CS2040S
Data Structures and Algorithms

DFS

Housekeeping

Midterms are still being graded.

We also still have a make-up to run before grading everyone entirely.

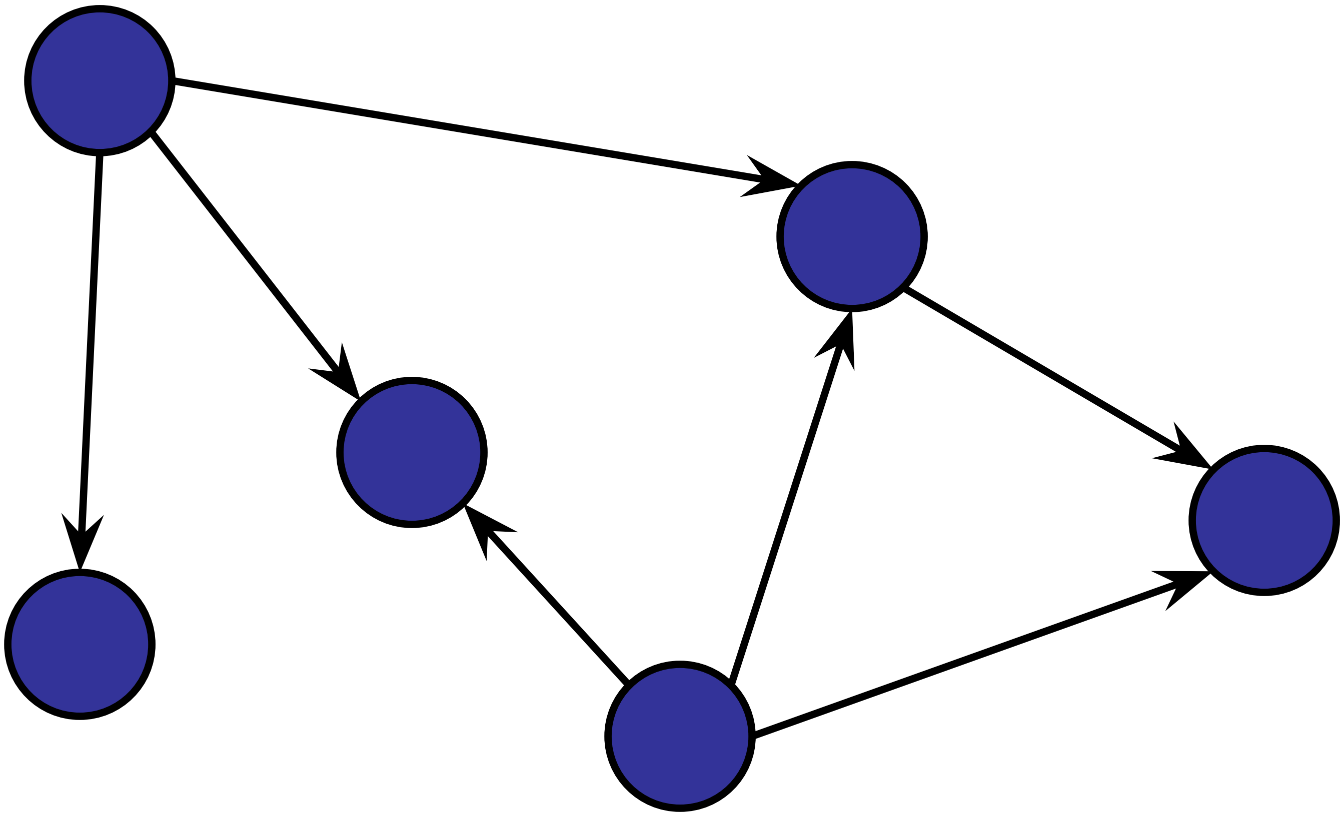
Roadmap

Algorithms on Directed Graphs

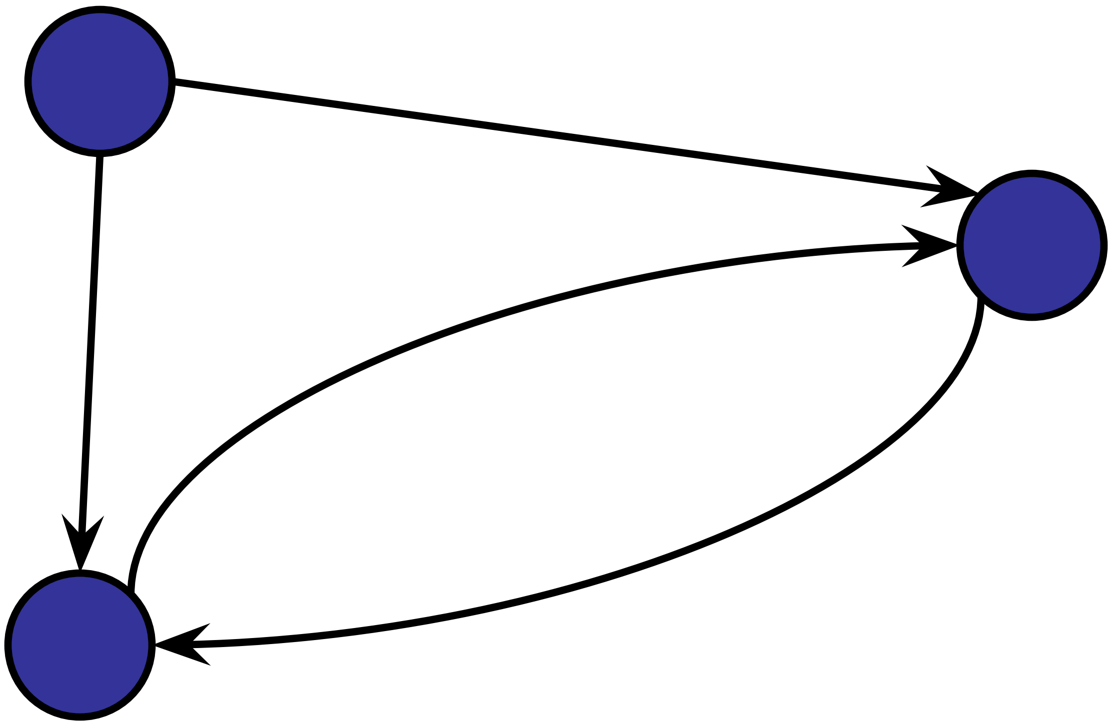
- Searching directed graphs (DFS / BFS)
- Topological Sort
- Connected Components

More Algorithms on Undirected Graphs

Examples of Directed Graphs



Examples of Directed Graphs



Recall: Directed Graph

Graph consists of two types of elements:

Nodes (or vertices)

- At least one.

Edges (or arcs)

- Each edge connects two nodes in the graph
- Each edge is unique.
- Each edge is **directed**.

Directed Graphs Are Great For:

- Modelling Dependencies

- Modelling one-way connections

Directed Graphs Are Great For:

- Modelling Dependencies

- Modelling one-way connections

C++ does not allow circular type definitions.

Directed Graphs Are Great For:

- Modelling Dependencies

- Modelling one-way connections

When you install packages/libraries

how do we know which ones to install first?

Example: Scheduling

Set of tasks for baking cookies:

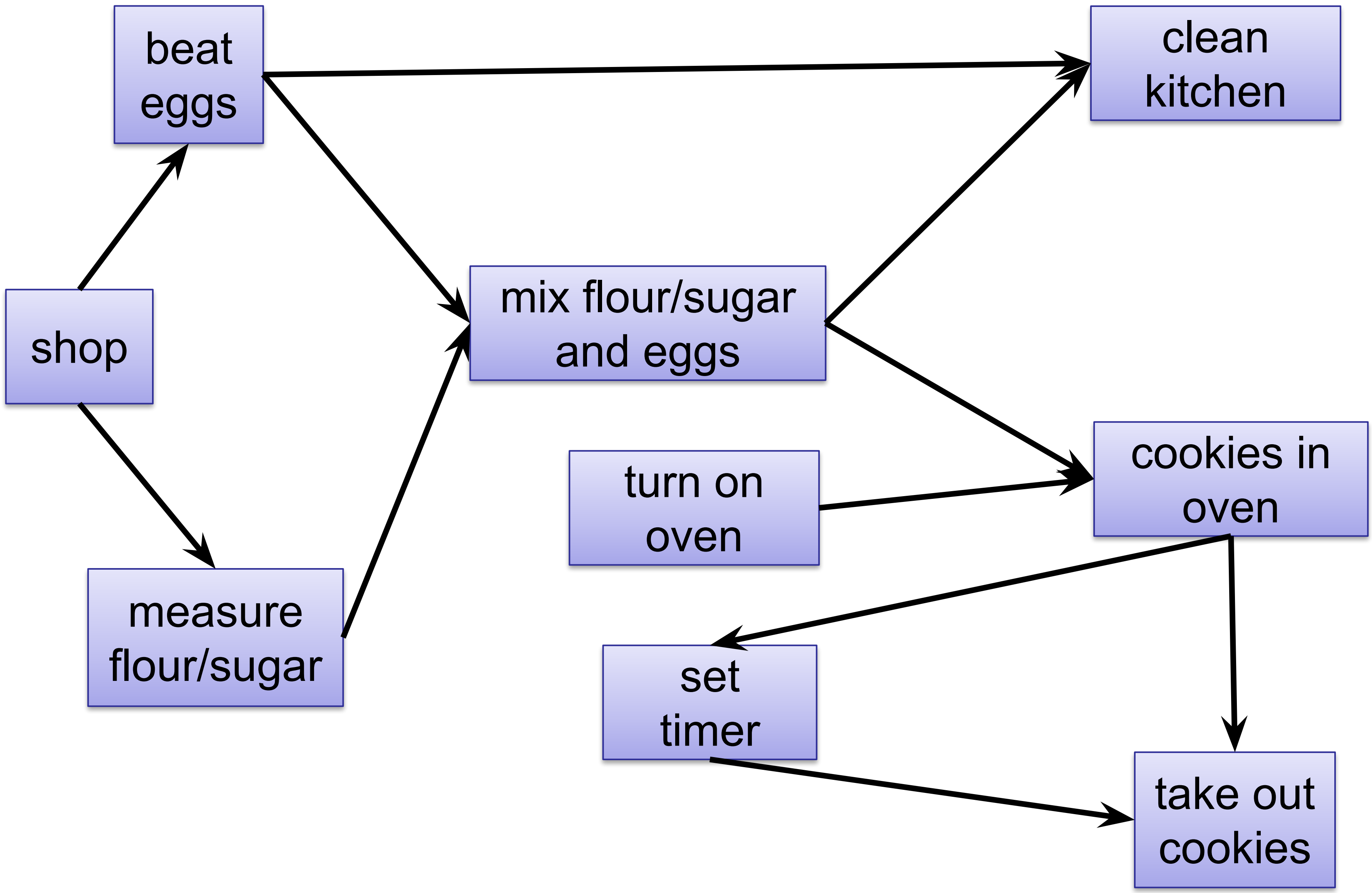
- Shop for groceries
- Put the cookies in the oven
- Clean the kitchen
- Beat the eggs in a bowl
- Measure the flour and sugar in a bowl
- Mix the eggs with the flour and sugar
- Turn on the oven
- Set the timer
- Take out the cookies

Scheduling

Ordering:

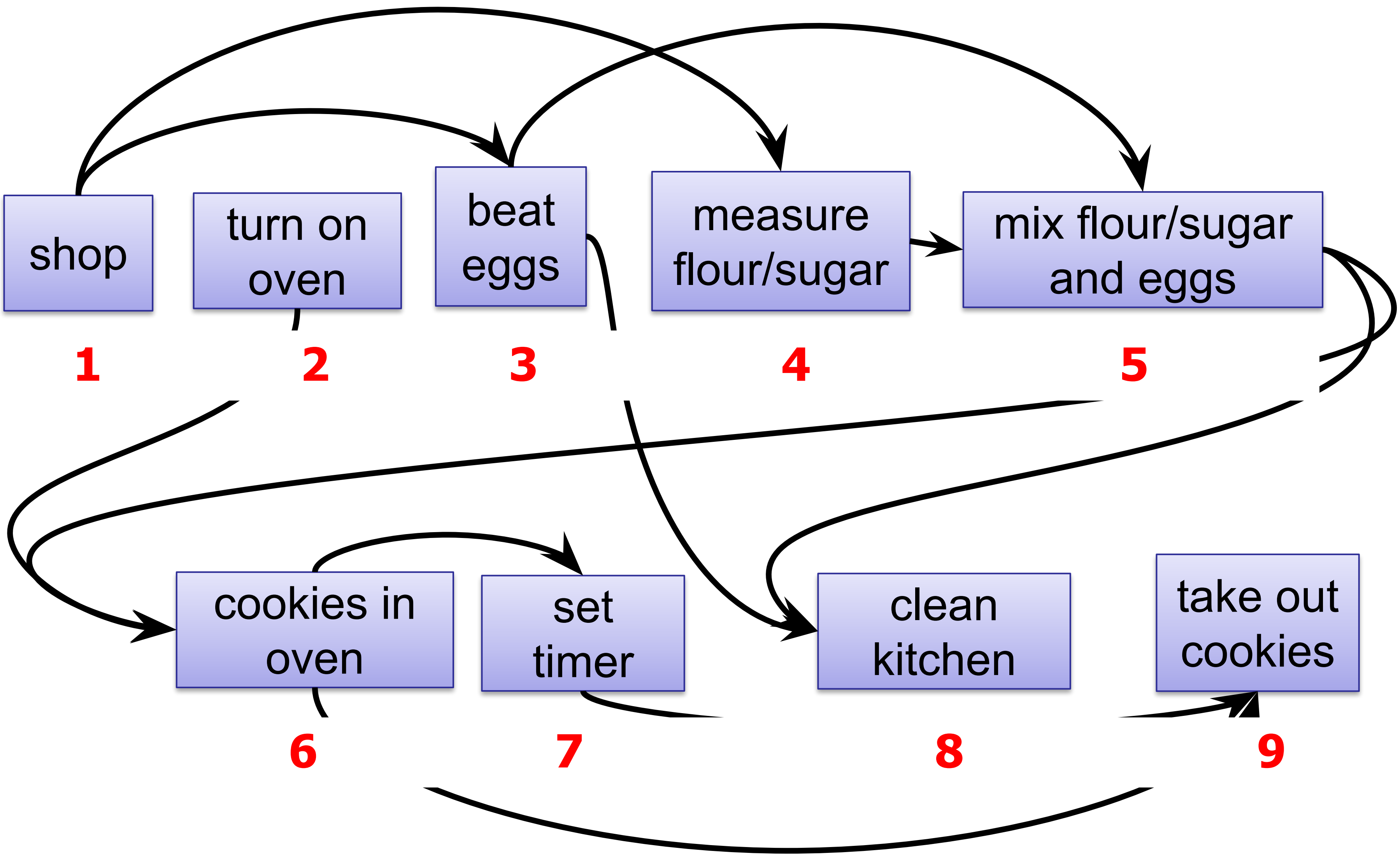
- Shop for groceries before beat the eggs
- Shop for groceries before measure the flour
- Turn on the oven before put the cookies in the oven
- Beat the eggs before mix the eggs with the flour
- Measure the flour before mix the eggs with the flour
- Put the cookies in the oven before set the timer
- Measure the flour before clean the kitchen
- Beat the eggs before clean the kitchen
- Mix the flour and the eggs before clean the kitchen

Scheduling



beat eggs
clean kitchen
shop
mix flour/sugar and eggs
turn on oven
cookies in oven
measure flour/sugar
set timer
take out cookies

Topological Ordering



Topological Order

Properties:

1. Sequential total ordering of all nodes

1. shop

2. turn on oven

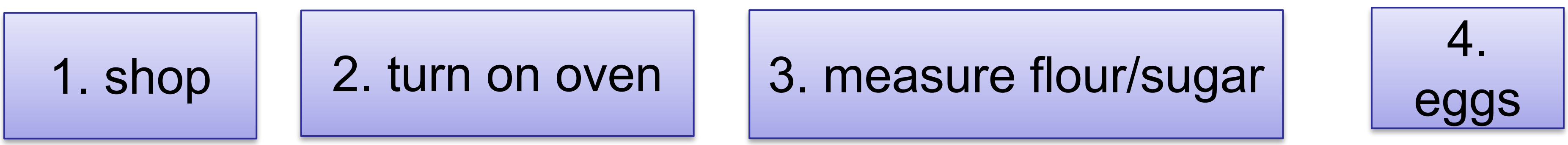
3. measure flour/sugar

4. eggs

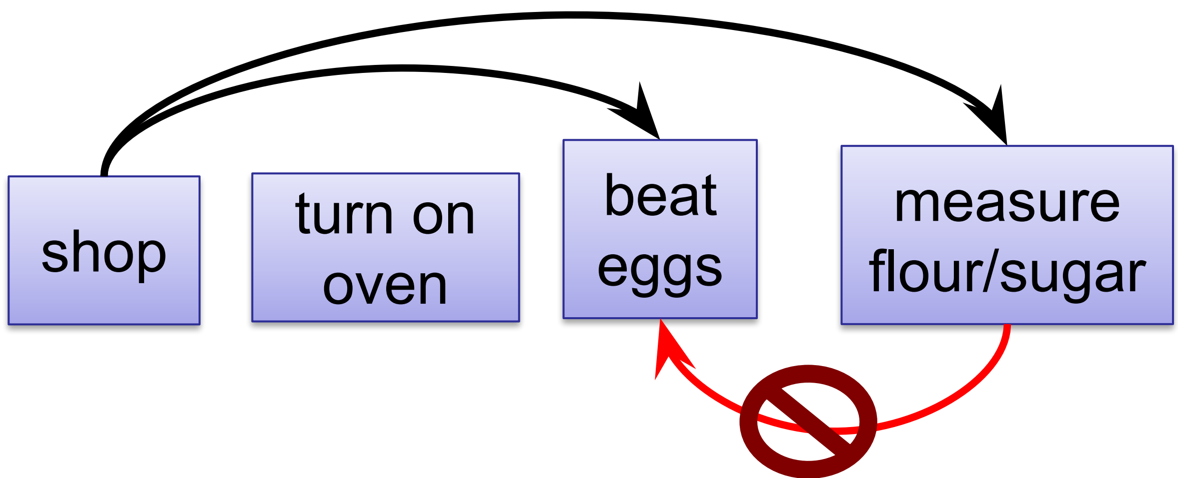
Topological Order

Properties:

1. Sequential total ordering of all nodes



2. Edges from original graph only point forward

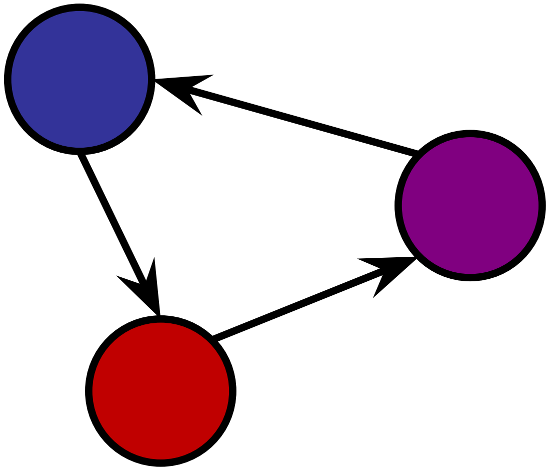


Does every directed graph have a topological ordering?

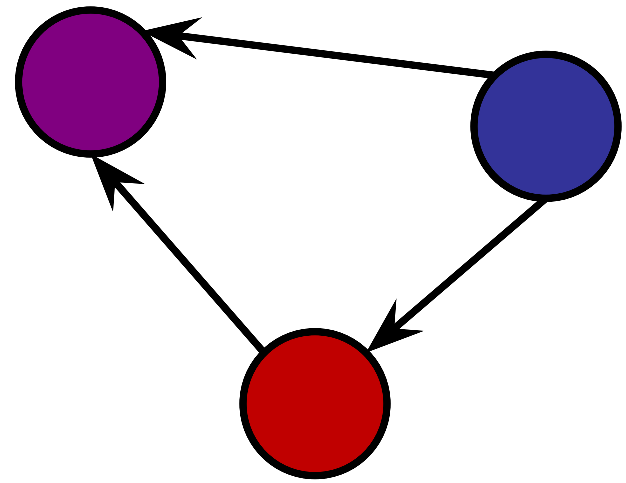
1. Yes
2. No
3. Only if the adjacency matrix has small second eigenvalue.

Directed Acyclic Graphs

Cyclic

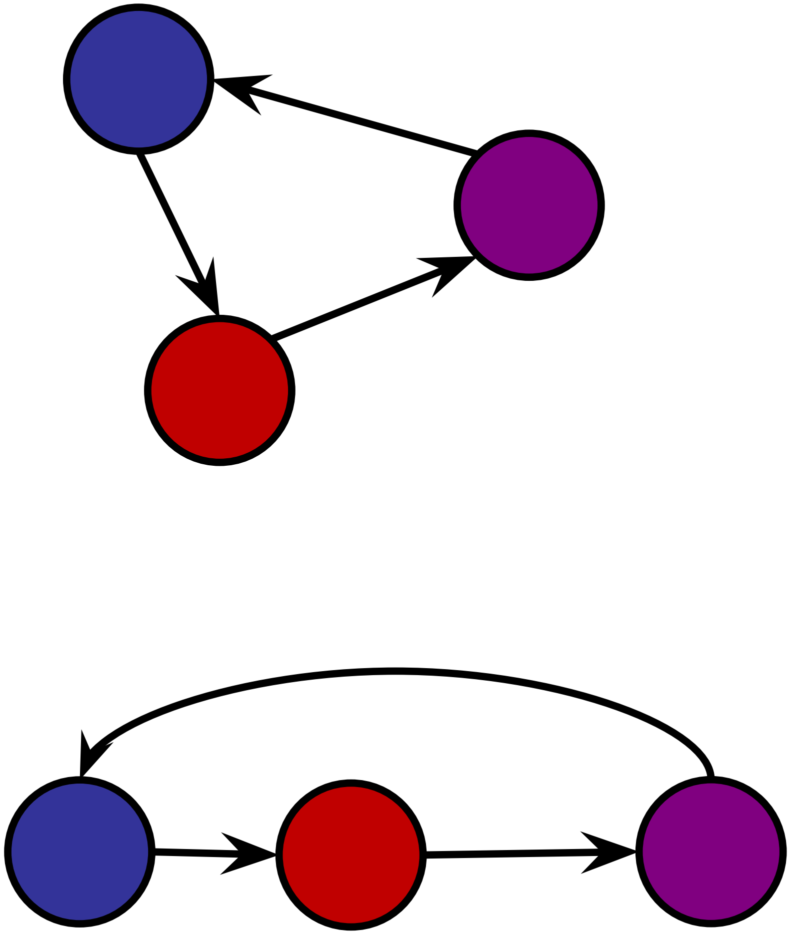


Acyclic

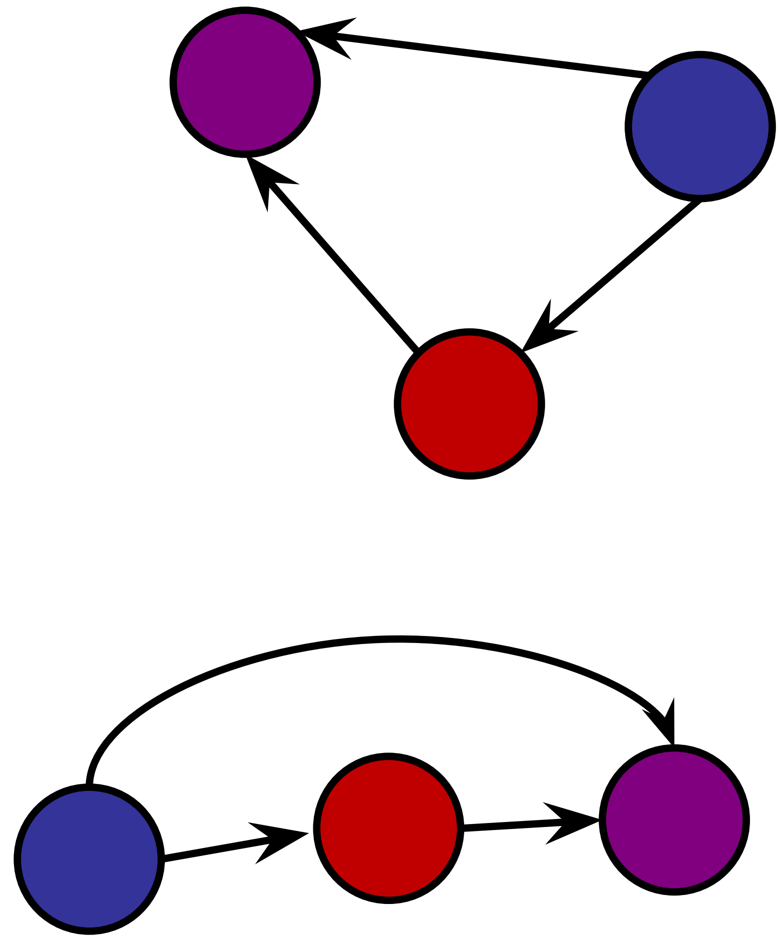


Directed Acyclic Graphs

Cyclic

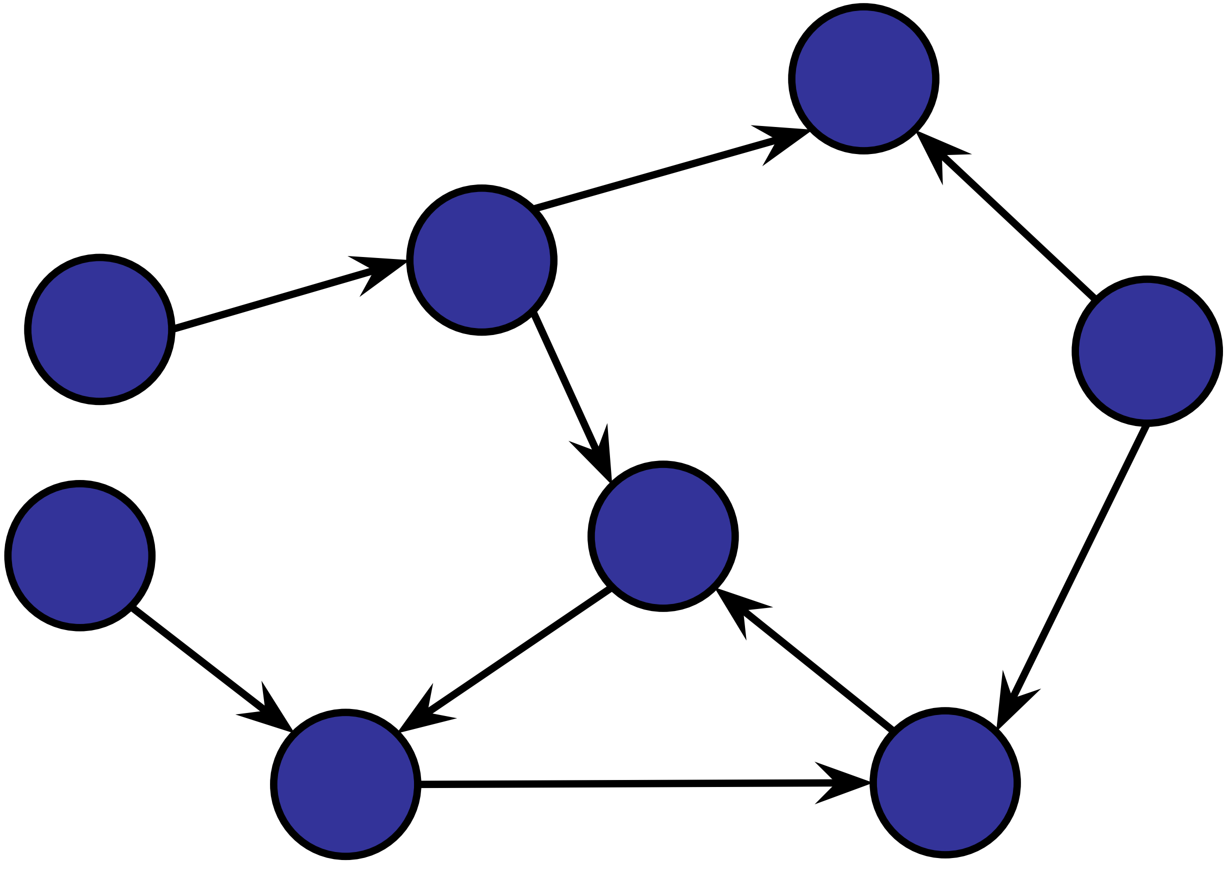


Acyclic



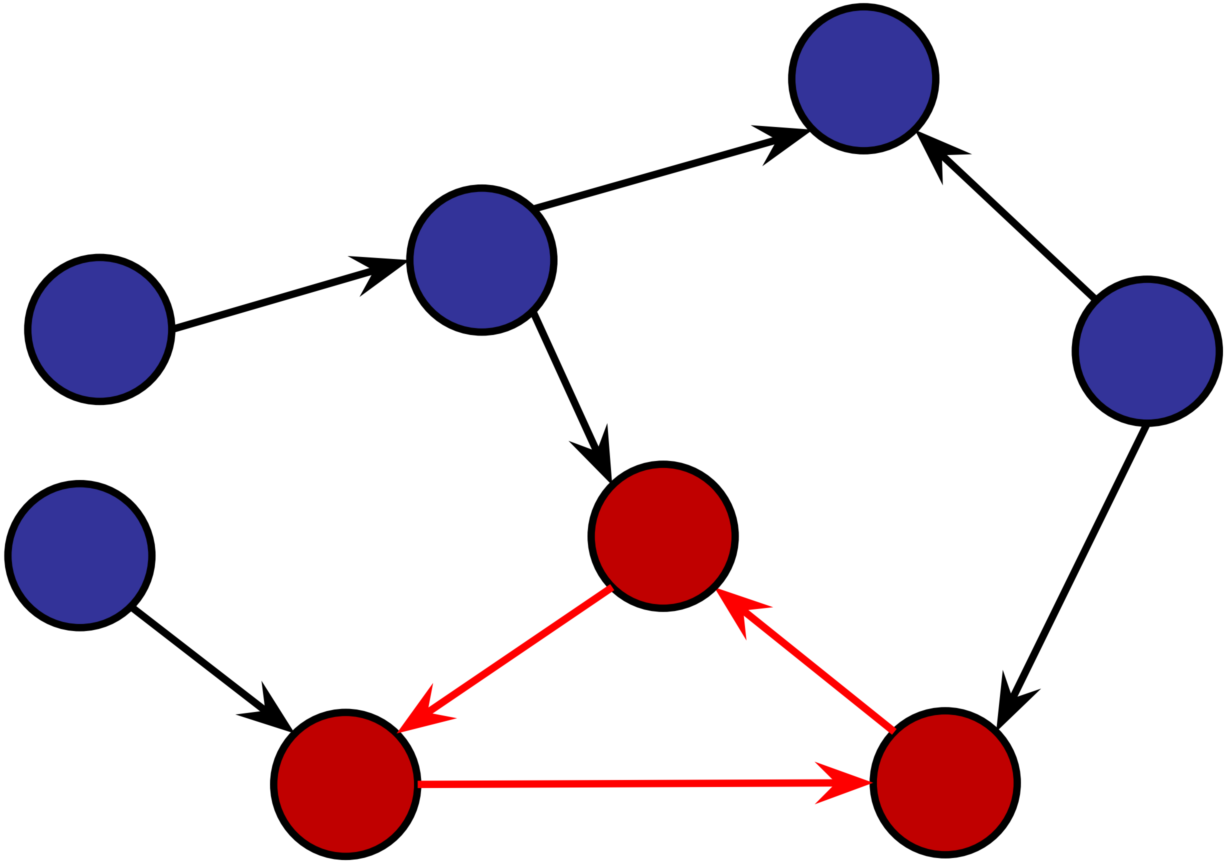
Directed Acyclic Graphs

Does it have a topological ordering?

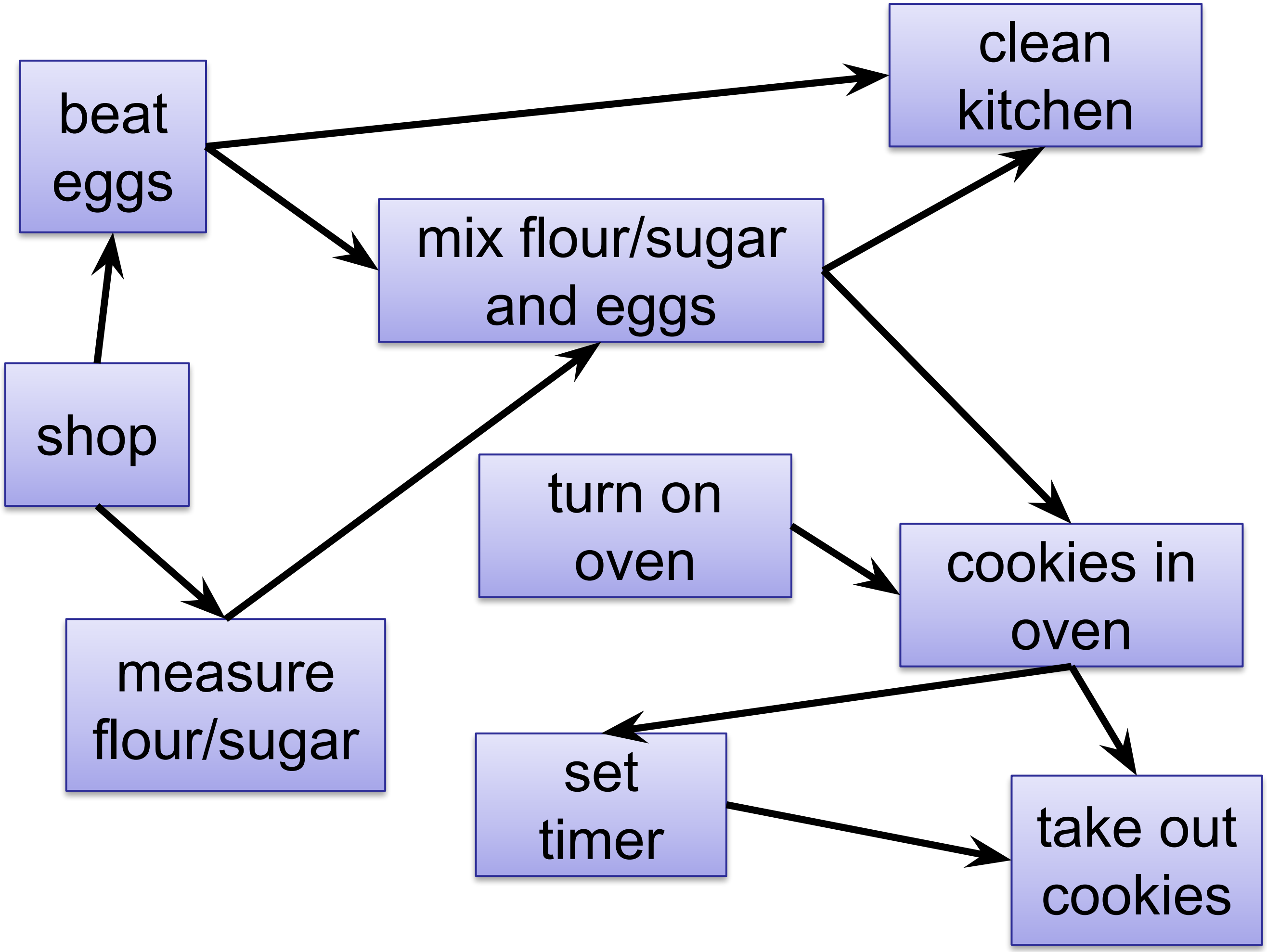


Directed Acyclic Graphs

Does it have a topological ordering?



Directed Acyclic Graph



Topological Sorting

Assuming a graph is acyclic:

A topological sort of the graph produces an ordering of the nodes.

If edge (u, v) is in G, then u must appear before v in the toposort.

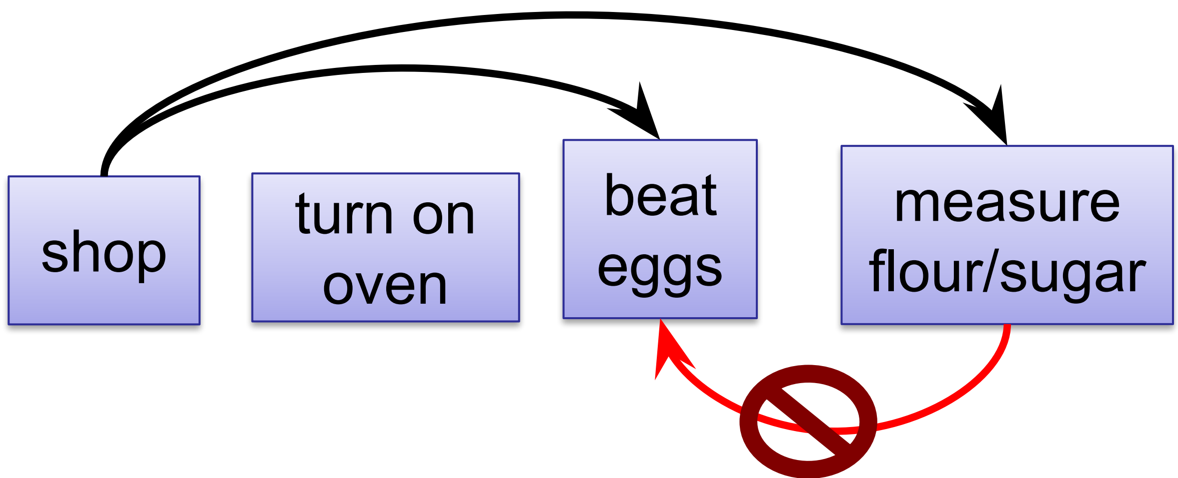
Topological Order

Properties:

1. Sequential total ordering of all nodes

1. shop | 2. turn on oven | 3. measure flour/sugar | 4. eggs

2. Edges only point forward

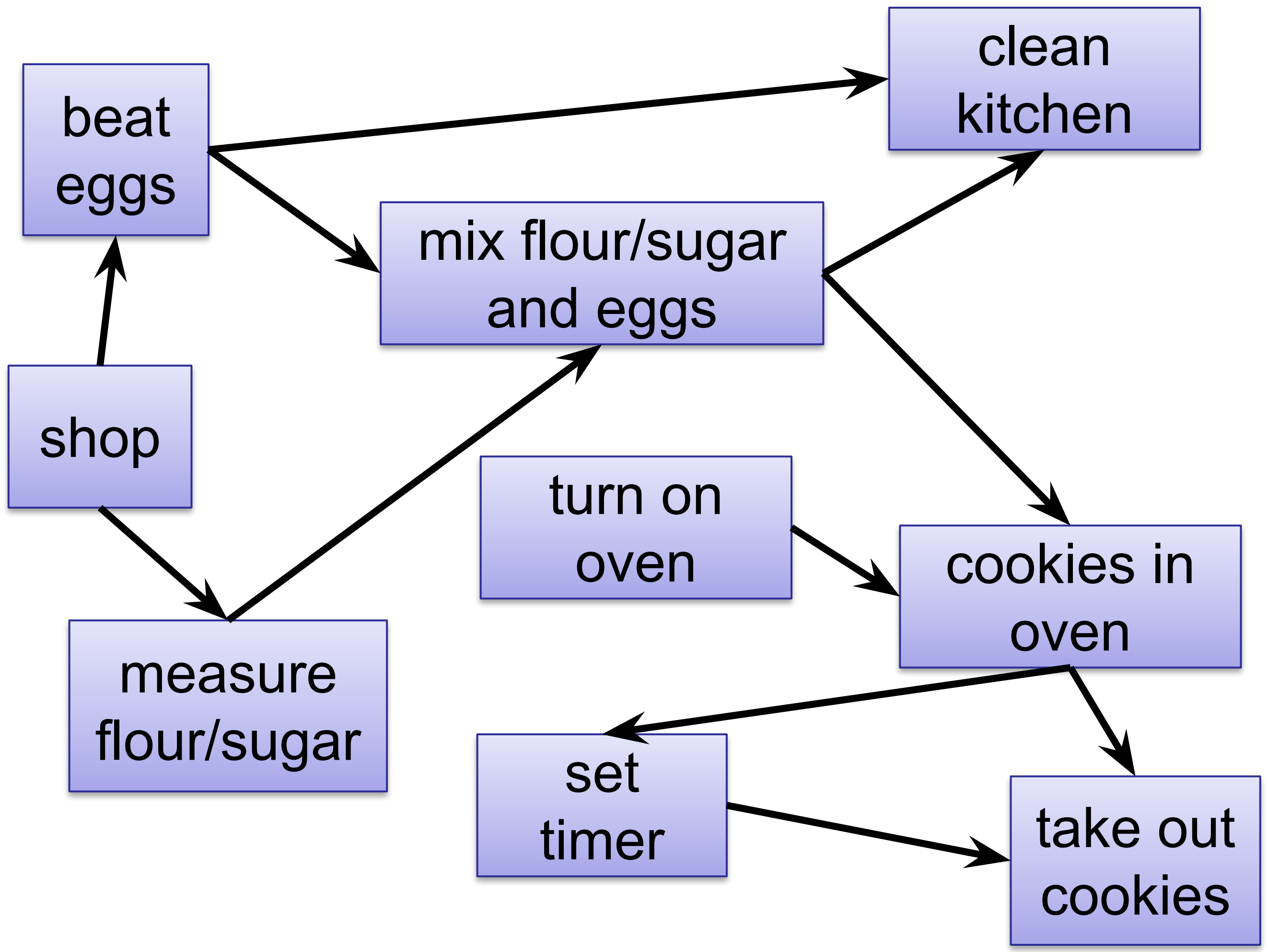


Topological Sorting

How do we produce the graph?

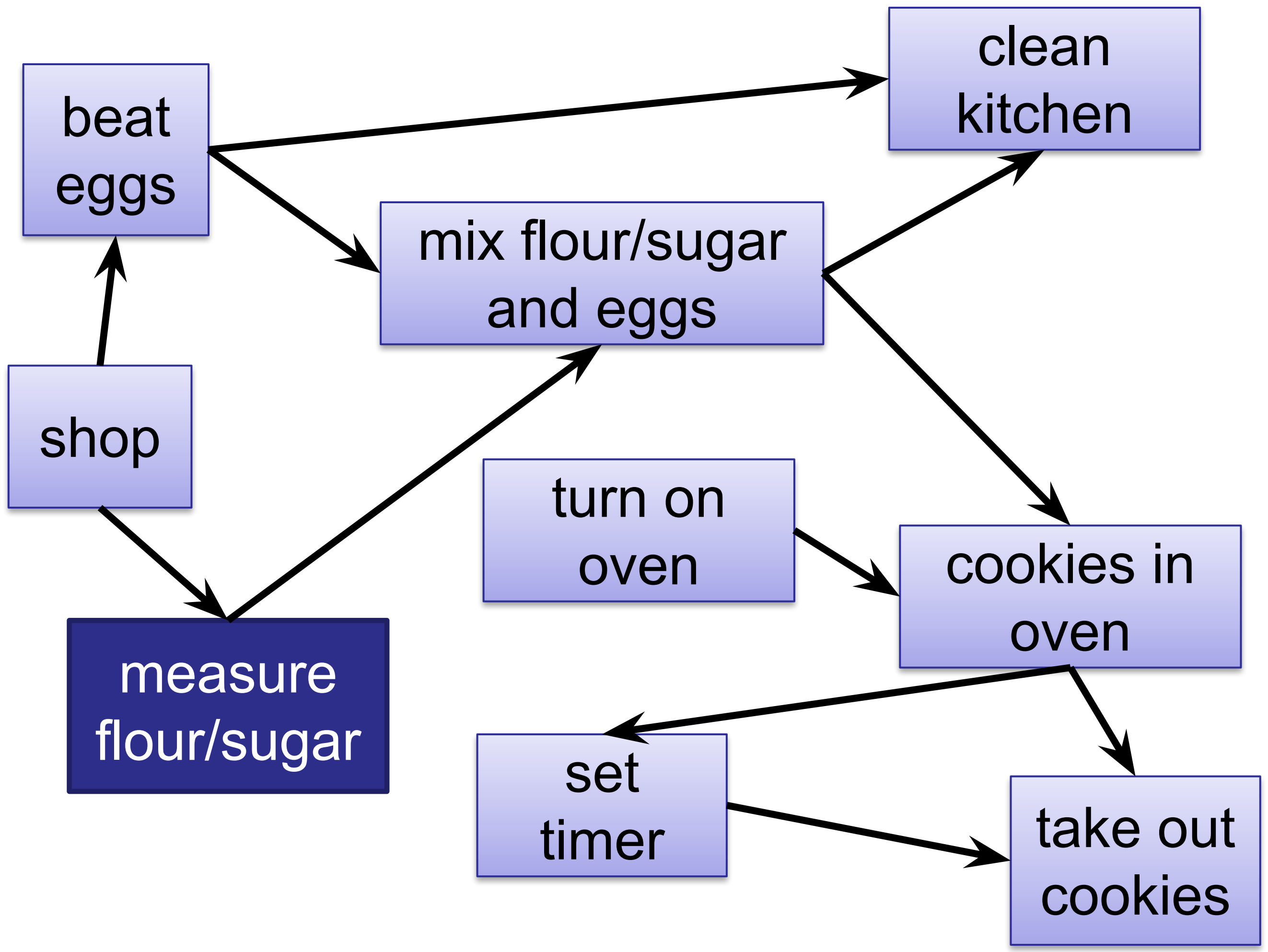
Can we just run DFS?

Depth-First Search



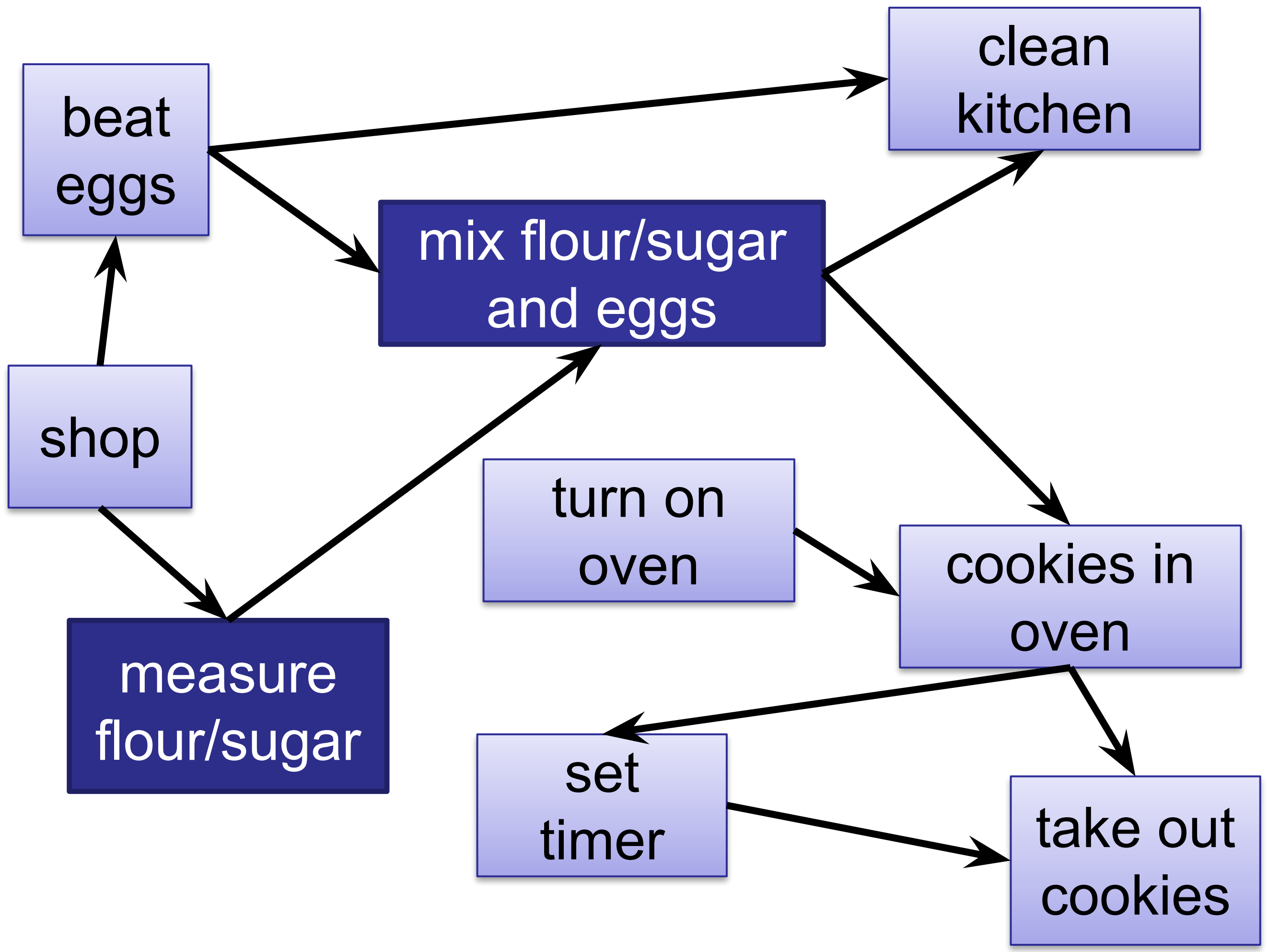
Depth-First Search

1. measure



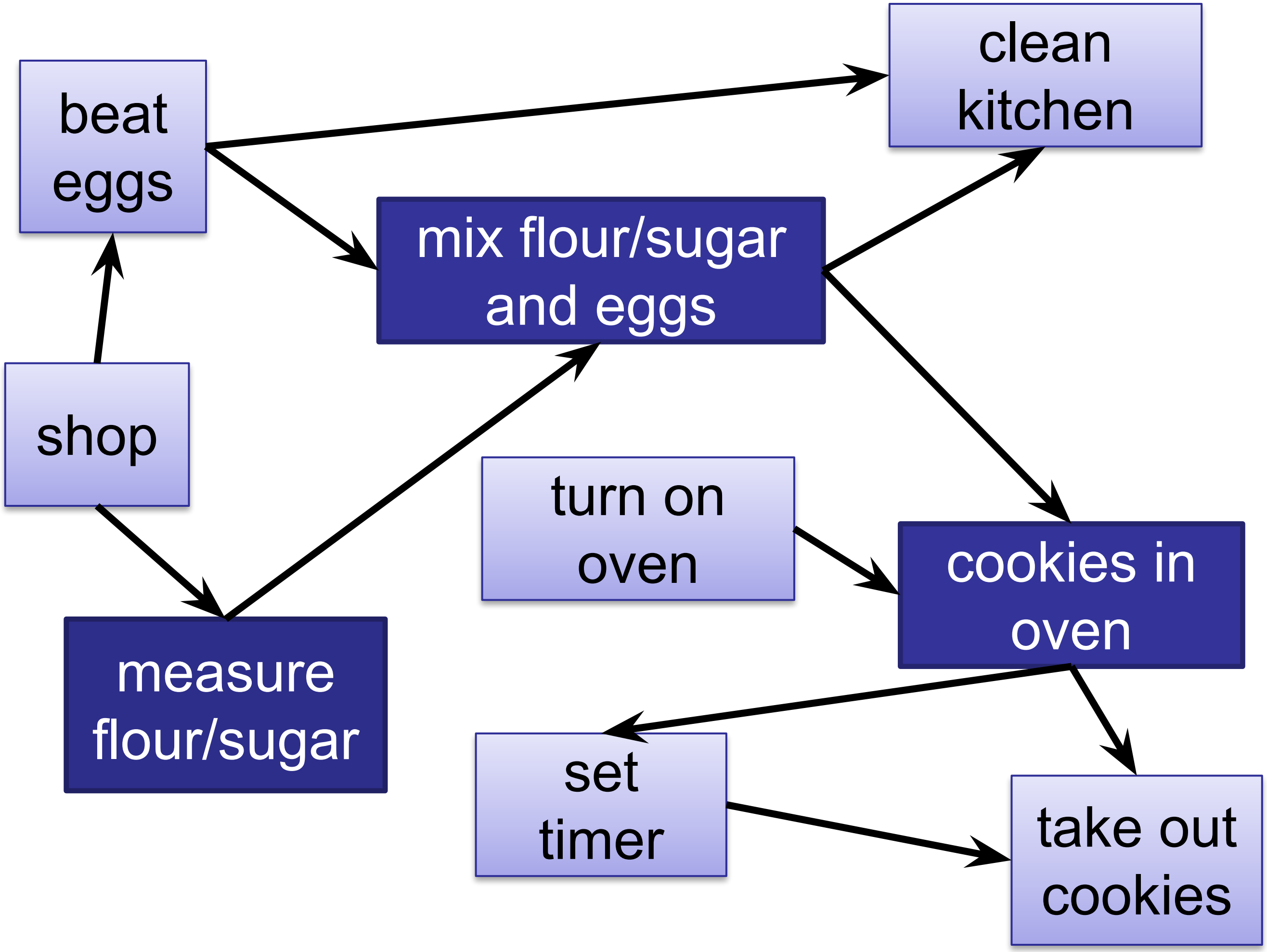
Depth-First Search

1. measure
2. mix



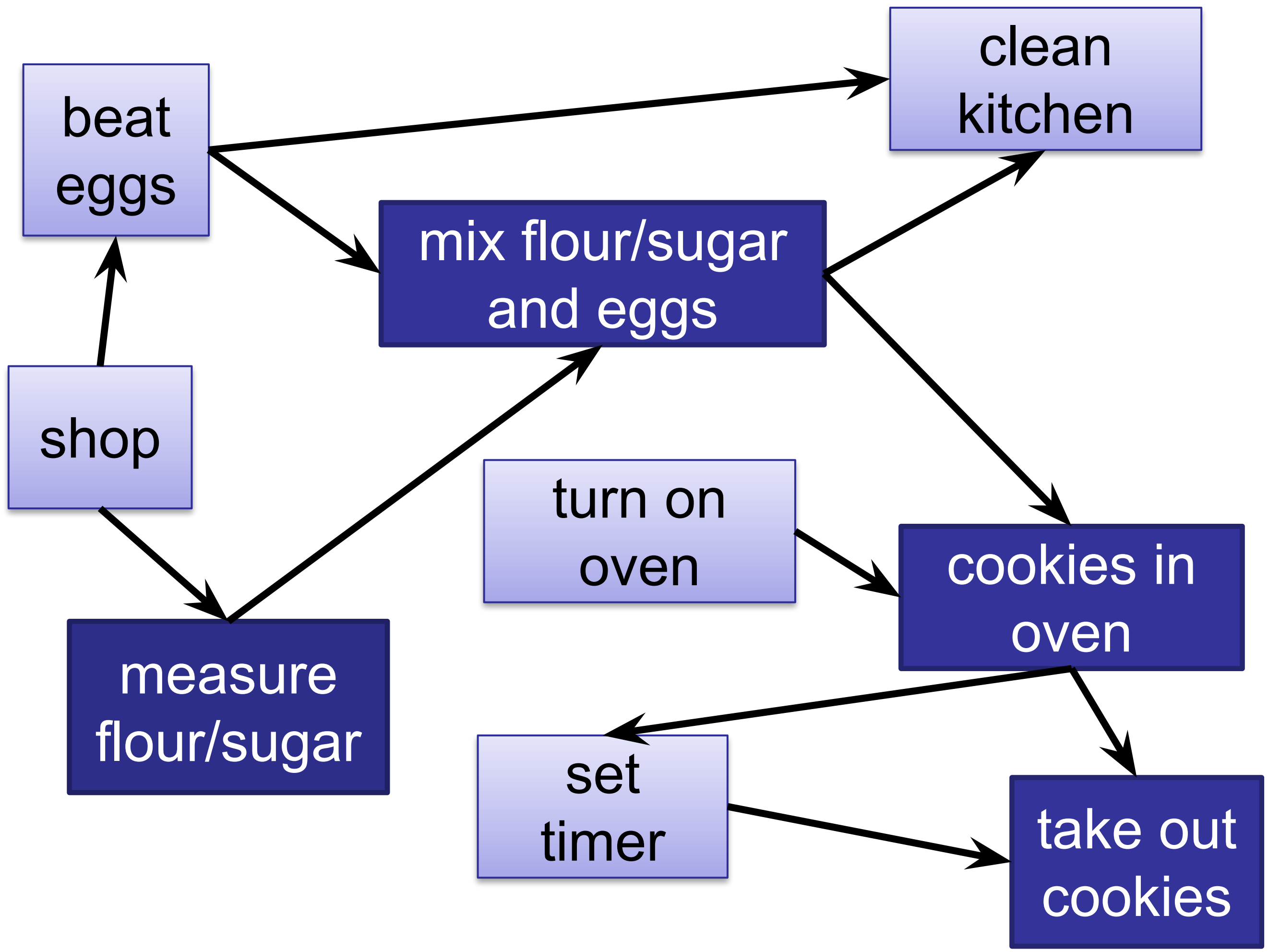
Depth-First Search

1. measure
2. mix
3. in oven



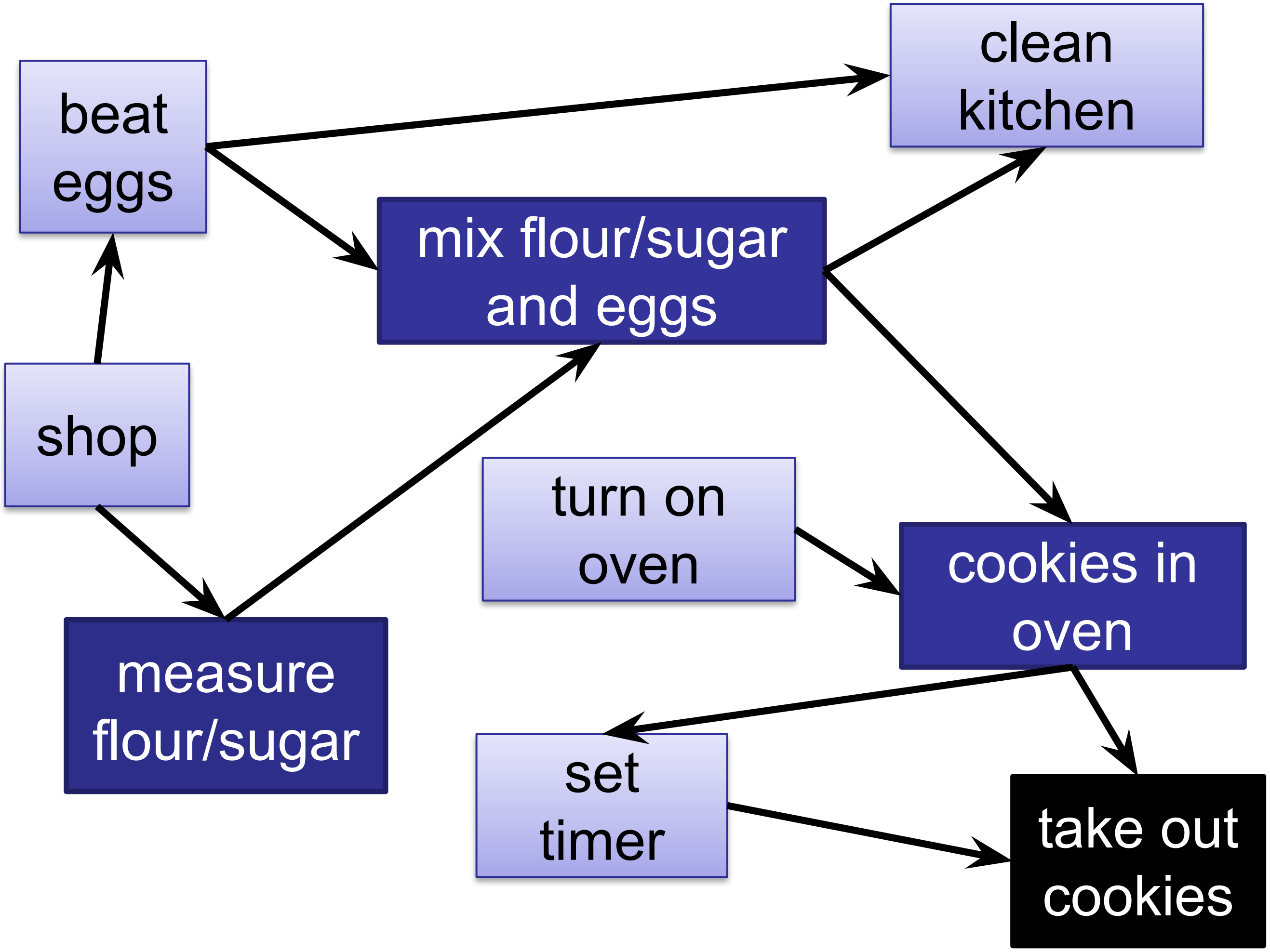
Depth-First Search

1. measure
2. mix
3. in oven
4. take out



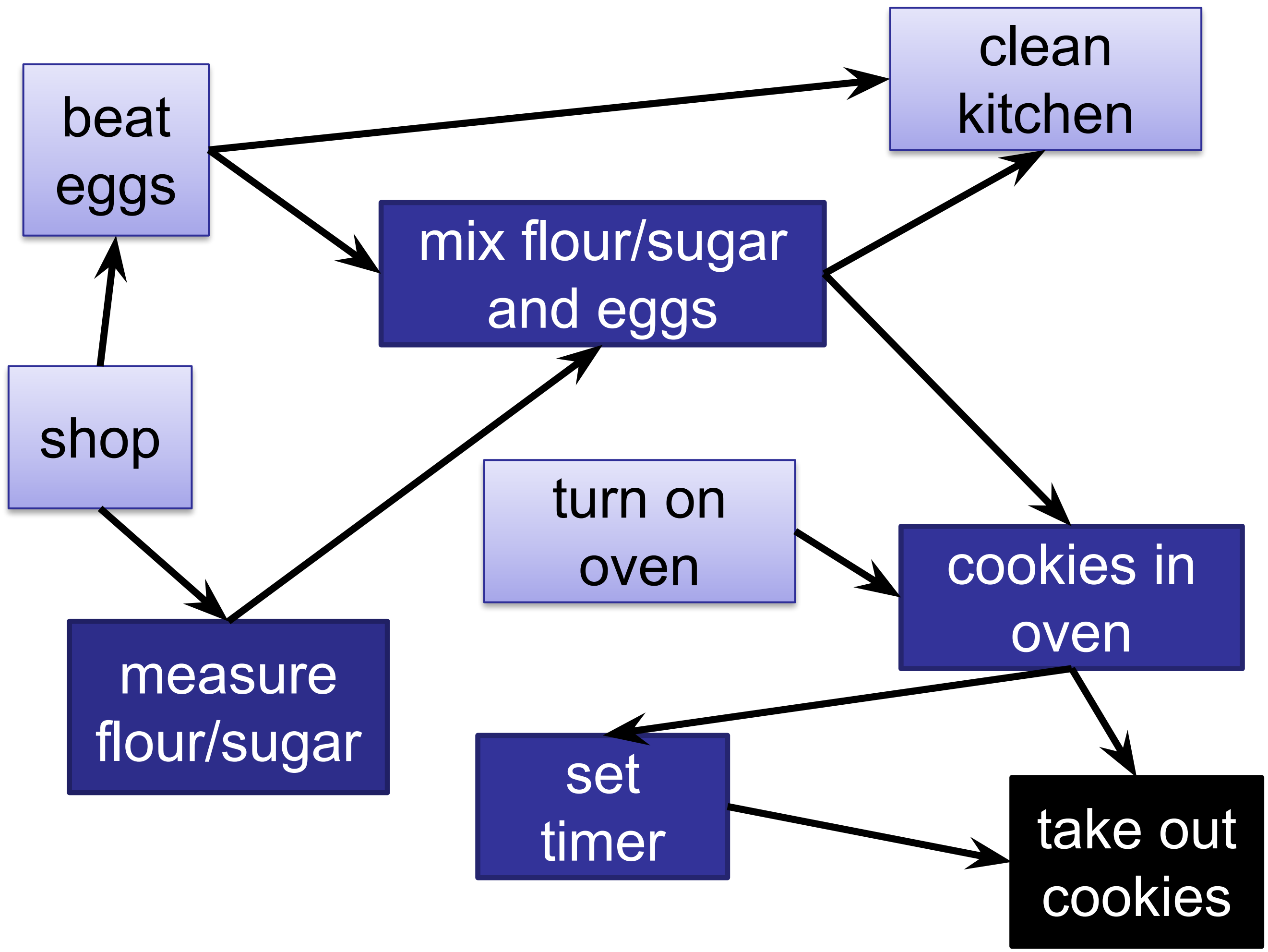
Depth-First Search

1. measure
2. mix
3. in oven
4. take out



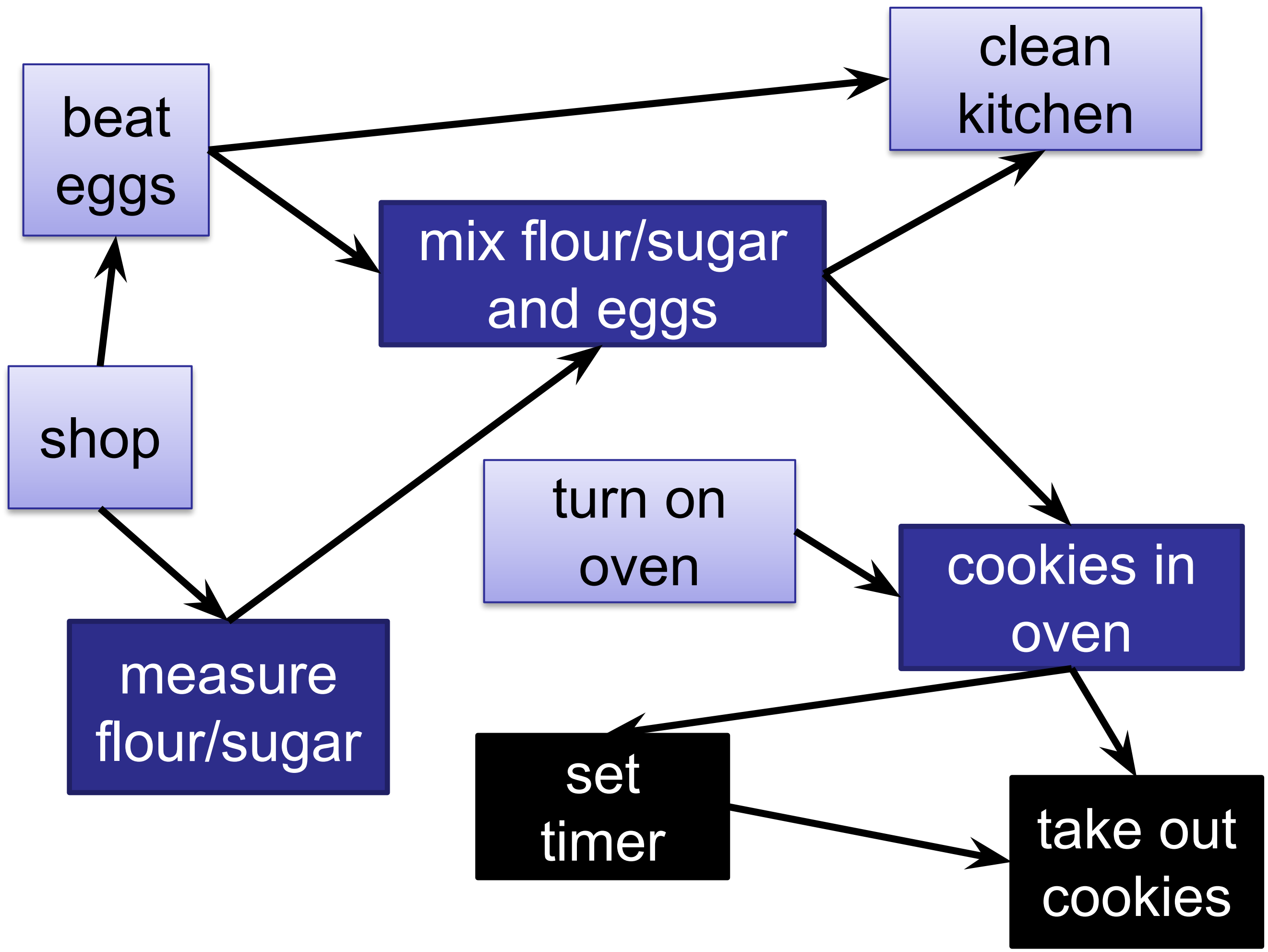
Depth-First Search

1. measure
2. mix
3. in oven
4. take out
5. set timer



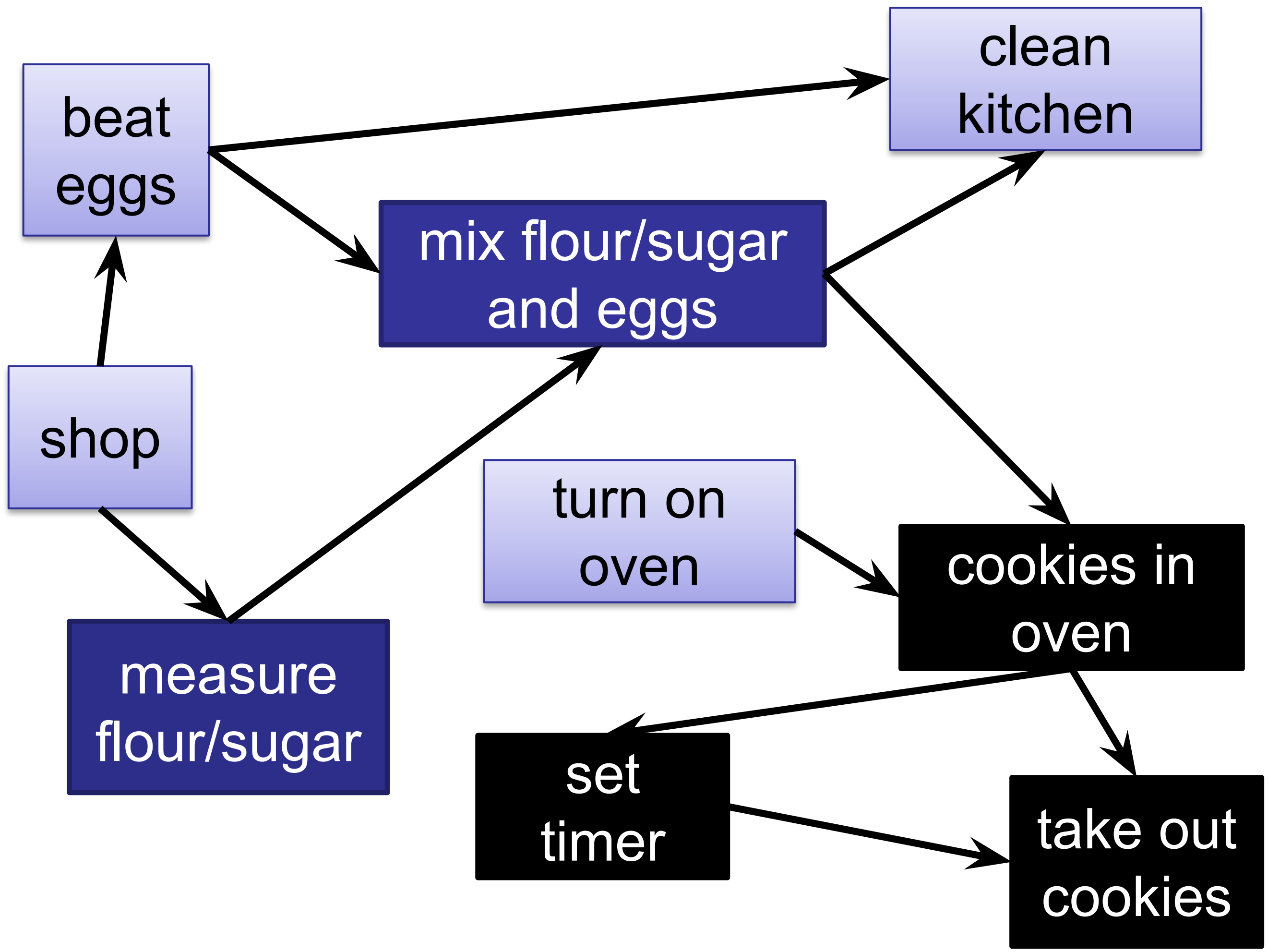
Depth-First Search

1. measure
2. mix
3. in oven
4. take out
5. set timer



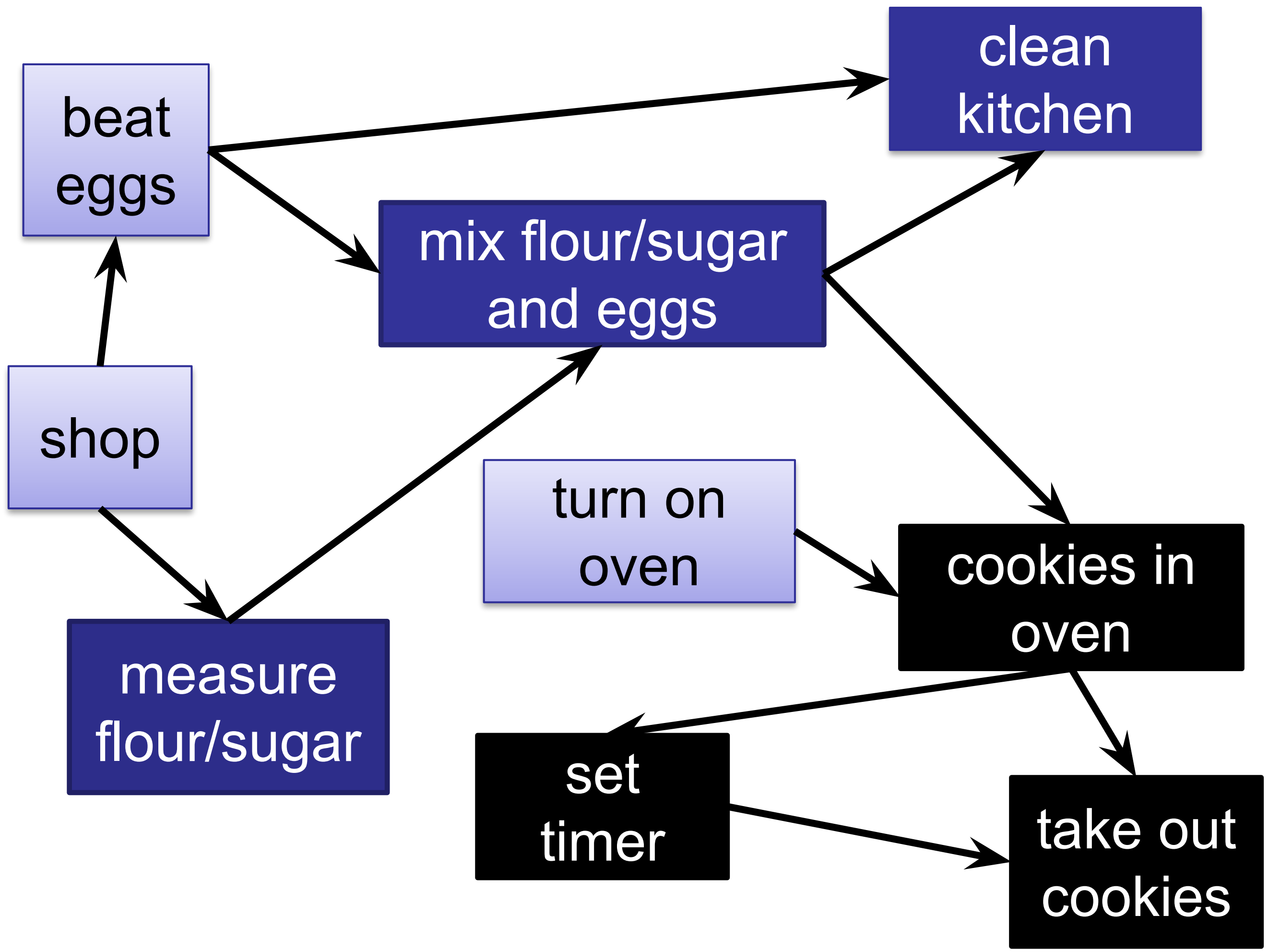
Depth-First Search

1. measure
2. mix
3. in oven
4. take out
5. set timer



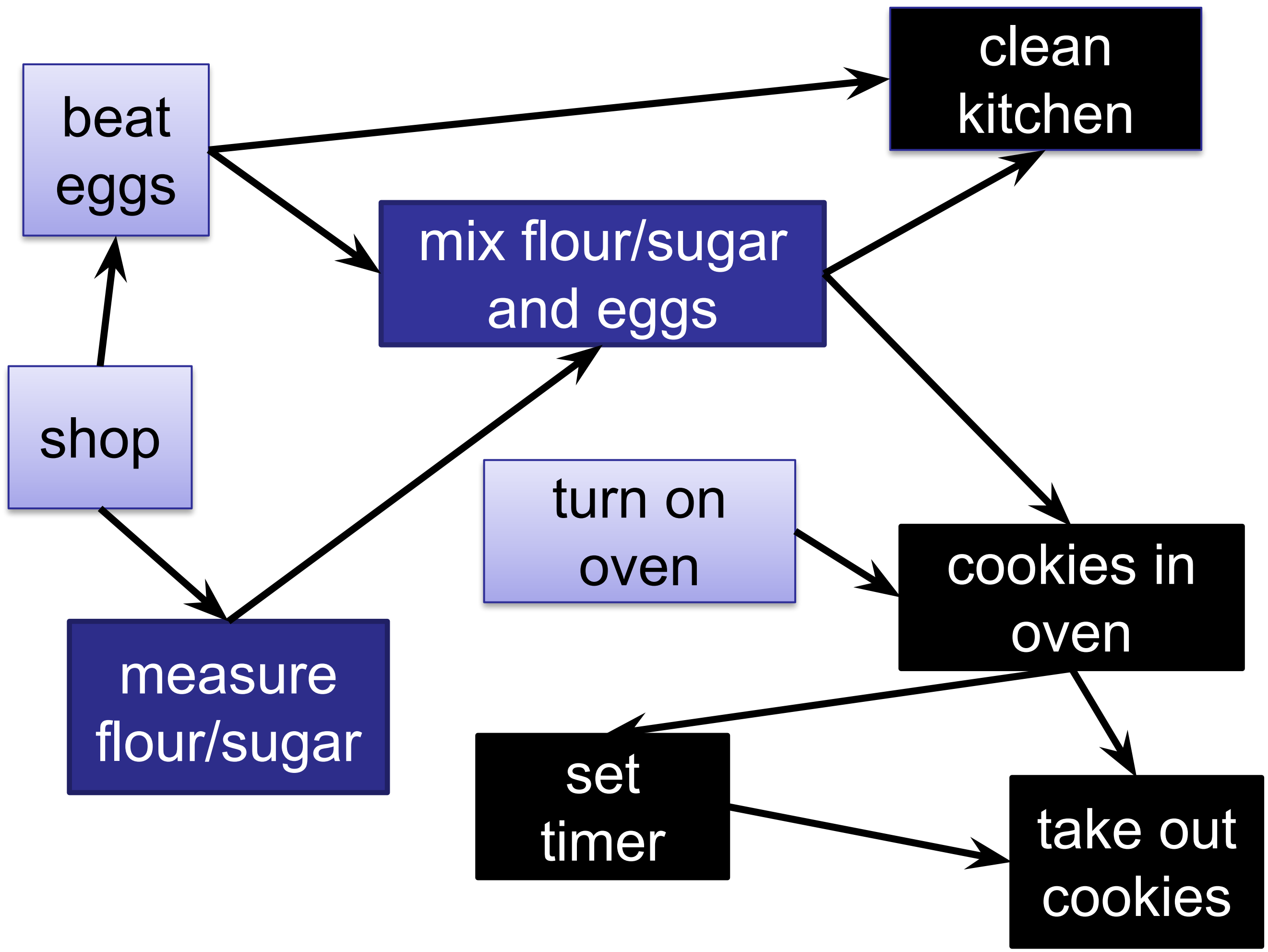
Depth-First Search

1. measure
2. mix
3. in oven
4. take out
5. set timer
6. clean



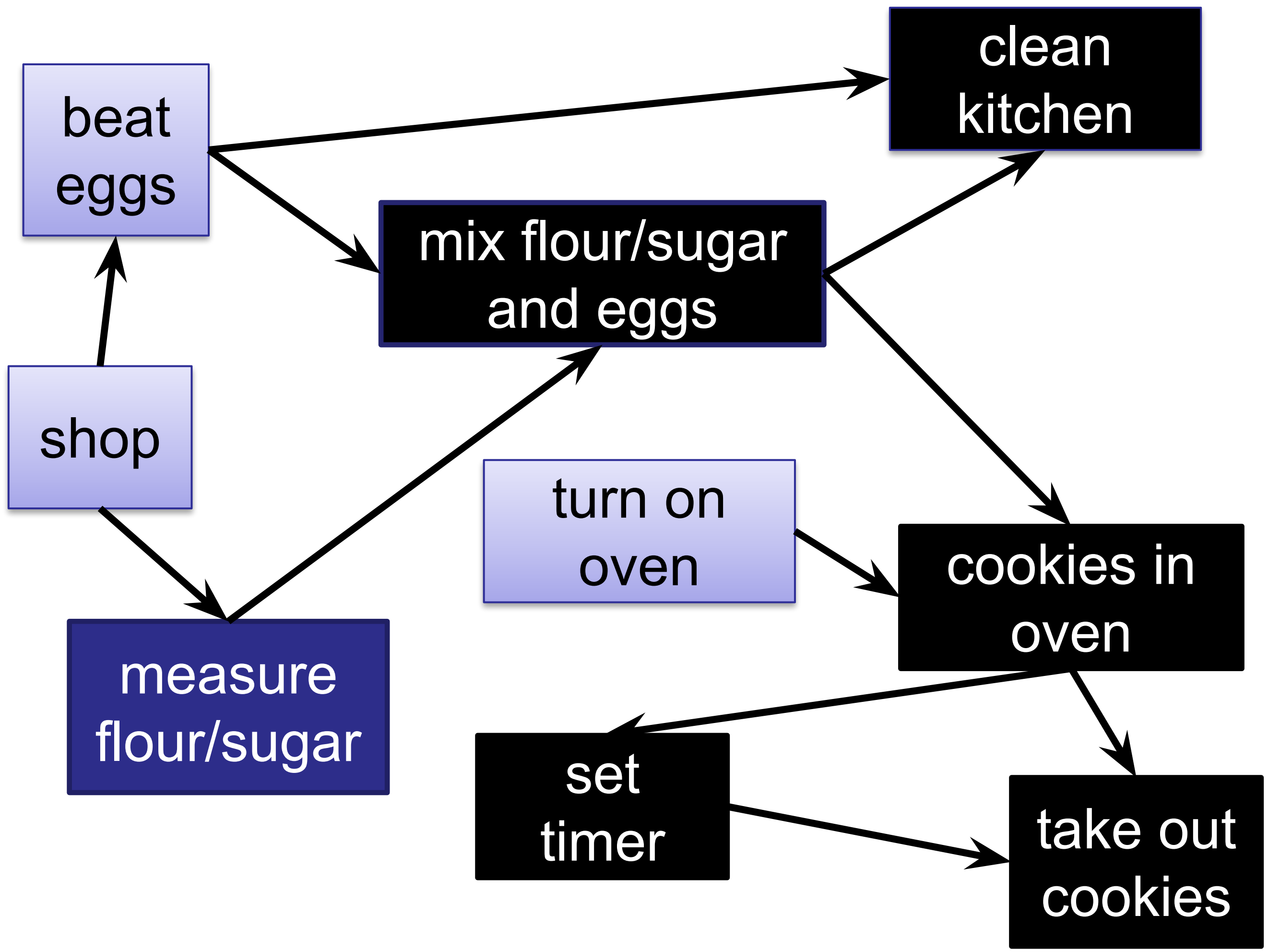
Depth-First Search

1. measure
2. mix
3. in oven
4. take out
5. set timer
6. clean



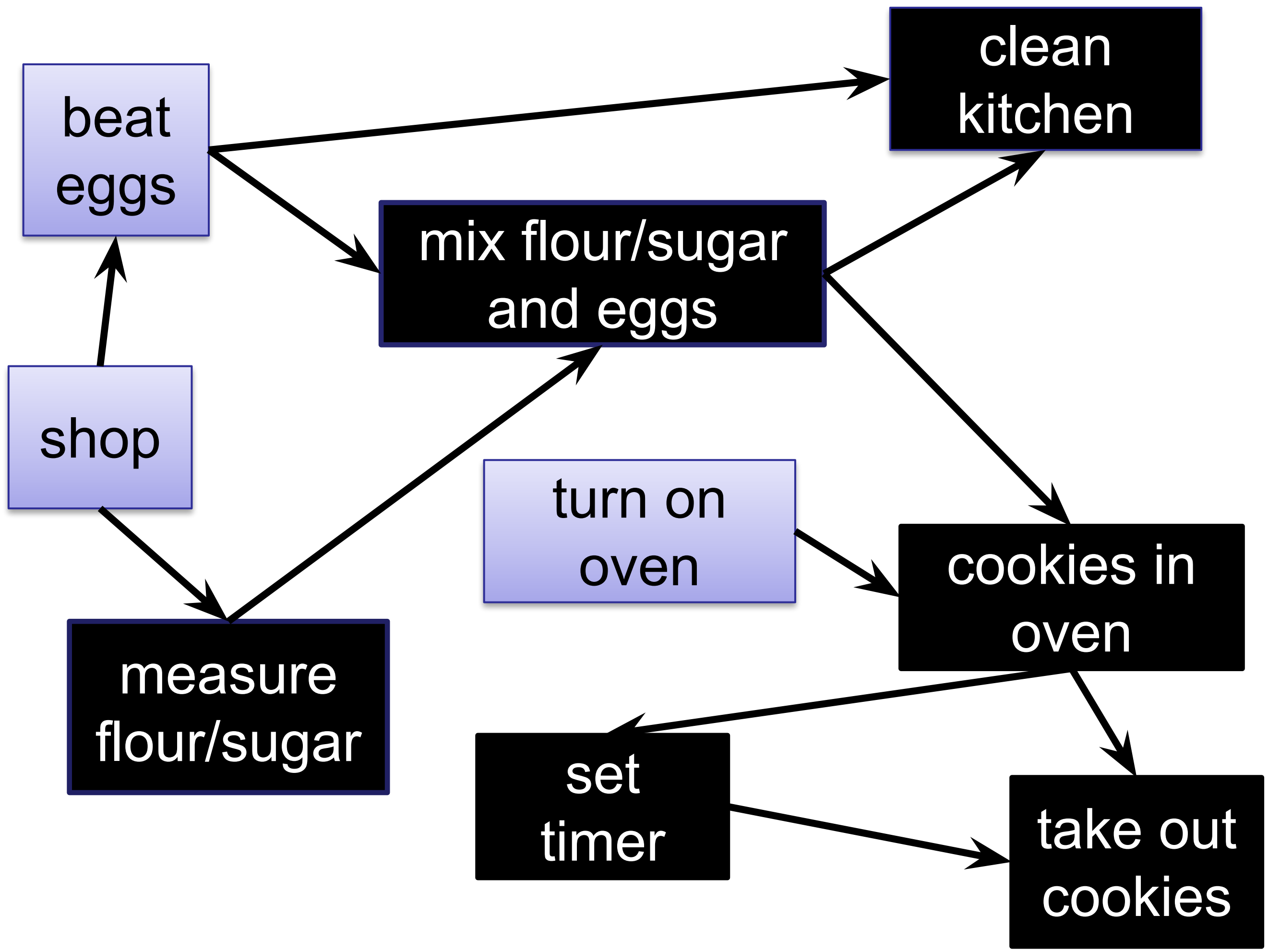
Depth-First Search

1. measure
2. mix
3. in oven
4. take out
5. set timer
6. clean



Depth-First Search

1. measure
2. mix
3. in oven
4. take out
5. set timer
6. clean



Searching a (Directed) Graph

Pre-Order Depth-First Search:

- Process each node when it is *first* visited.

Searching a (Directed) Graph

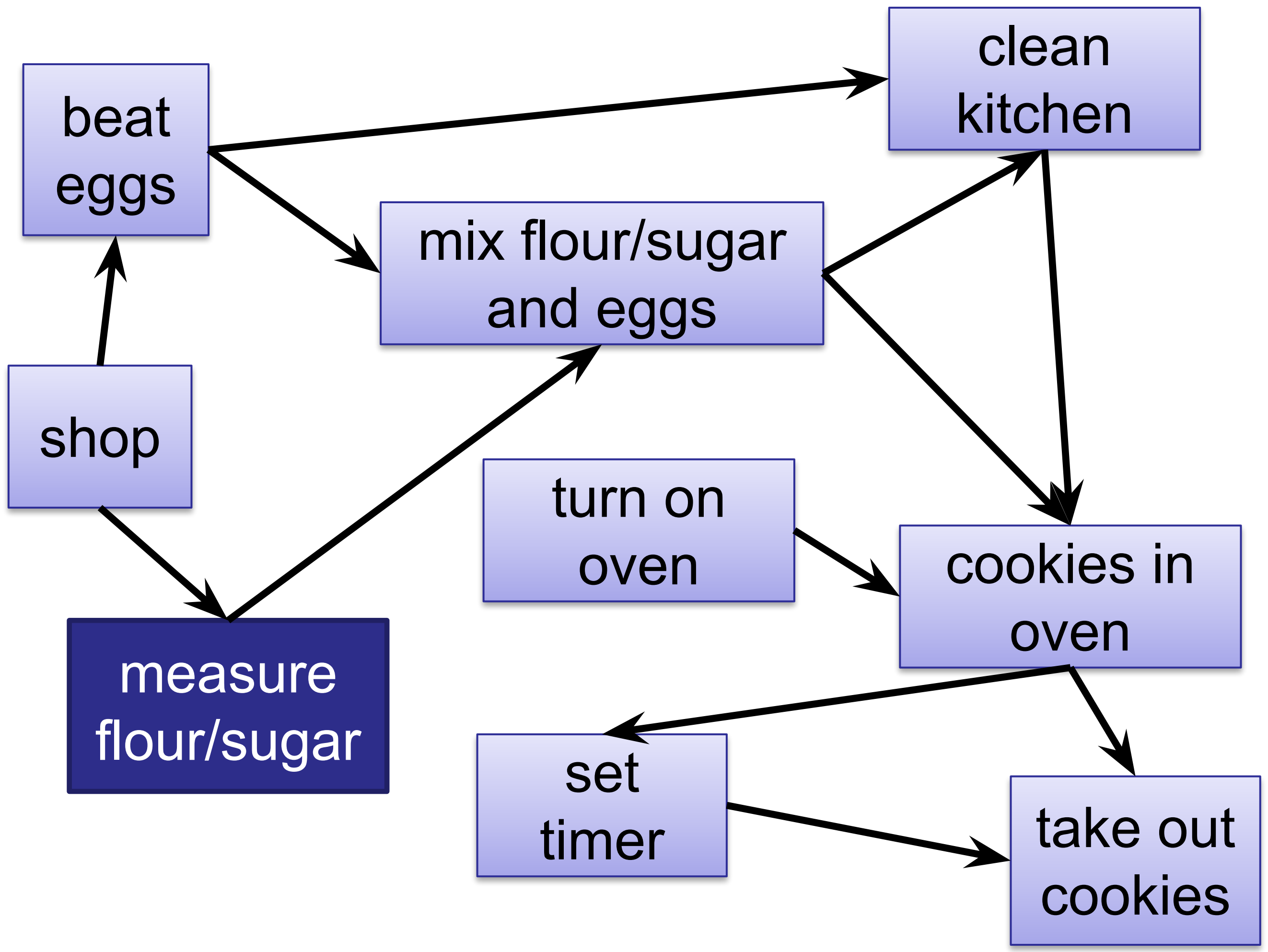
Pre-Order Depth-First Search:

– Process each node when it is *first* visited.

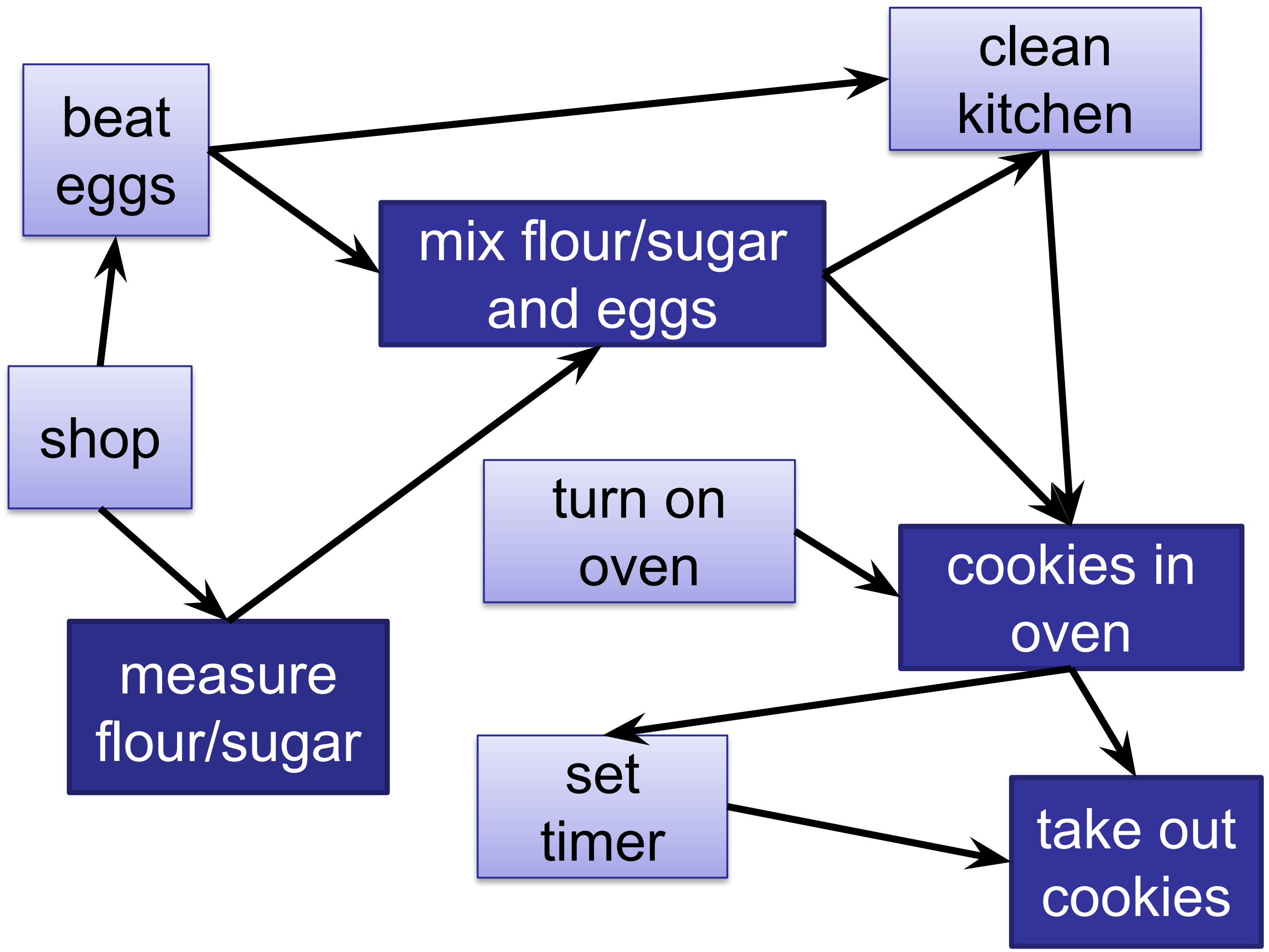
Post-Order Depth-First Search:

– Process each node when it is *last* visited.

Post-Order Depth-First Search

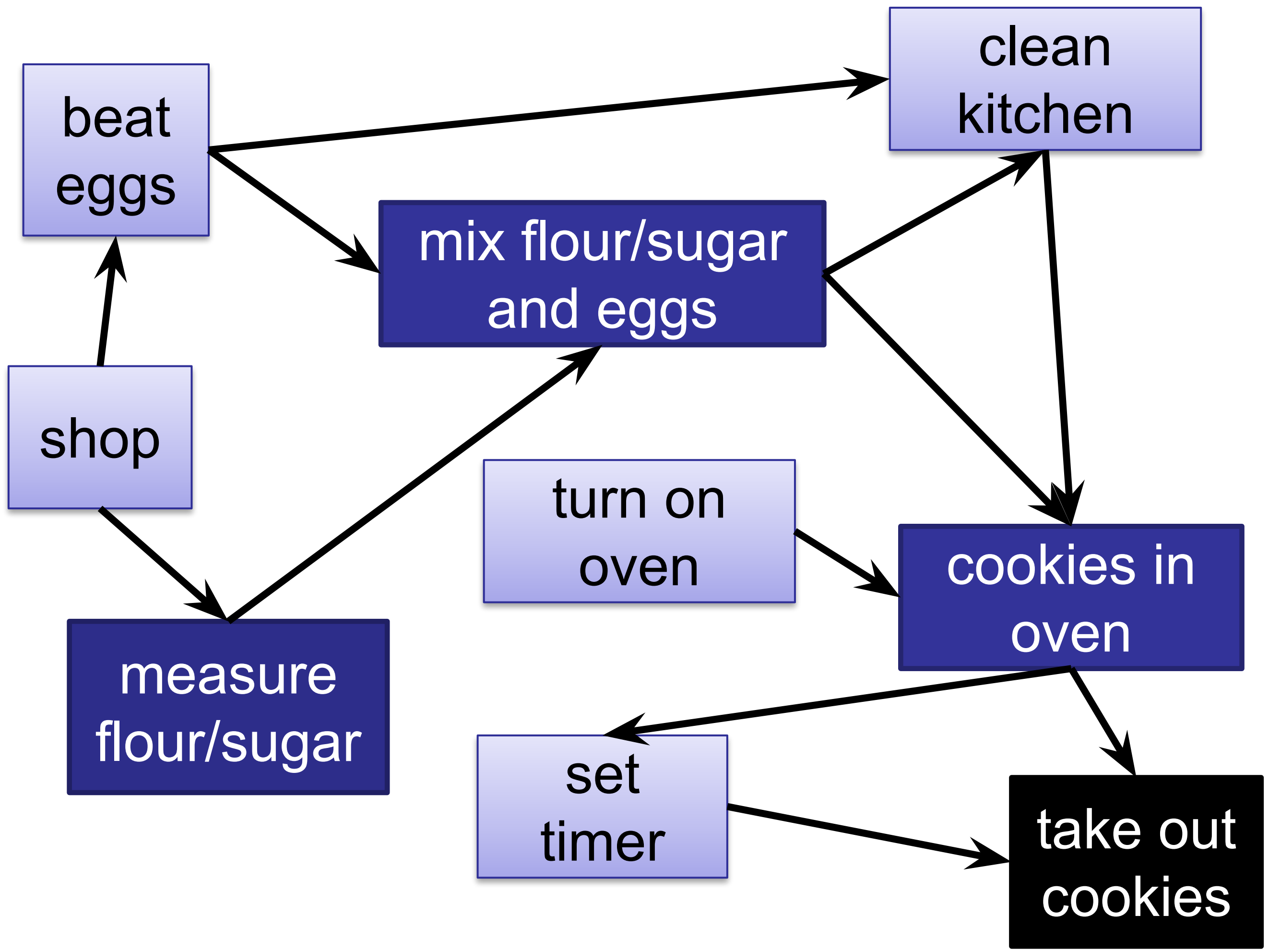


Post-Order Depth-First Search



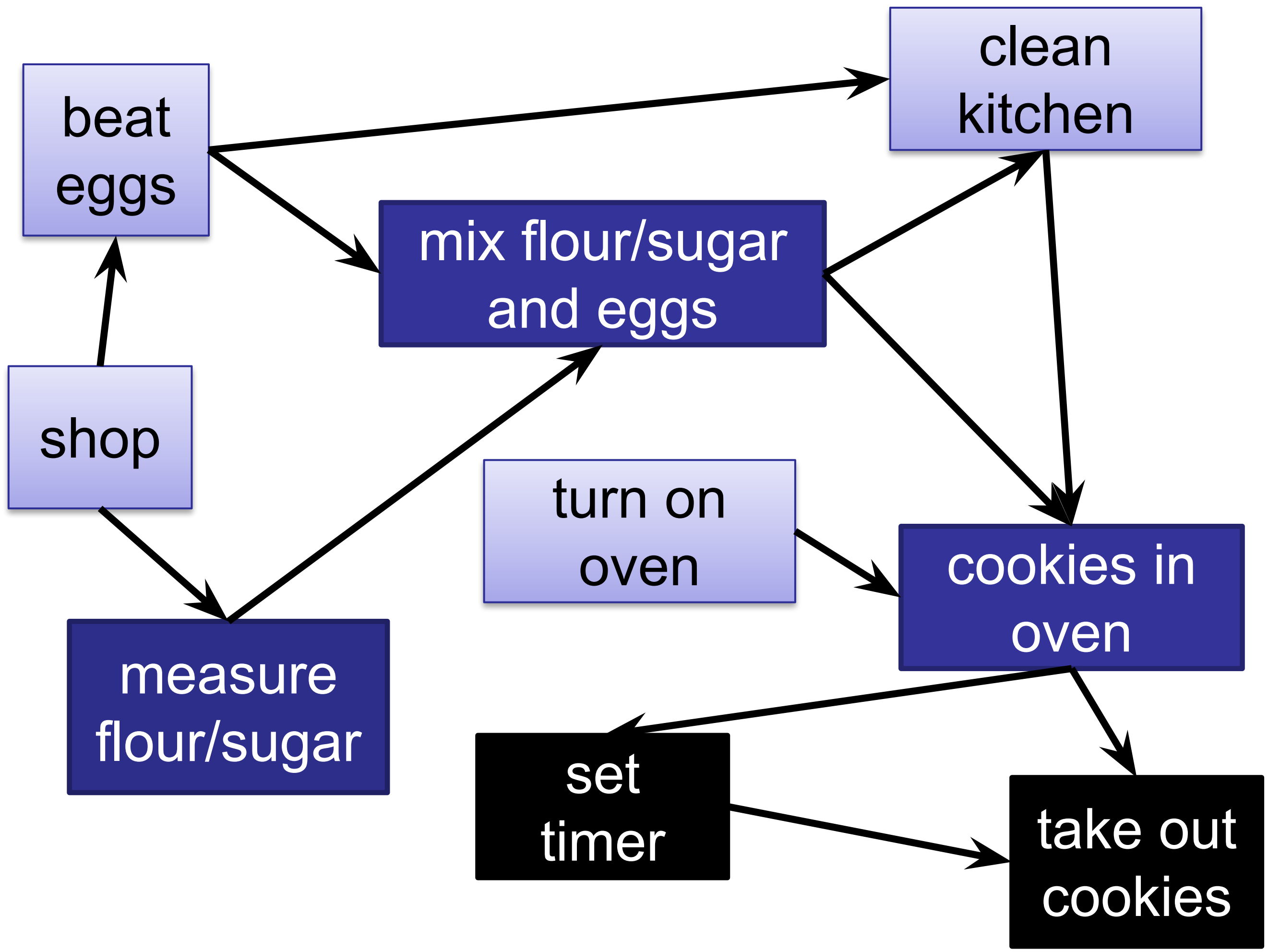
Post-Order Depth-First Search

1.
2.
3.
4.
5.
6.
7.
8.
9. take out



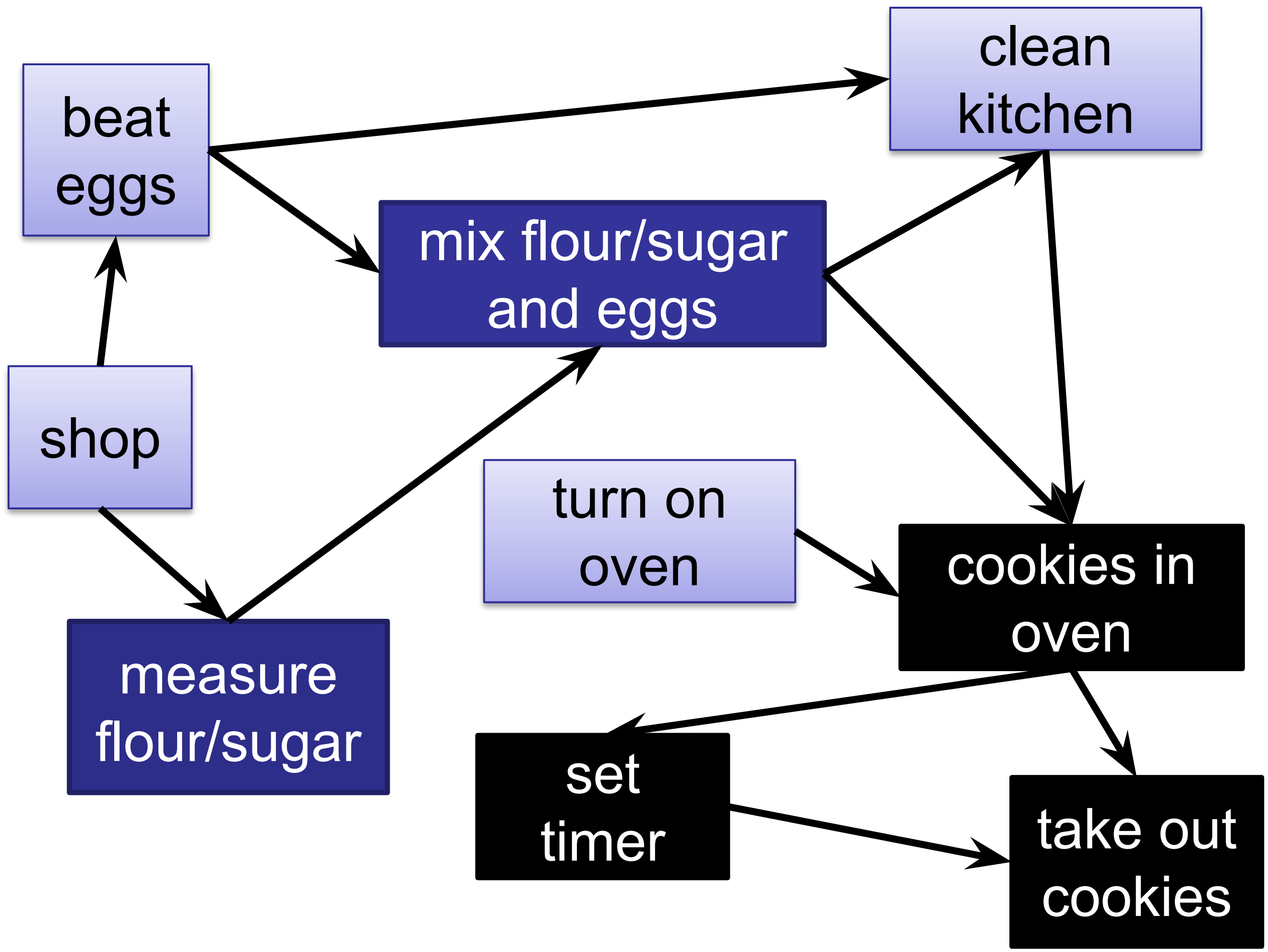
Post-Order Depth-First Search

1.
2.
3.
4.
5.
6.
7.
8. set timer
9. take out



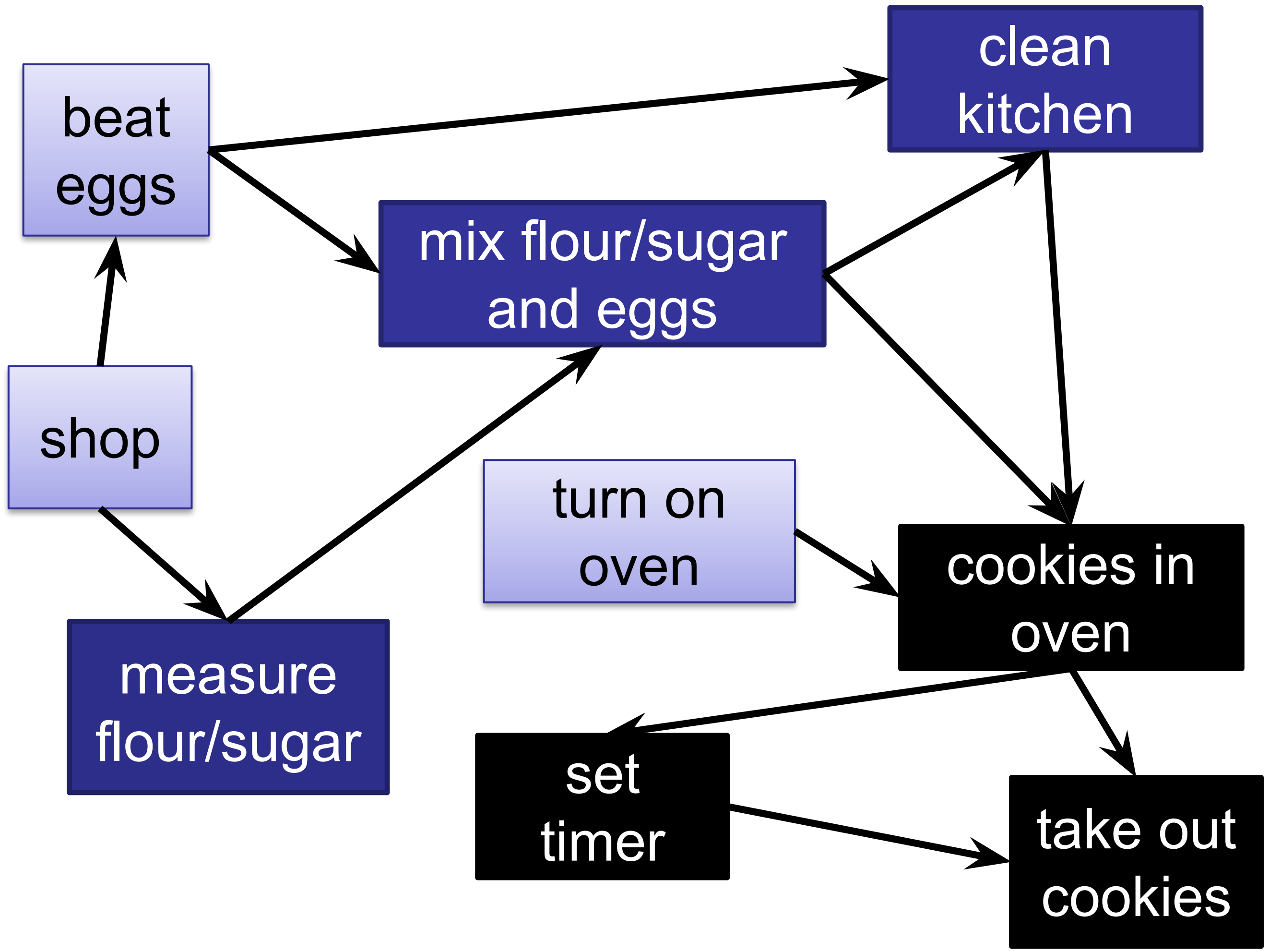
Post-Order Depth-First Search

1.
2.
3.
4.
5.
6.
7. in oven
8. set timer
9. take out



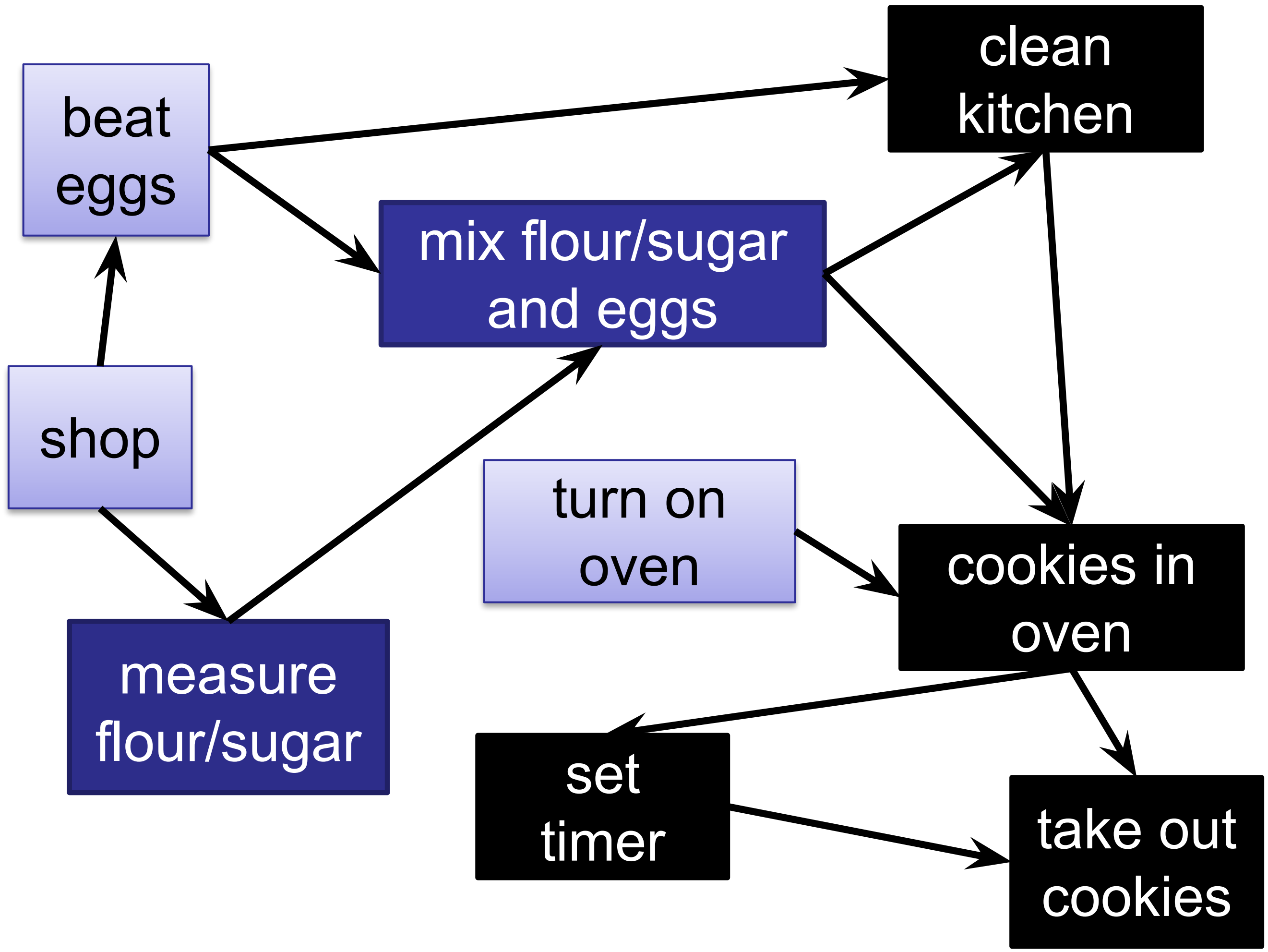
Post-Order Depth-First Search

1.
2.
3.
4.
5.
6.
7. in oven
8. set timer
9. take out



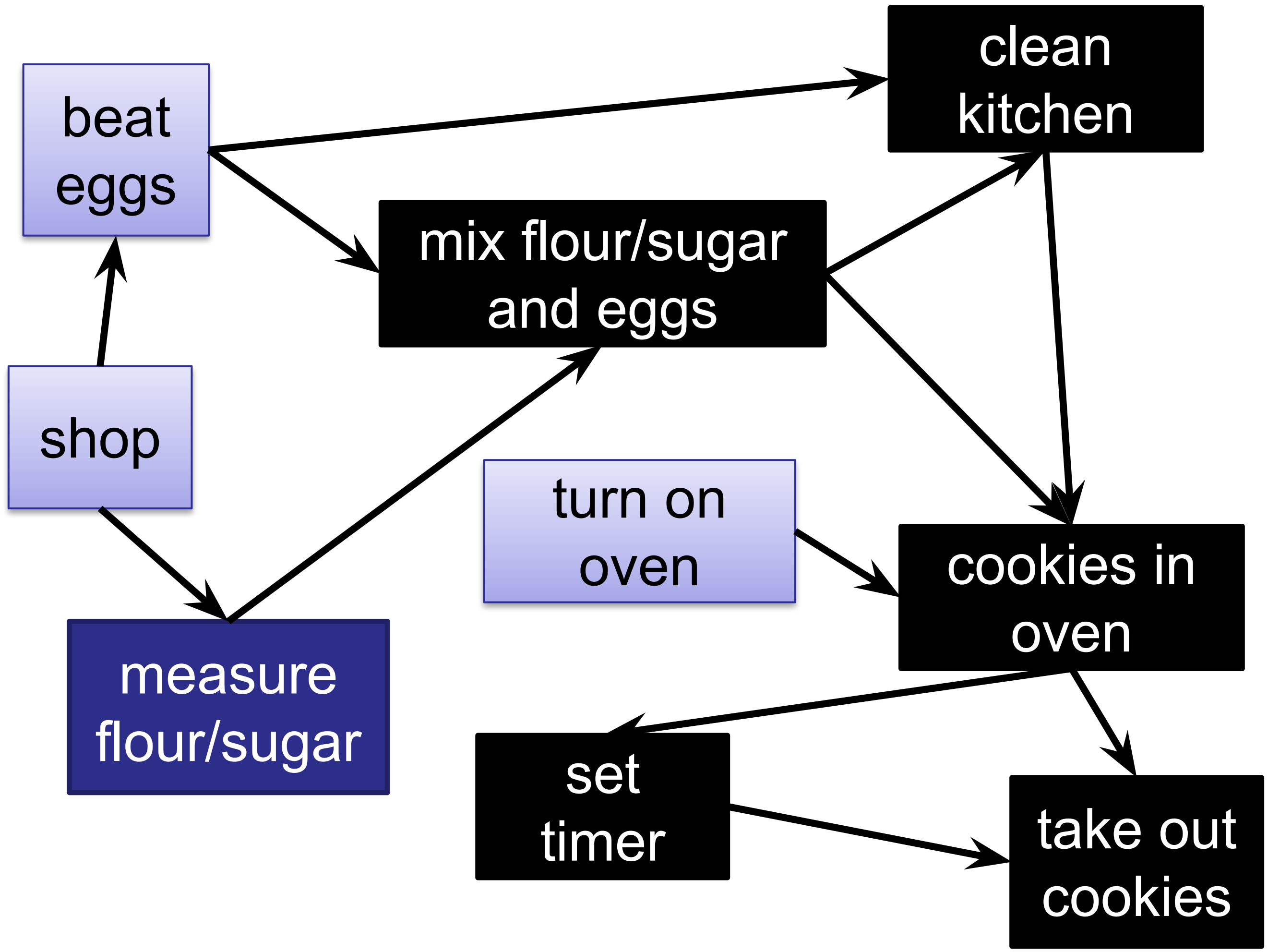
Post-Order Depth-First Search

1.
2.
3.
4.
5.
6. clean
7. in oven
8. set timer
9. take out



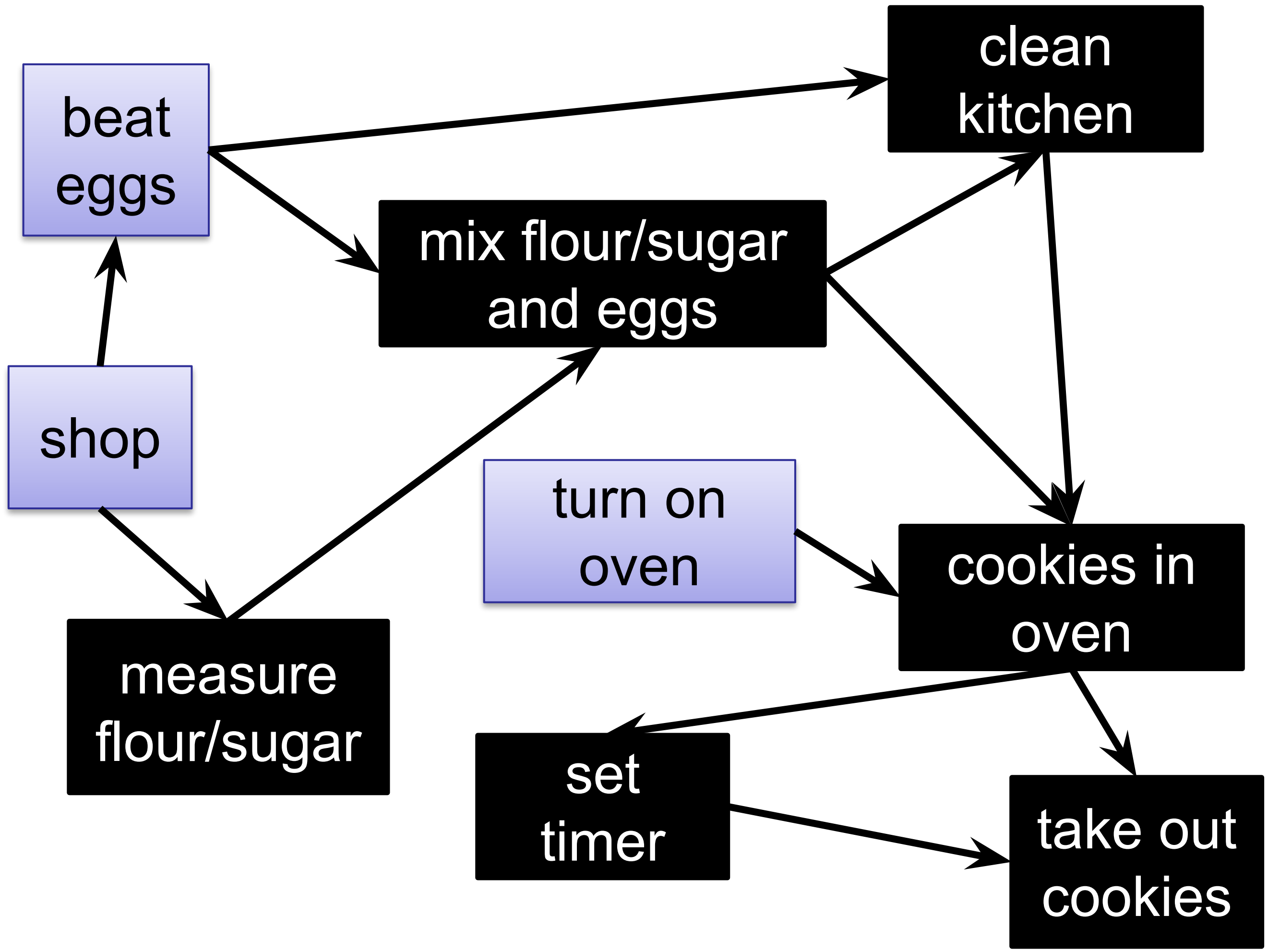
Post-Order Depth-First Search

1.
2.
3.
4.
5. mix
6. clean
7. in oven
8. set timer
9. take out



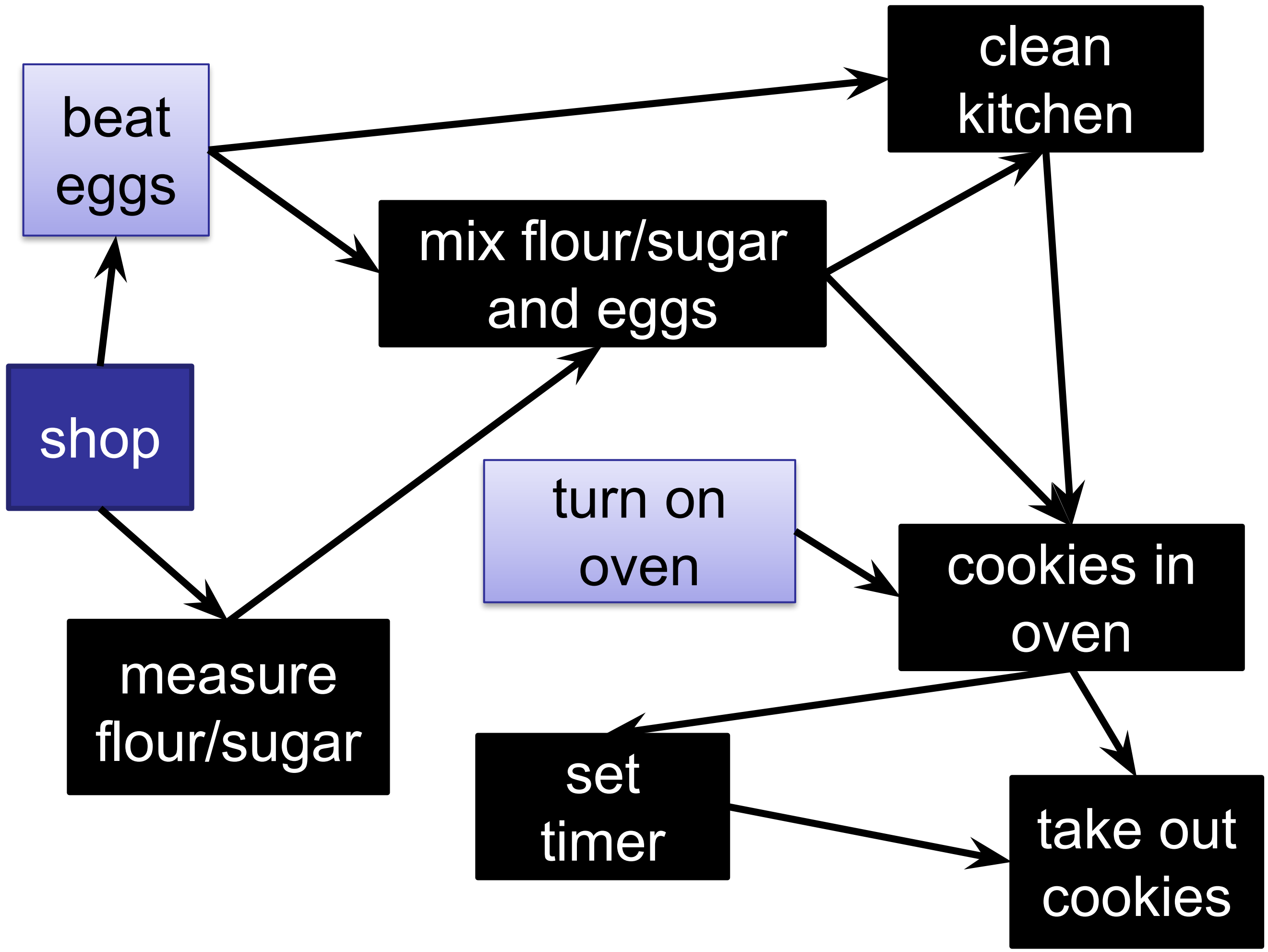
Post-Order Depth-First Search

1.
2.
3.
4. measure
5. mix
6. clean
7. in oven
8. set timer
9. take out



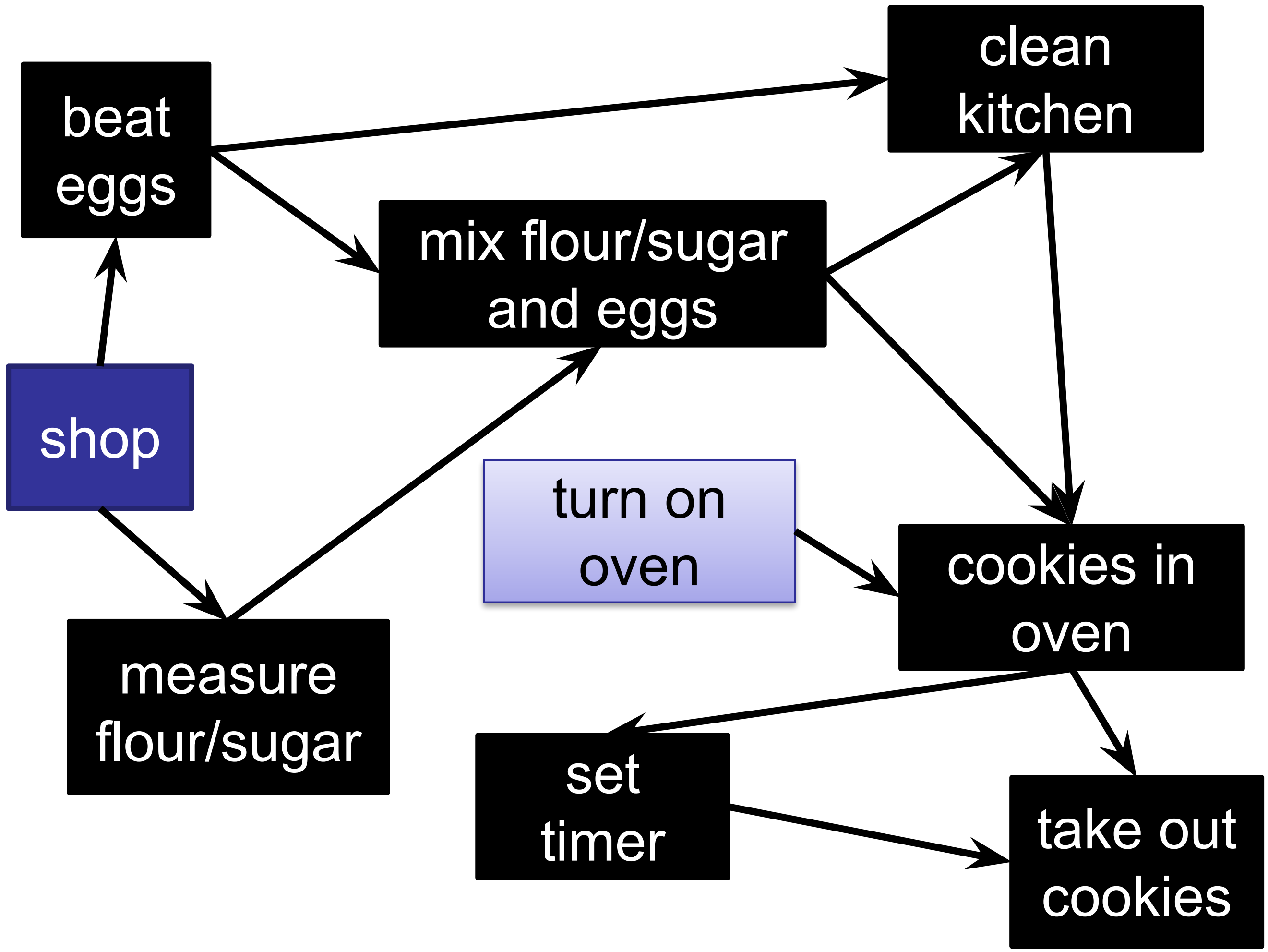
Post-Order Depth-First Search

1.
2.
3.
4. measure
5. mix
6. clean
7. in oven
8. set timer
9. take out



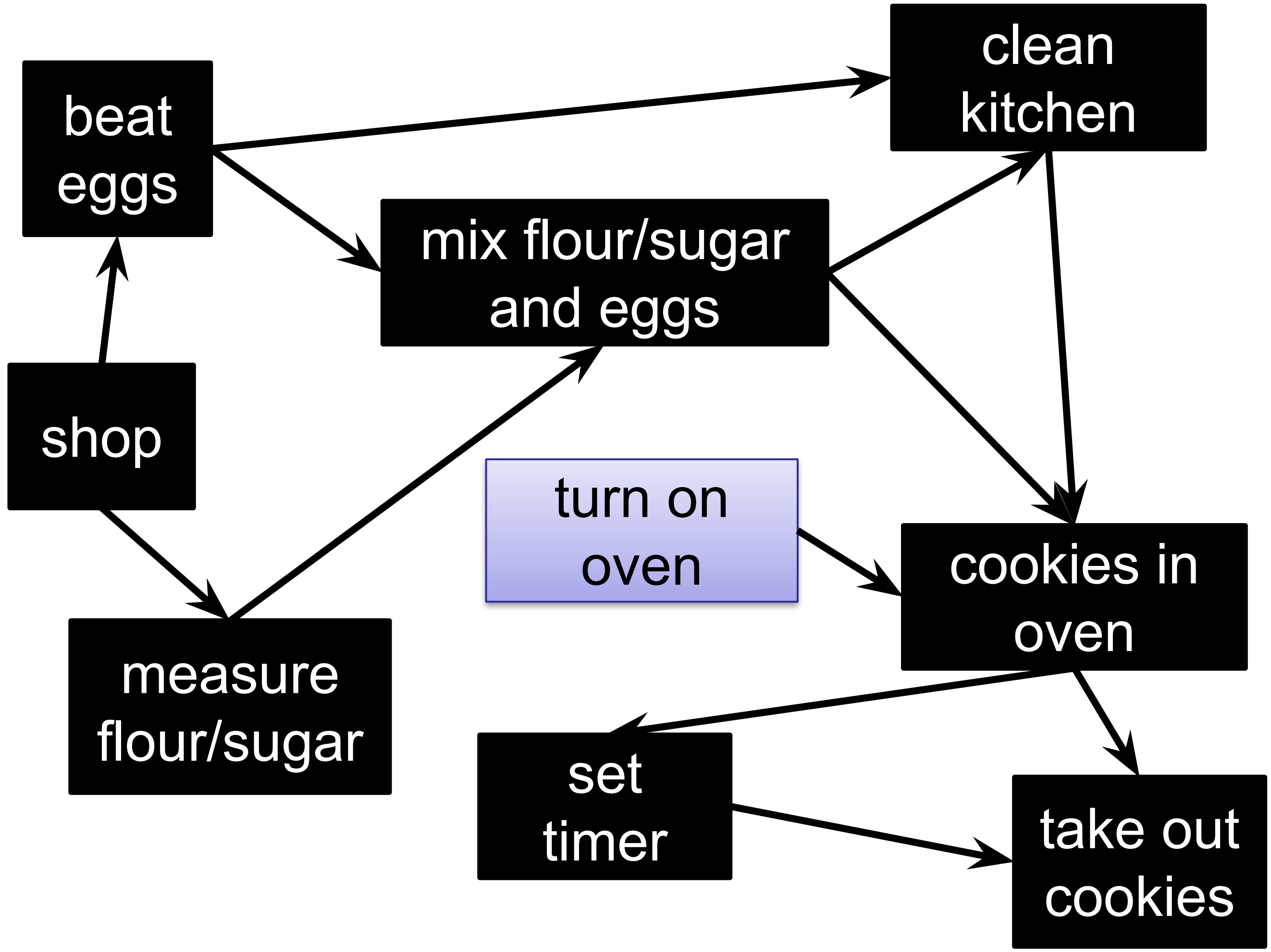
Post-Order Depth-First Search

1.
2.
3. beat
4. measure
5. mix
6. clean
7. in oven
8. set timer
9. take out



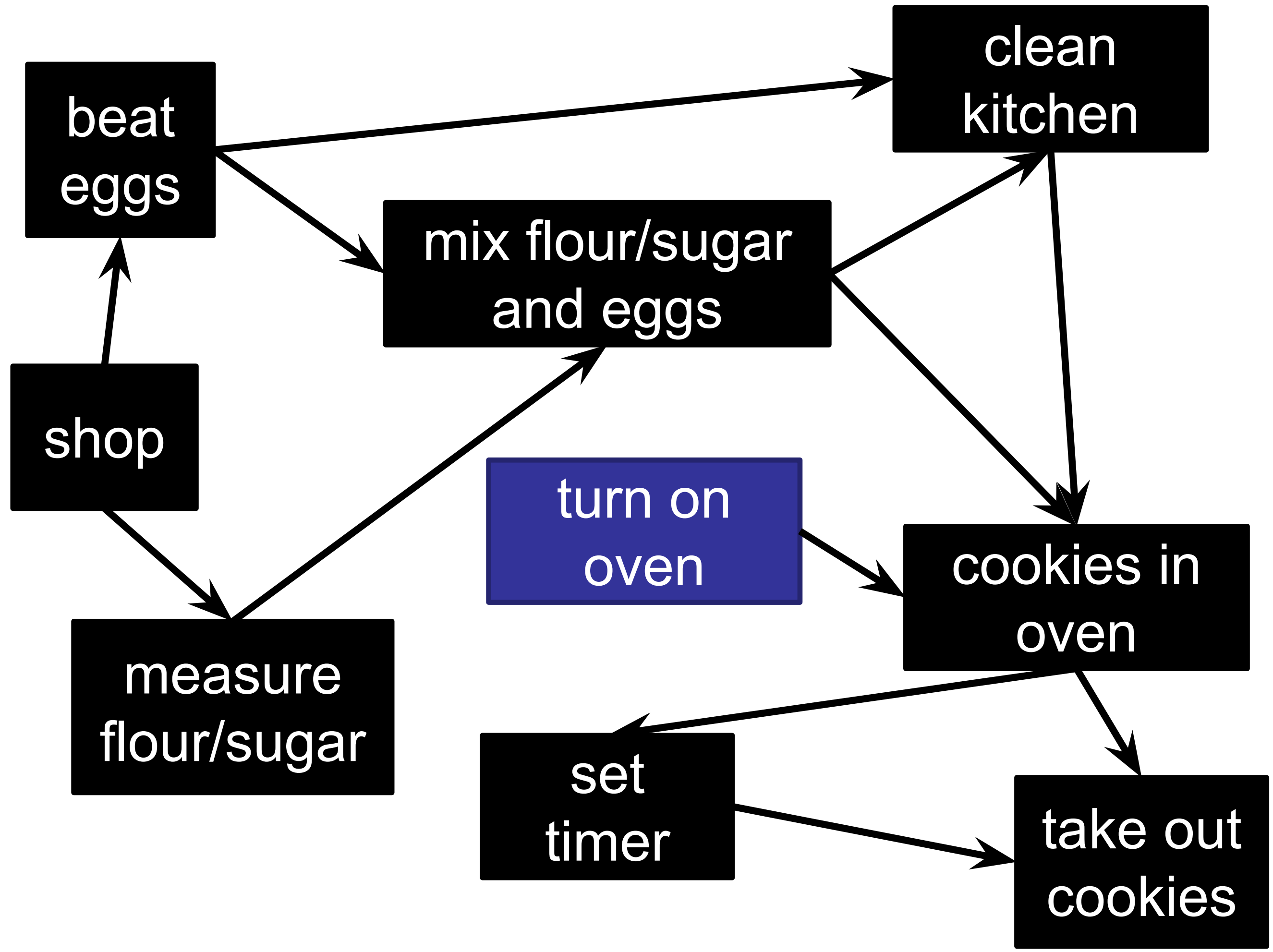
Post-Order Depth-First Search

1.
2. shop
3. beat
4. measure
5. mix
6. clean
7. in oven
8. set timer
9. take out



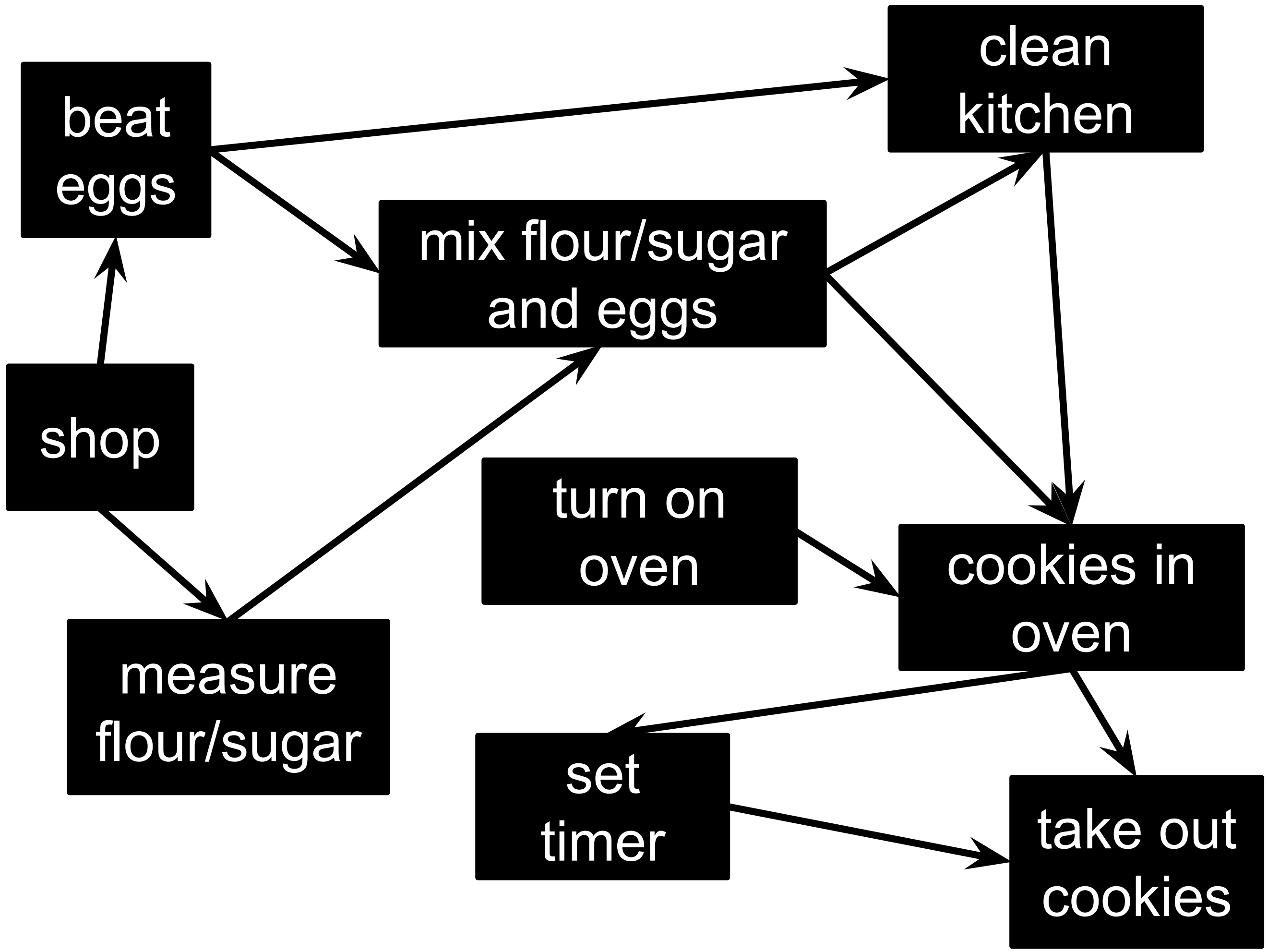
Post-Order Depth-First Search

1.
2. shop
3. beat
4. measure
5. mix
6. clean
7. in oven
8. set timer
9. take out



Post-Order Depth-First Search

1. on oven
2. shop
3. beat
4. measure
5. mix
6. clean
7. in oven
8. set timer
9. take out



Depth-First Search

```
toposort(Node[] nodeList, boolean[] visited, int startId){
    for (Integer v : nodeList[startId].nbrList) {
        if (!visited[v]){
            visited[v] = true;
            toposort(nodeList, visited, v);
             post operation here!
        }
    }
}
```

Depth-First Search

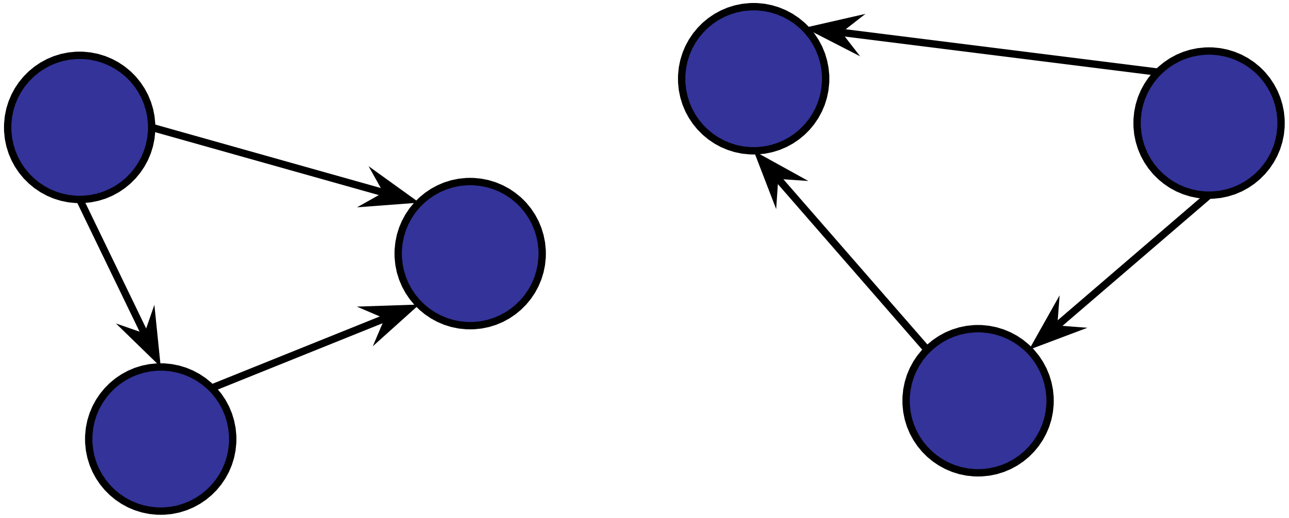
```
toposort(Node[] nodeList, boolean[] visited, int startId){
    for (Integer v : nodeList[startId].nbrList) {
        if (!visited[v]){
            visited[v] = true;
            toposort(nodeList, visited, v);
            schedule.prepend(startId);
        }
    }
}
```

Depth-First Search

```
toposort(Node[] nodeList, boolean[] visited, int startId){
    for (Integer v : nodeList[startId].nbrList) {
        if (!visited[v]){
            visited[v] = true;
            toposort(nodeList, visited, v);
            schedule.prepend(startId);
        }
    }
}
```

Does it toposort the graph?

What about this graph?



Depth-First Search

```
void toposort(
  Integer current_node,
  ArrayList<ArrayList<Integer>> adj_list,
  List<Integer> topo_list,
  boolean[] visited){

    for(Integer neighbour : adj_list[current_node]){
      if(visited[neighbour]) { continue; }
      visited[neighbour] = true;
      toposort(neighbour, adj_list, topo_list, visited);
    }

    topo_list.prepend(current_node);
}
```

Depth-First Search

```
List<Integer> toposort_all(ArrayList<ArrayList<Integer>> adj_list){
  // initially all false
  boolean[] visited = new boolean[adj_list.size()]

  // topo list
  List<Integer> topo_list;

  for(int start = 0; start < adj_list.size(); ++start){
    if(visited[start]) { continue; }
    toposort(start, adj_list, topo_list, visited);
  }

  return topo_list;

}
```

Topological Sort

What is the time complexity of topological sort?

Topological Sort

What is the time complexity of topological sort?

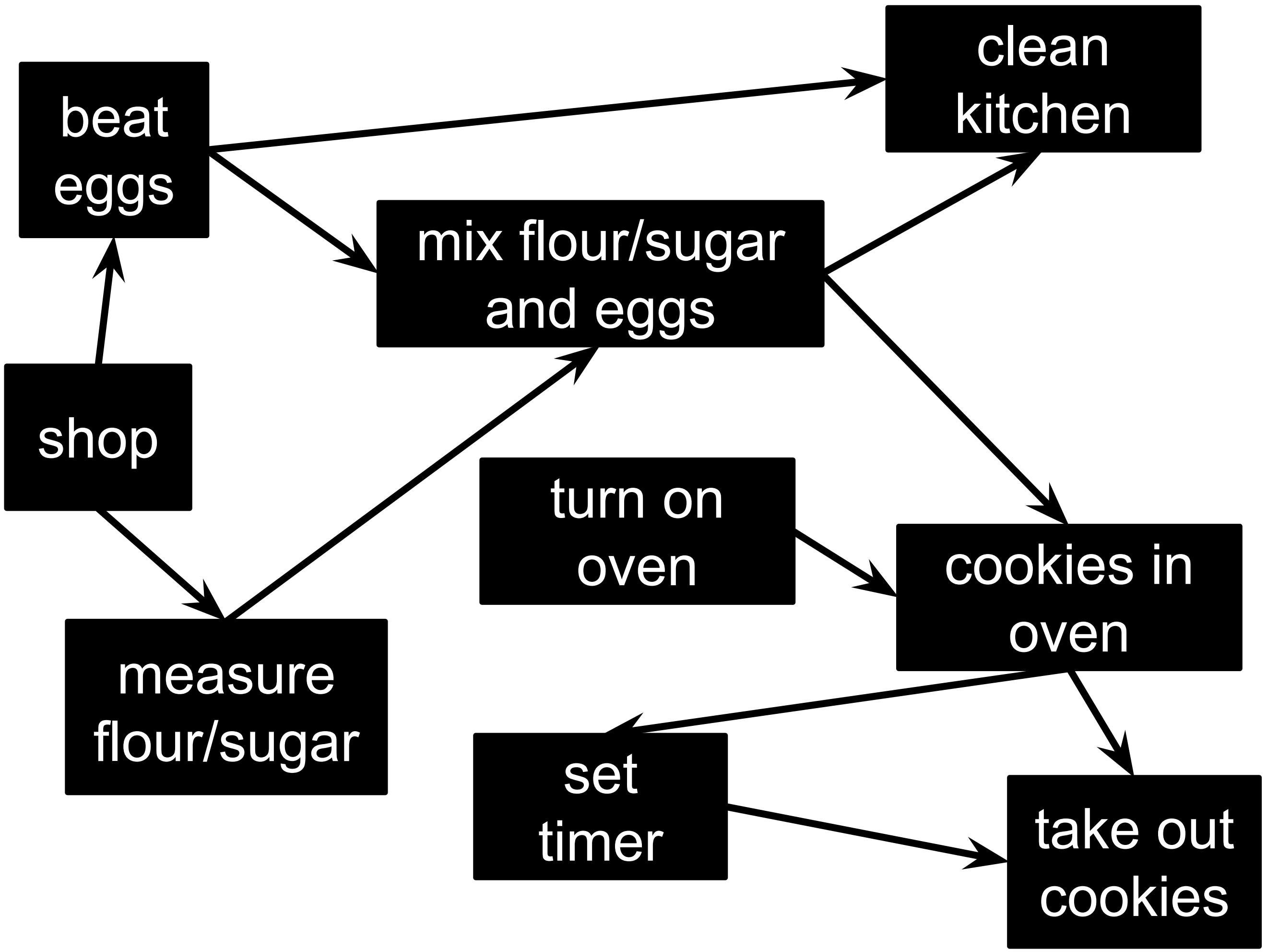
DFS: O(V+E)

Is a topological ordering unique?

1. Yes
2. No
3. Only on Thursdays.

Post-Order Depth-First Search

1. **on oven**
2. **shop**
3. beat
4. measure
5. mix
6. **clean**
7. in oven
8. **set timer**
9. take out



Topological Sort

Input:

- Directed Acyclic Graph (DAG)

Output:

- Total ordering of nodes, where all edges point forwards.

Algorithm:

- Post-order Depth-First Search
- O(V + E) time complexity

Topological Sort

Alternative algorithm:

Input: directed graph G

Repeat:

- S = all nodes in G that have *no* incoming edges.
- Add nodes in S to the topo-order
- Remove all edges adjacent to nodes in S
- Remove nodes in S from the graph

Time:

- O(V + E) time complexity

Topological Sort

But how do we tell if the directed graph is cyclic or not?

Some other DFS-able problems

1. How to tell if a graph is cyclic?

Some other DFS-able problems

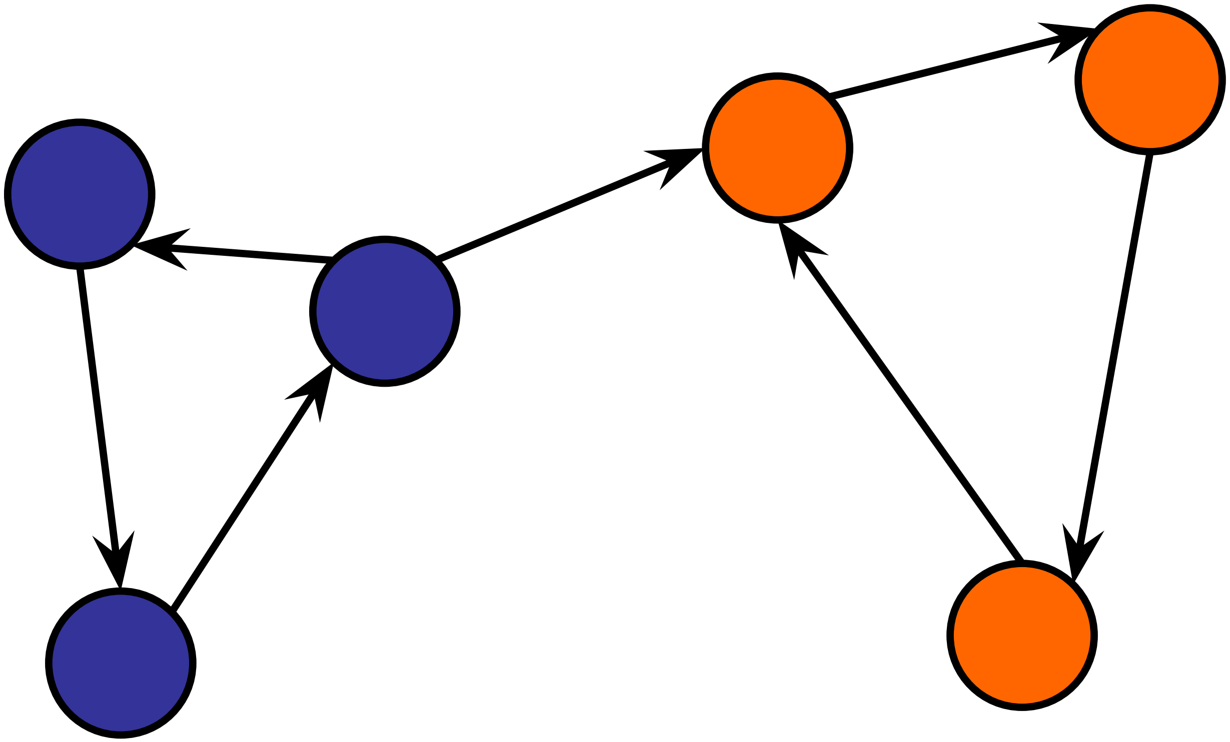
1. How to tell if a graph is cyclic?
2. How to find strongly connected components?

Connected Components

Strongly connected component

For every vertex v and w:

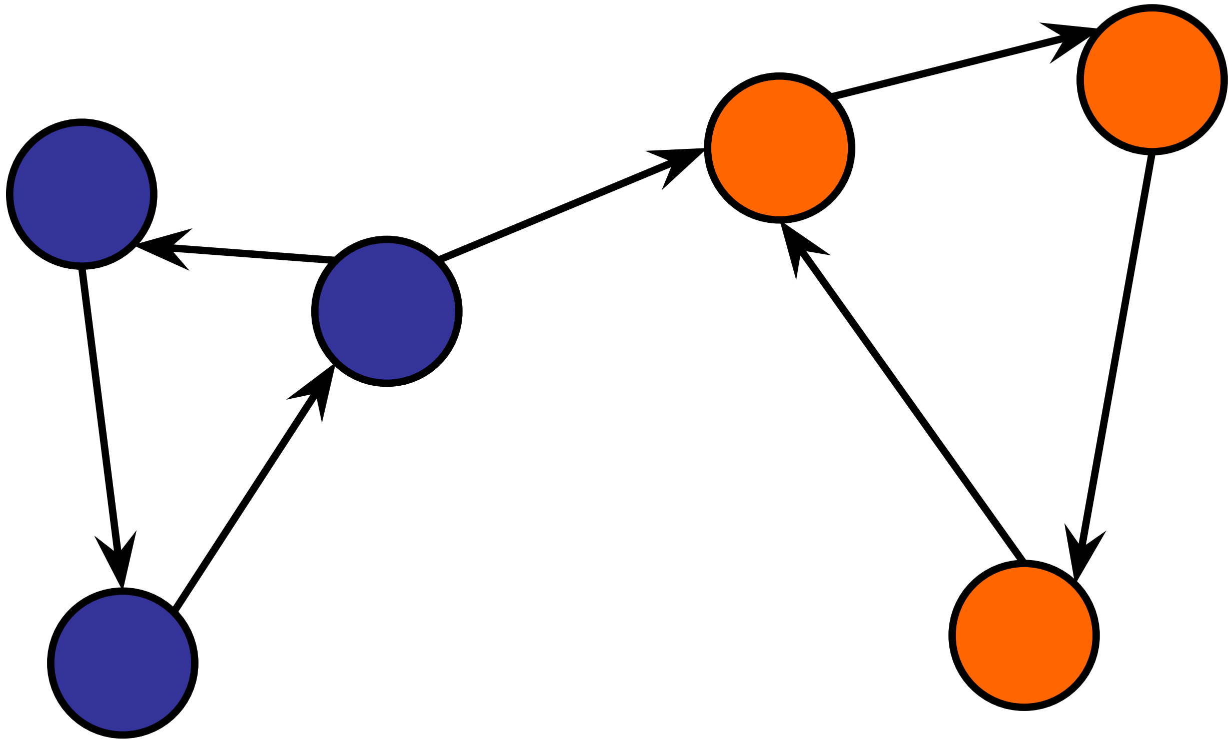
- There is a path from v to w.
- There is a path from w to v.



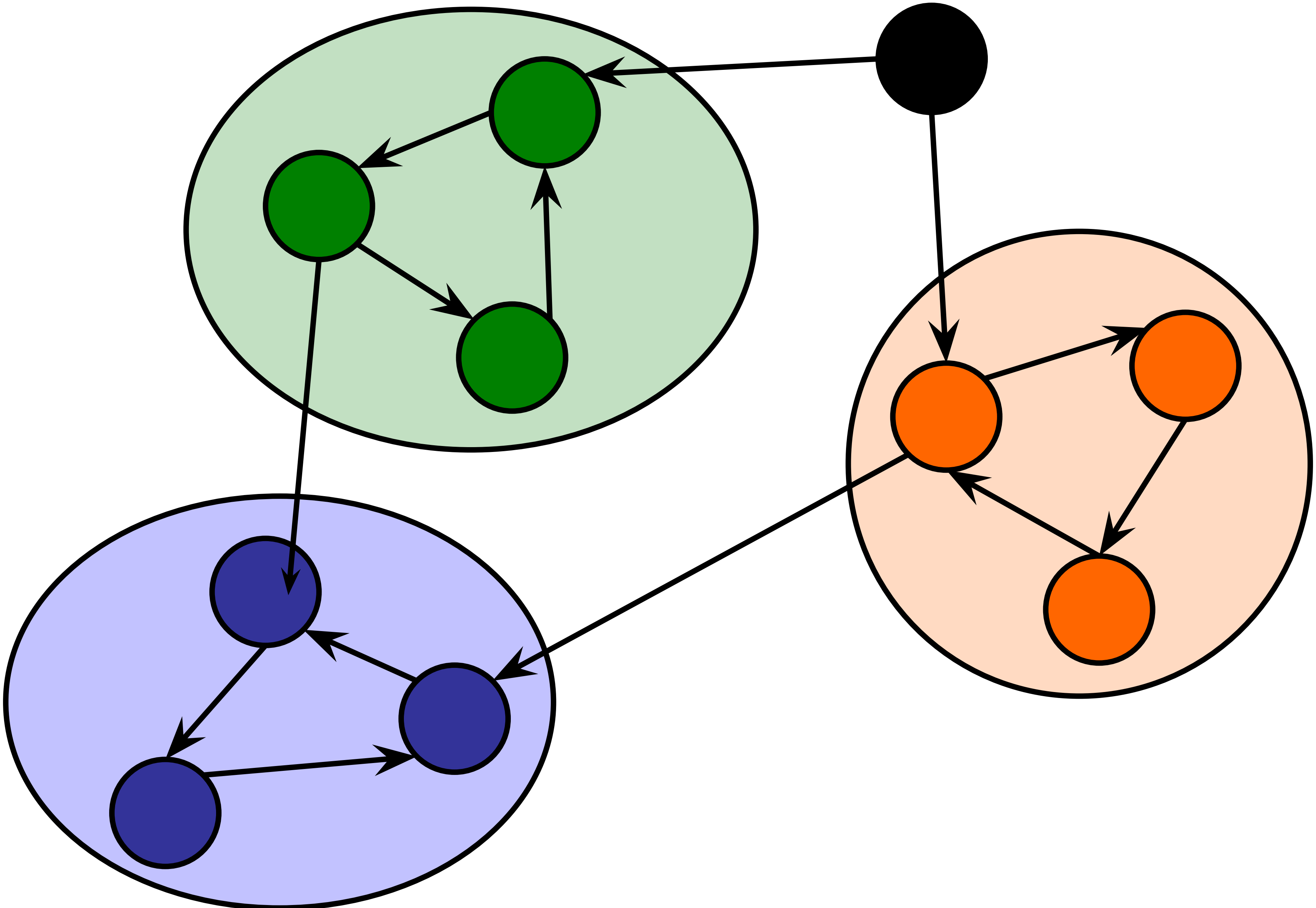
Connected Components

Strongly connected component

Two nodes v, w in a SCC are reachable to/from each other.

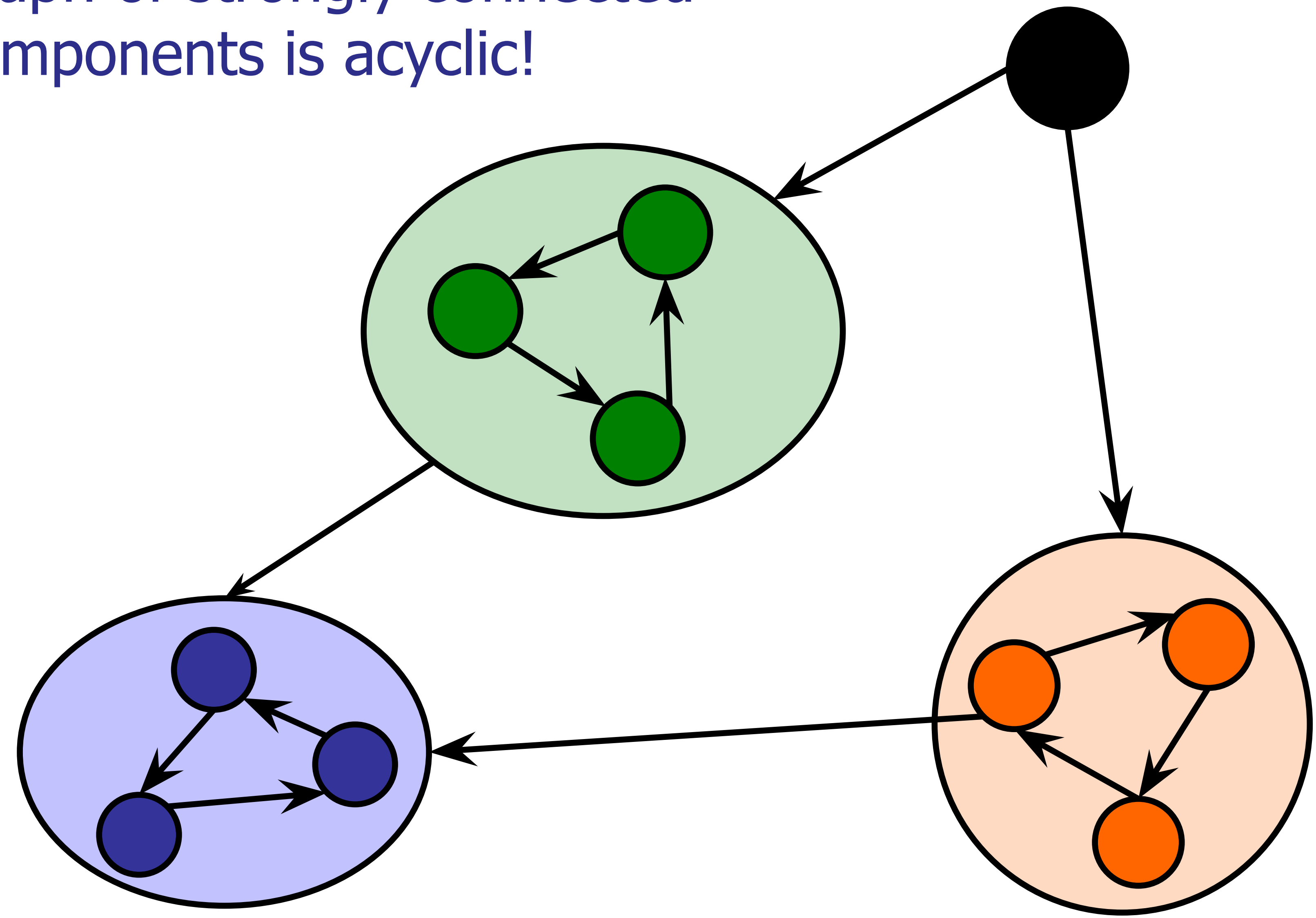


Connected Components



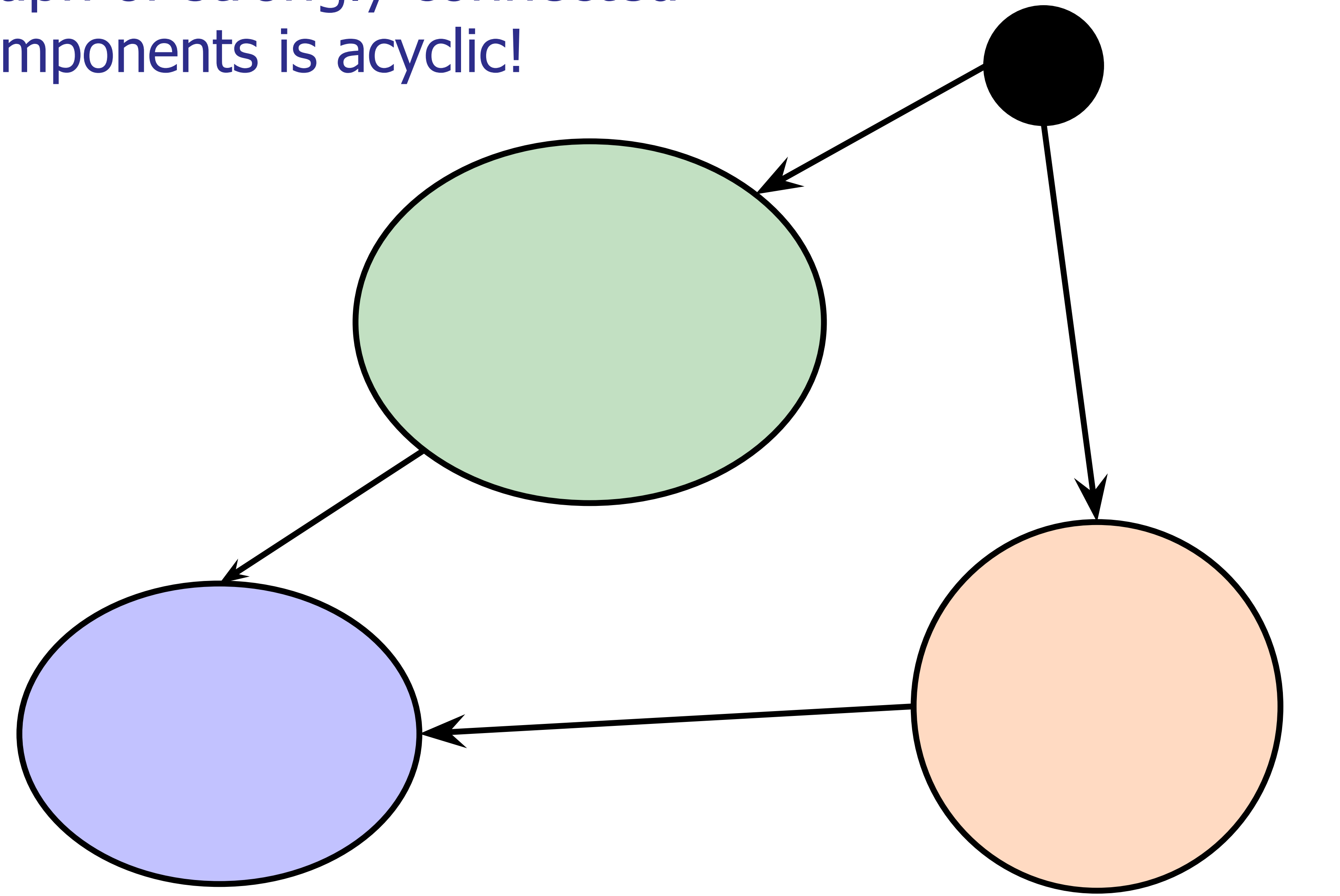
Connected Components

Graph of strongly connected components is acyclic!



Connected Components

Graph of strongly connected components is acyclic!



Strongly Connected Components

Input:

- Directed Graph

Output:

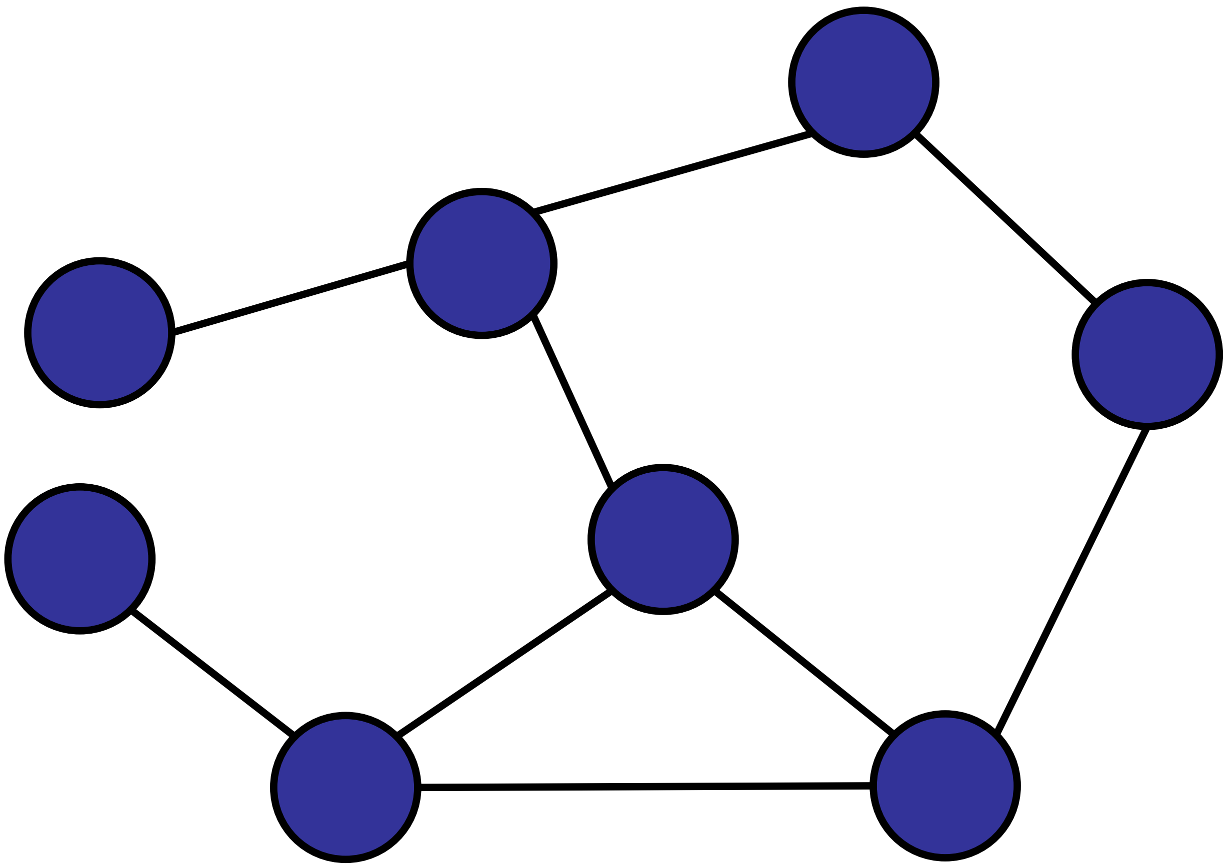
- A labelling of the nodes to denote which component it belongs to.
- If we consider the graph based on the components, it is acyclic.

Some other DFS-able problems

1. How to tell if a graph is cyclic?
2. How to find strongly connected components?
3. How do we find articulation points?

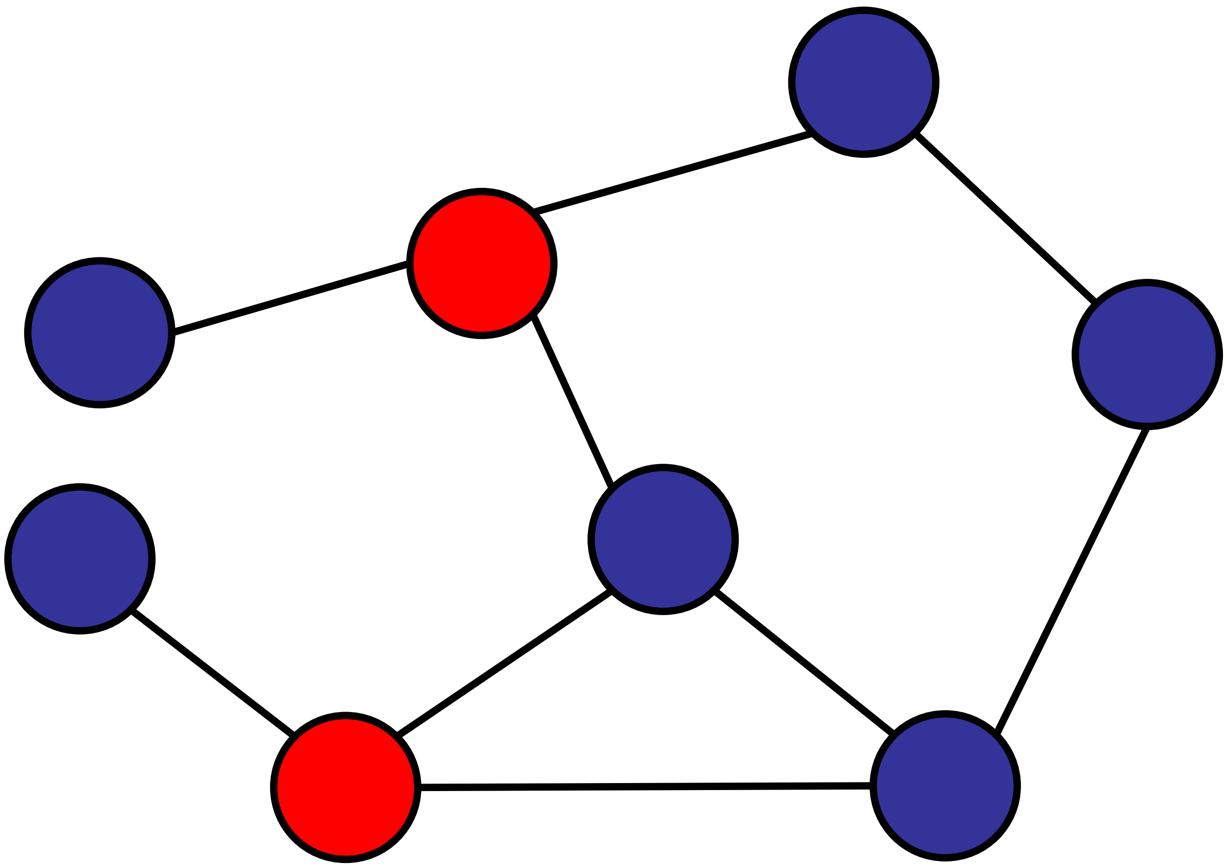
Articulation Points

A node is an **articulation point** if removing it disconnects the graph



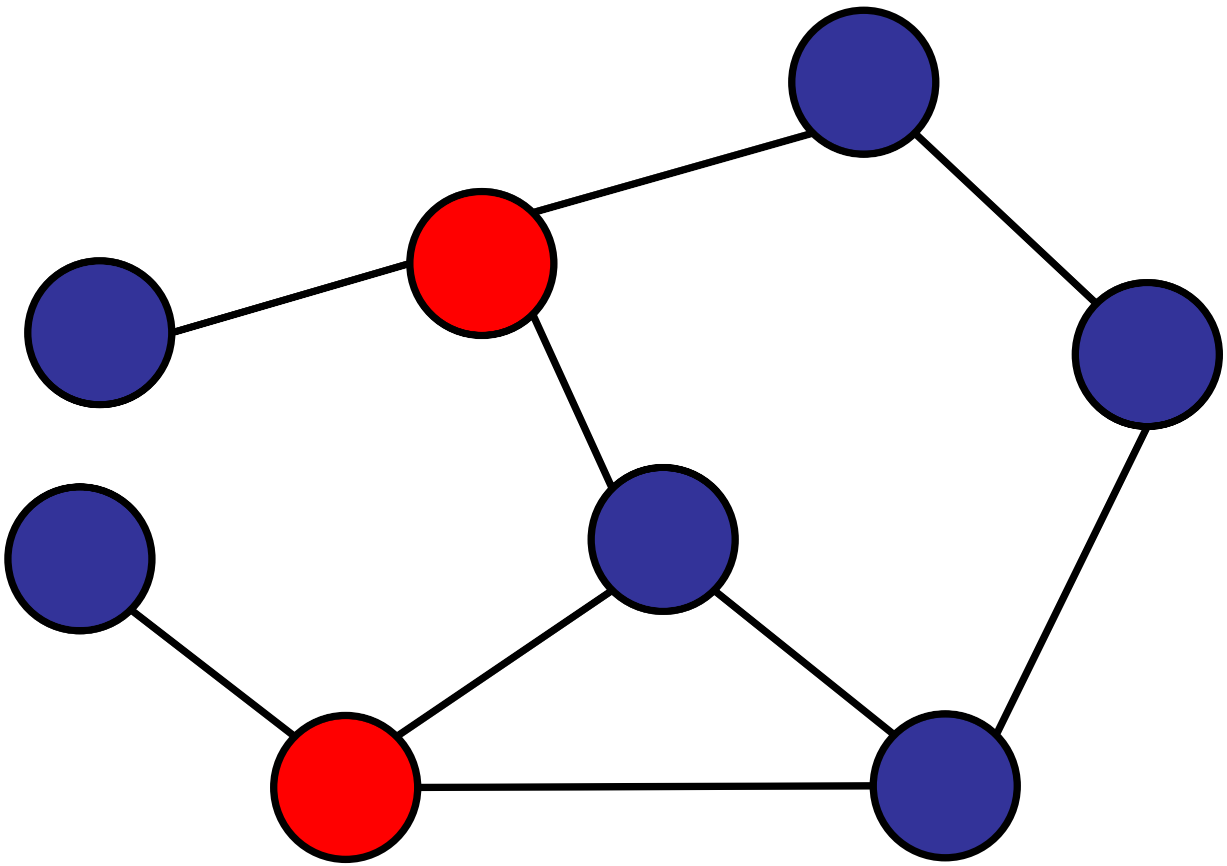
Articulation Points

A node is an **articulation point** if removing it disconnects the graph



Articulation Points

Goal: Given an undirected graph, find all articulation points.



DFS: Template

Today:

One DFS to rule them all

- Bridge edges / Articulation Points / Cycle Detection

DFS: Template

Idea: What if we marked each node we DFS with 2 things:

1. The time we visited it.
2. The lowest time we can reach from our neighbours.

DFS: Template

Clarification:

Idea:

1. Mark each node with the time we visited it.

2. If we ever visit a neighbour whose lowest time is not set, we will consider taking their time as our lowest time.

3. Setting lowest time is only done after our recursion as a post-traversal operation.

Cycle Finding

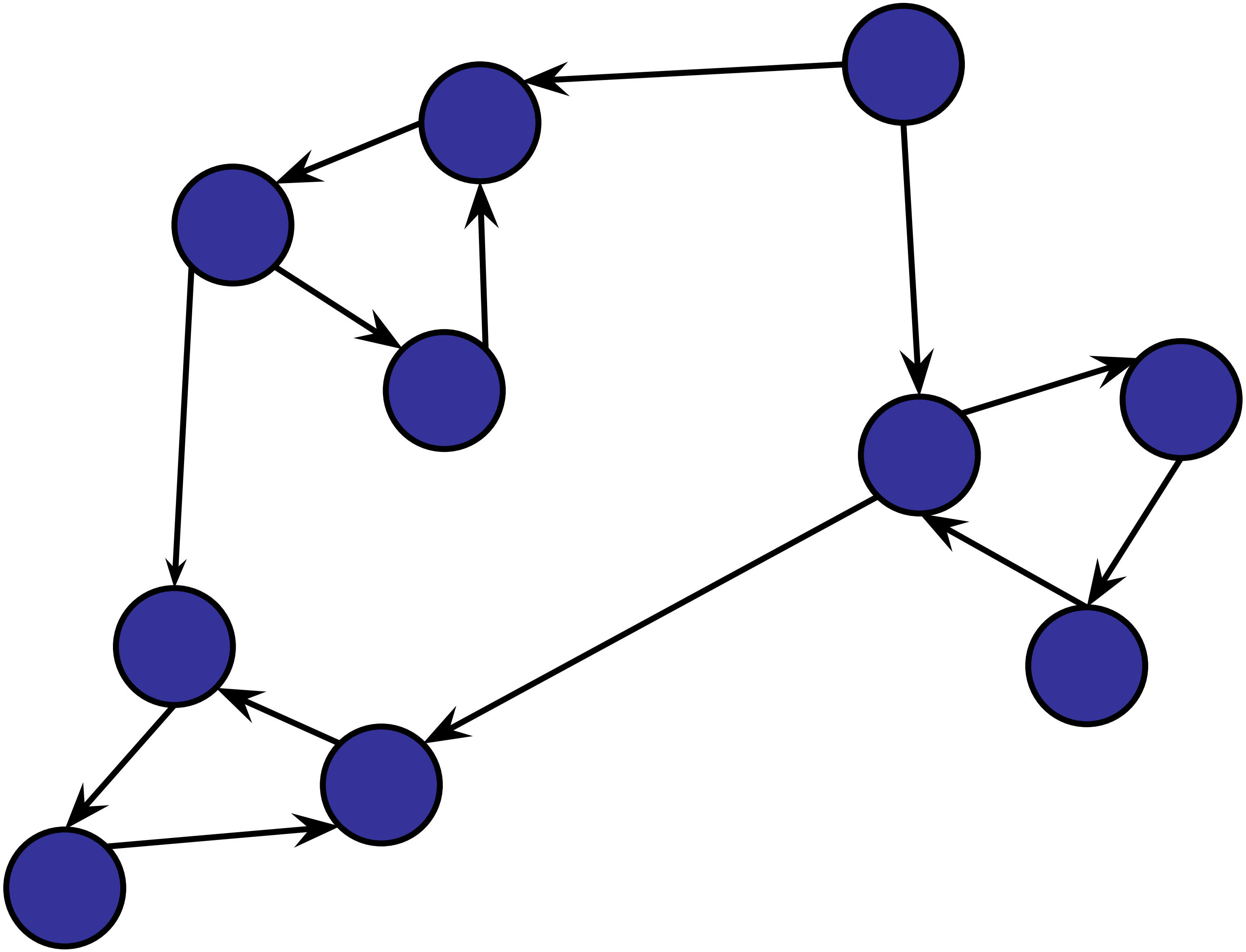
Clarification:

There are 3 possible cases of values to consider for setting a node **u**'s low time. This will be the minimum of:

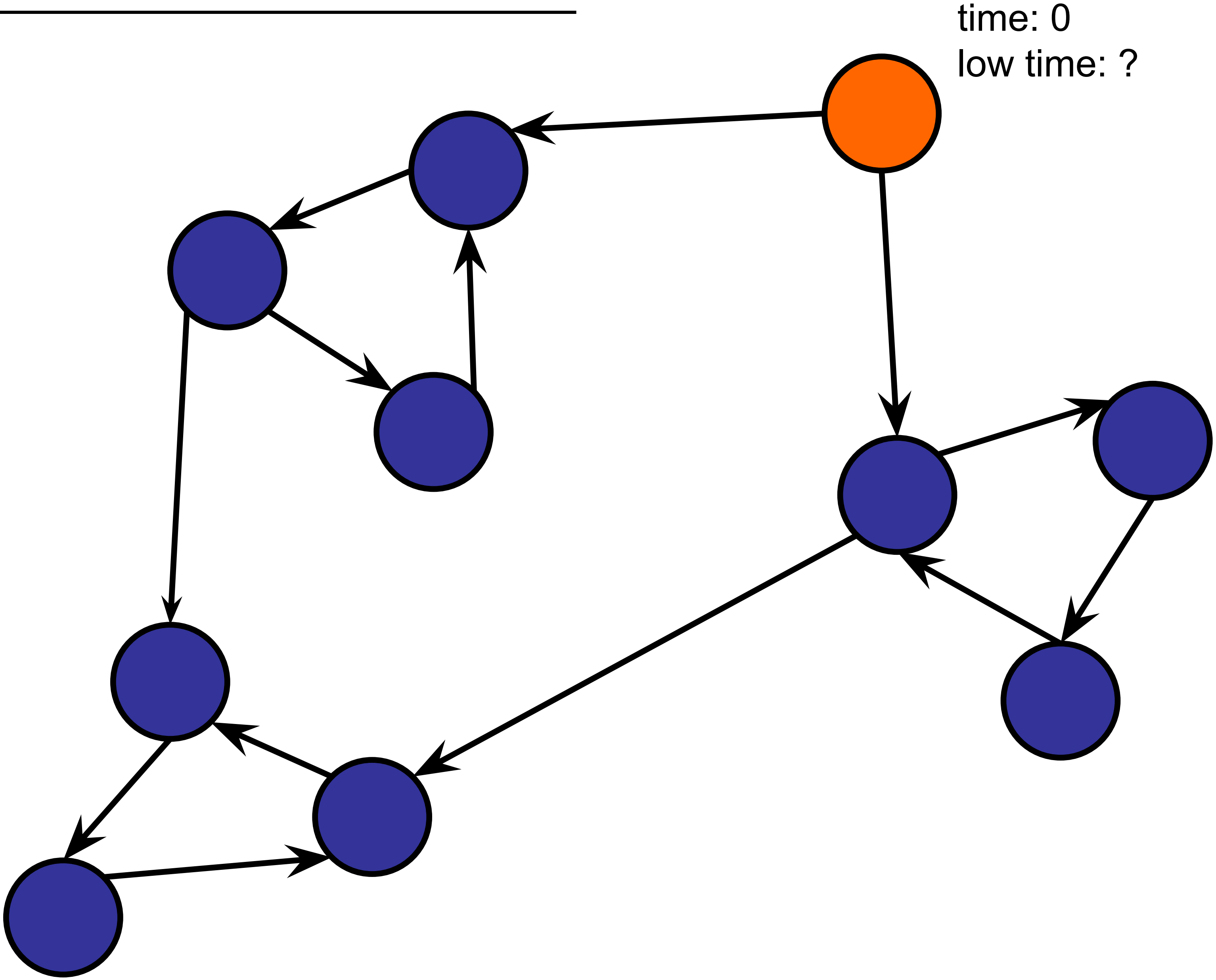
1. Node **u**'s time itself.
2. For any neighbour **v** of **u**, whose time is set, but low time is not set. We consider their time.
3. For any neighbour **v** of **u**, whose time is not set, we will first recurse on them. And consider their resulting low time.

(We'll ignore any neighbours whose low time is set before we visit them)

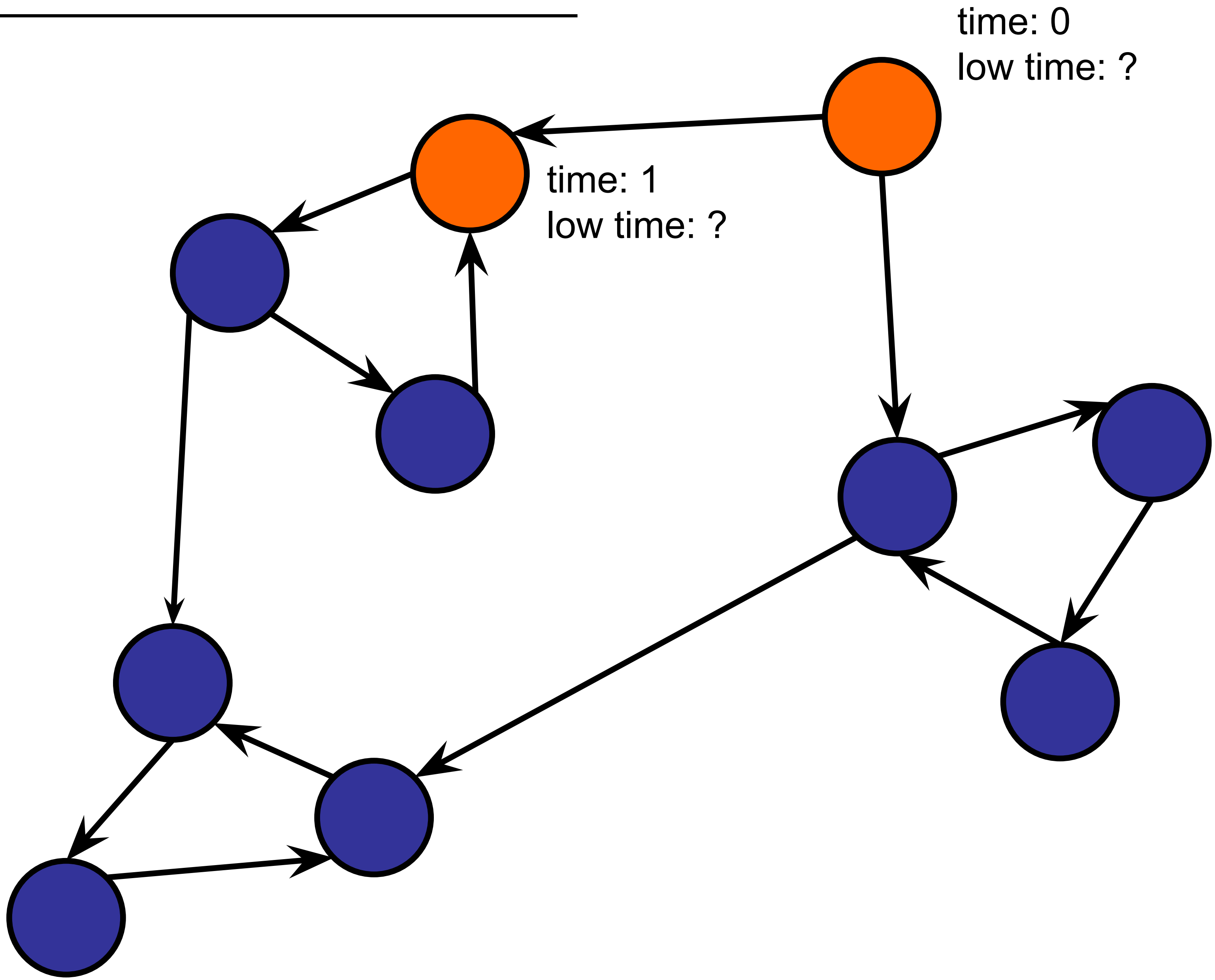
Connected Components



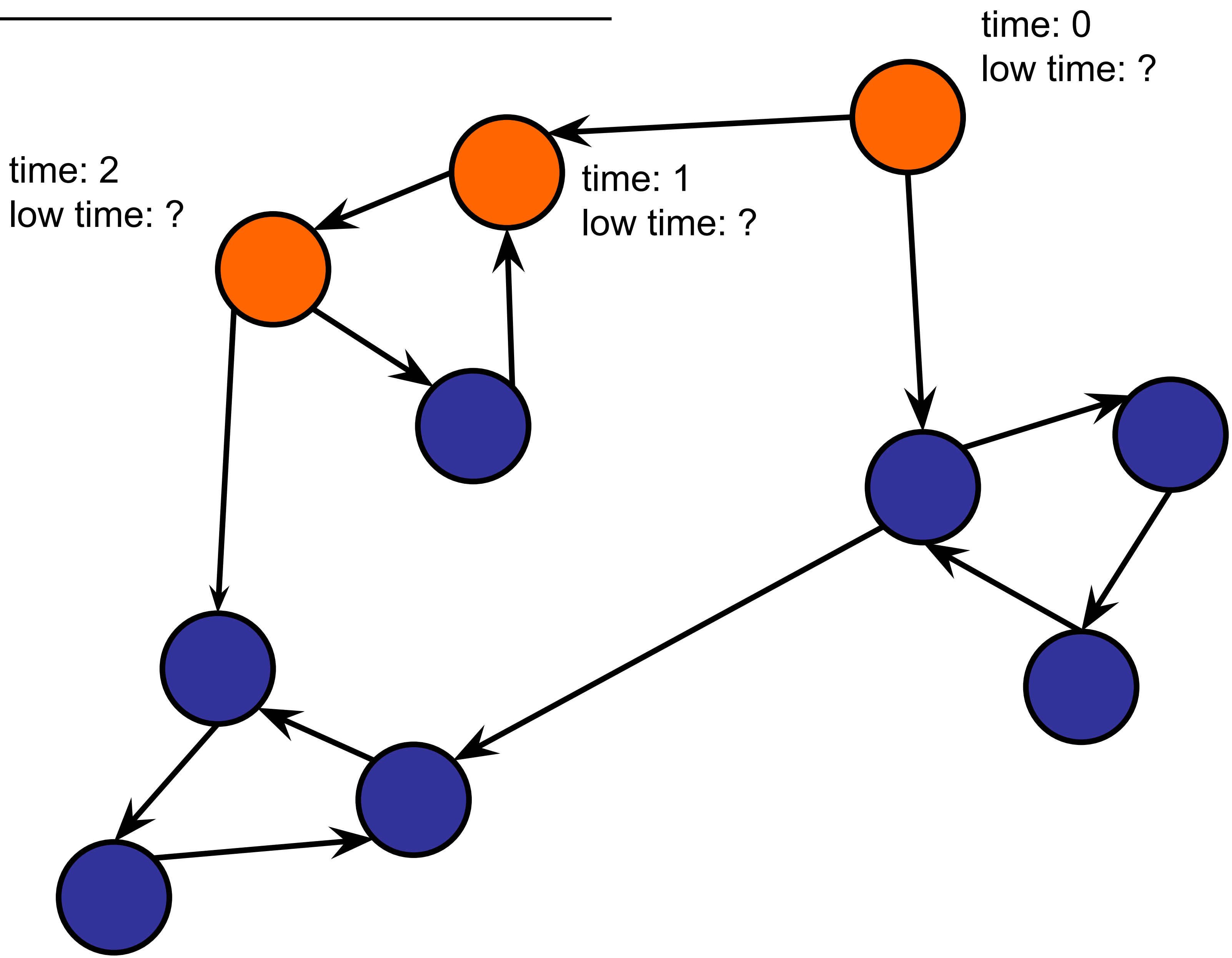
Connected Components



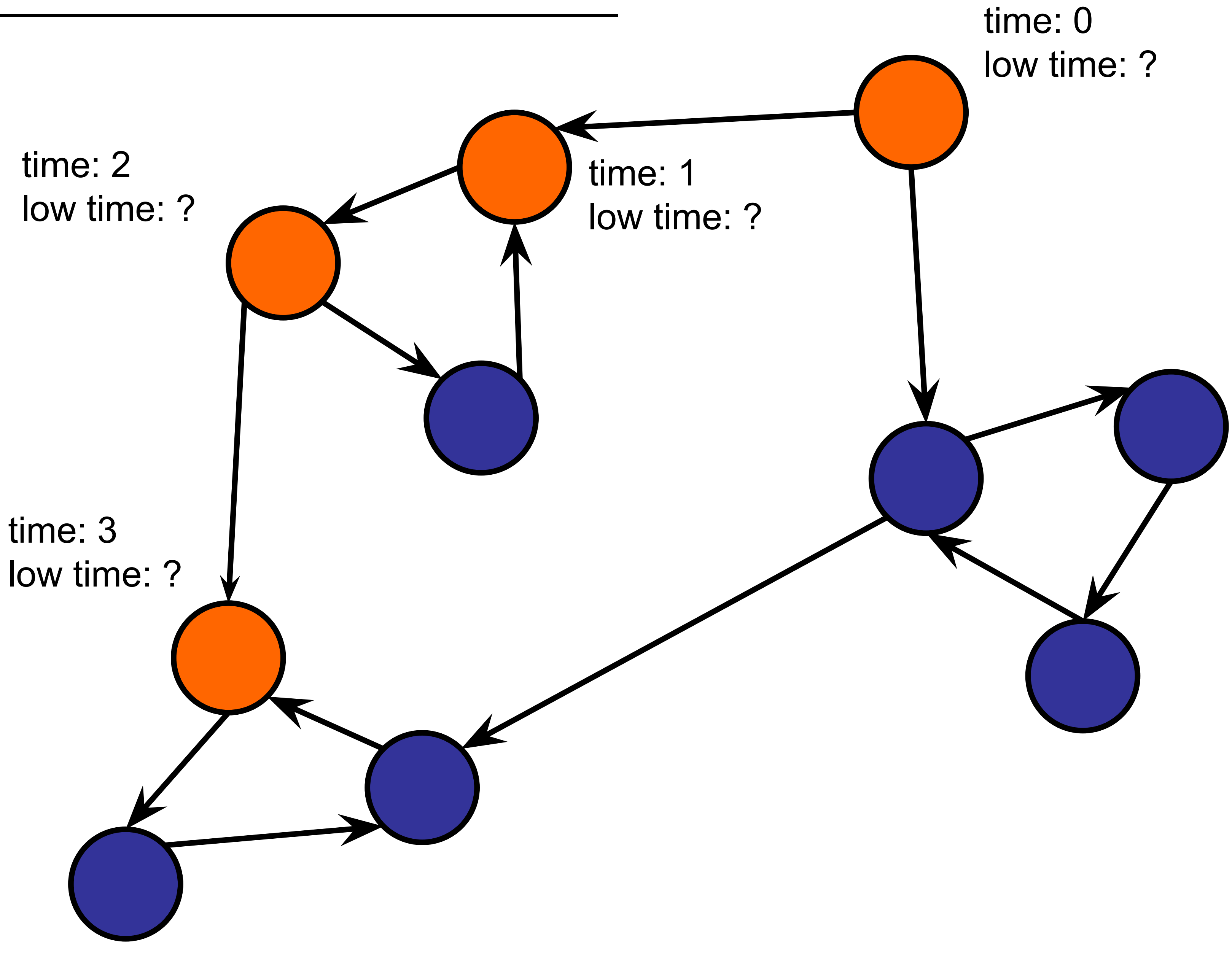
Connected Components



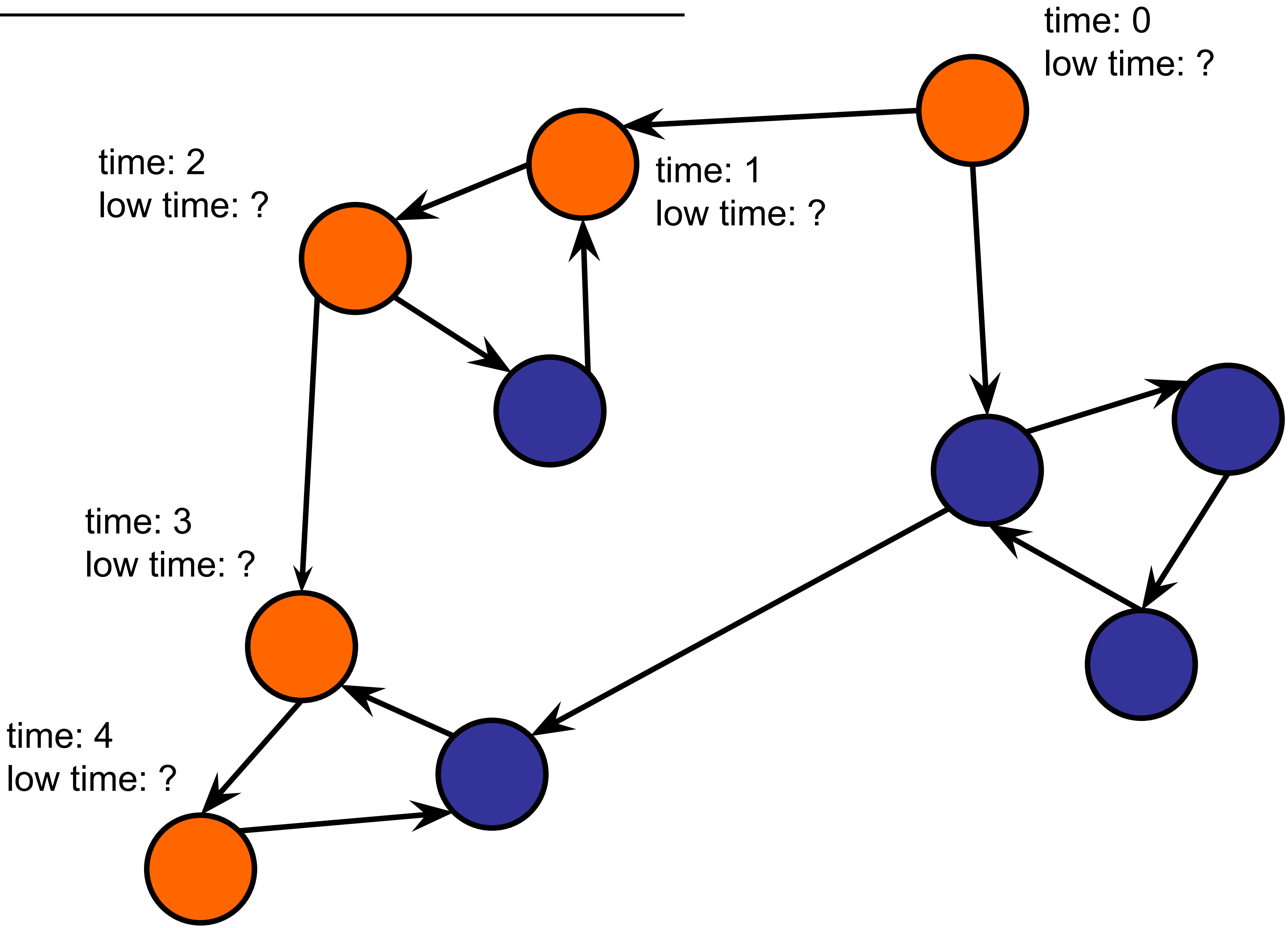
Connected Components



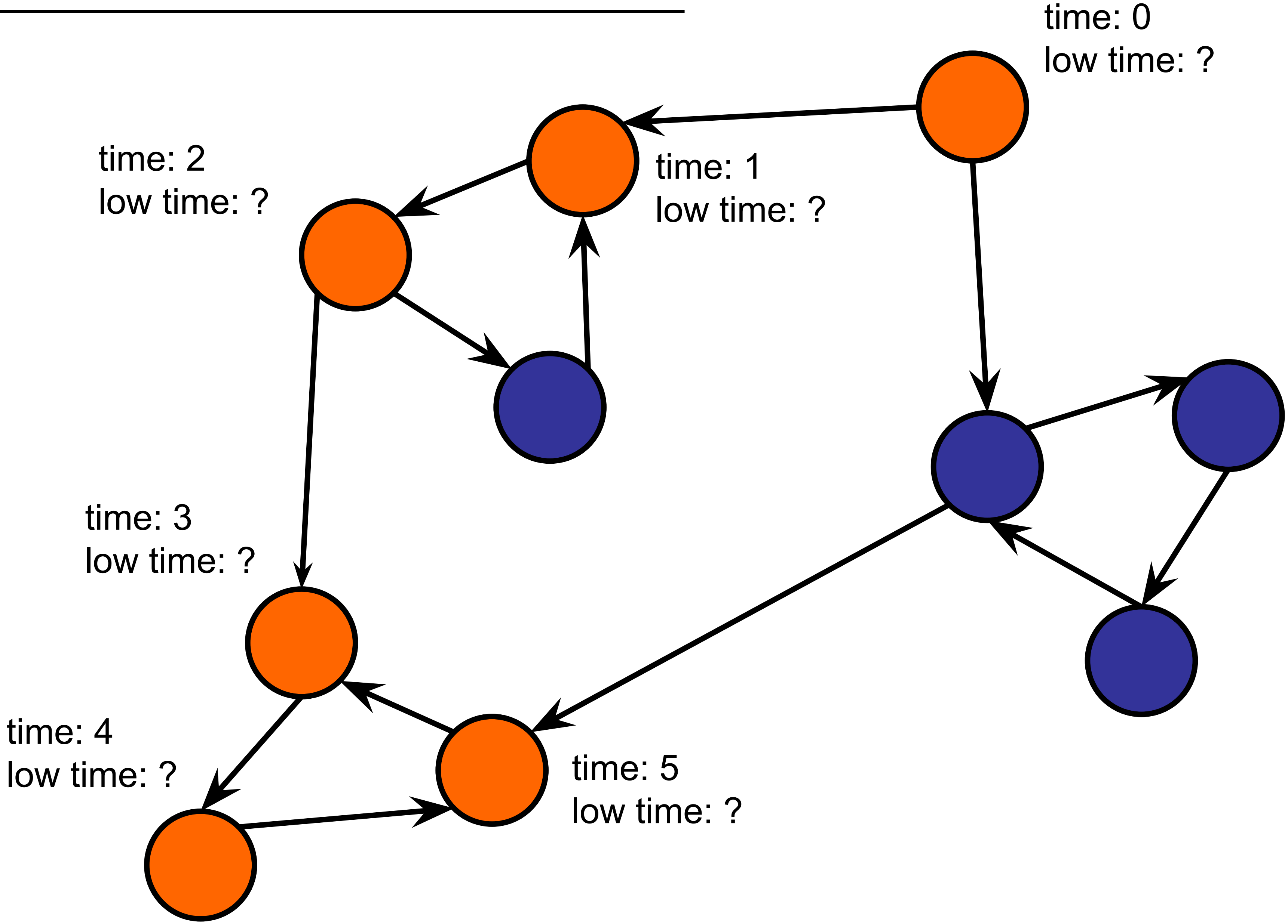
Connected Components



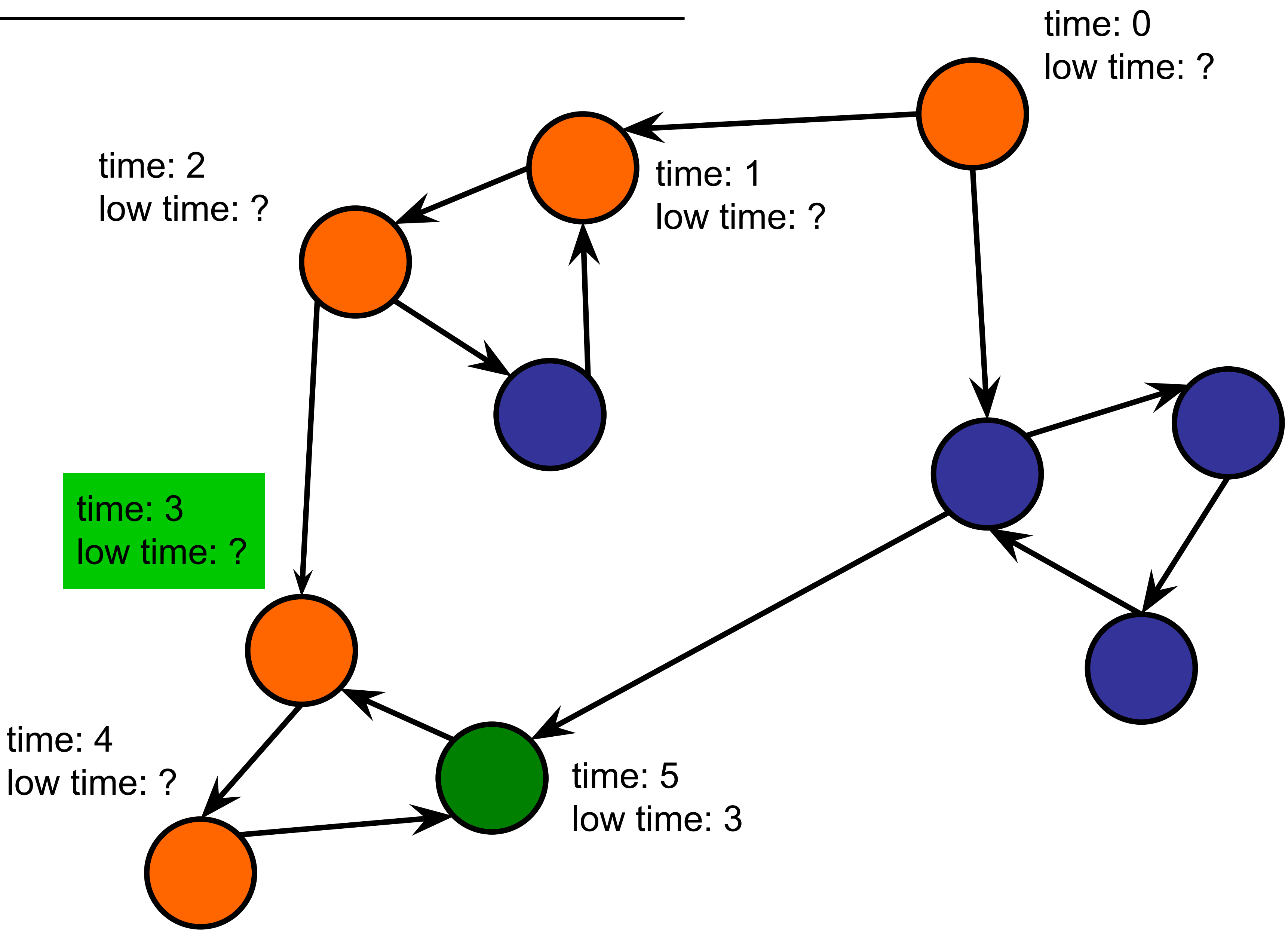
Connected Components



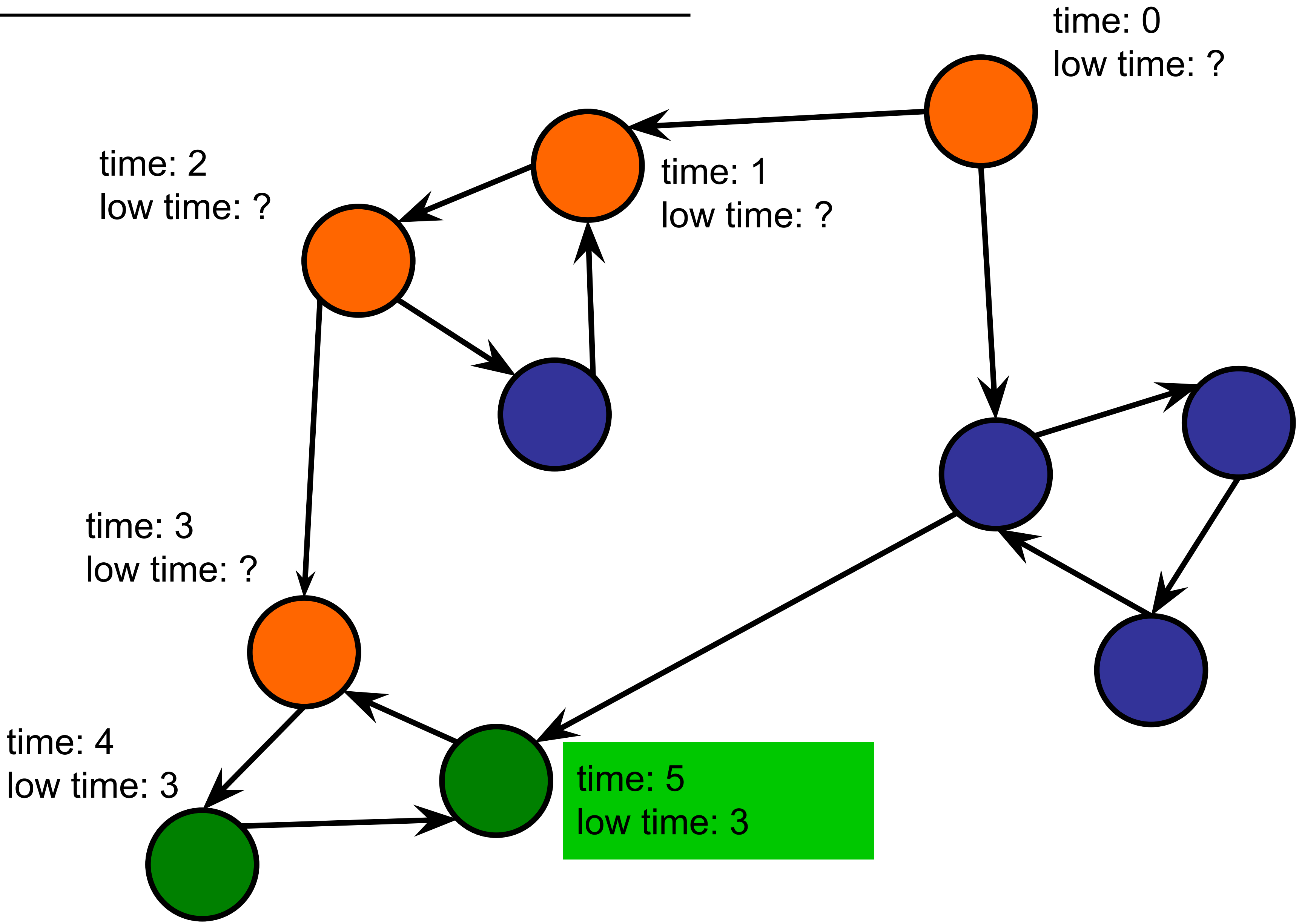
Connected Components



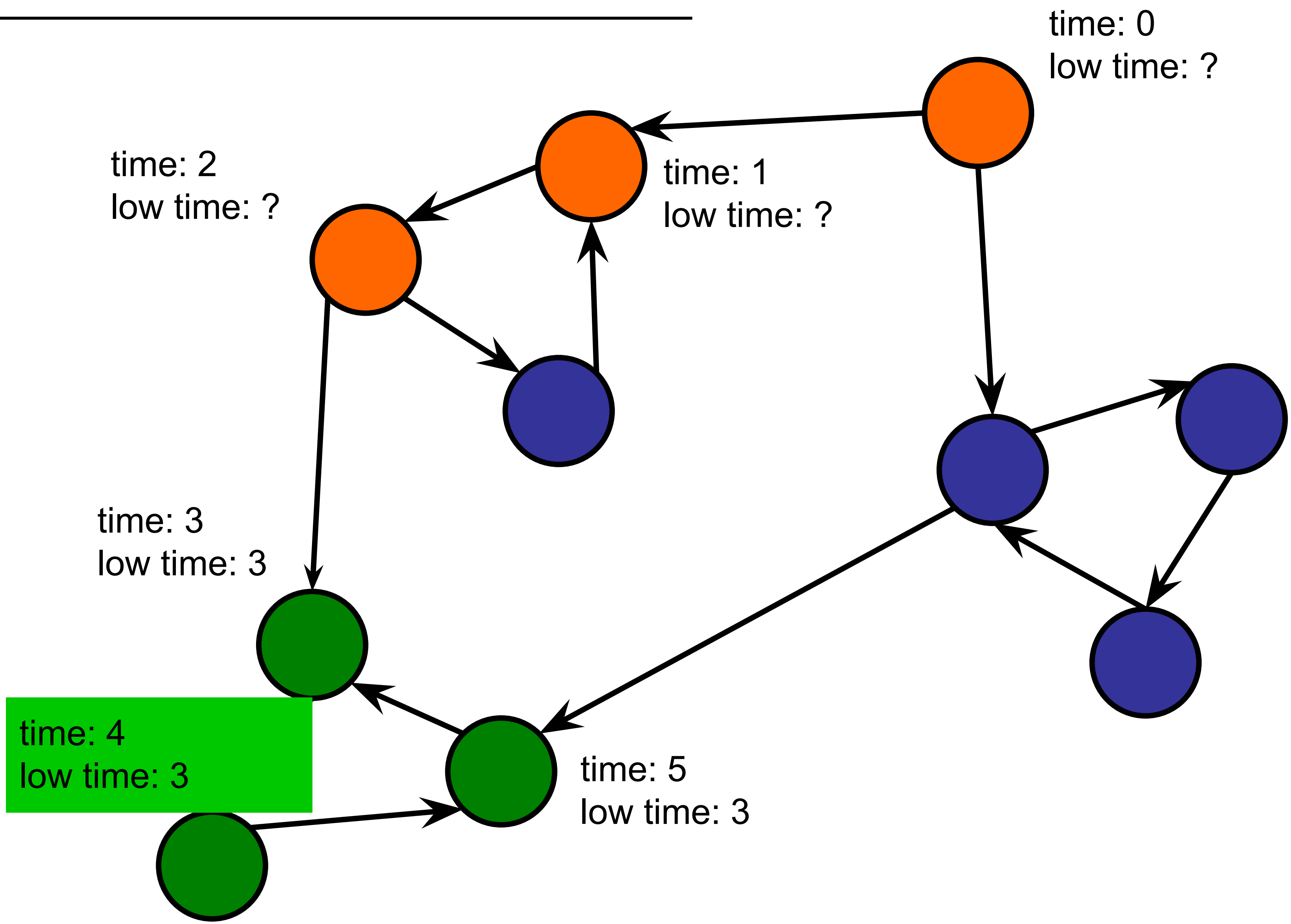
Connected Components



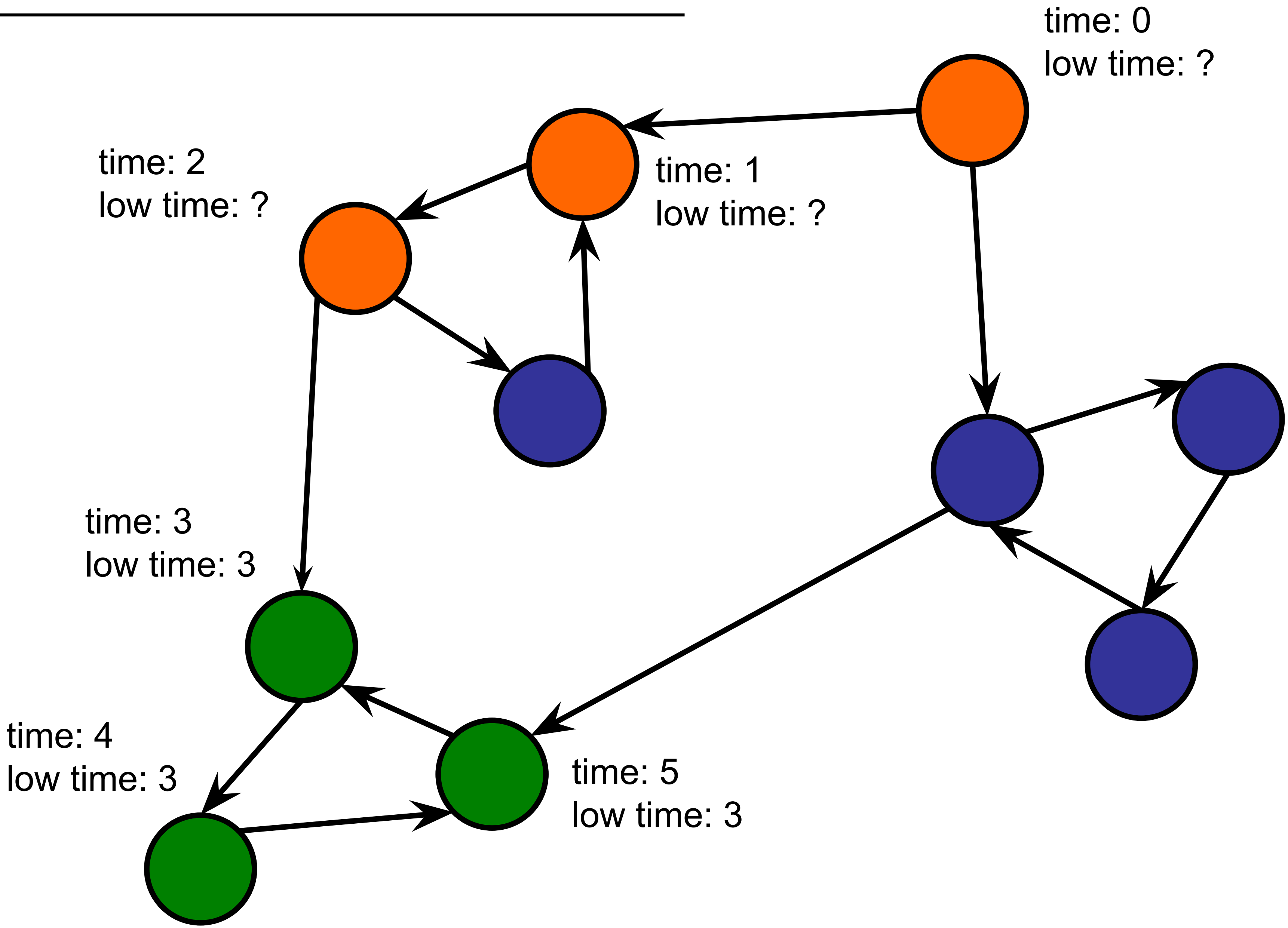
Connected Components



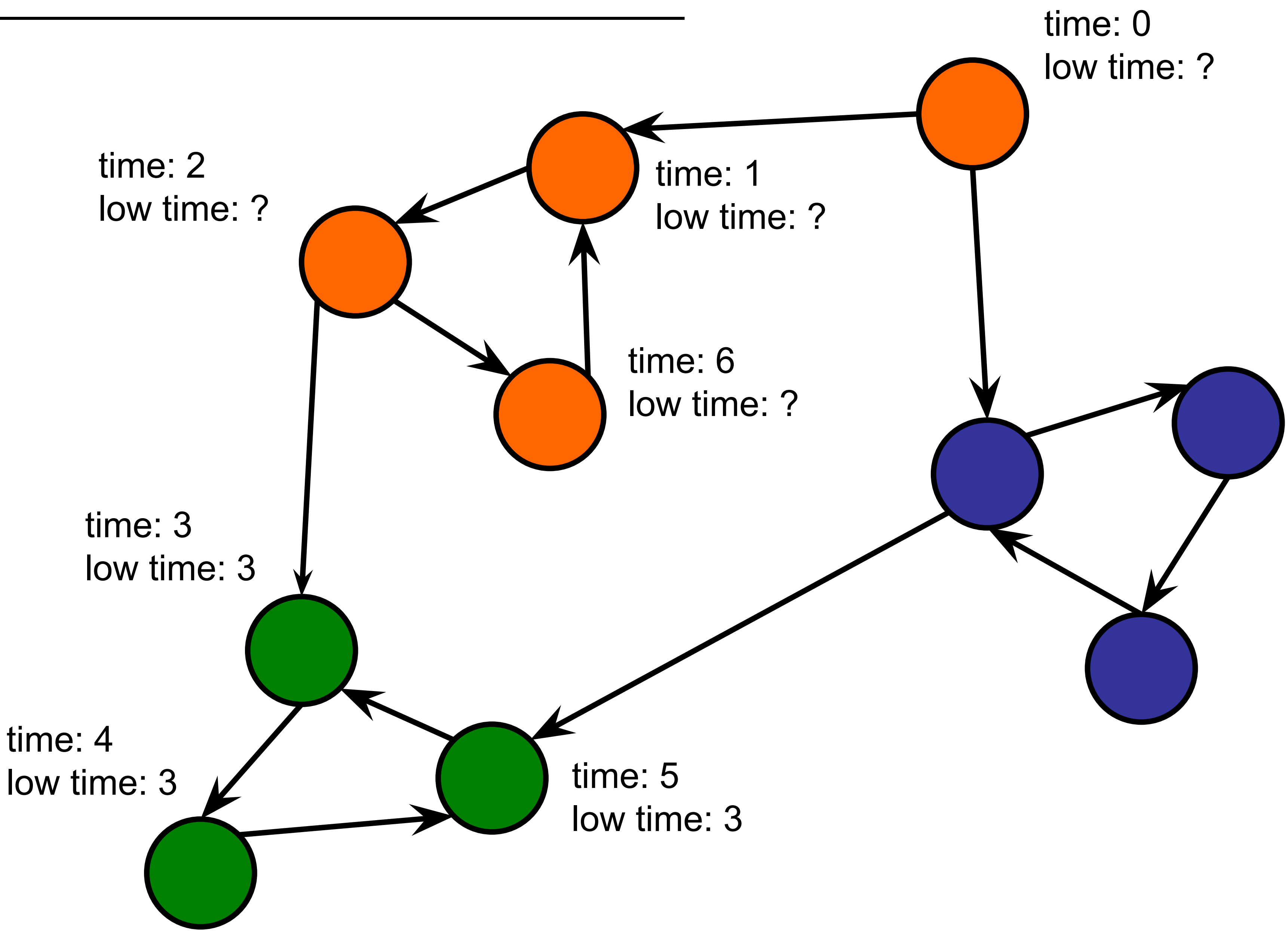
Connected Components



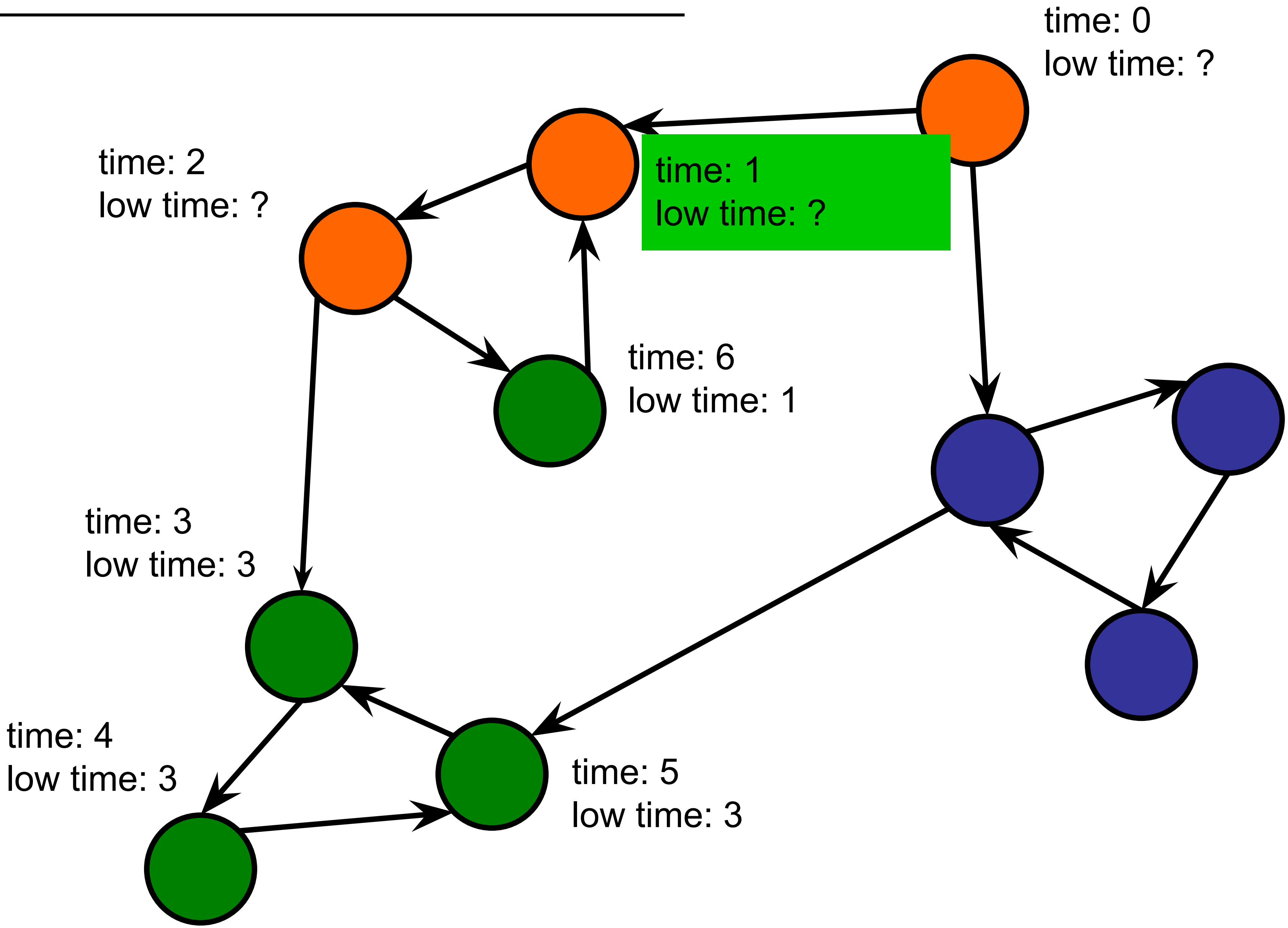
Connected Components



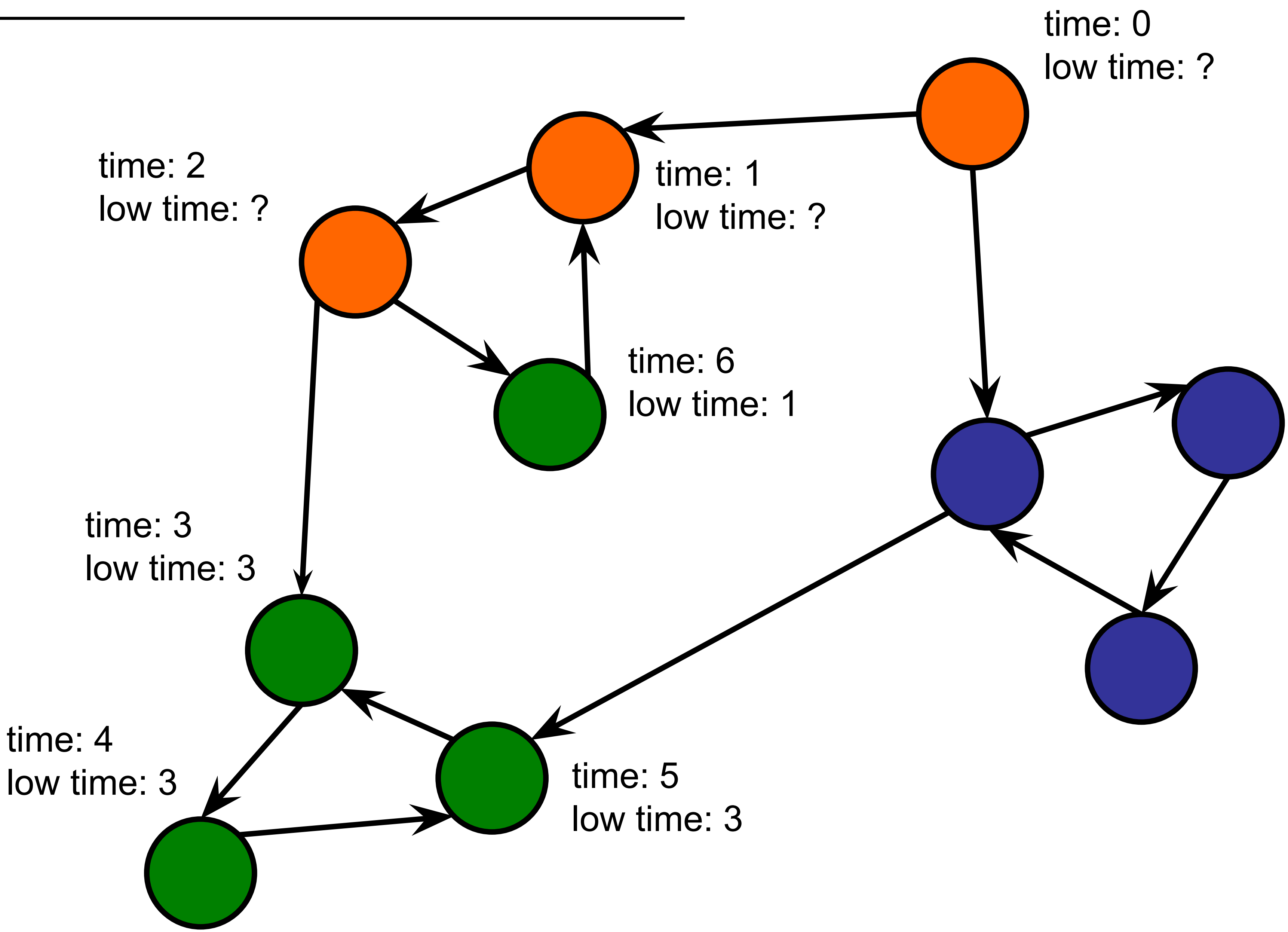
Connected Components



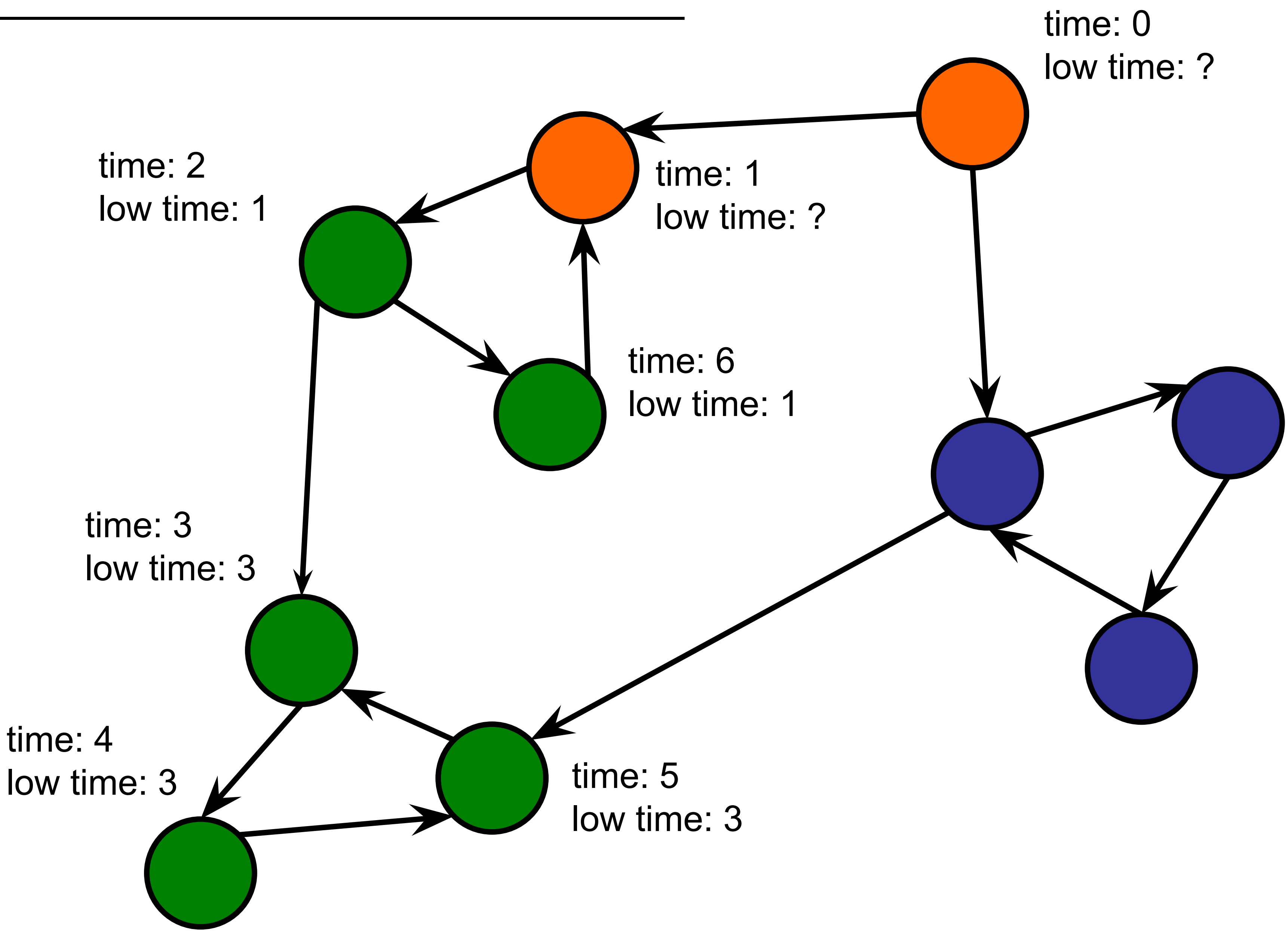
Connected Components



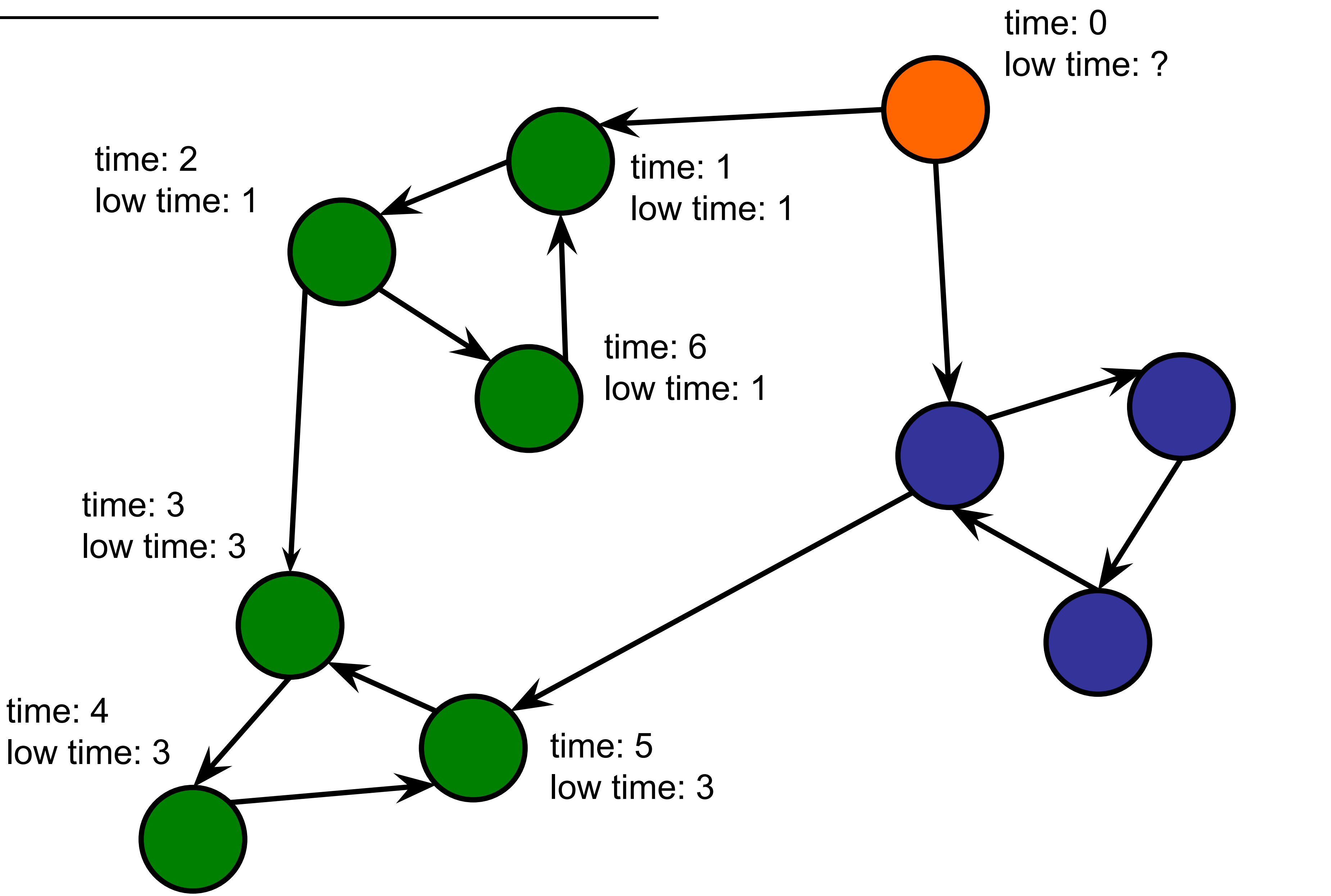
Connected Components



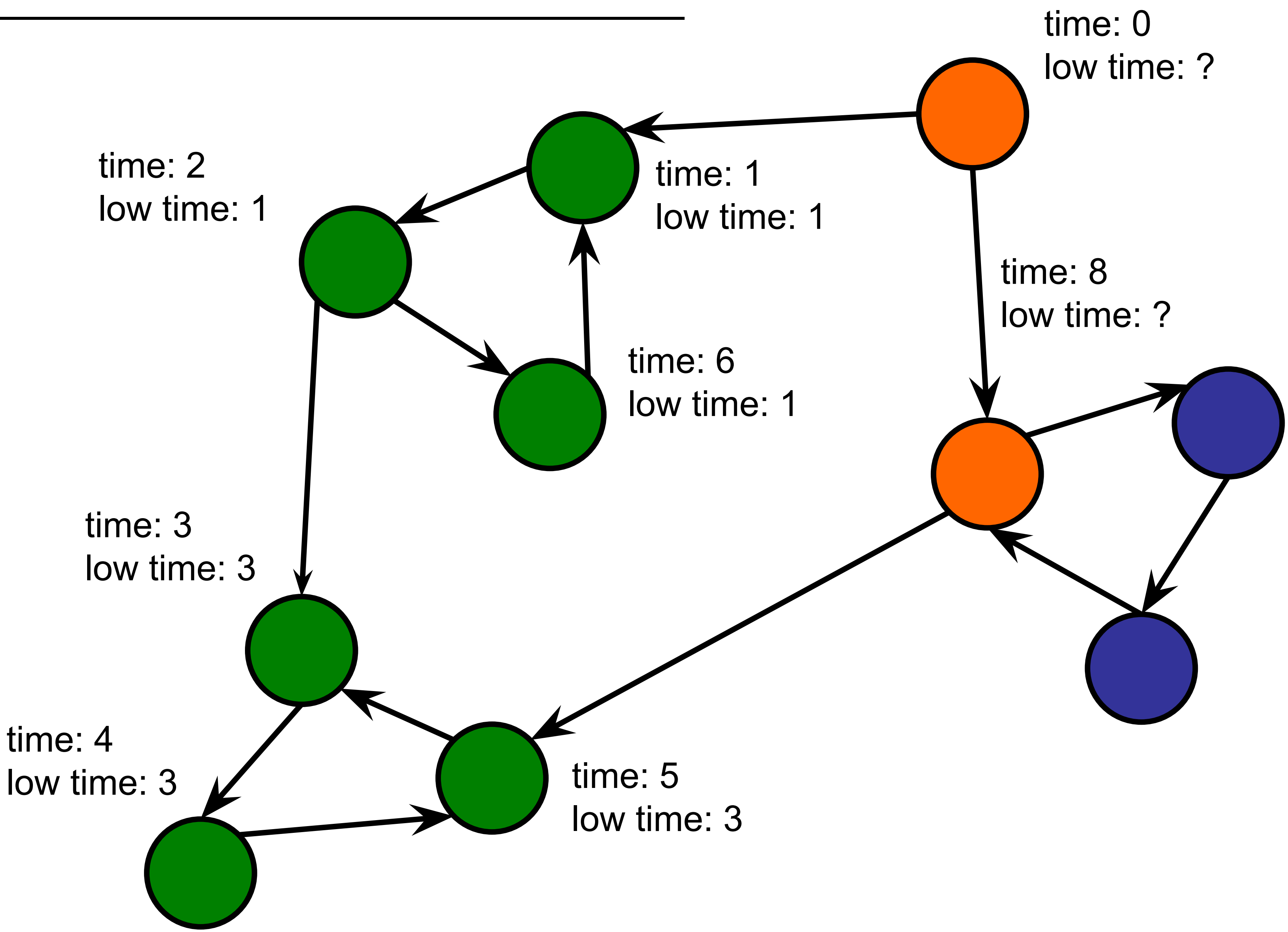
Connected Components



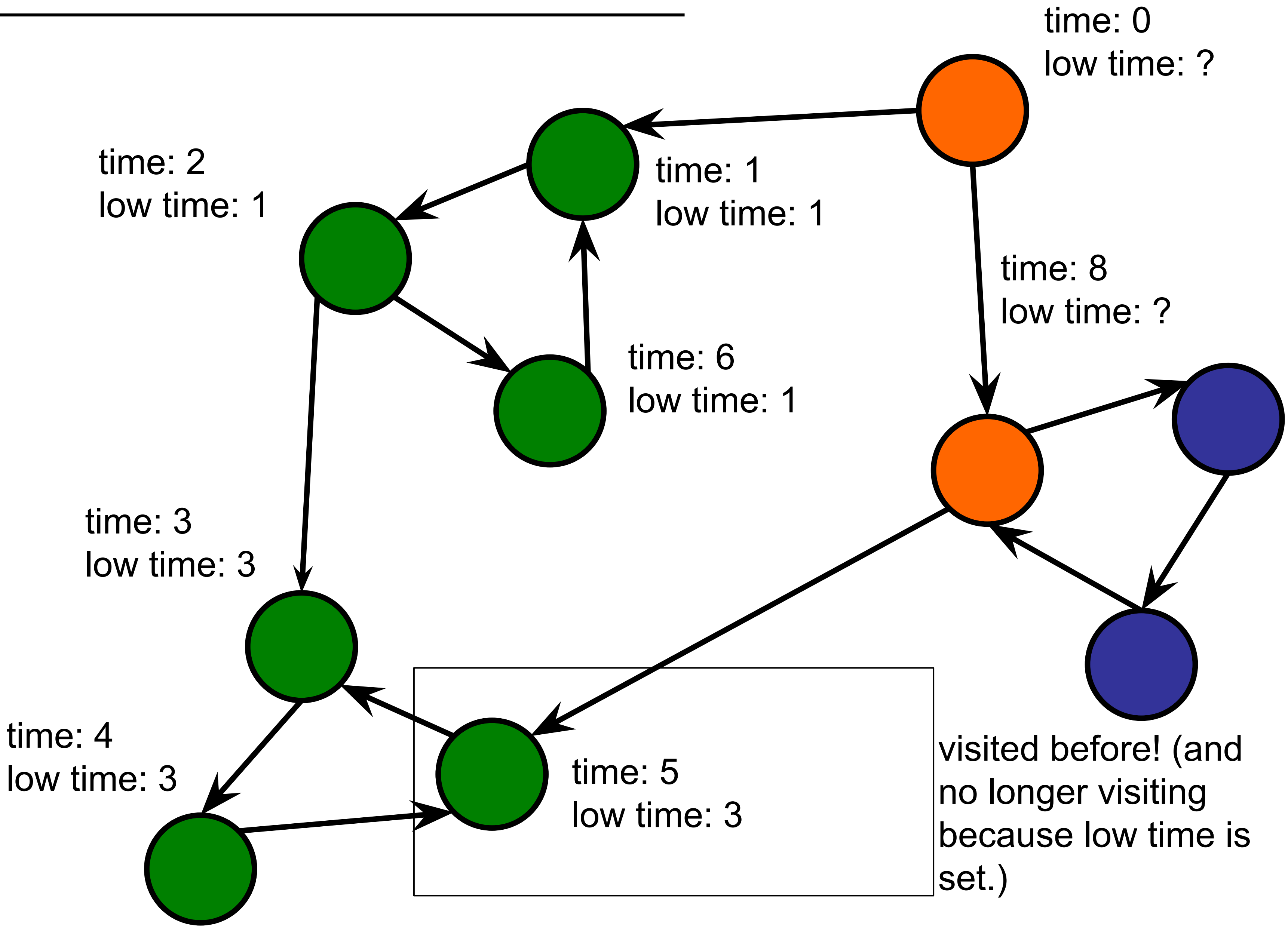
Connected Components



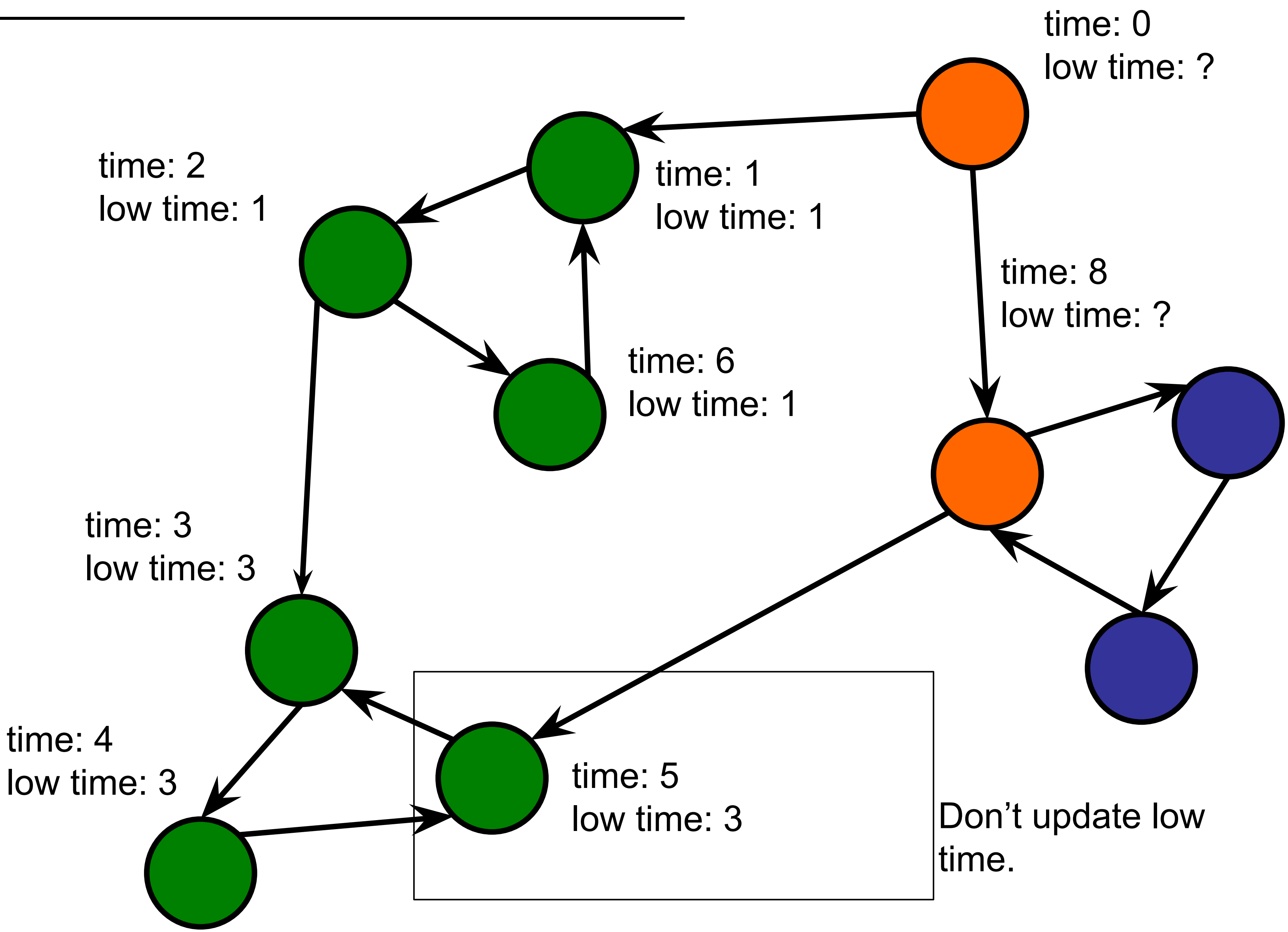
Connected Components



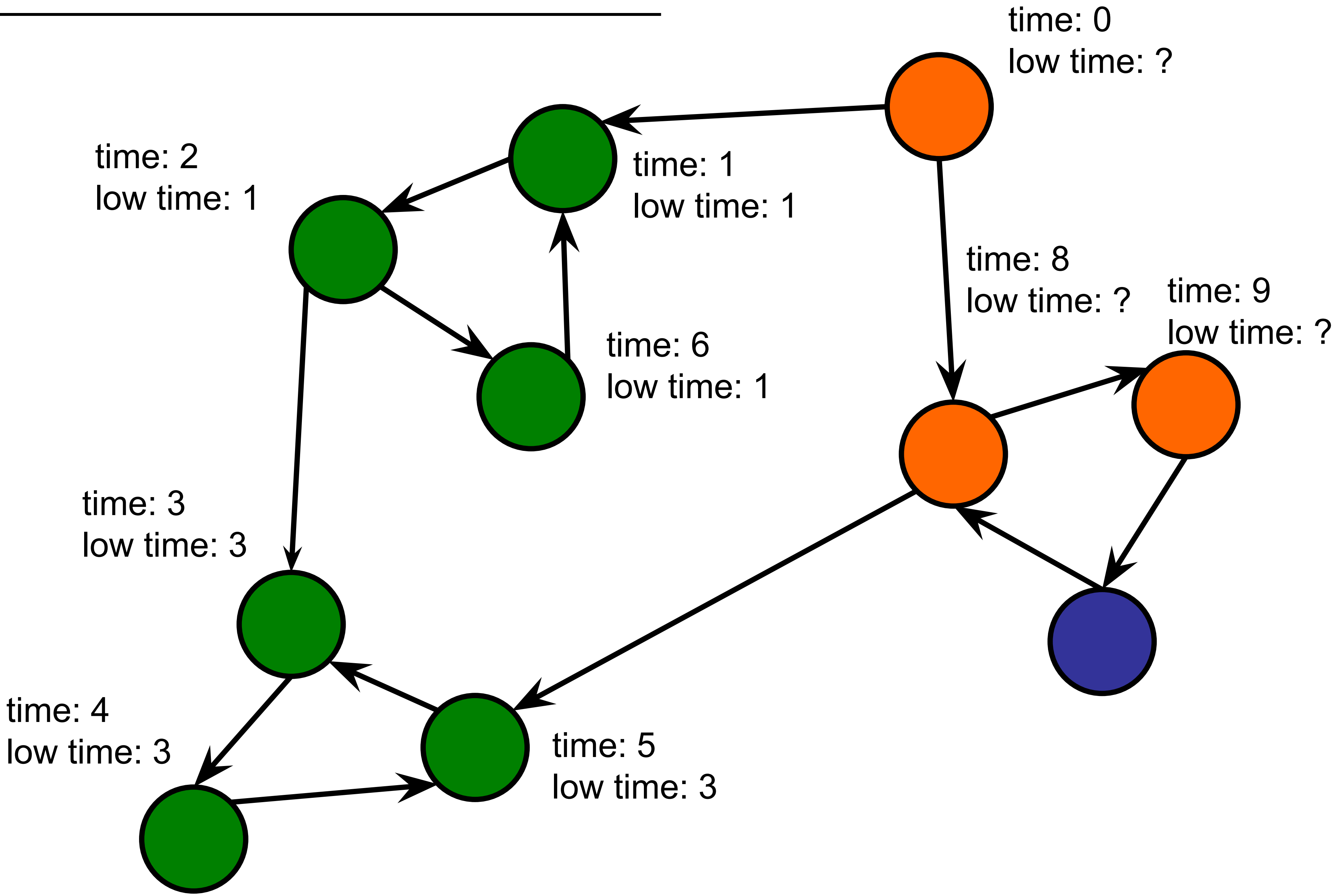
Connected Components



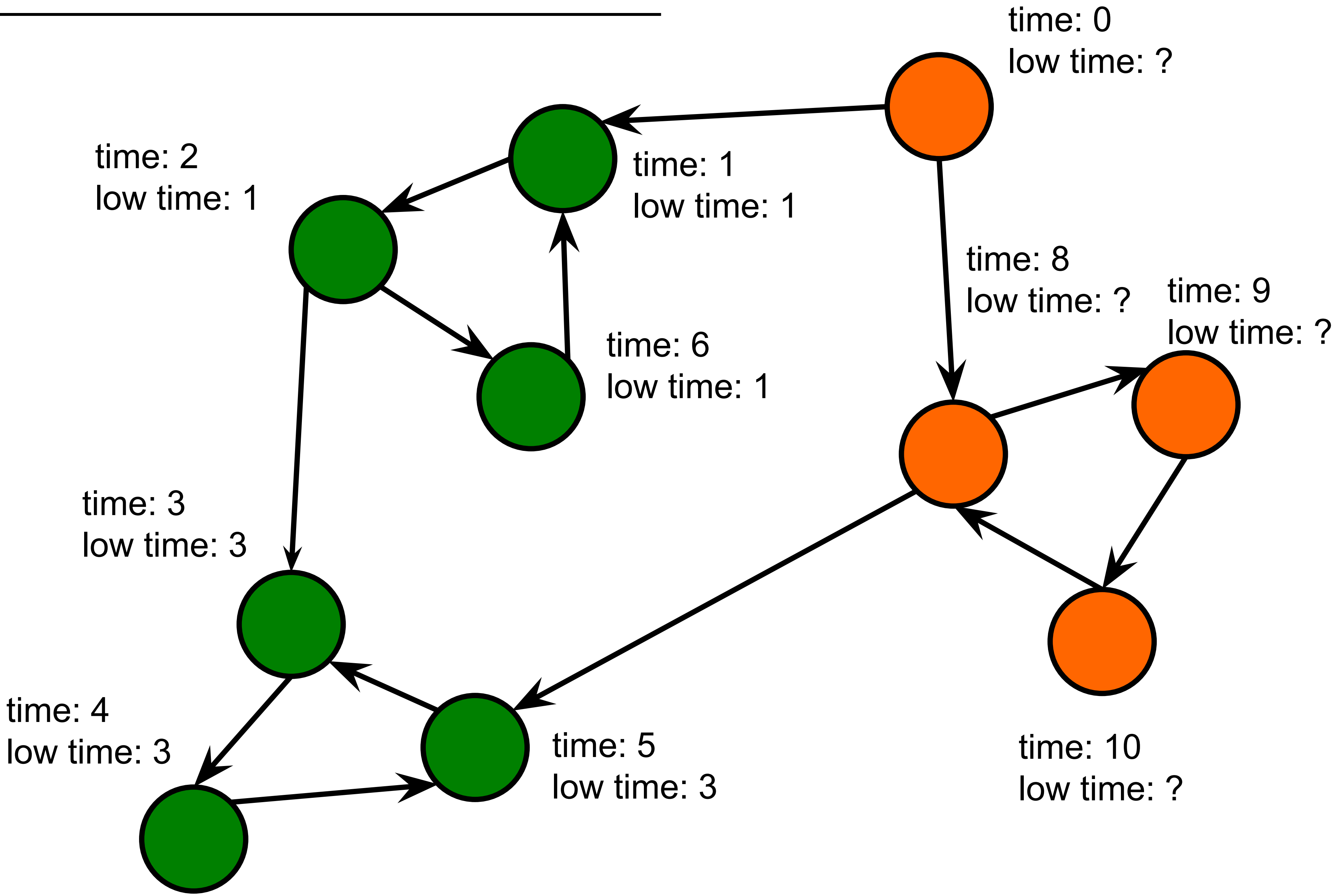
Connected Components



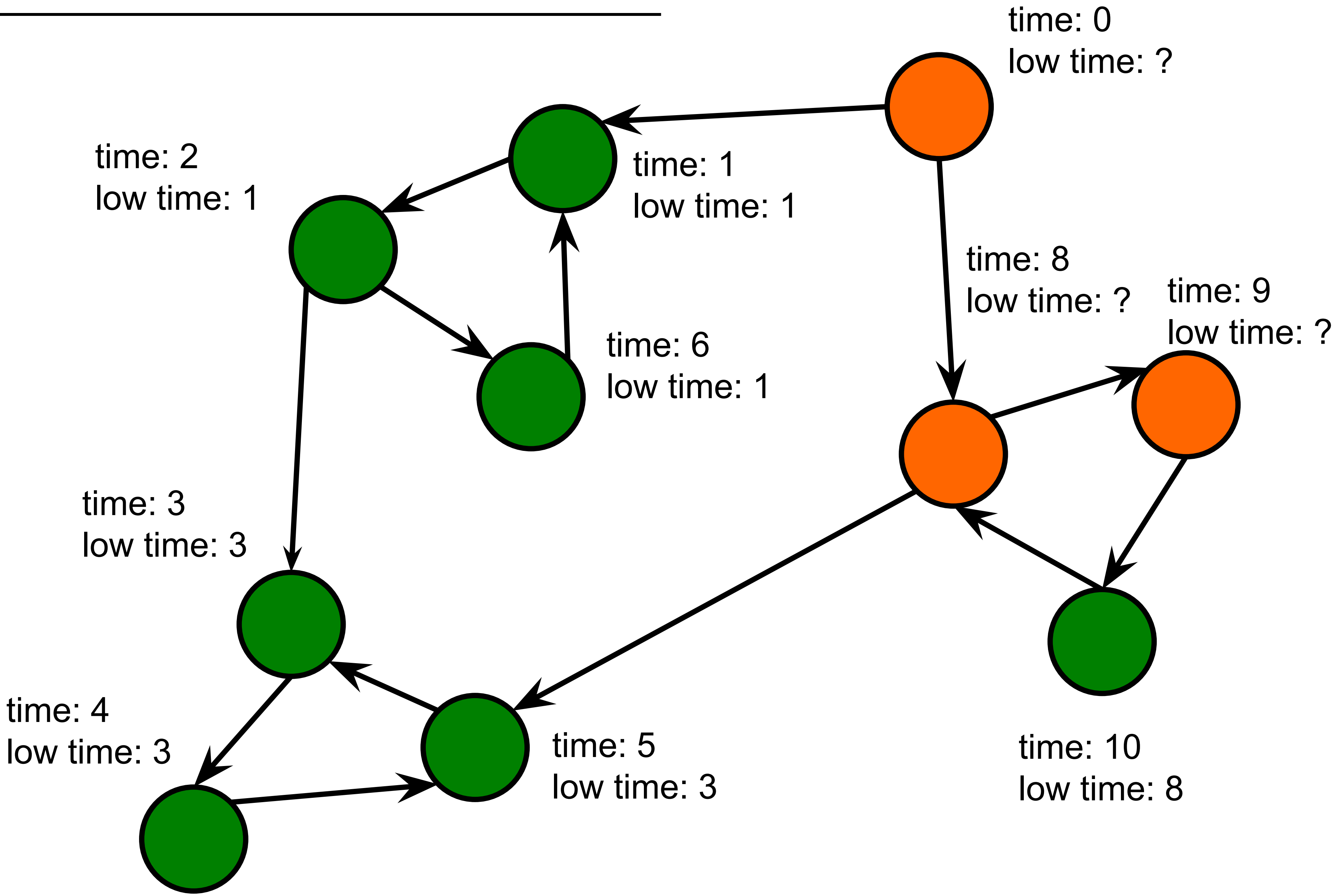
Connected Components



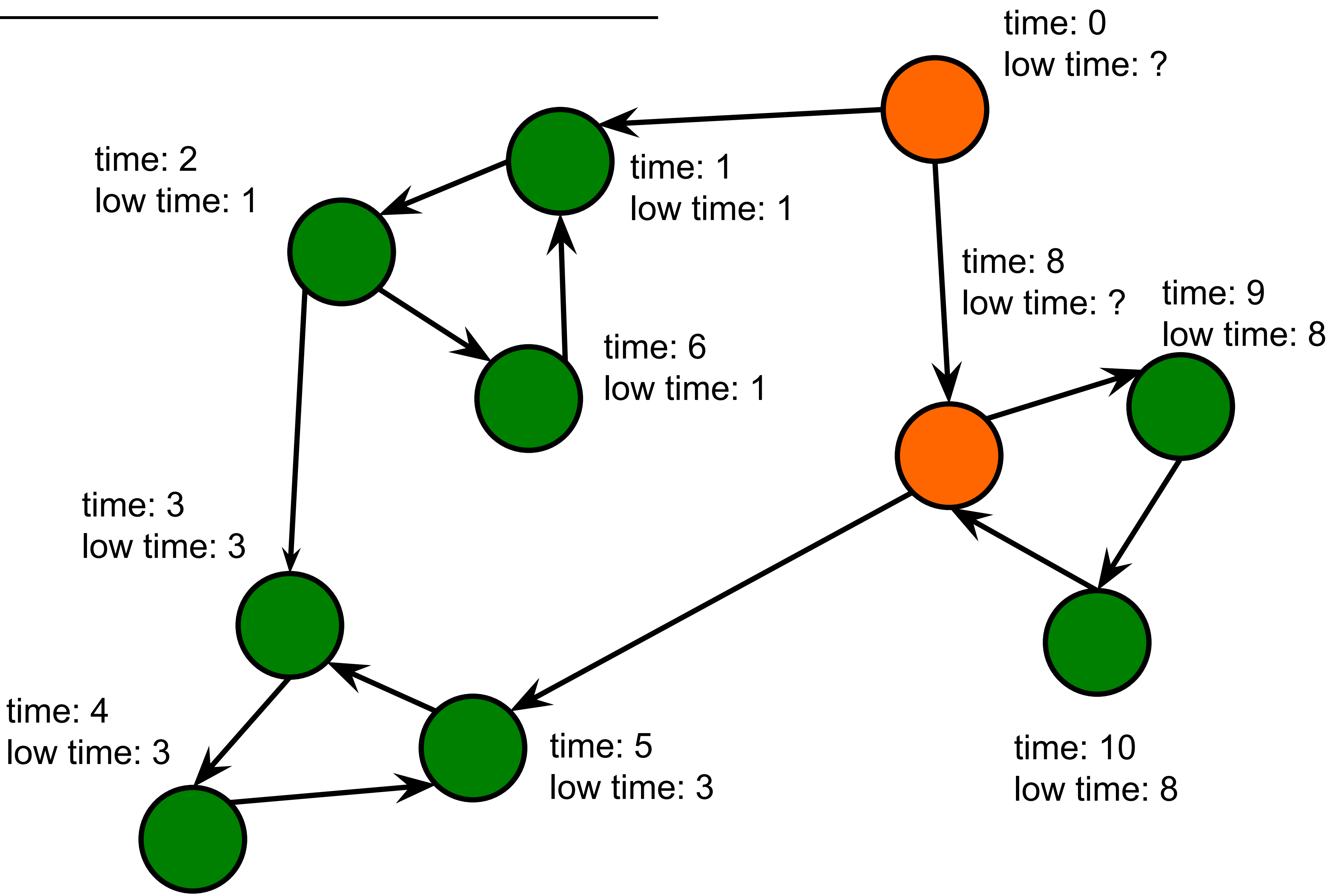
Connected Components



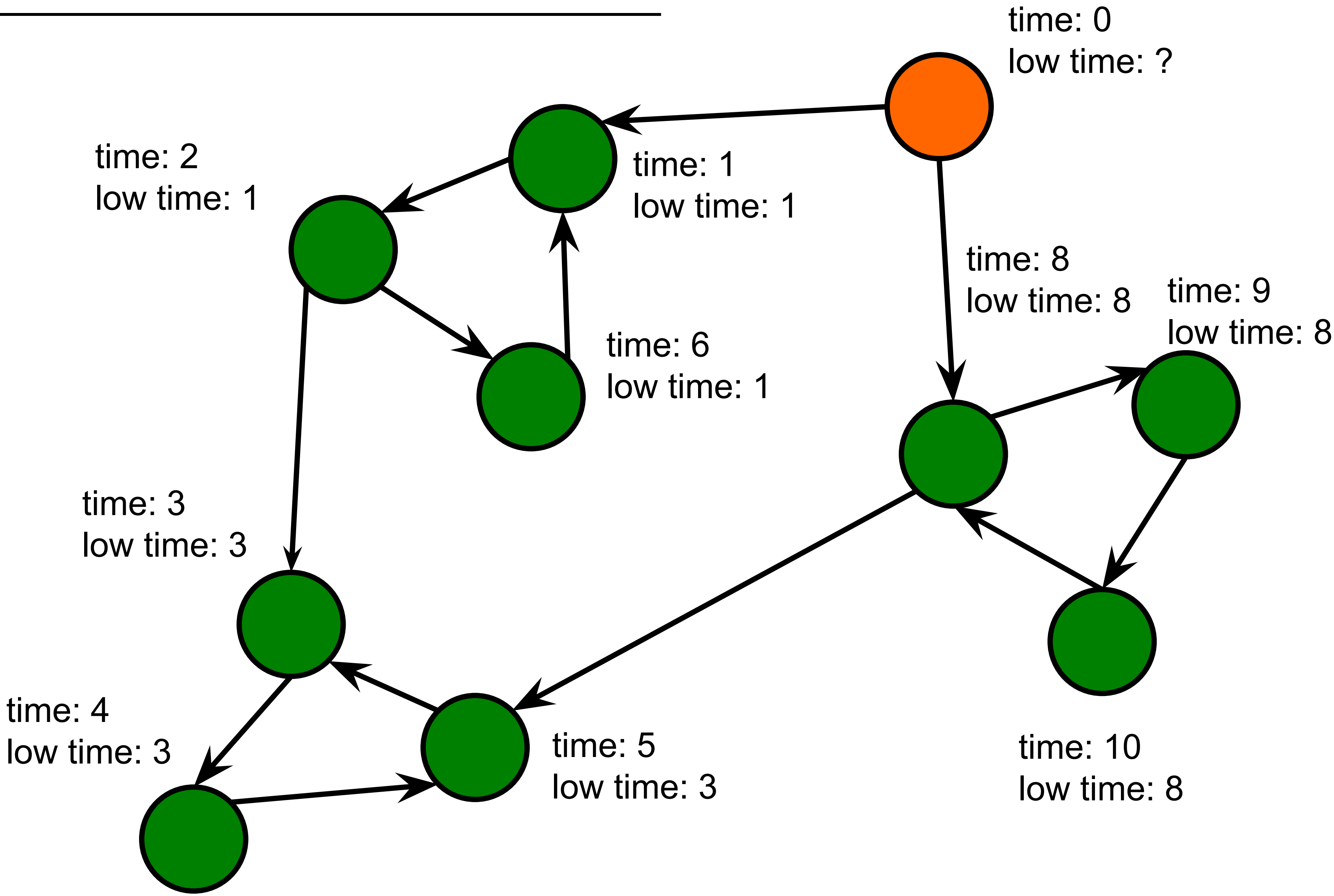
Connected Components



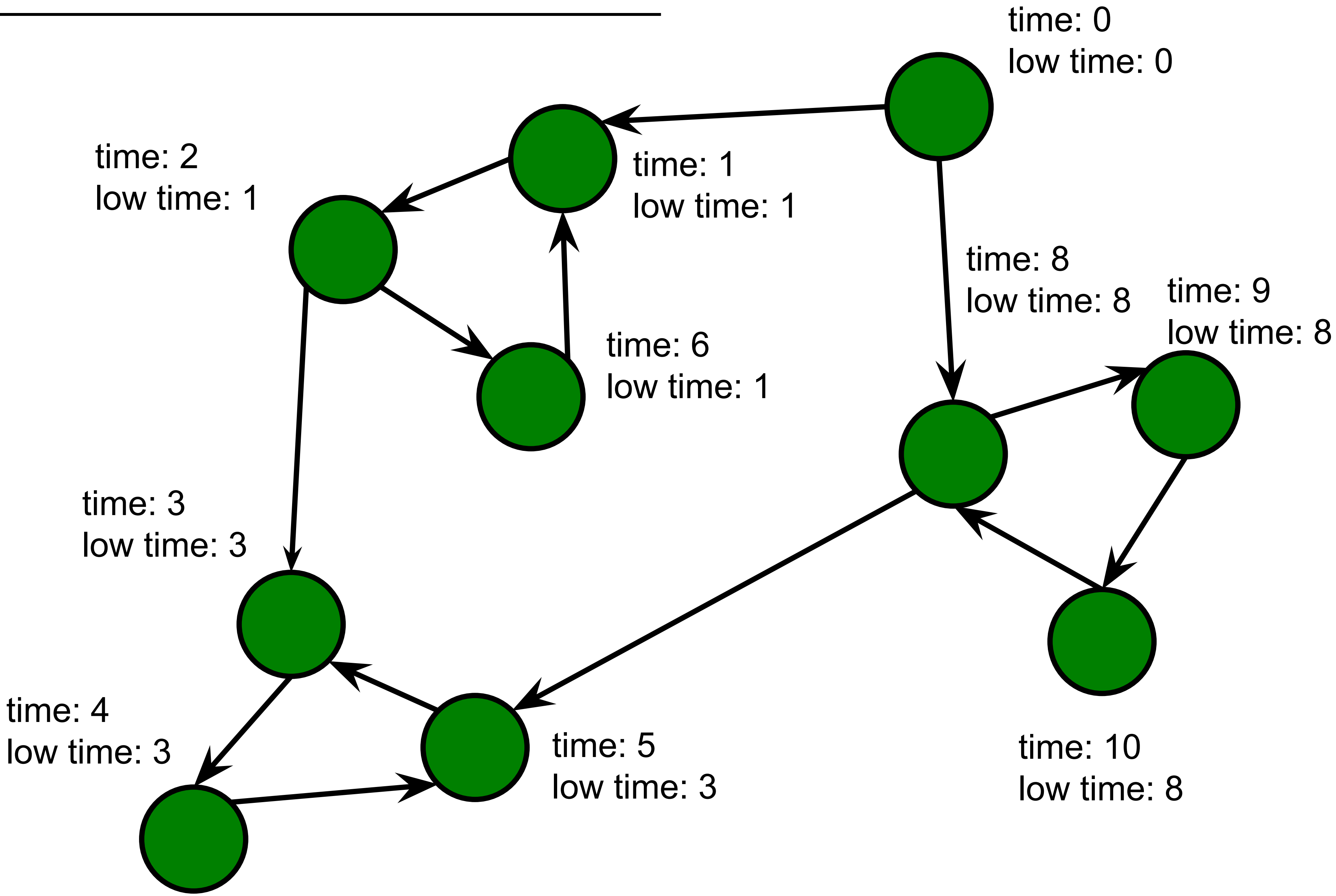
Connected Components



Connected Components



Connected Components



Connected Components

Low time of a node is the minimum of:

1. Its own time
2. Low time of children that we just (visited)/(recursed from)

Cycle Finding

Clarification:

There are 3 possible cases of values to consider for setting a node **u**'s low time. This will be the minimum of:

1. Node **u**'s time itself.
2. For any neighbour **v** of **u**, whose time is set, but low time is not set. We consider their time.
3. For any neighbour **v** of **u**, whose time is not set, we will first recurse on them. And consider their resulting low time.

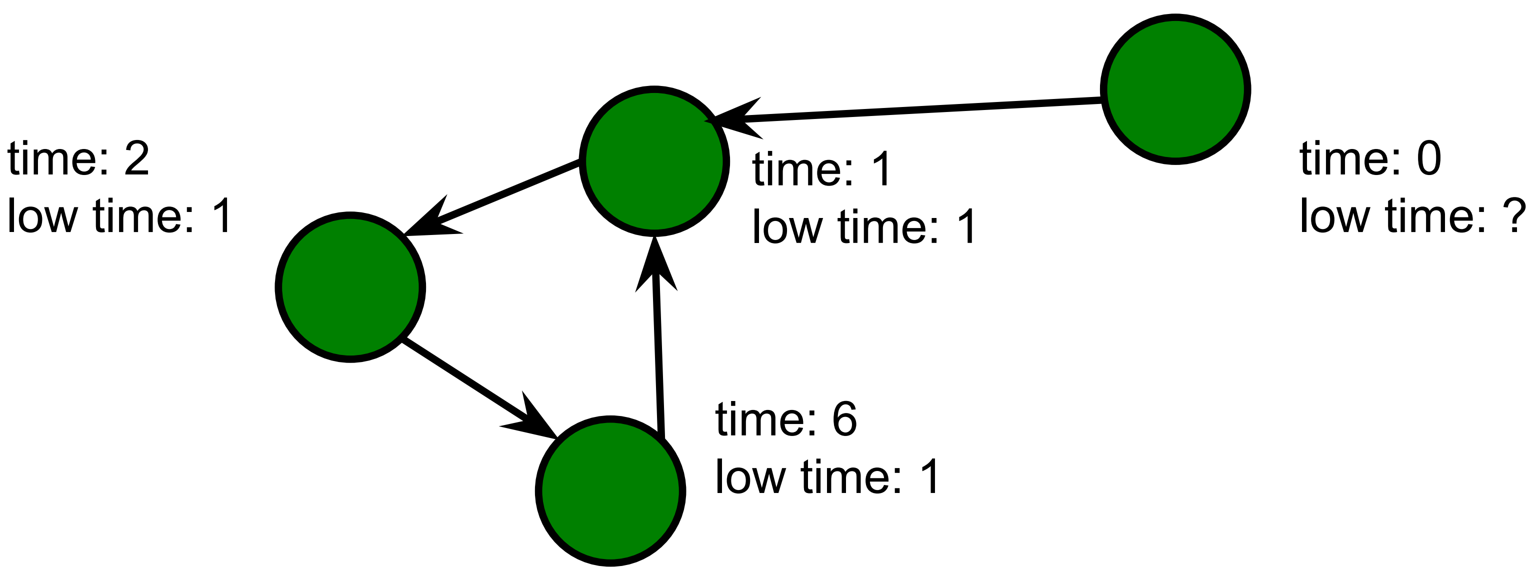
(We'll ignore any neighbours whose low time is set before we visit them)

Cycle Finding

Clarification:

What's the point of low time?

The point of the low time is to identify the "earliest" node that we can visit (that is actively being visited).



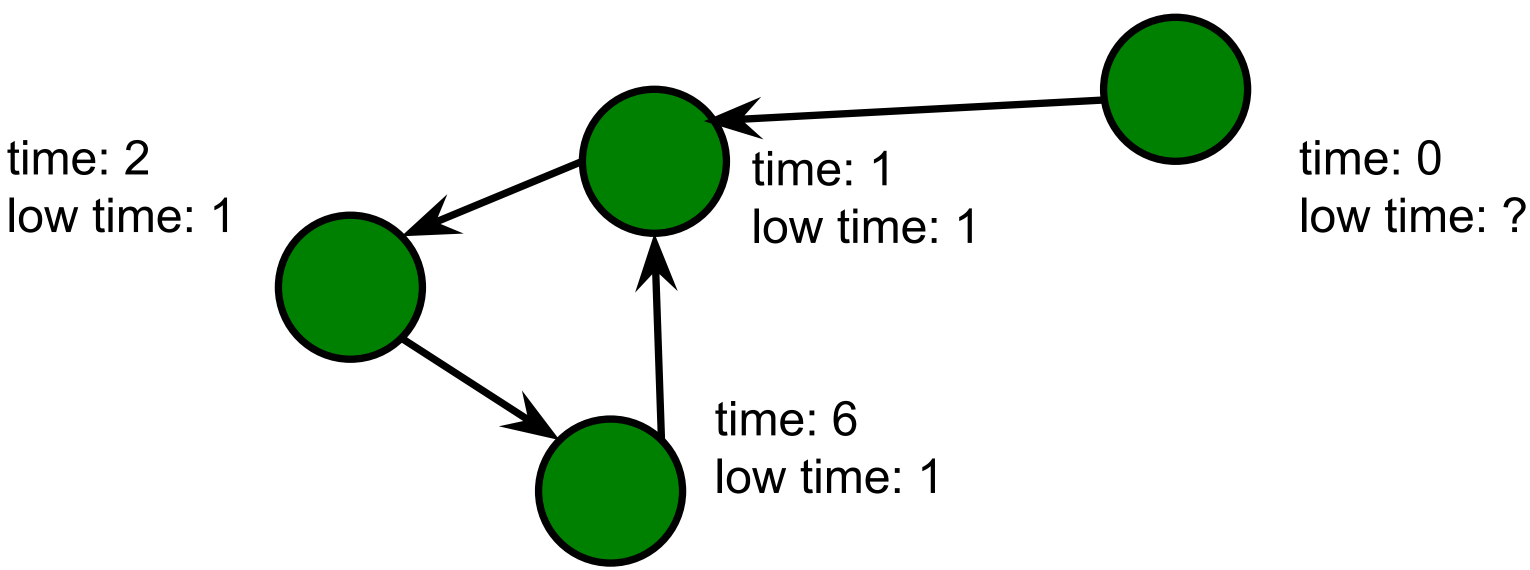
Cycle Finding

Clarification:

What's the point of low time?

The point of the low time is to identify the "earliest" node that we can visit (that is actively being visited).

Important: We only set low times as a post traversal operation.



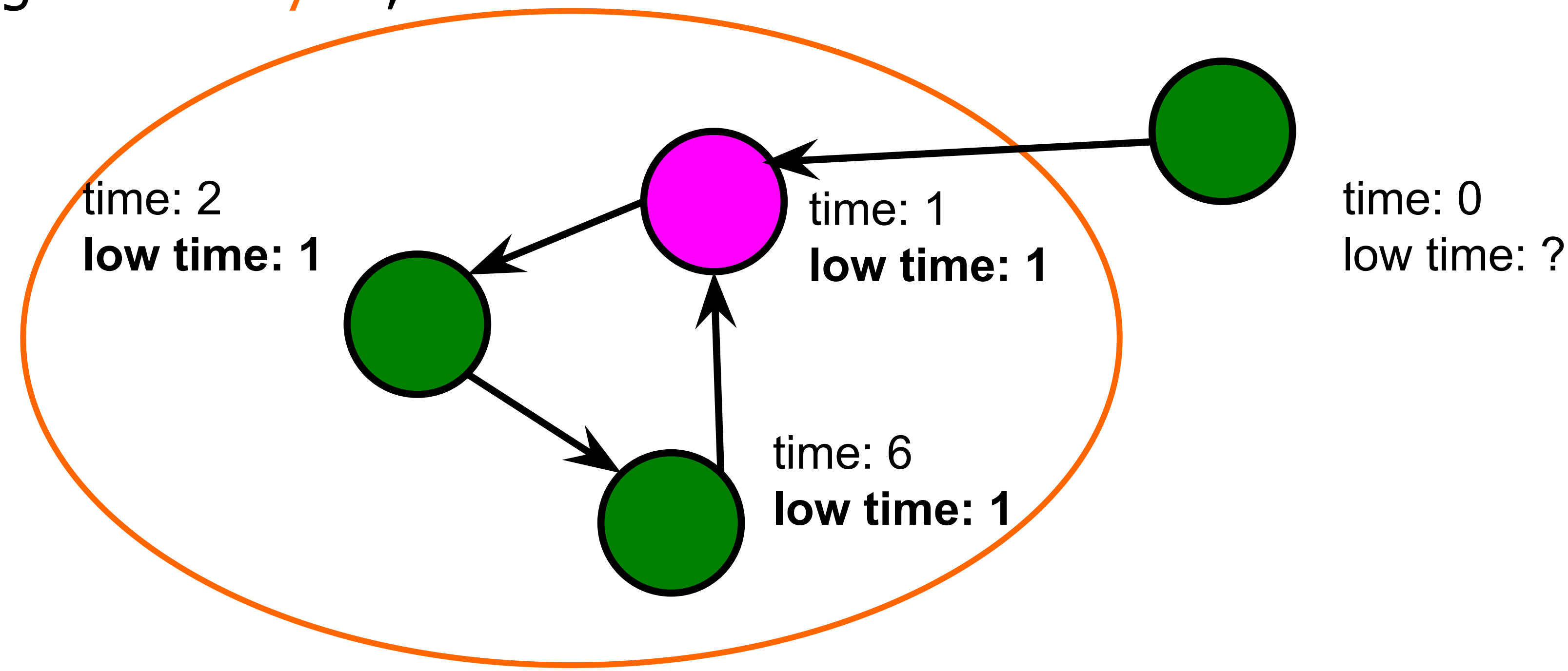
Cycle Finding

Clarification:

What's the point of low time?

The point of the low time is to identify the "earliest" node that we can visit (that is actively being visited).

E.g. In this cycle, all the nodes have low time = 1



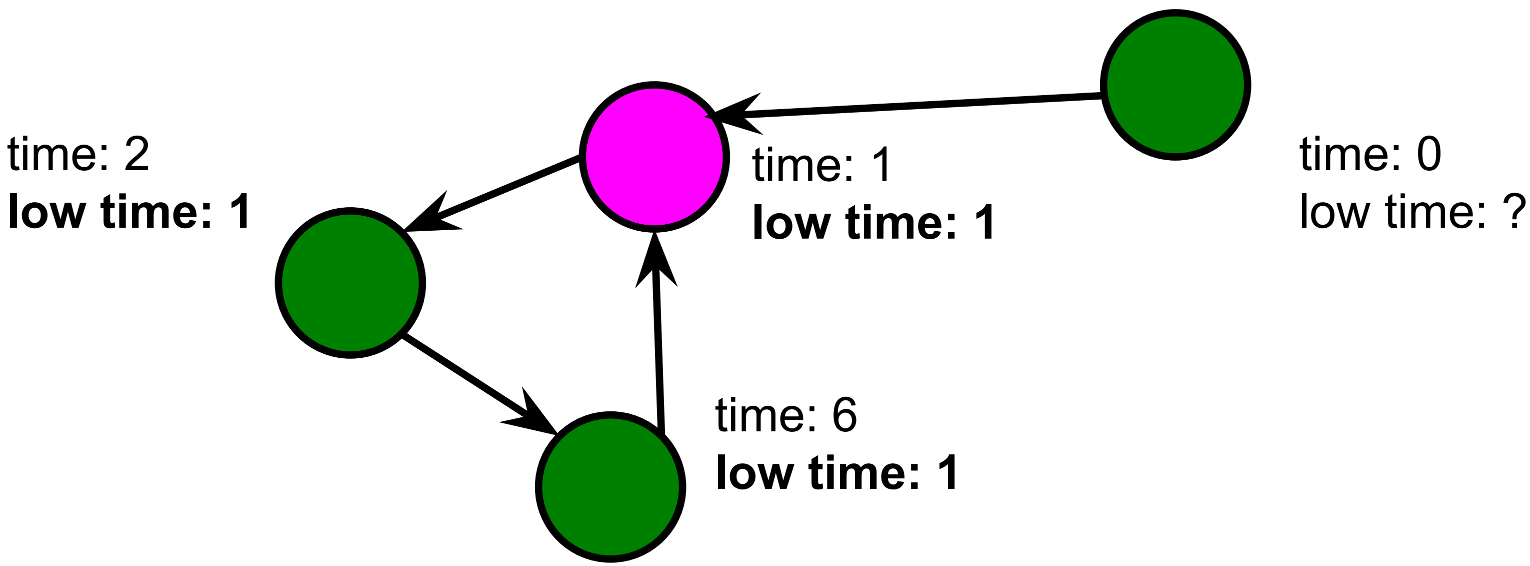
Cycle Finding

Clarification:

What's the point of low time?

The point of the low time is to identify the "earliest" node that we can visit (that is actively being visited).

Want to show: we set our low times correctly.



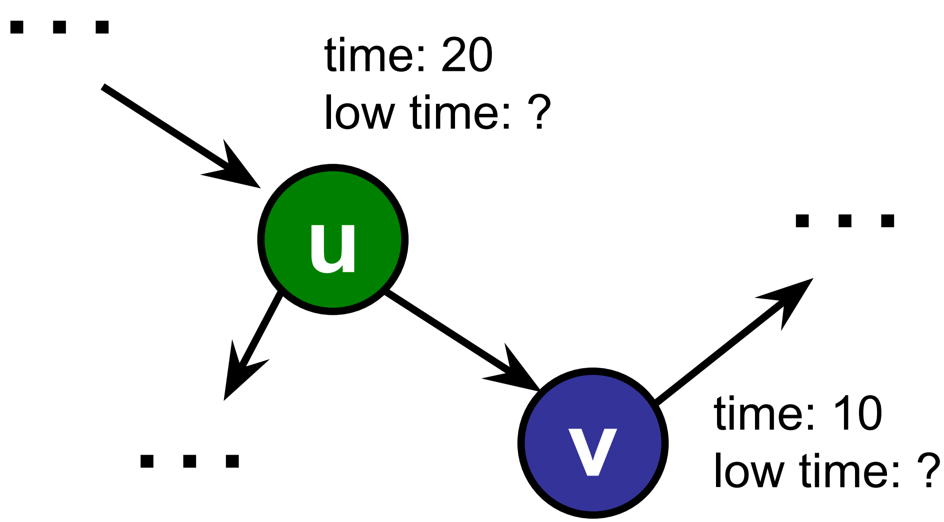
Cycle Finding

Clarification:

If we are at a node **u**, we look at our neighbour **v**, what does **v** tell us about our low time?

Case 1: **v**'s visited already (therefore its time is already set).

Either **v**'s low time is set or it isn't

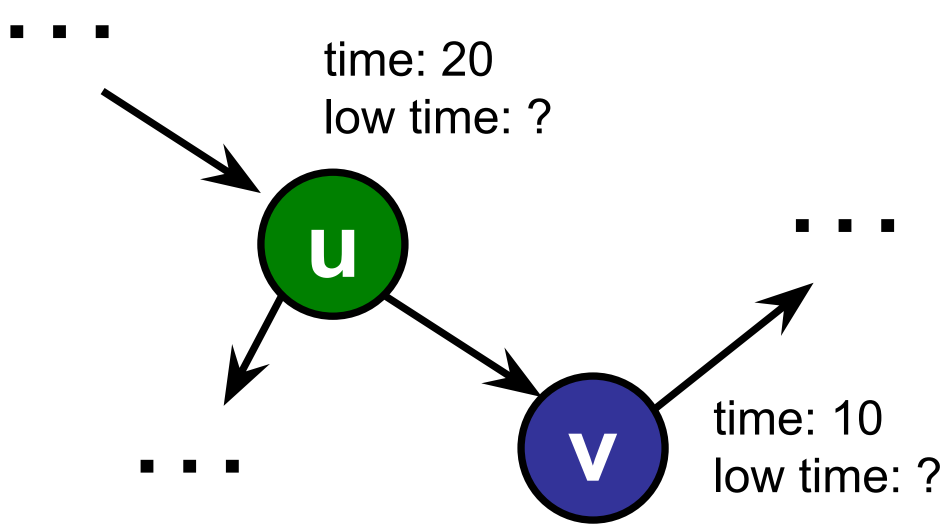


Cycle Finding

Clarification:

If we are at a node **u**, we look at our neighbour **v**, what does **v** tell us about our low time?

Case 1a: **v**'s visited already and low time is not set yet.

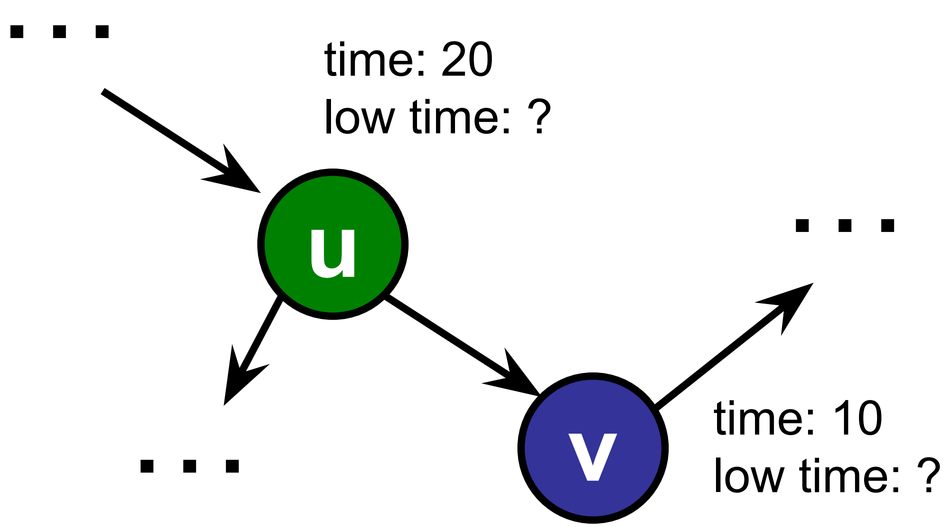


Cycle Finding

Clarification:

If we are at a node **u**, we look at our neighbour **v**, what does **v** tell us about our low time?

Case 1a: **v**'s visited already and low time is not set yet.
Since we only set low time at the end of the traversal, (**v** is still being traversed)/(**v** is still part of the recursion).

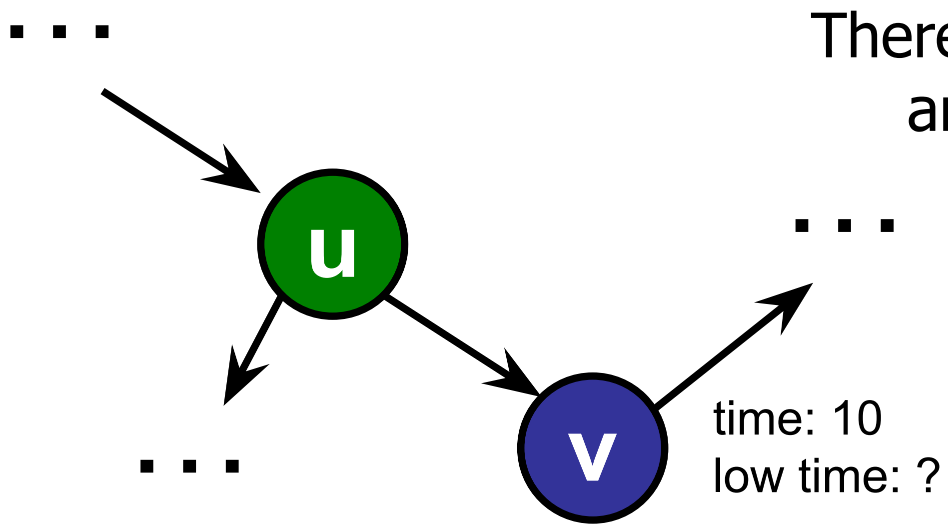


Cycle Finding

Clarification:

If we are at a node **u**, we look at our neighbour **v**, what does **v** tell us about our low time?

Case 1a: **v**'s visited already and low time is not set yet.
Since we only set low time at the end of the traversal, (**v** is still being traversed)/(**v** is still part of the recursion).



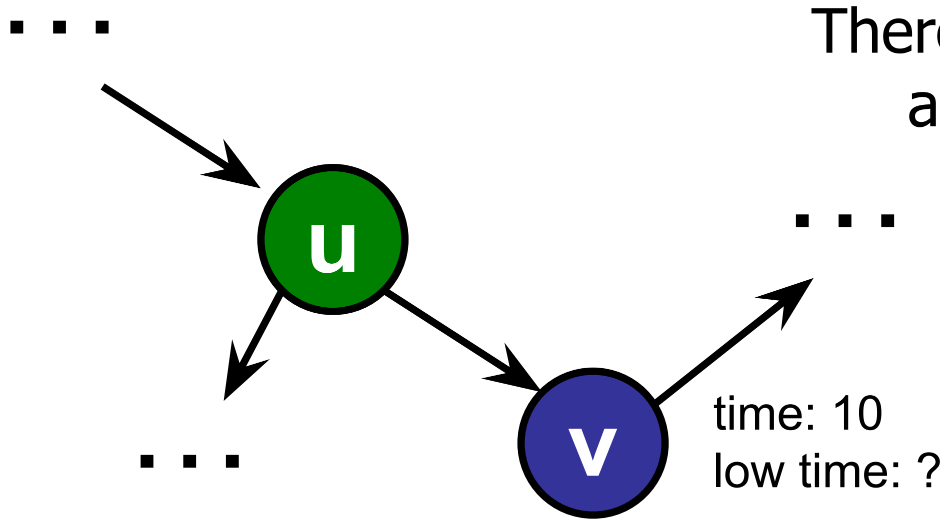
Therefore, **u** reaches **v** and **v** reaches **u**.

Cycle Finding

Clarification:

If we are at a node **u**, we look at our neighbour **v**, what does **v** tell us about our low time?

Case 1a: **v**'s visited already and low time is not set yet.
Since we only set low time at the end of the traversal, (**v** is still being traversed)/(**v** is still part of the recursion).



Therefore, **u** reaches **v** and **v** reaches **u**.

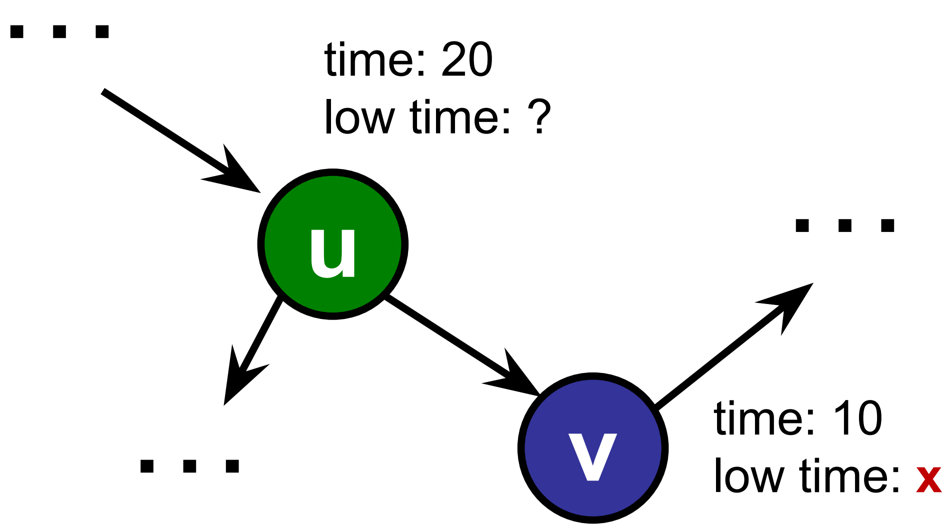
This means that node **u** needs to consider **v**'s time as a potential low time.

Cycle Finding

Clarification:

If we are at a node **u**, we look at our neighbour **v**, what does **v** tell us about our low time?

Case 1b: **v**'s visited already and low time is already set.

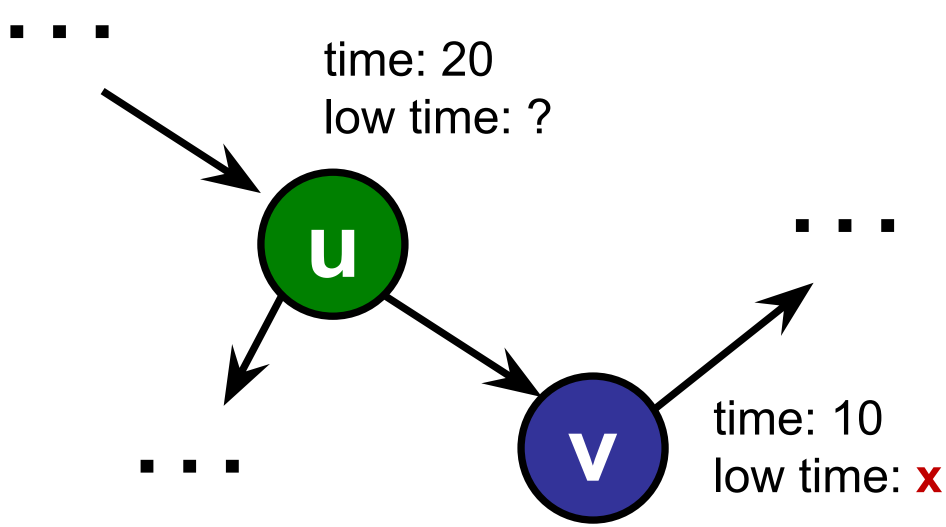


Cycle Finding

Clarification:

If we are at a node **u**, we look at our neighbour **v**, what does **v** tell us about our low time?

Case 1b: **v**'s visited already and low time is already set.
Since we only set low time at the end of the traversal, (**v** is no longer being traversed)/(**v** is not part of the recursion).

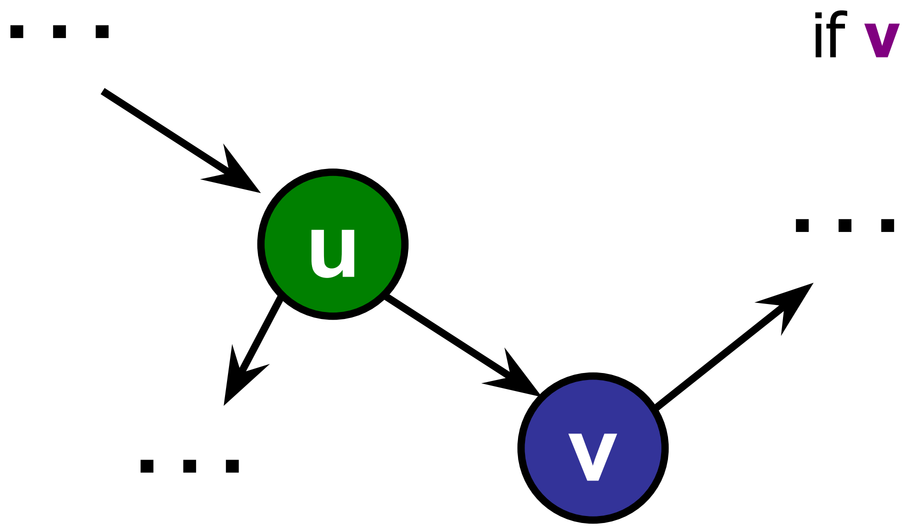


Cycle Finding

Clarification:

If we are at a node **u**, we look at our neighbour **v**, what does **v** tell us about our low time?

Case 1b: **v**'s visited already and low time is already set.
Since we only set low time at the end of the traversal, (**v** is no longer being traversed)/(**v** is not part of the recursion).



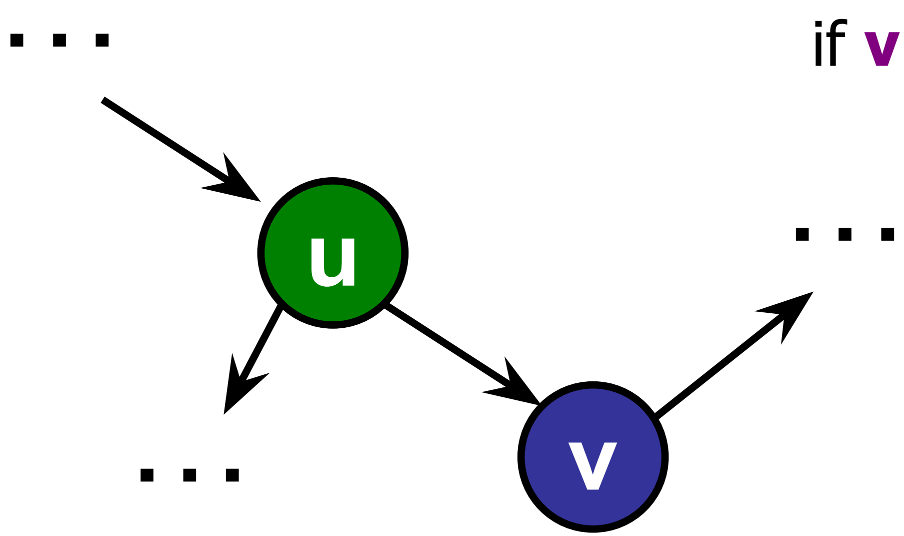
if **v** could have reached us, we would not be visiting **v** after it was done recursing.

Cycle Finding

Clarification:

If we are at a node **u**, we look at our neighbour **v**, what does **v** tell us about our low time?

Case 1b: **v**'s visited already and low time is already set.
Since we only set low time at the end of the traversal, (**v** is no longer being traversed)/(**v** is not part of the recursion).



if **v** could have reached us, we would not be visiting **v** after it was done recursing.

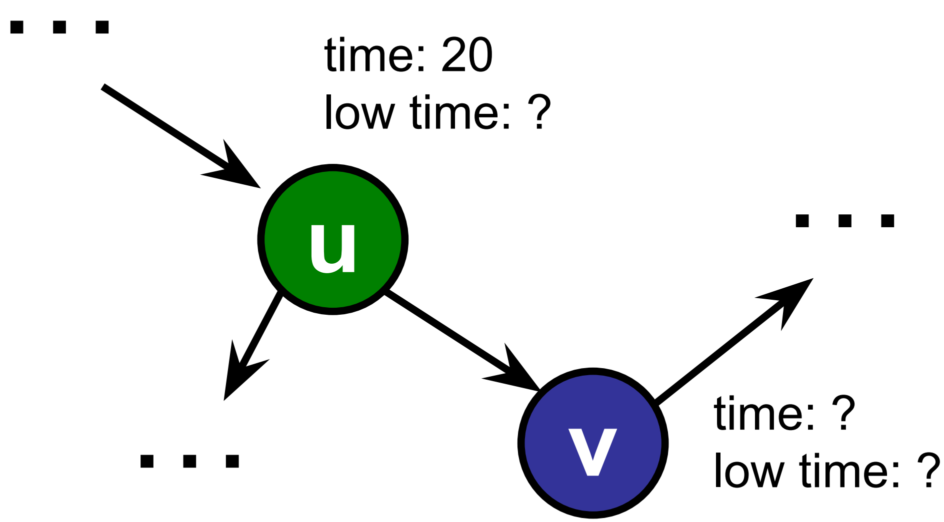
So we don't consider taking **v**'s low time.

Cycle Finding

Clarification:

If we are at a node **u**, we look at our neighbour **v**, what does **v** tell us about our low time?

Case 2: **v**'s not visited yet
(therefore its time and low time is not set).



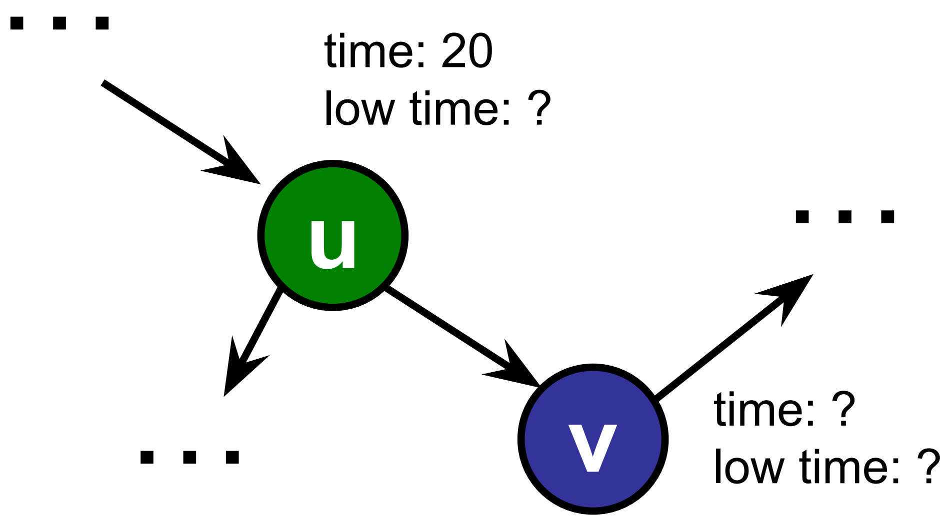
Cycle Finding

Clarification:

If we are at a node **u**, we look at our neighbour **v**, what does **v** tell us about our low time?

Case 2: **v**'s not visited yet
(therefore its time and low time is not set).

So we will recurse on **v** to compute its time and low time.



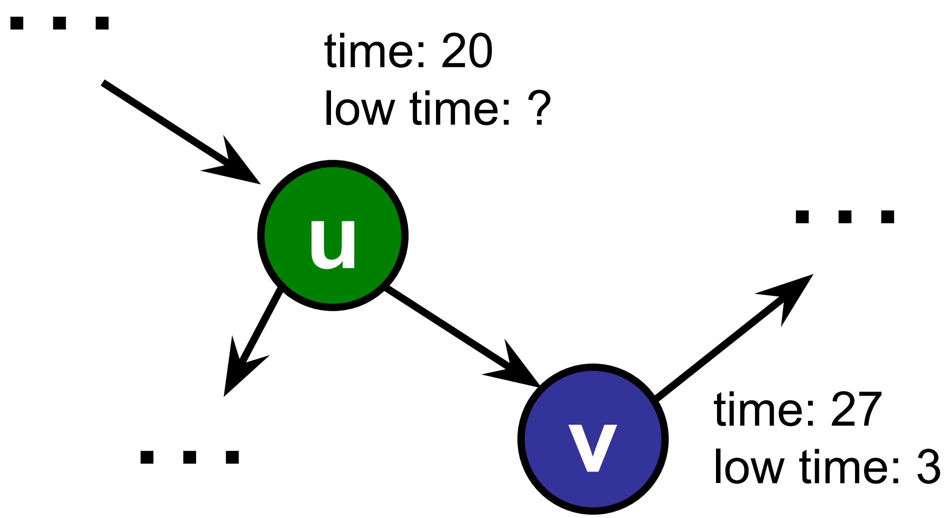
Cycle Finding

Clarification:

Case 2: **v**'s not visited yet (therefore its time and low time is not set).

So we will recurse on **v** to compute its time and low time.

If after that, its **v**'s low time is smaller than **u**'s time,



Cycle Finding

Clarification:

Case 2: **v**'s not visited yet (therefore its time and low time is not set).

So we will recurse on **v** to compute its time and low time.

If after that, its **v**'s low time is smaller than **u**'s time,

Cycle Finding

Clarification:

Case 2: **v**'s not visited yet (therefore its time and low time is not set).

So we will recurse on **v** to compute its time and low time.

If after that, its **v**'s low time is smaller than **u**'s time,

This means that **v** reached a node **w** that was actively being visited.

Cycle Finding

Clarification:

Case 2: **v**'s not visited yet (therefore its time and low time is not set).

So we will recurse on **v** to compute its time and low time.

If after that, its **v**'s low time is smaller than **u**'s time,

This means that **v** reached a node **w** that was actively being visited.

notice that:

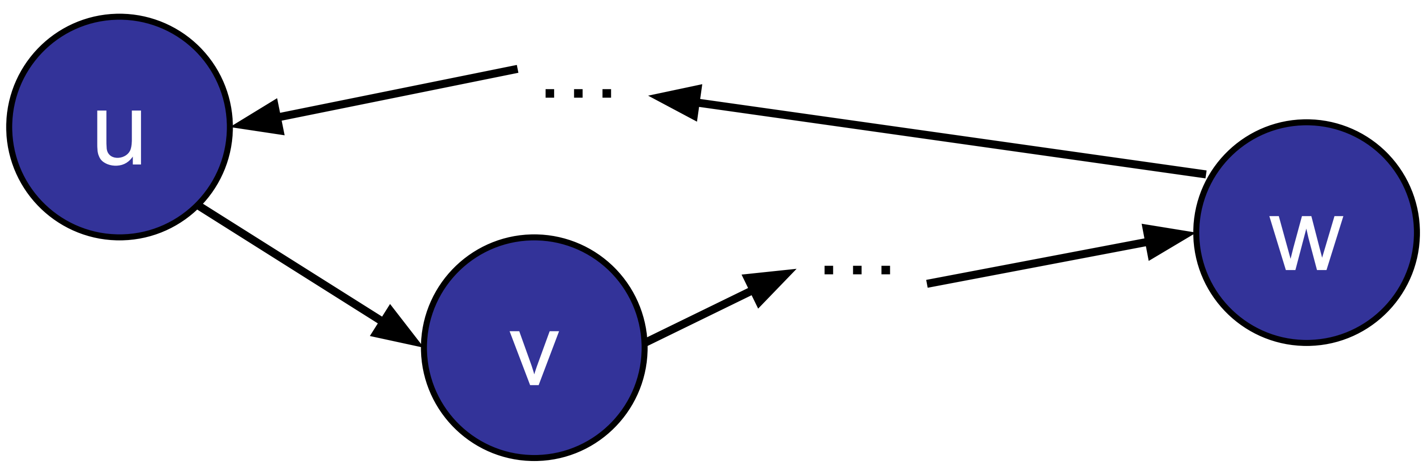
w's time = **v**'s low time < **u**'s time < **v**'s time

Cycle Finding

Clarification:

Case 2: **v**'s not visited yet (therefore its time and low time is not set).

w's time = **v**'s low time < **u**'s time < **v**'s time



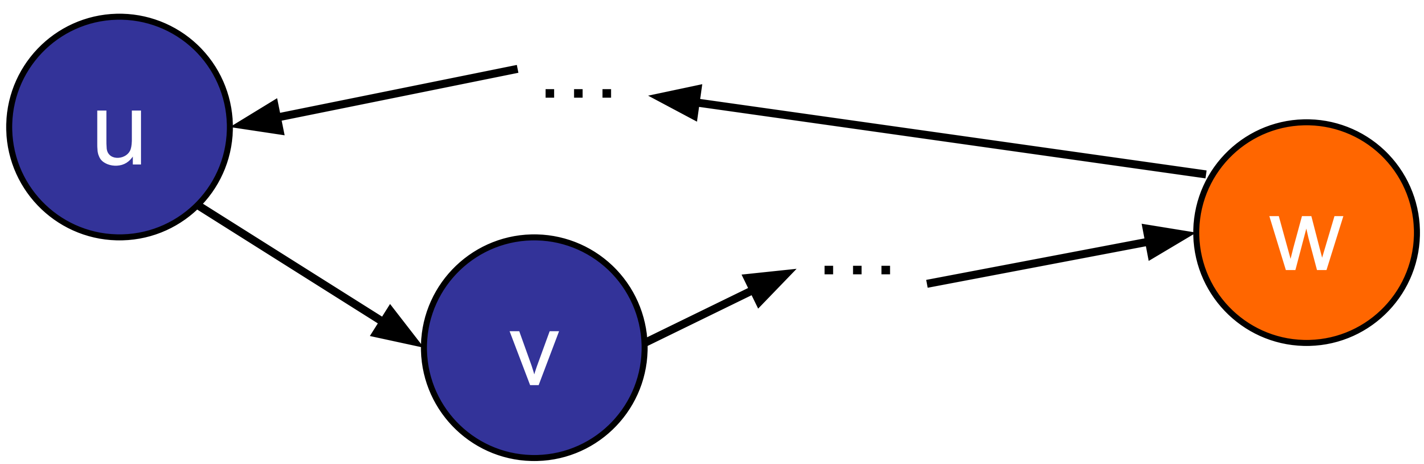
Cycle Finding

Clarification:

Case 2: **v**'s not visited yet (therefore its time and low time is not set).

w's time = **v**'s low time < **u**'s time < **v**'s time

So **v** can reach **w**. And **w** was part of the current traversal.



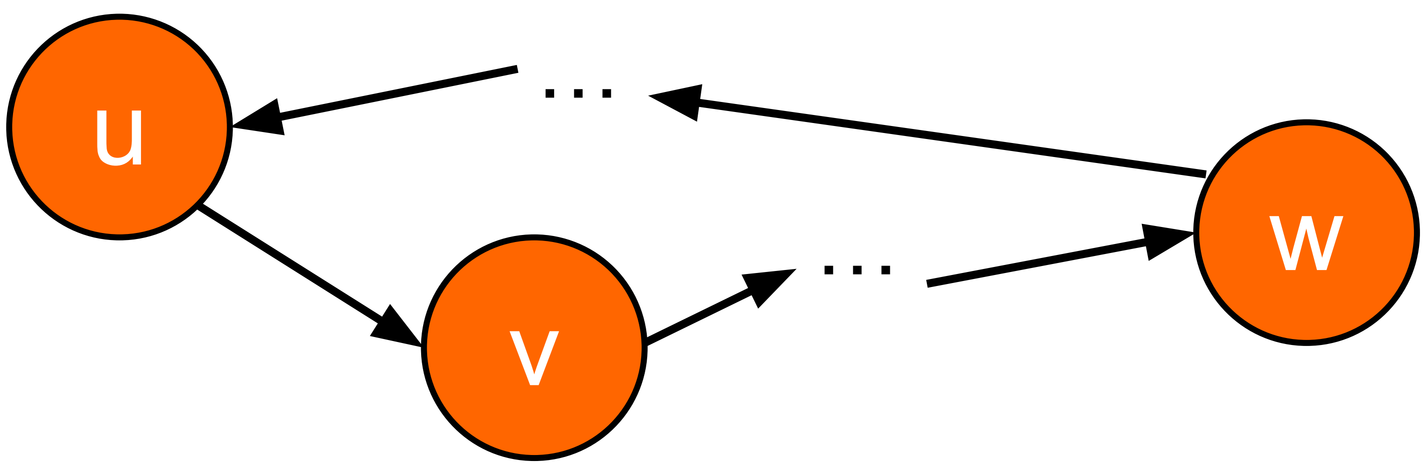
Cycle Finding

Clarification:

Case 2: **v**’s not visited yet (therefore its time and low time is not set).

w’s time = **v**’s low time < **u**’s time < **v**’s time

So **v** can reach **w**. And **w** was part of the current traversal. And both **u** and **v**, were also being traversed when **v** reached **w**.



Cycle Finding

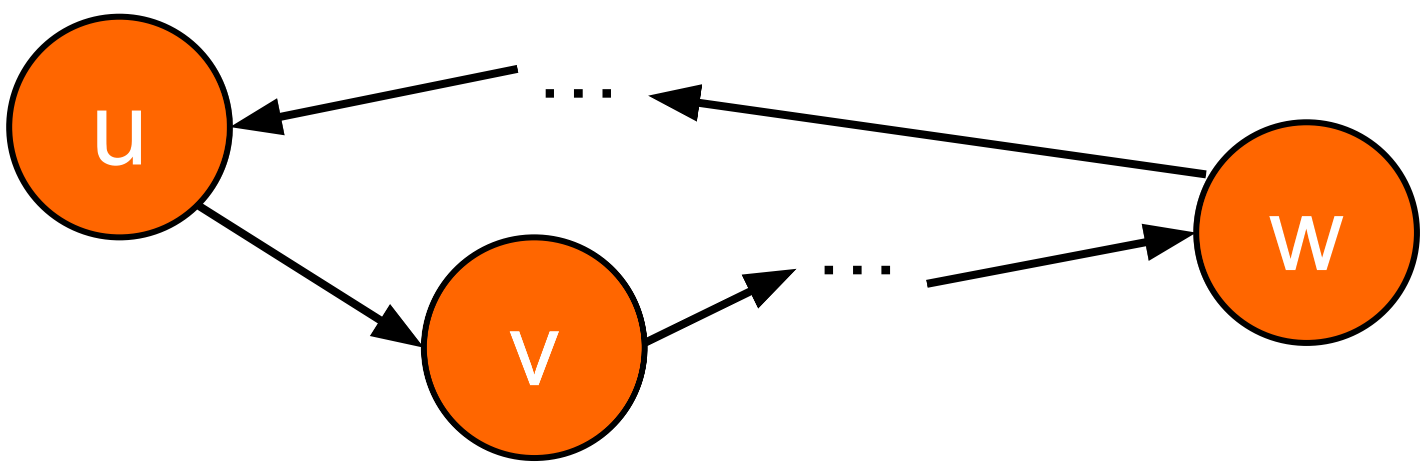
Clarification:

Case 2: **v**'s not visited yet (therefore its time and low time is not set).

w's time = **v**'s low time < **u**'s time < **v**'s time

So **v** can reach **w**. And **w** was part of the current traversal. And both **u** and **v**, were also being traversed when **v** reached **w**.

Since **w**'s time < **u**'s time, this means **w** reaches **v**.



Connected Components

Low time of a node is the minimum of:

1. Its own time
2. Low time of children that we just (visited)/(recursed from)

Compute the low time at the end of the traversal.

Post order traversal!

Connected Components

How does low time help us?

Connected Components

How does low time help us?

Grouping by low time gives us the connected components!

Connected Components

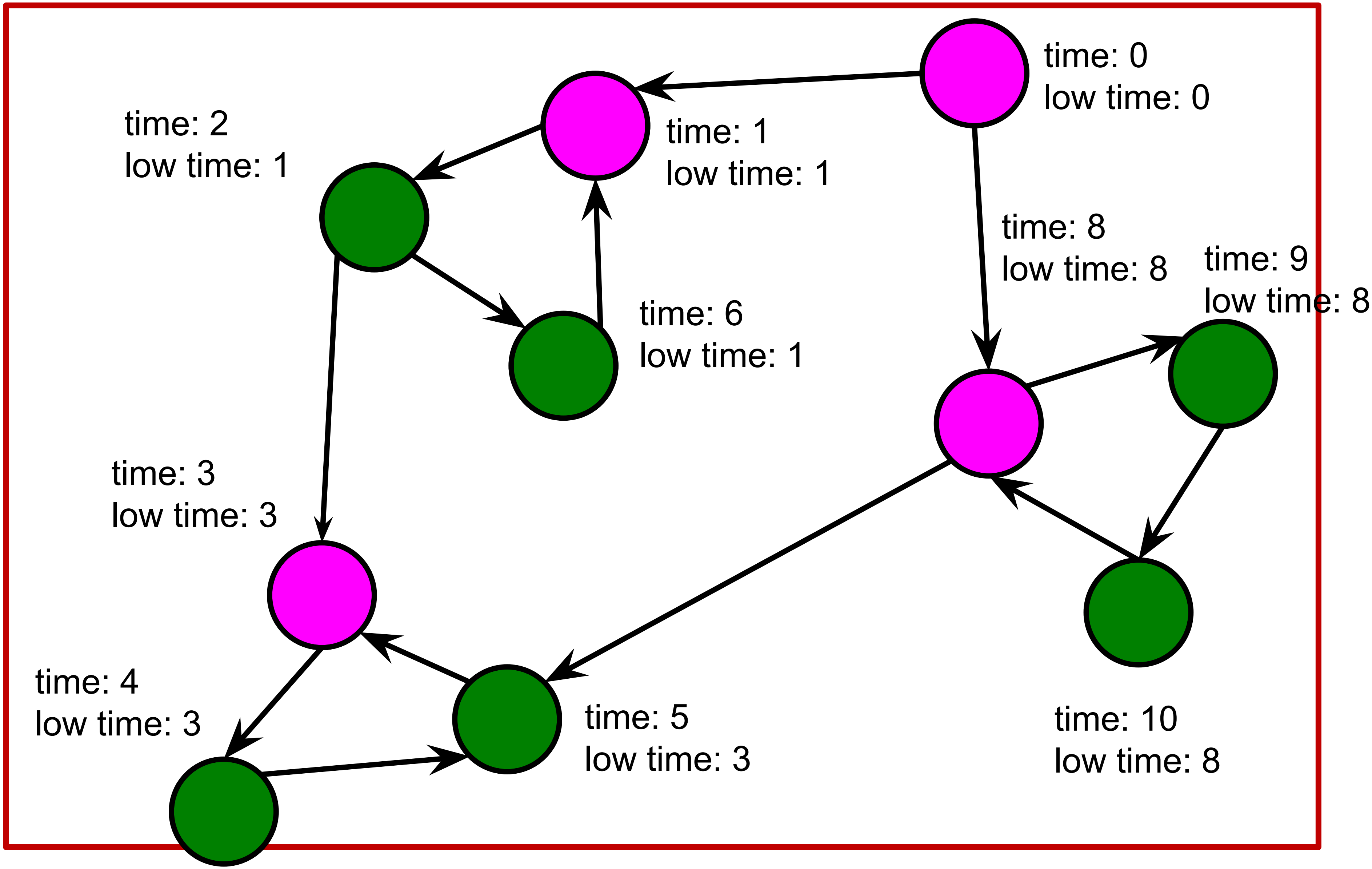
Correction:

How does low time help us?

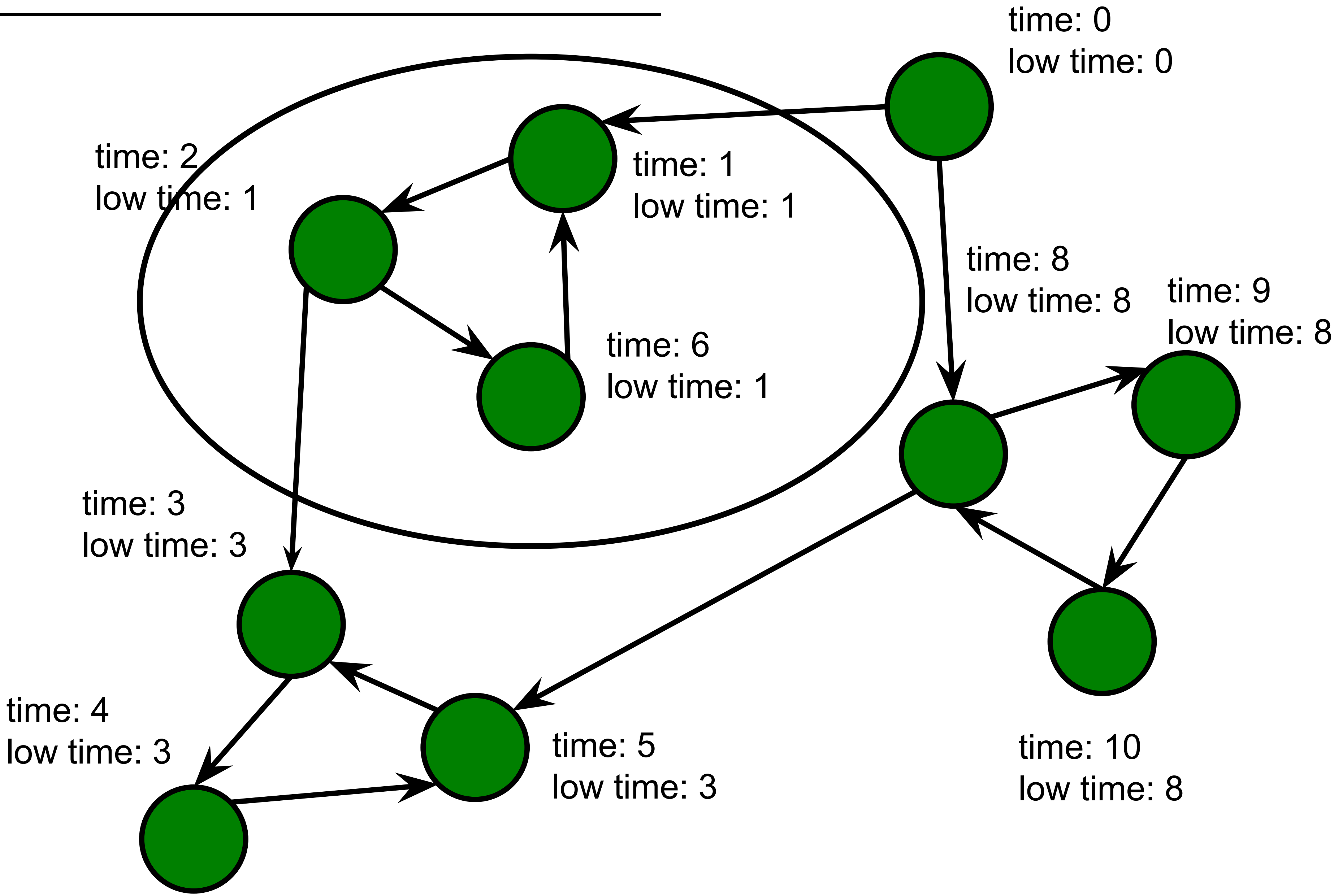
Nodes where their low time = time is are “roots” of their strongly connected components.

Connected Components

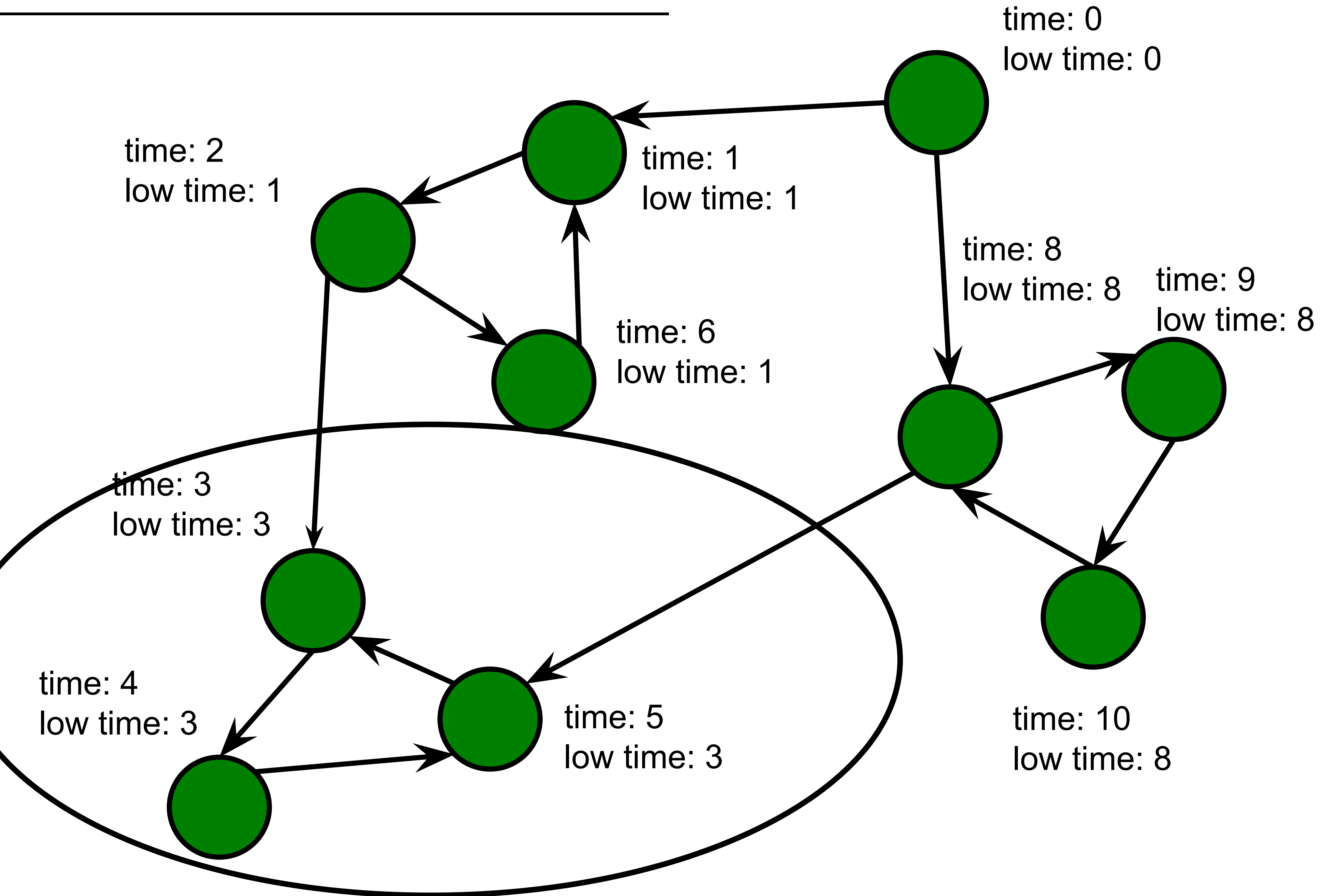
Correction:



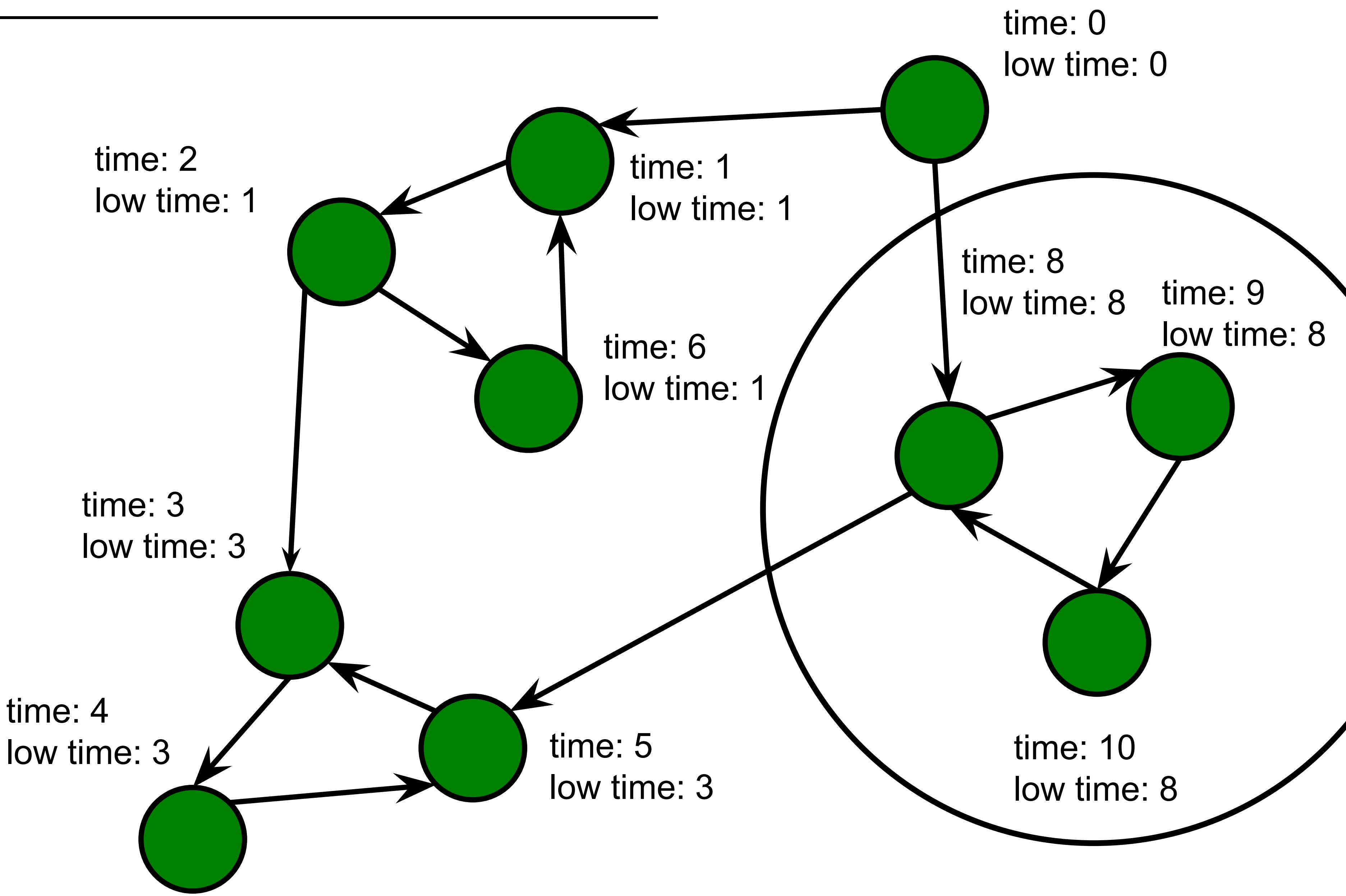
Connected Components



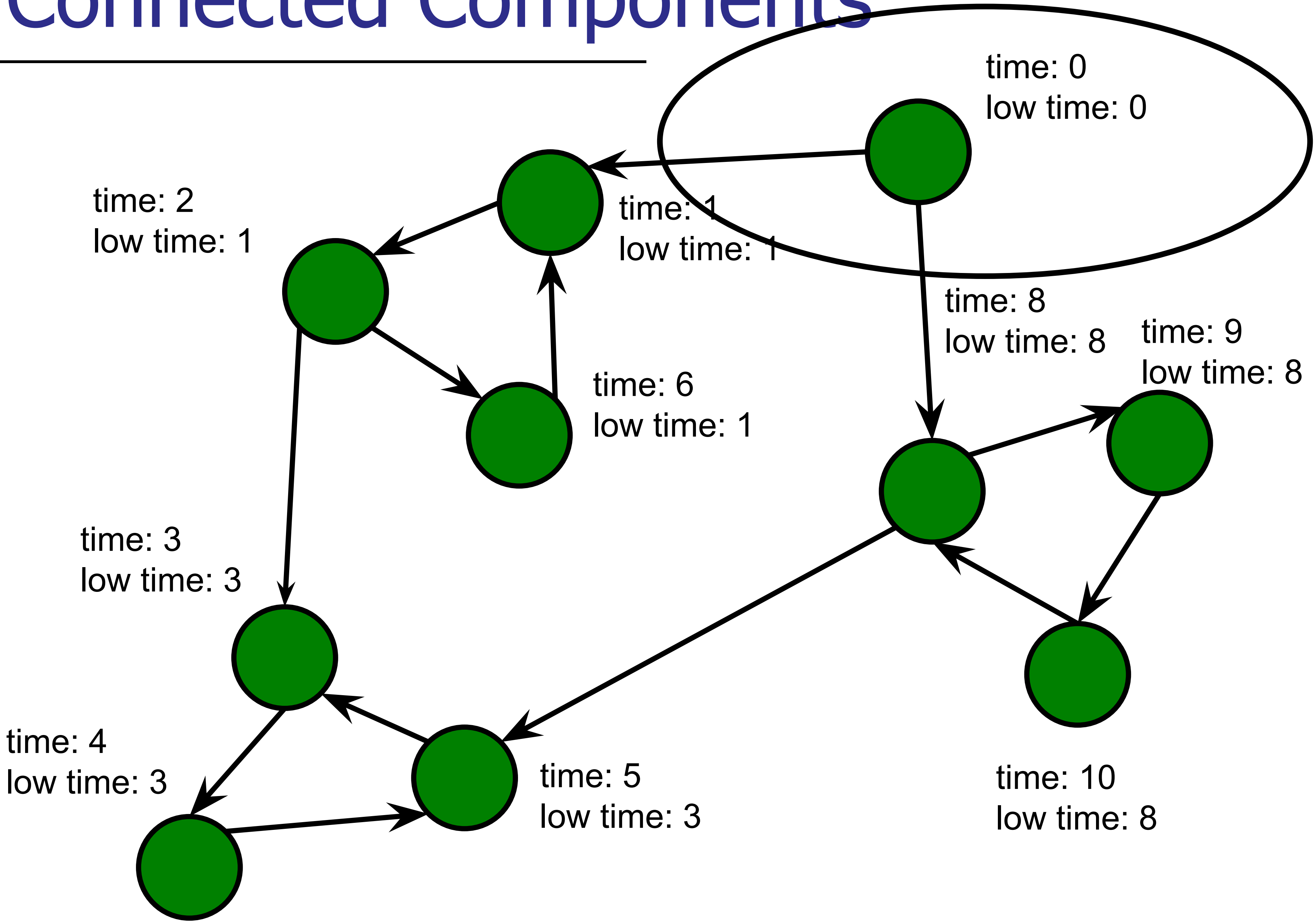
Connected Components



Connected Components



Connected Components

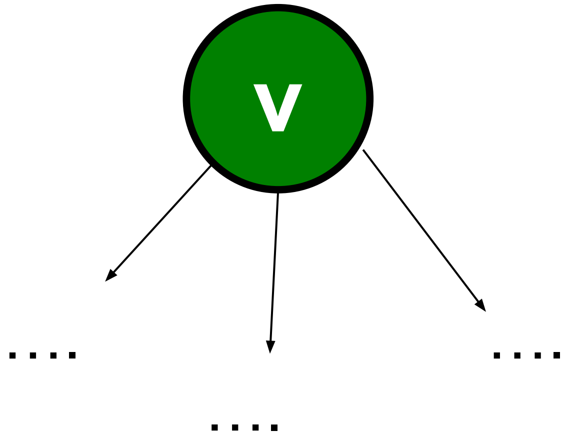


Connected Components

Correction:

Rough Idea:

After we are finished recursing from a node **v**, just before we return, we have processed all nodes **v** could have reached.

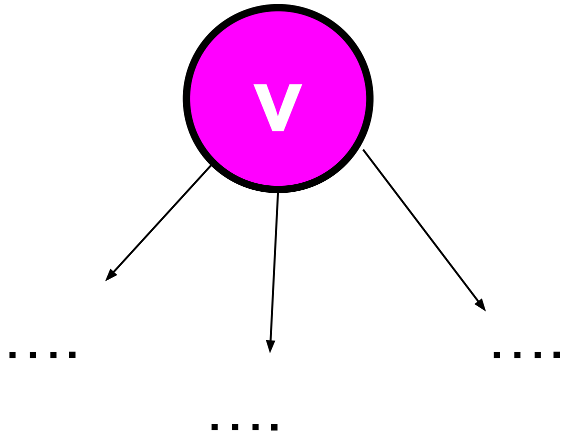


Connected Components

Correction:

Rough Idea:

If node **v**'s time = low time, then **v** is the "root" of an SCC.



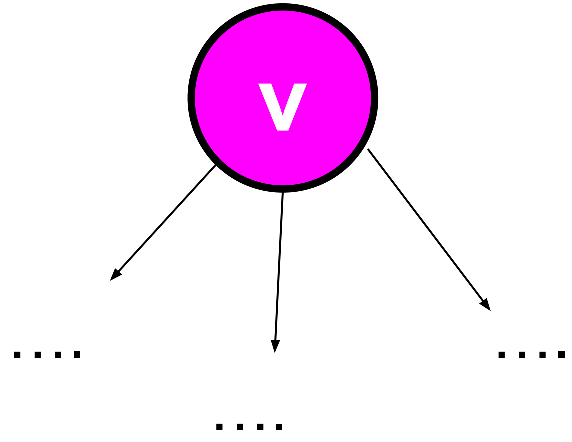
Connected Components

Correction:

Rough Idea:

If node **v**'s time = low time, then **v** is the "root" of an SCC.

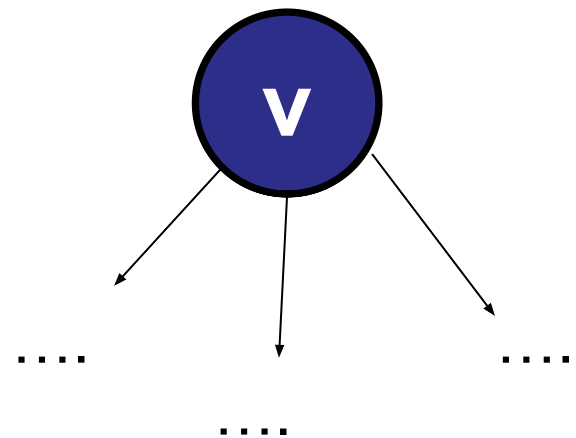
Want to make use of the fact that by the time we are done with **v**, all nodes in the SCC are "ready" to be grouped.



Connected Components

Correction:

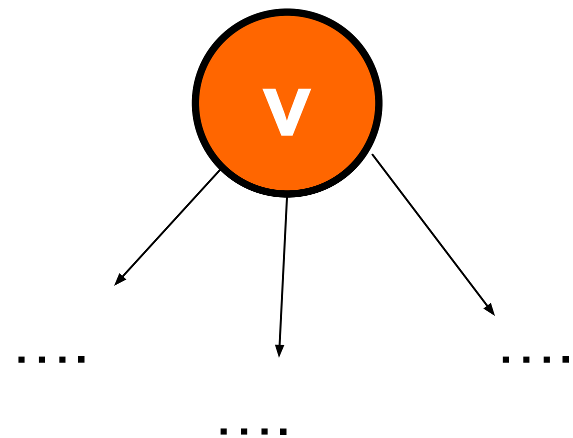
1. When we first visit a node **v**, push it onto a stack.



Connected Components

Correction:

1. When we first visit a node **v**, push it onto a stack.
2. Run the DFS (that tags every node with time/low time)



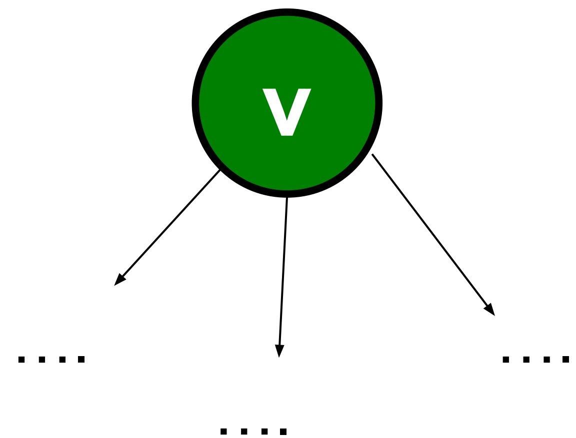
Connected Components

Correction:

1. When we first visit a node **v**, push it onto a stack.
2. Run the DFS (that tags every node with time/low time)
3. After we finish recursing. Before we return,

   if v's time =low time, then repeatedly pop from

   the stack until we pop **v** also.

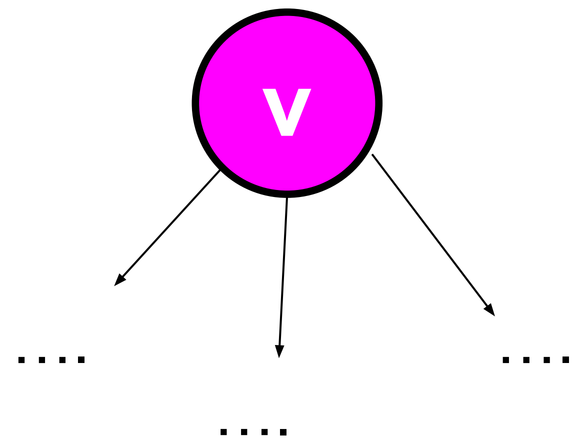


Connected Components

Correction:

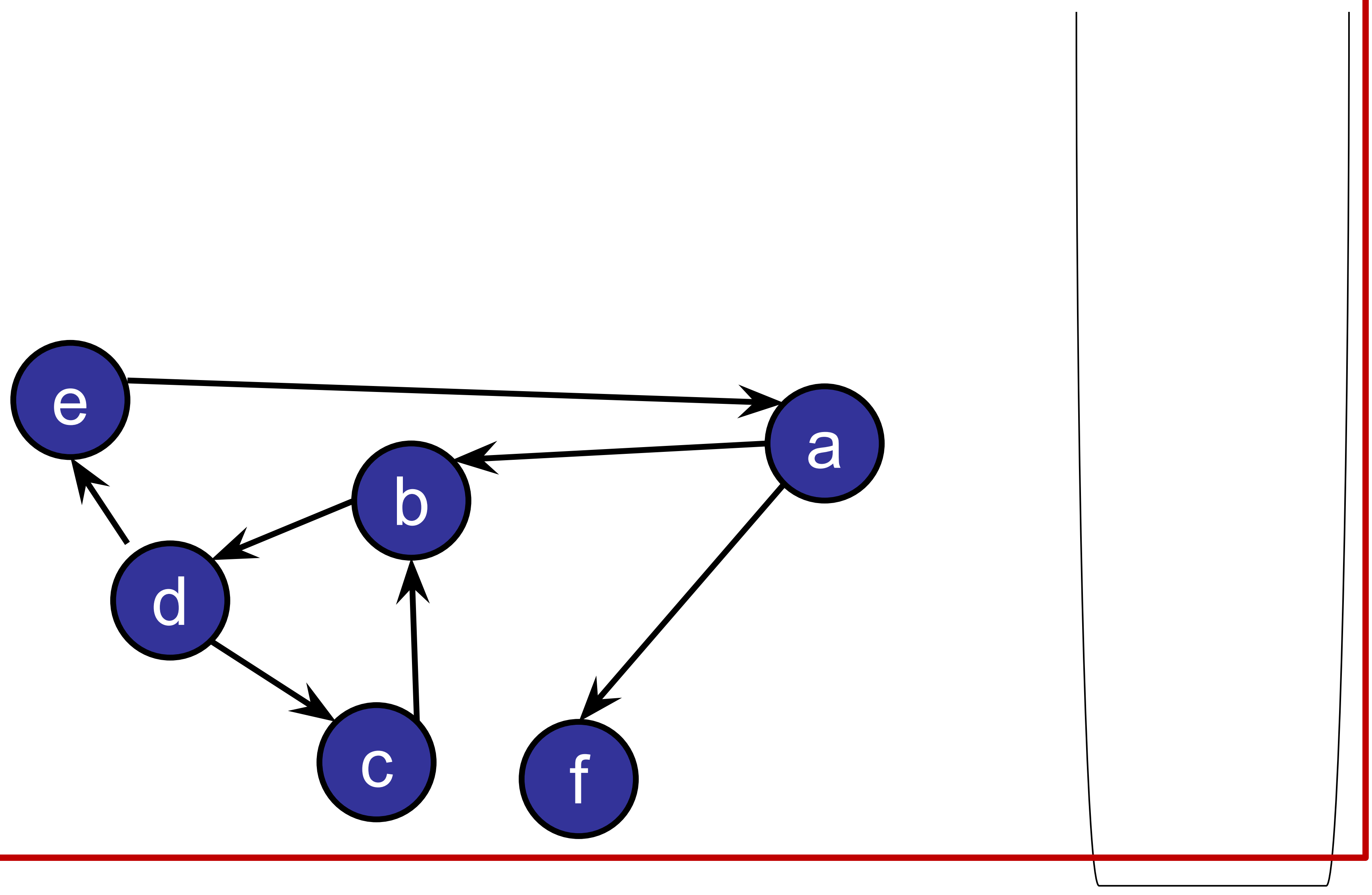
1. When we first visit a node **v**, push it onto a stack.
2. Run the DFS (that tags every node with time/low time)
3. After we finish recursing. Before we return,
   if v's time =low time, then repeatedly pop from
   the stack until we pop **v** also.

Everything we popped is part of the SCC rooted at **v**



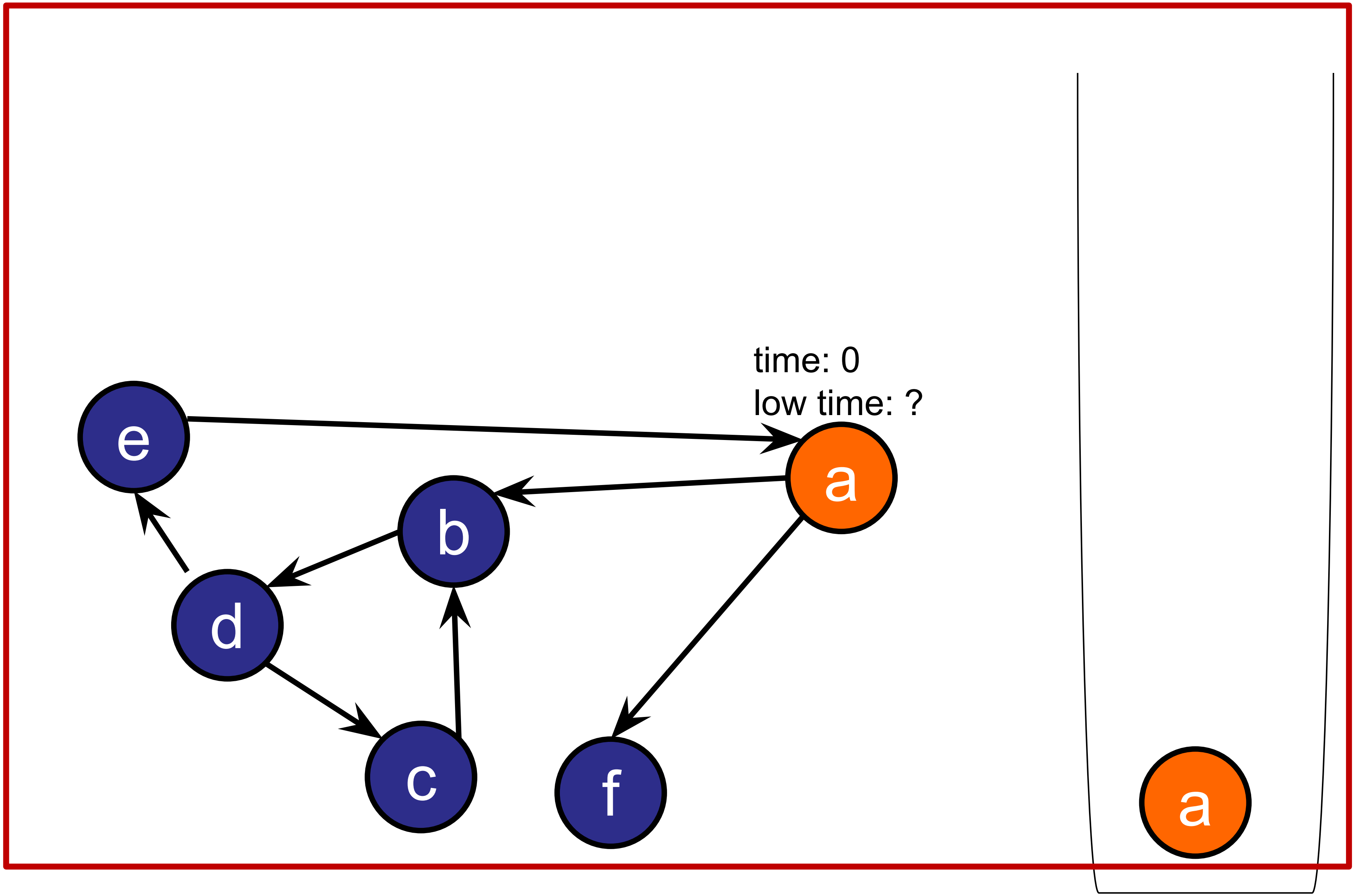
Connected Components

Correction:



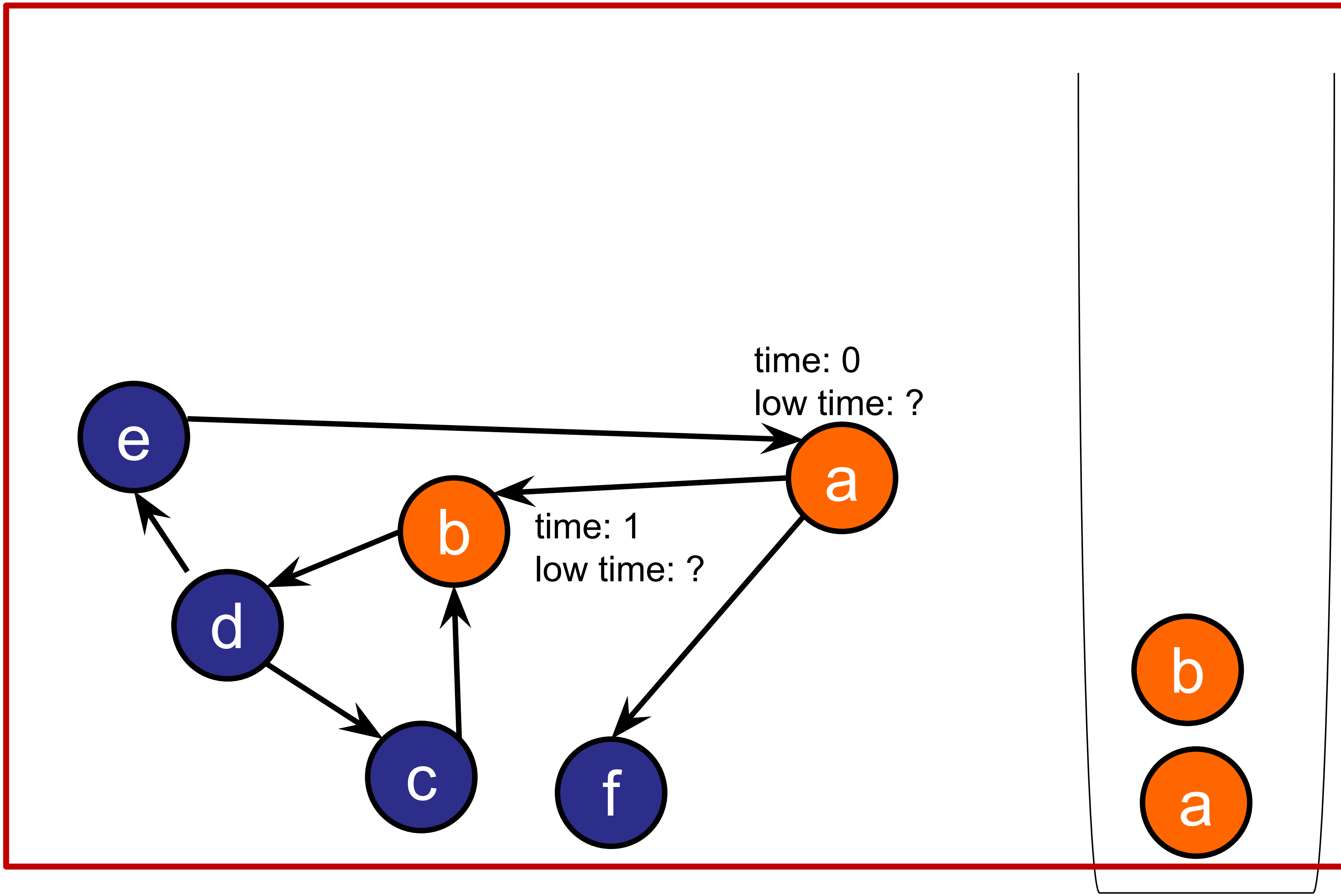
Connected Components

Correction:



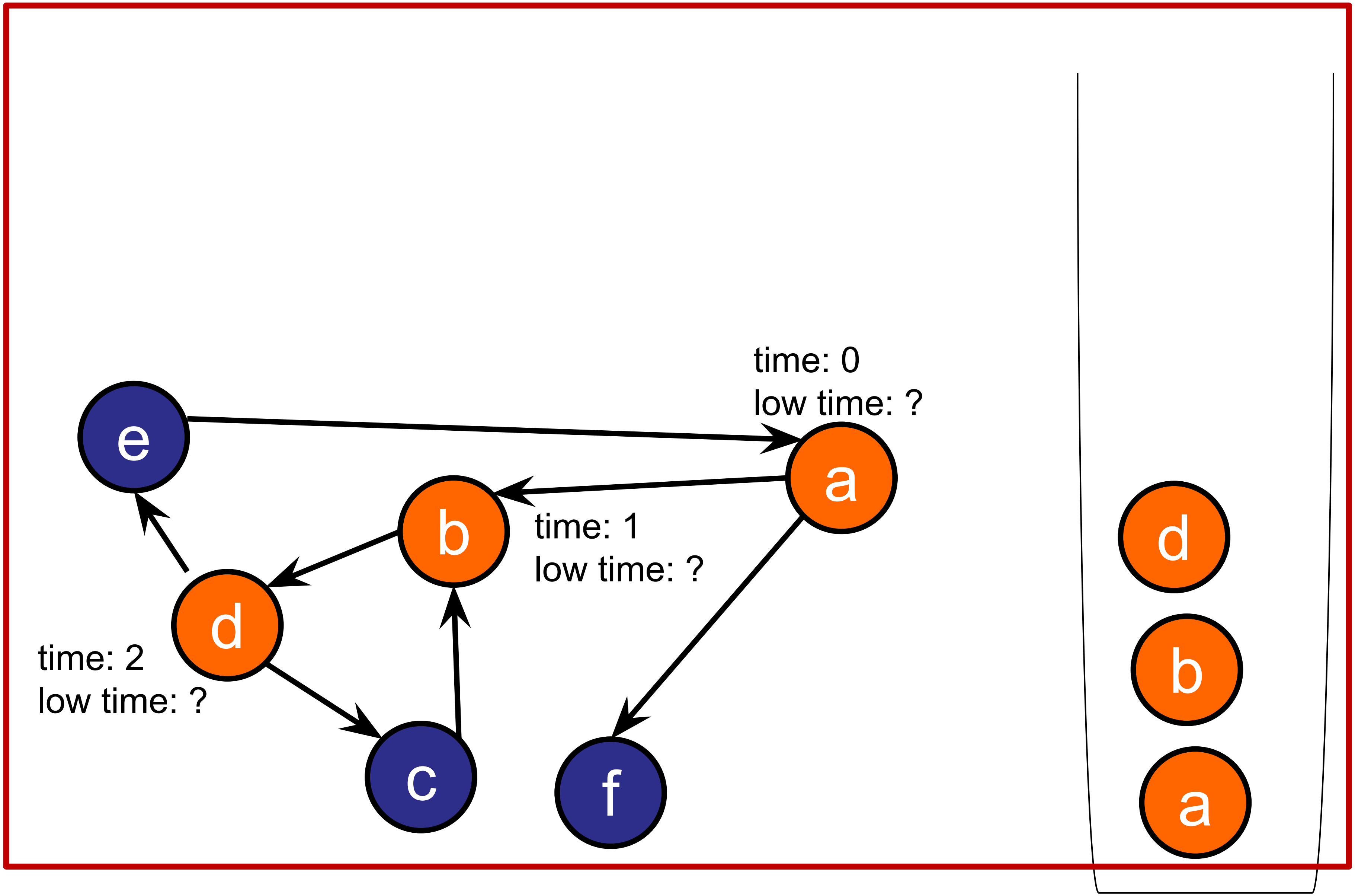
Connected Components

Correction:



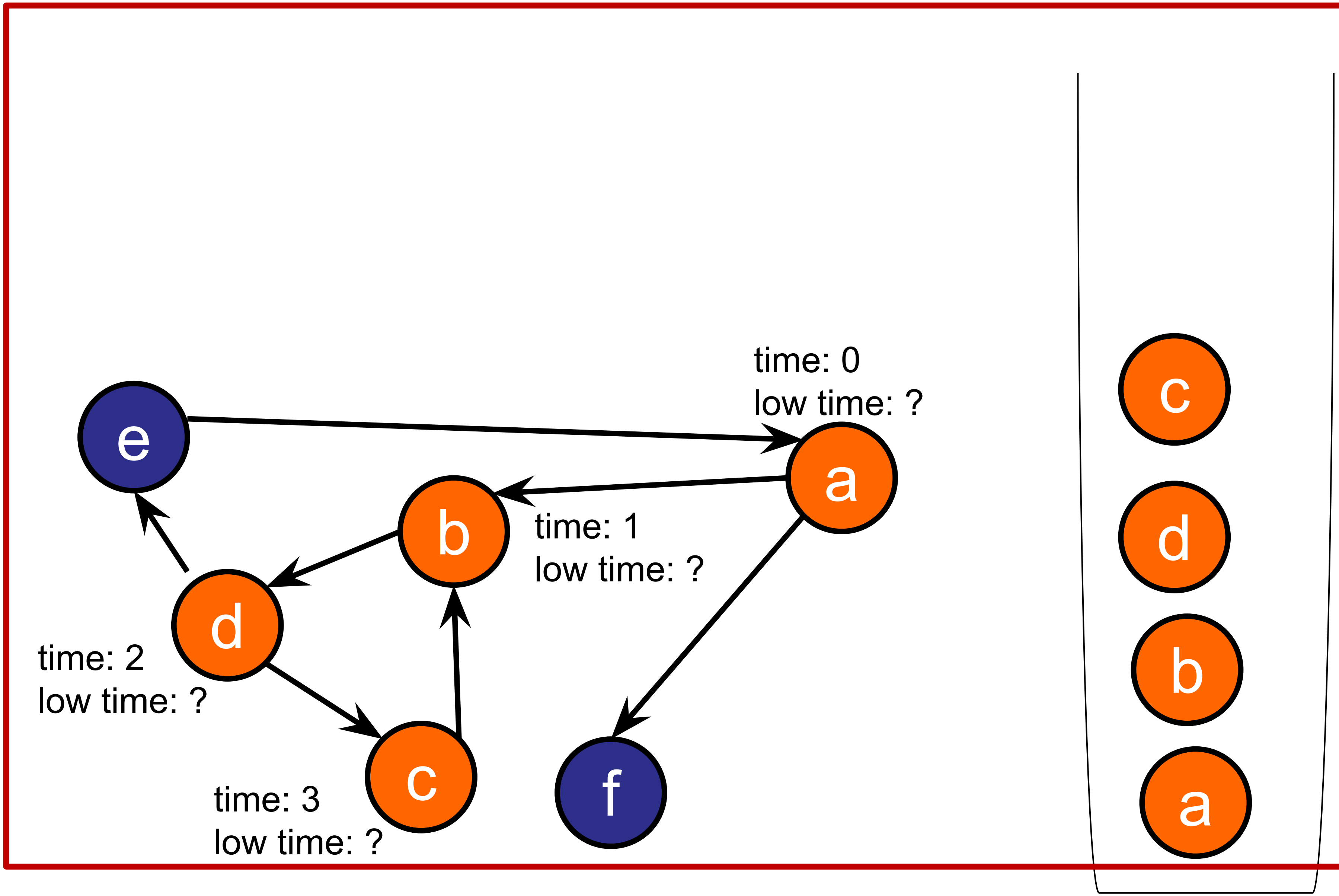
Connected Components

Correction:



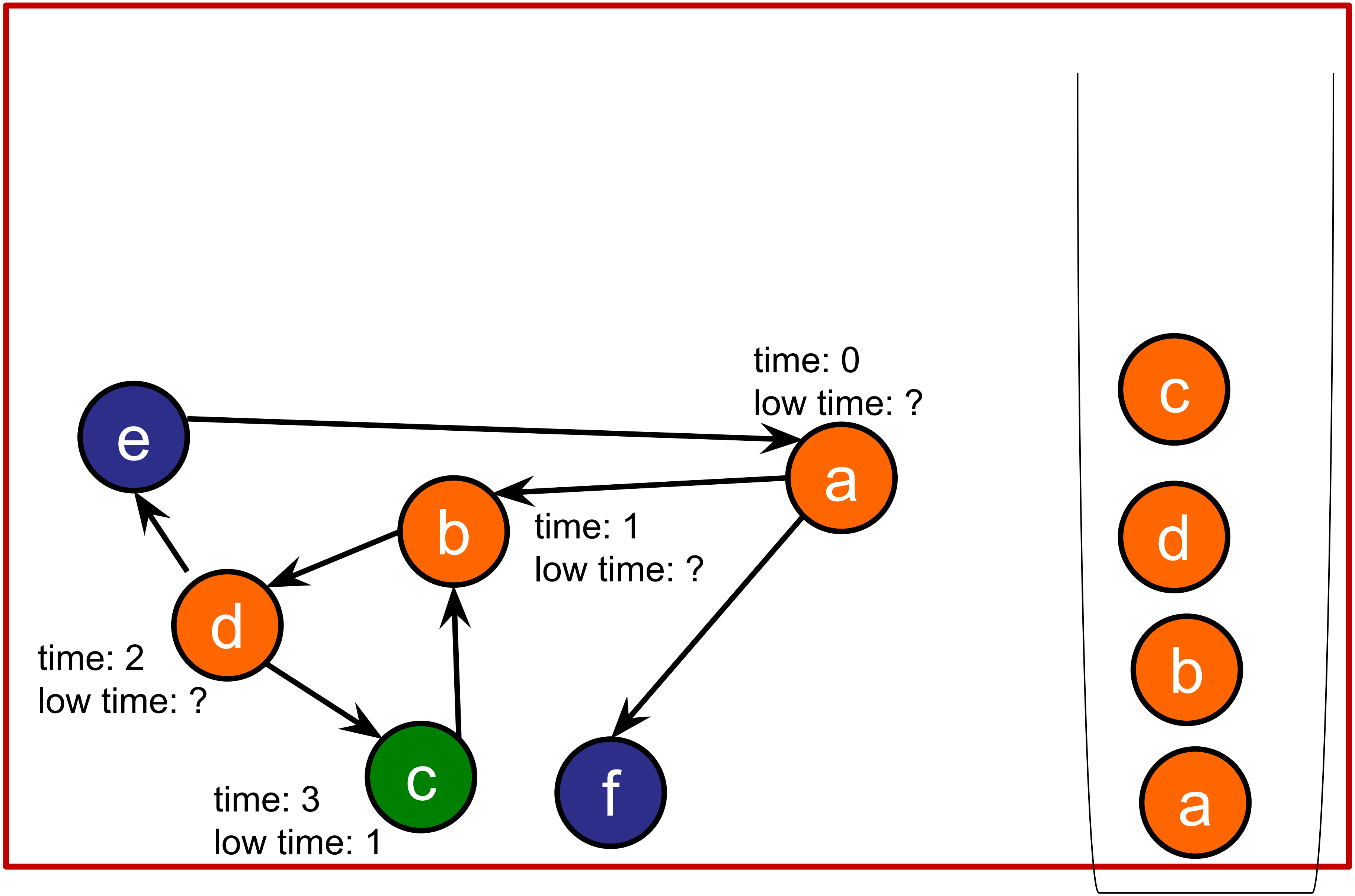
Connected Components

Correction:



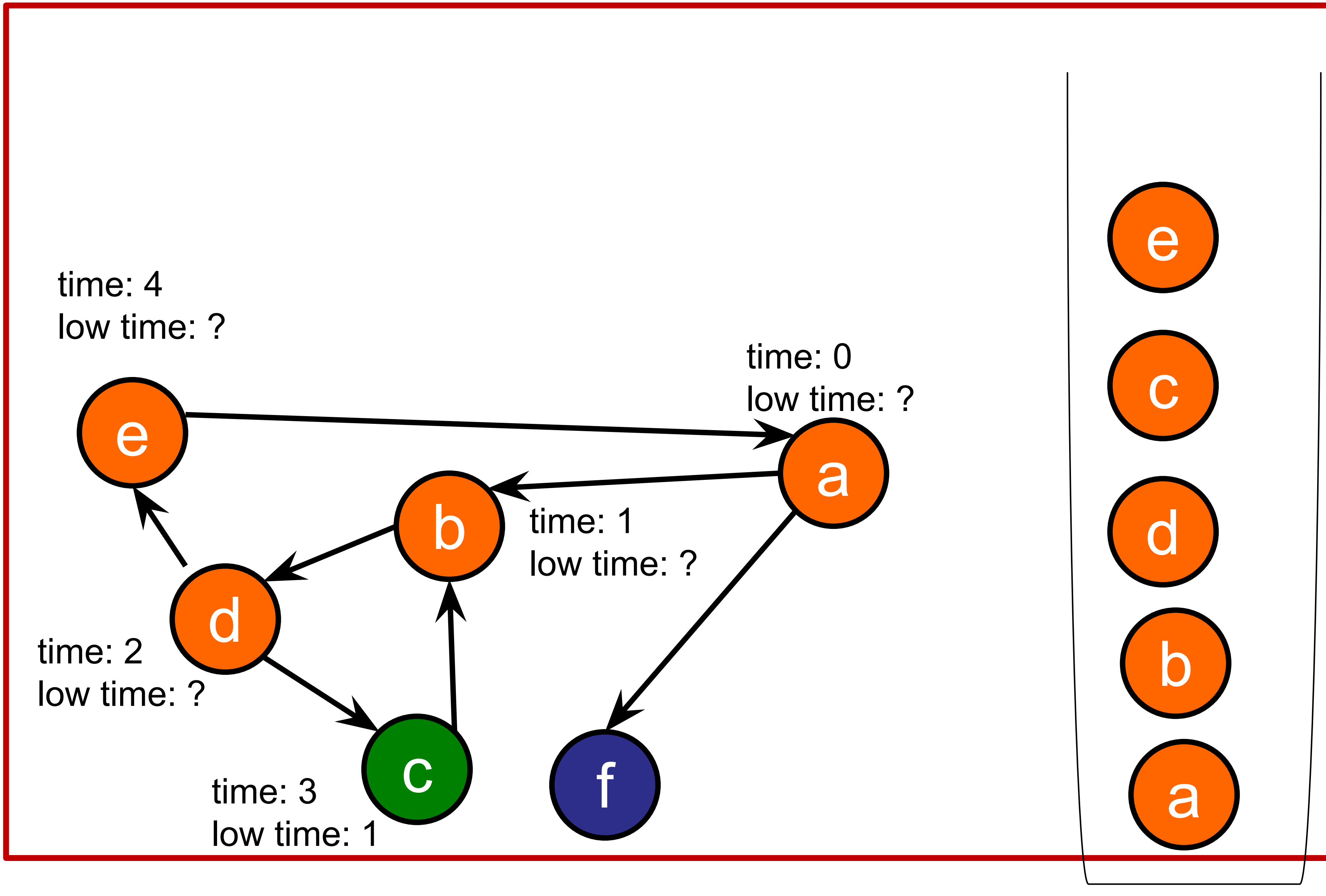
Connected Components

Correction:



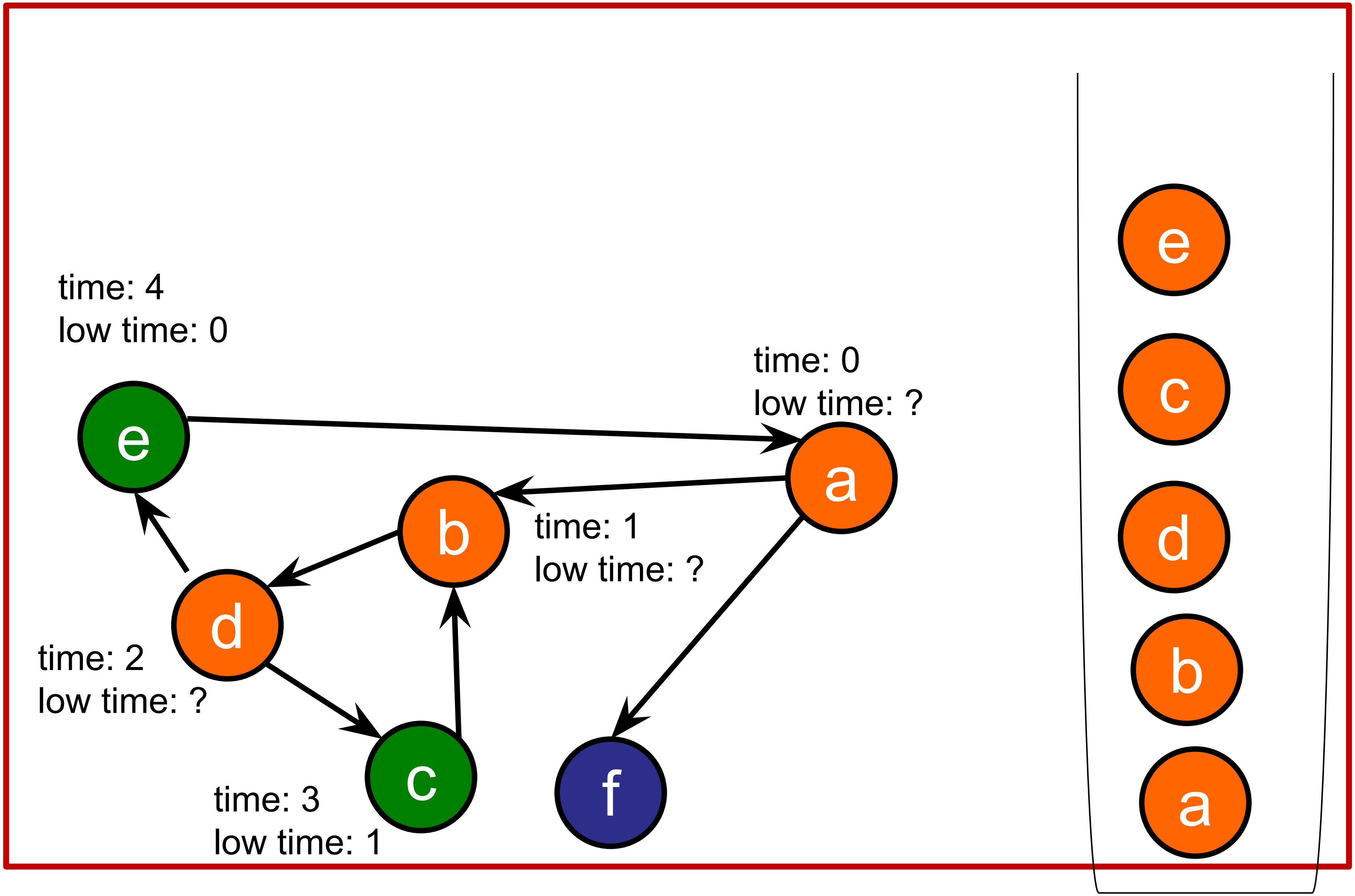
Connected Components

Correction:



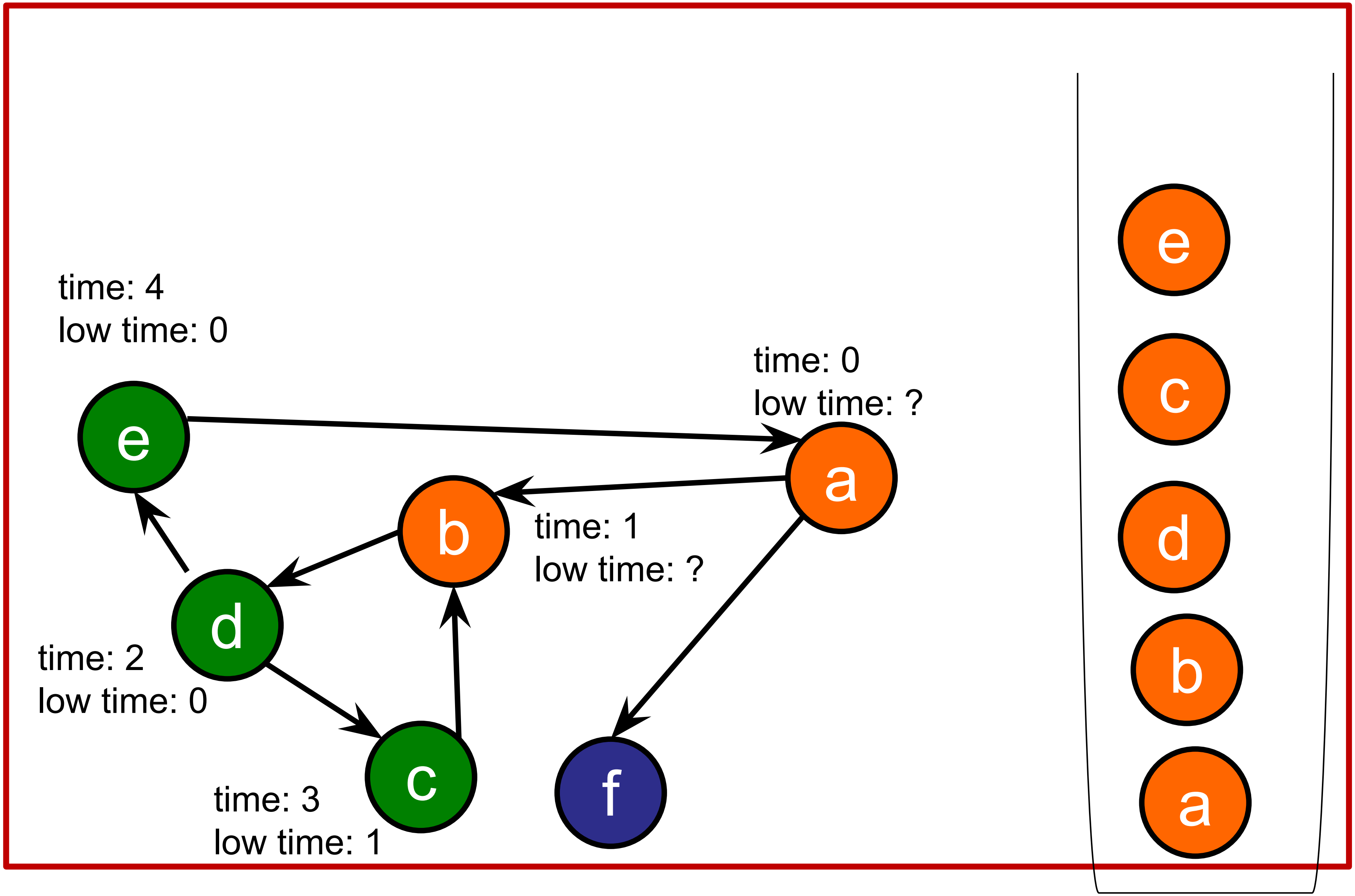
Connected Components

Correction:



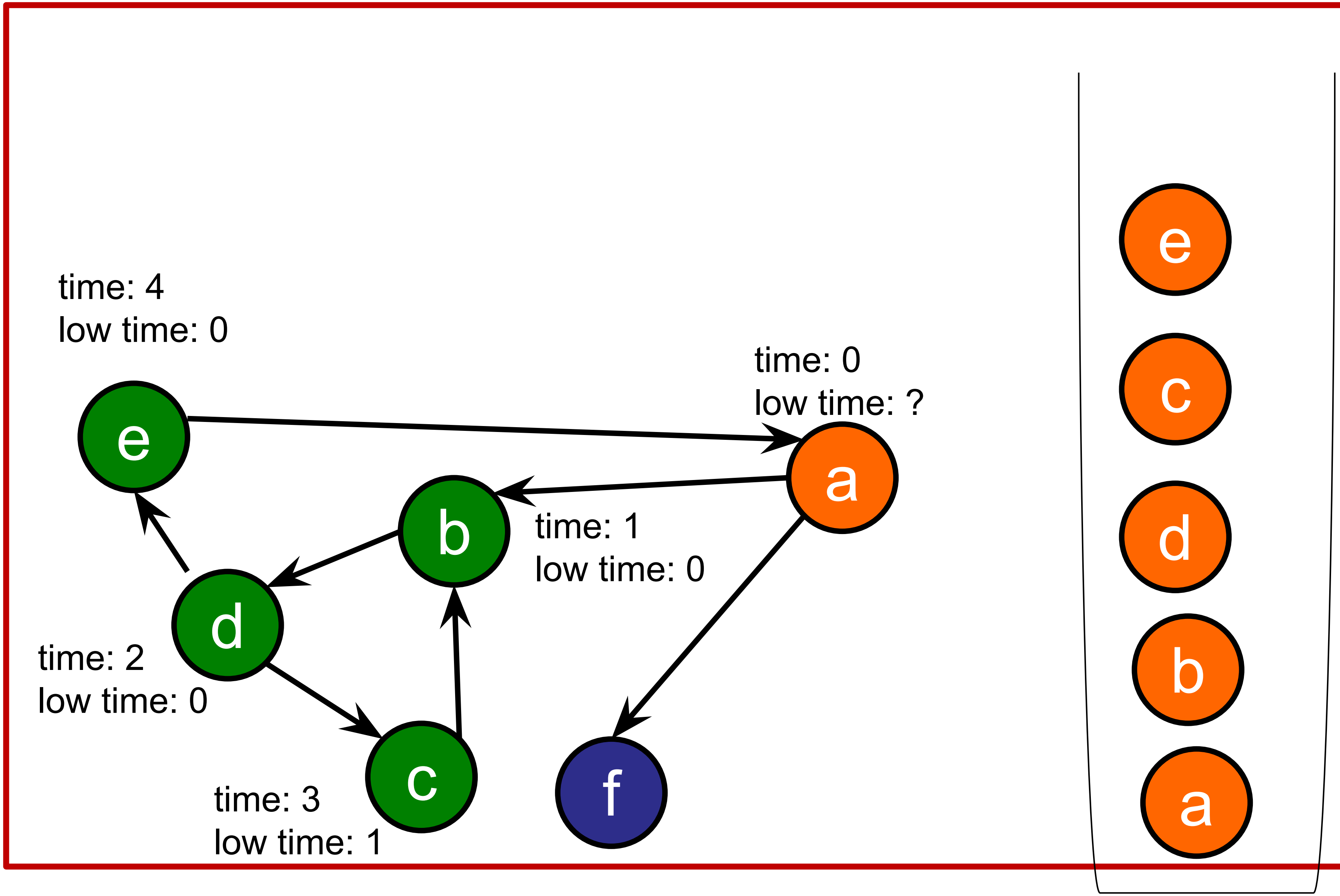
Connected Components

Correction:



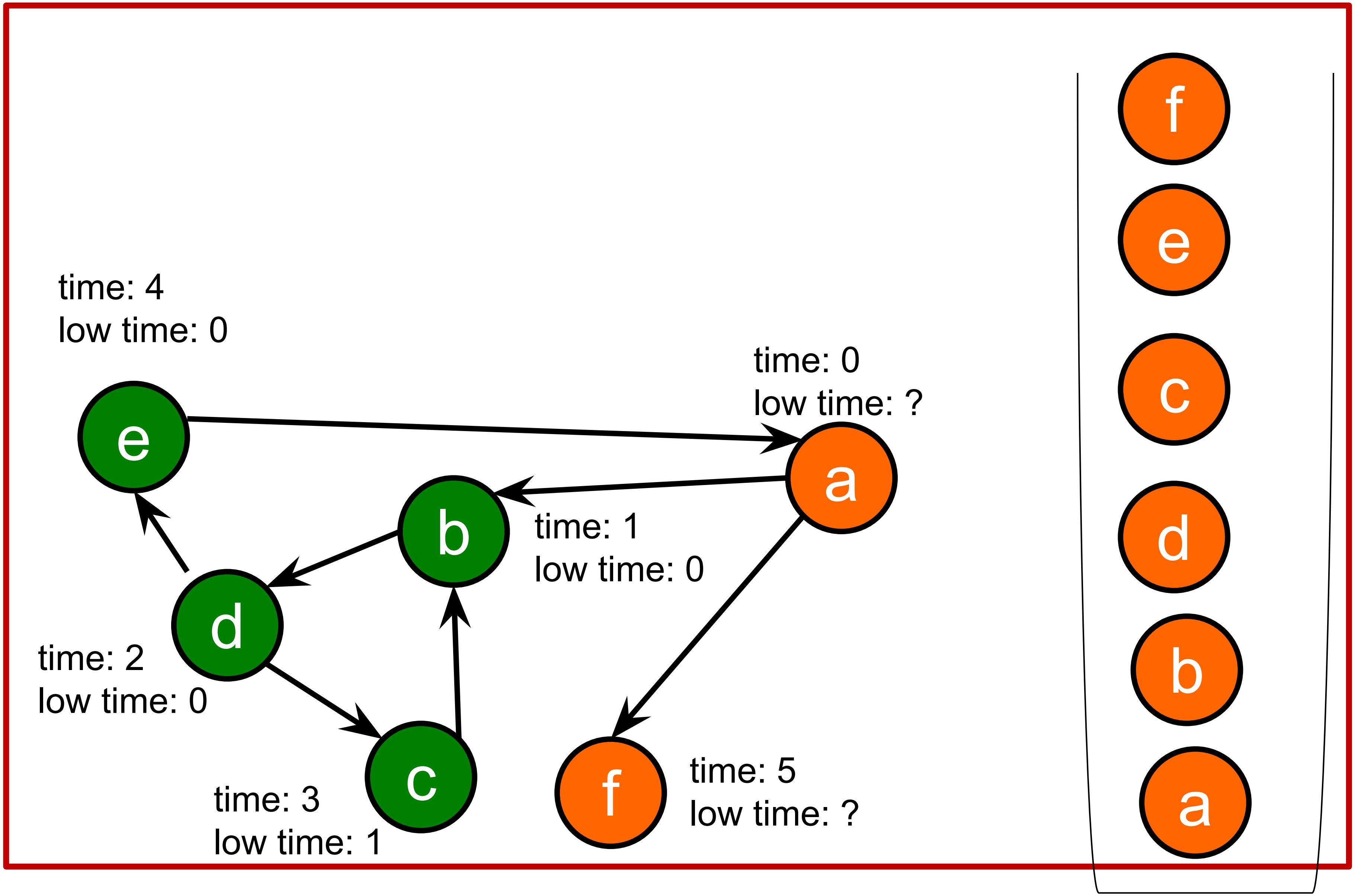
Connected Components

Correction:



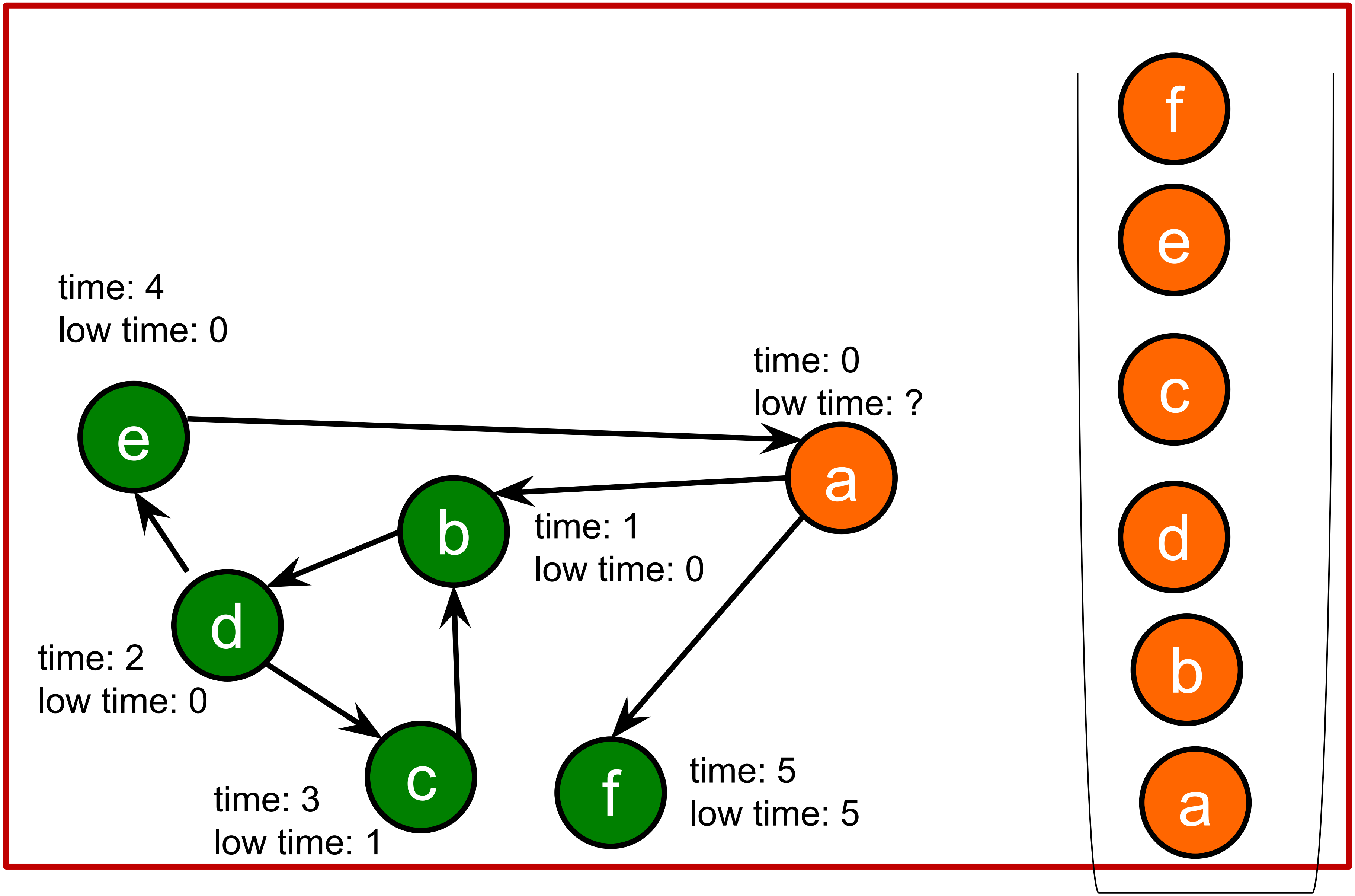
Connected Components

Correction:



Connected Components

Correction:

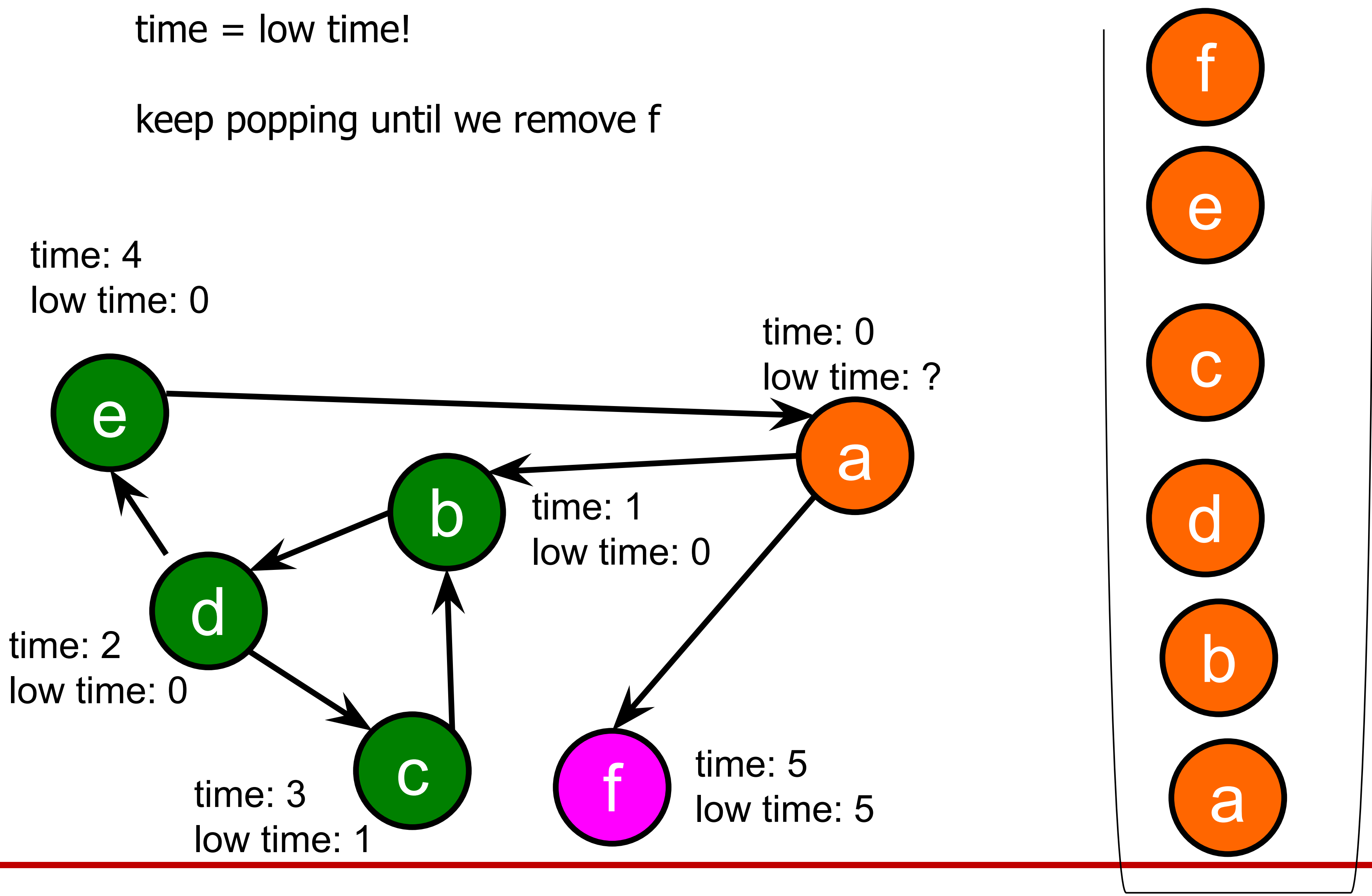


Connected Components

Correction:

time = low time!

keep popping until we remove f

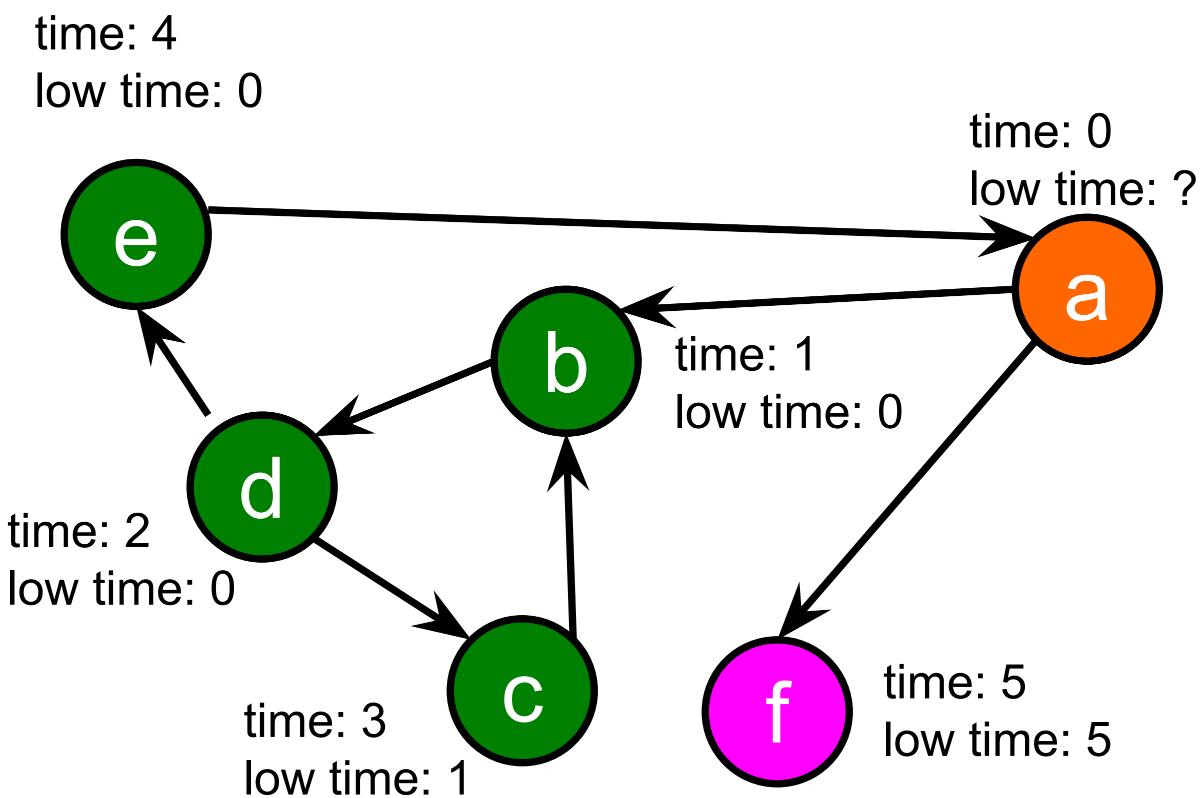


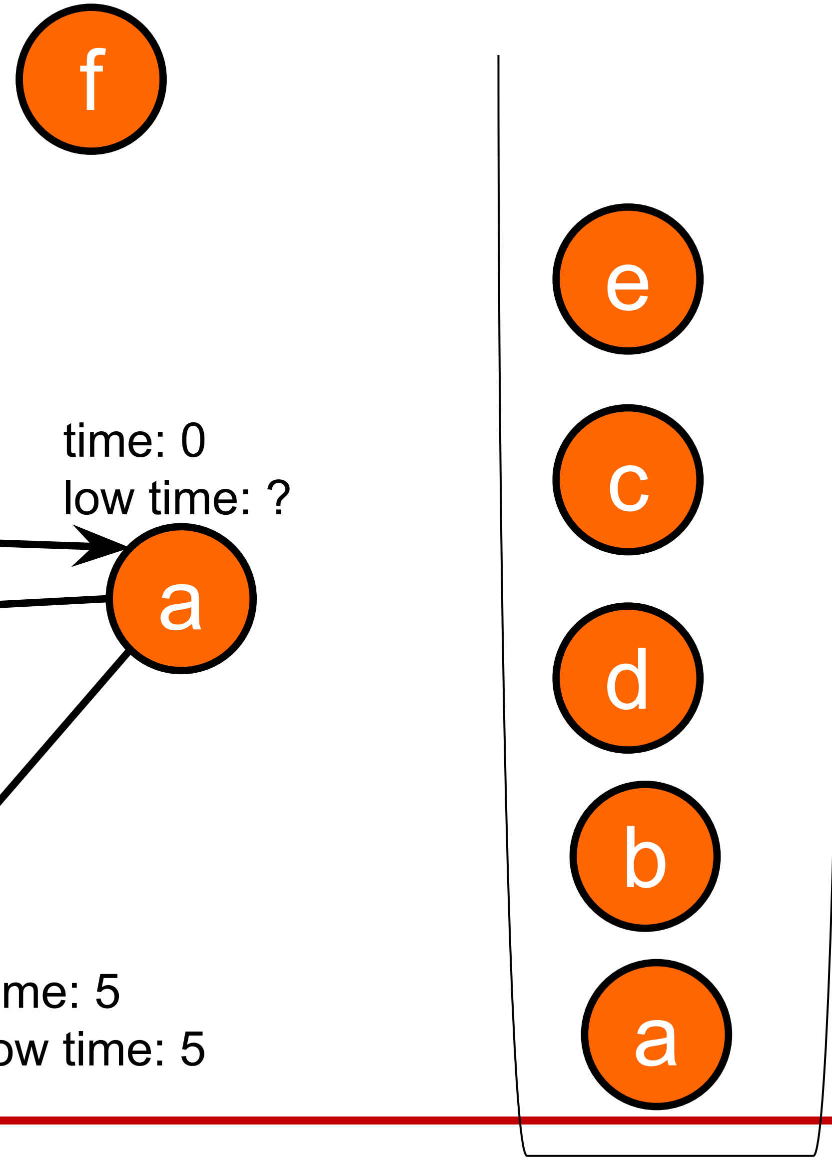
Connected Components

Correction:

time = low time!

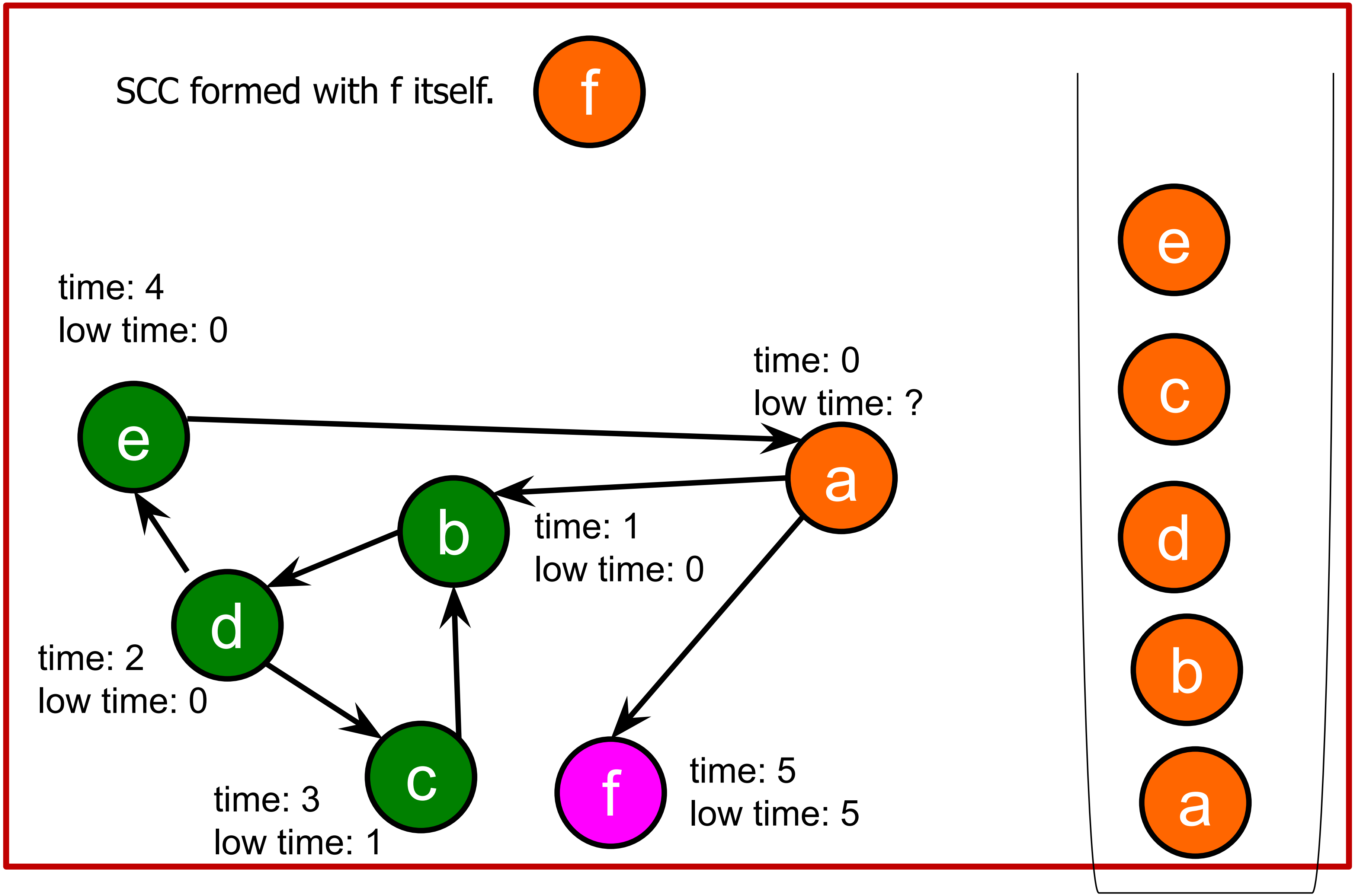
keep popping until we remove f





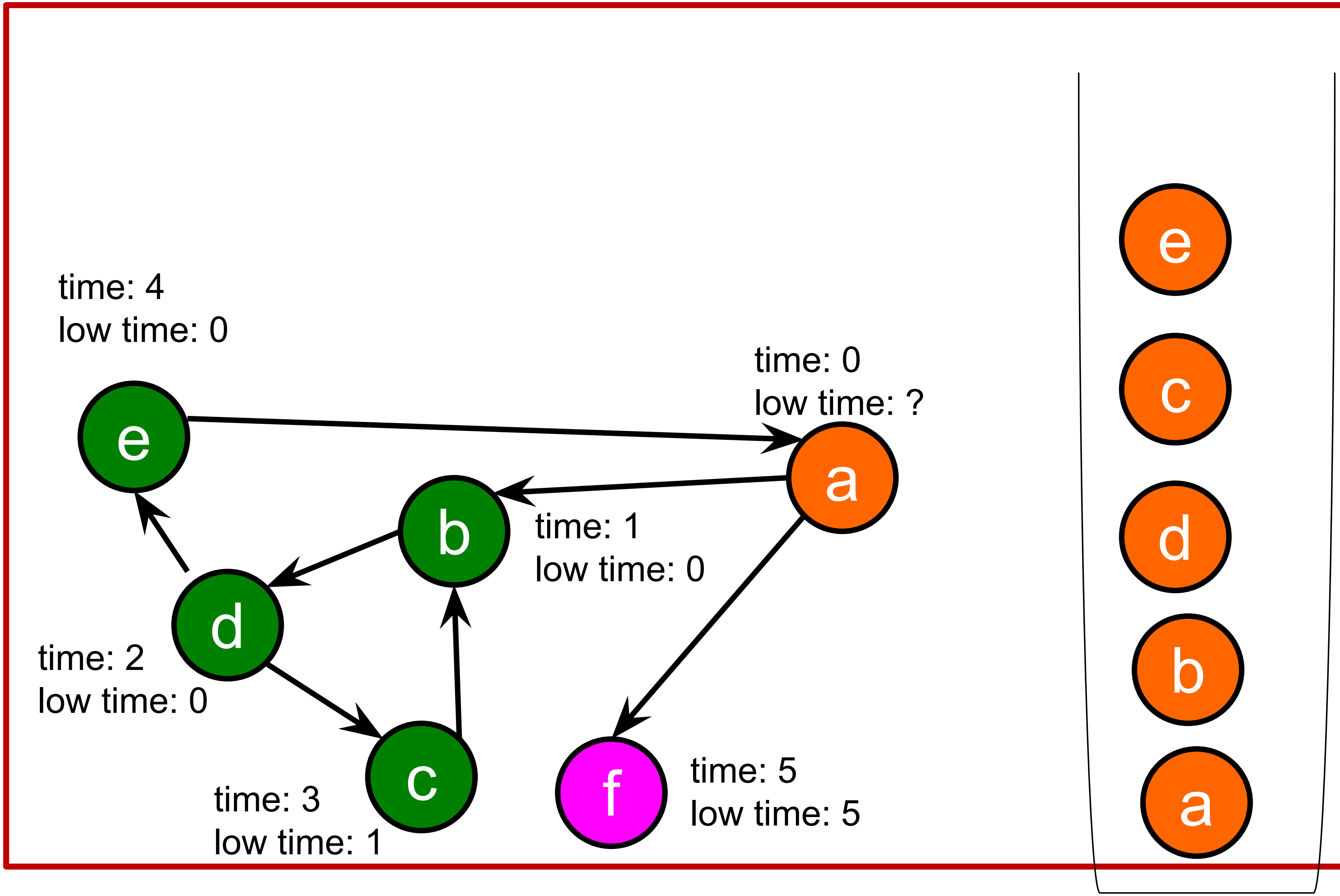
Connected Components

Correction:



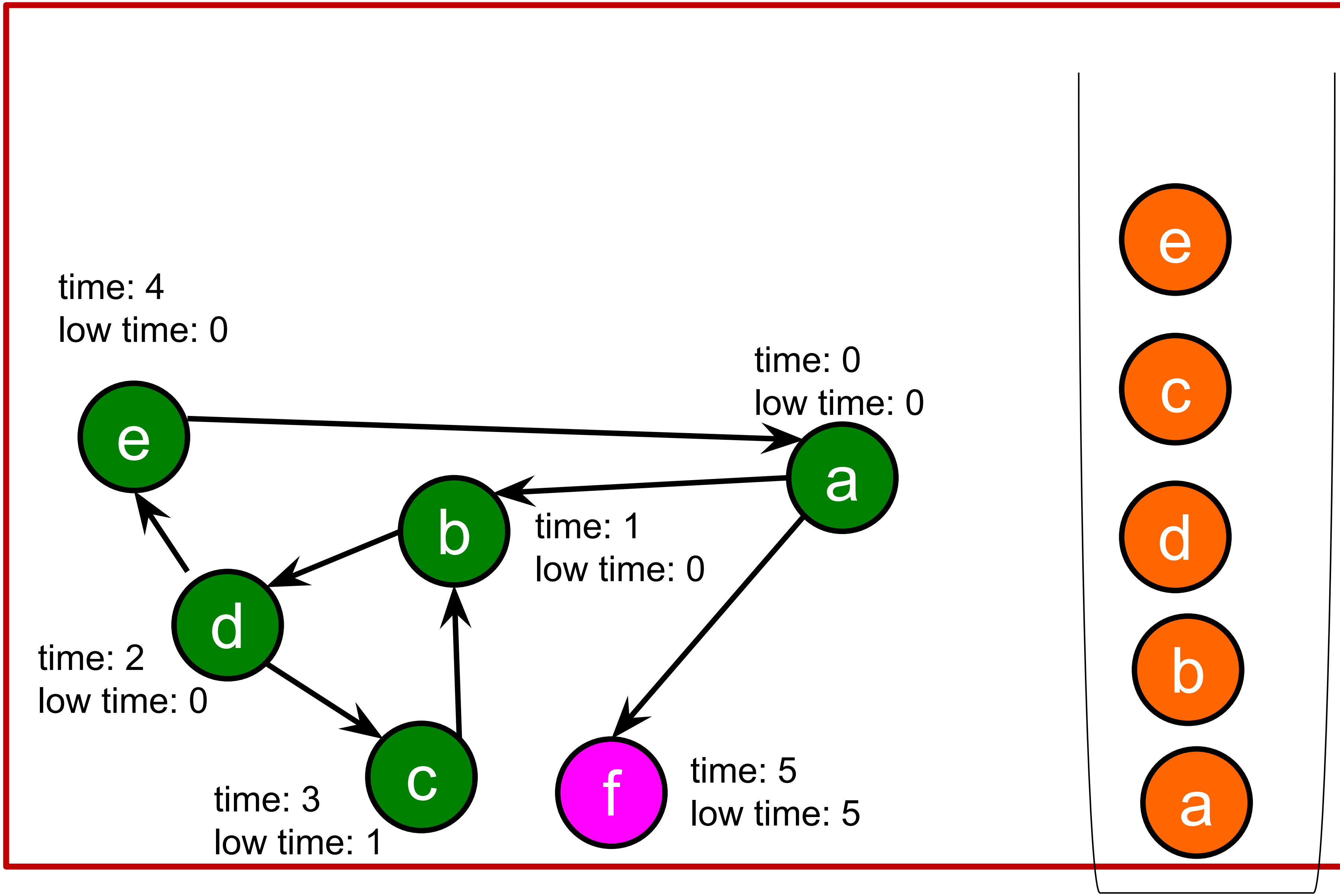
Connected Components

Correction:



Connected Components

Correction:

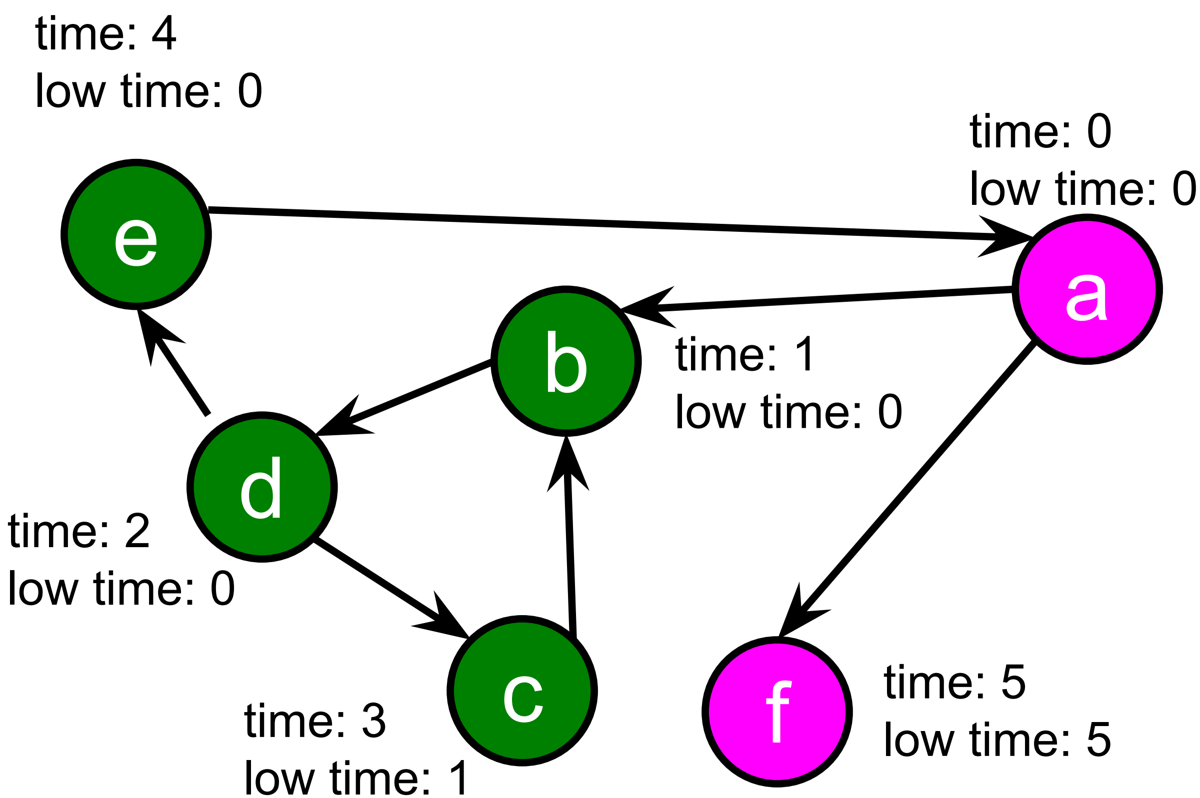


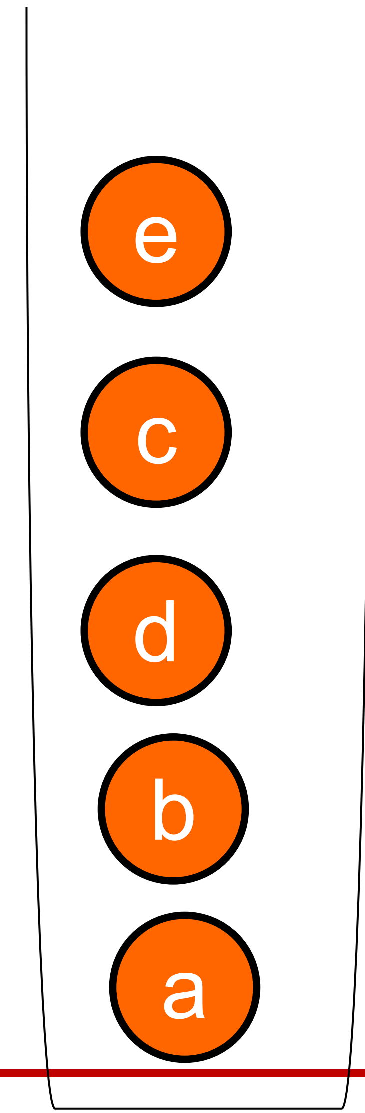
Connected Components

Correction:

time = low time!

keep popping until we remove a



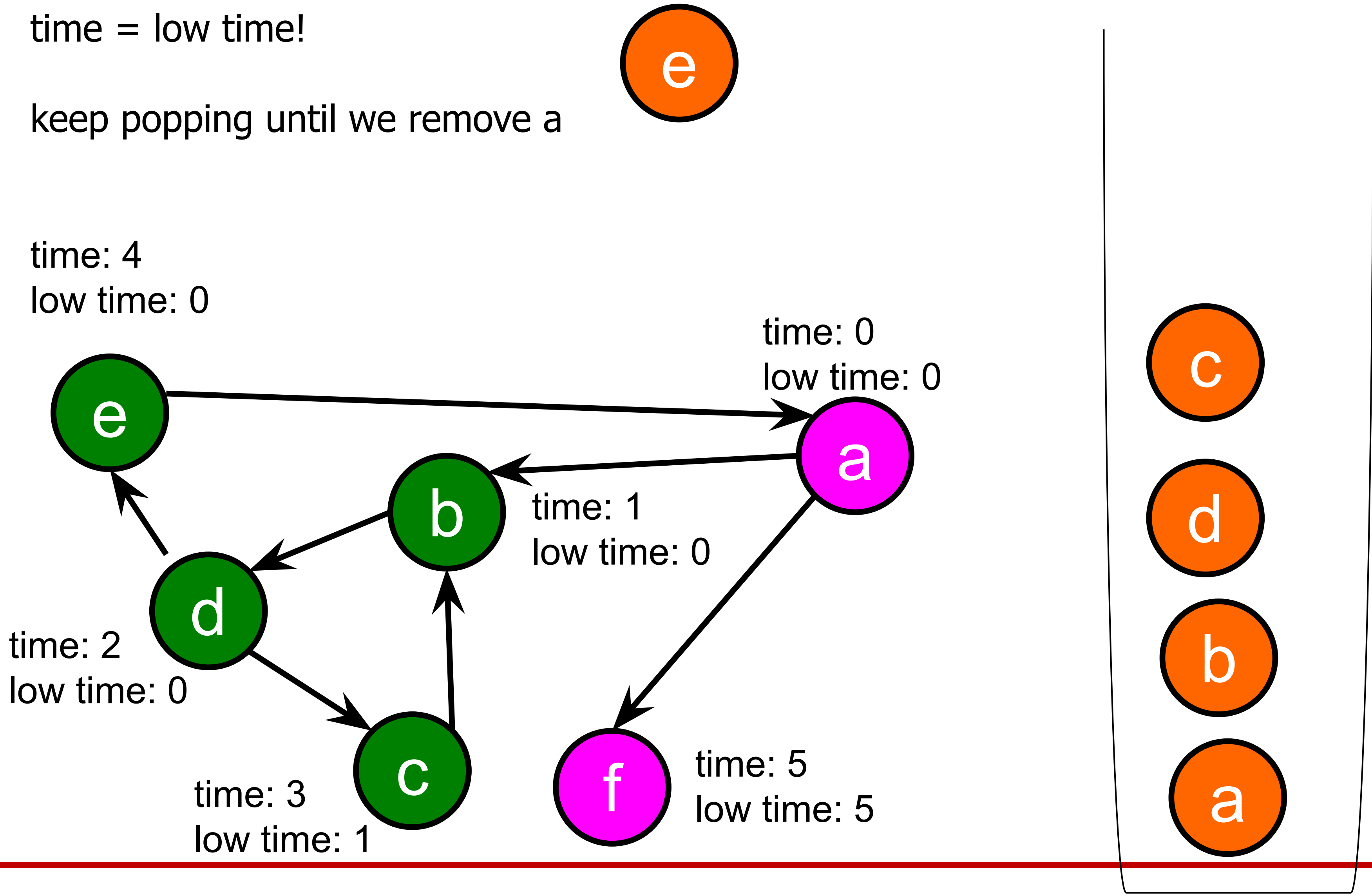


Connected Components

Correction:

time = low time!

keep popping until we remove a

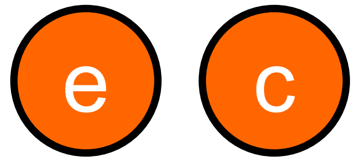


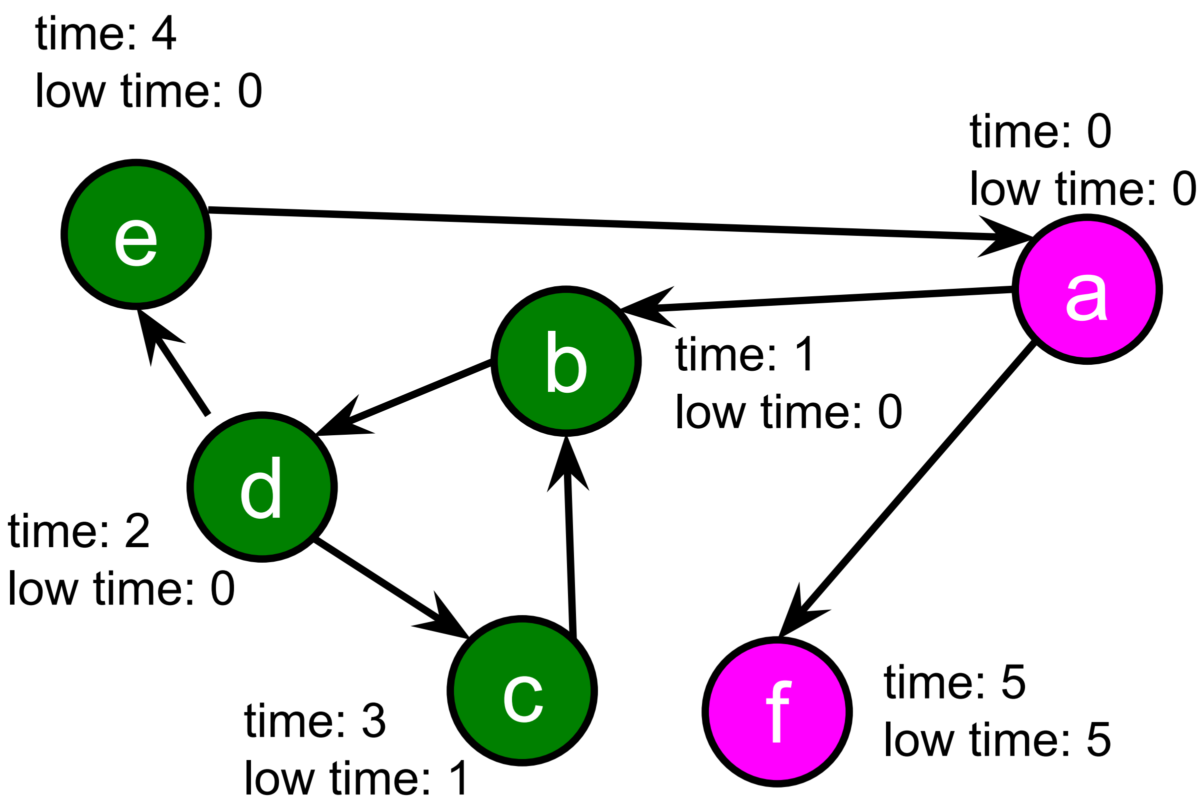
Connected Components

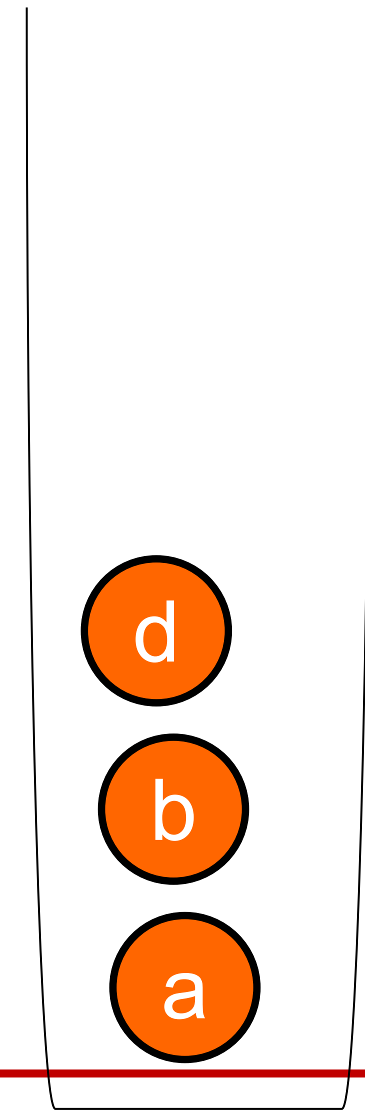
Correction:

time = low time!

keep popping until we remove a





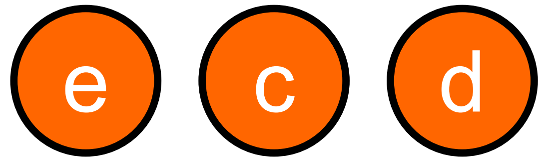


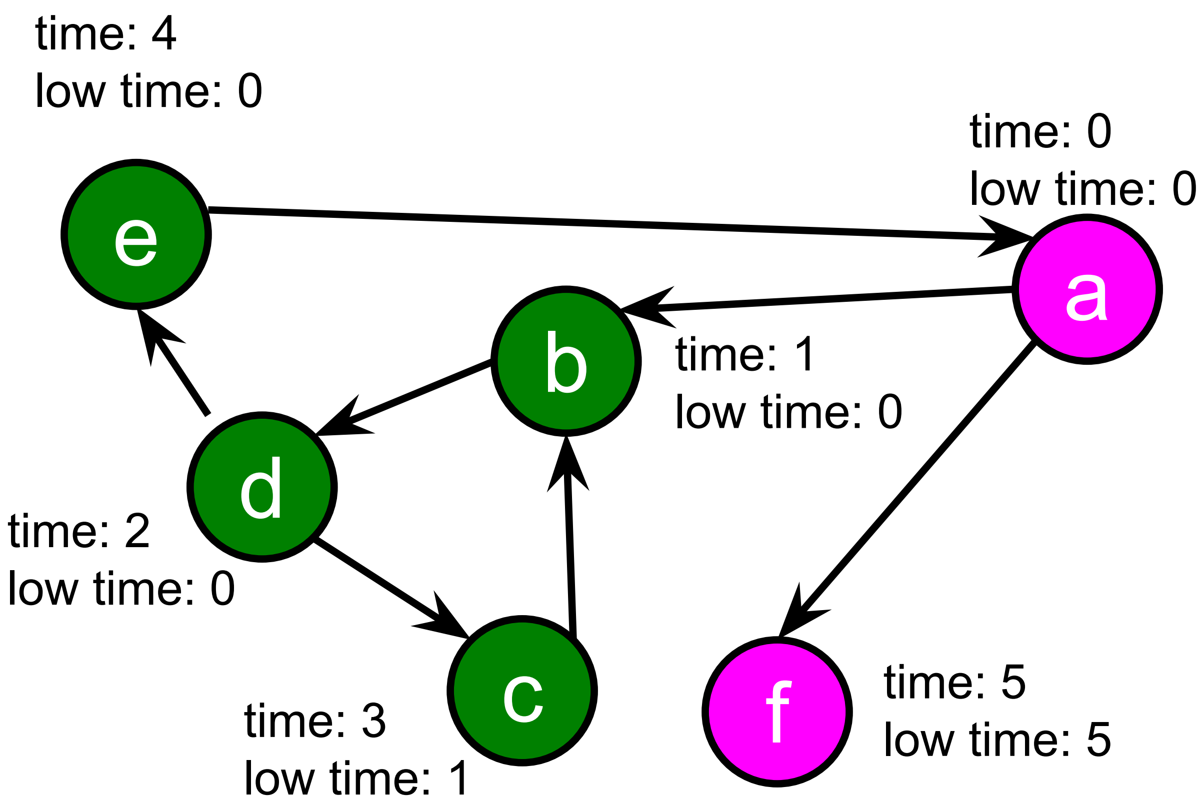
Connected Components

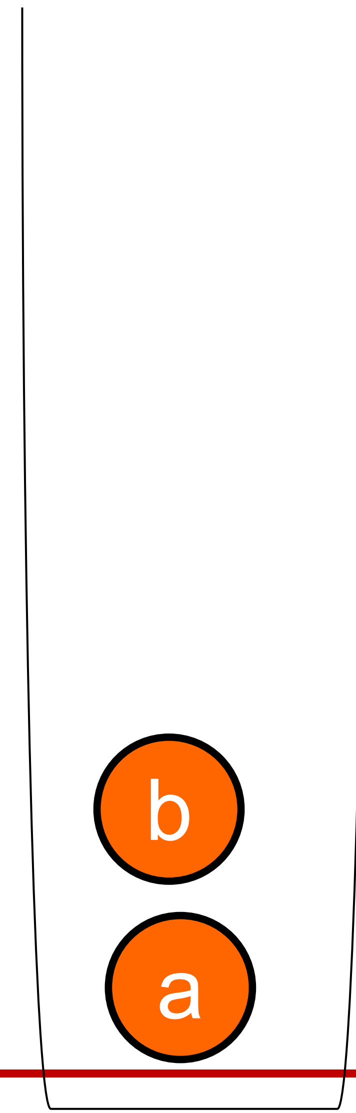
Correction:

time = low time!

keep popping until we remove a





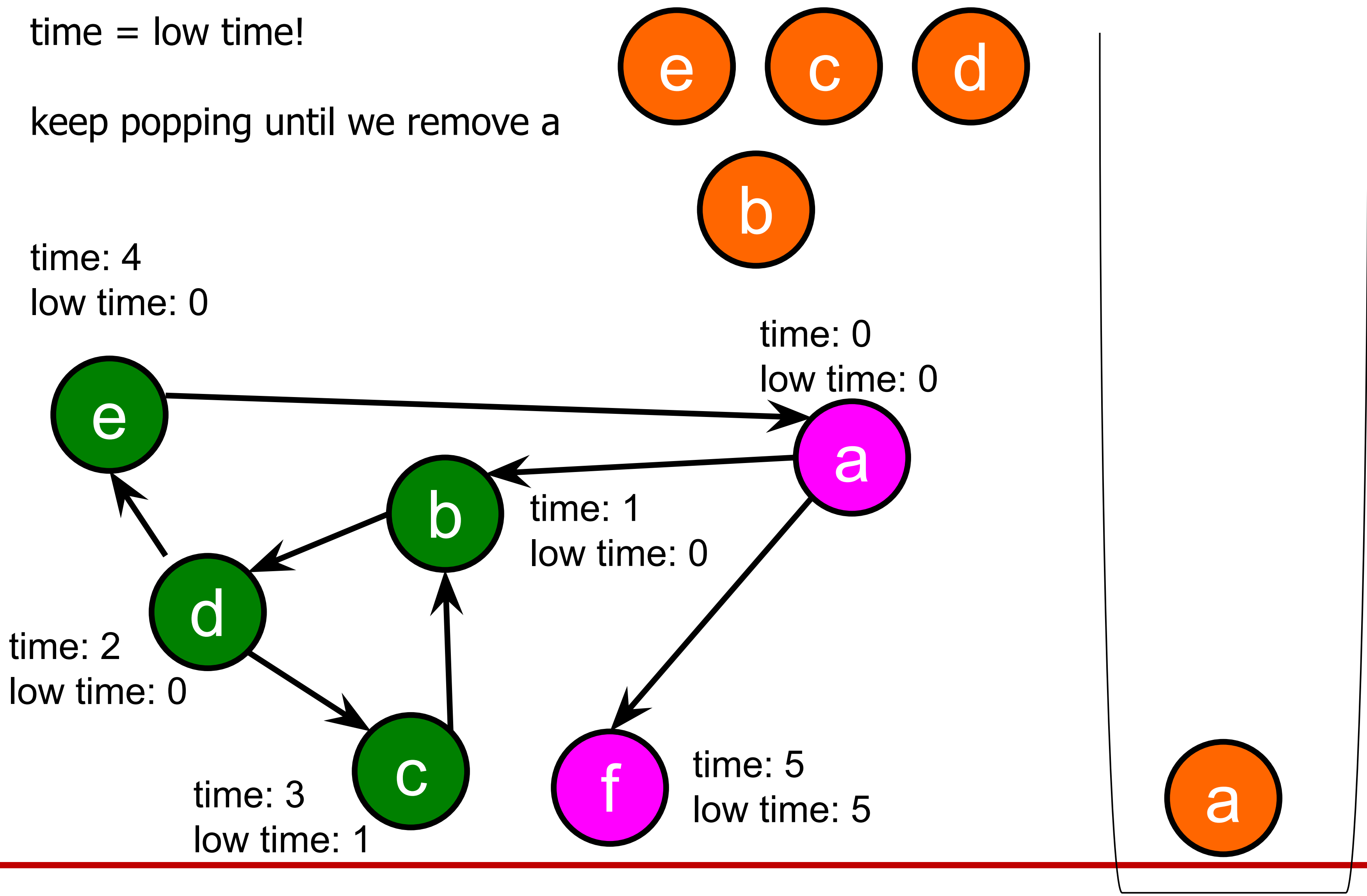


Connected Components

Correction:

time = low time!

keep popping until we remove a

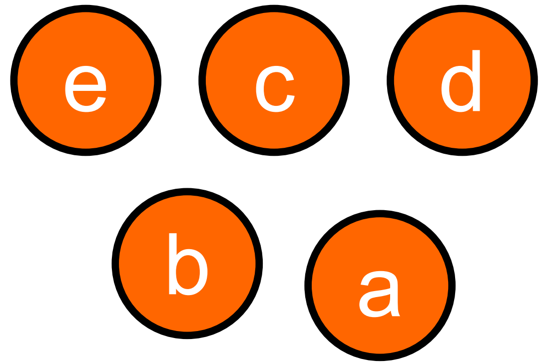


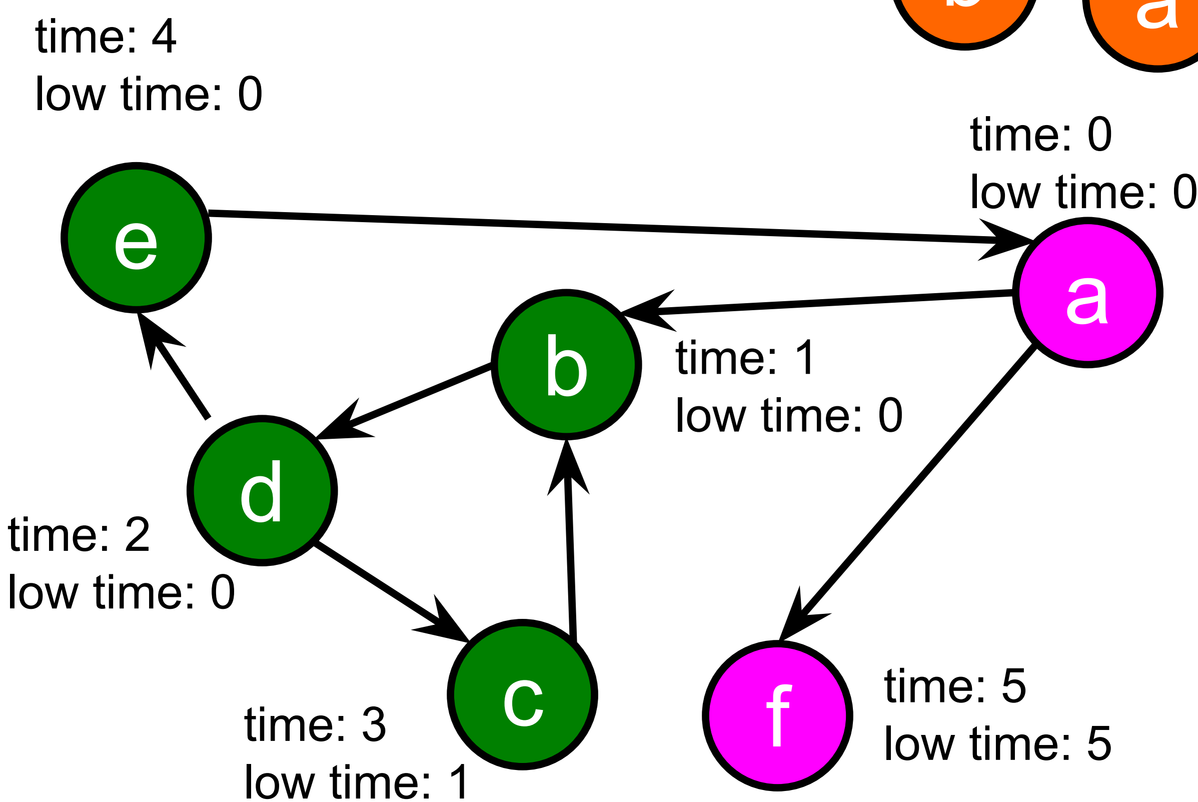
Connected Components

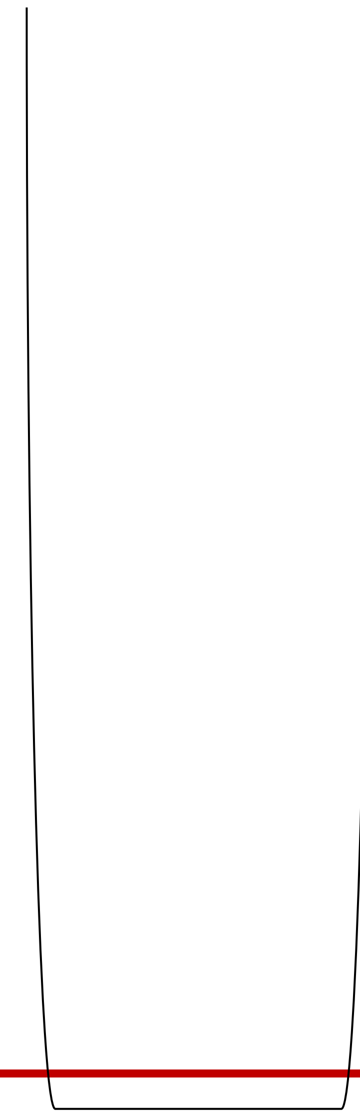
Correction:

time = low time!

keep popping until we remove a



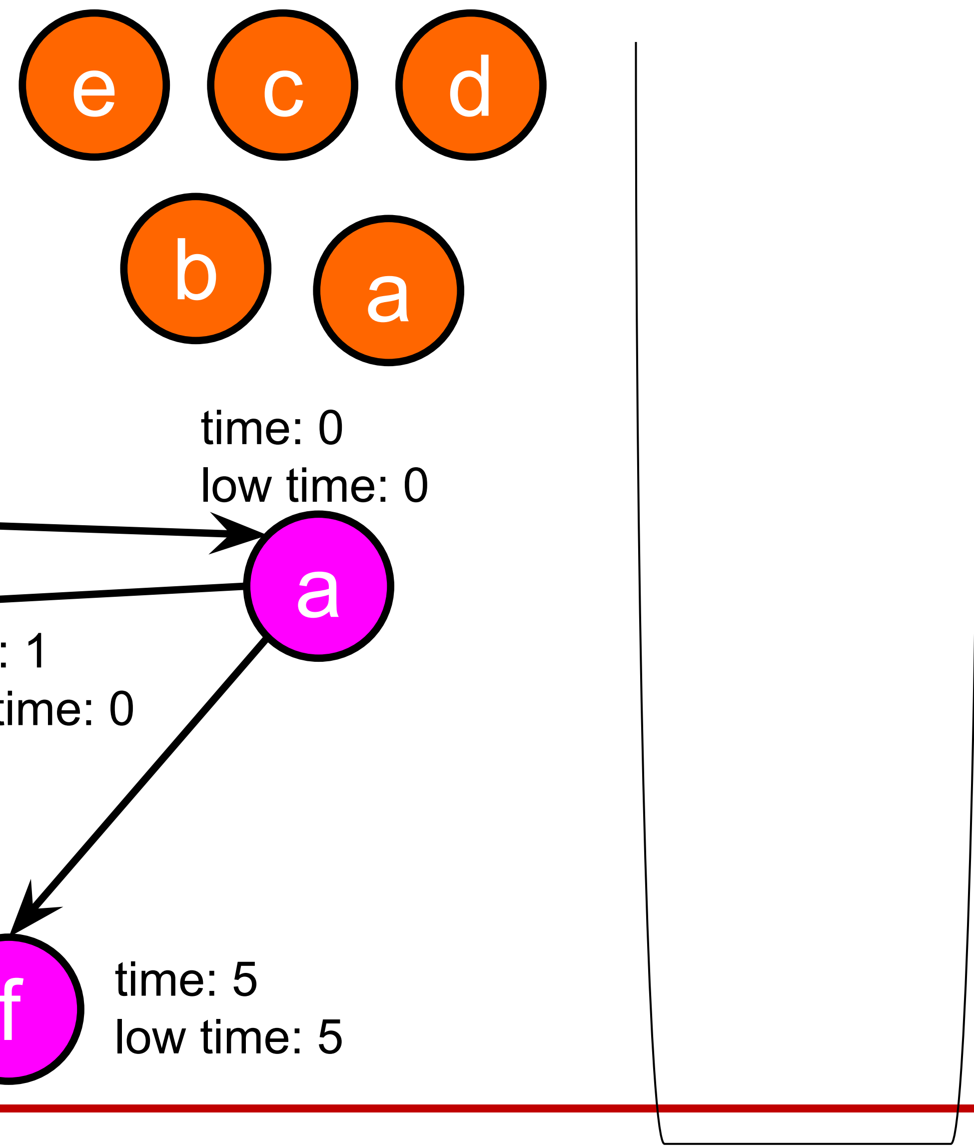


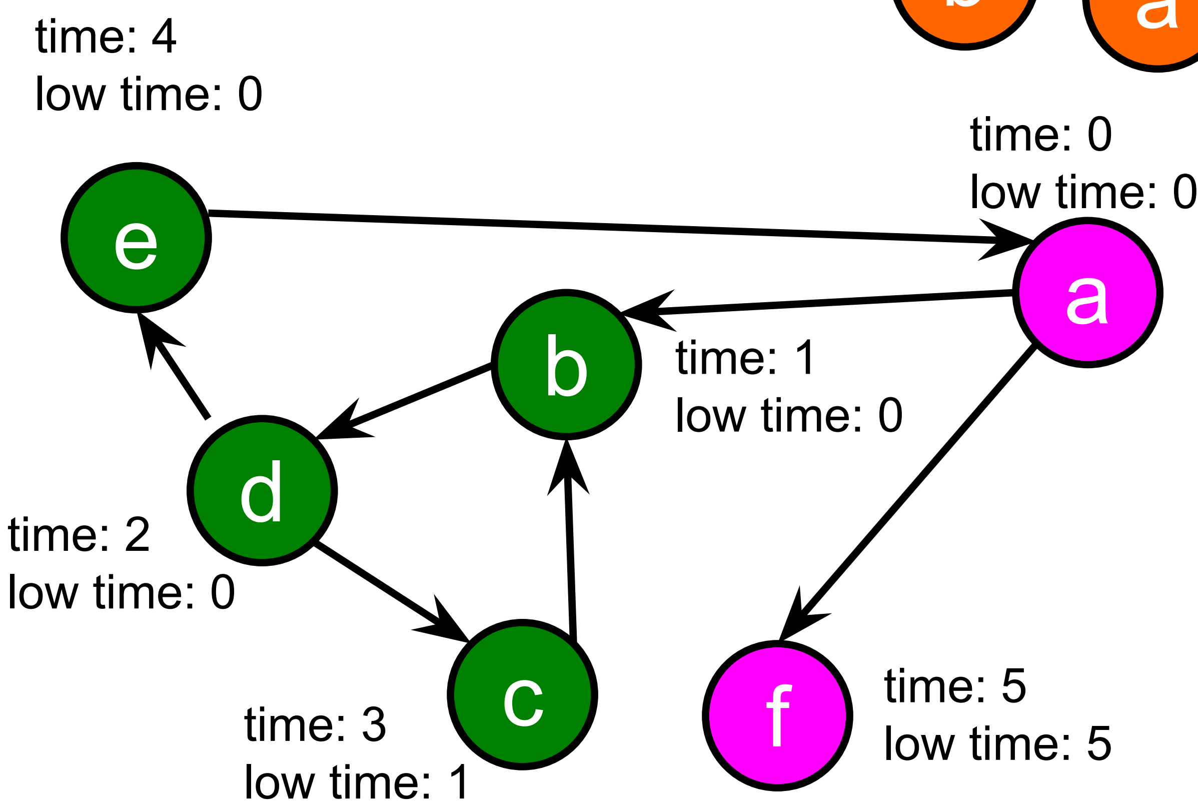


Connected Components

Correction:

SCC formed with
{a b c d e}



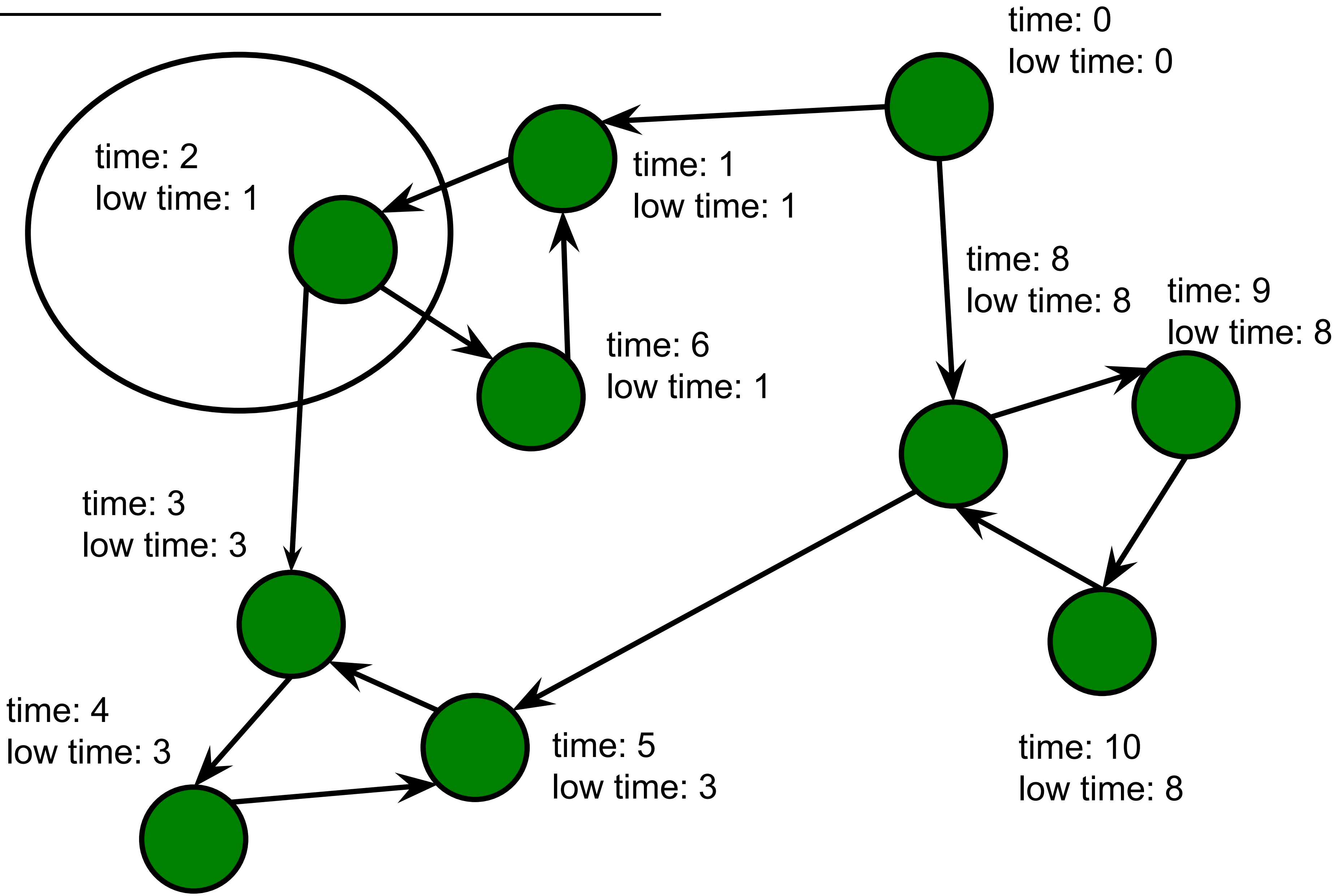


Cycle Detection

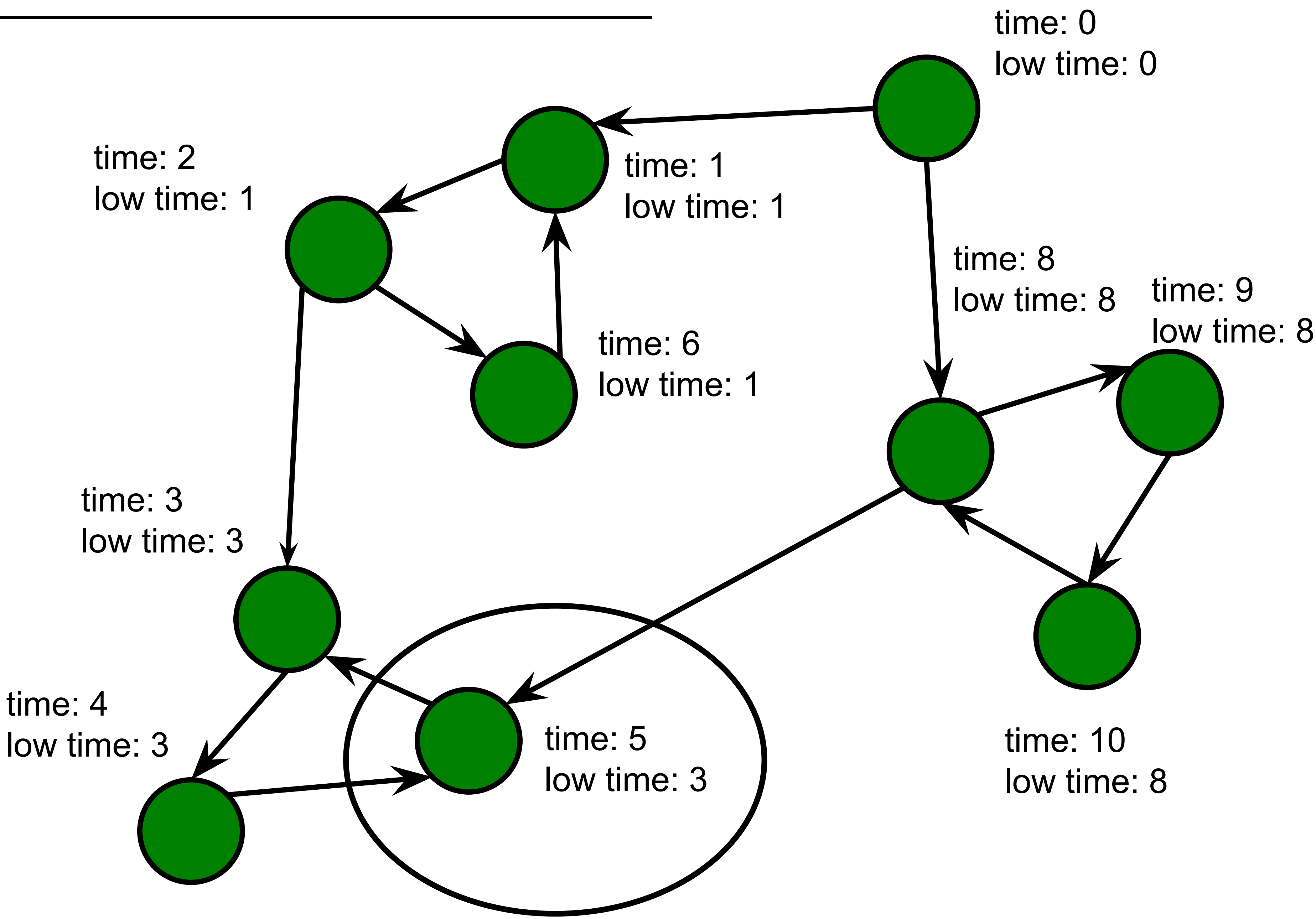
How does low time help us?

If we ever find a node whose low time < time, then there is a cycle!

Cycle Detection



Cycle Detection



Cycle Detection

How does low time help us?

If we ever find a node whose low time < time, then there is a cycle!

Cycle Detection

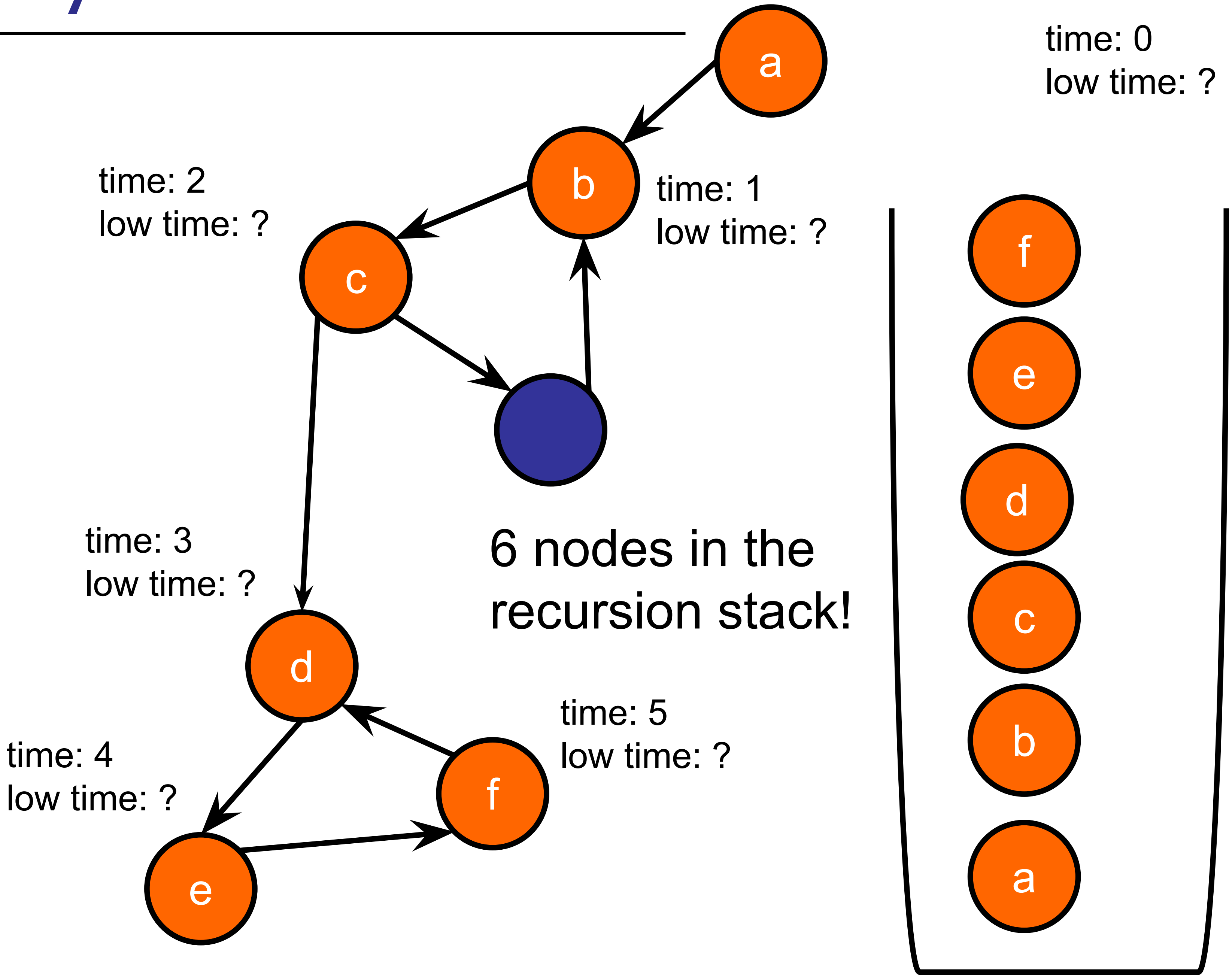
How does low time help us?

If we ever find a node whose low time < time, then there is a cycle!

Recall: We only update low time based on nodes whose low times are not set.

Intuition: A low time that is not yet set -> that node is still in the recursion stack

Cycle Detection



6 nodes in the recursion stack!

Articulation Points?

Challenge: Figure out how to run DFS on a directed graph (how should the algorithm change) so that we can find articulation points using low time and time?

Intuition: If a node's low time < time, then it is not an articulation point. Otherwise, it is.

But how do we handle bidirectional edges?

Roadmap

Algorithms on Directed Graphs

- Searching directed graphs (DFS / BFS)
- Topological Sort
- Connected Components

More Algorithms on Undirected Graphs

Next Week:

More shortest pathfinding!