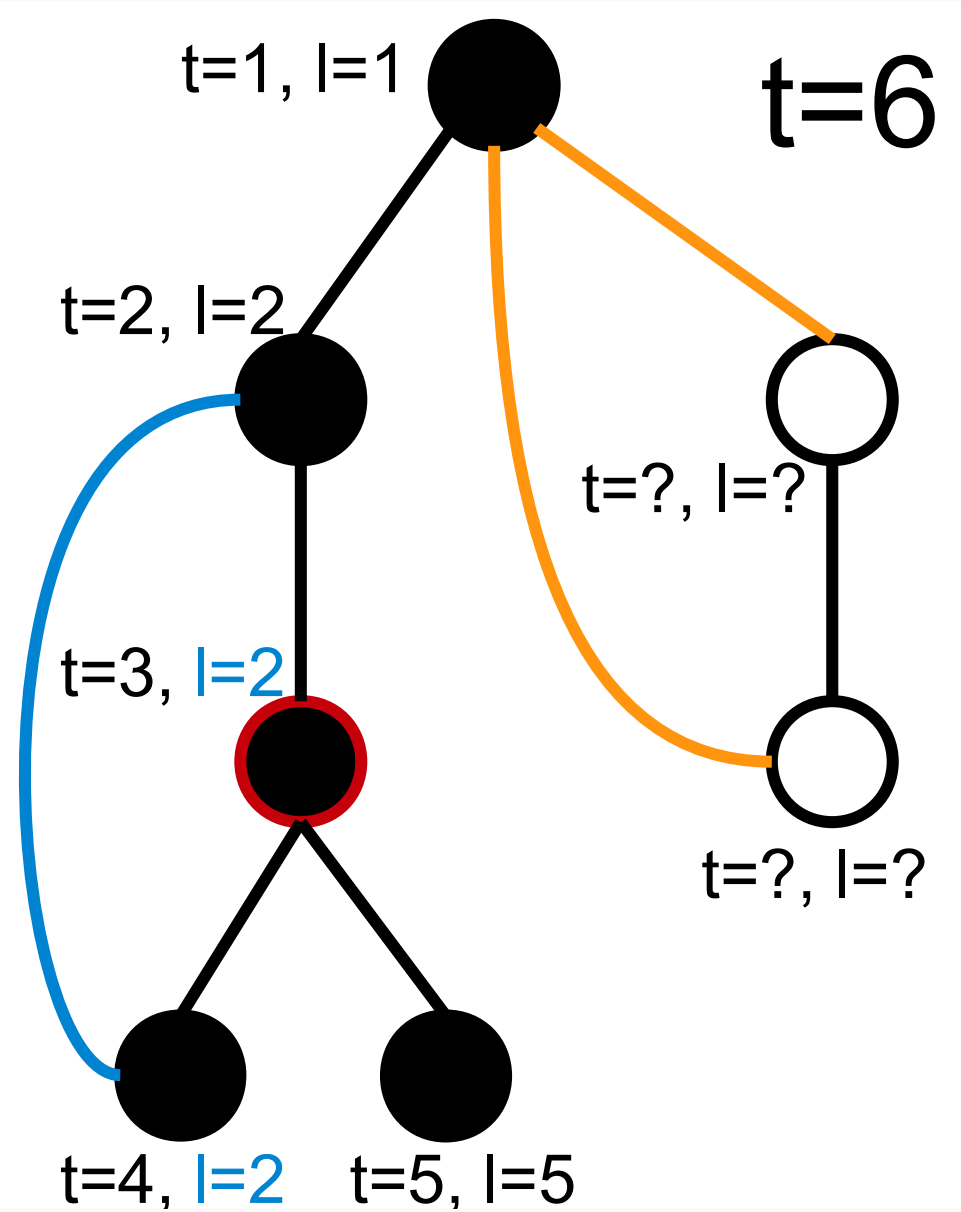
Example



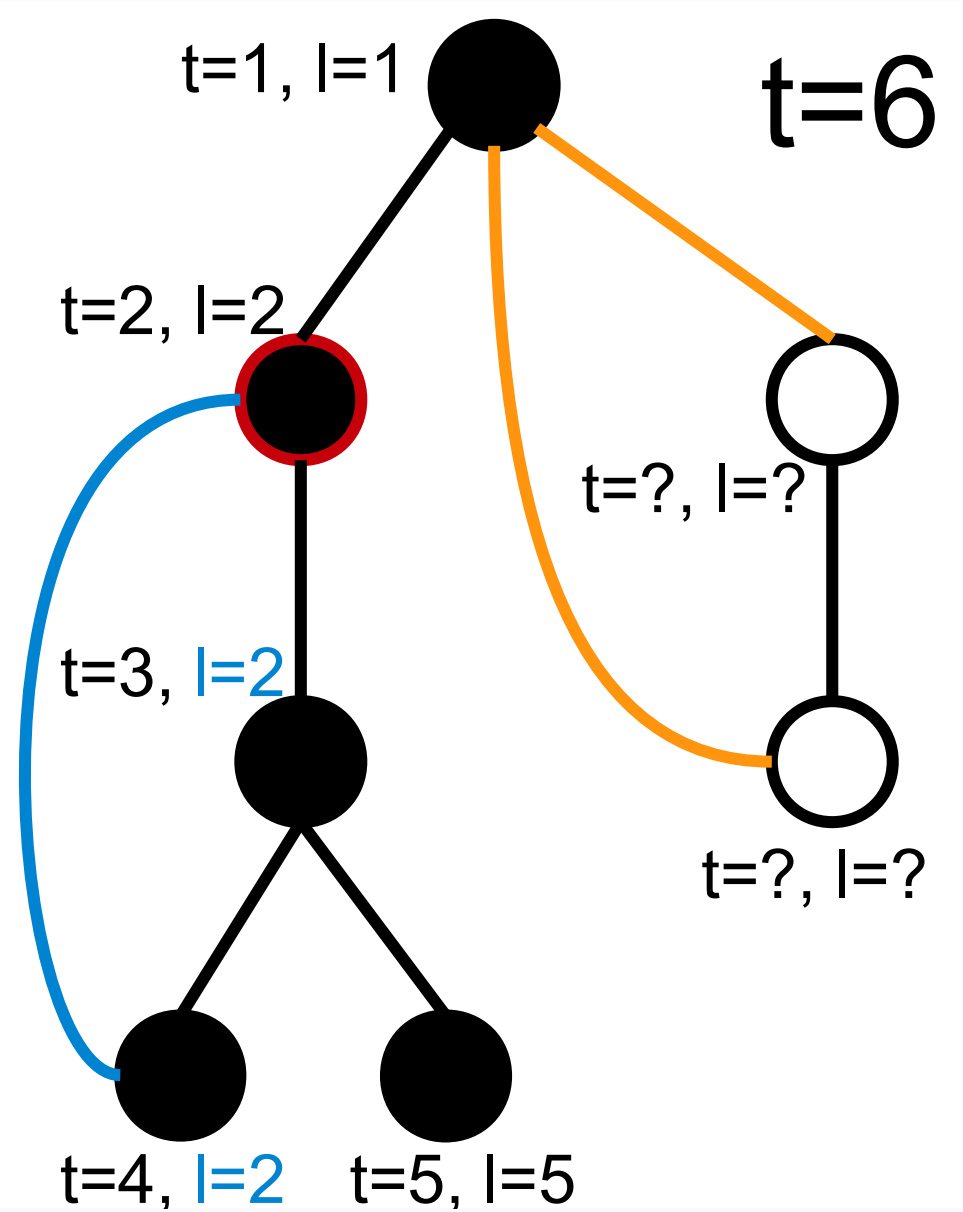
Example



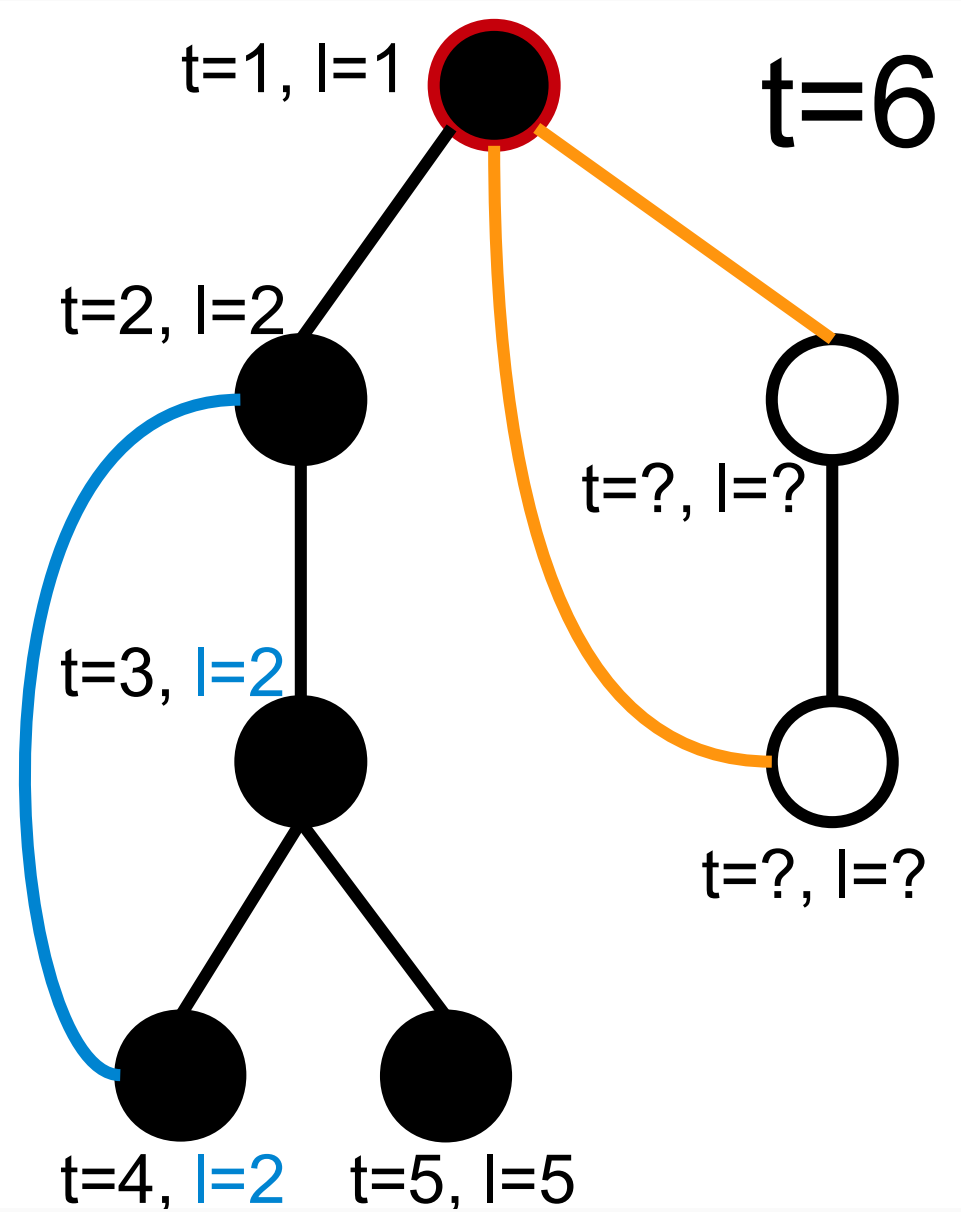
Example



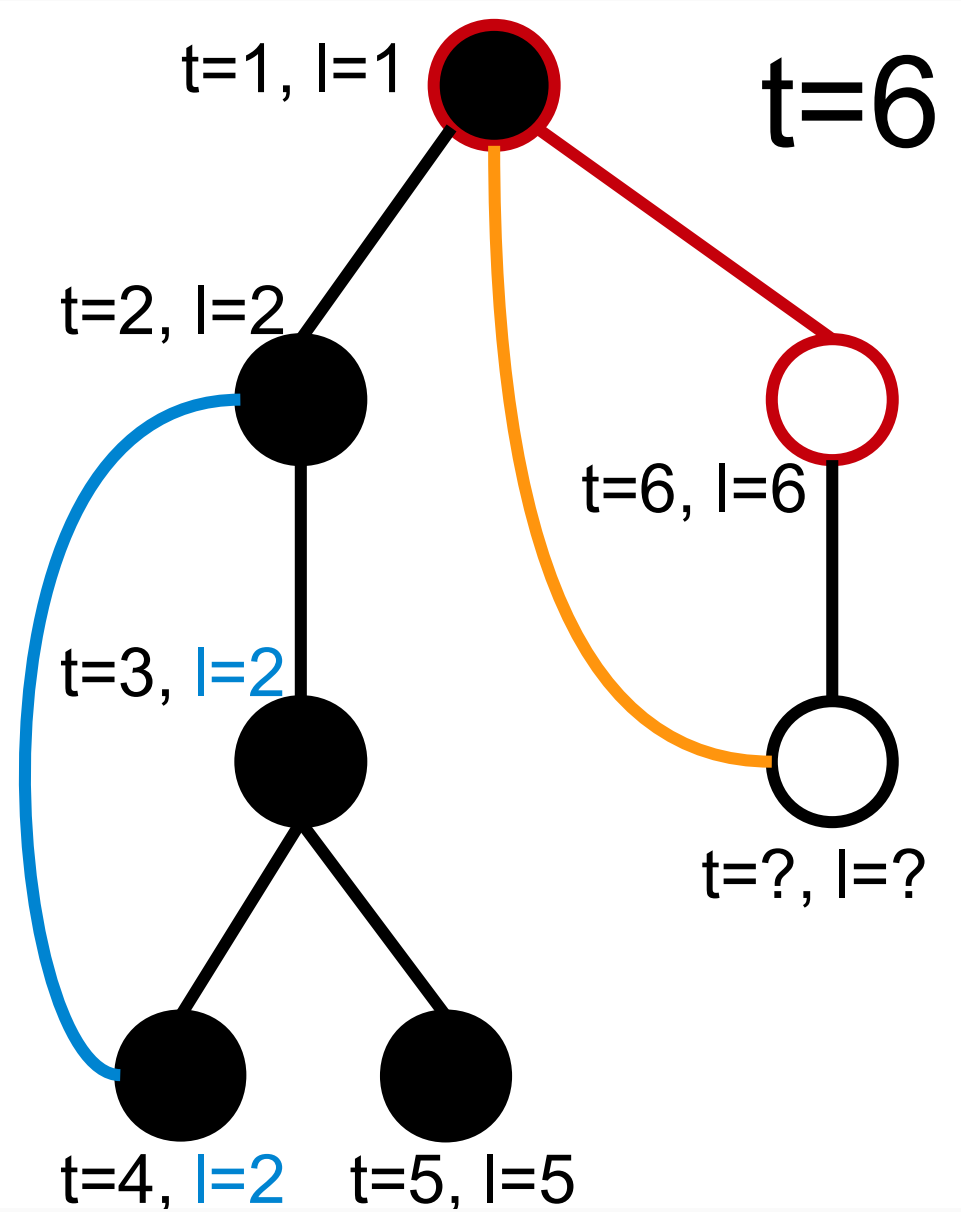
Example



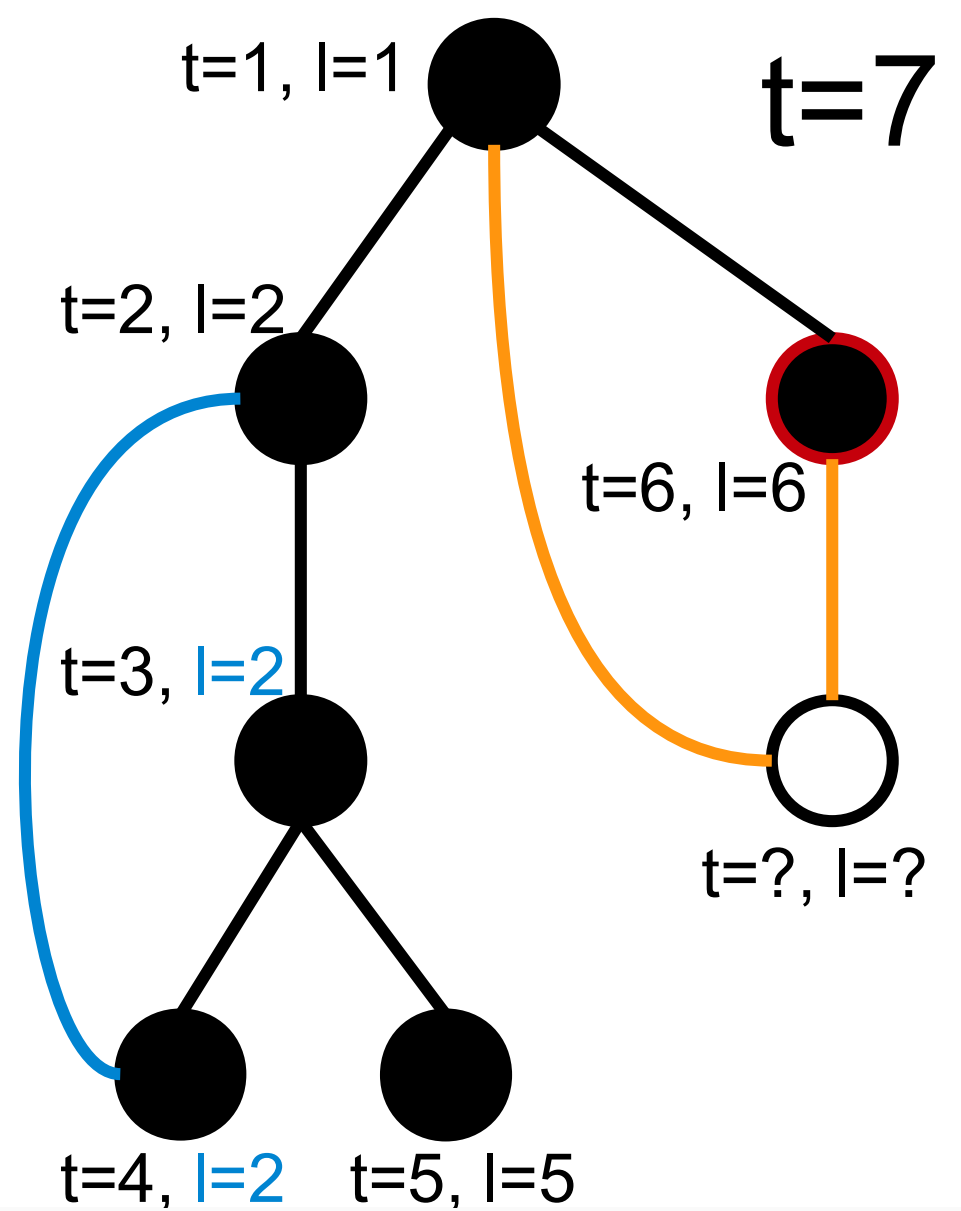
Example



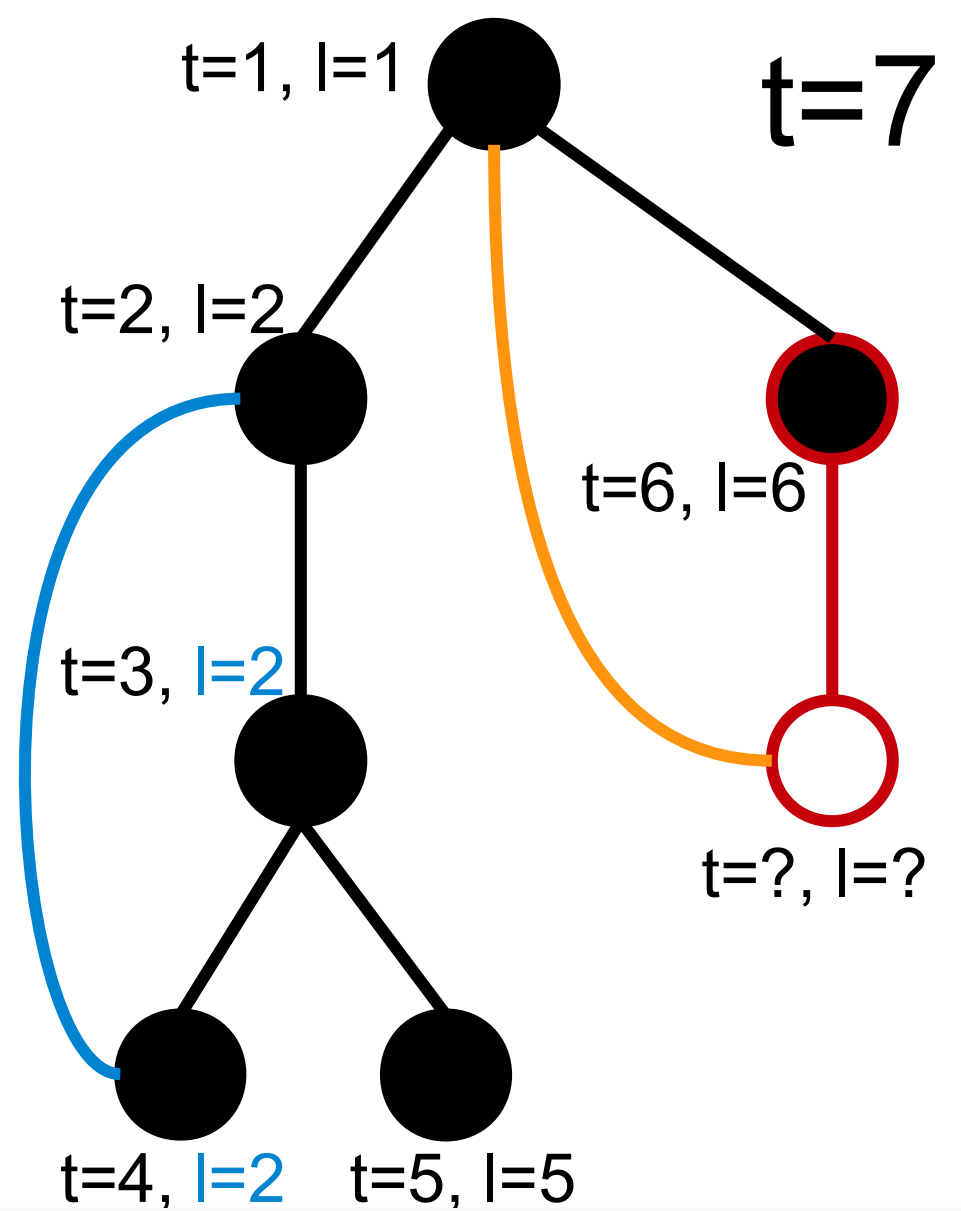
Example



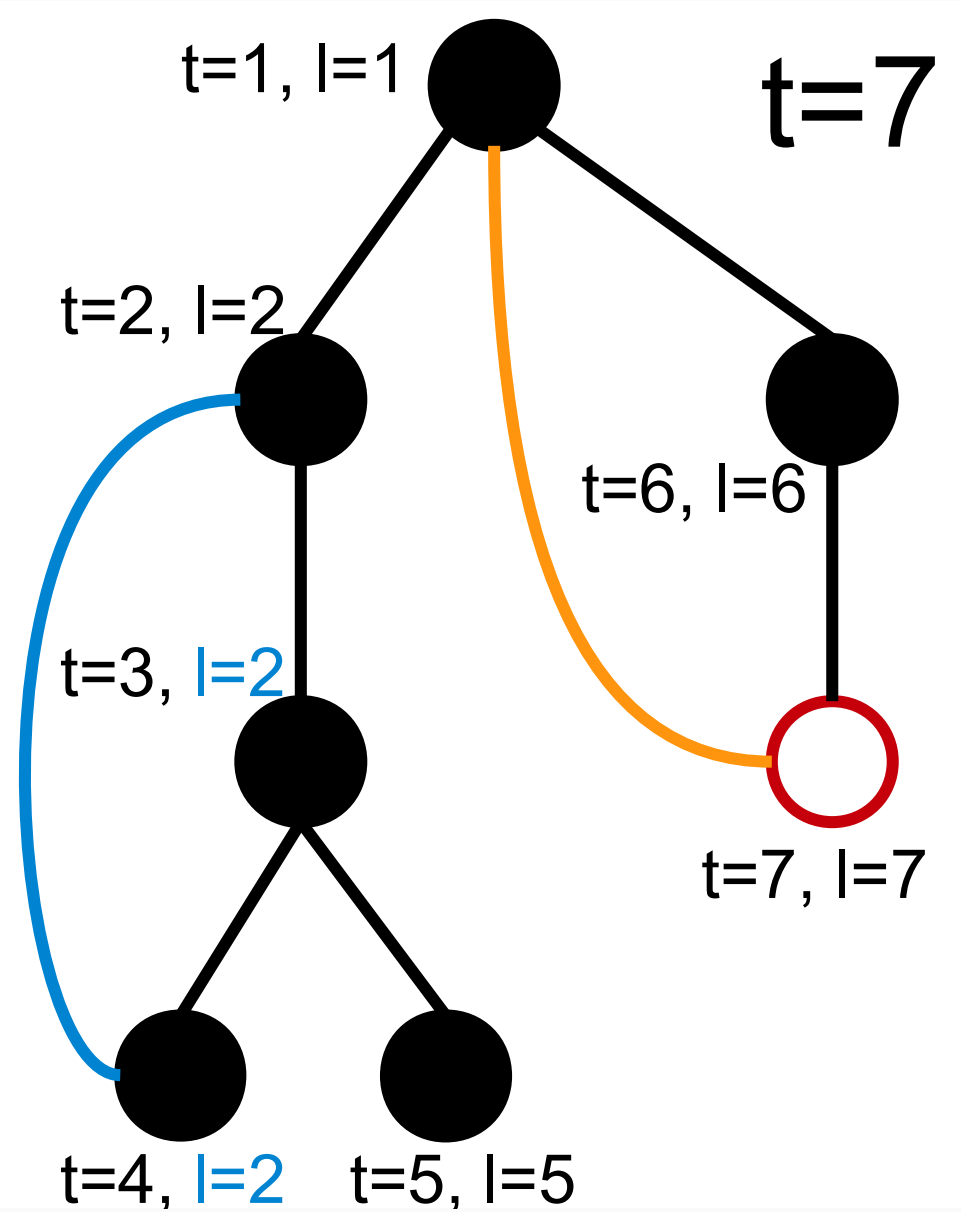
Example



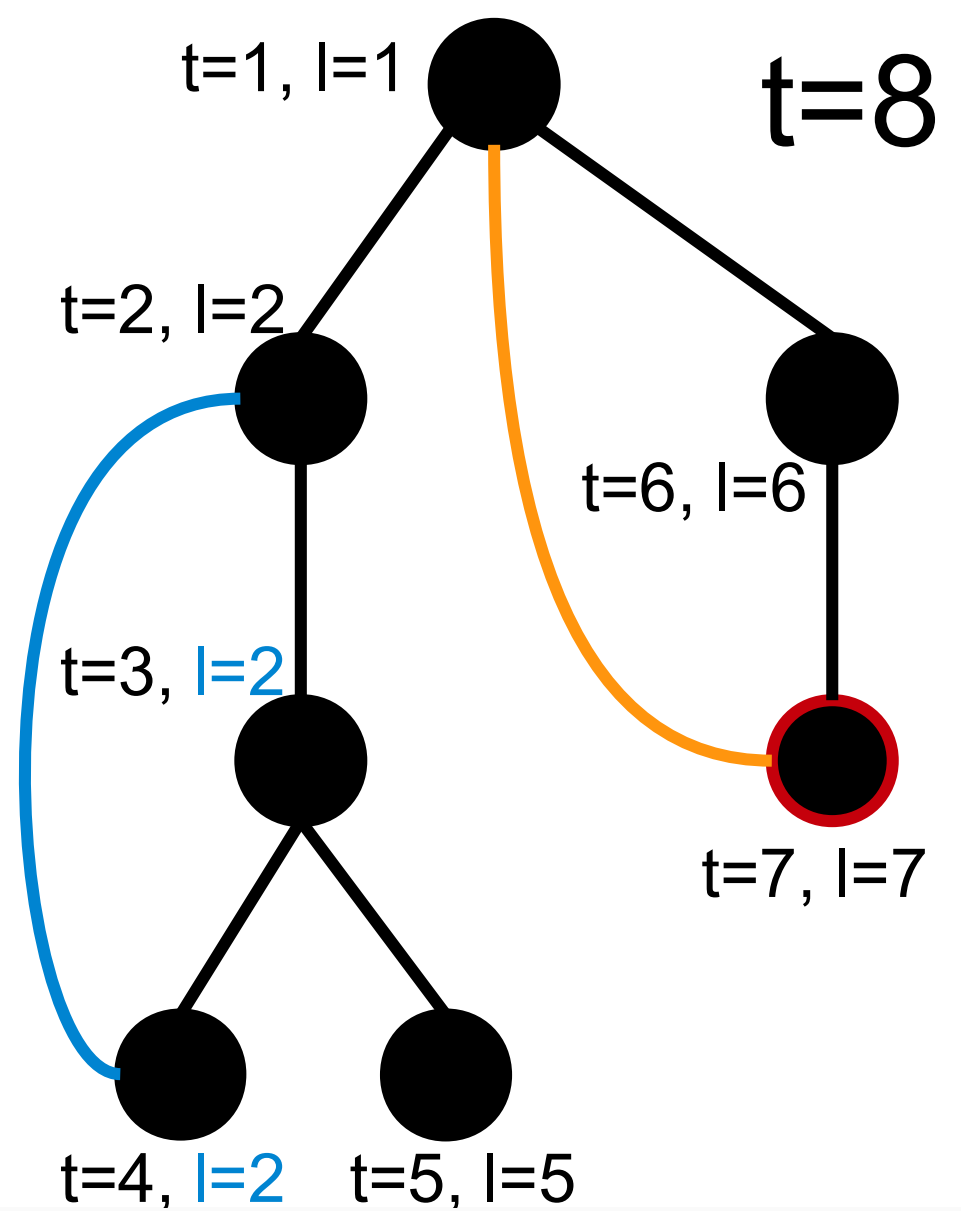
Example



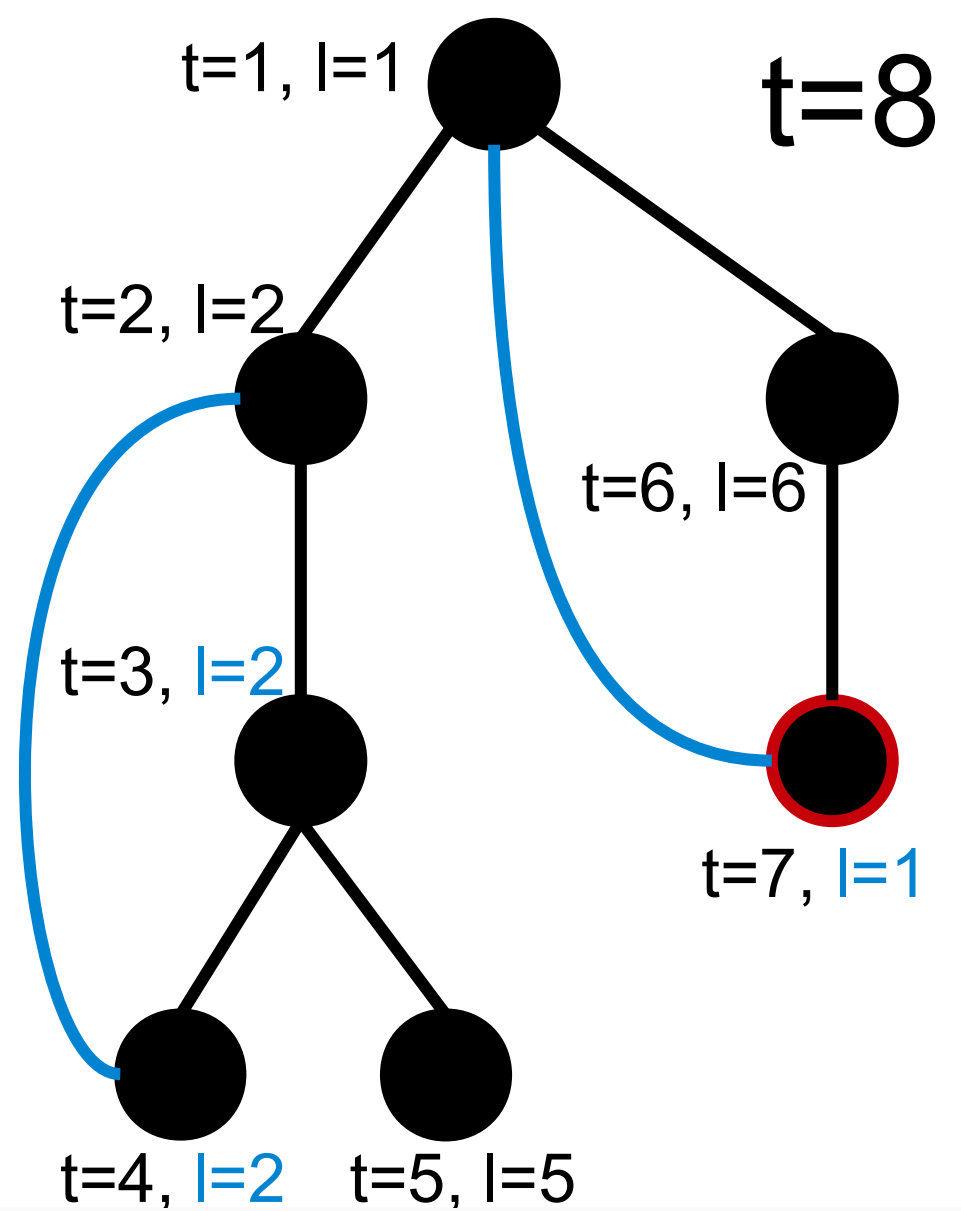
Example



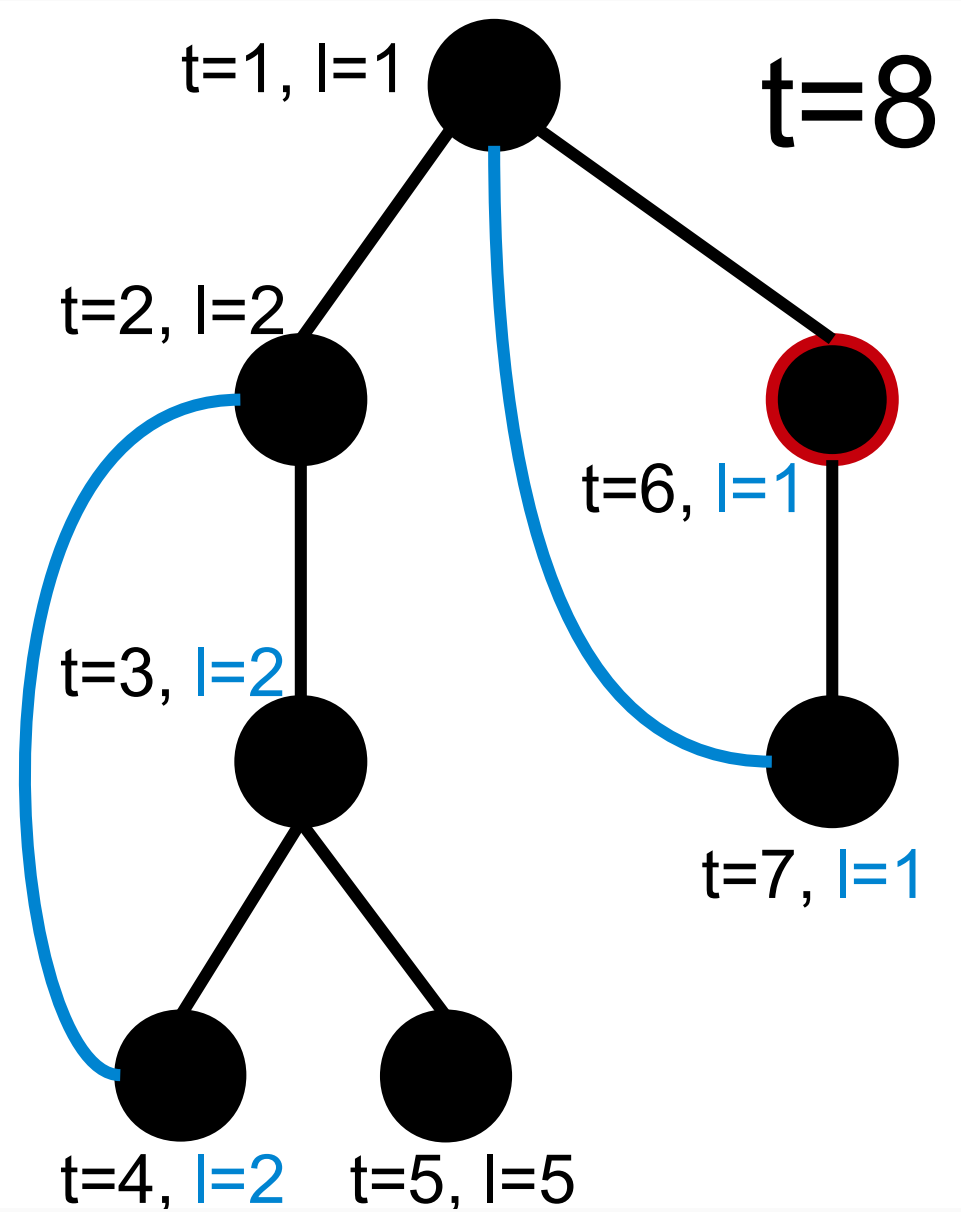
Example



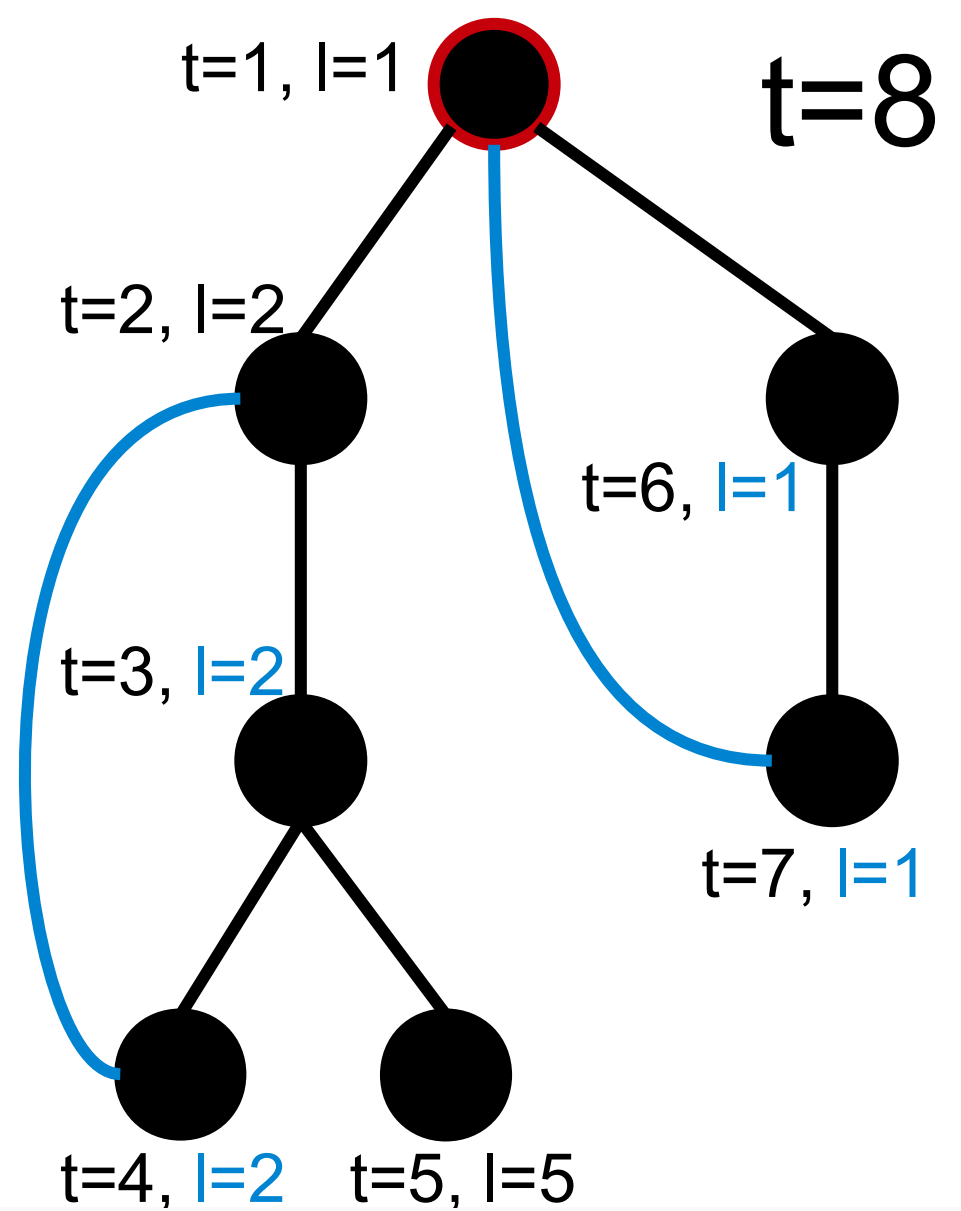
Example



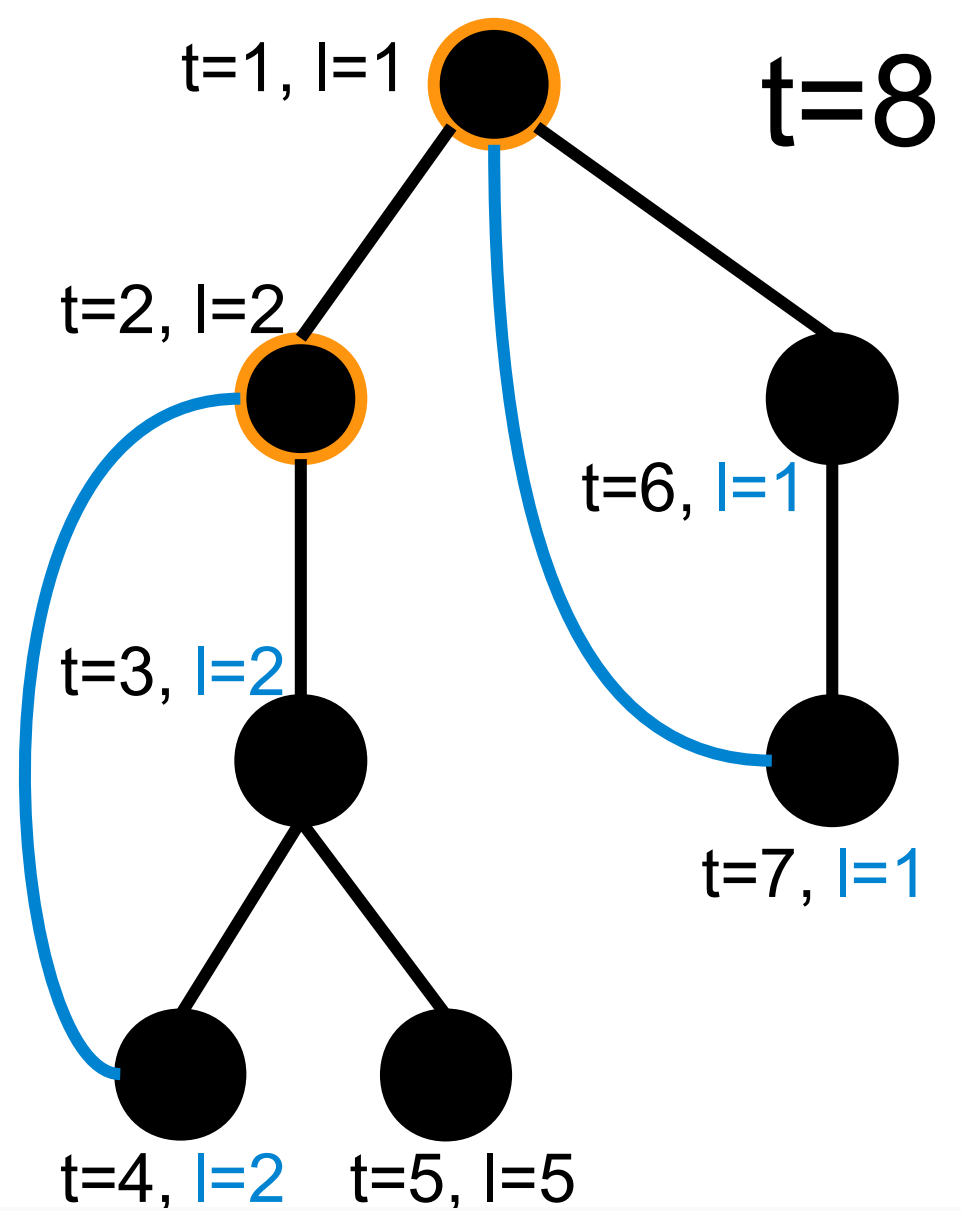
Example



Example



Example



Ordering nodes

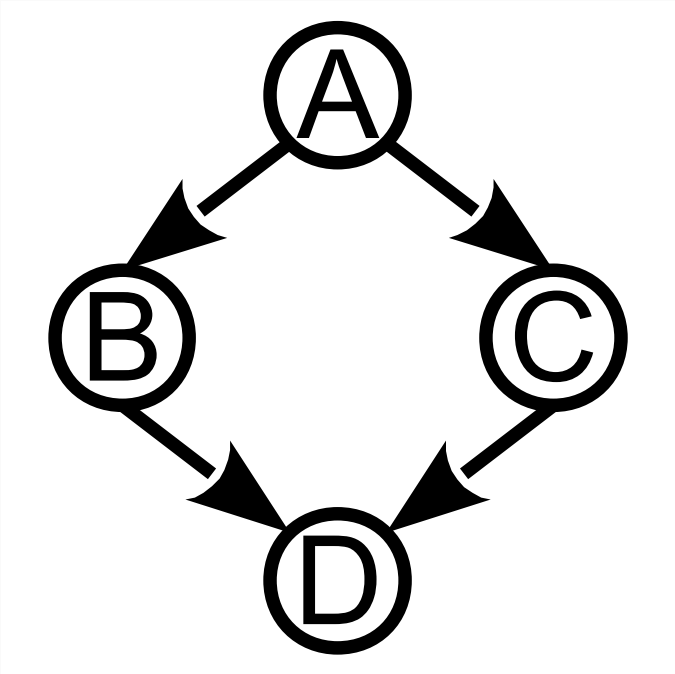
Types of Node Orderings

DFS $\rightarrow$ order **linearly** nodes in G.

- **Preordering:** Order of the 1st visit.
- **Postordering:** Order of the last visit.
- **Reverse Postordering:**
  Opposite order of the last visit $\rightarrow$ **topological sorting**

- **Topological sorting:**
  $\forall \langle u, v \rangle \in E$, u comes before v in the ordering.

Example



DFS Traversal (A, B, D, B, A, C, A) or (A, C, D, C, A, B, A).

- Preordering: (A, B, D, C) or (A, C, D, B)
- Postordering: (D, B, C, A) or (D, C, B, A)
- Reverse Postordering: (A, C, B, D) or (A, B, C, D)