451: DFS and Biconnected Components

G. Miller, K. Sutner
Carnegie Mellon University
2020/09/22

Ancient History

A quote from a famous mathematician (homotopy theory) in the first half of the 20th century:

> *Graph theory is the slum of topology.*
>
> *J. H. C. Whitehead*

He is a nephew of the other Whitehead. Actually, he talked about "combinatorics", but early in the 20th century "graph theory" was a big part of combinatorics (everything outside of classical mathematics).

Not a good sign.

Less Ancient History

But, things have improved quite a bit since.

We have not begun to understand the relationship between combinatorics and conceptual mathematics.

J. Dieudonné (1982)

Some would suggest that computers will turn out to be the most important development for "conceptual mathematics" in the 20th century. Since a lot of combinatorics and graph theory is highly algorithmic, it naturally fits perfectly into this new framework.

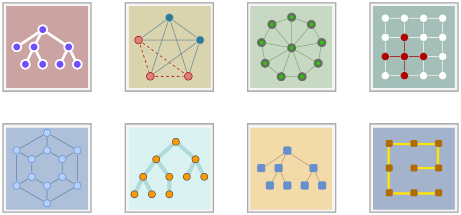
What's a Graph?

Depends on the circumstances. We have a structure $G = \langle V, E \rangle$ comprised of

Vertices a collection of objects (also called nodes, points).

Edges connections between these objects (also called arcs, lines).

We'll formalize in a minute, but always think in terms of pictures—but with requisite caution.



Different Types

There are several different versions of graphs in the literature. Key issues are

- directed versus undirected (digraphs vs. ugraphs)
- self-loops
- multiple edges

Also, very often one needs to attach additional information to vertices and/or edges.

- vertex-labeled graphs
- edge-labeled graphs

We will not deal with multiple edges, but may occasionally allow loops.

Abstractly, we are really dealing with a structure $\langle V, \rho \rangle$ where ρ is a binary relation on V (at least when multiple edges are not allowed).

Notation

n is the cardinality of V, and m the cardinality of E.

Directed edges are often written $e = u \to v$ or $e = (u, v)$ or $e = uv$.

Undirected edges are $e = u{-}v$ or $e = \{u, v\}$ or $e = uv$.

$u = \mathsf{src}(e) \in V$ is the source,

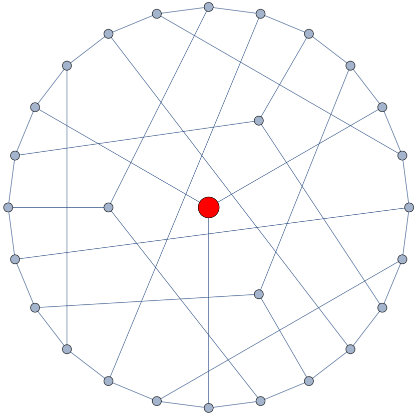
$v = \mathsf{trg}(e) \in V$ is the target.

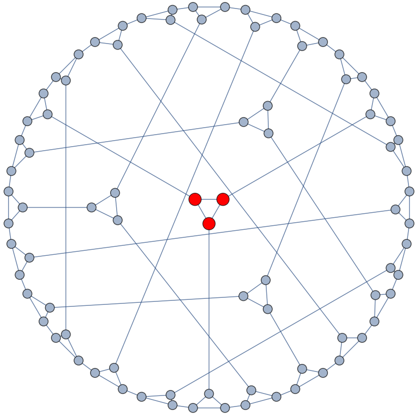
We use the same terminology for paths $u \longrightarrow v$.

Standard implementations:

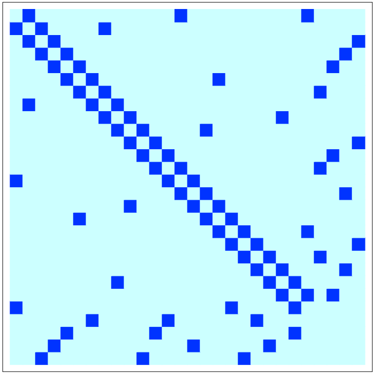
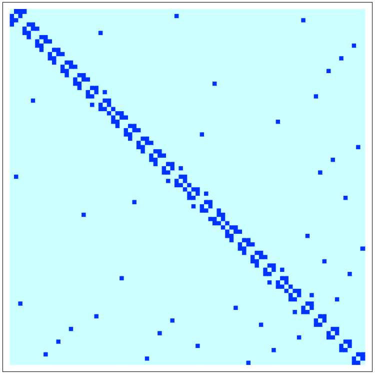
- adjacency lists
- adjacency matrices

Don't automatically knock the latter, packed and sparse arrays can be quite efficient. Also, the matrix perspective opens doors to non-combinatorial methods (spectral graph theory).



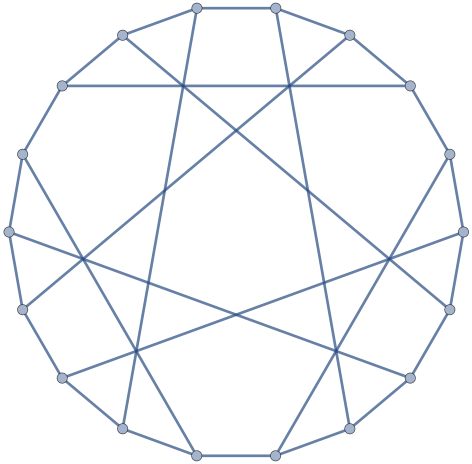


The Coxeter graph, and its triangular version.

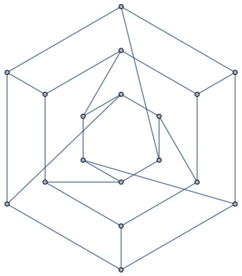
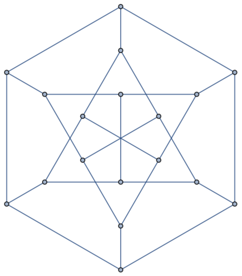
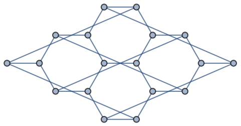
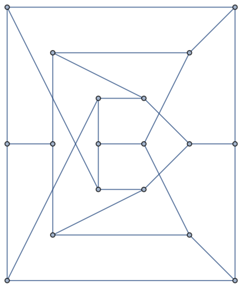
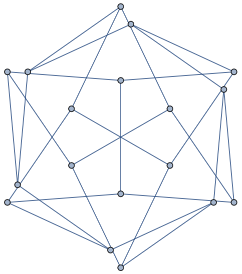
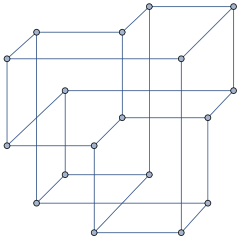



In this presentation, it is utterly impossible to understand what is going on.

But Beware: Pappus



Other Embeddings

Graph Properties

Ever since Felix Klein's Erlanger Progamm it has become popular to study any particular domain math in terms of the structure preserving maps of the domain, and in particular in terms of symmetries (or automorphisms).

For graph theory, this means that one should study properties of graphs that are invariant under graph automorphisms. For example, permuting vertices (and adjusting the edges accordingly) should be irrelevant.

That works nicely in theory, but causes friction for computation: we have to use data structures to represent graphs, and these are definitely not well-behaved: e.g., reordering adjacency lists leaves the graph unchanged, but affects the execution of lots of graph algorithms.

And pictures are just about hopeless computationally.

Reachability

For a graph $G = \langle V, E \rangle$ and a vertex s let the set of points reachable from s be

$$R_s = \{ x \in V \mid \exists \text{ path } s \longrightarrow x \}$$

To compute R_s we can use essentially the same prototype algorithm as for shortest path:

We initialize a set R to $\{s\}$. This time an edge $e = uv$ requires attention if, at some point during the execution of the algorithm, $u \in R$ but $v \notin R$. We relax the edge by adding v to R.

Exercise

Give termination and correctness proofs.

We have to organize the order in which edges uv are handled: usually several edges will require attention, we have to select a specific one.

To this end it is best to place new nodes into a container C (rather than just in R): C holds the frontier, all the vertices that might be the source of an edge requiring attention.

```
$R = C = \{s\}$

while $C \neq \emptyset$
    $u =$ extract from $C$
    if some edge $uv$ requires attention
        add $v$ to $R$, $C$
```

R is a set data structure, and C must support inserts and extracts: a stack or queue will work fine. And the stack can be handled implicitly via recursion.

Depth First Search (DFS)

The basic idea behind DFS. This version just computes reachable points, we will see far more interesting modifications in a while.

```
defun dfs($x : V$)
    put $x$ into $R$
    forall $xy \in E$ do
        if $y \notin R$
        then dfs($y$)
```

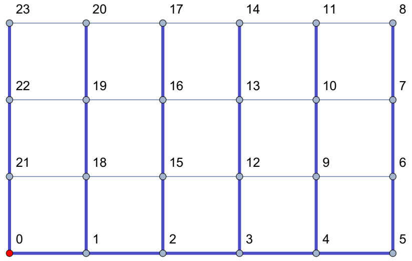
In practice, there usually is a wrapper that calls dfs repeatedly until all vertices have been found.

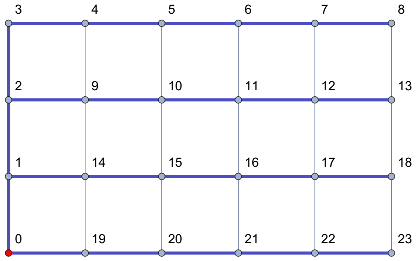
Running time is clearly $O(n + m)$.

Note that the additional space requirement is $O(n)$ for the set R and the recursion stack (which might reach depth n).

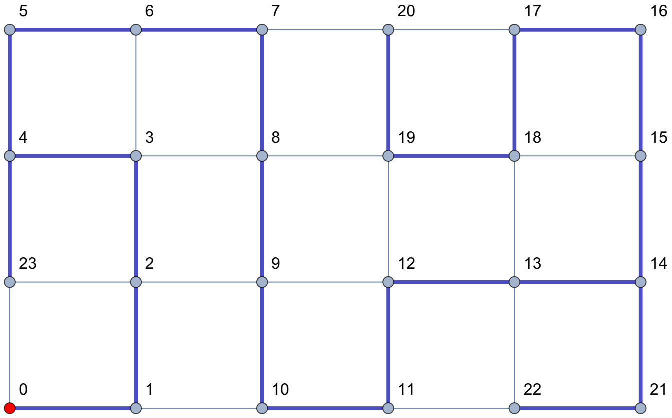
The DFS Tree

The edges requiring and receiving attention in DFS form a tree (or a forest if we run DFS repeatedly on nodes not yet encountered).





Vertices are labeled in order of discovery. So this is east-first versus north-first.



Adjacency lists are randomized.

Recall: Trees and Order

Consider some tree T. Then T induces a partial order on its nodes:

$$x \preceq y \iff x \xrightarrow{T} y$$

This order is usually partial: unless T is degenerate, there are incomparable elements x and y: we have neither $x \preceq y$, nor $y \preceq x$.

Classifying Edges

Suppose we run DFS on vertex s in graph G, producing a tree T. We can classify the edges as follows, based on the partial order induced by the tree:

tree	edges xy that are explored and part of T: y is the successor of x in $\prec$.
forward	edges xy from some node in T to a node below: $x \prec y$.
back	edges xy from some node in T to a node above: $y \prec x$.
cross	edges from some node in T to an incomparable node.

Again: this decomposition of E depends not just on the graph but on the actual adjacency lists: it is not a graph property (something that is invariant under isomorphisms), but a property of the representation. We have to make sure that this does not matter in correctness arguments.

Undirected Case

The classification needs to be adjusted a bit for ugraphs. Think of an undirected edge as two directed edges going back and forth.

Claim: In a ugraph we still have tree edges and back edges, but there are no forward edges and no cross edges.

To see why, suppose uv is a forward edge in T and let $P = u \xrightarrow{T} v$ be a path in T. Then the call to u that started building P did inspect $uv = vu$ and is already done when activity returns to u.

The argument for cross edges is essentially the same.

Write E_t, E_f, E_b and E_c for these edge classes. How do we compute them?

Here is the vanilla version of DFS: we augment the procedure by adding a counter t that determines two timestamps dsc(x) and cmp(x), the discovery (aka DFS number) and completion time. Say, the counter is initialized to 1.

```
defun dfsvanilla($x : V$)
    dsc($x$) = $t$++                  // vertex $x$ discovered
    forall $xy \in E$ do
        if dsc($y$) == 0              // edge $xy$ requires attention
        then dfsvanilla($y$)
    cmp($x$) = $t$++                  // vertex $x$ completed
```

Here we assume that dsc and cmp are initialized to 0 before the first call; so we can use dsc as a replacement for the set R.

Tree and Reachability

Lemma

Run DFS on vertex u and let T be the resulting tree. Then for all v reachable from u there is a tree path $u \xrightarrow{T} v$.

Proof.

Claim: Assume that v has at least one predecessor w such that $u \xrightarrow{T} w$. Then $u \xrightarrow{T} v$.

Suppose otherwise and let w be the predecessor of v with minimum discovery time. During the call to w, v is not found when other children are explored. But then edge wv will be added to T when that edge is explored, contradiction.

Done by induction on the distance from u to v.

□

The timestamp intervals $\mathsf{dur}(x) = [\mathsf{dsc}(x), \mathsf{cmp}(x)]$ describe the duration of the call to x (in logical time). These intervals cannot overlap without containment.

Lemma

Duration intervals are either disjoint or nested.

Proof.

We may safely assume $\mathsf{dsc}(x) < \mathsf{dsc}(y)$. If $\mathsf{dsc}(y) > \mathsf{cmp}(x)$ the intervals are clearly disjoint. If $\mathsf{dsc}(y) < \mathsf{cmp}(x)$ then we must have a tree path $x \xrightarrow{T} y$. But then $\mathsf{dur}(y) \subset \mathsf{dur}(x)$.

□

So nested intervals correspond to nested calls:

$$x \xrightarrow{T} y \iff \mathsf{dur}(y) \subseteq \mathsf{dur}(x)$$

Classifying Edges

Lemma

Let xy be an edge in digraph G. Then

$$\begin{aligned} xy \in E_{\mathrm{t}} \cup E_{\mathrm{f}} &\iff \mathsf{dur}(y) \subset \mathsf{dur}(x) \\ xy \in E_{\mathrm{b}} &\iff \mathsf{dur}(x) \subset \mathsf{dur}(y) \\ xy \in E_{\mathrm{c}} &\iff \mathsf{dur}(y) < \mathsf{dur}(x) \end{aligned}$$

Exercise

Prove the lemma.

Lemma

The part reachable from s has a cycle iff DFS from s produces a back edge.

Proof.

Clearly a back edge indicates the existence of a cycle.

So suppose there is a cycle $c_1, c_2, \dots, c_m$ and a path $s \xrightarrow{T} c_1$ which touches the cycle for the first time. Then a call to c_m is nested inside the call to c_1. But then $c_m c_1$ is a back edge.

$\square$

Considering a vertex where the search touches some part of the graph for the first time is an idea that we will use repeatedly.

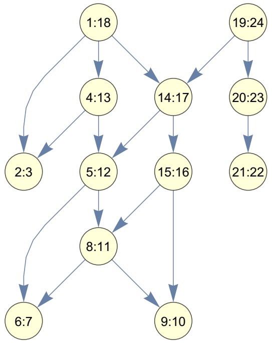
Suppose G is a DAG: a directed acyclic graph. In view of our "look at pictures" philosophy, how about drawing the graph in such a way that all edges go from left to right? Here is brute force.

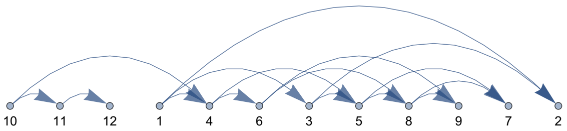
```
$L = \text{nil}$
$S = \{ x \in V \mid x \text{ has indegree } 0 \}$

**while** $S \neq \emptyset$
    extract $x$ from $S$
    append $x$ to $L$
    **forall** $xy \in E$ **do**
        remove $xy$ from graph
        **if** $y$ has indegree 0
        **then** insert $y$ into $S$
    **od**
**return** $L$
```

Better: DFS

- run DFS on the indegree 0 points, compute completion numbers,
- sort the vertices in terms of descending completion numbers.





Decomposing a Ugraph

Let $G = \langle V, E \rangle$ be a ugraph and $U \subseteq V$.

Definition

U is connected if there is a path between any two points in U.
U is a connected component if U is a maximal connected subset.

Proposition

Distinct connected components are disjoint.

So we can decompose a graph into its connected components; the original graph G is the disjoint sum of the corresponding subgraphs.

Finding connected components is easy via Boolean matrices (using multiplication or Warshall's algorithm) or repeated applications of DFS (linear time).

Both the Boolean matrix approach and Warshall's algorithm compute the reflexive transitive closure $E^\star$ of the symmetric edge relation E.

In other words, we compute the finest equivalence relation that extends the edge relation, and represent it by a Boolean matrix.

Given that matrix, one can easily compute the equivalence classes in $O(n^2)$ steps. Likewise, we can compute the condensation graph.

Exercise

Figure out in detail how to do this.

An articulation point of a connected graph G is a vertex v such that $G - v$ is disconnected. A biconnected graph is a connected graph that has no articulation points. A biconnected component (BCC) (or block) of a connected undirected graph is a maximal biconnected subgraph.

In many graph applications (think communication network) articulation points are undesirable.

Exercise

Show that any ugraph other than K_2 is biconnected iff for any two distinct nodes u and v there are two vertex-disjoint paths from u to v.

Two biconnected components can have at most one vertex in common.

Lemma

Biconnected components are edge-disjoint.

Proof. Assume edge uv belongs to two BCCs A and B. Since $A \cup B$ is no longer biconnected, there must be an articulation point w. Suppose a and b are points in two components of $(A \cup B) - w$. Since A and B are biconnected we may assume $a \in A$ and $b \in B$; furthermore, $u \neq w$. But then there is a path from a to u and from u to b, contradiction. $\square$

Let G be some connected ugraph. Given the biconnected components $B_1, B_2, \ldots, B_k$ and articulation points $a_1, a_2, \ldots, a_\ell$ of G we can associate the following graphs with G:

Block graph: vertices are the biconnected components of G. There is an edge between B and B' if they share a vertex (an articulation point).

Block tree: vertices are the biconnected components and articulation points. There is an edge between a and B if $a \in B$.

Clearly it suffices to compute the biconnected components and articulation points to produce these graphs.

```
defun dfsbcc($x : V$)
    dsc($x$) = low($x$) = $t$++
    $c = 0$
    forall $xy \in E$ do
        if dsc($y$) = 0                    // $xy$ tree edge
        then
            par($y$) = $x$
            $c$++
            push $xy$ onto stack
            dfsbcc($y$)
            low($x$) = $\min$(low($x$), low($y$))
            if $x$ is articulation point
            then pop the stack down to $xy$
        else                               // $xy$ back edge
            if $y \neq$ par($x$)
            then
                low($x$) = $\min$(low($x$), dsc($y$))
                if dsc($y$) < dsc($x$) then push $xy$ onto stack
    od
```

Comments

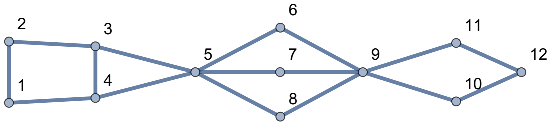
- timestamp t is initialized to 1
- arrays dsc, low and par are initialized to 0 (vertices are $[n]$)
- c is the number of children of x, and
- it is used in the test for x being an articulation point:

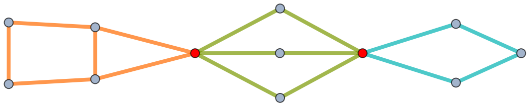
$$(\mathsf{dsc}(x) = 0 \wedge c > 1) \vee 0 < \mathsf{dsc}(x) \leq \mathsf{low}(y)$$

- upon completion of a call dfsbcc(x) there may still be edges of the last BCC on the stack
- low(x) is the smallest DFS number reachable from x (see below), so it refers to a place *highest* in the tree—and so could also be called hi(x).

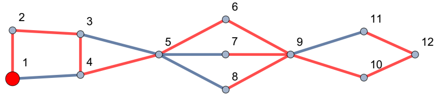
Quoi?

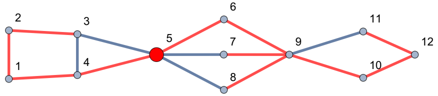
First, and example:





BCCs: $\{1, 2, 3, 4, 5\}$, $\{5, 6, 7, 8, 9\}$, $\{9, 10, 11, 12\}$





x	vis	low	par
1	0	0	0
2	1	0	1
3	2	0	2
4	3	0	3
5	4	2	4
6	5	4	5
7	10	4	9
8	11	4	9
9	6	4	6
10	7	6	9
11	9	6	12
12	8	6	10

x	vis	low	par
1	9	0	4
2	10	0	1
3	11	0	2
4	8	0	5
5	0	0	0
6	1	0	5
7	6	0	9
8	7	0	9
9	2	0	6
10	3	2	9
11	5	2	12
12	4	2	10

Suppose we execute a call dfsbcc(r), r the root of the DFS tree T. Define

$$\lambda(x) = \min\big(\,\mathsf{dsc}(v) \mid x \xrightarrow{T} u \to v\,\big)$$

Thus $\lambda(x)$ indicates how far up the tree we can go with a tree path plus one back edge. For the time being, ignore the edge stack and focus on identifying articulation points.

Since there are no cross edges, it is easy to deal with the root itself.

Claim 1: r is an articulation point iff r has at least two children.

So assume vertex $u \in T$ is not the root.

Claim 2: u is an articulation point iff for some child v of u, $\lambda(v) \geq \mathsf{dsc}(u)$.

Correctness 2

To see why, consider $G - u$. Then any path from v must be contained in T_v. But $r \notin T_v$, so $G - u$ is disconnected.

On the other hand, if $G - u$ is disconnected, then at least one child v of u must have $\lambda(v) \geq \mathsf{dsc}(u)$.

Great, but what does λ have to do with the algorithm?

Lemma

$\mathsf{low}(x) = \lambda(x)$.

Inspecting the code, we can see that

$$\mathsf{low}(x) = \min \begin{cases} \mathsf{dsc}(x) & \\ \mathsf{low}(y) & xy \in E, \\ \mathsf{dsc}(v) & x \xrightarrow{T} u,\ uv \text{ back} \end{cases}$$

And the Edge Stack?

We know that the algorithm correctly identifies articulation points. But how do we output the right edge sets for the biconnected components?

Suppose u is an articulation point, and assume it's not the root of the DFS tree. Let v be the "special" child, and assume that T_v contains no further articulation points.

All the tree edges in T_v are placed into the stack during the call to dfsbcc(v). If a back edge in that subtree is encountered, it also enters the stack.

But then, when the call to dfsbcc(v) terminates, all these edges are removed from the stack and returned.

By induction, this method works throughout T.

Upshot

Theorem

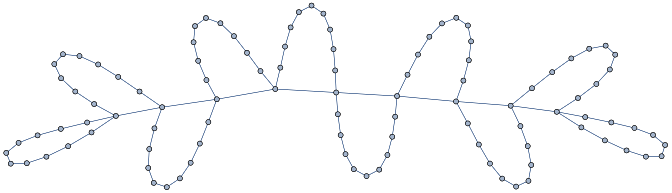
We can compute the articulation points and biconnected components of a ugraph in linear time.

Vanilla DFS is obviously linear, and our modifications only add a linear overhead. For example, every edge enters the stack only once.

By comparison, the brute-force method would be $O(n(n+m))$ and involve a lot of recomputation.

Randomized graphs tend to have one giant BCC, and a number of tiny ones. For example, $n = 100\,000$, $m = 105\,000$ produces $42\,889$ BCCs, one of size $55\,305$, the others of size 1 and 2. Takes 0.44 seconds on my laptop.

With higher edge probability we get down to 0.02 seconds.



The version of this graph with $n = 100\,001$ and loops of length 1000 produces 101 components of size 1000; takes 0.015 seconds.

Exercise

Explain in more detail why the BCC algorithm returns the correct edge sets.

Exercise

Figure out how to compute bridges in a ugraph.

Exercise

Implement the algorithm and explain the apparent speed-up on random graphs when the edge probability increases.

Exercise

Biconnected components are also called 2-connected. What would k-connected mean for $k > 2$? How would you go about computing k-connected components?