NOI.PH Training: Graphs 4

Connectivity and Biconnectivity

Kevin Charles Atienza

Contents

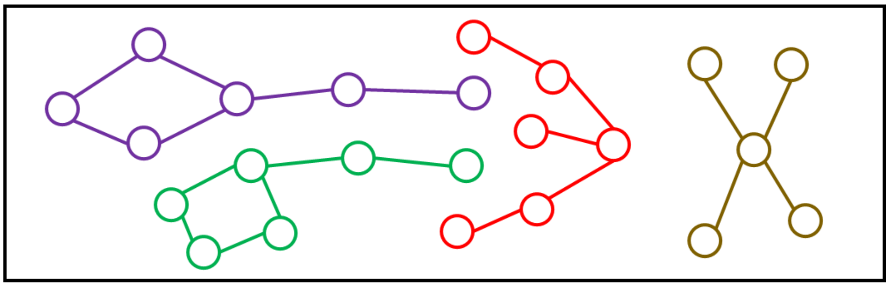
1 Connectivity in Undirected Graphs

This week, we'll connect ourselves with various topics related to connectivity in graphs.

Connectivity is easier to understand with undirected graphs. That's because the edges are *bidirectional*, which implies that if there's a path from a to b, then there's also a path from b to a. So we'll start with undirected graphs.

A **connected component** is a maximal[1] set of nodes that is pairwise connected.

Here are the connected components in an example graph, where each connected component is colored by a single color:



connected components

More formally, if we let $a \sim b$ mean "a is connected to b", then:

- $\sim$ is reflexive, i.e., $a \sim a$.
- $\sim$ is symmetric, i.e., $a \sim b$ implies $b \sim a$.
- $\sim$ is transitive, i.e., $a \sim b$ and $b \sim c$ implies $a \sim c$.

Hence, $\sim$ is what you would call an **equivalence relation**, and the **equivalence classes** under $\sim$ are the connected components of the graph.[2]

Finding connected components can be done with standard graph traversals such as BFS or DFS. You probably already know that. Either algorithm should work fine, although note that DFS usually gets deeper in recursion than BFS.

A **cycle** is a nontrivial path from a to itself. We say a graph is **acyclic** if it has no cycles. An undirected acyclic graph is called a **forest**. A connected forest is a **tree**.

One natural question is how to detect if there's a cycle in an undirected graph. For this, you can use BFS or DFS again to detect if a cycle exists and find such a cycle in $\mathcal{O}(n + e)$ time. A fancier (and perhaps easier-to-implement) approach is **union-find**, although it runs in $\mathcal{O}(n + e \cdot \alpha(n))$, where α is the slow-growing *inverse Ackermann function*. Due to this factor, union-find is noticeably slower when $n \geq 2^{2^{2^{65536}}}$, so this is unacceptable. Just kidding! $\alpha(2^{2^{2^{65536}}}) \leq 5$, so in practice this is fine. (Though if the time limits are particularly tight, that factor of 5 sometimes bites, so it could be better to use a BFS/DFS instead. You need to judge if union-find will pass the time limit or not.)[3]

[1] "Maximal" means you can't add any more nodes without violating the requirement.

[2] If you're unfamiliar with equivalence relations and equivalence classes, please ask in Discord.

[3] Also, note that this only detects a cycle; it's also harder to find the cycle with union-find.

Exercise 1.1. Explain how to use BFS or DFS to detect if a cycle exists in an undirected graph, and find one, in $\mathcal{O}(n+e)$.

1.1 Depth-first search revisited (undirected version)

It's time to take a closer look at one of the simplest graph traversal methods: DFS. The DFS will be central in the algorithms we'll discuss later on, so now is a good time to revisit DFS with more detail.

DFS, in its simplest form, is just a particular order of traversal of the graph determined by the following recursive procedure: (in pseudocode)[4]

```
def DFS(i):
    # perform a DFS starting at node i

    visited[i] = True # mark it as visited
    for j in adj[i]:
        if not visited[j]:
            DFS(j)

def DFS_all():
    # perform a DFS on the whole graph
    for i from 0 to n-1:
        visited[i] = False

    for i from 0 to n-1:
        if not visited[i]:
            DFS(i)
```

On its own, it's not very interesting, since all this does is *visit* all nodes (and mark "`visited[i]`" as true). But we can actually extract more information from this DFS procedure. First, one useful thing we can do is generalize:

```
def DFS(i, p):
    visited[i] = True
    parent[i] = p

    visit_start(i) # do something once a new node is visited

    for j in adj[i]:
        if not visited[j]:
            DFS(j, i)

    visit_end(i) # do something once a node has finished expanding

def DFS_all():
```

[4] I expect you can easily convert pseudocode into real code by now!

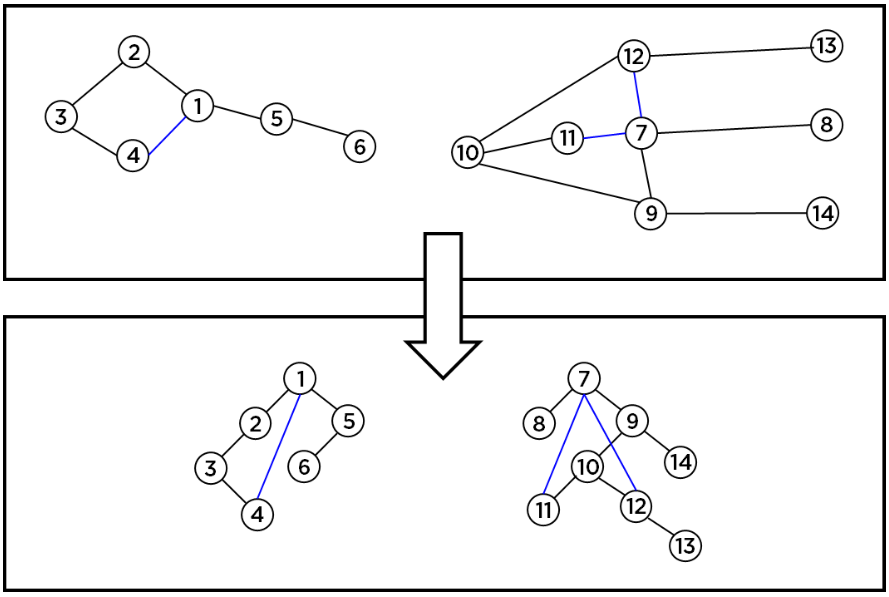
```
    for i from 0 to n-1:
        visited[i] = False
        parent[i] = -1

    for i from 0 to n-1:
        if not visited[i]:
            DFS(i, -1)
```

Here, the functions “visit_start” and “visit_end” are whatever you wanted to do with the nodes. They will be called once for each node, in order of the starting and ending times of the nodes, respectively. This generalized version is quite useful in many cases.

Notice also that here we’re computing the “parent” array which contains the parents of the nodes in the traversal. This could be useful in some cases.

But actually, DFS is more interesting and useful than that; there are still other bits of information we can extract from this traversal that can help us solve some problems.

To find this hidden information, let’s try to consider a particular DFS traversal, and then let’s draw the nodes on paper so that each node appears *beneath* its parent, and the “children” of a node are ordered from left to right according to the order of visitation. For instance, it might look like this:



tree edges, back edges

The black edges represent the edges that are traversed by the DFS. We call such edges **tree edges**. The remaining edges, colored in blue, are the ones ignored by the DFS and are called **back edges**. We can now clearly see that DFS classifies the edges into one of two types, depending on how they were handled by the DFS traversal. Later on, we’ll see how this classification will be useful for us.

Note that, due to the depth-first nature of DFS, other types of edges in the DFS forest can’t appear, such as nontree edges that don’t point to an ancestor. Please convince yourself of this.

Exercise 1.2. Explain why every nontree edges always point "upwards", i.e., from a root to one of its ancestors.

Hint: The graph is undirected. In particular, an edge can be traversed in either direction. What does this imply for the DFS?

If we consider only the black edges, then we get a forest. For this reason, this is sometimes called the **DFS forest** of the graph.

To be more specific, the classification of the edges is done by the following procedure:

```
def DFS(i, p):
    start_time[i] = time++
    parent[i] = p

    for j in adj[i]:
        if start_time[j] == -1:
            mark (i, j) as a tree edge
            DFS(j, i)
        else if j != p:
            mark (i, j) as a back edge

    finish_time[i] = time++

def DFS_all():
    time = 0
    for i from 0 to n-1:
        start_time[i] = -1
        finish_time[i] = -1
        parent[i] = -1

    for i from 0 to n-1:
        if start_time[i] == -1:
            DFS(i, -1)
```

An important thing to note here is that the condition `j != p` is checked before marking some edge as a back edge—this is very important, otherwise, we will be marking all edges as back edges! [5]

Exercise 1.3. Why is it that if we don't check the condition `j != p`, all edges will be marked back edges?

Notice that we've also replaced the `visited` array with two new arrays `start_time` and `finish_time`, which will contain the starting and finishing times of each node's visitation. There aren't many uses for them yet, but there will be later on.

The running time is still $\mathcal{O}(n + e)$, but along the way, we've gained more information about the DFS traversal, namely the edge classifications, the starting and ending times of each visitation, and the parents of the nodes! These pieces of information will prove valuable in the

[5] An even further complication in this part of the code is when there are **parallel edges**, that is, multiple edges that connect the same pair of nodes. In this discussion, we'll assume that there are no parallel edges. But if you want to learn how to deal with parallel edges, you can try to figure it out yourself, or just ask :)

upcoming algorithms.

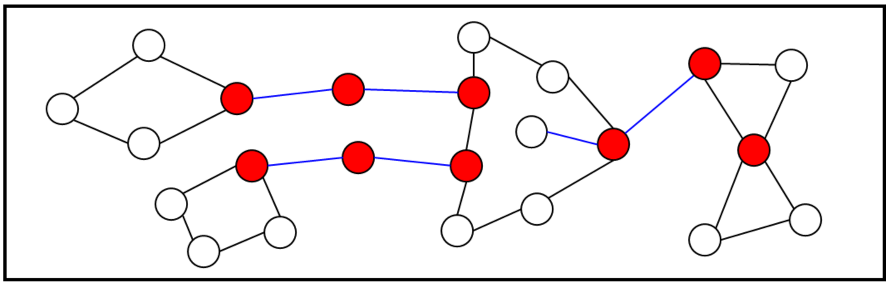
By the way, note that the implementation above is written *recursively*. In some large trees, stack overflow might be a concern, especially if the stack size is limited. In those cases, you might want to implement an *iterative* version.[6]

1.2 Bridges and articulation points

A **bridge** is an edge whose removal increases the number of connected components. An **articulation point** is a node whose removal increases the number of connected components. Note that when you remove a node, you also remove the edges adjacent to it. These are sometimes called **cut edges** or **cut points**, which are terms you probably like more, but I'll use "bridges" and "articulation points" to annoy you!

For simplicity, let's assume here that the graph is connected; if not, we can consider each connected component separately. Thus, we will use the following specialized definitions: In a connected graph, a **bridge** is an edge whose removal disconnects the graph, and an **articulation point** is a node whose removal disconnects the graph.

In the following picture, the blue edges are the bridges, and the red nodes are the articulation points:



bridges, articulation points

You should note that vertices of degree 1 are not articulation points.

It's easy to see why one would want to identify and study these edges/nodes. Roughly speaking, these edges and nodes are the *weakest links* of your network. For example, if your graph represents a network of computers, then a bridge represents a critical connection, and an articulation point represents a critical computer.

Bridges and articulation points are also sometimes called "cut edges" and "cut vertices", respectively, for obvious reasons.

1.2.1 Finding bridges and articulation points

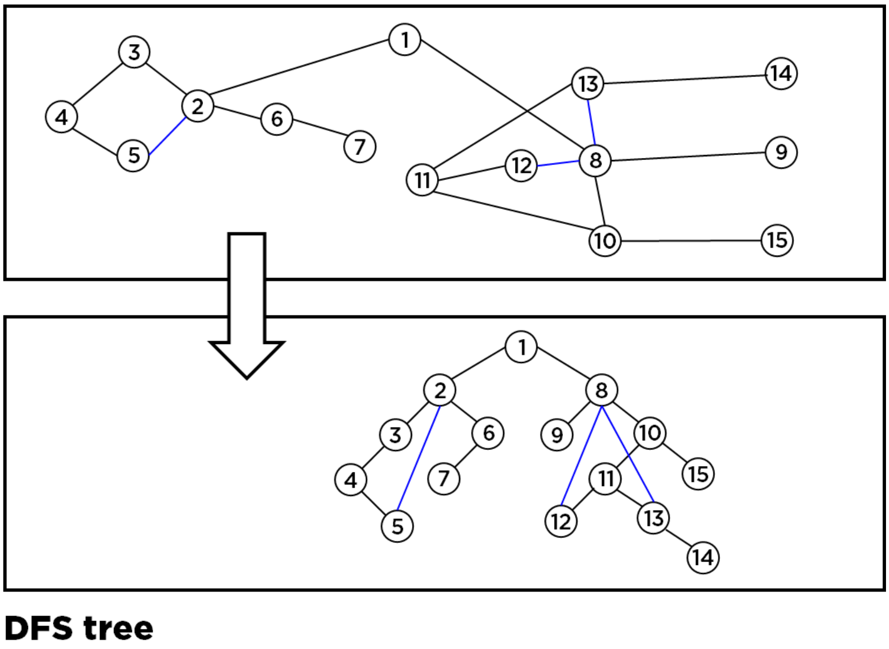
Now that you know what bridges and articulation points are, it's time to put on our algorithms hat.

Problem 1.1. Given a (connected) undirected graph, find all its bridges and articulation points.

[6] A generic way to do that conversion is to simulate the call stack with an actual stack, and the stack entries describe the whole state of the function at that point. In this case, you only need to push "`i`" and the index of "`j`" in "`adj[i]`".

If the graph is small enough, you can just individually check each edge/node if they are a bridge/articulation point by removing it and checking if the resulting graph is disconnected. Since it takes $\mathcal{O}(n+e)$ time to traverse a graph, it takes $\mathcal{O}(e(n+e))$ and $\mathcal{O}(n(n+e))$ time to find the bridges and articulation points this way.

But it turns out that we can compute both in $\mathcal{O}(n+e)$ time using DFS! To see how, let's say we performed DFS on our graph. The following is a picture of the resulting "DFS forest" (which is really just a "DFS tree" since the graph is connected):



Exercise 1.4. Stare at this picture for a while and think about exactly when an edge is a bridge, or when a node is an articulation point.

After a bit of pondering, you may stumble upon the following precise conditions on when an edge is a bridge.[7]

- Back edges can never be bridges.
- Let $a \to b$ be a tree edge.[8] Then $a \to b$ is a bridge if and only if there's no back edge from any descendant of b to any ancestor of a.

Exercise 1.5. Why is a back edge never a bridge?

The condition for articulation points are a bit trickier, so I suggest paying careful attention.

Exercise 1.6. Show that a node a is an articulation point iff either of the following is true:

- a is not the root of a DFS tree and a has a child b such that there's no back edge from any descendant of b to any *proper* ancestor[a] of a.

[7] Note that the terms "**ancestor**" and "**descendant**" refer to the nodes' relationships in the DFS tree, and a node is considered an ancestor and a descendant of itself.

[8] We use the notation "$a \to b$" to denote the edge (a, b) while also emphasizing that "a is a parent."

- a is the root of a DFS tree and a has at least two children.

Take note of the second case regarding articulation points; it's easy to miss, but important.

[a] A proper ancestor of a is an ancestor distinct from a.

With these observations, we can now use DFS to compute the bridges and articulation points. First, we augment DFS with additional data:

- Let `disc`$[i]$ be the discovery time of i. (Identical to `start_time`$[i]$ above.)
- Let `low`$[i]$ be the lowest discovery time of any ancestor of i that is reachable from any descendant of i with a single back edge. If there are no such back edges, we say `low`$[i] =$ `disc`$[i]$.

Using `disc`$[i]$ and `low`$[i]$, we can now state precisely when something is a bridge or an articulation point:

- Let (a, b) be a tree edge, where a is the parent. Then (a, b) is a bridge iff `low`$[b] >$ `disc`$[a]$.
- Let a be a node. Then a is an articulation point iff either of the following is true:
  - a is not the root of a DFS tree and a has a child b such that `low`$[b] \geq$ `disc`$[a]$.
  - a is the root of a DFS tree and a has at least two children.

Thus, the only remaining task is to compute `disc`$[i]$ and `low`$[i]$ for each i. But we can compute these values *during the DFS*, like so:

```
def DFS(i, p):
    disc[i] = low[i] = time++

    children = 0
    has_low_child = False
    for j in adj[i]:
        if disc[j] == -1:
            # this means (i, j) is a tree edge
            DFS(j, i)

            # update low[i] and other data
            low[i] = min(low[i], low[j])
            children++

            if low[j] > disc[i]:
                mark edge (i, j) as a bridge

            if low[j] >= disc[i]:
                has_low_child = True

        else if j != p:
            # this means (i, j) is a back edge
            low[i] = min(low[i], disc[j])

    if (p == -1 and children >= 2) or (p != -1 and has_low_child):
        mark i as an articulation point

def bridges_and_articulation_points():
    time = 0
    for i from 0 to n-1:
        disc[i] = -1

    for i from 0 to n-1:
        if disc[i] == -1:
            DFS(i, -1)
```

This procedure now correctly identifies all bridges and articulation points in the graph!

An important thing to understand here is how `low`[i] is being computed. Please ensure that you understand it.

2 Connectivity in Directed Graphs

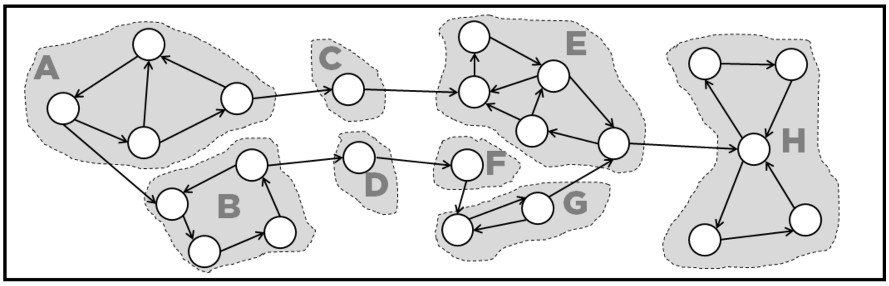
Since connectivity with undirected graphs is so boring, let's consider directed graphs instead. The important difference is that paths are not *reversible* anymore, so the nice picture of "connected components" we had above no longer applies.

Let's explain it a bit more formally If we let $a \rightsquigarrow b$ mean "there is a path from a to b", then $\rightsquigarrow$ is still reflexive and transitive, but is not necessarily symmetric any more. Thus, "equivalence classes" are not well-defined any more.

But we can fix this: If we let $a \sim b$ mean "$a \rightsquigarrow b$ and $b \rightsquigarrow a$", i.e., "there is a path from a to b and a path from b to a", then it's easy to verify that $\sim$ is reflexive, symmetric and transitive, hence, it's an equivalence relation!

If $a \sim b$, then we say **a and b are strongly connected**. A **strongly connected component**, or **SCC**, is a maximal set of nodes that is pairwise strongly connected. In other words, the SCCs are the equivalence classes under $\sim$. SCCs are the best analogue of connected components in directed graphs.

The following offers a picture of the strongly connected components of an example directed graph:



strongly connected components (SCC)

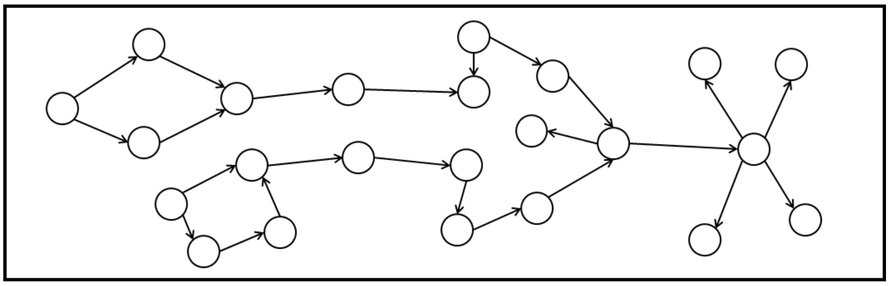
It is instructive to verify that the pairs of nodes in each highlighted region are indeed strongly connected.

Note that, this time, there could be edges from one SCC to another. This is more exciting than before!

2.1 Cycles and DAGs

A **cycle** is a nontrivial path from a to itself. We say a graph is **acyclic** if it has no cycles. A directed acyclic graph is called, well, a **directed acyclic graph**, or **DAG**.

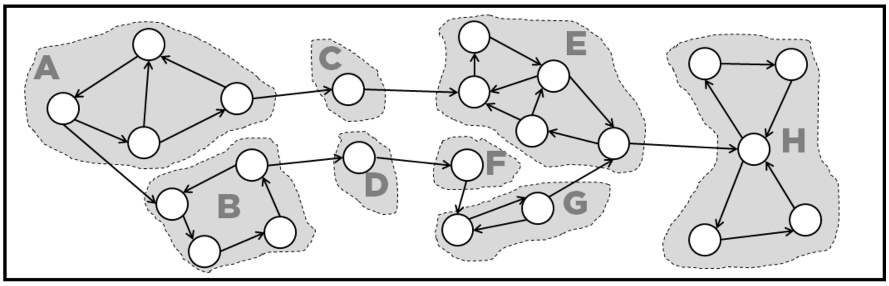
Here's an example of a DAG:



directed acyclic graph (DAG)

One nice thing about DAGs is that you can *serialize* the nodes: you can find a total order of the nodes such that every edge *goes forward*, that is, every edge connects a node to a further node in the total order. This is called a **topological sort**, or *toposort*, of a graph. You probably already know how to compute the topological sort in linear time using simple data structures, but we will learn another algorithm later on.

Now, let's go back to our previous example:



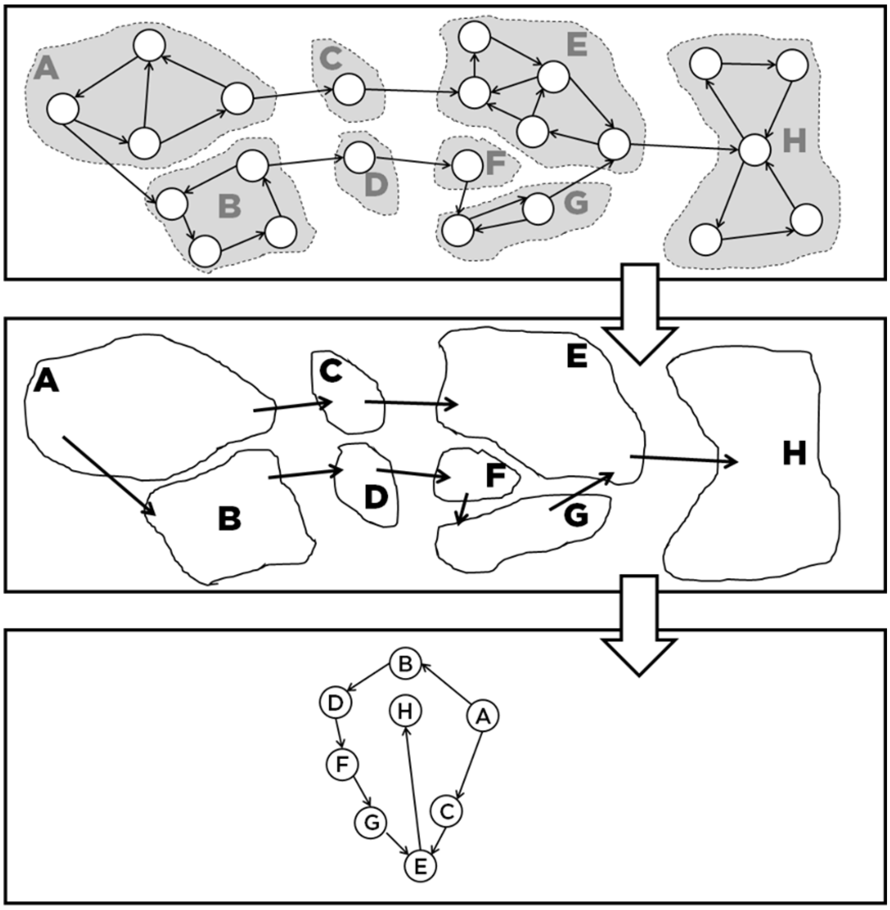
strongly connected components (SCC)

Suppose we "shrink" each SCC into a single node, and for every edge $a \to b$ connecting two nodes from different SCCs, we add an edge $\text{SCC}_a \to \text{SCC}_b$, where SCC_x is the node of the SCC containing x. We then obtain a new, smaller graph. My question is: can there be a cycle in a graph formed this way?

It turns out that there can't be any cycle in such a graph! The simple reason is that if there were a cycle, then the SCCs in that cycle could be compressed further into a single SCC, contradicting the fact that they're maximal. Thus, a cycle cannot exist.

Hence, if we shrink the SCCs into a single node, then we get a DAG, which we'll just call the **DAG of SCCs**.[9]

[9] You may also see this called the **condensation** of the graph. Since $\sim$ is an equivalence relation, you may even see it called as "**the graph modulo $\sim$".**



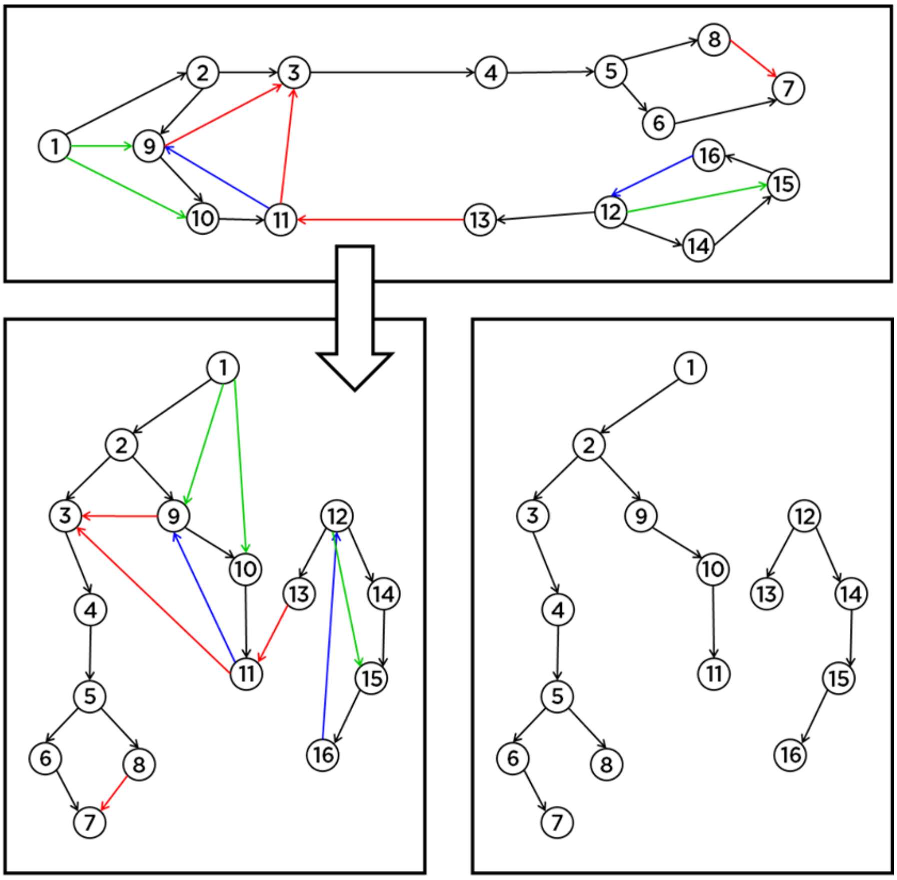
DAG of SCCs

Note that this is already more interesting and has more structure than in the undirected case, where shrinking the connected components results in just a graph with no edges!

2.2 Depth-first search revisited (directed version)

Let's discuss how DFS works in the case of directed graphs. It turns out that the DFS forest in a directed graph provides a similar set of useful information as in the undirected case.

For instance, let's look at how the "DFS forest" might look like in a directed graph:



tree edges, back edges, forward edges, cross edges

same graph but only tree edges are drawn

Note that there are still black and blue edges, representing tree and back edges. However, it looks like there are two new types of edges! It seems that the DFS on a directed graph classifies the edges into one of *four* types this time:

1. The black edges are the **tree edges**, which are the edges genuinely traversed by the DFS, i.e., $i \to j$ is a tree edge if the first time j is visited is through i.
2. The blue edges are the **back edges**, which are edges that point to an ancestor of a node in the DFS forest.
3. The green edges are the **forward edges**, which are edges that point to a descendant of a node in the DFS forest.[10]
4. The red edges are the **cross edges**, which are edges that point to neither an ancestor nor a descendant of a node in the DFS forest.

Exercise 2.1. Explain why there are no forward and cross edges in any DFS forest of any undirected graph.

Now, let's look at how a DFS procedure could identify these edges:

[10] In some applications, it's sometimes convenient to consider tree edges as forward edges as well.

```
def DFS(i):
    # perform a DFS starting at node i

    start_time[i] = time++

    for j in adj[i]:
        if start_time[j] == -1:
            mark (i, j) as a tree edge
            DFS(j)
        else if finish_time[j] == -1:
            mark (i, j) as a back edge
        else if finish_time[j] > start_time[i]:
            mark (i, j) as a forward edge
        else:
            mark (i, j) as a cross edge

    finish_time[i] = time++

def DFS_all():
    time = 0
    for i from 0 to n-1:
        start_time[i] = -1
        finish_time[i] = -1

    for i from 0 to n-1:
        if start_time[i] == -1:
            DFS(i)
```

Notice that we use the values `start_time[i]` and `finish_time[i]` to distinguish between back, forward and cross edges. In particular, assume `start_time[j] != -1`. This means that we have already started visiting node j. Thus, in the inner **for** loop above,

- If `finish_time[j] == -1`, then j is being visited while we're on i, which means j must be an ancestor of i in the DFS forest, so $i \to j$ is a **back edge**.
- If `finish_time[j] != -1` and `finish_time[j] > start_time[i]`, then we deduce that `start_time[i] < start_time[j] < finish_time[j]` (How?). This now says that j's visitation is already finished, and its visitation period is completely contained in the i's visitation period, so it means that j must be a descendant of i, and $i \to j$ is a **forward edge**.
- If `finish_time[j] != -1` and `finish_time[j] < start_time[i]`, then j's visitation is already finished even before we started visiting i, hence j is neither an ancestor or descendant of i, and $i \to j$ is a **cross edge**.

Exercise 2.2. Explain why there can't be any two nodes i and j such that

$$\texttt{start\_time}[i] < \texttt{start\_time}[j] < \texttt{finish\_time}[i] < \texttt{finish\_time}[j]$$

in the DFS for both directed and undirected graphs. (In other words, if the visiting intervals of i and j intersect, then one must contain the other.)

This fact is important when we deduced that `start_time[i]` < `start_time[j]` < `finish_time[j]` < `finish_time[i]` in the "forward edge" case above.

The running time is still $\mathcal{O}(n+e)$, but along the way, we have obtained useful information about our directed graph!

As with the undirected case, note that the implementation above is written *recursively*. In some large trees, stack overflow might be a concern, especially if the stack size is limited. In those cases, you might want to implement an *iterative* version.[11]

2.3 Cycle finding

Now, let's go back to discussing cycles. Given a directed graph, how do we detect if it has a cycle? Note that it already looks more interesting than the undirected case, and in fact there are many interesting approaches.

You probably already knew one approach, which is to run a toposort algorithm on the graph, and check if it failed. The toposort algorithm fails if and only if there is a cycle in the graph, hence this correctly solves our problem! But if you were asked to actually find a cycle, then it could get tricky depending on the toposort algorithm used.

But it's still worthwhile to discuss additional approaches to this problem. For instance, a simple algorithm arises from the following insight:

Exercise 2.3. Show that there is a cycle if and only if there is a back edge in the DFS forest.

Hint: Consider the DFS forest, and suppose there are no back edges. Can you somehow find a cycle using the only the remaining edge types?

Hint 2: Consider the sequence of `finish_time`s of the nodes you visit while only using forward edges, cross edges, and tree edges.

Thus, we can detect a cycle by performing a DFS (like above) and stopping once we find a back edge! Another advantage of this is that it's easy to actually find a cycle if one is detected.[12]

Exercise 2.4. How do you find a cycle once the algorithm detects that there exists one?

In fact, by pursuing this idea further, you can use a DFS to actually extract a toposort of a DAG: Just order the nodes by decreasing finishing time! Think about why that works.

2.4 Floyd's cycle finding algorithm

Let's restrict ourselves to a special kind of graph. Specifically, let's only consider graphs where each node has exactly one outgoing edge. Let's call such graphs **function graphs** because on such a graph, we can define a function $f : V \to V$ where $f(x) = y$ if and only if $x \to y$ is an edge.[13] Since there is exactly one outgoing edge from each node, f is well-defined. Conversely,

[11] Please see the footnote on the undirected case version for one way of dealing with this.

[12] You can also detect and find a cycle using BFS, although we will leave it to you to discover.

[13] We can also consider more general graphs with *at most one* outgoing edge per node. We can convert such graphs into function graphs by adding a new node, say *trash*, and pointing all nodes without an outgoing edge to *trash* (including *trash* itself).

every function $f : S \to S$ corresponds to a function graph whose node set is S and whose edges are $\{(x, f(x)) : x \in S\}$ (we're implicitly allowing self-loops here, but that's fine).

Now, starting at any node, there's only one path we can follow. In terms of f, starting at some node x and following the (only) path corresponds to iteratively applying f on x, thus, the sequence of nodes we visit is[14]

$$(x, f(x), f(f(x)), \dots, f^n(x), \dots).$$

Now, if we assume that our node set is finite, then this will eventually repeat, i.e., there will be indices $0 \le i < j$ such that $f^i(x) = f^j(x)$.[15] This is just the pigeonhole principle. In fact, once this happens, the subsequence $(f^i(x), f^{i+1}(x), \dots, f^{j-1}(x))$ will repeat forever. This always happens regardless of which node you start at.

This gives rise to a natural question: Given a starting point x, when is the first time that a repeat happens? Furthermore, how long is the cycle? To make the question more interesting, suppose we don't know anything about f apart from:

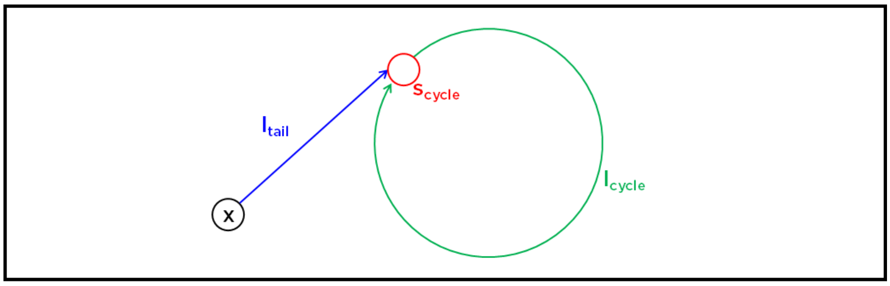
1. f's domain and codomain are the same and are finite.
2. We can evaluate $f(x)$ at any x. (For this discussion, we'll assume we can do it in $\mathcal{O}(1)$.)

We can formalize the **cycle-finding problem** as follows:

Problem 2.1. Given a function f with the above properties, and a starting point x, compute the following values:

1. s_{cycle}, defined as the first node that repeats in the sequence.
2. l_{cycle}, defined as the length of the (repeating) cycle.
3. l_{tail}, defined as the distance from x to s_{cycle} (in terms of the number of edges).

We can visualize these values with the following:



cycle finding

A simple approach is to use BFS or DFS, which is equivalent to just following the (only) path and storing the visited nodes until we encounter a node we've already visited.

[14] Here, $f^n(x)$ is f applied to x a total of n times.
[15] Note that this is no longer true if the graph is infinite. Why?

```
# "Just walk" algorithm
def cycle_find(x):
    visit_time = new empty map
    time = 0
    s = x
    while not visit_time.has_key(s):
        visit_time[s] = time++
        s = f(s)

    s_cycle = s
    l_tail = visit_time[s]
    l_cycle = time - l_tail
    return (s_cycle, l_cycle, l_tail)
```

Assuming no preprocessing and no specialized knowledge on f, this is probably close to the fastest we can do. It needs $\mathcal{O}(l_{\text{tail}} + l_{\text{cycle}})$ time.

It also needs $\mathcal{O}(l_{\text{tail}} + l_{\text{cycle}})$ memory, but one might wonder if it can be improved upon. Amazingly, there's actually a way to compute it using $\mathcal{O}(1)$ memory, called **Floyd's cycle-finding algorithm**!

Now, you might ask, why the need for a fancy algorithm? Surely it's trivial to find an $\mathcal{O}(1)$-memory solution. Here's one:

```
# Bogo-cycle-finding algorithm
def cycle_find(x):
    for time in 1,2,3,4...
        s = x
        for i = 1..time:
            s = f(s)

        s_cycle = s

        s = x
        l_tail = 0
        while s_cycle != s:
            s = f(s)
            l_tail++

        if l_tail < time:
            l_cycle = time - l_tail
            return (s_cycle, l_cycle, l_tail)
```

Let's call this the **bogo-cycle-finding algorithm**. Although it might not be obvious why this works, clearly this is $\mathcal{O}(1)$ memory. Hmm, well, that's true, but that's not enough, because this is an incredibly slow solution! The idea is to use only $\mathcal{O}(1)$ memory without sacrificing running time.

It turns out that there's a simple algorithm that satisfies our requirements: **Floyd's algorithm**. This is also sometimes called the **tortoise and the hare algorithm**, since we will only use two pointers, called the *tortoise* and the *hare*, respectively.

The idea is that both the tortoise and the hare begin walking at the starting point, but the hare is twice as fast. This means that at the beginning, the hare might be quite ahead of the tortoise, but once they both enter the cycle, they will eventually meet. This means we have just found a node that belongs in the cycle.

At this point, you might now be able to compute l_{cycle} (just go around the cycle once). However, s_{cycle} and l_{tail} seem to be inaccessible just yet. But then the magical "second act" of the algorithm saves the day: Once we they meet, the hare *teleports* back to the starting point. They then proceed walking *at the same speed* and stop once they meet. This meeting point will be s_{cycle}!

Note that l_{tail} can easily be computed at the same time, by just counting the number of steps it took for them to meet there.

Here's the pseudocode:

```
# Floyd's cycle-finding algorithm
def cycle_find(x):
    tortoise = hare = x
    do:
        tortoise = f(tortoise)
        hare = f(f(hare))
    while tortoise != hare

    # teleport, and walk at the same speed
    # compute l_tail along the way
    hare = x
    l_tail = 0
    while tortoise != hare:
        tortoise = f(tortoise)
        hare = f(hare)
        l_tail++

    # store s_cycle
    s_cycle = hare

    # compute l_cycle by walking around once.
    l_cycle = 0
    do:
        hare = f(hare)
        l_cycle++
    while tortoise != hare

    return (s_cycle, l_cycle, l_tail)
```

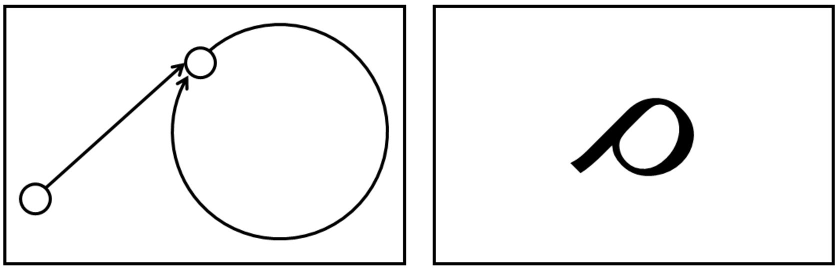
This clearly uses $\mathcal{O}(1)$ memory. We've also mentioned that this runs in $\mathcal{O}(l_{\text{tail}} + l_{\text{cycle}})$, but I didn't provide a complete convincing proof. It's not even clear why this correctly computes s_{cycle}. We leave it to you to prove its running time and correctness as an exercise!

Exercise 2.5. Prove that Floyd's cycle-finding algorithm correctly computes s_{cycle}, l_{cycle} and l_{tail}.

Exercise 2.6. Prove that Floyd's cycle-finding algorithm takes $\mathcal{O}(l_{\text{tail}} + l_{\text{cycle}})$ time.

2.4.1 Cycle-finding and factorization: Pollard's ρ algorithm

One interesting application of Floyd's algorithm lies in integer factorization. **Pollard's ρ algorithm** [16] is a factorization algorithm that can sometimes factorize numbers faster than trial-and-error division. The name comes from the shape of the path when starting at some value:



side-by-side comparison of pollard-rho diagram (left) and a tilted greek letter rho (right)

The algorithm accepts N, the number to be factorized, along with two additional parameters: a starting value s, and a function $f : \{0, 1, \ldots, N-1\} \to \{0, 1, \ldots, N-1\}$, which must be a polynomial modulo N. The algorithm then attempts to find a divisor of N. One issue with this algorithm is that *it's not guaranteed to succeed*, so you may want to run the algorithm multiple times, with differing s (and possibly f).

Suppose we want to factorize a large number N. Also, suppose $s = 2$ and $f(x) = (x^2 + 1) \bmod N$. Here is the pseudocode of Pollard's ρ-algorithm.

```
def try_factorize(N):
    x = y = 2 # starting point
    do:
        x = f(x)
        y = f(f(y))
        d = gcd(|x - y|, N)
    while d == 1

    if d == N:
        failure
    else:
        return d
```

One can clearly see f being used as the iteration function, and x and y are assuming roles that are similar to the tortoise and hare, respectively. If this fails, you could possibly try again with a different starting point, or perhaps a different f. (Of course, this will always fail if N is prime.)

[16] ρ is pronounced "rho".

A more thorough explanation of why this works, and why cycle-finding appears, can be seen on its Wikipedia page.[17]

2.5 Computing strongly connected components

So far we've discussed what SCCs are, along with some of their properties. But we haven't explained how to compute them yet. Unlike in the undirected case, a naïve BFS or DFS won't work here. (Why?)

We will discuss two algorithms. It's instructive to learn both, and then possibly choose your preferred algorithm later on.

We will be skipping some details of the proofs of correctness. Please see Appendix A for the complete proofs.

2.5.1 Kosaraju's algorithm

Here, we describe **Kosaraju's algorithm** which computes the strongly connected components of a directed graph. It performs two DFS traversals along the way, which we will call *phase one* and *phase two*.

In more detail, here are the steps of Kosaraju's algorithm:

1. Mark all vertices as not visited.
2. (*Phase one.*) Perform a DFS on the whole graph (in an arbitrary order). Take note of the *finishing times* of the nodes.
3. Reverse the directions of the edges.
4. Again, mark all vertices as not visited.
5. (*Phase two.*) Perform another DFS traversal, this time in decreasing order of their finishing time in phase one. Every time we start a new top-level DFS traversal `DFS(i)`, all the nodes visited in that run constitutes a strongly connected component.

Since we already know how to perform a DFS, this is easy to implement! Also, this runs in $\mathcal{O}(n+e)$ time. You might wonder why not $\mathcal{O}(n \log n + e)$ since we need to sort the nodes in decreasing order of finishing time, but there's actually no need to do that since we can simply push every node that we just finished visiting onto a *stack*. Then, in phase two, we simply pop from the stack to determine the order. This works because the nodes that are finished last will be the ones popped first from the stack! In fact, using this stack, we don't really even need to store the finishing times.

Here's a pseudocode of the algorithm:

```
def DFS(i, adj, group):
    # DFS starting at i, using the adjacency list 'adj'
    # then push all visited nodes to the vector 'group'
    visited[i] = True
    for j in adj[i]:
        if not visited[j]:
```

[17] https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm.

```
            DFS(j, adj, group)

    group.push(i)

def SCC_Kosaraju():
    # mark all nodes as unvisited
    for i from 0 to n-1:
        visited[i] = False

    stack = new empty vector
    for i from 0 to n-1:
        if not visited[i]:
            DFS(i, adj, stack)

    # create the reversal graph
    jda = new adjacency list
    for i from 0 to n-1:
        for j in adj[i]:
            jda[j].push(i)

    # reinitialize visited
    for i from 0 to n-1:
        visited[i] = False

    # do DFS again on jda, in decreasing order of finishing time
    sccs = new empty vector of vectors
    while not stack.empty():
        i = stack.pop()
        if not visited[i]:
            scc = new empty vector
            DFS(i, jda, scc)
            sccs.push(scc)
    return sccs
```

Now, why does it work? We'll provide a proof in Appendix A, but you may want to skip it if you want spend some time thinking about it yourself.

In my opinion, the most important part of the proof relies on the following claim:

Claim 2.7. Let A and B be SCCs. If there is a path from A to B, then for any DFS traversal, regardless of the order in which we visit the nodes, there will always be a node in A that has a greater finishing time than all the nodes in B.

The correctness of the algorithm relies on the order in which we visit the nodes in phase two. Let's denote by G^R the graph G with all its edges reversed.[18]

Exercise 2.8. Explain why the SCCs of G are exactly the same as the SCCs of G^R.

[18] G^R is also called the **transpose** of G.

Lemma 2.9. In phase two, whenever we start a top-level DFS in G^R, we do it in such an order that all the reachable nodes that belong to different SCCs have already been visited.

Claim 2.7 and Lemma 2.9 can be used to prove the correctness of Kosaraju's algorithm.

Theorem 2.10. Kosaraju's algorithm correctly computes the strongly connected components of a graph G.

Proof. In order to prove the correctness, we need to show that every time we begin a new DFS in phase two (line 37), we only visit the nodes of some SCC, say B, and not anything else.

So let's look at some SCC B, and consider the first time we visit any node in B. Note that the first time we visit a node in B is when we actually start the DFS on a node in B, not on a node in a different SCC that can reach B, otherwise, this would contradict Lemma 2.9.

Also, once we start a DFS in B, all nodes in B will be visited by this DFS (because they are reachable from each other), and only those in B will be visited, because all the nodes in the other SCCs reachable from B have already been visited (again according to Lemma 2.9). Therefore, whenever we start a new DFS, we visit exactly those nodes that belong to an SCC, and so the algorithm correctly identifies the SCCs. □

As a side note, Claim 2.7 can be repurposed to prove that we can get a topological sort of the DAG by ordering the nodes by decreasing finishing time:

Theorem 2.11. A topological sort of a DAG is obtained by performing a DFS on it and ordering the nodes in decreasing finishing time.

Proof. Claim 2.7 says that *for two SCCs A and B, if there is a path from some node in A to some node in B, then there is a node in A with a greater finishing time than all nodes in B.*

But the SCCs of a DAG consist of single nodes, thus A and B has exactly one element each, say a and b (respectively), so Claim 2.7 is equivalent to saying that *if there is a path from a to b, then a has a greater finishing time than b.* In particular, paths of length 1, i.e., single edges, point from a node to another node with a lower finishing time, hence ordering the nodes in decreasing finishing time results in a valid topological sort! □

As in Kosaraju's algorithm, you can construct the toposort without computing the finishing times explicitly by pushing the just-finished nodes onto a stack, and then reversing the stack in the end.

2.5.2 Tarjan's SCC algorithm

We describe another algorithm called *Tarjan's SCC algorithm.* Tarjan's algorithm also uses a DFS, but unlike Kosaraju's algorithm, it needs only one DFS traversal. It works by augmenting the DFS procedure with additional bookkeeping data that's enough to identify the SCCs.

Here's the pseudocode:

```
def DFS(i):
    disc[i] = low[i] = time++

    stack.push(i)
    instack[i] = True
    for j in adj[i]:
        if disc[j] == -1:
            DFS(j)
            low[i] = min(low[i], low[j])
        else if instack[j]:
            low[i] = min(low[i], disc[j])

    if low[i] == disc[i]:
        get_scc(i)

def get_scc(i):
    # pop the stack until you pop i, and collect those as an SCC
    scc = new empty vector
    do:
        j = stack.pop()
        instack[j] = False
        scc.push(j)
    while j != i
    SCCs.push(scc)

def SCC_Tarjan():
    stack = new empty vector
    sccs = new empty vector of vectors
    time = 0
    for i from 0 to n-1:
        disc[i] = -1
        instack[i] = False

    for i from 0 to n-1:
        if disc[i] == 0:
            DFS(i)
    return sccs
```

A couple of things to notice here:

- This algorithm uses the `disc` and `low` arrays, just like in our algorithm for bridges and articulation points! However, there's also a stack this time, which we simply call `stack`.
- We extract a single SCC whenever `get_scc` is called. We pop the stack a couple of times to do this.
- A certain condition `low[i] == disc[i]` is checked before `get_scc(i)` is called. We will see what this means later on.
- The `stack` stores all nodes that have been discovered (in order of discovery time), but haven't been put into their appropriate SCCs yet. This is why whenever we call `DFS(i)`, we push i onto `stack`, but at the end of `DFS(i)`, we don't necessarily pop i yet. (Although `get_scc(i)` definitely pops i if it is called.)

`disc`$[i]$ and `low`$[i]$ are defined similarly as before:

- Let `disc`$[i]$ be the discovery time of i.

- Let low$[i]$ be the lowest discovery time of any ancestor of i that is reachable from any descendant of i with a single back edge. If there are no such back edges, we say low$[i]$ = disc$[i]$.

It's worth mentioning that low$[i]$ ignores forward edges or cross edges.

But the stack plays a crucial role in this algorithm, since it turns out that it neatly groups the nodes according to their containing SCCs. More specifically, for every SCC containing some nodes in the stack, all its nodes can be found in consecutive positions in it. You can kinda see why this is true if you imagine how the algorithm plays out during the DFS traversal, and may help you get a feel for why the algorithm is correct.

But if you're looking for a complete formal proof of correctness, then as it turns out, I struggled a lot with writing it formally,[19] and in the end, my proof isn't really that enlightening, so I'm leaving it in Appendix A. Someday I hope to be able to simplify it further.

The following property of the DFS forest is important to the proof.

Lemma 2.12. The nodes of any SCC form a rooted "subtree" in the DFS forest.[a]

[a] Note that "subtree" means something slightly different here. This doesn't mean that all nodes *down to the leaves* are part of the SCC. It means that, if you consider only the nodes of some SCC and ignore the rest (including possibly some nodes below them), then you get a rooted tree.

Now, clearly, the time complexity is $\mathcal{O}(n+e)$. However, although the time complexity is the same with Kosaraju's algorithm, Tarjan's algorithm can still be seen as somewhat of an improvement over Kosaraju's algorithm in a few ways:

- Only one DFS is required.
- We don't have to build the reversal graph (which could be memory-intensive).
- The correctness of the algorithm is more easily seen, once you get a feel for how low and disc work.[20]
- It's usually faster in practice.

Exercise 2.13. In the pseudocode for Tarjan's algorithm, is line 11 being run only when $i \to j$ is a back edge? Is it being run on forward edges as well? What about cross edges? If yes on either, does that mean Tarjan's algorithm is incorrect? Why or why not?

Exercise 2.14. What happens if we move the line `instack[j] = False` from line 21 to the very end of the DFS function?

2.5.3 Which algorithm to use?

Now that you know both algorithms, which one should you now use? Well, it's really up to you. For me, the choice here is essentially whether to choose an easy-to-implement solution or a slightly faster solution. I usually choose Kosaraju's algorithm since it's easier to understand (and remember), although I know a lot of others who prefer Tarjan. In fact, it seems I'm in the minority. So it's up to you if you want to go mainstream or hipster.

[19] and in fact, I feel like there are still a few fundamental errors in it

[20] at least to some; honestly it took me quite some time to understand it myself.

2.6 DAG of SCCs

Finally, it's at least worth mentioning how we can construct the DAG of SCCs. Once we can compute the SCCs using any of the algorithms above, we can now construct the DAG of SCCs. In high level, the steps are:

1. Compute the SCCs of the graph.
2. "Shrink" the SCCs into single nodes.
3. Remove the self-loops and duplicate edges.
4. The resulting graph is the DAG of SCCs.

The "steps" here might be too vague, especially the "shrinking" part, but actually, these can easily be transformed into a slightly more explicit procedure. Let's restate the steps above more explicitly (I call this the *engineer approach*):

1. Compute the SCCs of the graph. Suppose there are k SCCs.
2. Compute the array f, where $f[i]$ represents the index of the SCC containing i.
3. Construct a new graph with k nodes and initially 0 edges.
4. For every edge $a \to b$ in the original graph, if $f[a] \neq f[b]$, then add the edge $f[a] \to f[b]$ in the new graph (if it doesn't already exist).
5. The new graph is now the DAG of SCCs.

Congratulations! You may now use the DAG of SCCs (if you need it).

Note that this still runs in $\mathcal{O}(n + e)$ time.

3 Biconnectivity in Undirected Graphs

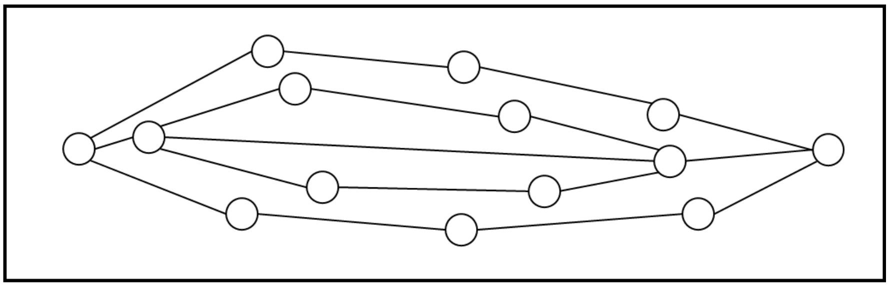
Let's return to undirected graphs. We mentioned that the directed case is more interesting, but that's not entirely true! Here we'll describe another aspect of connectivity in undirected graphs.

We say that a connected undirected graph is **2-edge-connected** if removing any edge doesn't disconnect the graph. Alternatively, an undirected graph is 2-edge-connected if it is connected and doesn't have any bridges.

We say that a connected undirected graph is **biconnected** if removing any vertex doesn't disconnect the graph. Alternatively, an undirected graph is biconnected if it is connected and doesn't have any articulation points.

Note that these are stronger notions than mere connectivity. Having these properties tells us that the graph is more interconnected than usual (since there are no weak points).

Here's an example of a graph that's both 2-edge-connected and biconnected:



biconnected graph

The two notions are similar, but be careful not to confuse the two! They're not exactly the same.

> **Exercise 3.1.** Explain the relationship and differences between 2-edge-connected and biconnected components.

We can generalize the definitions above naturally:

- An undirected graph is k**-edge-connected** if removing less than k edges doesn't disconnect the graph.
- An undirected graph is k**-vertex-connected**, or k**-connected**, if it has more than k nodes and removing less than k vertices doesn't disconnect the graph.

Being biconnected and 2-connected are the same except for extreme cases: connected graphs of ≤ 2 nodes are considered biconnected but not 2-connected. (This may seem like a bizarre exception, but it turns out useful in some algorithms related to biconnectivity.) Also, note that being 1-connected is the same as being connected (except when the graph has a single node).

3.1 Menger's theorem

There's a nice result concerning k-edge-connectivity and k-vertex-connectivity called **Menger's theorem**. The theorem provides two results:

Theorem 3.2 (Menger)**.** The following are true in any undirected graph:

1. Let x and y two *distinct* vertices. The minimum number of *edges* whose removal disconnects x and y equals the maximum number of pairwise *edge*-independent paths from x to y.
2. Let x and y two *distinct, nonadjacent* vertices. The minimum number of *vertices* whose removal disconnects x and y equals the maximum number of pairwise *vertex*-independent paths from x to y.

Aren't these results nice? Even nicer, they hold for both directed and undirected graphs!

In particular, they imply that:

1. An undirected graph is k-edge-connected if and only if for every pair of vertices x and y, it is possible to find k edge-independent paths from x to y.
2. An undirected graph is k-vertex-connected if and only if for every pair of vertices x and y, it is possible to find k vertex-independent paths from x to y.

We won't be proving these properties for now (they will be proven in a future module on network flows), but it will probably be useful to know them now.[21]

3.2 Robbins' theorem

Another interesting fact about 2-edge-connected graphs is that *you can orient*[22] *the edges of the graph such that it becomes strongly connected.* Such graphs are called **strongly orientable**. In fact, the converse is true as well: every strongly orientable graph is 2-edge-connected. This equivalence is called **Robbins' theorem** and is not that difficult to prove.

Theorem 3.3 (Robbins' theorem)**.** A graph is strongly orientable if and only if it is 2-edge-connected.

Proof. If a graph has a bridge, then obviously there's no strong orientation. But if a graph has no bridges, then let's perform a DFS and orient the edges according to the way we traversed it; i.e., tree edges point away from the root, and back edges point up the tree. (Remember that there are no forward and cross edges in undirected DFS.)

That there are no bridges means that for every tree edge (a, b) in the DFS forest, $\texttt{low}[b] \leq \texttt{disc}[a]$. This means that from any node b, one can always climb to an ancestor of b (using at least one back edge). By repeating this process, one can reach the root from any node b. Since the root can reach all the other nodes as well (just using tree edges), it means the graph is strongly connected! As a bonus: this DFS-based proof can easily be converted into a DFS-based algorithm to compute a strong orientation of the graph. □

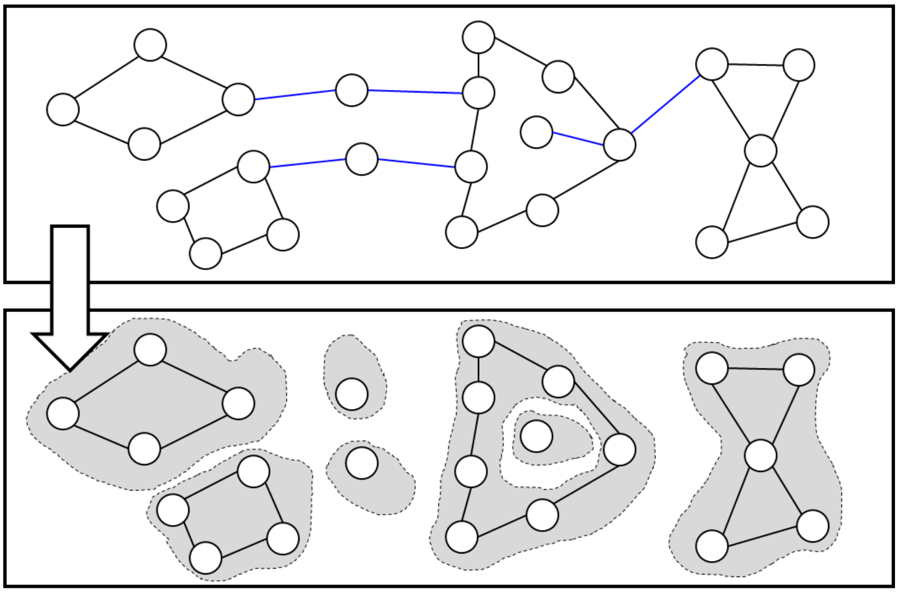
[21] These properties are related to other similar results in graph theory such as the **min-cut max-flow theorem** (the minimum cut equals the maximum flow) and **König's theorem** (the size of the minimum vertex cover equals the size of the maximum matching in bipartite graphs).

[22] To "orient" an edge means to choose its direction, hence the edge becomes directed.

3.3 2-edge-connected components

A **2-edge-connected component** is a maximal subgraph that is 2-edge-connected. Given a graph, a natural question is: How can we find the 2-edge-connected components, and what can we say about their structure? Let's assume the graph is connected for simplicity; we can simply do the same algorithm for each connected component otherwise.

Well, by definition, a 2-edge-connected component cannot have bridges, so let's say we remove the bridges in the original graph first. (We can find those with DFS.) Then what we're left with are subgraphs that don't contain any bridges, hence are 2-edge-connected![23]

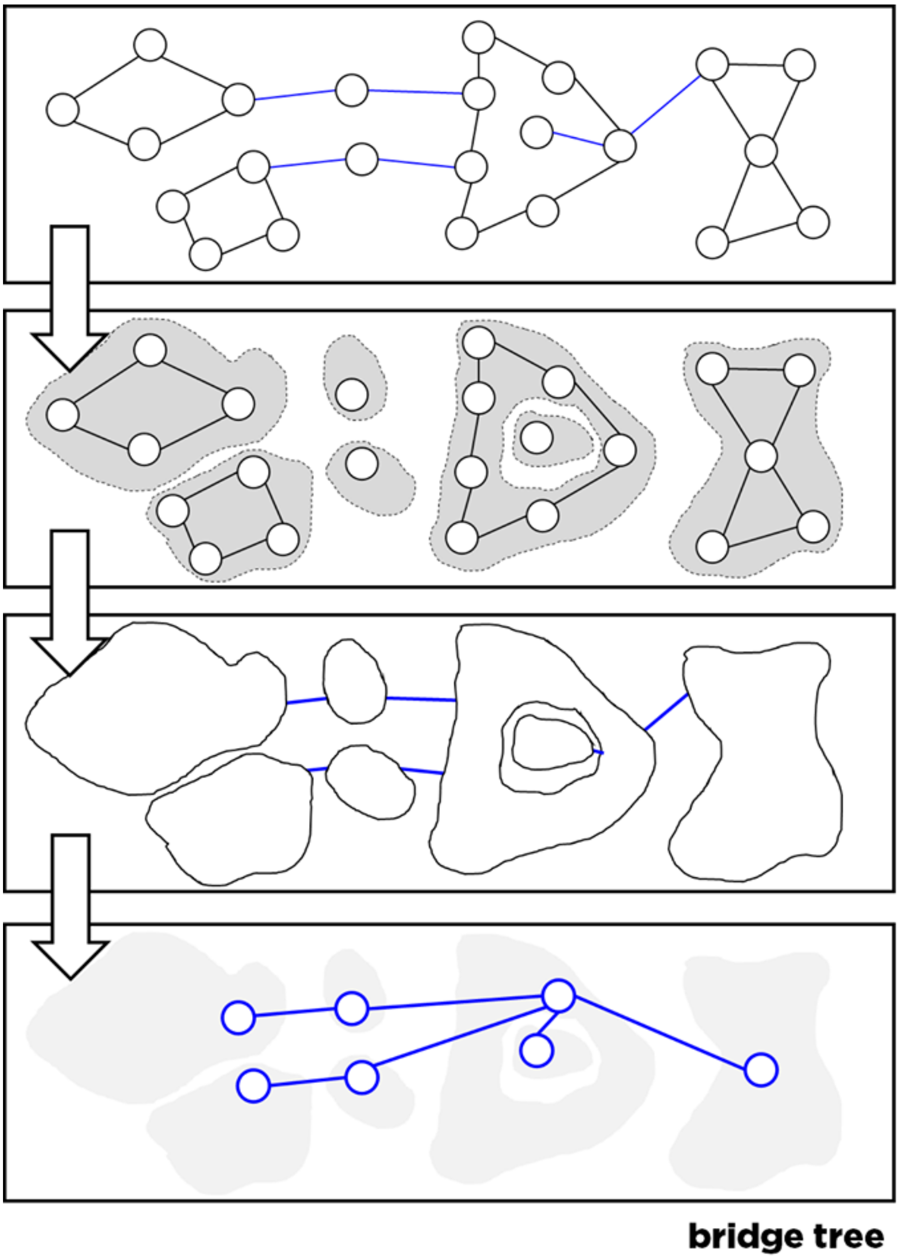


removing bridges results in 2-edge connected subgraphs

Furthermore, suppose we "shrink" each 2-edge-connected component into a single node. Then, observing that a bridge is not a part of any cycle (otherwise it couldn't have been a bridge at all), we find that the resulting graph is actually a *tree*!

We can call this resulting tree the **bridge tree** of the graph.

[23] Actually, it doesn't follow from the definition that removing bridges doesn't introduce new bridges, but it shouldn't be hard to convince oneself of this.



As just described, constructing this tree is straightforward:

1. Detect all the bridges.
2. Remove (burn) all the bridges
3. “Shrink” the remaining connected components into single nodes.
4. Put the bridges back.
5. The resulting graph is the bridge tree.

Like before, the whole process might be too vague, but an engineer approach will work just as well:

1. Detect all the bridges.

2. Remove all the bridges temporarily; store them in an array for now.
3. Collect all connected components. Assume there are k connected components.
4. Construct an array f, where $f[i]$ denotes the index of the connected component containing i.
5. Construct a graph with k nodes and initially 0 edges.
6. For every bridge (a, b), add the edge $(f[a], f[b])$ to the graph.
7. The resulting graph is the bridge tree.

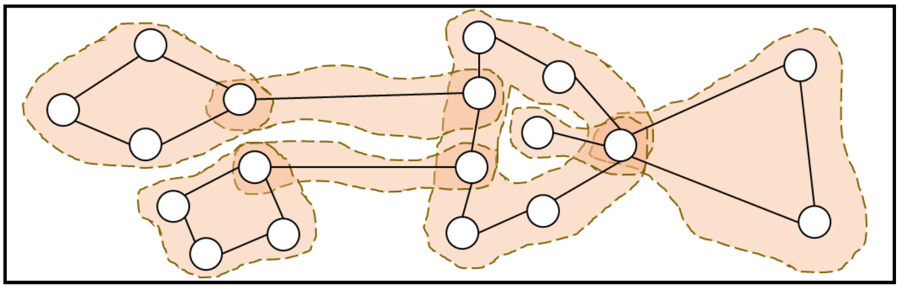
Congratulations! You have now constructed the bridge tree and you may now use it to solve some problems.

> **Exercise 3.4.** Show that removing a bridge doesn't turn any non-bridge edge into a bridge.

3.4 Biconnected components

A **biconnected component**, or **BCC**, is a maximal subgraph that is biconnected. A natural question is: How can we find the BCCs, and what can we say about their structure? Again, assume the graph is connected for simplicity.

The structure of the BCCs of a graph is quite different from the structure of the 2-edge-connected components. In particular, BCCs can overlap! See the following picture:



overlapping BCCs

So given this complication, how can we compute the BCCs?

The natural first step is to compute all the articulation points. After that, notice that pairs of BCCs can only overlap on at most one node, and that node must be an articulation point.

However, it looks like we're stuck. Even given that information, there doesn't seem to be a simple way to compute the BCCs.

Fortunately, we can actually modify DFS (again!) to compute the BCCs *alongside* the articulation points! The key here is to think of a BCC as a set of *edges*, rather than a set of nodes; this way, each edge belongs to exactly one BCC, which is very helpful. Furthermore, similar to Tarjan's SCC algorithm, we can again collect the *edges* that belong to the same BCC on a stack, and pop whenever we detect an articulation point!

```
def DFS(i, p):
    disc[i] = low[i] = time++

    children = 0
    has_low_child = False
    for j in adj[i]:
        if disc[j] == -1:
            stack.push(edge(i, j))
            DFS(j, i)

            low[i] = min(low[i], low[j])
            children++

            if low[j] >= disc[i]:
                has_low_child = True
                get_bcc(edge(i, j))

        else if j != p:
            low[i] = min(low[i], disc[j])

    if (p == -1 and children >= 2) or (p != -1 and has_low_child):
        mark i as an articulation point

def get_bcc(e):
    # pop the stack until you pop e, and collect those as a BCC
    bcc = new empty vector
    do:
        E = stack.pop()
        bcc.push(E)
    while E != e
    bccs.push(bcc)

def articulation_points_and_BCCs():
    stack = new empty vector
    bccs = new empty vector of vectors
    time = 0
    for i from 0 to n-1:
        disc[i] = -1

    for i from 0 to n-1:
        if disc[i] == -1:
            DFS(i, -1)
    return bccs
```

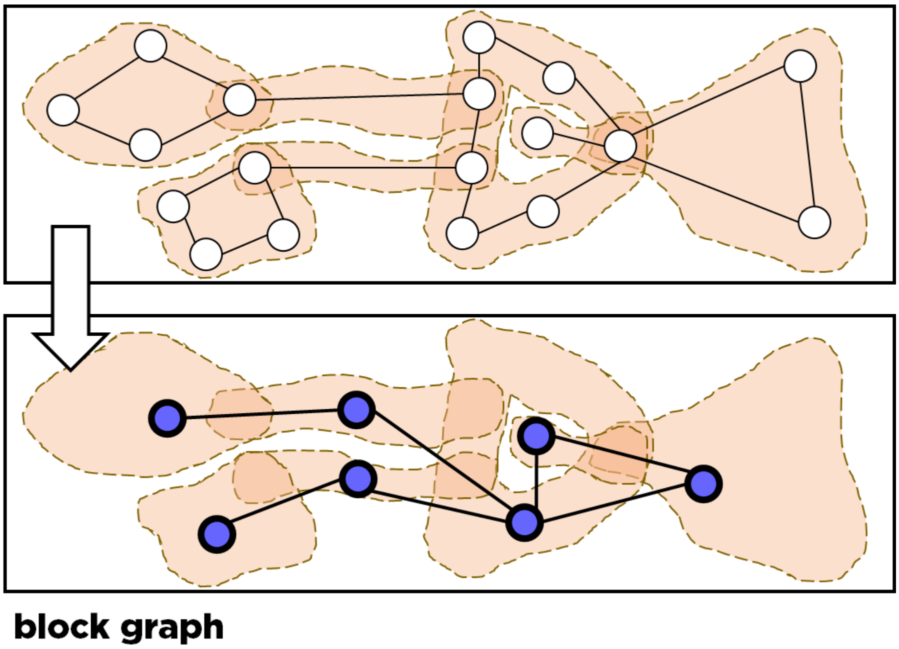
After this $\mathcal{O}(n+e)$ process, we now have the articulation points and the BCCs of the graph!

3.4.1 Block graph

The next natural question is: what structure do the BCCs have? Remember that they can only overlap in at most one node, and this must be an articulation point. Therefore, the articulation points somehow serve the role of "edges" in the same way the bridges were the "edges" in the bridge tree.

A natural graph structure is constructed by compressing each BCC into a single node, and declaring that two BCCs are adjacent iff they share an articulation point in common. This is called the **block graph**, and it's easy to see that this forms a valid connected graph structure on the BCCs, and that it encodes a bit of information about the connectivity of the graph as a whole. Constructing the block graph from this definition is straightforward.[24]

Here's an example of a block graph:



Unfortunately, as you can see above, the block graph is not (always) a tree! That's sad :(And that's the reason it's not called a "block tree".

In fact, what's even sadder is that this can fail very badly. Specifically, a block graph formed from a graph with n nodes and $\mathcal{O}(n)$ edges can have up to $\Omega(n^2)$ edges![25] So that's really sad.

> **Exercise 3.5.** Show that the block graph formed from a graph with n nodes and $\mathcal{O}(n)$ edges can have $\Omega(n^2)$ edges.

3.4.2 Block-cut tree

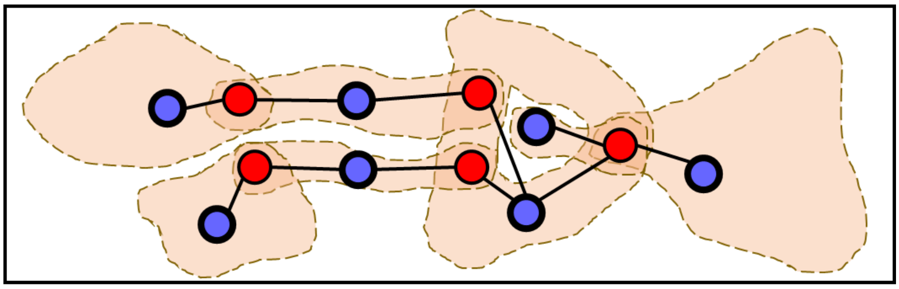
Fortunately, there's an alternative structure on the BCCs where the number of edges doesn't explode. The key is to *represent the articulation points* as nodes in their own right. Thus, in our compressed graph, we have a node a_x for each articulation point x and a node b_Y for each BCC Y.[26] We say there is an edge between a_x and b_Y if the BCC Y contains the articulation point x. It can be seen that this structure is a tree (why?), and this is more commonly known as the **block-cut tree**. ("Block" stands for BCC and "cut" stands for cut vertex.)

[24] Use the engineer approach.

[25] We say $f(n) = \Omega(g(n))$ if there exists a constant c such that for all sufficiently large n, $f(n) \geq c \cdot g(n)$. Thus, "$f(n) = \Omega(g(n))$" is the same as "$g(n) = \mathcal{O}(f(n))$". Informally, you can think of $\mathcal{O}$ as "asymptotically at most" and Ω as "asymptotically at least".

[26] Note that this "compressed" graph can actually be larger than the original graph!

Here's an example of a block-cut tree:



block-cut tree

Thankfully, the block-cut tree is indeed a tree, thus it doesn't have that many more nodes and edges than the original one. And it still encodes a good amount of information about the connectivity between the BCCs. As an added bonus, the articulation points are represented as well!

Exercise 3.6. Explain why the block-cut tree is really a tree.

Constructing a block-cut tree from the definition is straightforward. (Use the engineer approach.)

4 Problems

Solve as many as you can! Ask me if anything is unclear.[27] In general, the harder problems will be worth more points.

4.1 Warmup problems

Most of these are straightforward applications of some of the algorithms discussed. You may want to use these problems to test your implementations of the algorithms above.

There is no point target for warmup problems, but the points here will be counted as bonus points in coding problems.

W1 [1★] **Test:** UVa 10731

W2 [1★] **Tourist Guide:** UVa 10199 (implement a linear-time solution)

W3 [1★] **Network:** UVa 315 (implement a linear-time solution)

W4 [1★] **Lonely People in the World:** `https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/lonely-people-in-the-world`

W5 [1★] **Putogethers:** `https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/putogethers`

[27] Especially for ambiguities! Otherwise, you might risk getting fewer points even if you *technically* answered the question correctly.

4.2 Non-coding problems

No need to be overly formal in your answers; as long as you're able to convince me, it's fine!

Answering any of the Exercises in the text above will give you bonus points! Each one of them is worth at least [1★].

First-timers, feel free to solve for extra points. Veterans, solve at least [45★].

N1 [2★] Prove or disprove: an undirected graph is acyclic if and only if every edge is a bridge.

N2 [2★] Prove or disprove: an undirected graph is acyclic if and only if every node of degree > 1 is an articulation point.

N3 [2★] Prove or disprove: each endpoint of a bridge is either of degree 1 or an articulation point.

N4 [2★] Prove or disprove: every articulation point is an endpoint of some bridge.

N5 An **arborescence** is a directed graph that is formed by starting with an undirected tree, choosing some node to be a root, and *orienting* the edges to point away from the root. A **polytree** is a directed graph such that if you ignore the directions of the edges, you get an undirected tree.

(a) [2★] Prove or disprove: every arborescence is a polytree.

(b) [2★] Prove or disprove: every polytree is an arborescence.

(c) [2★] A **multitree** is a directed graph such that if you pick any node t, and remove all nodes and edges not reachable from t, you get an arborescence. Prove or disprove: every polytree is a multitree.

(d) [2★] Prove or disprove: every multitree is a polytree.

N6 [2★] Prove or disprove: Every 2-edge-connected graph is biconnected.

N7 [2★] Prove or disprove: Every biconnected graph is 2-edge-connected.

N8 [2★] Prove or disprove: Every 2-connected graph is 2-edge-connected.

N9 [2★] Show that the block-cut tree can have both more nodes and edges than the original graph.

N10 [4★] Let G be a graph with $n > 0$ nodes and $e > 0$ edges. Suppose its block-cut tree has n' nodes and e' edges. Find an upper bound on the ratios n'/n and e'/e. Also, what are the tightest possible upper bounds? Show why your upper bounds and least upper bounds are correct.[28]

N11 These problems are about the cycle-finding algorithms.

(a) [3★] Prove that the bogo-cycle-finding algorithm correctly computes s_{cycle}, l_{cycle} and l_{tail}. Determine its running time.

(b) [2★] Exactly how many calls to f are done by the bogo-cycle-finding algorithm as written above, in terms of l_{tail} and l_{cycle}?

[28] Hint for those who aren't familiar with proving least upper bounds: Suppose you want to show that some number u is a least upper bound to n'/n. Then you have to show that (a) u is an upper bound, and (b) n'/n can get *arbitrarily close* to u. You can do the latter by exhibiting an infinite family/sequence of graphs where n'/n approaches u as n gets large.

(c) [3★] Exactly how many calls to f are done by Floyd's cycle-finding algorithm as written above, in terms of l_{tail} and l_{cycle}?[29]

N12 The goal here is to find an algorithm that computes $a^x \bmod m$ for any integer $x \geq 0$ quickly, given a and m.

(a) [4★] Show that for any two integers $m > 0$ and a, there exist two integers s and t such that $0 \leq s < t \leq m$ and $a^s \equiv a^t \pmod{m}$. *Hint:* Consider the function $f(x) := ax \bmod m$.

(b) [2★] Show that, in addition, you can find s and t that also satisfy $t \leq \max(1, m-1)$ and $s \leq \lfloor \log_2 m \rfloor$.

(c) [3★] Show that each bound in **N12**b is tight for some $m \geq 2^{16}$.

(d) [2★] Show that each in **N12**b bound is tight for infinitely many ms.

(e) [4★] Find an algorithm that, given a and m, allows you to compute $a^x \bmod m$ for any integer $x \geq 0$ in $\mathcal{O}(1)$ time, with $\mathcal{O}(m)$ time preprocessing. *Bonus:* [4★] $\mathcal{O}(1)$ time query and $\mathcal{O}(\sqrt{m})$ time preprocessing. *Bonus:* [4★] Implement it!

N13 [8★] The k-**SAT** problem, or k-**satisfiability**, is the problem of determining whether one can assign truth values to a set of boolean variables $x_1, x_2, \ldots, x_n$ so that a given boolean formula of the form[30]

$$\underbrace{(\mathbf{x} \vee \mathbf{x} \vee \ldots \vee \mathbf{x})}_{k} \wedge \underbrace{(\mathbf{x} \vee \mathbf{x} \vee \ldots \vee \mathbf{x})}_{k} \wedge \ldots \wedge \underbrace{(\mathbf{x} \vee \mathbf{x} \vee \ldots \vee \mathbf{x})}_{k}$$

(where each $\mathbf{x}$ is either x_i or $\neg x_i$ for some i) evaluates to true. The "k" denotes the length of each term, i.e., the number of $\mathbf{x}$'s.

For any k, a straightforward $\mathcal{O}(2^n m)$-time algorithm for k-SAT exists, where m is the length of the formula. [1★] What's that algorithm?

3-SAT (and above) is known to be **NP-complete**, thus there's no known solution for them that runs in polynomial time[31] in the worst case. On the other hand, show that 1-SAT and 2-SAT are solvable in $\mathcal{O}(n+m)$ time by providing an algorithm. Also, show that you can determine one such assignment in $\mathcal{O}(n+m)$ time. *Bonus:* [4★] Implement them!

N14 A **clique** of an undirected graph is a subset of its nodes where each pair is connected by an edge. In other words, a clique is a subset of the nodes that induces a complete subgraph.

- [8★] Consider the problem of finding the maximum clique in a graph. For general graphs, we don't (yet) know of any polynomial-time solution. (The decision version of the problem is actually NP-complete.) On the other hand, for a given graph G with n nodes and e edges, show that one can find the maximum clique in the *block graph of* G in $\mathcal{O}(n+e)$ time. (Be careful: even if $e = \mathcal{O}(n)$, the block graph can have $\Omega(n^2)$ edges!)
- [6★] Let G be an undirected graph with n nodes and e edges. Suppose G is the block graph of some graph. Show how to find the maximum clique in G in $\mathcal{O}(n+e)$ time. (Note: you don't know which graph has G as its block graph.)

[29] This will verify that Floyd's algorithm indeed runs in $\mathcal{O}(l_{\text{tail}} + l_{\text{cycle}})$, and will in fact show the constant hidden in the $\mathcal{O}$ notation. On the other hand, minimizing this constant is important in other applications, e.g., if computing f isn't $\mathcal{O}(1)$ anymore; one example that improves upon Floyd's algorithm's constant is **Brent's algorithm**.

[30] $(a \vee b)$ means "a *or* b", $(a \wedge b)$ means "a *and* b" and $\neg a$ means "not a".

[31] To be more precise, *deterministic polynomial time*.

N15 A **tournament graph** is a directed graph that is obtained by *orienting* the edges of an undirected *complete* graph. In other words, if you ignore the directions of the edges, you get a graph where each pair of nodes is connected by exactly one edge.

You can think of a tournament graph as the results of all rounds in a round-robin tournament with n participants, and an edge $a \to b$ means "a won against b". Such a graph is not necessarily transitive; sometimes a lower-skilled participant beats a higher-skilled participant.

A **Hamiltonian path** is a path that passes through all vertices exactly once. In the context of a tournament, a Hamiltonian path is a possible "ranking" of the participants.[32]

(a) [6★] Prove that any tournament graph has a Hamiltonian path. (*Hint:* Use induction, and notice that removing a single node v yields a smaller tournament graph. Where do you insert v in the Hamiltonian path of the smaller graph?)

(b) [4★] Describe an algorithm that finds a Hamiltonian path in a tournament graph in $\mathcal{O}(n^2)$ time. (*Hint:* use your proof above.) *Bonus:* [5★] Implement it!

(c) [5★] Suppose you have a tournament graph, but you don't (initially) have access to its edges. However, you can *query* for the edges, i.e., you are given a function $f(a, b)$ which returns true if $a \to b$ is an edge and false otherwise.[33] Assume a single call to $f(a, b)$ runs in $\mathcal{O}(1)$ time. Describe an algorithm that finds a Hamiltonian path in the tournament graph that uses only $\mathcal{O}(n \log n)$ calls to f. (An $\mathcal{O}(n^2)$ *running time* is acceptable.) *Bonus:* [4★] Implement it!

(d) [3★] Improve the algorithm above to $\mathcal{O}(n \log n)$ time. *Bonus:* [4★] Implement it!

(e) [5★] A **Hamiltonian cycle** is a cycle that passes through all vertices exactly once. It's easy to see that any graph with a Hamiltonian cycle is strongly connected. Prove that a strongly connected tournament graph always has a Hamiltonian cycle. (*Hint:* Start with a cycle and try growing it.)

(f) [6★] Describe a polynomial-time algorithm that finds a Hamiltonian cycle in a strongly connected tournament graph.

(g) [5★] Given some tournament graph G, prove that the following things are equivalent:

i. G is transitive, i.e., if $a \to b$ and $b \to c$ are edges, then $a \to c$ is an edge as well.

ii. G is acyclic.

iii. The outdegrees of G's nodes are distinct. (In fact, they form the set $\{0, 1, \ldots, n-1\}$)

iv. G has a *unique* Hamiltonian path.

v. G has no cycles of length 3.[34]

(h) [4★] Show that for two nodes x and y in a tournament graph, the following are equivalent:

i. x and y are strongly connected.

ii. There is a cycle containing x and y.[35]

iii. There is a Hamiltonian path where x comes before y, and another Hamiltonian path where y comes before x.

Hint: use your result from **N15**e.

[32] There are other, perhaps better ways to rank participants, such as ranking by the number of wins.

[33] Obviously, if $a \neq b$, exactly one of $f(a, b)$ and $f(b, a)$ is true.

[34] In general graphs, this is a weaker condition than being acyclic, but in tournament graphs, it turns out that they're equivalent, i.e., if there are no cycles of length 3, then there are no cycles at all.

[35] Note: a cycle cannot contain duplicate nodes.

(i) [4★] Does the previous statement hold for general directed graphs? If not, show which implications are true and which implications are false, and show why they are so. (There are 6 implications in total.)

(j) [3★] Given two nodes x and y in a tournament graph such that there is a path from x to y, describe an $\mathcal{O}(n^2)$ algorithm that finds a Hamiltonian path where x comes before y.

(k) [10★] Find a tournament graph that has ≤ 75 nodes and *exactly* $2^{64} - 1$ Hamiltonian paths.[36]

[36] You also need to show that your answer is correct.

4.3 Coding problems

Class-based implementations are strongly recommended; the idea is to make the implementation easily reusable.[37] Making the implementation self-contained in a class makes it very easy for you to reuse, which is handy for your future contests!

First-timers, feel free to solve for extra points. Veterans, solve at least [75★].

C1 [6★] Implement an iterative version of Tarjan's SCC algorithm.[38]

C2 [3★] Implement an algorithm that takes an undirected graph and orients each edge to make the resulting directed graph strongly connected, or determines if it is impossible.

In addition to submitting in the respective online judge, please also attach your submissions to Google classroom.

S1 [5★] **Checkposts:** https://codeforces.com/problemset/problem/427/C

S2 [5★] **Fedor and Essay:** https://codeforces.com/problemset/problem/467/D

S3 [5★] **Dominos:** UVa 11504:

S4 [6★] **Cutting Figure:** https://codeforces.com/problemset/problem/193/A

S5 [6★] **Chef Land:** https://www.codechef.com/problems/CHEFLAND

S6 [5★] **Sub-dictionary:** UVa 1229: ($\mathcal{O}(n+e)$ required)

S7 [6★] **Proving Equivalences:** UVa 12167:

S8 [5★] **Sherlock and Queries on the Graph:** https://www.hackerrank.com/contests/101hack26/challenges/sherlock-and-queries-on-the-graph

S9 [6★] **What Day is it?:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/day-today

S10 [12★] **Beaming With Joy**: https://code.google.com/codejam/contest/5314486/dashboard#s=p2&a=2

S11 [10★] **Mr. Kitayuta's Technology:** https://codeforces.com/problemset/problem/505/D

S12 [10★] **Safe Routes:** https://www.codechef.com/problems/ROUTES

S13 [10★] **Case of Computer Network:** https://codeforces.com/problemset/problem/555/E

S14 [10★] **Tourists:** https://codeforces.com/problemset/problem/487/E

S15 [20★] **Chef and Sad Pairs:** https://www.codechef.com/problems/SADPAIRS

S16 [20★] **Three-Degree-Bounded Maximum Cost Subtree:** https://www.codechef.com/problems/DEG3MAXT

[37] See also: https://en.wikipedia.org/wiki/Modular_programming.

[38] Once you have this, it's easy to write the other `low-disc` algorithms iteratively as well. Save your code; it'll be useful if the online judge has a small stack.

Acknowledgment

Thanks to Jared Asuncion for the images.

A Proof of correctness of SCC algorithms

A.1 Kosaraju's algorithm

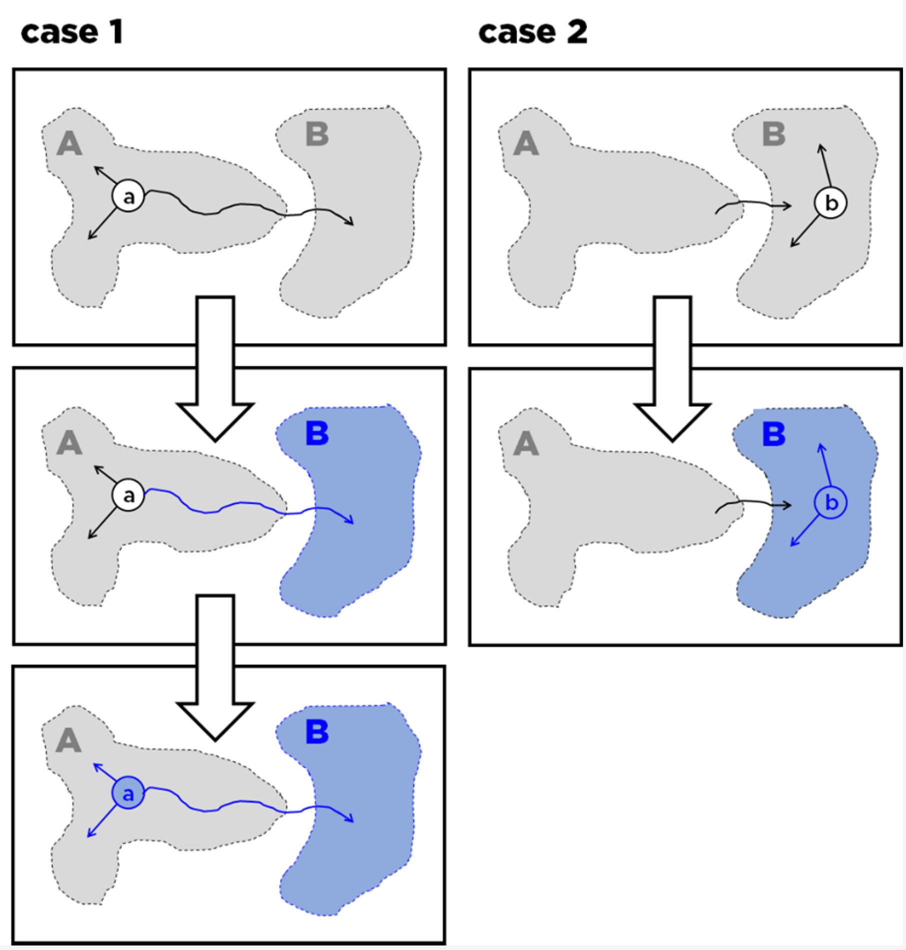
Here, we describe the proof of **Kosaraju's algorithm**.

Theorem A.1. Let A and B be SCCs. If there is a path from A to B, then for any DFS traversal, regardless of the order in which we visit the nodes, there will always be a node in A that has a greater finishing time than all the nodes in B.

Proof. Since A and B are SCCs, it follows that there is no path from any node in B to any node in A (why?). Now, there are two cases (illustrated below):

1. *A node in A, say a, is discovered before any node in B.* In this case, the DFS will be able to traverse the path from A to B and visit all nodes in B before it finishes its visit on node a. Therefore, the finishing time of a is greater than the finishing time of any node in B.

2. *A node in B, say b, is discovered before any node in A.* In this case, the DFS will finish visiting all nodes in B before it ever reaches any node in A, because there is no path from B to A. Therefore, all nodes in A have greater finishing times than all nodes in B.



Thus, regardless of the order we visit the nodes for the DFS, as long as there exists a path from component A to B, there will always be a node in A that has a greater finishing time than all nodes in B. □

The correctness of the algorithm relies on the order in which we visit the nodes in phase two. Let's denote by G^R the graph G with all its edges reversed.[39]

Lemma A.2. In phase two, whenever we start a top-level DFS in G^R, we do it in such an order that all the reachable nodes that belong to different SCCs have already been visited.

Proof. Consider two distinct SCCs, say A and B such that there is a path from B to A in G^R. (Remember that G and G^R have the same SCCs.) We want to show that A will have already been visited by the DFS in phase two. Since there is a path from B to A in G^R, reversing this path, we know there is a path from A to B. Thus, by Theorem A.1, there must be a node a in A that has a greater finishing time than all nodes in B. Thus, by the nature of phase two, this a will be visited by the DFS before any of B's nodes, and since there's no path from A to B in G^R, no nodes from B will be visited at all. Thus, all nodes in A will already have been visited well before we discover any nodes in B in phase two.

□

We are now ready to prove the correctness of Kosaraju's algorithm.

Theorem A.3. Kosaraju's algorithm correctly computes the strongly connected components of a graph G.

Proof. Note that the first time we visit a node in some SCC B in phase two is when we actually start the DFS on a node in B, not on a node in a different SCC that can reach B, otherwise, this would contradict Lemma A.2, and once we start a DFS in B, all nodes in B will be visited by this DFS (because they are reachable from each other), and only those in B will be visited, because all the nodes in the other SCCs reachable from B have already been visited (again according to Lemma A.2). Therefore, whenever we start a new DFS, we visit exactly those nodes that belong to an SCC, and so the algorithm correctly identifies the SCCs. □

A.2 Tarjan's SCC algorithm

Here, we prove that Tarjan's algorithm correctly computes the SCCs. Let's first describe the following property of the DFS forest.

Lemma A.4. The nodes of any SCC form a rooted "subtree" in the DFS forest.[a]

[a] Note that "subtree" means something slightly different here. This doesn't mean that all nodes *down to the leaves* are part of the SCC. It means that, if you consider only the nodes of some SCC and ignore the rest (including possibly some nodes below them), then you get a rooted tree.

A proof is as follows. Here, we define the **head** of an SCC as the node that's discovered first among all nodes in the SCC.

Proof. Consider two nodes a and b from a single SCC such that a is an ancestor of b in the DFS forest. Then every node from the path between them must belong to the same SCC. This is because for every node c in the path, a reaches c, c reaches b and b reaches a (since a and b

[39] G^R is also called the **transpose** of G.

are in an SCC), so c must also belong to the same SCC as a and b.

Next, let h be the head of an SCC. Then every other node in the SCC is a descendant of h in the forest, because they are all connected, and h is visited earliest. Therefore, combining the above with this, it follows that the SCC forms a rooted tree in the forest, with h as the root. □

Note that this also proves that the head is the root of that tree, and that the head is also the last node that the DFS will finish visiting among all nodes in an SCC.

Using the ever-useful `low`$[i]$ and `disc`$[i]$, we can identify whether a node is a *head* or not.

Lemma A.5. A node i is a head of some SCC if and only if `low`$[i]$ = `disc`$[i]$.

Proof. By the definition of `low`, we find that `low`$[i] \le$ `disc`$[i]$.

Thus, the lemma is equivalent to saying that a node i is *not* a head of some SCC if and only if `low`$[i] <$ `disc`$[i]$.

Now, note that `low`$[i] <$ `disc`$[i]$ happens if and only if there is a node j reachable from i that is an ancestor of i in the tree (using only tree edges and back edges). But for such a j, i reaches j and j reaches i, so j is another member of the SCC containing i that has been discovered earlier. Therefore, i is not a head if and only if such a j exists, if and only if `low`$[i] <$ `disc`$[i]$, which is what we wanted to prove. □

So Lemma A.5 shows that `get_scc`(i) will be called if and only if i is a head.

A couple more observations will allow us to prove the correctness of Tarjan's algorithm.

Lemma A.6. If h is a head that's in the stack, then all nodes above h in the stack are descendants of h in the DFS forest.

Proof. At the beginning of the `DFS`(h) call, h will be pushed onto the stack. Now, during the call, `DFS`(j) will only be called (recursively) for descendants j of h, and only these nodes will be pushed onto the stack. Finally, once `DFS`(h) finishes, `get_scc`(h) will be called (from Lemma A.5), and h will be popped from the stack, so no other nodes can be above h in the future.

Thus, at any point that h is in the stack, the only nodes above h will be descendants of h. □

Lemma A.7. When `DFS`(h) finishes for a head h, the DFS will already have finished visiting all descendants of h in the DFS forest, and all descendants will also have been popped from `stack`.

Proof. By definition, h is discovered by the DFS before all its descendants, and thus `DFS`(h) finishes only after `DFS`(j) finishes for all its descendants j. This means that all other descendants of h are either on the stack, or already been popped. They will all be above h since h is not removed from the stack during this time, so at the end of `DFS`(h), `get_scc`(h) will be called and h and all nodes above will be popped. This means that no more descendant of h will be on the stack. □

Note that Lemma A.7 is not quite enough to prove correctness, but it's nearly there. We just need to prove that when `get_scc`(h) is called, all members of the SCC of h, and only those, will be popped along with h.

Lemma A.8. For a head h, when `get_scc`(h) is called, there will be no other heads above h in the stack.

Proof. If h' is a head above h in the stack at the time `get_scc`(h) is called, then h' is a descendant of h (from Lemma A.6), so `DFS`(h') should have already finished (from Lemma A.7), so h' should have already been popped (since h' is a head), a contradiction. □

Lemma A.9. For a head h, when `get_scc`(h) is called, all nodes above h will belong to the same SCC as h.

Proof. Let i be a node above h in the stack. Note that i cannot be a head, from Lemma A.8.

Now, since i is not a head, let h' be the head of the SCC containing i. Since i is still in the stack, from Lemma A.7, h' should still be in the stack.

Also h' can't be above h (Lemma A.8), so either $h = h'$ (which proves our lemma), or h' is below h. In the latter case, h' reaches h (Lemma A.6), h reaches i (Lemma A.6), and i reaches h' (since they belong to the same SCC), so h and i must belong to the same SCC, which proves our lemma (and in fact also shows that $h = h'$). □

Note that this is enough to show that the algorithm is correct!

Theorem A.10. Tarjan's algorithm correctly computes the strongly connected components.

Proof. Let i be a node, and let h be the head such that the call `get_scc`(h) pops i. Then i is a node above h in the stack, so by Lemma A.9, h and i belong to the same SCC. Thus, every node i is popped at the correct time, i.e., on the head of its SCC. □