NOI.PH 2017 Training Week 8

Kevin Charles Atienza

April 2017

Contents

1 Connectivity in Undirected Graphs **2**
1.1 Depth-first search revisited (undirected version) . . . 3
1.2 Bridges and articulation points . . . 7
1.2.1 Finding bridges and articulation points . . . 7

2 Connectivity in Directed Graphs **11**
2.1 Cycles and DAGs . . . 11
2.2 Depth-first search revisited (directed version) . . . 13
2.3 Cycle finding . . . 16
2.4 Floyd's cycle finding algorithm . . . 16
2.4.1 Cycle-finding and factorization: Pollard's ρ algorithm . . . 20
2.5 Computing strongly connected components . . . 21
2.5.1 Kosaraju's algorithm . . . 21
2.5.2 Tarjan's SCC algorithm . . . 25
2.5.3 Which algorithm to use? . . . 28
2.6 DAG of SCCs . . . 28

3 Biconnectivity in Undirected Graphs **30**
3.1 Menger's theorem . . . 31
3.2 Robbins' theorem . . . 31
3.3 2-edge-connected components . . . 32
3.4 Biconnected components . . . 34
3.4.1 Block graph . . . 37
3.4.2 Block-cut tree . . . 38

4 Problems **39**
4.1 Warmup coding problems . . . 39
4.2 Coding problems . . . 39
4.3 Non-coding problems . . . 40
4.4 Bonus problems . . . 43

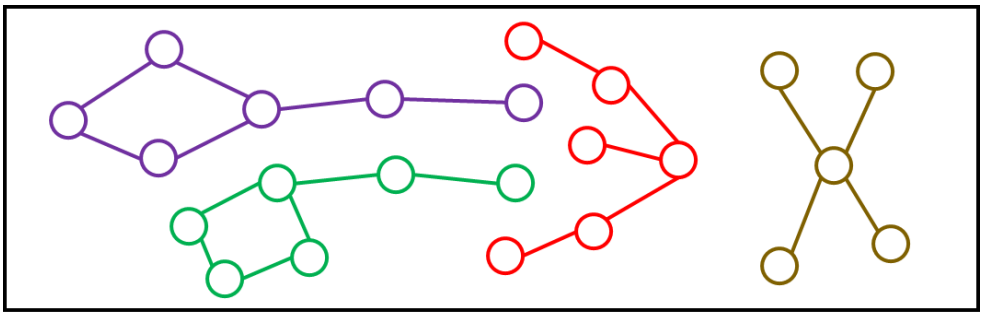
This week, we'll connect ourselves with various topics related to connectivity in graphs.

1 Connectivity in Undirected Graphs

It turns out that connectivity is easier to understand with undirected graphs. That's because the edges are *bidirectional*, meaning that if there's a path from a to b, then there's also a path from b to a.

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 reflective, 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.

Finding connected components can be done with BFS or DFS, which you probably already knew. Both algorithms work fine.[2]

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**.

The natural question is how to detect if there's a cycle in an undirected graph. For this, you

[1] "Maximal" means you can't add any more nodes without violating the requirement.
[2] though DFS usually gets deeper in recursion than BFS.

can use BFS or DFS again to detect if a cycle exists (how?) and find such a cycle (how?) in $O(n+e)$ time. A fancier (and perhaps easier-to-implement) approach is **union-find**, although it runs in $O(n+e\cdot\alpha(n))$, 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 have to judge when to use union-find or not.)[3]

1.1 Depth-first search revisited (undirected version)

Now 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]

```
1  function DFS(i):
2     // perform a DFS starting at node i
3
4     visited[i] = true // mark it as visited
5     for j in adj[i]:
6        if not visited[j]:
7           DFS(j)
8
9  function DFS_all():
10    // perform a DFS on the whole graph
11    for i = 0..n-1:
12       visited[i] = false
13
14    for i = 0..n-1:
15       if not visited[i]:
16          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. The first (useful) thing we can do is generalize:

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

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

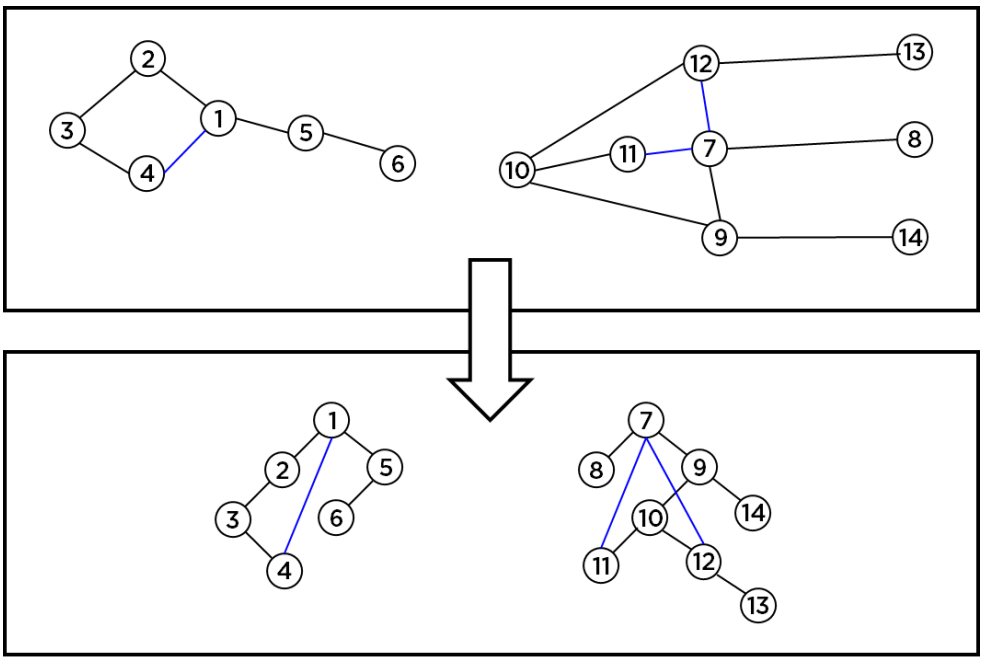
```
function DFS(i, p):
   visited[i] = true
   parent[i] = p

   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

function DFS_all():
   for i = 0..n-1:
      visited[i] = false
      parent[i] = -1

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

Here, the function "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 we're computing the "parent" array, denoting the parent 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 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**. Thus, we can clearly see that DFS classifies the edges into one of two types, depending on how they were handled by the DFS traversal! We'll see later on that 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.

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:

```
1  function DFS(i, p):
2     start_time[i] = time++
3     parent[i] = p
4
5     for j in adj[i]:
6        if start_time[j] == -1:
7           mark (i, j) as a tree edge
8           DFS(j, i)
9        else if j != p:
10          mark (i, j) as a back edge
11
12    finish_time[i] = time++
13
14 function DFS_all():
15    time = 0
16    for i = 0..n-1:
17       start_time[i] = -1
18       finish_time[i] = -1
19       parent[i] = -1
20
21    for i = 0..n-1:
22       if start_time[i] == -1:
23          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! (Why?) [5]

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. For now, there aren't many uses to them, but they will be more useful for us later on.

The running time is still $O(n + e)$, 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 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]

[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 :)

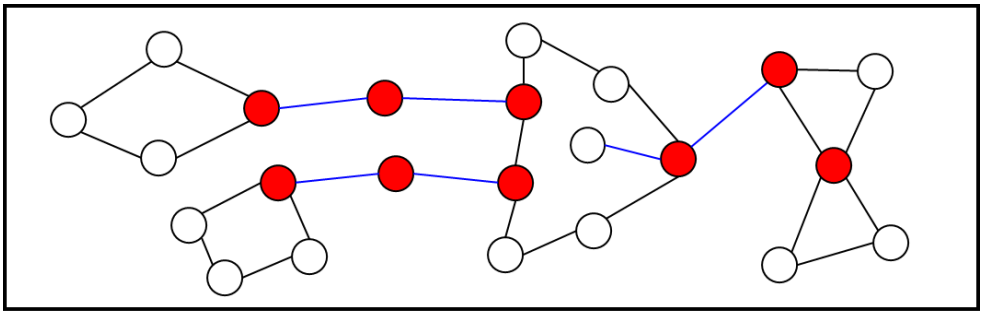
[6] A generic way to do that conversion is to simulate the call stack with an actual stack, and the stack entries describe the whole state of the function at that point. In this case, you only need to push "`i`" and the index of "`j`" in "`adj[i]`".

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.

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

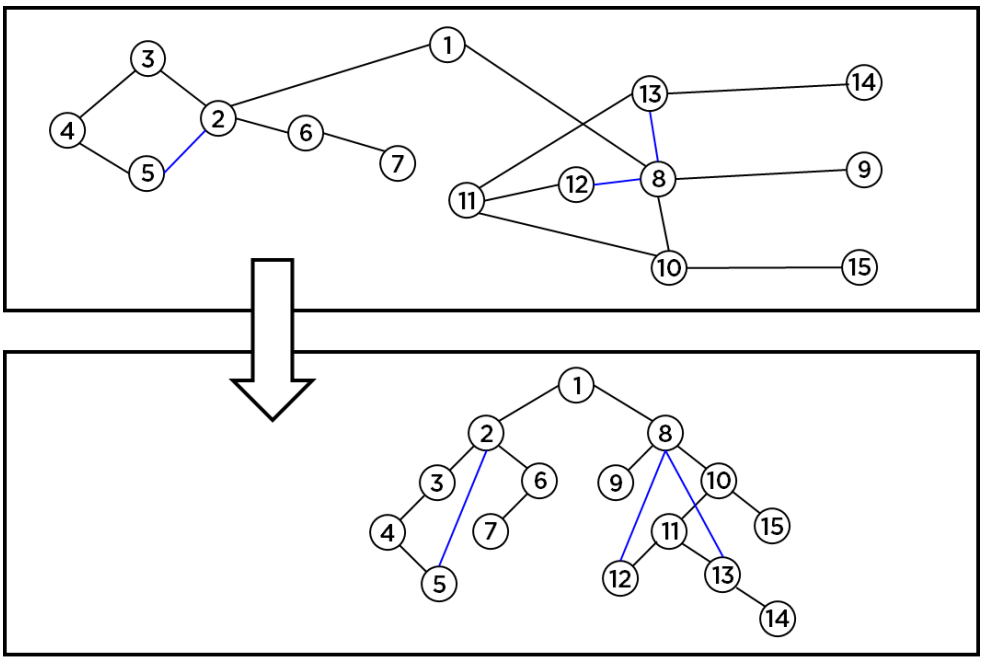
It's easy to see why one would identify and study these edges/nodes. Roughly speaking, these edges and nodes are the "weakest links" of your network; if you're modelling a computer network, then a bridge could represent a critical connection, and an articulation point could represent a critical computer.

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

Given a (connected) undirected graph, how do you find all its bridges and articulation points? 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 $O(n+e)$ time to traverse a graph, it takes $O(e(n+e))$ and $O(n(n+e))$ time to find the bridges and articulation points this way.

But it turns out that we can compute both in $O(n+e)$ 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):



DFS tree

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, we can state the conditions precisely. 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.

- Back edges can never be bridges.
- Let (a, b) be a tree edge, where a is the parent. Then (a, b) is a bridge if and only if there's no back edge from any descendant of b to any ancestor of 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 there's no back edge from any descendant of b to any *proper* ancestor[7] of a.
  - 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.

With these observations, we can now compute the bridges and articulation points by augmenting DFS with additional data:

- Let $\texttt{disc}[i]$ be the discovery time of i. (Identical to $\texttt{start\_time}[i]$ above.)

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

- Let $\texttt{low}[i]$ 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 $\texttt{low}[i] = \texttt{disc}[i]$.

Using $\texttt{disc}[i]$ and $\texttt{low}[i]$, 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 $\texttt{low}[b] > \texttt{disc}[a]$.
- 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 $\texttt{low}[b] \geq \texttt{disc}[a]$.
  - a is the root of a DFS tree and a has at least two children.

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

```
function 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

function bridges_and_articulation_points():
   time = 0
   for i = 0..n-1:
      disc[i] = -1

   for i = 0..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 $\texttt{low}[i]$ is being computed. Please ensure that you understand it.

2 Connectivity in Directed Graphs

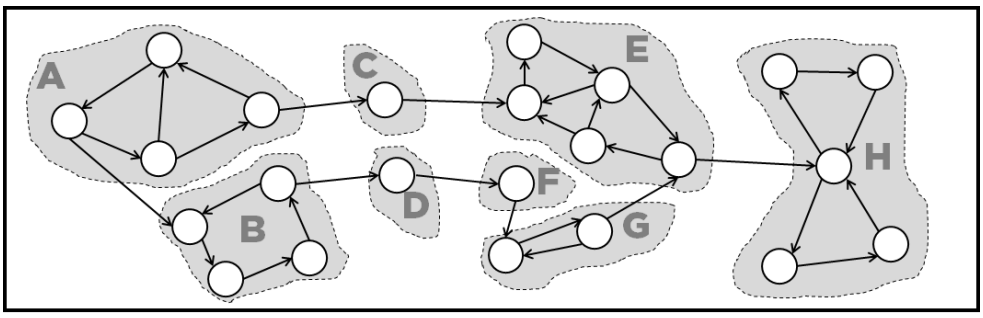
Since connectivity with undirected graphs is so boring, let's consider directed graphs instead. The important difference is that paths are not *reversible* anymore, so the nice picture of "connected components" above does not apply any more.

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 vice-versa",[8] 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 undirected graphs.

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



strongly connected components (SCC)

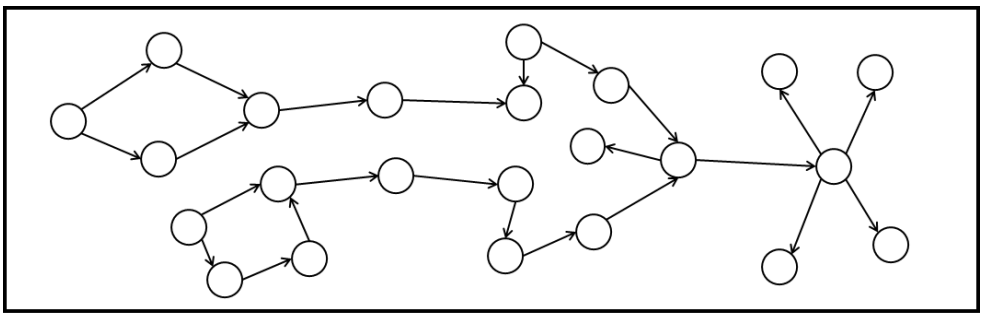
Note that this time, there could be edges from one SCC to another. This is more exciting than before!

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:

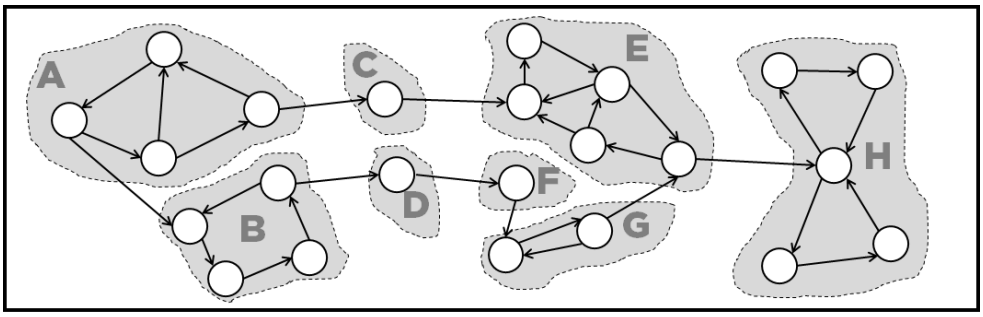
[8] obviously these are two different paths.



directed acyclic graph (DAG)

One nice thing about DAGs is that you can *serialize* the nodes, i.e., find a total order of the nodes such that 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 learned how to compute the topological sort in linear time.

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



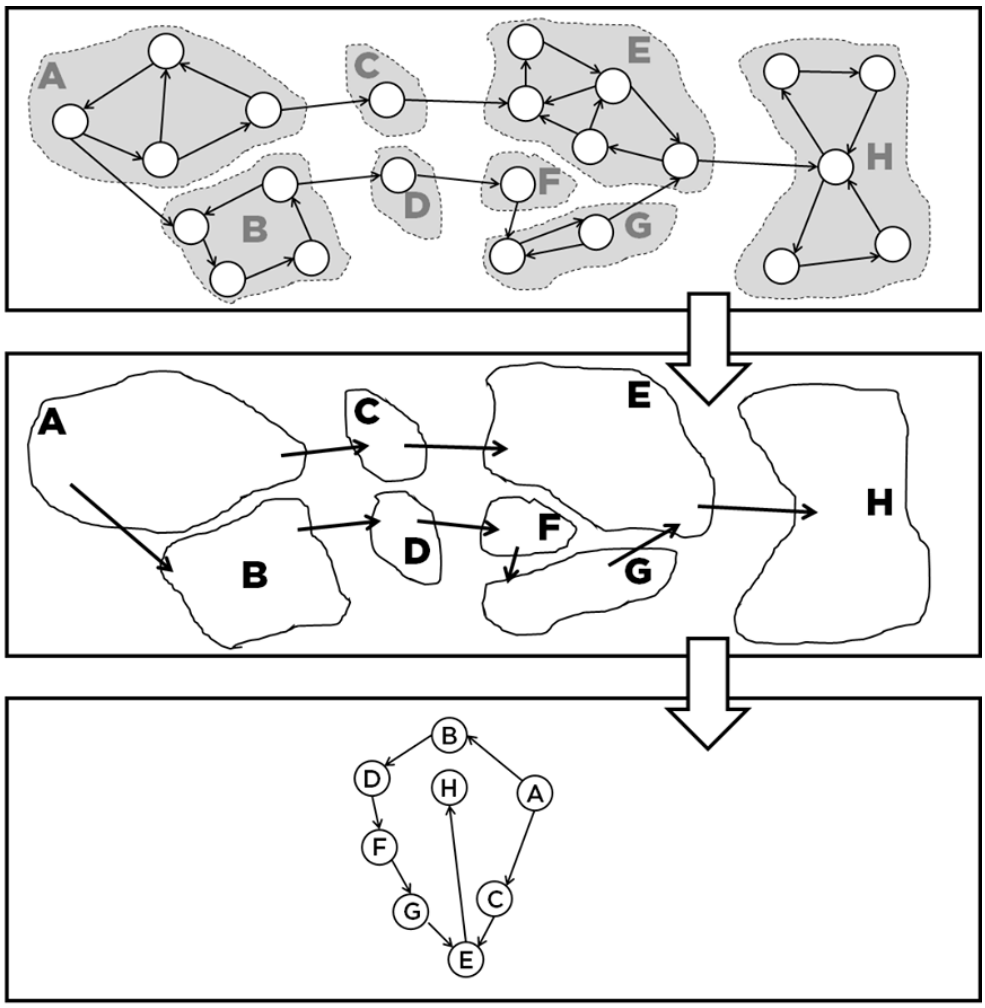
strongly connected components (SCC)

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 $\mathrm{SCC}_a \to \mathrm{SCC}_b$, 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'll 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 $\sim$**".



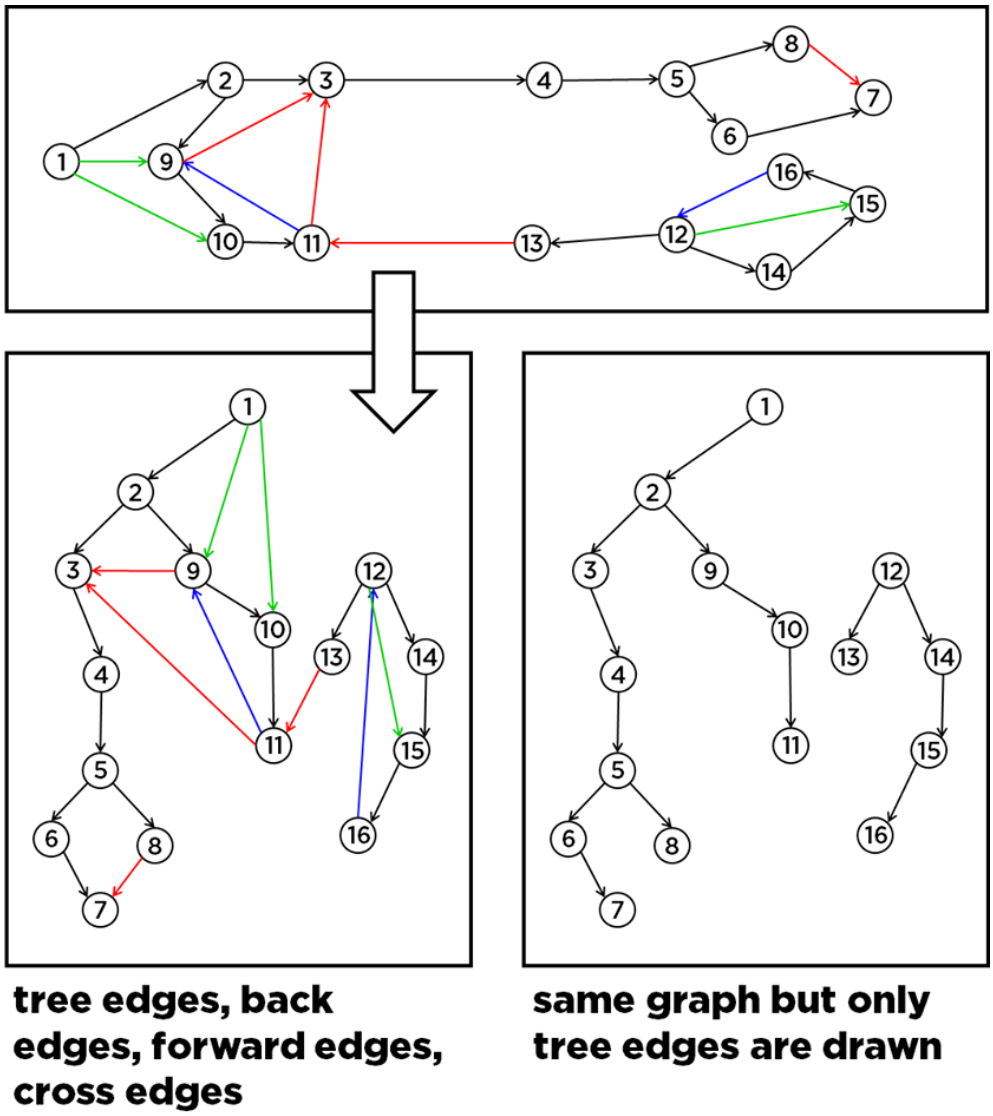
DAG of SCCs

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:



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.

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.

It's sometimes convenient to consider a tree edge as a type of forward edge as well, although there are forward edges that are not tree edges (unlike in the undirected case).

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

```
1  function DFS(i):
2     // perform a DFS starting at node i
3
4     start_time[i] = time++
5
6     for j in adj[i]:
7        if start_time[j] == -1:
8           mark (i, j) as a tree edge
9           DFS(j)
10       else if finish_time[j] == -1:
11          mark (i, j) as a back edge
12       else if finish_time[j] > start_time[i]:
13          mark (i, j) as a forward edge
14       else:
15          mark (i, j) as a cross edge
16
17    finish_time[i] = time++
18
19 function DFS_all():
20    time = 0
21    for i = 0..n-1:
22       start_time[i] = -1
23       finish_time[i] = -1
24
25    for i = 0..n-1:
26       if start_time[i] == -1:
27          DFS(i)
```

Notice how we made use of 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.
- If `finish_time[j] != -1` and `finish_time[j] > start_time[i]`, then j's visitation is already finished, but only after i's visitation began, so it means j must be a descendant of i.
- 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.

The running time is still $O(n + e)$, but along the way, we've again 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.[10]

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: *there is a cycle if and only if there is a back edge in the DFS forest*. (Why?) 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. (How?)[11]

One can also detect (and find) a cycle using BFS, although we will leave it to you to discover.

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.[12] Since there is exactly one outgoing edge from each node, f is well-defined. Conversely, 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).

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

[11] 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.

[12] 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).

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:[13]

$$(x, f(x), f(f(x)), \ldots, f^{(n)}(x), \ldots)$$

Now, if we assume that our node set is finite, then (by the pigeonhole principle) this will eventually repeat, i.e., there will be indices $0 \leq i < j$ such that $f^{(i)}(x) = f^{(j)}(x)$.[14] In fact, once this happens, the subsequence $(f^{(i)}(x), f^{(i+1)}(x), \ldots, f^{(j-1)}(x))$ will repeat forever. This always happens regardless of which node you start at.

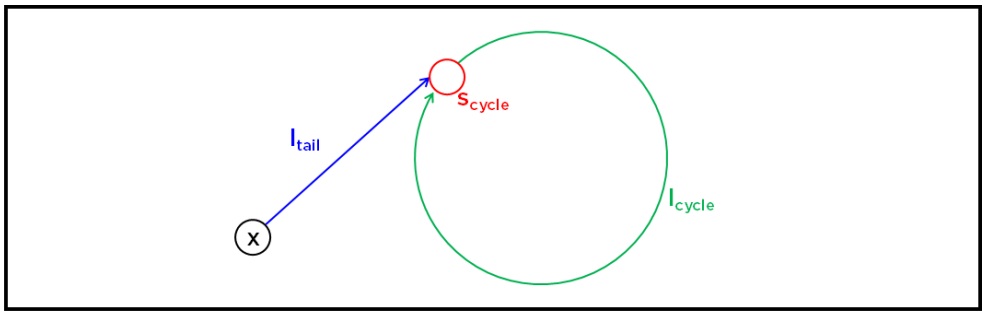
This gives rise to a natural question: Given a starting point x, when is the first time that a repeat happens? Furthermore, how long is the cycle? To make the question more interesting, suppose we don't know anything about f apart from:

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 $O(1)$.)

We can formalize the **cycle-finding problem** as: Given a function f with the above properties, and a starting point x, compute the following:

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}.

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.

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

```
// Just−walk algorithm
function 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 $O(l_{\text{tail}} + l_{\text{cycle}})$ time.

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

```
// Bogo−cycle−finding algorithm
function 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 $O(1)$ memory! Well, that's certainly true, but this is an incredibly slow solution! The idea is to use only $O(1)$ memory without sacrificing running time.

Let's discuss 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. Once we they meet, they stop, and 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}!

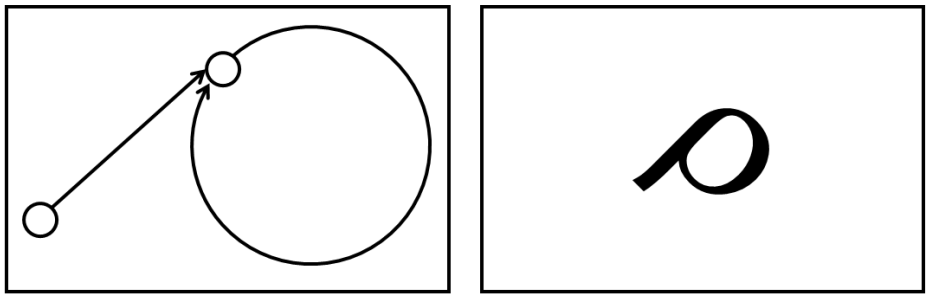
Once we get s_{cycle}, l_{tail} and l_{cycle} can easily be computed, e.g., l_{cycle} can be computed by going around the cycle once. Here's the pseudocode:

```
1  // Floyd's cycle-finding algorithm
2  function cycle_find(x):
3     tortoise = hare = x
4     do:
5        tortoise = f(tortoise)
6        hare = f(f(hare))
7     while tortoise != hare
8
9     // teleport, and walk at the same speed
10    hare = x
11    l_tail = 0
12    while tortoise != hare:
13       tortoise = f(tortoise)
14       hare = f(hare)
15       l_tail++
16
17    s_cycle = hare
18
19    // compute l_cycle by walking around once.
20    l_cycle = 0
21    do:
22       hare = f(hare)
23       l_cycle++
24    while tortoise != hare
25
26    return (s_cycle, l_cycle, l_tail)
```

This clearly uses $O(1)$ memory. We've also mentioned that this runs in $O(l_{\text{tail}}+l_{\text{cycle}})$, 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 as an exercise to prove the running time and correctness of this algorithm!

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

An interesting application of Floyd's algorithm (or at least its idea) is with integer factorization. **Pollard's ρ algorithm**[15] 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.

```
1  function try_factorize(N):
2     x = y = 2 // starting point
3     do:
4        x = f(x)
5        y = f(f(y))
6        d = gcd(|x - y|, N)
7     while d == 1
8
9     if d == N:
10       failure
11    else:
12       return d
```

[15] ρ is pronounced "rho".

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.)

A more thorough explanation of why this works, and why cycle-finding appears, can be seen on the Wikipedia page: https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm.

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 search 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.

Note that these sections go into quite some detail in proving the correctness of the algorithms, so unless you're comfortable, you might want to skip the proofs on first reading and just learn how the algorithms work first.

2.5.1 Kosaraju's algorithm

Here, we describe **Kosaraju's algorithm** which computes the strongly connected components of a directed graph.

Here are the steps of Kosaraju's algorithm:

1. Mark all vertices as not visited.
2. 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. Perform another DFS traversal, this time in decreasing order of finishing time (which were calculated earlier). 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 $O(n + e)$ time. You might wonder why not $O(n \log n + e)$ 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 the next phase, 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:

```
function 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]:
         DFS(j, adj, group)

   group.push(i)

function SCC_Kosaraju():
   // mark all nodes as unvisited
   for i = 0..n-1:
      visited[i] = false

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

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

   // reinitialize visited
   for i = 0..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 here, but you may want to skip it if you want

spend some time thinking about it yourself.

The following is a rough proof of why this algorithm is correct. It relies on the fact that the SCCs of a graph is the same as the SCCs of its reversal graph (this can be proved very easily). Let us denote by G^R the graph G with all its edges reversed.[16]

Claim 2.1. *Kosaraju's algorithm correctly computes the strongly connected components of a graph G.*

Proof. In order to prove that the algorithm is correct, we only need to ensure that 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 a different SCC have already been visited*. This is sufficient because if this is true, then the first time we visit a node in some SCC 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 the above statement), 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 the statement). Therefore, whenever we start a new DFS, we visit exactly those nodes that belong to an SCC.

Now, consider two distinct SCCs of G, say A and B, and suppose there is a path from some node in A to some node in B. Since A and B are SCCs, it follows that there is no path from any node in B to any node in A. Now, during the first phase, where we are performing the DFS in an arbitrary order, there are two cases:

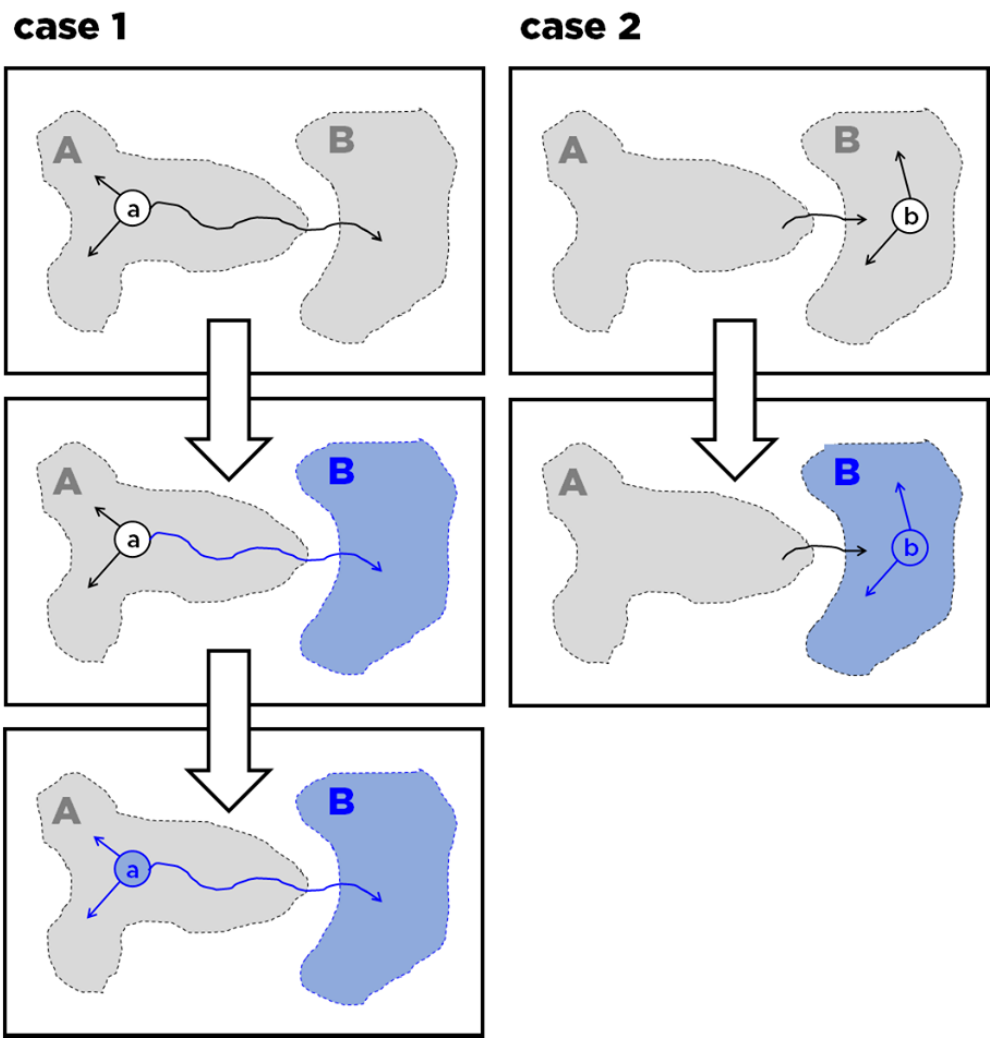
1. *A node in A, say a, is discovered before any node in B.* In this case, the DFS will be able to traverse the path from A to B and visit all nodes in B before a itself finishes expanding.[17] Therefore, the finishing time of a is greater than the finishing time of any node in B.

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 a greater finishing time than all nodes in B.

The two cases are illustrated below:

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

[17] By "finishing expanding" I mean finishing the DFS run on the node and returning from the call. In the DFS pseudocode, it's precisely the time when we set a node's `finish_time`.



What this shows is that 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. More generally, if $A_1, A_2, \ldots, A_k$ are all the SCCs that can reach B, then there exists a node in each one of those components that have a greater finishing time than all nodes in B.

In the reversal graph G^R, $A_1 \ldots A_k$ are precisely the SCCs that B can reach, and therefore none of the A_i can reach B. And since in the second phase we are performing the DFS in decreasing order of finishing times, it follows that we will have done a DFS on each A_i before we visit any on B, and thus, all nodes in $A_1 \ldots A_k$ will have been visited before any node in B. Therefore, once we start visiting a node in B, all the nodes reachable from it that belong to a different SCC have already been visited. This is exactly what we wanted to prove. □

As a side note, the main idea in this proof can be repurposed to prove that we can get a topological sort of the DAG by ordering the nodes by decreasing finishing time:

Claim 2.2. *A topological sort of a DAG is obtained by performing a DFS on it and ordering the nodes in decreasing finishing time.*

Proof. Note that the main idea in the previous proof is to show 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 has exactly one element, say a, and B has exactly one element, say b, so it simply says 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

Here, we describe another algorithm called *Tarjan's SCC algorithm*. Tarjan's SCC 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.

This algorithm uses the `disc` and `low` arrays, just like in our algorithm for bridges and articulation points!

Here's the pseudocode:

```
function 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)

function 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)

function SCC_Tarjan():
   stack = new empty vector
   sccs = new empty vector of vectors
   time = 0
   for i = 0..n-1:
      disc[i] = -1
      instack[i] = false

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

Let's try to explain how this works. Let's first describe the following property of the DFS forest.

Claim 2.3. *The nodes of any SCC form a rooted subtree in the DFS forest.*[18]

A simple proof sketch is as follows. Here, we define the *head* of an SCC as the node that's

[18]Note that "subtree" means 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.

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 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 to finish expanding among all nodes in an SCC.

Let's now describe $\texttt{disc}[i]$ and $\texttt{low}[i]$. They are defined similarly as before:

- Let $\texttt{disc}[i]$ be the discovery time of i.
- Let $\texttt{low}[i]$ 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 $\texttt{low}[i] = \texttt{disc}[i]$.

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

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

Claim 2.4. *A node i is a head of some SCC if and only if $\texttt{low}[i] = \texttt{disc}[i]$.*

Here's a rough proof:

Proof. By the definition of $\texttt{low}$, we find that $\texttt{low}[i] \leq \texttt{disc}[i]$.

Thus, the claim is equivalent to saying that a node i is *not* a head of some SCC if and only if $\texttt{low}[i] < \texttt{disc}[i]$.

Now, note that $\texttt{low}[i] < \texttt{disc}[i]$ 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 $\texttt{low}[i] < \texttt{disc}[i]$, which is what we wanted to prove. □

Now, we're calculating $\texttt{low}[i]$ and $\texttt{disc}[i]$ on the fly as we perform the DFS traversal, and since the head is the last to finish expanding, we can collect all the members of the SCC containing that head right at that moment. Conveniently enough, it turns out that the members of this SCC are all on top of the stack!

To see this, note that whenever we visit a node, we simply push it onto the stack. But when a node finishes expanding, we don't necessarily pop it from the stack. We only pop the stack whenever we finish expanding a head h, and we keep popping until h is popped.

Now, due to Claim 2.3 above, and the fact that we only pop when we finish expanding a head, we are guaranteed that for every two nodes i and j in the stack belonging to a single SCC, all nodes between them in the stack also belong to the same SCC, therefore all nodes in a single SCC in the stack are found in contiguous locations. Also, when we finish expanding a head h, all other nodes in the SCC of h are still in the stack. Therefore, whenever we pop from the stack, all the nodes popped belong to a single SCC!

After the traversal, we would have computed all the SCCs of the graph.

Clearly, the time complexity is $O(n + e)$. 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.[19]
- It's usually faster in practice.

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.

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.

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

3. Remove the self-loops and duplicate edges.

4. The resulting graph is the DAG of SCCs.

The "shrinking" part might be too vague, but a simple "engineer" approach can be used. Let's restate the steps above, but also expand that part:

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 exist already).

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 $O(n + e)$ time.

3 Biconnectivity in Undirected Graphs

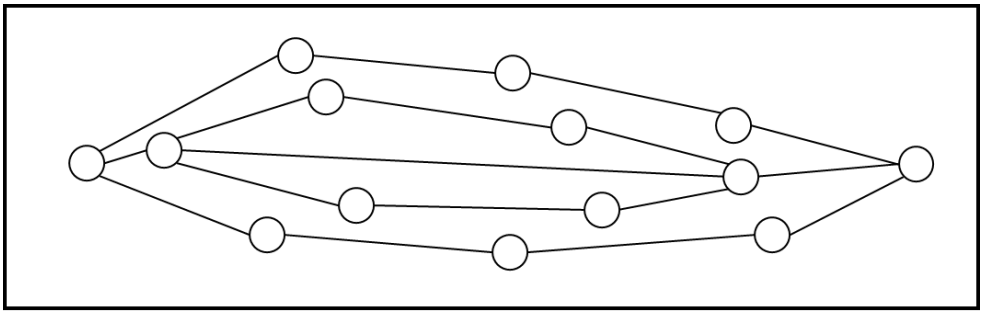
Let's return to undirected graphs. We mentioned that the directed case is more interesting, but that's not entirely true! Here we'll describe another aspect of connectivity in undirected graphs.

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.

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. (Why?)

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 that connected graphs of ≤ 2 nodes are considered biconnected but not 2-connected. 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:

1. Let x and y two *distinct* vertices. Then 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. Then 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, but at least you know them; they could be useful.

There are other similar results like that all throughout 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).[20]

3.2 Robbins' theorem

Another interesting fact about 2-edge-connected graphs is that *you can orient*[21] *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.

If a graph has a bridge, then obviously a strong orientation is impossible. 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 since this is an undirected graph.) Since there are no bridges, it 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

[20] Unsurprisingly, these results are all related.

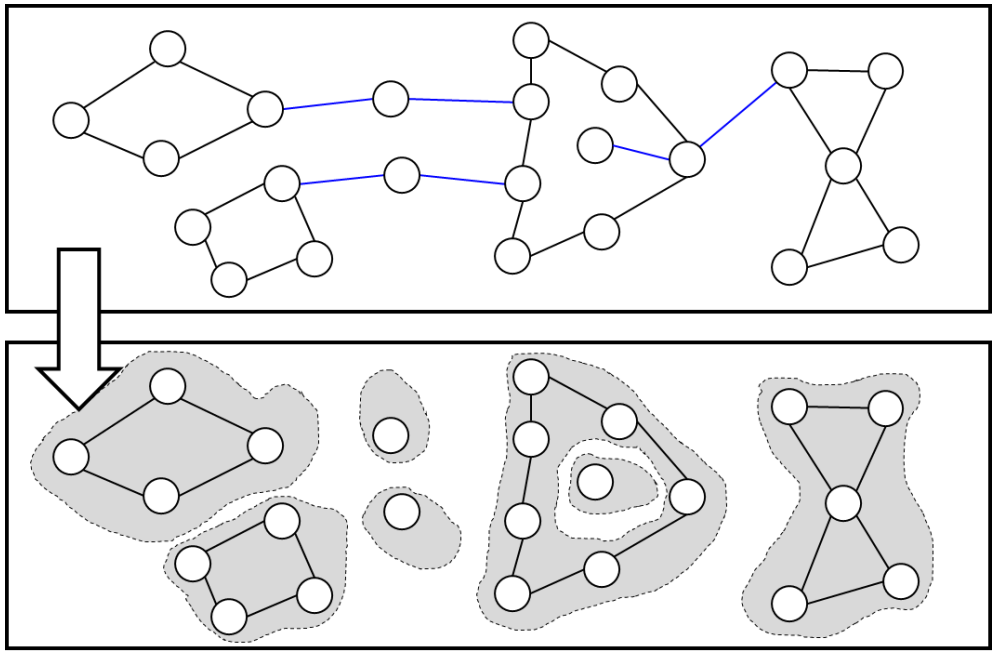
[21] To "orient" an edge means to choose its direction, hence the edge becomes directed.

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.

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.

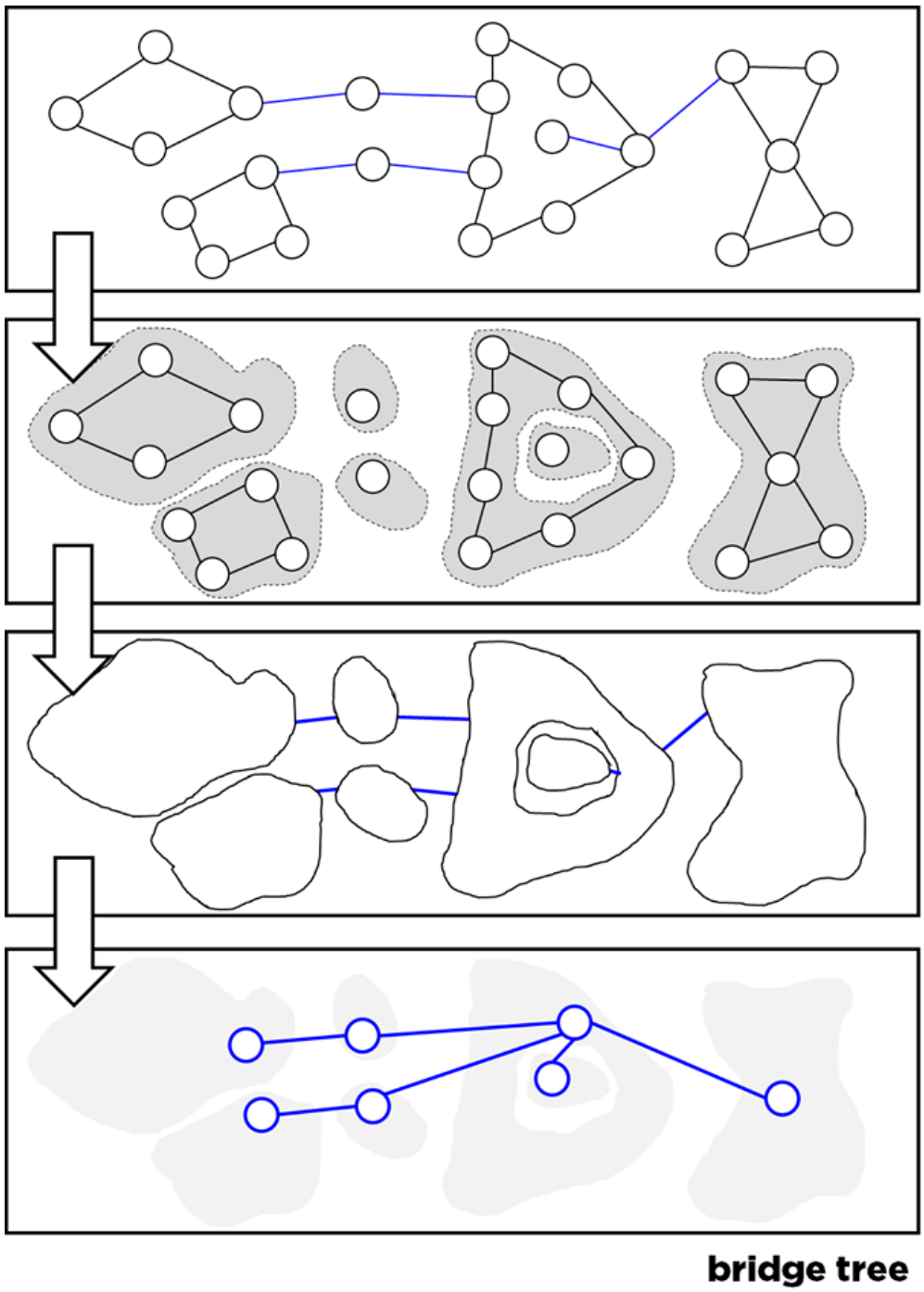
Well, by definition, a 2-edge-connected component cannot have bridges, so let's say we remove the bridges in the original graph first. (We can find those with DFS.) Then what we're left with are subgraphs that don't contain any bridges, hence are 2-edge-connected![22]



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*!

[22] 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.

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



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 "shrinking" 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.

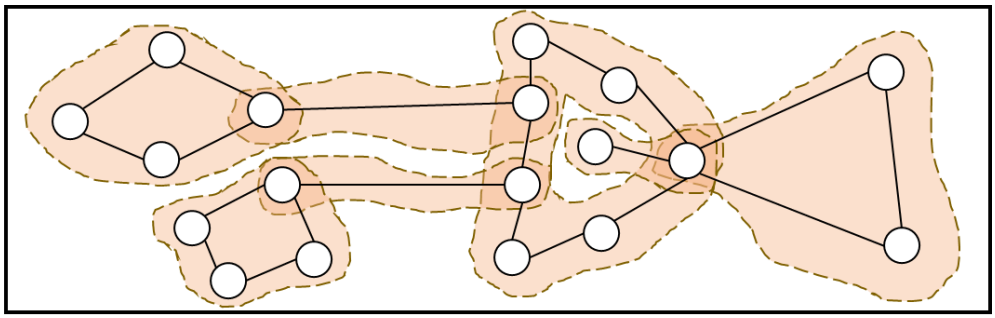
7. The resulting graph is the bridge tree.

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

3.4 Biconnected components

A **biconnected component**, or **BCC**, is a maximal subgraph that is biconnected. The 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!

```
function 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

function 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)

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

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

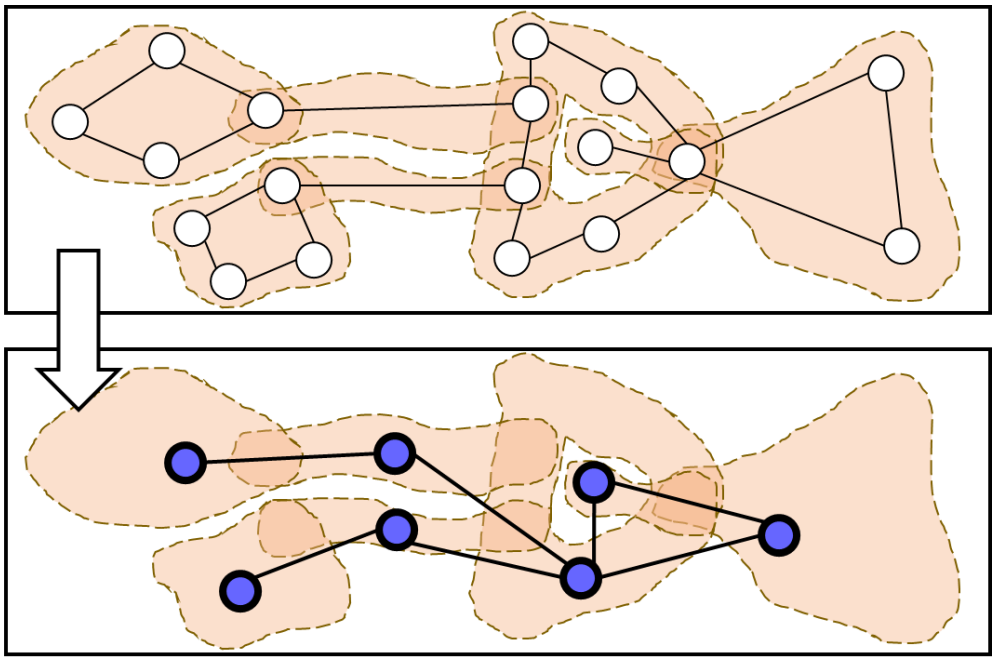
After this $O(n+e)$ process, we now have the articulation points and the BCCs of the graph!

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.

The natural graph structure we can form, then, is to compress each BCC into a single node, and declare 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.[23]

Here's an example of a block graph:



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 $O(n)$ edges can have up to $\Omega(n^2)$ edges![24] So that's really sad.

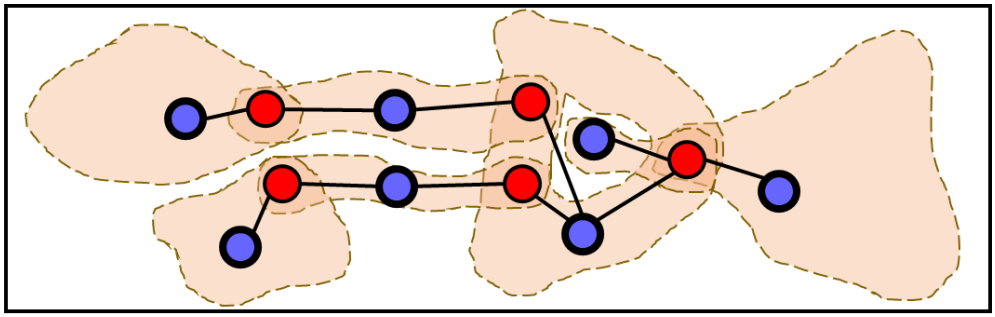
[23] Use the engineer approach.

[24] $f(n) = \Omega(g(n))$ is the same as $g(n) = O(f(n))$. Informally, you can think of O as "asymptotically at most" and Ω as "asymptotically at least".

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.[25] 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.)

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!

Constructing a block-cut tree from definition is straightforward.[26]

[25] Note that this "compressed" graph can actually be larger than the original graph!
[26] Use the engineer approach.

4 Problems

I realize the problems here are tougher than usual; the general rule here is to solve as many as you can!

4.1 Warmup coding problems

Most of these are straightforward applications of some of the algorithms discussed. Not required, but you may want to use these problems to test your implementations of the algorithms above.

1. **Test**: UVa 10731
2. **Tourist Guide**: UVa 10199 (implement a linear-time solution)
3. **Network**: UVa 315 (implement a linear-time solution)
4. **Lonely People in the World**: https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/lonely-people-in-the-world
5. **Putogethers**: https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/putogethers

4.2 Coding problems

Here are the required problems:

1. **Checkposts**: http://codeforces.com/problemset/problem/427/C
2. **Dominos**: UVa 11504
3. **Cutting Figure**: http://codeforces.com/problemset/problem/193/A ($O(nm)$ required)
4. **Chef Land**: https://www.codechef.com/problems/CHEFLAND
5. **Sub-dictionary**: UVa 1229 ($O(n+e)$ required)
6. **Proving Equivalences**: UVa 12167
7. **Sherlock and Queries on the Graph**: https://www.hackerrank.com/contests/101hack26/challenges/sherlock-and-queries-on-the-graph
8. **Mr. Kitayuta's Technology**: http://codeforces.com/problemset/problem/505/D

9. **Case of Computer Network**: http://codeforces.com/contest/555/problem/E
10. **Tourists**: http://codeforces.com/contest/487/problem/E
11. **Chef and Sad Pairs**: https://www.codechef.com/problems/SADPAIRS

Please feel free to ask if some of the problems are unclear.

4.3 Non-coding problems

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

If any definition is unclear, please ask.

1. In an undirected graph, why is a back edge never a bridge?
2. Prove or disprove: an undirected graph is acyclic if and only if every edge is a bridge.
3. Prove or disprove: an undirected graph is acyclic if and only if every node of degree > 1 is an articulation point.
4. Prove or disprove: each endpoint of a bridge is either of degree 1 or an articulation point.
5. Prove or disprove: every articulation point is an endpoint of some bridge.
6. 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. Prove or disprove: every arborescence is a polytree.
7. Prove or disprove: every polytree is an arborescence.
8. 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.
9. Prove or disprove: every multitree is a polytree.
10. Prove that the bogo-cycle-finding algorithm correctly computes s_{cycle}, l_{cycle} and l_{tail}. Determine its running time.
11. Prove that Floyd's cycle-finding algorithm correctly computes s_{cycle}, l_{cycle} and l_{tail}.
12. Prove that Floyd's cycle-finding algorithm takes $O(l_{\text{tail}} + l_{\text{cycle}})$ time.
13. Prove or disprove: Every 2-edge-connected graph is biconnected.

14. Prove or disprove: Every biconnected graph is 2-edge-connected.

15. Show that the block graph formed from a graph with n nodes and $O(n)$ edges can have $\Omega(n^2)$ edges.

16. 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[27]

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

    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 $O(2^n m)$-time algorithm for k-SAT exists, where m is the length of the formula. (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[28] in the worst case. On the other hand, show that 1-SAT and 2-SAT are solvable in $O(n + m)$ time by providing an algorithm. Also, show that you can determine one such assignment in $O(n + m)$ time. Bonus: Implement them!

17. 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 in a round over 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.[29]

    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?)

18. Describe an algorithm that finds a Hamiltonian path in a tournament graph in $O(n^2)$ time. (Hint: use your proof above.) Bonus: Implement it!

19. 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 provided with a function $f(a, b)$ which

[27] $(a \vee b)$ means "a *or* b", $(a \wedge b)$ means "a *and* b" and $\neg a$ means "not a".
[28] To be more precise, *deterministic polynomial time*.
[29] There are other (perhaps better) ways to rank participants, such as ranking by the number of wins.

returns true if $a \to b$ is an edge and false otherwise.[30] Assume a single call to $f(a, b)$ runs in $O(1)$ time. Describe an algorithm that finds a Hamiltonian path in the tournament graph that uses only $O(n \log n)$ calls to f. (An $O(n^2)$ *running time* is acceptable.) Bonus: Implement it!

20. Improve the algorithm above to $O(n \log n)$ time. Bonus: Implement it!

21. 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.)

22. Describe a polynomial-time algorithm that finds a Hamiltonian cycle in a strongly connected tournament graph.

23. Given some tournament graph G, prove that the following things are equivalent:

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

    (b) G is acyclic.

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

    (d) G has a *unique* Hamiltonian path.

    (e) G has no cycles of length 3.[31]

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

[31] 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.

4.4 Bonus problems

Here are bonus problems. Solve them for extra credit.

1. In the DFS pseudocode above for undirected graphs, why is it that if we don't check the condition `j != p`, all edges will be marked back edges?

2. Show that there are no forward and cross edges in a DFS forest of any undirected graph.

3. Show that 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.)

4. 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}?

5. 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}?[32]

6. In the pseudocode for Tarjan's SCC algorithm, is the line `low[i] = min(low[i], disc[j])` 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?

7. Implement an iterative version of Tarjan's SCC algorithm.[33]

8. Show that removing a bridge doesn't turn any non-bridge edge into a bridge.

9. Show that the block-cut tree is indeed a tree.

10. Show that the block-cut tree can have both more nodes and edges than the original graph.

11. 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.[34]

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

[33] 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.

[34] Hint for those who aren't familiar with proving least upper bounds: Suppose you want to show that 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.

12. Show that for two nodes x and y in a tournament graph, the following are equivalent:

    (a) x and y are strongly connected.

    (b) There is a cycle containing x and y.[35]

    (c) There is a Hamiltonian path where x comes before y, and another Hamiltonian path where y comes before x.

    Hint: use your result from 21.

13. 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.)

14. Given two nodes x and y in a tournament graph such that there is a path from x to y, describe an $O(n^2)$ algorithm that finds a Hamiltonian path where x comes before y.

15. 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.

    Consider the problem of finding the maximum clique in a graph. For general graphs, we don't (yet) know of any polynomial-time solution. 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 $O(n+e)$ time. (Be careful: even if $e = O(n)$, the block graph can have $\Omega(n^2)$ edges!)

16. 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 $O(n+e)$ time. (Note: you don't know which graph has G as its block graph.)

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

Acknowledgment

Thanks to Jared Asuncion for the images and the initial feedback!

[35] Note: a cycle cannot contain duplicate nodes.
[36] Of course, show that your answer is correct.