Comp 610 Lecture 6: Depth First Search and 2-connected Components

A connected graph G is 2-connected if for every vertex v, G-v is connected.
The blocks of a graph are its maximal 2-connected subgraphs. We noted that two edges lie in a block together precisely if they lie on a cycle. If two two-connected graphs B_1 and B_2 intersect in at least two vertices then their union is 2-connected. We noted that this implied that if two blocks of a graph intersect then they intersect in exactly a cutvertex.

We noted further that for any connected graph G, the bipartite graph with vertex set the blocks and cutvertices of G, where vB is an edge precisely if B is a block containing the cutvertex v, is a tree. It is called the *block tree* of G.

We presented an algorithm to find the blocks, cutvertices and block tree of a connected graph G on n vertices which ran in O(|V|+|E|) time. This is presented as an exercise in Chapter 22 of the text and the relevant definitions and pictures are given there.

Our algorithm worked with a depth first search tree T of G, given by a parent pointer for each node which is nil for the root, and associated values v.d and v.f between 1 and 2n for very vertex v. For every nontree edge uv, with u.f<v.f we have that uv is a back edge from u to its ancestor v. For each vertex s we let T_s be the subtree of T formed by s and its descendants. We note that we can mark which edges of G are tree edges by using the parent pointers. We can then build a list of all the tree edges incident to each vertex and record which is the edge to its parent in total time O(|V|+|E|).

We first construct (the adjacency lists for) an auxiliary graph G′ whose vertices are the edges of G and where an edge from v to its parent in the tree, par(v), is joined precisely to (a) every backedge from v, and (b) any tree edge from v to a child s such that there is a back edge from T_s to an ancestor of v. This allows us to construct the blocks, as the edge sets of the blocks correspond to the connected components of G'. We note that the tree edges in a block form a subtree of T whose root has only one chld. We will name/label each block using the edge from the root of the corresponding subtree to its child in the subtree.

We can find the edges as in (a) by traversing the edge list of each vertex v once. Letting h(s) be the largest integer i such that there is an edge xy with x in T_s and y.f=i, we see that for every node s of the tree with parent v which is neither the root nor a child of the root, (sv,vpar(v)) is an edge of G' as set out in (b), precisely if h(s)>v.f. Furthermore, if v is not the root, v is a cutvertex precisely if it has a child s with h(s) at most v.f. If v is the root, it is a cutvertex of G precisely if it is a cutvertex of T.

Thus if we can determine the h values, not only can we determine the edges of G' and hence the blocks, but we can also determine the cutvertices.

Our algorithm first used a bucket sort to obtain a list of the vertices by increasing f value. It then computed h(s) for the vertices in this order, and stored these values in an array H.

Initially H[v]=v.f for every vertex v. For each non-root v in turn, for each edge vw on the adjacency list for v, if w.f>H[v], we set H[v]=w.f. If v is not a leaf then for each child
s of v, if H[s]>H[v] we set H[v]=H[s]. If H[v]>par(v).f then we add an edge of G' between vpar(v) and par(v)par(par(v) otherwise we do not dd this edge to G' and unless par(v) is the root we add par(v) to the set of cutvertices of G. If v is the root we add it to the set of cutvertices precisely if it has two children.

We next compute the components of G', and then the blocks of G (whose edge sets are the components of G'). We next construct an array INBLOCK, indexed by the edges of G, where INBLOCK[e] is the name of the block containing e.

We consider the vertices in the reverse order (so in decreasing v.f value). For each node v, if vpar(v) is joined to par(v)par(par(v)) by an edge of G' (or equivalently if H[v]>par(v).f, we assign INBLOCK[vpar(v)] to be INBLOCK[par(v)par(par(v))]. Otherwise we assign INBLOCK[vpar(v)] to be vpar(v). We traverse the edges once and for each backedge from v to an ancestor u we set INBLOCK[vu]=INBLOCK[vpar(v)].

We now consider each cutvertex, x. We traverse the tree edges incident to x and make x incident in the block tree to every block whose name is INBLOCK[e] for some tree edge e incident to x.

Having completed our discussion of this linear time algorithm, we turned to tri-connected components of a graph. A two cut in a graph is a pair of vertices (a,b) such that G-a-b is disconnected A graph is 3-connected if for every pair (a,b) of its vertices, G-a-b is connected. A 2-cut tree for a 2-connected graph G, has vertices which are either 2-cut nodes labelled by 2-cuts of G or graph nodes labelled by graphs, and whose edges join cut nodes to graph nodes. We construct such a tree as follows. We take any 2-cut (a,b), construct for each component U of G-a-b, a 2-cut tree T_U of the graph G_U formed from the subgraph of G induced by U+a+b by adding the edge ab, and then combine these trees into one new tree by adding a new 2-cut node labelled (a,b) and, for each component U of G-a-b, adding an edge from it unique subgraph node of T_U whose label contains ab.

A 2-cut (a,b) is *strong* if a and b are joined by three paths which share no vertices other than a and b (this may just be 3 copies of the edge ab). We can build a 2-cut tree (the strong 2-cut tree) by using all the strong 2-cuts of G and no others. This is unique and breaks G up into three connected subgraphs and cycles. If we continue

decomposing on two cuts until none exist then the subgraphs are labelled by the 3-connected labels of graph nodes of the strong 2-cut tree and (|C|-1) triangles for each cycle C labeling a graph node of the strong 2-cut tree. However, the tree need not be unique as we can use cuts corresponding to any triangulation of C for each such C.

Next class, we will discuss an O(|V|+|E|) time algorithm to find a 2-cut tree for a 2-conncted graph.