Undirected DFS and Biconnected Components

In this lecture we describe the general DFS algorithm on undirected graphs, and apply it to compute the connected components, and the biconnected components of a graph.

Objectives of this lecture

In this lecture, we will

- Describe the general DFS algorithm in unidirected graphs, and its properties
- Describe a simple connected components algorthm
- Define the biconnected components of a graph
- Give a DFS algorithm for computing the biconnected components.

Recommended study resources

- CLRS, *Introduction to Algorithms*, Chapter 22, Elementary Graph Algorithms
- DPV, *Algorithms*, Chapter 3, Decompositions of graphs

1 Connectivity

Let $G = (V, E, \text{Adj})$ be a simple[1] undirected graph, where the edges are represented using an adjacency list $\text{Adj}(u) = \{v \mid \{u, v\} \in E\}$. It will often be useful to think of the undirected edge $\{u, v\}$ instead as a pair of directed edges (u, v) and (v, u). For convenience throughout this lecture we let $n = |V|$, and $m = |E|$.

Since it's going to come up twice in this lecture, let's review the notion of an *equivalence relation* on a set A. This is a relation, denoted $\equiv$, on pairs of elements of A. So for every pair of elements $x, y \in A$ either $x \equiv y$ or $x \not\equiv y$. The relation satisifies these three properties:

reflexive: For every element $x \in A$ we have $x \equiv x$.

symmetric: For every two elements $x, y \in A$ we have $x \equiv y$ iff $y \equiv x$.

transitive: For every three elements $x, y, z \in A$ if $x \equiv y$ and $y \equiv z$ then $x \equiv z$.

[1] There are not multiple edges between a pair of vertices, and no self-loops (an edge from a vertex to itself).

An equivalence relation partitions the set A into equivalence classes of elements. All elements within each class satisfy $\equiv$, and any pair of elements from different classes satisfy $\not\equiv$.

For a graph G two vertices u, and v are said to be *connected* if there exists a path in G from u to v. Since the graph is undirected it's easy to see that connectivity is an equivalence relation. The connected components of the graph is the partitioning of the vertices of G into those equivalence classes.

The following algorithm computes the connected components of G in time $O(n+m)$.

Algorithm: Connected Components

```
c ← 0
for all v ∈ V do compnum(v) ← 0
for all v ∈ V do
    if compnum(v) = 0 then
        c ← c + 1
        COMP(v)

function COMP(v)
    compnum(v) ← c
    for w ∈ Adj(v) do
        if compnum(w) = 0 then COMP(w)
end
```

Lemma 1

After running this algorithm we have:

1. For all $v \in V$ $\text{compnum}(v) > 0$,
2. there is a path from u to v iff $\text{compnum}(u) = \text{compnum}(v)$,
3. c is the number of connected components of G.

Proof. Property (1) is guaranteed to be true by syntactic analysis of the pseudocode. At the very beginning it calls COMP(v) for every vertex where $\text{compnum}(v) = 0$. The first thing that COMP(v) does is to set $\text{compnum}(v)$ to a non-zero value. Also, once assigned, the compnum value never changes.

For part (2), I claim that for every edge $\{v, w\} \in E$ we must have $\text{compnum}(v) = \text{compnum}(w)$. The reason is that one of the vertices in $\{v, w\}$ is first to get assigned a non-zero compnum value. WLOG assume that v is first. Immediately after "$\text{compnum}(v) \leftarrow c$" is executed, all the neighbors of v are processed. So by the time we reach "if $\text{compnum}(w) = 0$ then COMP(w)" either $\text{compnum}(w)$ is already equal to c, or it's imediately assigned to c as a result of the call to COMP(w).

The above argument implies (by induction on the length of the path from v to w) that if there

is a path from v to w then they have the same compnum value. The converse is that if there is no path from v to w then the search of the component containing v will not find w. So that after v is processed and its component is labeled, then c will be incremented before the next search. So w will end up with a different compnum value.

Finally part (3) follows from the fact that the components are numbered $1, 2, \ldots, c$. □

2 General DFS in Undirected Graphs

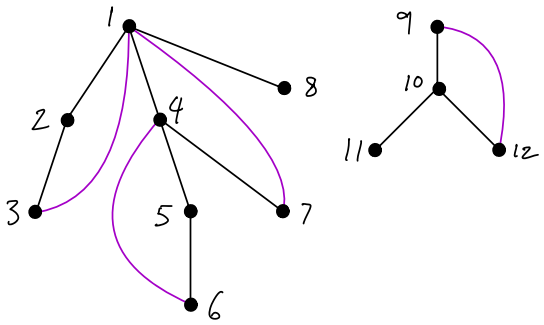
In order to solve more intricate problems with DFS we're going to take a more nuianced look at the process. The version below partitions all the edges of the graph into *tree edges* and *back edges*. The tree edges form forest of rooted trees, each of which spans a connected component of G. And we'll prove that the back edges connect a node to one of its ancestors in the tree.

Algorithm: Generic DFS in Undirected Graphs

```
i ← 0
for v ∈ V do num(v) ← 0
for v ∈ V do if num(v) = 0 then DFS(0, v)

function DFS(u, v)                    [u is v's parent or 0, v is node being searched]
    i ← i + 1
    num(v) ← i
    for w ∈ Adj(v) do
        if num(w) = 0 then DFS(v, w)  [(v, w) is a tree edge]
        else if w = u then            [(v, w) is a reverse tree edge]
        else if num(w) < num(v) then  [(v, w) is a back edge]
        else                          [(v, w) is reverse back edge]
end
```

This classification of the edges is not a property of the graph alone. It also depends on the ordering of the vertices in Adj(v) and on the ordering of the vertices in the loop that calls the DFS procedure. The following figure shows a possible DFS tree. The tree edges are shown in black and the back edges are shown in purple.



This process examines all the vertices and edges of the graph. The call DFS$(*, v)$ is made exactly once for each vertex v in the graph. Each edge is placed into one of two classes by the algorithm: tree edges, and back edges. The for loop inside of DFS is executed exactly twice for each edge of the graph, once in one direction and once in the reverse direction.

The first time a vertex v is found it is the result of either a top-level call to DFS$(0, v)$, or it is the result of a recursive call to DFS(u, v). The first case the vertex v becomes a root of a new tree. The second case results from exploring tree edge (u, v), and v becomes a child of u in the DFS tree. As a result the tree edges form a forest of rooted trees, one for each connected component of G. The lowest numbered node in a tree is its root. And each child has a number that is greater than its parent. All of these are obvious. The following lemma is not quite as obvious.

Lemma 2

The back edges discovered in a DFS go from a node to one of its ancestors.

Proof. First a general observation about the DFS process. Consider the call DFS$(*, x)$ which gave x its num value. What happens between the moment this call is made and the moment that the call completes? What happens during that period is that exactly the entire subtree of the DFS tree rooted at x is explored.

Say you're looking at an edge (v, w) inside the for loop inside DFS. And suppose that num$(w) <$ num(v). Consider the moment in the past when w was first explored. At that time v had not been explored (it had no num value). By virtue of the existance of the edge (w, v) we know that by the time the call DFS$(*, w)$ (that labeled w) completed it must be the case that v has been explored. Because if it had not been discovered before the edge (w, v) was processed, then it will be discovered when that edge is processed.

Therefore we know that v is in the subtree of the DFS tree rooted at w, and w is therefore an ancestor of v. This completes the proof, since for a back edge (v, w) num$(w) <$ num(v). □

By the way, when work continues to complete DFS$(*, w)$ it will find the edge (w, v) (and label it a reverse back edge) – the reverse of back edge (v, w).

3 Biconnected Components

From now on we're going to assume our graph $G = (V, E)$ is simple, connected, and has at least two vertices. Recall that a *simple cycle* is a cycle with no repeated vertices.

Definition 1

A vertex v is an *articulation point* if its removal[a] disconnects the graph.

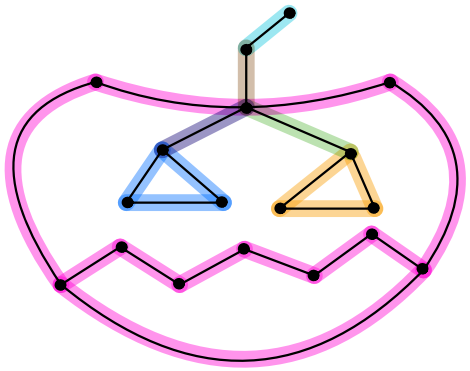
[a] Removing a vertex from the graph means removing it and all of its incident edges.

Definition 2

A graph G is *biconnected* iff it has no articulation point.

It turns out that if the graph is not biconnected, then its edges can be partitioned into disjoint sets such that the graph represented by each of the sets of edges is biconnected. To do this we define an equivalence relation on the edges, and this will partition the graph into biconnected components. We say that edges $e, f \in E$ satisfy $e \equiv f$ if either $e = f$, or the graph has a simple cycle passing through e and f.

In the graph below there are seven bi-connected components (colored different colors), and four articulation points. The articulation points are the vertices which are in more than one biconnected component.



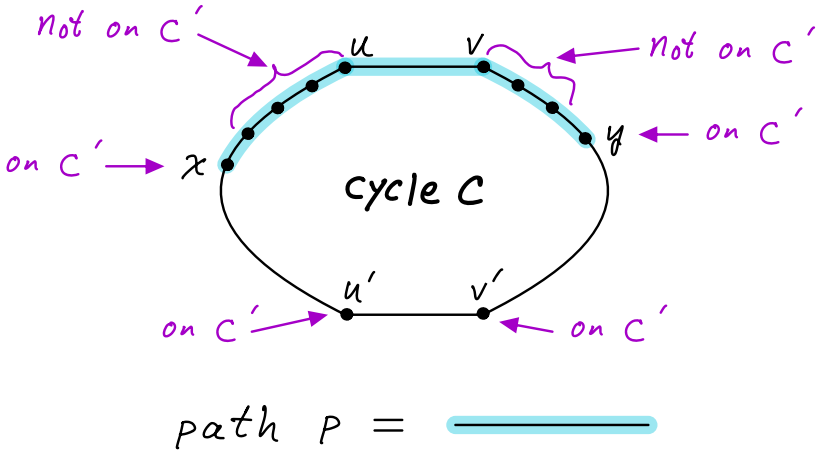
Lemma 3

$\equiv$ is an equivalence relation.

Proof. [2] We need to prove the three properties of an equivalence relation: reflexivity: $e \equiv e$, symmetry: $(e \equiv f) \Rightarrow (f \equiv e)$, and transitivity: $(e \equiv f) \wedge (f \equiv g) \Rightarrow (e \equiv g)$. The first two follow trivially from the definition of $\equiv$. The tricky one is transitivity.

Let's focus on three edges $\{u, v\} \equiv \{u', v'\}$ and $\{u', v'\} \equiv \{u'', v''\}$. We need to show that $\{u, v\} \equiv \{u'', v''\}$. Let c and c' be the two cycles involved, respectively. Assume WLOG that u, u', v', v occur in that order around c. Let x be the first vertex on the segment of c from u to u' that also lies in c'; x must exist since $u' \in c'$, at least. Similarly, let y be the first vertex on the segment of c from v to v' that also lies on c'; y must exist since $v' \in c'$. Also $y \neq x$ since c is simple. The following figure illustrates this construction.

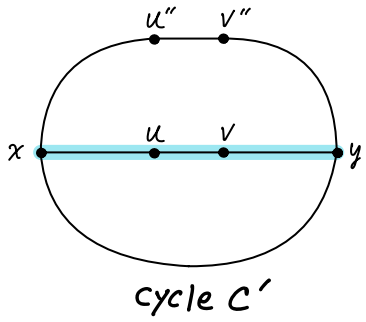
[2] This proof is adapted from Kozen's 1991 book, *The Design and Analysis of Algorithms*.



What is shown is all the vertices of the cycle c. Cycle c' is not shown because it could be intermingled in an almost arbitrary way with c. Note that in this figure it is possible that the parts that are "not on c'" might be the empty sets (e.g. x could equal u). It's also possible that $x = u'$. Similarly it is possible that $y = v$ or $y = v'$.

But what we do know, in any case, is that the path shown in turquoise exists. Call that path p. Path p starts at x, walks around c until it gets to u, then traverses edge (u, v) then continues on to y. This path has the property that (1) it starts at x and ends at y, both of which are on c', (2) none of the other vertices on the path are on c', and (3) the path contains edge (u, v).

Now we will make use of path p and cycle c' to complete the proof. The following figure shows topologically the relationship between p (shown in turquoise) and the cycle c'. Other arrangements are possible, such as the swapping of u'' and v'', but in any case there is a simple cycle through both (u, v) and (u'', v'').

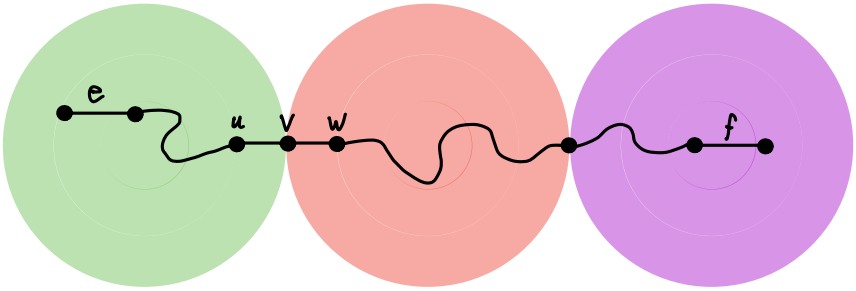


□

Theorem 1

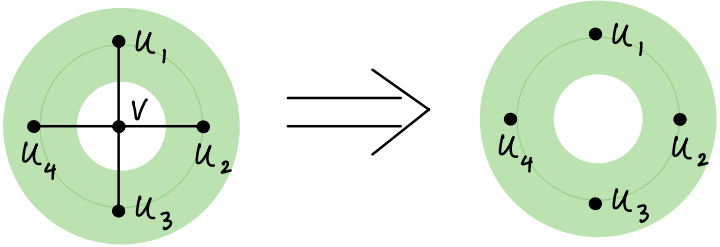
A graph G is biconnected $\Longleftrightarrow$ it either consists of a single edge, or for every two distinct edges e and f in G we have $e \equiv f$.

Proof. ($\Rightarrow$): Assume the right hand side is false. This means that G is not a single edge and also that it is not the case that all edges e and f satisfy $e \equiv f$. This means that there are two edges that e and f such that $e \not\equiv f$. Since the graph is connected, there must be a path from one end of e to one end of f. Along this path there must be three distinct vertices u, v, and w such that edge $(u,v) \not\equiv (v,w)$. This means that there cannot be a simple path from u to w in G that does not make use of v, because such a path would prove that $(u,v) \equiv (v,w)$. Therefore removal of v disconnects the graph. So v is an articulation point, and the graph is not biconnected, and the left hand side is false.



In the figure above, each of the shaded blobs represents a biconnected component of the graph. The path from e to f must cross a point where the edge $(u,v) \not\equiv (v,w)$, thus proving that v is an articulation point.

($\Leftarrow$): Assuming the right hand side is true, we'll prove the left hand side. Take an arbitrary vertex in the graph v. We'll show that v cannot be an articulation point. If v is incident to just one edge, then it cannot be an articulation point (the remaining graph remains connected upon its removal.) So suppose v is incident to at least two edges. Let the vertices to which v is adjacent be $u_1, u_2, \ldots, u_k$. For any $1 \leq i < j \leq k$, we know by the equivalence relation that there exists a simple path from u_i to u_j that does not make use of v. Thus the graph remains connected upon the removal of vertex v and edges $\{v,u_1\}, \ldots, \{v,u_k\}$. Thus v is not an articulation point.



In the figure above, the edges in the green region (not shown) must connect u_1, u_2, u_3 and u_4 together. Thus the graph remains connected when v and its incident edges are removed. □

The proof above gives us the following corrolary.

Corollary 1

A vertex v is an articulation point if and only if it is in two biconnected components.

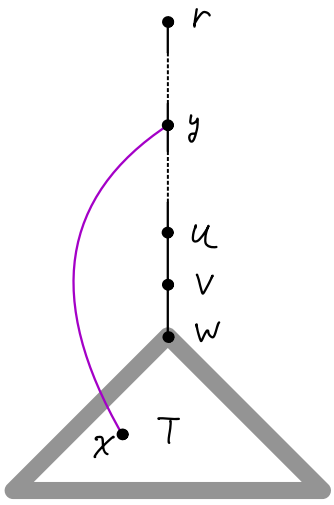
4 Using DFS to Compute the Biconnected Components

Consider the DFS tree that occurs when a DFS is done in a graph with several biconnected components. We'll be looking at the DFS tree that results from such a search. When we use the terms "descendant" and "ancestor" we always consider a vertex v to be an ancestor of itself and a descendant of itself.

Lemma 4

Let (u,v) and (v,w) be two adjacent tree edges in a DFS tree searching a graph G. Then $(u,v) \equiv (v,w) \Longleftrightarrow$ there exists a back edge from some descendant of w to some ancestor of u.

Proof. ($\Rightarrow$) Suppose $(u,v) \equiv (v,w)$. Then, by the definition of $\equiv$ there must be a simple path in the graph that goes from w to u without passing through v. Let T be the subtree of the DFS tree rooted at w. The path may first wander around inside of T for a while. It could use tree edges or back edges within T. Eventually it must leave T. The *only way that this can happen* is if it follows a back edge from some node x that is a descendant of w to some node y that is an ancestor of u (the path is not allowed to use v). The figure below shows the required back edge in purple.



($\Leftarrow$) The purple back edge from x to y exists. There is a path of tree edges from w to x, and there is a path of tree edges from y to v. This proves that $(u,v) \equiv (v,w)$. □

When the search DFS(v,w) returns it will be useful to know if (u,v) and (v,w) are in the same biconnected component. There is a very elegant way to do this using the following definition.

Definition 3

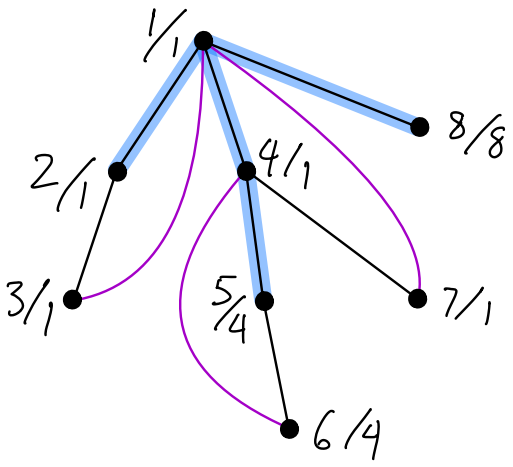
In a DFS, lowpt(v) is defined to be the lowest numbered vertex reachable from v by following zero or more tree edges, and then zero or one back edge.

To see the use of this, consider a tree edge (v,w). If lowpt$(w)<$ num(v), then using Lemma 4 we can immediately conclude that $(u,v)\equiv(v,w)$.

Also, we can easily compute this function during the DFS. You can look at how it's done in the pseudo-code below. But the idea is that lowpt(v) is just the minimum value of num(v), and the lowpt values of all the children (in the tree) of v, and the num values of all the nodes you can reach via a back edge from v. This inductively gives you exactly what the definition requires.

A biconnected component could consist of a single tree edge. (It cannot consist of a single back edge because a back edge creates a cycle with some tree edges.) Or it could consist of both tree edges and back edges. But in all cases the first edge of the biconnected component that is discovered in a DFS must be a tree edge. We call this edge the *base edge* of the biconnected component. It's going to be important in the algorithm we develop to be able to identify the base edges.

The figure below shows the DFS tree of a graph. The vertices are labeled with num/lowpt values. The edges highlighted in blue are tree edges that are the base edges of each of the four biconnected components of this graph.



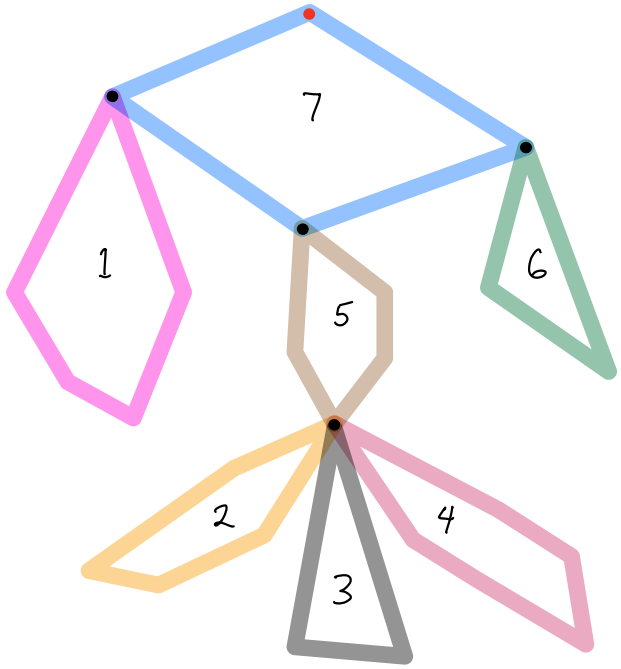
Theorem 2

A tree edge (v,w) is the base edge of its biconnected component $\iff$ lowpt$(w)\geq$ num(v).

Proof. ($\Rightarrow$): Assume that the righthand side is false, and that lowpt$(w)<$ num(v). We can apply

Lemma 4. By virtue of the fact that lowpt$(w) <$ num(v) we know there must be a back edge from some descendant of w to some u, a strict ancestor of v. (See the figure in the proof of that lemma.) It follows from the lemma that $(u,v) \equiv (v,w)$. But the edge (u,v) was explored earlier than (v,w), which means that (v,w) is not the base of its biconnected component.

($\Leftarrow$): Let's look for a cycle that starts at v, uses edge (v,w), then continues, eventually returning to v. The path can continue for some time in the subtree rooted at w, using back edges and tree edges. All of these edges are discovered later than edge (v,w). Eventually it could use a back edge that takes it back to v. But it cannot take a back edge like that to any ancestor of v because lowpt$(w) \geq$ num(v). Therefore such a cycle can not use any edge that is discovered earlier than (v,w). □



There is one more idea that we need to complete an algorithm. As we do the search, we will put the tree edges and back edges visited onto a stack in the order that we discover them. At opportune moments (as we'll see) we pop groups of edges off of it, that will form our biconnected components.

The figure above is a schematic diagram of the DFS tree. Each colored polygon represents a biconnected component (BC). The black dots represent articulation points. The red dot is the root of the tree. The DFS search proceeds from top to bottom and left to right.

The search begins in BC 7. As we search it, edges are being put onto the stack. Eventually the search enters BC 1. Let (x,y) be the first edge put onto the stack when the search enters enters BC 1. All the edges of BC 1 are put, consecutively, onto the stack. When the call to DFS(x,y) finally completes, we recognize that edge as the base of a BC. (We do it by comparing lowpt(y) with num(x) and finding that lowpt$(y) \geq$ num(x).) So we bulk-pop off all the edges on the

stack up to and including (x, y).

This process continues inside BC 7. Eventually the search enters BC 5, then it enters BC 2. And at the right time BC 2 is all popped off the stack. The same for BC 3 and BC 4. Now the only edges on the stack are some from BC 5 (on the top) followed by some from BC 7. When we finally recognize the end of BC 5 and pop those off, we resume work on BC 7. Eventually BC 6 is processed and removed. Finally, at long last, we finish BC 7 when the call to the first edge of BC 7 completes.

Below is the full commented pseudo-code for this $O(n+m)$ algorithm.

Algorithm: Biconnected Components of a Graph

```
i ← 0
S ← empty stack
for v ∈ V do num(v) ← 0
for v ∈ V do if num(v) = 0 then BICON(0, v)

function BICON(u, v)
    i ← i + 1
    num(v) ← i
    lowpt(v) ← num(v)
    for w ∈ Adj(v) do
        if num(w) = 0 then                    [(v, w) is a tree edge]
            push (v, w) onto S
            BICON(v, w)
            lowpt(v) ← min(lowpt(v), lowpt(w))
            if lowpt(w) ≥ num(v) then
                [Edge (v, w) is the base of its biconnected component.]
                [v is either the root of the tree, or an articulation point, or both.]
                Form a new biconnected component consisting of
                all the edges on the stack above and including
                (v, w). Remove these edges from the stack.
        else if w = u then                    [do nothing – (v, w) is a reverse tree edge]
        else if num(w) < num(v) then          [(v, w) is a back edge]
            Push (v, w) onto S
            lowpt(v) ← min(lowpt(v), num(w))
        else                                  [do nothing – (v, w) is a reverse back edge]
end
```