Connectivity

Benjamin Kleyn

Some definitions that are not important but are fun

A walk is a finite or infinite sequence of edges which joins a sequence of vertices.

A trail is a walk in which all edges are distinct.

A path is a trail in which all vertices are distinct.

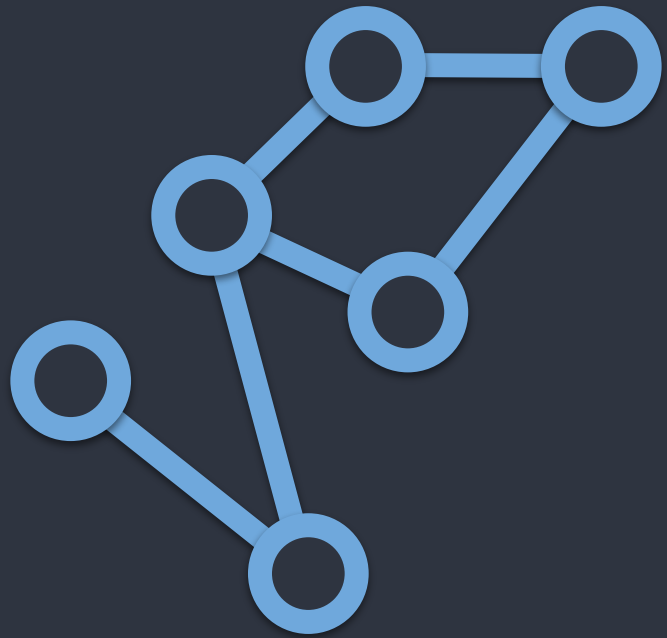
A circuit is a non-empty trail in which the first and last vertices are equal.

A cycle is a circuit in which only the first and last vertices are equal.

What exactly *is* connectivity?

In an undirected graph G, two vertices u and v are called *connected* if there exists a path from u to v.

G is then said to be connected if every pair of vertices in the graph is connected.



A connected graph.

Connected Components

A connected component of a graph is a maximal connected subgraph.

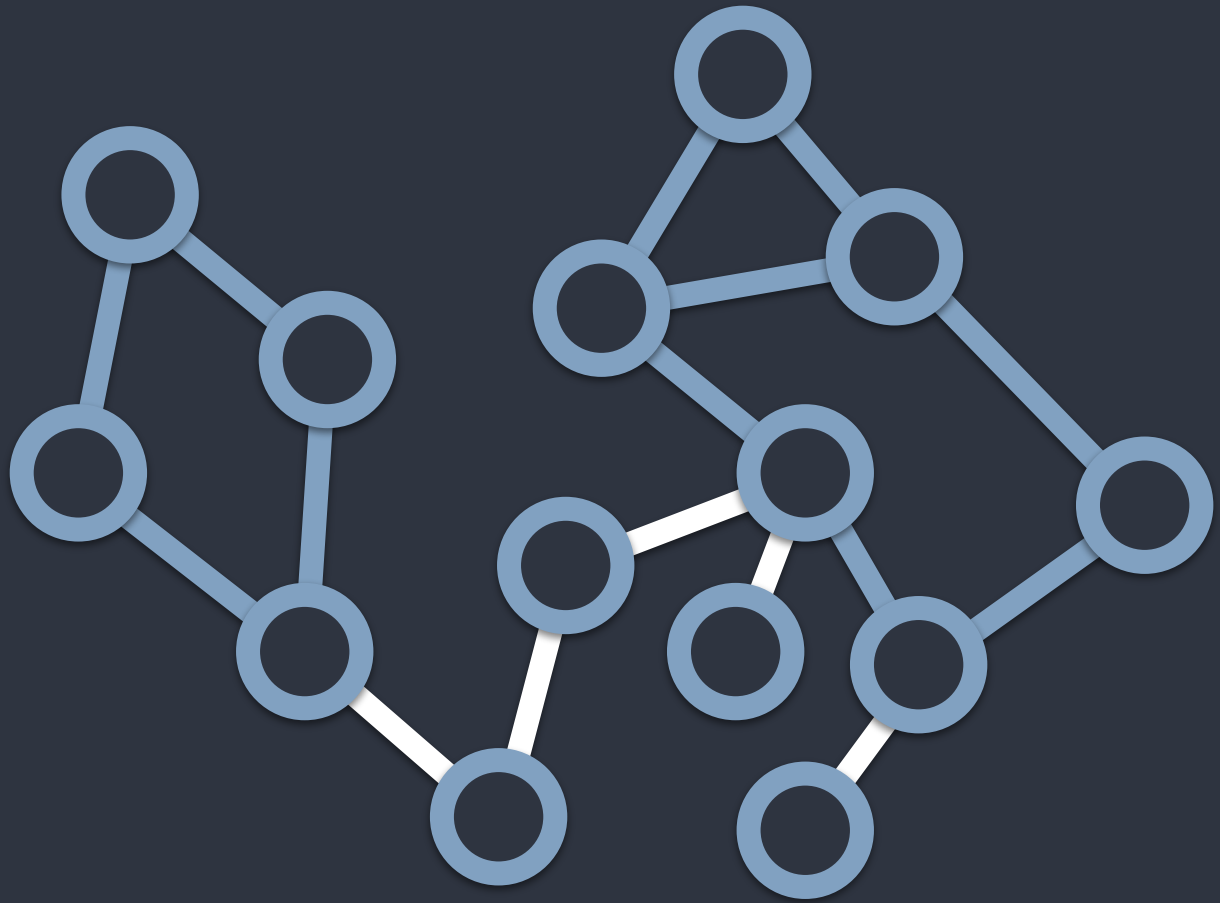
Namely, a connected component is a connected subgraph which is not a subgraph of any larger connected subgraph.

A graph with three connected components.

Bridges

A bridge, or cut-edge, is an edge whose removal increases the number of connected components in a graph.

We can find bridges in $O(n + m)$ time.



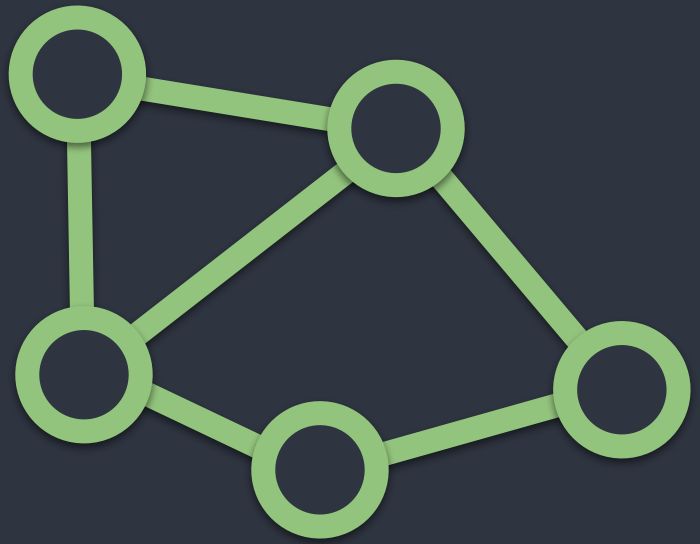
A familiar-looking graph with bridges highlighted in white.

2-Edge-connectivity

A 2-edge-connected graph is a graph where if any edge is removed, it stays connected.

In other words, it is a graph with no bridges.

(In general, a k-edge-connected graph is a graph where removing any k-1 edges will not make it disconnected.)



A 2-edge-connected graph

2-Edge-connectivity

A 2-edge-connected component of a graph is a maximal 2-edge-connected subgraph. (similar definition to connected components)

2CCs denoted by colors.

The 2CCs are connected by the bridges of the graph, and since there can be no cycle that contains a bridge,
we can conclude that compressing each 2CCs into a single vertex will always leave us with a tree. (or forest)

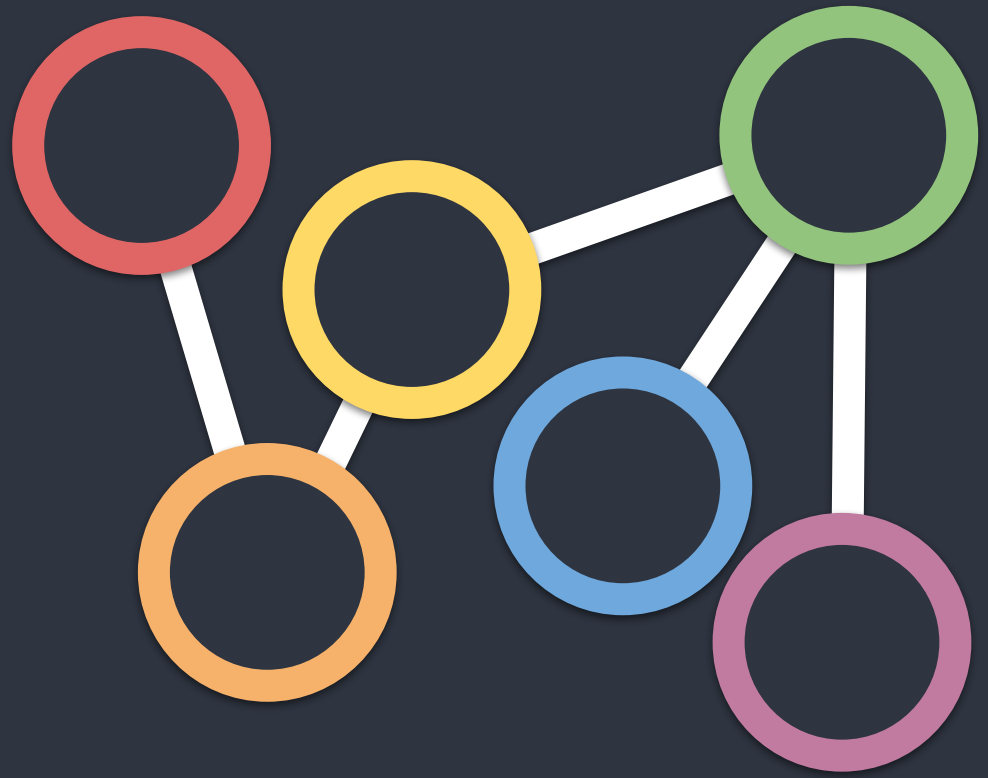
2CCs denoted by colors.

Bridge Tree

Applying above-mentioned compression on our example gives us the following tree:

Any path between vertices in the original graph must go through the bridges along the unique path between their 2CCs on the bridge tree.

We can compress a graph into a bridge tree in $O(n + m)$ time.

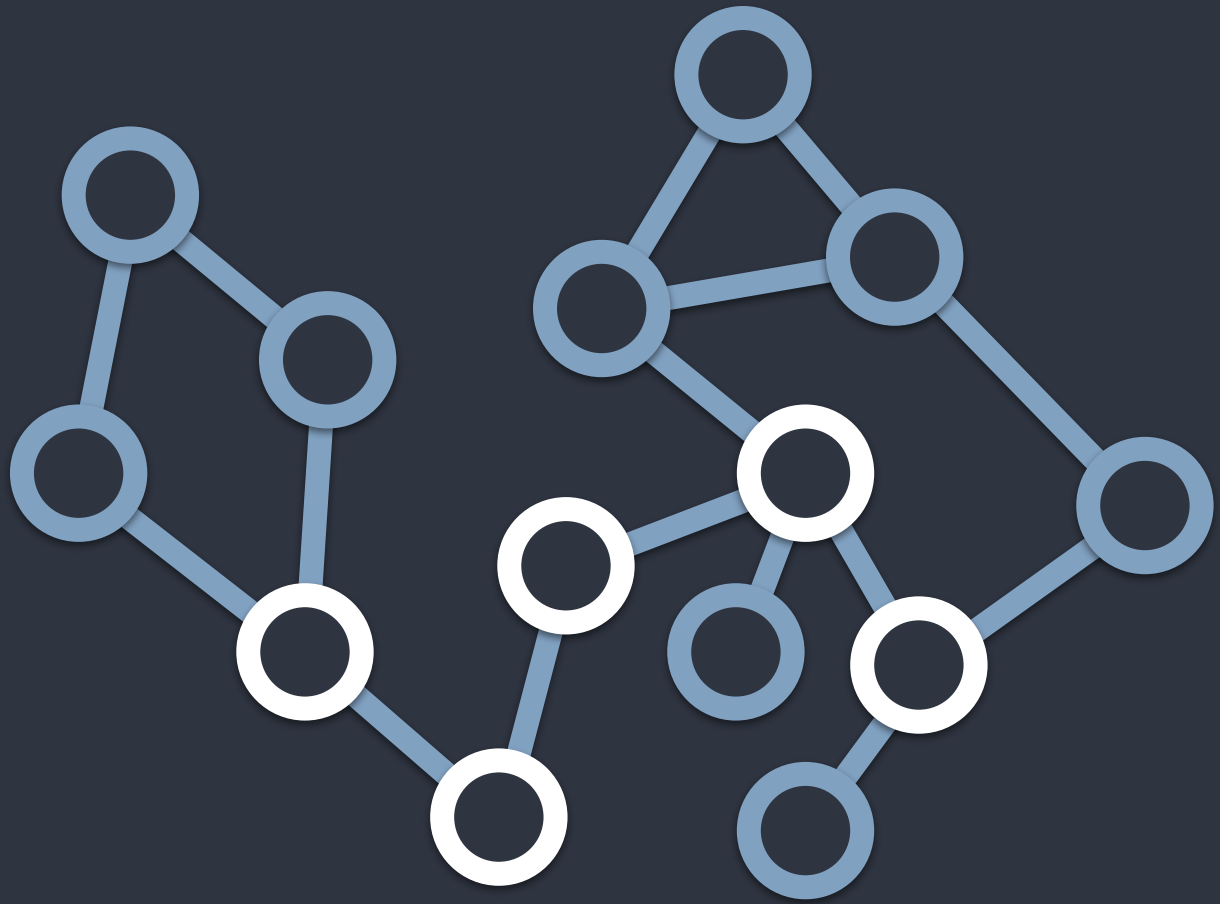


The same graph compressed into a bridge tree.

Articulation Points

An articulation point, or cutpoint, is a vertex whose removal, increases the number of connected components in a graph.

We can find articulation points in $O(n + m)$ time.



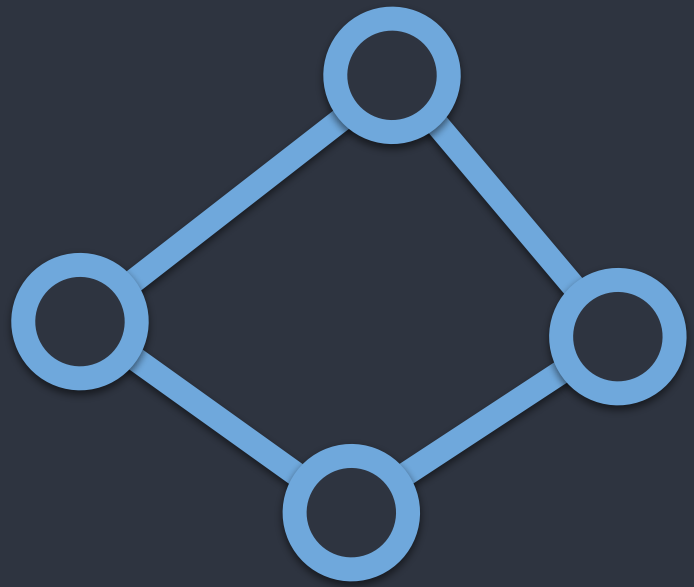
Articulation points highlighted in white.

Biconnectivity

A biconnected graph is a graph where if any vertex is removed, it stays connected.

In other words, it is a graph with no cutpoints.

If a graph has more than 2 vertices, it is biconnected if and only if there are two distinct paths between any two vertices.



A biconnected graph.

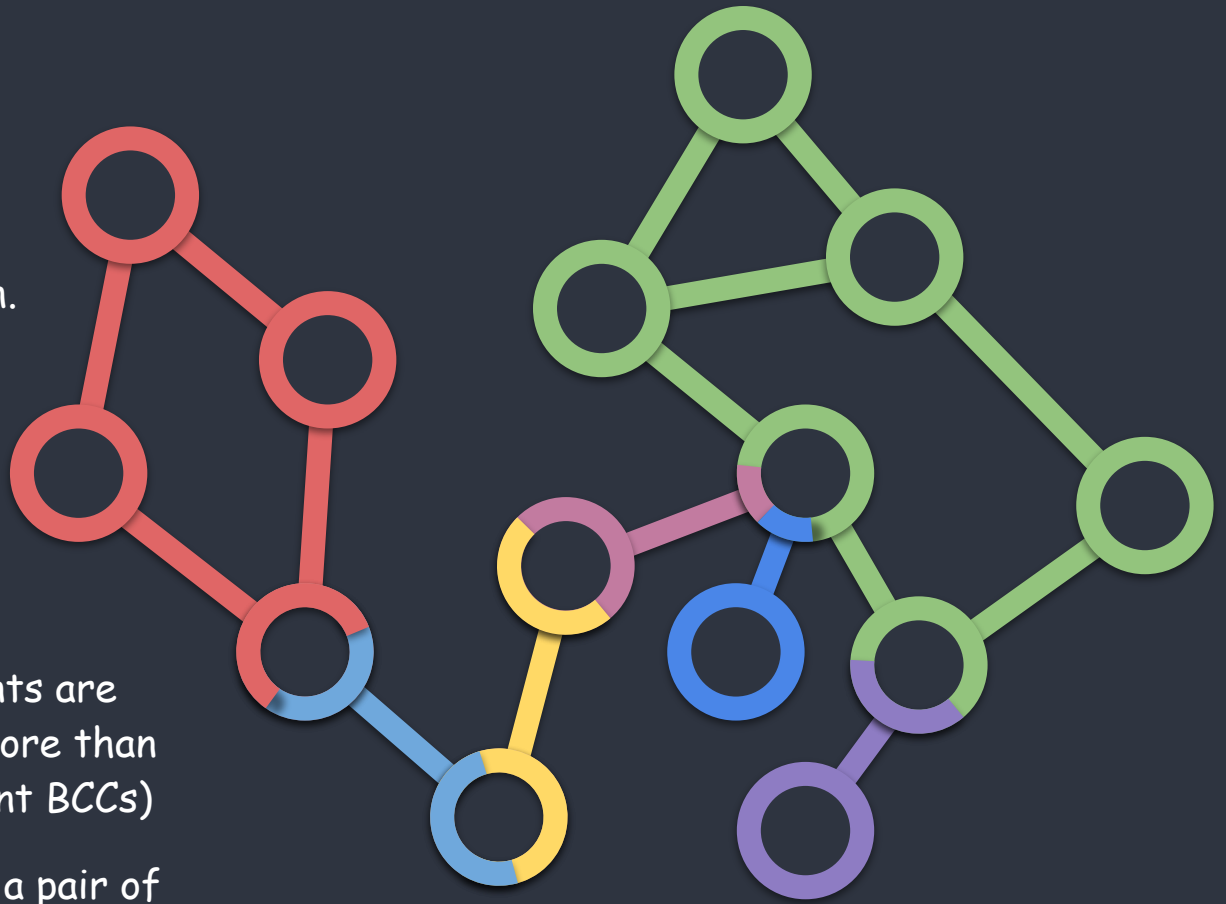
Biconnectivity

A biconnected component is a maximal biconnected subgraph.

Unlike with 2CCs, a vertex can be in multiple BCCs, as shown. (Thus we can't as easily compress the graph into a useful form)

Notice how the articulation points are exactly the points that are in more than one BCC. (They connect different BCCs)

If there exists a cycle through a pair of vertices, they are in a common BCC.

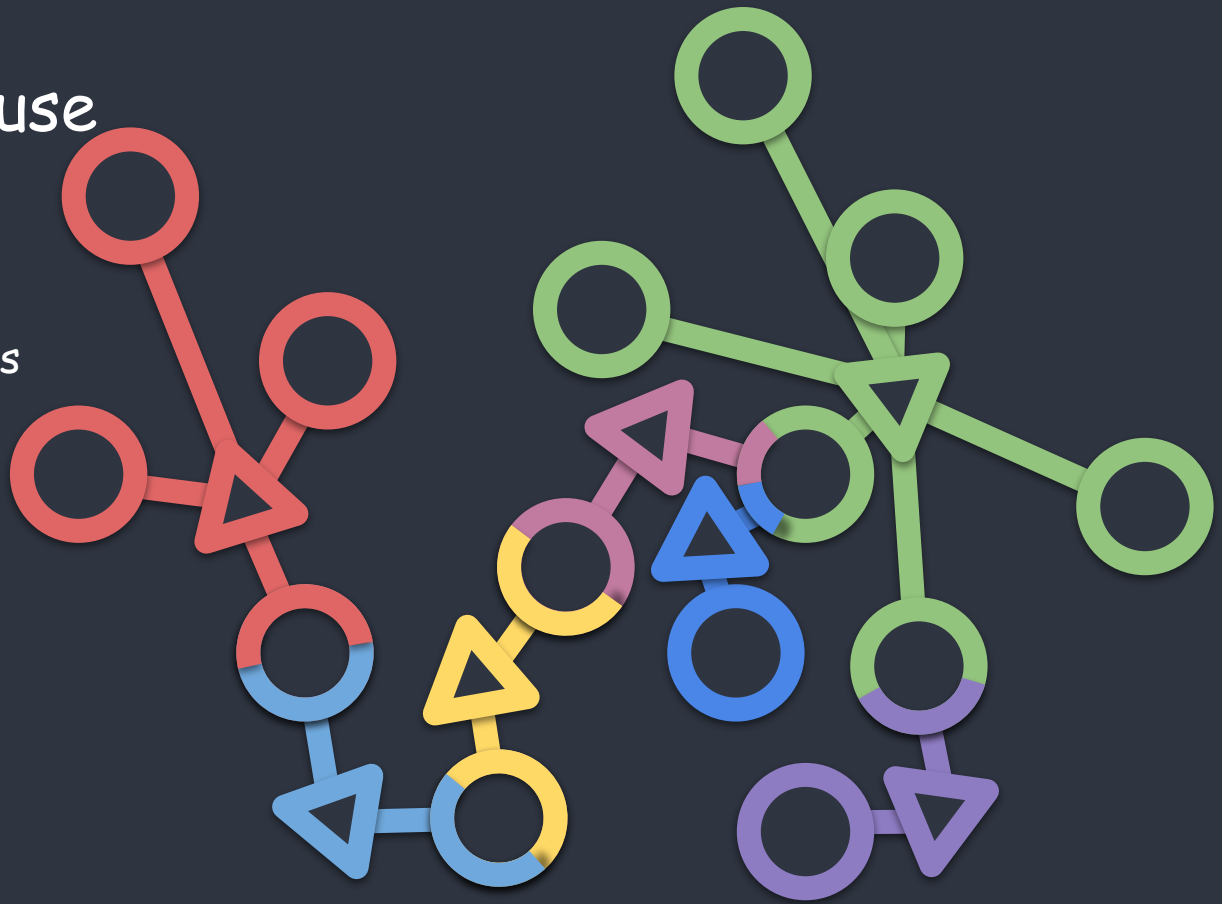


Biconnected components denoted by colors.

This is useful because

Any path between vertices in our original graph, must go through the articulation points that are on the unique path between them in this tree.

We can create this tree in $O(n + m)$ time.

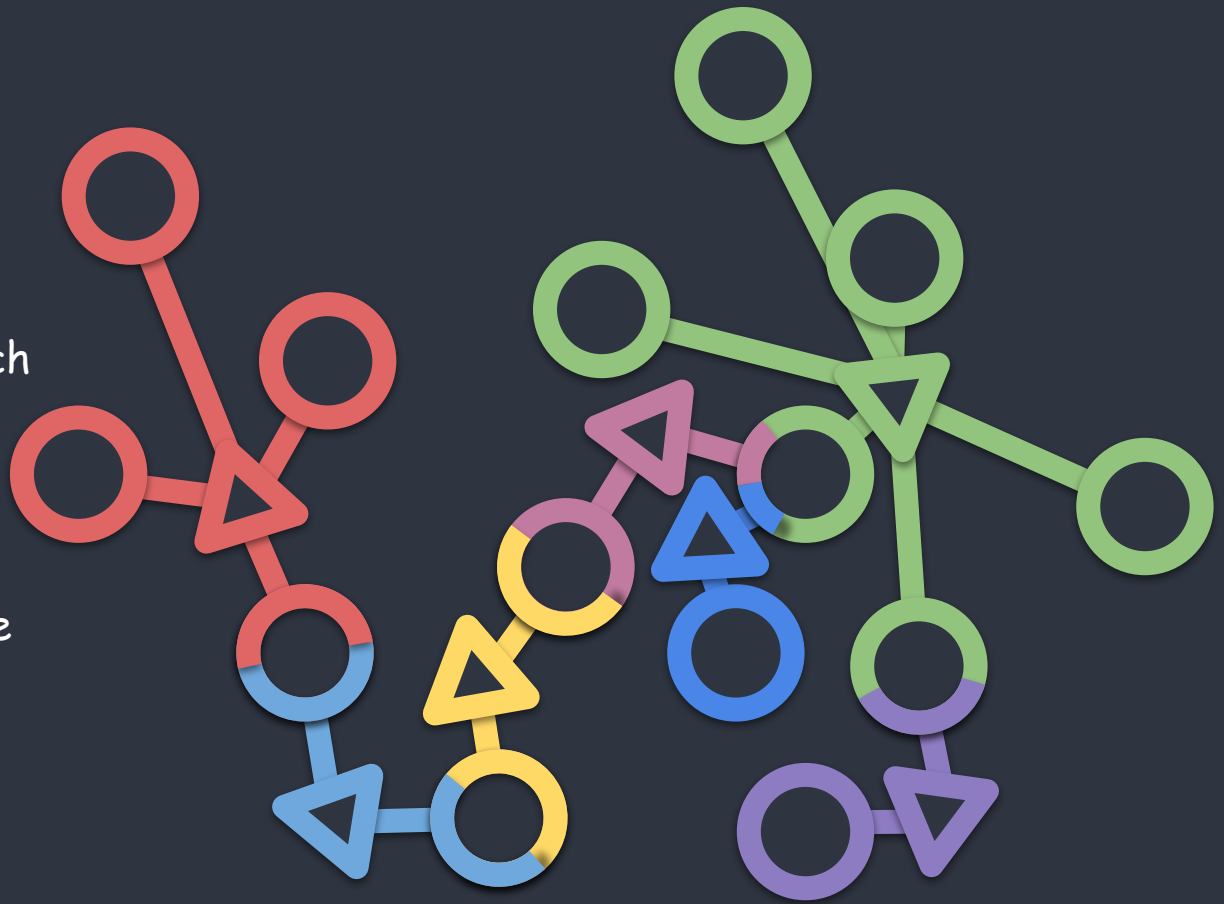


Our new tree.

What now?

Create a new graph with all the same vertices and a new representative vertex for each biconnected component.

Add edges between each biconnected component's representative vertex and the vertices which are contained within it.

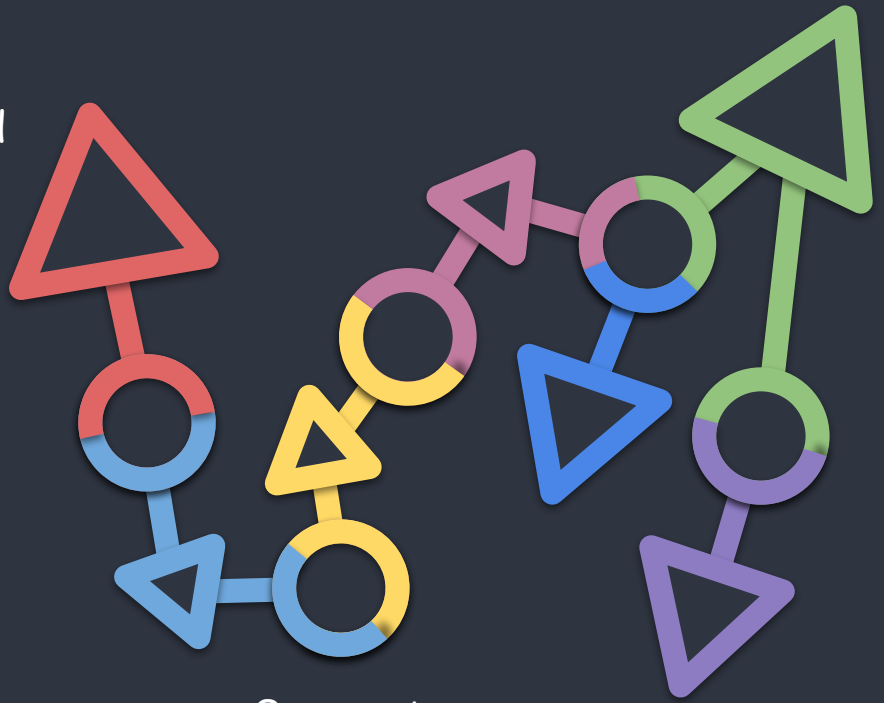


Our new tree.

Block-cut tree

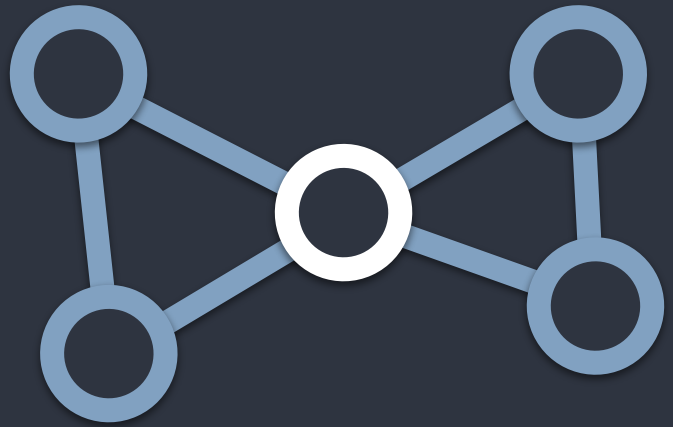
If we remove all the vertices except for the representative vertices and cut-points then what we have is called a block-cut tree.

This is more efficient and could work just as well in some cases but sometimes it helps to have all the vertices present.



Our new tree.

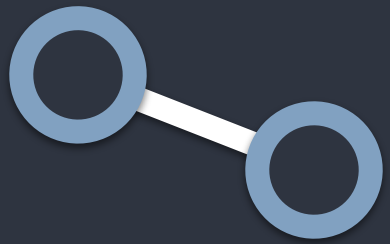
Not all 2-edge-connected graphs are biconnected.



An example of a graph that is 2-edge-connected but not biconnected.

Pretty sure that in 2-edge-connected graphs, there exists a circuit through any two vertices.

Not all biconnected graphs are 2-edge-connected.



An example of a graph that is biconnected but not 2-edge-connected.

Also an example of a biconnected graph where there is not a cycle through each pair of vertices.

Example Problems

While we let the theory sink in, let's look at some example problems to motivate the usefulness of knowing the theory and get a hang of spotting the required algorithms before actually moving on to the code.

CSES : Building Roads

Given an undirected graph with $n \leq 10^5$ vertices and $m \leq 2 \times 10^5$ edges, find a set of new edges with minimal size such that adding them would make the graph connected.

SACO 2013 : Shipping Routes

Given a simple undirected graph of $n \leq 10^5$ vertices and $m \leq 3 \times 10^5$ edges, output a list of vertices that are not on any cycle.

CSES : Strongly Connected Edges

Given a simple undirected graph of $n \leq 10^5$ vertices and $m \leq 2 \times 10^5$ edges, find a strong orientation of the graph or say if it's impossible.

i.e. choose a direction for each edge such that the resulting graph is strongly connected.

CEOI 2015 : Pipes

Given an undirected graph with $n \leq 10^5$ vertices and $m \leq 6 \times 10^6$ edges, output a list of all its bridges.

Memory limit is 16mb. (not enough to store the edges, and the graph is not necessarily simple)

PAIO 2020 : Traffic jams

Given an undirected graph with $n \leq 3 \times 10^5$ vertices and $m \leq 3 \times 10^5$ edges, answer $q \leq 3 \times 10^5$ queries of the form :

What is the minimal number of bridges over all paths from s to t?

CSES : Forbidden Cities

Given an undirected graph with $n \leq 10^5$ vertices and $m \leq 2 \times 10^5$ nodes, answer $q \leq 10^5$ queries of the form :

Does there exist a path from a to b that does not go through c?

APIO 2018 : Duathlon

Given an undirected graph with $n \leq 10^5$ vertices and $m \leq 2 \times 10^5$ edges, count the number of triples of vertices (a, b, c) such that there exists a path that goes through a, b, c in that order.

USACO Platinum December 2017 : Push a box

Given an $n \times m$ grid ($1 \le n, m \le 1500$) with hay in some cells, a starting cell, and a cell with a box, answer $q \le 5 \times 10^4$ queries of the form :

"Can Bessie push the box to the cell (r, c)?"

Note that in every query, the starting cells of Bessie and the box are the same, and it is not possible for Bessie and the box to ever be in the same cell, or for either them to be in the same cell as hay.

We actually have to write some code too :(

Let's just quickly make sure we remember how to find connected components!

Finding connected components

```
// Using a depth-first search :

bool vis[N+1] = {false};
int cmp[N+1], cnt = 0;
void dfs(const int &u)
{
    vis[u] = true, cmp[u] = cnt;
    for (const int &v : g[u])
        if (!vis[v])
            dfs(v);
}

void find_connected_components()
{
    for (int u = 1; u <= N; u++)
        if (!vis[u])
        {
            cnt++;
            dfs(u);
        }
}
```

```
// Or using union-find :

int e[N+1];
int find(const int &u)
{
    return (e[u] < 0 ? u : e[u] = find(e[u]));
}
bool unite(int u, int v)
{
    u = find(u), v = find(v);
    if (u == v) return false;
    if (e[u] > e[v]) swap(u, v);
    e[u] += e[v], e[v] = u;
}

void find_connected_components()
{
    for (int u = 1; u <= N; u++)
        e[u] = -1;
    for (const auto &[u, v] : edges)
        unite(u, v);
}
```

Tarjan's Algorithm for finding bridges and cut-points

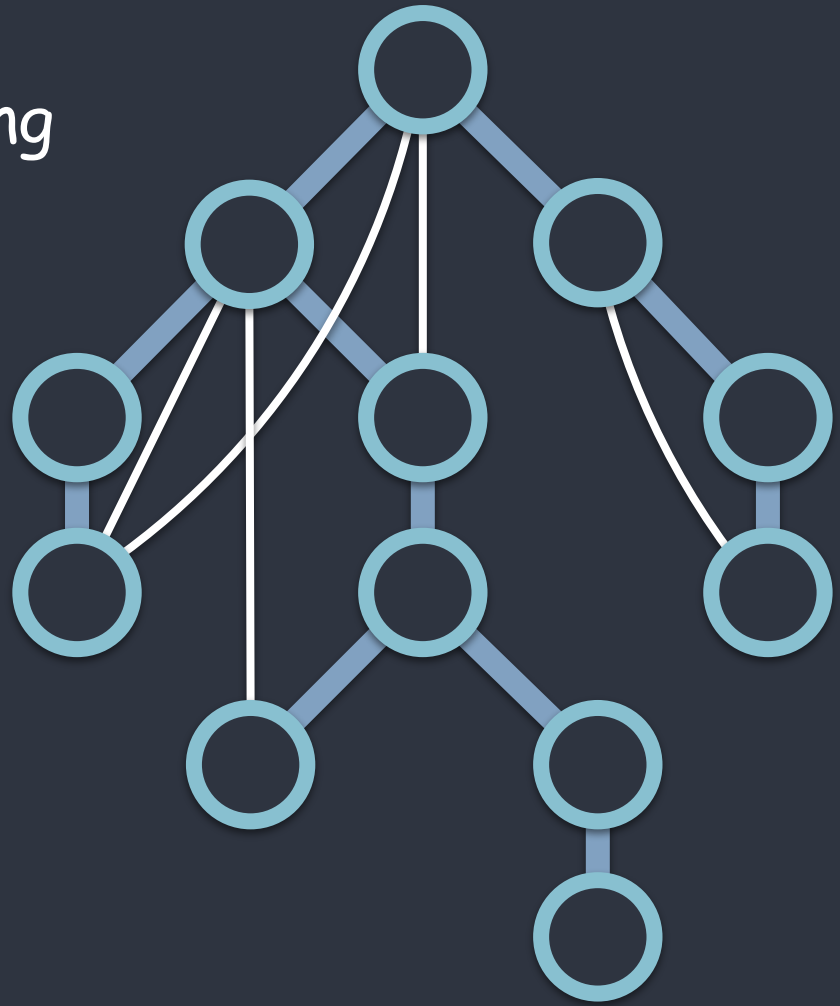
Consider the dfs-tree of our graph.

This is the tree formed by the edges traversed while doing our dfs.

The edges that are marked during the dfs are called *span-edges (or tree-edges)*. (blue)

The other edges are called *back-edges*. (white)

Note that back-edges always connect a vertex with one of its ancestors. (Otherwise it would have either been traversed earlier in the dfs, or been a span-edge)

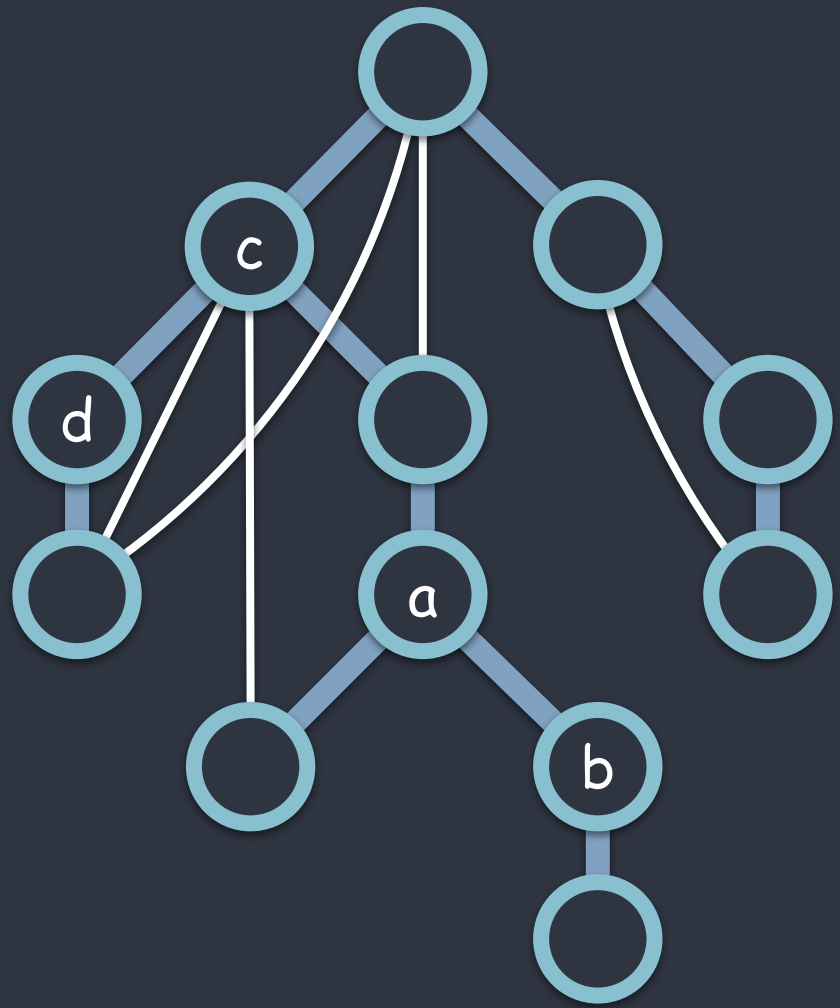


Finding bridges

Back-edges are never bridges and a span-edge (u, v) is a bridge iff there exists no back-edge that connects an ancestor of u with a descendant of v.

- (a, b) is a bridge because no back-edges edges connect an ancestor of a with a descendant of b.
- (c, d) is not a bridge because there is a back-edges connecting an ancestor of c with a descendant of d

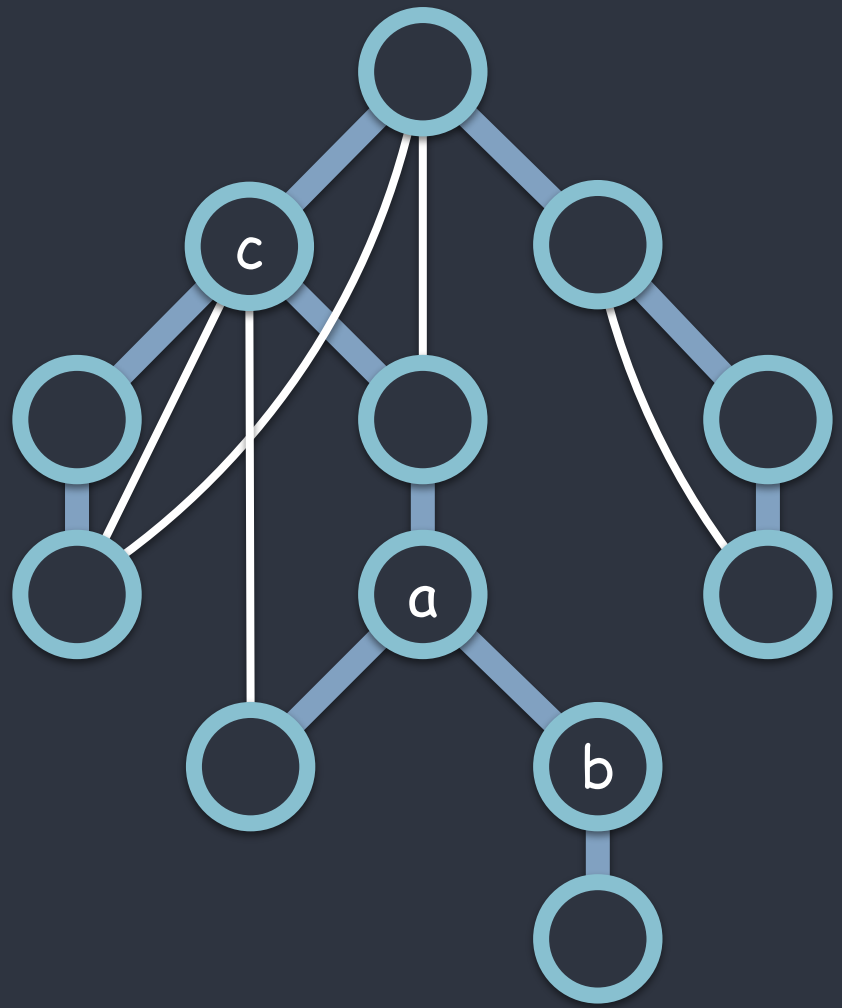
(btw, each vertex is a descendant and an ancestor of itself).



Finding cut-points

A vertex *u* is a cut-point iff it has a child v such there are no back-edges connecting a vertex in the subtree of v with a vertex strictly above u (with an exception at the root).

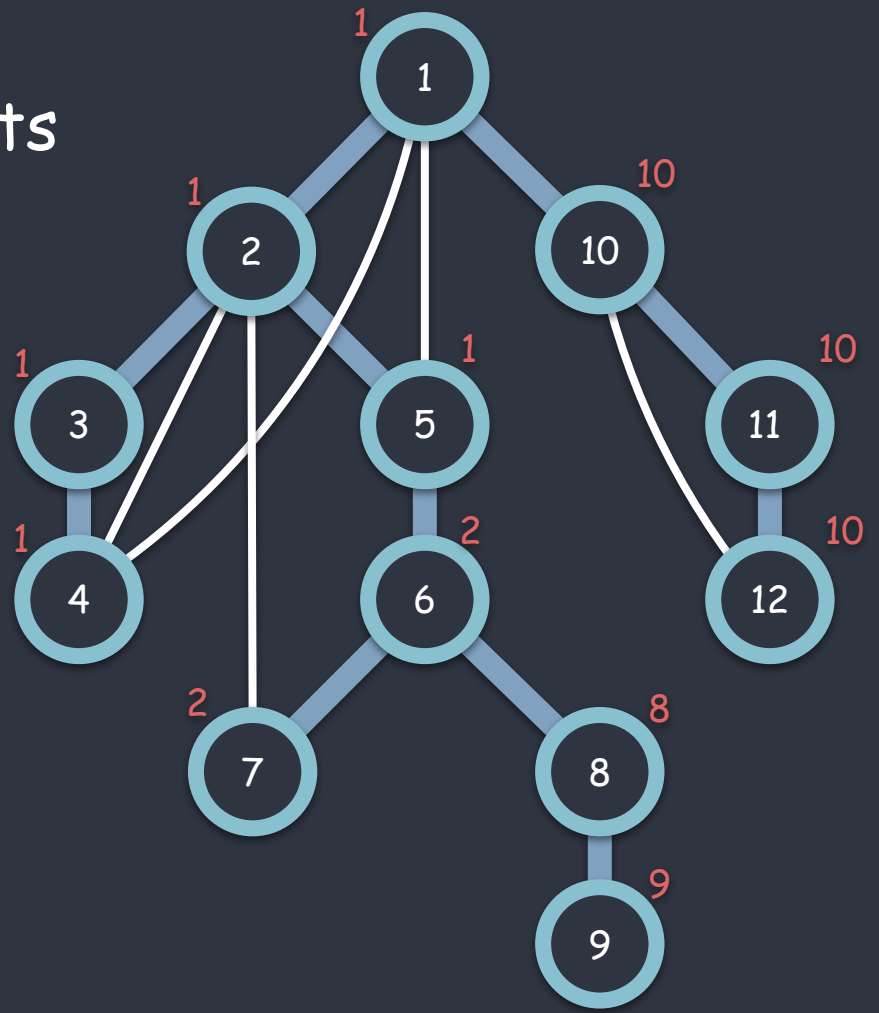
- a is a cut-point because there is no back-edge connecting a vertex in the subtree of b with a vertex strictly above b.
- c is not a cut-point because there are back-edges connecting vertices in the subtrees of each of its children with a vertex strictly above it.



Finding bridges and cut-points

One way to check this is to store for every vertex u, the entry time of u, and the earliest entry time of a vertex reached in the dfs from u (whether by a span-edge or a back-edge).

Here we have labeled the vertices by their entry times tin[u] (white) and by this new value low[u] (red).



Finding bridges and cut-points

To implement this, define the following for each u:

tin[u] = the entry time of u in the dfs.

$$\text{low}[u] = \min \left\{ \begin{array}{l} \text{tin}[u] \\ \text{tin}[v] \text{ where } (u, v) \text{ is a back-edge} \\ \text{low}[v] \text{ where } (u, v) \text{ is a span-edge} \end{array} \right\}$$

Now we can write our condition in terms of these values :

- (u, v) is a bridge if $\text{tin}[u] < \text{low}[v]$.
- u is a cut-point if u has a child v where $\text{tin}[u] \leq \text{low}[v]$, or if u is the root and has more than one child.

```
// finding bridges :
vector<int> g[N+1];
bool vis[N+1] = {false};
int tin[N+1], low[N+1], timer = 0;
```

first, declare the variables we will be needing

```
// finding bridges :
vector<int> g[N+1];
bool vis[N+1] = {false};
int tin[N+1], low[N+1], timer = 0;
void dfs(const int &u, const int &p = -1)
{
    vis[u] = true, tin[u] = low[u] = timer++;
```

create the dfs function, which passes a vertex u, and its parent p in the dfs-tree

mark u as visited, and initialize the value for tin[u] and low[u].

```cpp
// finding bridges :
vector<int> g[N+1];
bool vis[N+1] = {false};
int tin[N+1], low[N+1], timer = 0;
void dfs(const int &u, const int &p = -1)
{
    vis[u] = true, tin[u] = low[u] = timer++;
    for (const int &v : g[u]) if (v != p)
    {
        for each neighbour v of u that is not the parent of u in the dfs-tree,
```

```cpp
// finding bridges :
vector<int> g[N+1];
bool vis[N+1] = {false};
int tin[N+1], low[N+1], timer = 0;
void dfs(const int &u, const int &p = -1)
{
    vis[u] = true, tin[u] = low[u] = timer++;
    for (const int &v : g[u]) if (v != p)
    {
        if (vis[v])
            low[u] = min(low[u], tin[v]);
```

handle the case where (u, v) is a back-edge

```cpp
// finding bridges :
vector<int> g[N+1];
bool vis[N+1] = {false};
int tin[N+1], low[N+1], timer = 0;
void dfs(const int &u, const int &p = -1)
{
    vis[u] = true, tin[u] = low[u] = timer++;
    for (const int &v : g[u]) if (v != p)
    {
        if (vis[v])
            low[u] = min(low[u], tin[v]);
        else
        {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
```

otherwise, recurse from v and update low[u]

```cpp
// finding bridges :
vector<int> g[N+1];
bool vis[N+1] = {false};
int tin[N+1], low[N+1], timer = 0;
void dfs(const int &u, const int &p = -1)
{
    vis[u] = true, tin[u] = low[u] = timer++;
    for (const int &v : g[u]) if (v != p)
    {
        if (vis[v])
            low[u] = min(low[u], tin[v]);
        else
        {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if (tin[u] < low[v])
                is_bridge(u, v);
        }
    }
}
```

if (u, v) is a bridge, *do something*

```cpp
// finding cut-points :
vector<int> g[N+1];
bool vis[N+1] = {false};
int tin[N+1], low[N+1], timer = 0;
void dfs(const int &u, const int &p = -1)
{
    vis[u] = true, tin[u] = low[u] = timer++;
    int children = 0;
    for (const int &v : g[u]) if (v != p)
    {
        if (vis[v])
            low[u] = min(low[u], tin[v]);
        else
        {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if (tin[u] <= low[v] && p != -1)
                is_cutpoint(u);
            children++;
        }
    }
    if (p == -1 && children > 1)
        is_cutpoint(u);
}
```

Finding cut-points is mostly the same...

but we need to keep track of how many children the root has,

our condition here changes,

and this is how we deal with the root.

Finding 2CCs

We can easily adapt the bridge-finding algorithm to sort a graph into 2CCs and create a bridge-tree.

Simply check whether (u, v) is a bridge, and if it is not, u and v are in the same 2CC.

(Alternatively use a stack and pop from the stack after recursive dfs like Tarjan's SCC algorithm)

```cpp
// finding bridges :
vector<int> g[N+1];
bool vis[N+1] = {false};
int tin[N+1], low[N+1], timer = 0;
void dfs(const int &u, const int &p = -1)
{
    vis[u] = true, tin[u] = low[u] = timer++;
    for (const int &v : g[u]) if (v != p)
    {
        if (vis[v])
            low[u] = min(low[u], tin[v]);
        else
        {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            if (tin[u] >= low[v])
                unite(u, v);
        }
    }
}
```

If (u, v) is not a bridge, use a union-find data structure to keep track of them being in the same component.

And then to compress the graph into a new bridge graph, we add only the edges that were bridges in the original graph, between the representative nodes of each 2CC.

```
vector<int> G[N+1];
void compress_graph()
{
    for (int u = 1; u <= N; u++)
        for (const int &v : g[u])
            if (find(u) != find(v))
                G[find(u)].push_back(find(v));
}
```

Finding BCCs

To find BCCs and create our BCC graph, we add each vertex onto a stack, and when we find a span-edge (u, v) with tin[u] <= low[v], we know that u is either a cut-point or the root, so we pop from the stack until we reach v.

```cpp
// creating a BCC-graph.
vector<int> g[N+1], stck;
bool vis[N+1] = {false};
int tin[N+1], low[N+1], timer = 0;
void dfs1(const int &u, const int &p = -1)
{
    vis[u] = true, low[u] = tin[u] = timer++;
    stck.push_back(u);
    for (const int &v : g[u]) if (v != p)
    {
        if (vis[v])
            low[u] = min(low[u], tin[v]);
        else
        {
            dfs1(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] >= tin[u])
            {
                cout << u << ' ';
                while (stck.back() != v)
                {
                    cout << stck.back() << ' ';
                    stck.pop_back();
                }
                cout << stck.back() << '\n';
                stck.pop_back();
            }
        }
    }
}
```

Here we add u to the stack.

If u is a cut-point,

then we pop from the stack until we have popped v.

The vertices that we pop from the stack, along with u, form a BCC.

```cpp
// creating a BCC-graph.
vector<int> g[N+1], bcc_graph[2*N+1], stck;
bool vis[N+1] = {false};
int tin[N+1], low[N+1], timer = 0, bccs = 0;
void dfs1(const int &u, const int &p = -1)
{
    vis[u] = true, low[u] = tin[u] = timer++;
    stck.push_back(u);
    for (const int &v : g[u]) if (v != p)
    {
        if (vis[v])
            low[u] = min(low[u], tin[v]);
        else
        {
            dfs1(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] >= tin[u])
            {
                bccs++;
                bcc_graph[u].push_back(n + bccs);
                bcc_graph[n + bccs].push_back(u);
                do
                {
                    bcc_graph[n + bccs].push_back(stck.back());
                    bcc_graph[stck.back()].push_back(n + bccs);
                    stck.pop_back();
                } while (bcc_graph[n + bccs].back() != v);
            }
        }
    }
}
```

If we want a BCC graph,

we create a new representative vertex for this BCC and add edges between it and each vertex in the BCC.

```cpp
// creating a BCC-graph.
vector<int> g[N+1], bcc_graph[2*N+1], stck;
bool vis[N+1] = {false};
int tin[N+1], low[N+1], timer = 0, bccs = 0;
void dfs1(const int &u, const int &p = -1)
{
    vis[u] = true, low[u] = tin[u] = timer++;
    stck.push_back(u);
    for (const int &v : g[u]) if (v != p)
    {
        if (vis[v])
            low[u] = min(low[u], tin[v]);
        else
        {
            dfs1(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] >= tin[u])
            {
                bccs++;
                bcc_graph[u].push_back(n + bccs);
                do
                {
                    bcc_graph[n + bccs].push_back(stck.back());
                    stck.pop_back();
                } while (bcc_graph[n + bccs].back() != v);
            }
        }
    }
}
```

We could also root the tree and just store the children of each vertex

```cpp
// creating a BCC-graph.
vector<int> g[N+1], stck;
bool vis[N+1] = {false};
int tin[N+1], low[N+1], par[2*N+1], timer = 0, bccs = 0;
void dfs1(const int &u, const int &p = -1)
{
    vis[u] = true, low[u] = tin[u] = timer++;
    stck.push_back(u);
    for (const int &v : g[u]) if (v != p)
    {
        if (vis[v])
            low[u] = min(low[u], tin[v]);
        else
        {
            dfs1(v, u);
            low[u] = min(low[u], low[v]);
            if (low[v] >= tin[u])
            {
                bccs++;
                par[n + bccs] = u;
                do
                {
                    par[stck.back()] = n + bccs;
                    stck.pop_back();
                } while (par[v] != n + bccs);
            }
        }
    }
}
```

We could also store only the parent of each vertex in the tree.

This makes it quite easy and fast to check if two vertices are in a common BCC.
(The distance between them in the tree must be 2)

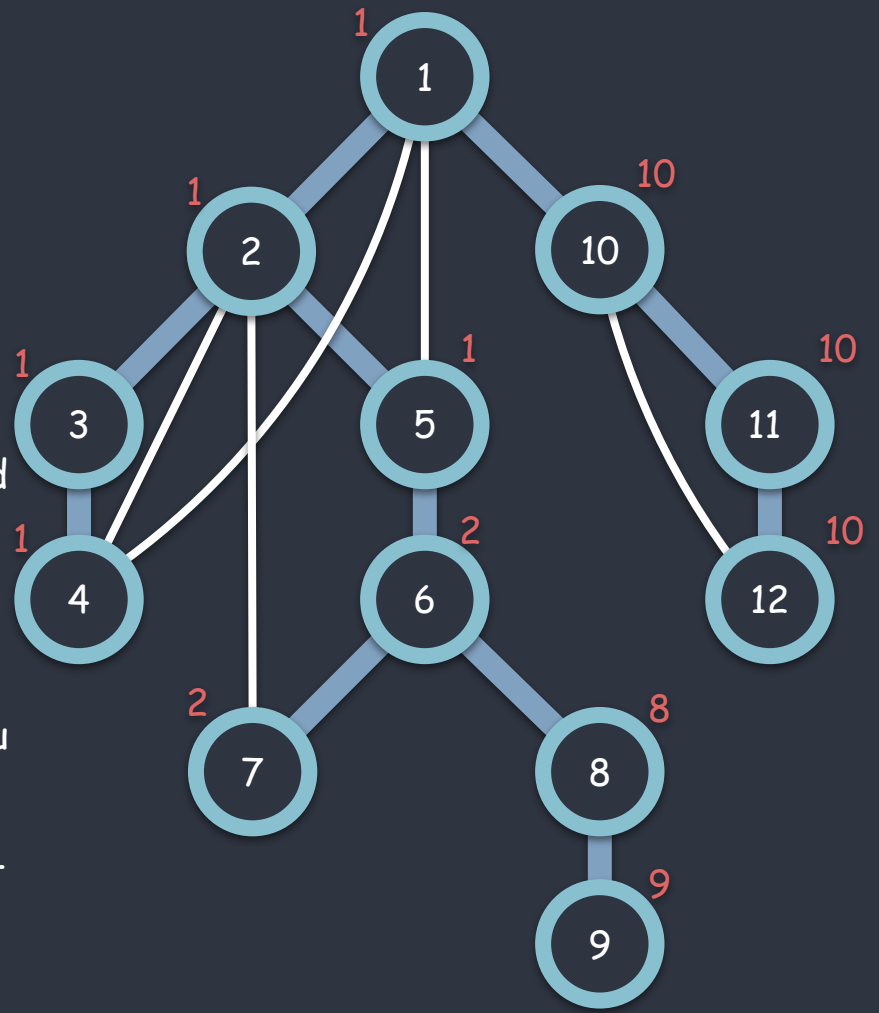
```
bool same_bcc(const int &u, const int &v)
{
    if (par[par[u]] == v) return true;
    if (par[u] == par[v]) return true;
    if (u == par[par[v]]) return true;
    return false;
}
```

how the stack thing works

if low[v] > tin[u], (u, v) is a bridge and thus {u, v} is a BCC. (so we remove v from the stack as it can't be in the same BCC as another ancestor of u) For example : (8, 9)

if low[v] == tin[u], there must be some back-edge in the subtree of v, connected to u and thus u and v are in a common BCC. (all the element on top of the stack up to u must be in this BCC, so remove up to v) For example : (10, 11)

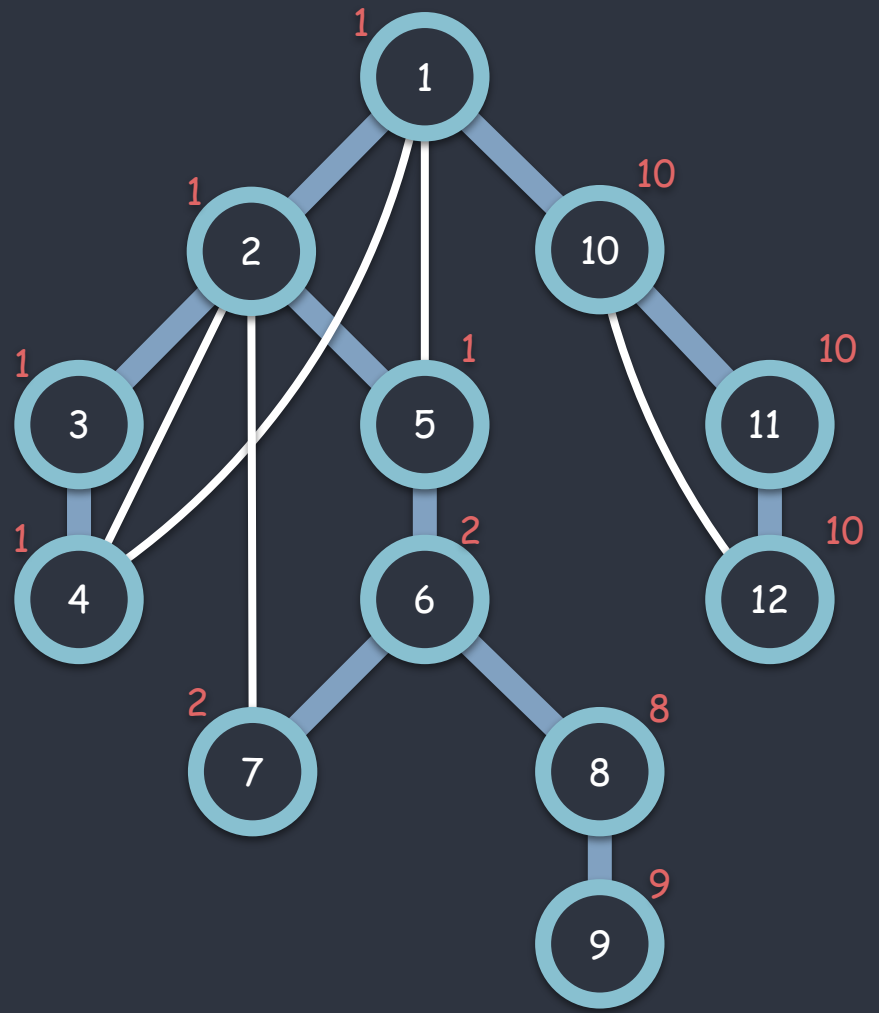
if low[v] < tin[u], there must be some back-edge in the subtree of v, connected to an ancestor of u other than u, which would mean that v is in the same BCC as u and its parent. (so we do nothing since it will be dealt with at the highest cut-point in this BCC) For example : (5, 6)



how the stack thing works

i still don't really know

just memorize the algorithm



Solutions to problems!

If you don’t like spoilers, close your eyes

CSES : Building Roads

Given an undirected graph with $n \leq 10^5$ vertices and $m \leq 2 \times 10^5$ edges, find a set of new edges with minimal size such that adding them would make the graph connected.

Solution:

For each vertex other than 1, check if it is already in the same component as 1, and add an edge if it is not.

Use DSU to do this.

SACO 2013 : Shipping Routes

Given a simple undirected graph of $n \leq 10^5$ vertices and $m \leq 3 \times 10^5$ edges, output a list of vertices that are not on any cycle.

Solution:

A vertex is on a cycle iff it is connected to a non-bridge.

(I think this is only true if the graph is simple)

CSES : Strongly Connected Edges

Given a simple undirected graph of $n \leq 10^5$ vertices and $m \leq 2 \times 10^5$ edges, find a strong orientation of the graph or say if it's impossible.

Solution:

By Robbins' Theorem, the graph has a strong orientation if and only if it is 2-edge-connected.

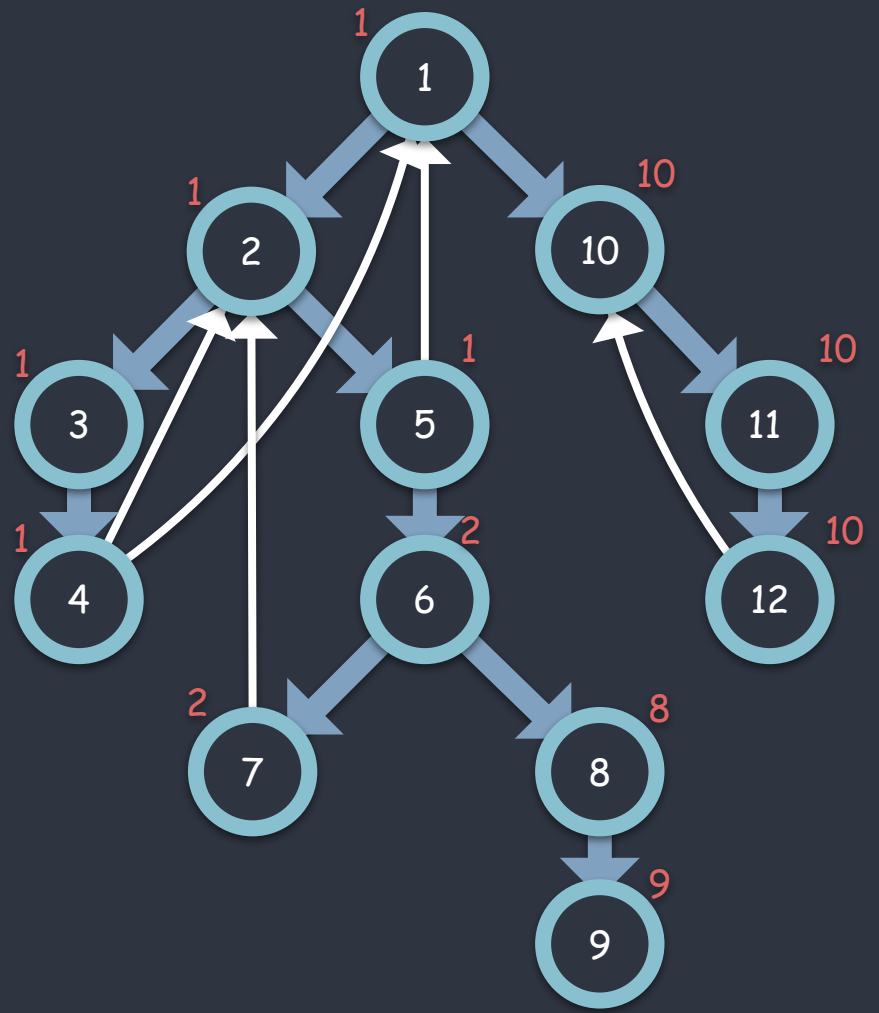
Then, simply direct each edge in the direction that it was traversed in the dfs.

Sidenote:

If we follow this process on any graph,

The 2-Edge-Connected components in the original graph must be exactly the Strongly Connected Components in this new graph.

Perhaps this is why Tarjan's bridge-finding algorithm is almost exactly the same as Tarjan's algorithm for finding strongly connected components?



CEOI 2015 : Pipes

Given an undirected graph with $n \leq 10^5$ vertices and $m \leq 6 \times 10^6$ edges, output a list of all its bridges.

Memory limit is 16mb. (not enough to store the edges, and the graph is not necessarily simple)

Solution:

Basically use two DSUs to keep track of pairs of vertices are already connected by more than one path. Edges between these vertices can't be bridges. This way we don't actually add more than 2n edges to the graph.

PAIO 2020 : Traffic jams

Given an undirected graph with $n \leq 3 \times 10^5$ vertices and $m \leq 3 \times 10^5$ edges, answer $q \leq 3 \times 10^5$ queries of the form :

What is the minimal number of bridges over all paths from s to t?

Solution:

Compress into bridge tree and do distance queries using LCA.

CSES : Forbidden Cities

Given an undirected graph with $n \leq 10^5$ vertices and $m \leq 2 \times 10^5$ nodes, answer $q \leq 10^5$ queries of the form :

Does there exist a path from a to b that does not go through c?

Solution:

Create BCC graph and check if c is on the path from a to b using LCA.

APIO 2018 : Duathlon

Given an undirected graph with $n \leq 10^5$ vertices and $m \leq 2 \times 10^5$ edges, count the number of triples of vertices (a, b, c) such that there exists a path that goes through a, b, c in that order.

Solution:

Create BCC graph and subtract bad triples using a dfs.

A triple (a, b, c) is bad if the paths from a to b and the paths from b to c must both go through some articulation point.

USACO Platinum December 2017 : Push a box

Given an n × m grid ($1 \le n, m \le 1500$) with hay in some cells, a starting cell, and a cell with a box, answer $q \le 5 \times 10^4$ queries of the form :

"Can Bessie push the box to the cell (r, c)?"

Solution:

Instead of something like

dfs(Position of Box, Position of Bessie) which would be $O(N^2M^2)$, we do

dfs(Position of Box, Direction from which Bessie pushed the box) in $O(NM)$, using

precomputed biconnected components to find the directions in which Bessie can push the box next.

Some resources where you can find better explanations and more problems to practice with :

- cp-algorithms.com/graph/bridge-searching.html
- cp-algorithms.com/graph/cutpoints.html
- usaco.guide/adv/BCC-2CC
- codeforces.com/blog/entry/99259
- codeforces.com/blog/entry/68138
- commons.wikimedia.org/wiki/File:Graph-Biconnected-Components.svg

ask me if you want to see my code for one of the problems I showed because my code is always better than the official solution even when it is slower.