Traversal of Trees

CS - 303 Programming Report #4

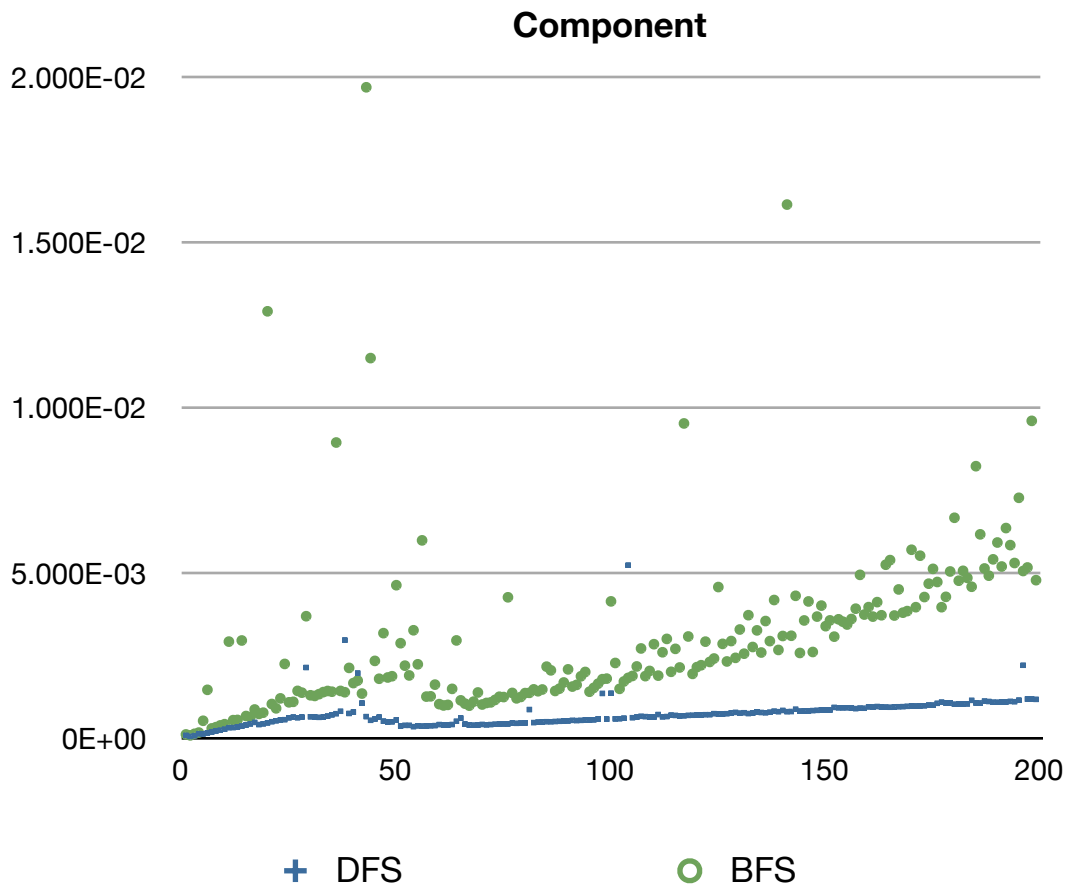
Evan Travers

Purpose:

To investigate the effects of three variables on breadth-first searches and depth-first searches, these three being how many connected-components are in the graph, the depth of each component, and the branching factor. We were given a program that generated graphs given these three inputs, and timing classes. I wrote a depth-first and a breadth first search and a driver class. There were some difficulties, as it is extremely easy to generate graphs with massive amounts of nodes for very low values for component, depth, and branching. To fight this, all data presented here uses 2 for every input except the one being incremented, unless otherwise noted. Also, when I ran the programs, I forced Java to give the program up to 2GBs of RAM.

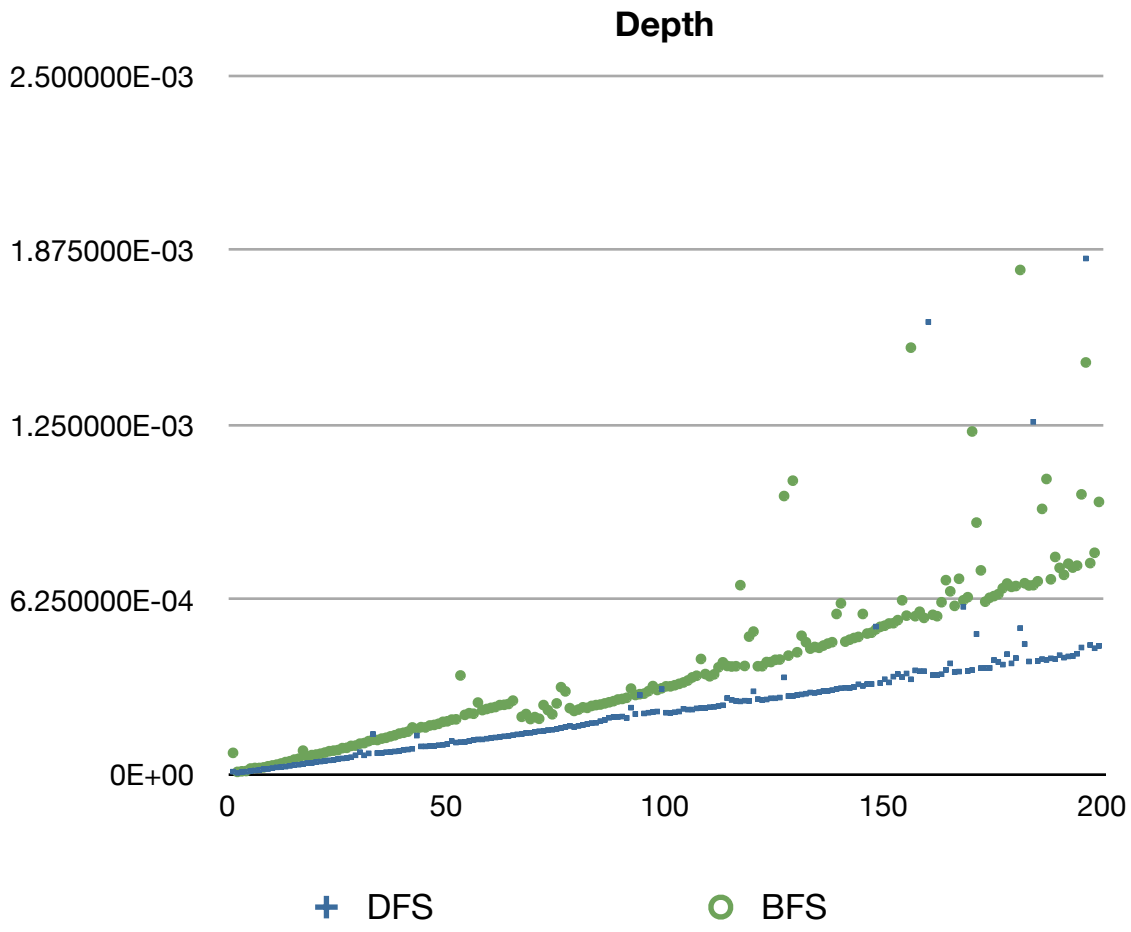
The Setup:

After coding the BFS and the DFS, I began generating data for graphs. I discovered that despite my giving Java half of my RAM (2GB,) I could quickly fill up the heap with very low values. I found that I couldn't get past 17 in depth while the branching was equal to 3. To combat this, I set all values other than the one I incremented to 2. I was able to get to 200 on every input under this setting. The speed at which the DFS and BFS run is directly related to the number of nodes in the graph, so the way the number of components, depth, and branching factor affects the number of nodes is very important.



Changing the numbers of components is linear. This is a logical assumption, but I was able to prove it by running a quickly-built loop that incremented a graph of 1,2,2 (size 5) and incremented the number of components. They went up by five like clockwork. Each component in the graph contains the same

number of nodes as every other component. Therefore, the number of nodes in a graph is simply the number of nodes in one graph, times the number of components. For extremely large numbers, the effect on the time can be greater than linear, but that is more an effect of managing so very many large arrays.

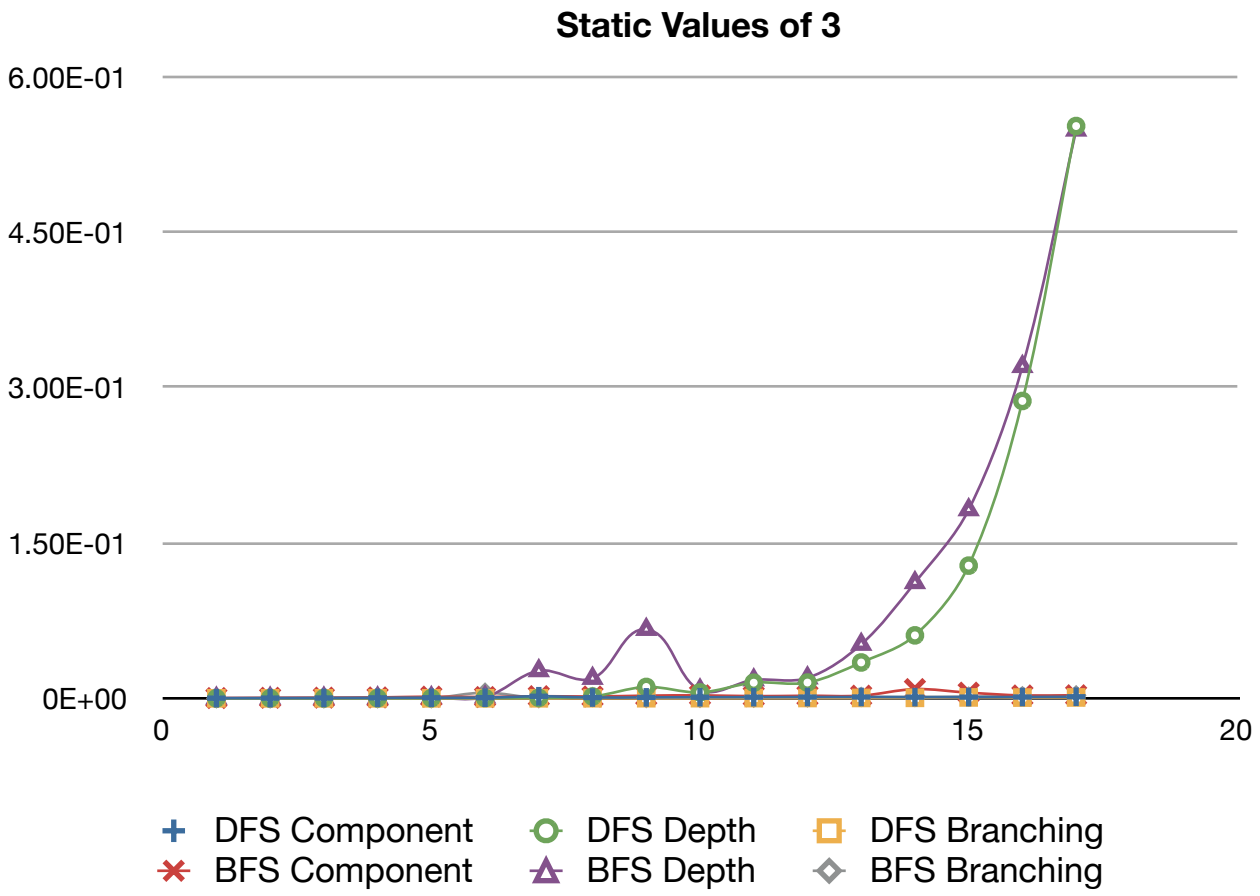


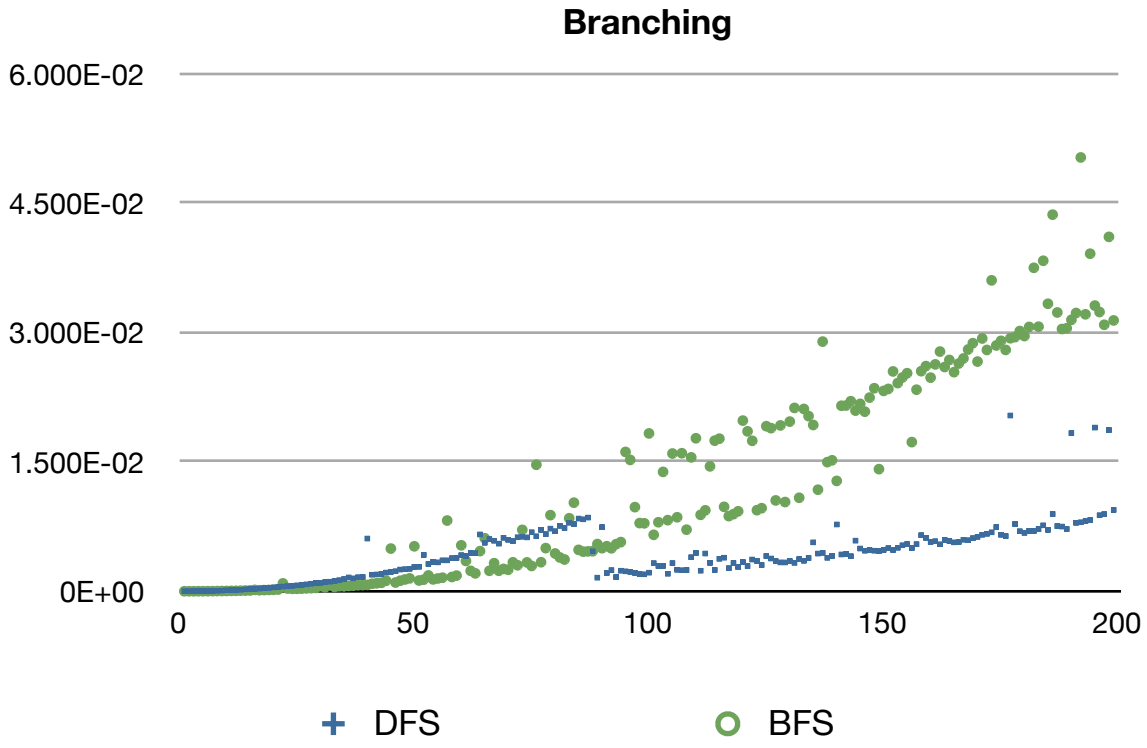
Depth is closely related to branching factor in complexity, and also owes much to branching to its effect on the number of nodes. While not linear like components, it's power isn't easily seen in this graph because of the ridiculously low branching factor I was forced to use by memory space. If you set the branching factor any higher than 2, you get more interesting results. (Tested values are 15.)

Static Values (All other values)	Component:15	Depth:15	Branching:15
2	75	62	452
3	330	294906	9498

Depth is clearly an exponential factor on node number, but it is heavily dependent on branching factor. When I set the branching factor to 4 I couldn't even make the graph without running out of memory.

Because of the massive number of nodes it creates, depth can have a massive effect on the time it takes to traverse the graph as can be clearly seen from the graph below.





Branching is also an exponential factor, but it's affect comes mostly when combined with higher values of depth, as seen in the graph in the last section. In most of my graphs, branching takes the most time because I had to use a value of two in most of my graphs for depth and therefore branching creates the most nodes. At 199, where most of my graphs end, and a static value of 2, the nodes created by the graph constructor for each value are:

Nodes Created

Component	Depth	Branching
995	798	79204

Therefore, branching will have the greatest time value while the branching factor is two. It will be number two in order of contributing to most nodes if it is set to any higher than two... depth catches up extremely quickly as before shown.

Conclusions:

Of the three variables, using reasonable values, the order in effect on the performance of DFS and BFS traversals of graphs from least to greatest are: component, branching, and depth. The number of components is a linear, if not constant factor. Branching is important, but is more an accelerant for the most crucial factor, that of depth. The depth of a graph for any value of branching greater than 2 is the most defining feature of a graph.

Appendix A: Pseudocode

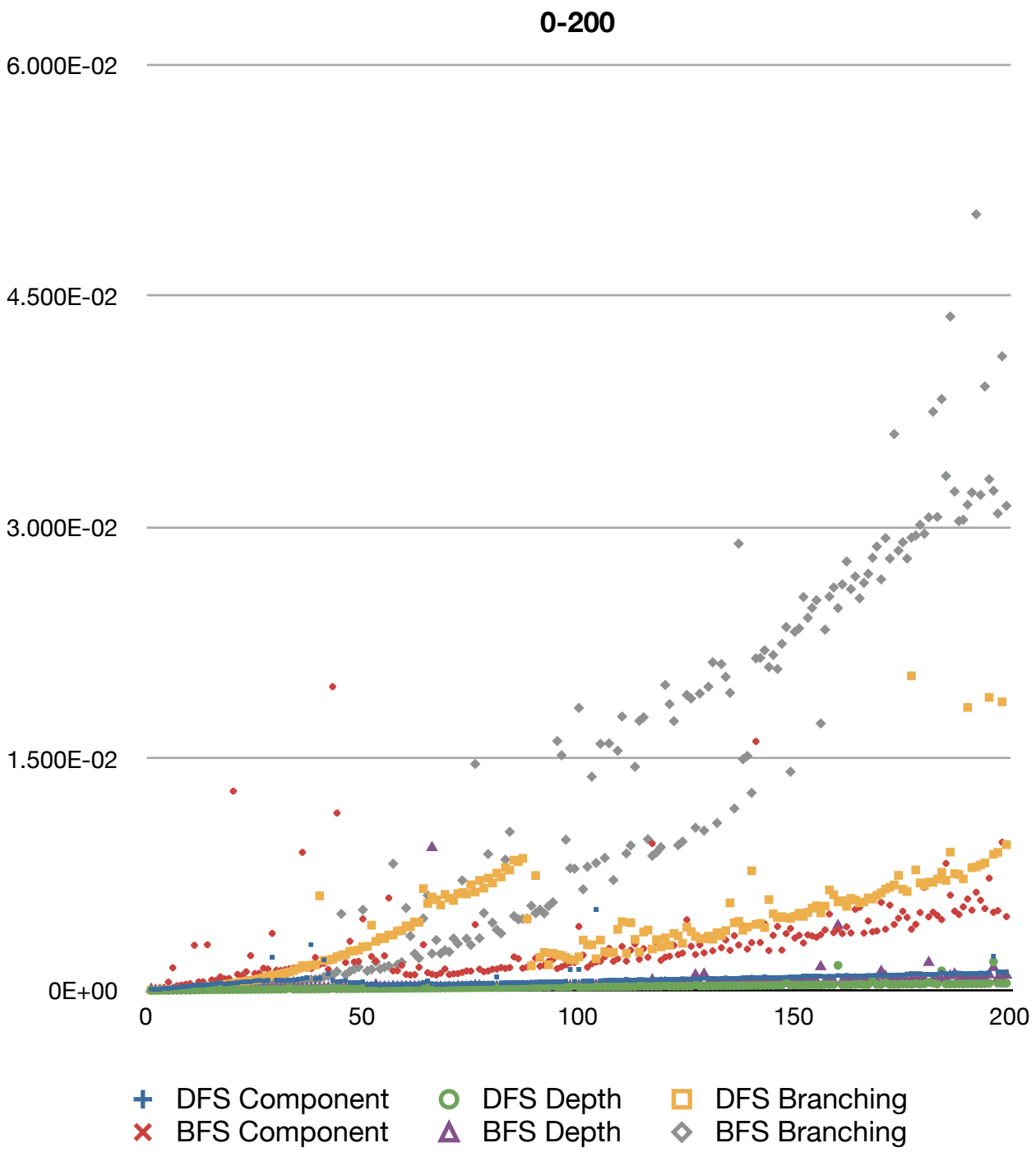
```
bfs(node)
        // this is the return value
        create empty list of nodes

        // this queue could be considered local
        create empty queue of nodes

        // add the current node to the return value
        // at the "leaves," the size of this list will be 1,
        // it will get bigger as the recursion returns
        add node to list

        // make sure you don't visit this node ever again
        set node as visited

        // set up the bfs
        add node's neighbors to queue
        while queue is not empty
                if front element is visited
                        // if we've seen it before
                        remove from queue
                else
                        // recursively build the return value on return
                        add result of bfs(queue pop) to list
        return list

dfs(node)
        // process this one
        set node visited

        // set up the dfs
        create empty stack of neighbors
        while neighbors has items
                if top of the stack is visited

                        // throw it away
                        pop it off
                else

                        // run dfs on the first item
                        dfs(top of the stack)

        // this is to set up the bfs
        return last node
```

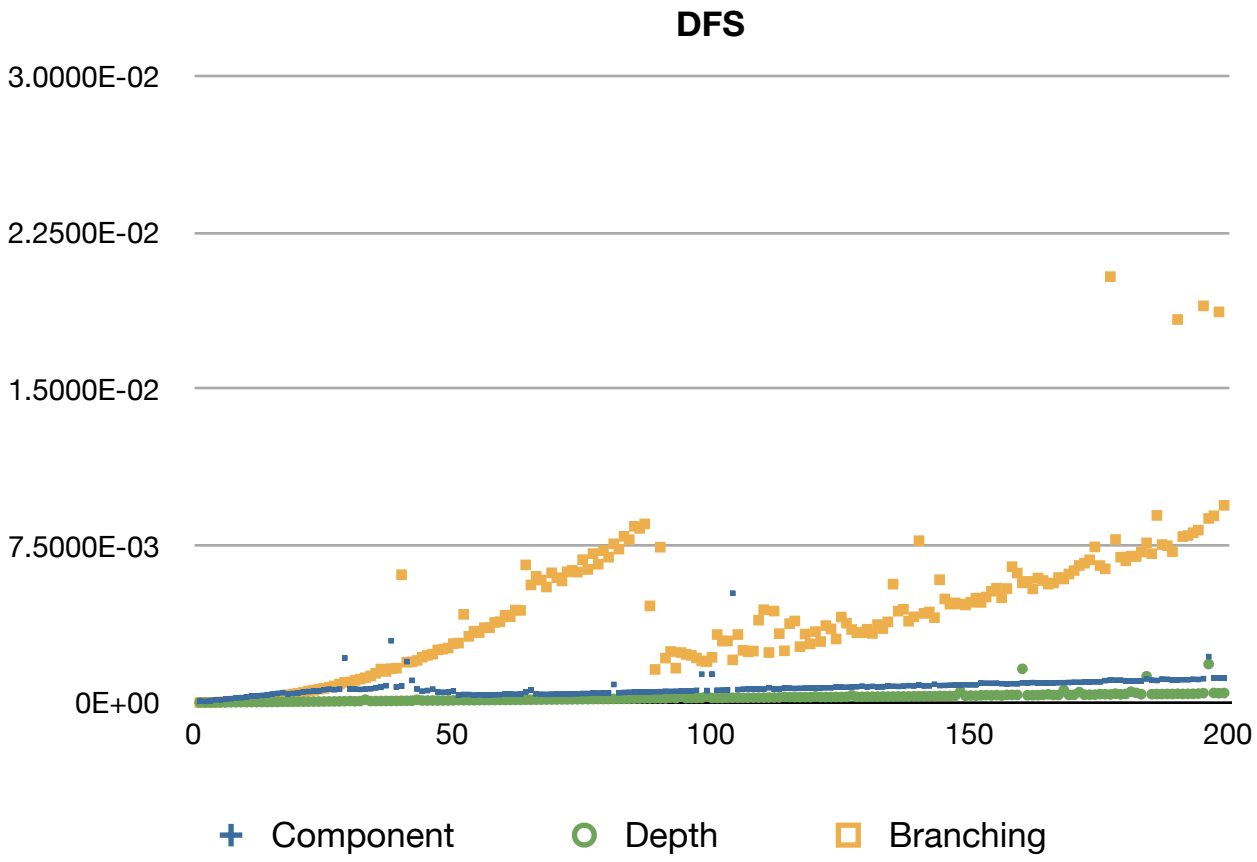
Appendix B: Code

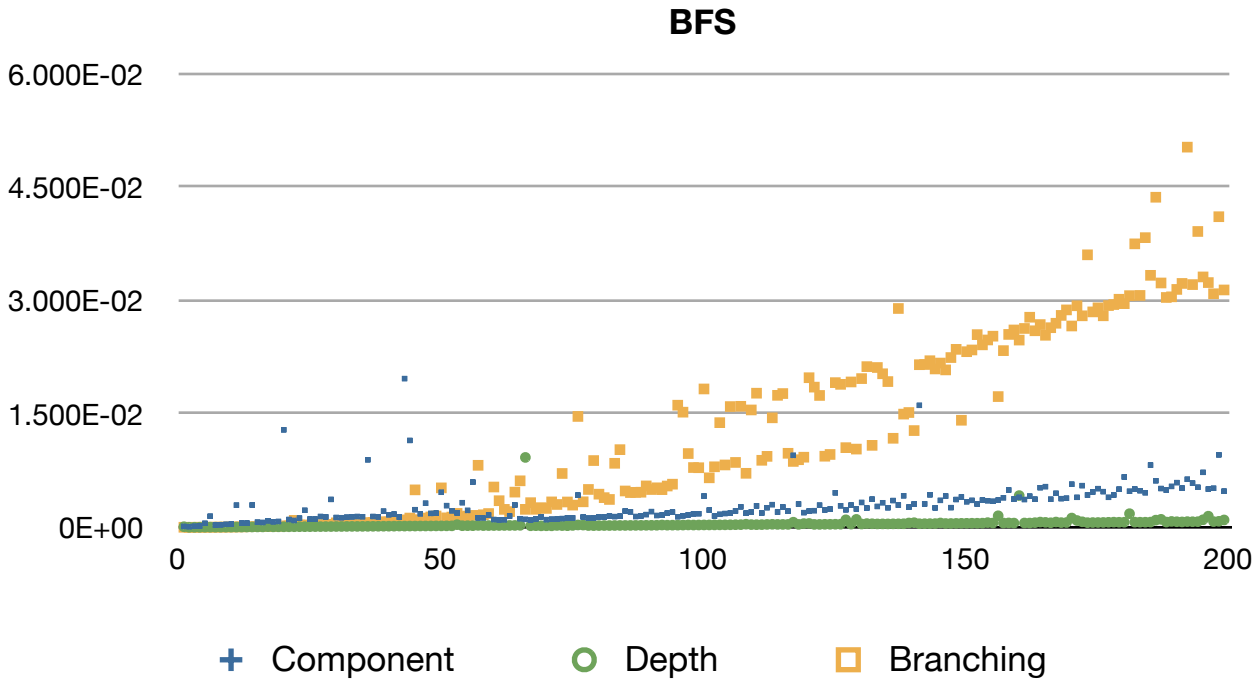
```
public Vector<Integer> bfs() {
     return bfs(0);
    }

public Vector<Integer> bfs(int node) {
     // you've encountered a wild node!
              // increment the count
              Vector<Integer> nodeList = new Vector<Integer>();
              nodeList.add(node);
              // mark node as read
              setVisited(node);
              // find its friends that you haven't met and repeat
              // get the neighbors
              Vector<Integer> neighbors = new Vector<Integer>();
              getNeighbors(node,neighbors);
              while (!neighbors.isEmpty()) {
                     if (getVisited(neighbors.firstElement())) {
                            neighbors.remove(0);
                     }
                     else {
                            nodeList.addAll(bfs(neighbors.remove(0)));
                     }
              }
              return nodeList;
    }

public int dfs() {
     return dfs(0);
    }

public int dfs(int node) {
              // DFS goes through the graph, marks them as read,
              // and returns a node in that component.
              // you've encountered a wild node!
              // increment the count
              // mark node as read
              setVisited(node);
              // find its friends that you haven't met and repeat
              // get the neighbors
              Stack<Integer> neighbors = new Stack<Integer>();
              getNeighbors(node,neighbors);
              while (!neighbors.empty()) {
                     if (getVisited(neighbors.peek())) {
                            neighbors.pop();
                     }
                     else {
                            dfs(neighbors.pop());
                     }
              }
              return node;
    }


public static Vector<Integer> numComponents(Graph g, int timerCount, Timing t, Boolean
timerEnable) {
              Vector<Integer> return1 = new Vector<Integer>();
              if (timerEnable) {
                     t.startRun(timerCount);
              }
              for (int i=0;i<g.graphLength();i++) {
                     if (!g.getVisited(i)) {
                            return1.add(g.dfs(i));
```

```
                        }
                }
                if (timerEnable) {
                        t.stopRun();
                }
                return return1;
        }

        public static void evaluate(Graph g, int timerCount, Timing t, Boolean
timerEnable) {
                String output = "";
                Vector<Integer> numComp = numComponents(g,timerCount,t,false);
                output += "Number of components: " + numComp.size() + "";
                g.setAllUnvisited();
                if (timerEnable) {
                        t.startRun(timerCount);
                }
                for (int i=0;i<numComp.size();i++) {
                        int node = numComp.get(i);
                        if (!g.getVisited(node)) {
                                output += "\nNodes of component " + i + " are: " +  g.bfs
(node);
                        }
                }
                if (timerEnable) {
                        t.stopRun();
                }
                System.out.println(output);
        }
```

Appendix C: Graphs




DFS
3.0000E-02
2.2500E-02
1.5000E-02
7.5000E-03
0E+00
0
50
100
150
200
Component
Depth
Branching


BFS
6.000E-02
4.500E-02
3.000E-02
1.500E-02
0E+00
0
50
100
150
200
Component
Depth
Branching