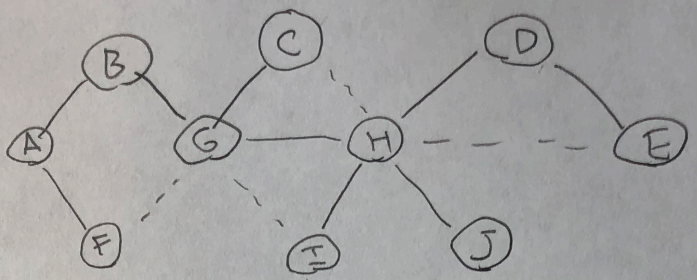
CSE 417 HW2

Zachary McNulty (zmcnulty, ID: 1636402)

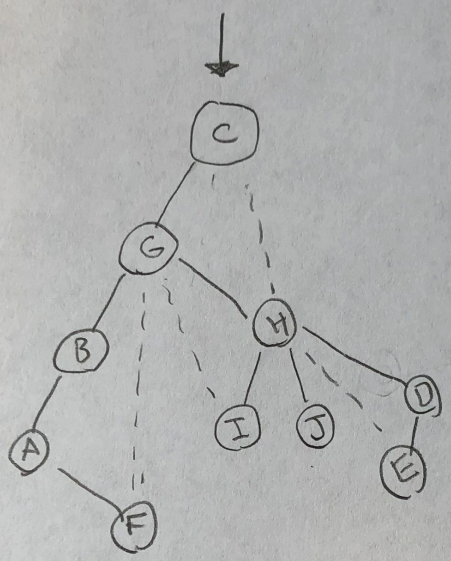
January 2019

(1)

abcd



vertex	dfs #	low
A	4	2
B	3	2
C	1	1
D	7	6
E	8	6
F	5	2
G	2	1
H	6	1
I	9	2
J	10	10



(e) articulation points G, H

(f) edge order

(C,G)(G,B)(B,A)(A,F)(F,G)(G,H)(H,C)(H,D)(D,E)
(E,H)(H,I)(I,G)(H,J)

1 Problem 2: Biconnected Components

component 1: $\{(A, B), (B, G), (G, F), (F, A)\}$
component 2: $\{(G, C), (C, H), (H, I), (I, G), (G, H)\}$
component 3: $\{(H, D), (D, E), (E, H)\}$
component 4: $\{(H, J)\}$

2 Problem 3: Biconnected Algorithm

Run the articulation point algorithm with the following modifications. Firstly, keep track of an ordered list of edges that have been traversed at each step in the algorithm (i.e. the order we find in problem 1f). If the articulation algorithm reaches a point where it determines that vertex v is an articulation point, traverse in reverse the edge list we have been building, removing the edges and adding them to a set, until we reach the first edge (v, x) where x is an arbitrary other vertex. This edge set generated will be a biconnected component. Continue the articulation algorithm. When the algorithm is complete, add the edges remaining in the edge list to a set to form a final biconnected component.

```
Global initialization: dfscounter = 0; v.dfs# = -1 for all v;
edgeList = new List(); biconnectedComponents = new List()

    DFS(v):
        v.dfs# = dfscounter++
        v.low = v.dfs# // initialization
        for each edge {v,x}
                    edgeList.add( {v,x}) // add edge to end of edge list
                if (x.dfs# == -1) // x is undiscovered
                    edgeList.add( {v,x}) // add edge to end of edge list
                    DFS(x)
                    v.low = min(v.low, x.low)
                    if (x.low >= v.dfs#)
                            print "v is art. pt., separating x"
                            biComponent = new Set()
                            next = edgeList.pop() // remove last element
                            biComponent.add(next)
                            while next not start in v
                                    next = edgeList.pop()
                                    biComponent.add(next)
                    biconnectedComponents.add(biComponent)
                else if (x is not v's parent)
                    edgeList.add( {v,x}) // add edge to end of edge list
                    v.low = min(v.low, x.dfs#)
```

Trace of Algorithm on Problem 1

Call stack: C
EdgeList: []

Current vertex: C
Call stack: $G, H, \not{C}$
EdgeList: []

Current vertex: G
Call stack: $BF\not{G}, H, \not{C}$
EdgeList: $[(C, G)]$

Current vertex: B
Call stack: $A\not{B}F\not{G}, H, \not{C}$
EdgeList: $[(C, G), (G, B)]$

Current vertex: A
Call stack: $F\not{A}\not{B}\not{G}, H, \not{C}$
EdgeList: $[(C, G), (G, B), (B, A)]$

Current vertex: F
Call stack: $\not{F}\not{A}\not{B}\not{G}, H, \not{C}$
EdgeList: $[(C, G), (G, B), (B, A), (A, F)]$

Call stack: $\not{F}\not{A}\not{B}\not{G}, H, \not{C}$
EdgeList: $[(C, G), (G, B), (B, A), (A, F), (F, G)]$

vertex G is determined to be articulation point

add biconnected component: $\{(F, G), (A, F), (B, A), (G, B)\}$
Current vertex: H
Call stack: $DIJ\not{F}\not{A}\not{B}\not{G}, \not{H}, \not{C}$
EdgeList: $[(C, G), (G, H)]$

Current vertex: D
Call stack: $E\not{D}IJ\not{F}\not{A}\not{B}\not{G}, \not{H}, \not{C}$
EdgeList: $[(C, G), (G, H), (H, C), (H, D)]$

Current vertex: E
Call stack: $\not{E}\not{D}IJ\not{F}\not{A}\not{B}\not{G}, \not{H}, \not{C}$
EdgeList: $[(C, G), (G, H), (H, C), (H, D), (D, E), (E, H)]$

vertex H determined to be articulation point
add biconnected component: $\{(H, D), (D, E), (E, H)\}$

Current vertex: I
Call stack: $\not{E}\not{D}\not{I}\not{J}\not{F}\not{A}\not{B}\not{G}, \not{H}, \not{C}$
EdgeList: $[(C, G), (G, H), (H, C), (H, I)]$

Current vertex: J
Call stack: $\not{E}\not{D}\not{I}\not{J}\not{F}\not{A}\not{B}\not{G}, \not{H}, \not{C}$
EdgeList: $[(C, G), (G, H), (H, C), (H, I), (I, G), (H, J)]$

vertex H determined to be articulation point separating J
add biconnected component: $\{(H, J)\}$

EdgeList: $[(C, G), (G, H), (H, C), (H, I), (I, G)]$

algorithm complete; add remaining edges to component
add biconnected component: $\{(C, G), (G, H), (H, C), (H, I), (I, G)\}$

Algorithm Correctness
In the DFS tree of the given graph, any biconnected component must be a subtree beneath an articulation point v where all non-tree edges only go between vertices of this subtree or v. If there was an edge (x, y) with x in the subtree going outside this set, then v would not be an articulation point as deleting it still leaves the subtree connected to the rest of the graph via y. Furthermore, if there was some vertex w that was not an articulation point between v and the biconnected component, then the component would not be biconnected as the edges connecting w to the component could be added without adding an articulation point, and thus the component is not maximal.

By the nature of DFS, all edges in this subtree (and non-tree edges as well) will be explored, without interuption, before the articulation point v is recursively returned to. At this point, we can find all the edges in this part of the graph simply be reading all the previous edges visited since leaving the articulation point (the last edge to start with v in the edge list). As proved above, this will be a biconnected component. Furthermore, all the remaining edges above these articulation points must also form a biconnected component as we have essentially removed all articulation points by removing the edges split by them into the other components.

Algorithm Complexity
The articulation point algorithm runs in $O(n+m)$ time, where $n = |V|, m = |E|$. Adding edges to the end of our edge list is an $O(1)$ operation with an appropriately chosen data structure. Since we add each edge only once, this process overall is $O(m)$. Similarly, when we traverse the edge list after finding an articulation point, we only visit each edge once as we remove the edge from the edge list after reaching it. Thus we traverse the edge set only once, and perform m removals from the edge list at $O(1)$, both giving $O(m)$. Therefore, our overall algorithm is $O(n + m) + 3 * O(m) = O(n + m)$ which is still graph linear.

Proof articulation point algorithm is O(n+m): Firstly, note that DFS is O(n+m). In DFS, we loop over every vertex exactly once after removing it from the stack. Since we loop over the edges for every vertex, each edge will be traversed exactly twice, once for each endpoint. For each edge, we perform only constant time operations (i.e. checking if a vertex is undiscovered or not, marking a vertex as discovered, or adding to the stack). Thus, in all there are O(V) operations for removing items from the stack and O(2E) operations for traversing each edge, giving a total of O(V + E). We could also see this just by comparison with the BFS algorithm. BFS is O(V+E) and the only difference is BFS uses a queue rather than a stack, but both of these have constant inserts and removes (from front). The articulation point algorithm is simply DFS which stores extra auxillary information on the dfs# and LOW of a vertex, and performs a few extra constant time checks, but all these operations are O(1), so it does not change the running time of the algorithm. Thus, the articulation point algorithm is O(m + n).

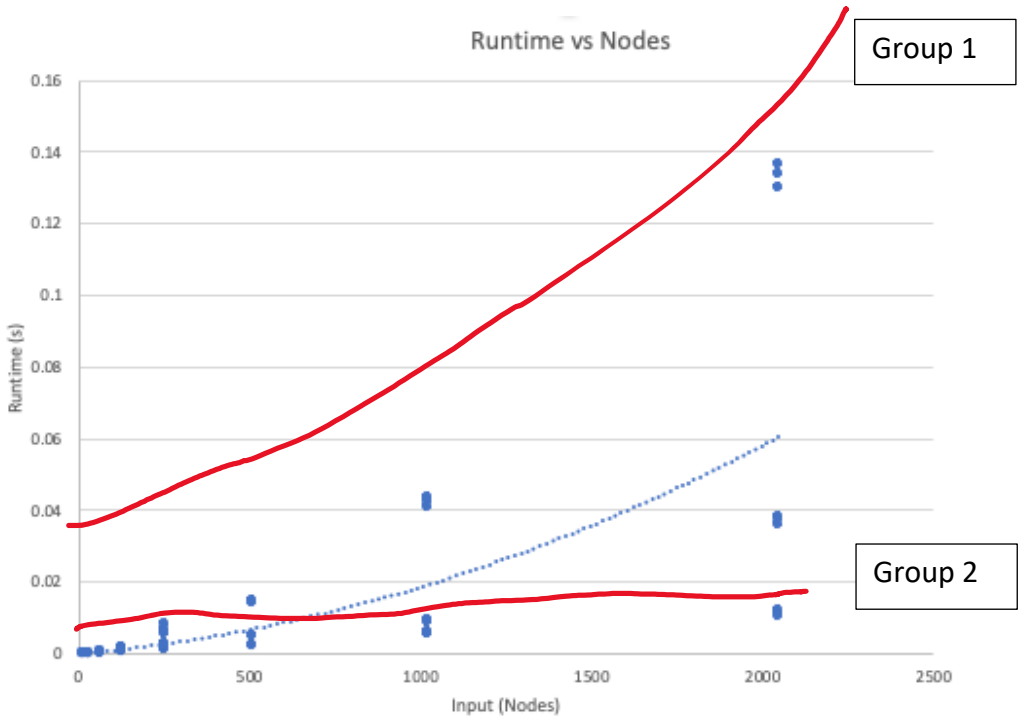
Zachary McNulty (zmcnulty, ID: 1636402)

CSE 417 HW 2 Report

I ran the algorithm I developed for finding the biconnected components on all of the provided sample graphs (with the help of a quick bash script). I used my 2015 MacBook air for the entire project. Below is some basic information on the processor and memory of the computer I ran the testing on.

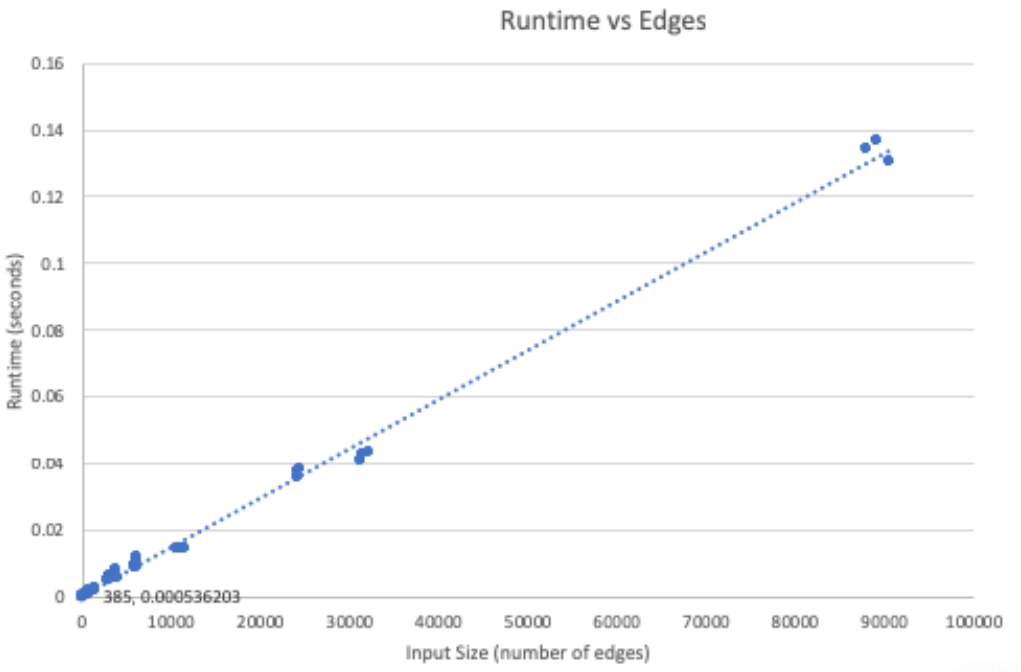
Processor: 1.6 GHz Intel Core i5
Memory: 8 GB 1600 MHz DDR3

I arbitrarily chose zero to be the start vertex for my algorithm every time, although I tested the algorithm using other start vertices as well, including articulation points, and it was still successful. I excluded the time it took to read the input files and generate the data structure to store the edge information in order to get a more specific picture of the performance of the algorithm itself. Below is a short summary of my results.



In this first graph, we see the input size (measured in the number of vertices in the graph) plotted against the runtime of the algorithm. Under this measure, the algorithm appears approximately polynomial order with respect to the input size, i.e. $O(|V|^2)$. However, it is clear the given polynomial fit poorly captures the data: there is a large amount of error in prediction. On closer inspection, there are two distinct groups. The lower data points are from sparse graphs where $|E| << |V|^2$ and the upper data points come from more complete graphs. The runtimes of the former group appear to have an almost linear relationship with the input size, while the latter group's runtimes clearly exhibit a superlinear relationship. This suggests that the number of

edges may play a significant role in the runtime behavior of the algorithm. Below, we explore this suspicion.



This graph plots the runtime versus the input size, as measured by the number of edges in the graph. Here, unlike before, we see that the linear relationship between edges and runtime captures the data very nicely. Unlike before, we do not see any distinct subgroups in the data, and the relation holds for large input sizes. This suggests my algorithm may be of the order O(|E|).

Both graphs suggest upperbounds consistent with our claim that the algorithm is theoretically graph linear— O(|E| + |V|) --- as discussed in the section justifying the algorithm. For sparse graphs, $|E| \sim |V|$, giving a bound of O(|V|) which is consistent with the linear like behavior of the lower group in the first graph, and for non-sparse graphs $|E| \sim |V|^2$, which is consistent with the polynomial behavior of the upper group in the first graph. As the algorithm primarily deals with connected graphs, it is usually the case that $|E| > |V|$, so the relationship between the algorithms runtime vs edge count is likely more important as the edges dominate the vertices. Again, the fact that we get two separate groups in the first graph suggests that the vertex count does not tell the full story. As a result, I feel the relationship between edge count and runtime is more informative, as the number of vertices alone is not enough information to predict the long-term behavior of the algorithm.

I feel the first graph supports the big-O dogma. If we were to only observe small input sizes, the vertex count may appear a reasonable way to approximate the runtime of the algorithm. However, as the input size grows, it becomes clear there are other factors at play that dominant the runtime relationship: the small differences in runtime at low inputs due to the sparsity of the graphs quickly diverge as the input size grows and the linear component due to the number of vertices fades away. This supports the big-O philosophy that, in the long-run, lower order functions quickly become negligible and that the relationship between runtime and the input size is truly appropriately defined by the upper level terms.

3 Problem 5

Qa1: Prove every edge is part of exactly one biconnected component.

Proof: First note that any edge is in at least one biconnected component. If the edge (u, v) is not part of any biconnected component, then the edge itself (u, v) forms a biconnected component as removing any vertex from this single edge leaves a connected induced subgraph. If this component was not maximal, it would imply (u, v) is part of some larger biconnected component, a contradiction.

Suppose there exists a graph G such that there exists an edge e_i that is part of two separate biconnected components, G_1 and G_2, which are induced subgraphs of G. Without loss of generality, suppose we delete a vertex from G_1. Then, either u or v remains undeleted. Thus, as G_1 is biconnected and $(u, v) \in G_1$, there is still a path to either u or v (the undeleted one) from any vertex $v_1 \in G_1$. As G_2 is biconnected, and hence connected, and $u, v \in G_2$, then there is a path from u and v to any $v_2 \in G_2$. Thus, any $v_1 \in G_1$ and $v_2 \in G_2$ remain connected, and as G_1 and G_2 are biconnected, any pair of vertices in G_1 or any pair in G_2 remain connected. Thus, $G_1 \cup G_2$ is connected. Identical logic can be used to show connectivity remains if we delete a vertex from G_2. Thus, $G_1 \cup G_2$ is biconnected, a contradiction to the fact that G_1 and G_2 were two separate biconnected components. Thus, no such edge can exist and all edges are part of at most 1 biconnected component.

Thus, all edges are part of exactly one biconnected component. **Q.E.D.**

Qa2: Prove two distinct edges lie on a common simple cycle if and only if they are in the same biconnected component.

Proof: If two edges distinct $e_1 = (x, y), e_2 = (u, v)$ lie on a common simple cycle, then there are at least 2 non-intersecting paths between any pair of x, y, u, v, i.e. clockwise or counterclockwise around the cycle. Removing a vertex v (i.e. an articulation point) interferes with at most one of these paths as v is either in the clockwise or counterclockwise (or neither) path of the cycle for any given vertex in the cycle. Thus, x, y, u, v remain connected despite the removal of a vertex. Any induced subgraph including x, y, u, v then includes both e_1, e_2, and as connectivity is maintained this is a biconnected component. Thus, e_1, e_2 lie in the same connected component.

If two edges e_1, e_2 are part of the same biconnected component, there exists at least 2 non-intersecting paths including e_1, e_2. If there was only one path, any vertex on this path could be removed to separate e_1, e_2 contradicting the

fact that they are in the same biconnected component. If all paths including e_1, e_2 intersect, removing this intersection vertex will separate e_1, e_2 again contradicting the fact that they are in the same biconnected component. Thus, there exists at least 2 non-intersecting paths including e_1, e_2, and this is naturally a cycle. **Q.E.D.**

Qf: Extend your algorithm to handle unconnected graphs

Simply keep track of all the vertices visited during the DFS part of the algorithm. Once the algorithm terminates, choose a new start vertex that has not already been seen so far and repeat the algorithm. Repeat until all vertices have been visited. As each connected component is a connected graph, the algorithm needs no adjustment to run on each individual component.