Program Structures and Algorithms
Spring 2023(SEC - 8)

NAME: Dinesh Singh Shekhawat
NUID: 002776484
TASK:

1. Implementing height-weighted Quick Union with Path Compression algorithm and check that the unit tests for this class all work.
2. Generate random pairs of integers between 0 and n-1, calling connected() to determine if they are connected and union() if not. Loop until all sites are connected then print the number of connections generated.
3. Determine the relationship between the number of objects (n) and the number of pairs (m).

RELATIONSHIP CONCLUSION:

1. The value of ‘n’ represents the number of nodes in a graph, while ‘m’ represents the number of pairs generated for connecting all nodes.
2. Union-Find without Path Compression:
    a. In this algorithm, we simply perform a union operation by linking the root of one set to the root of the other set. This operation takes O(1) time, but the time complexity for finding the root of a set can be O(n) in the worst case, where n is the number of elements in the set. To see why, consider a case where each node in the set is connected to a single node, forming a linked list. In this scenario, finding the root of a set would require us to traverse the entire linked list, resulting in a time complexity of O(n). On average, though, the time complexity is O(logn) due to the fact that each node only has one parent, and so we only need to traverse log n edges to find the root.
3. Union-Find with Path Compression:
    a. This algorithm modifies the standard union-find algorithm by compressing the path during the find operation. In other words, while finding the root of a set, we also make the nodes along the path point directly to the root, rather than their parent. This operation results in a more flattened tree structure, which can reduce the time complexity of future find operations. The time complexity of Union-Find with Path Compression is not always O(n), it is usually considered to be O(m * α(n)), where m is the number of operations (unions and finds) performed and α(n) is the inverse Ackermann function, which grows very slowly and can be considered to be almost constant. **This results in a practically linear time complexity for Union-Find with Path Compression, making it an efficient algorithm for solving dynamic connectivity problems.**

For Union-Find without Path Compression

$$m = O(n\,log(n))$$

For Union-Find with Path Compression

$$m = O(n)$$

EVIDENCE TO SUPPORT THAT CONCLUSION

Source Code

UF_HWQUPC

```
/**
 * Original code:
 * Copyright (c) 2000-2017, Robert Sedgewick and Kevin Wayne.
 * <p>
 * Modifications:
 * Copyright (c) 2017. Phasmid Software
 */
package edu.neu.coe.info6205.union_find;

import java.util.Arrays;

/**
 * Height-weighted Quick Union with Path Compression
 */
public class UF_HWQUPC implements UF {

    /**
     * Ensure that site p is connected to site q,
     *
     * @param p the integer representing one site
     * @param q the integer representing the other site
     */
    public void connect(int p, int q) {
        if (!isConnected(p, q)) union(p, q);
    }

    /**
     * Initializes an empty union-find data structure with {@code n} sites
     * {@code 0} through {@code n-1}. Each site is initially in its own
     * component.
     *
     * @param n               the number of sites
     * @param pathCompression whether to use path compression
     * @throws IllegalArgumentException if {@code n < 0}
     */
    public UF_HWQUPC(int n, boolean pathCompression) {
        count = n;
```

```java
        parent = new int[n];
        height = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            height[i] = 1;
        }
        this.pathCompression = pathCompression;
    }

    /**
     * Initializes an empty union-find data structure with {@code n} sites
     * {@code 0} through {@code n-1}. Each site is initially in its own
     * component.
     * This data structure uses path compression
     *
     * @param n the number of sites
     * @throws IllegalArgumentException if {@code n < 0}
     */
    public UF_HWQUPC(int n) {
        this(n, true);
    }

    public void show() {
        for (int i = 0; i < parent.length; i++) {
            System.out.printf("%d: %d, %d\n", i, parent[i], height[i]);
        }
    }

    /**
     * Returns the number of components.
     *
     * @return the number of components (between {@code 1} and {@code n})
     */
    public int components() {
        return count;
    }
    /**
     * Returns the component identifier for the component containing site
{@code p}.
     *
     * @param p the integer representing one site
     * @return the component identifier for the component containing site
```

```
{@code p}
     * @throws IllegalArgumentException unless {@code 0 <= p < n}
     */
    public int find(int p) {
        validate(p);
        int root = p;

        if (pathCompression) {
            doPathCompression(root);
        } else {
            while (root != parent[root]) {
                  root = parent[root];
            }
        }

        return parent[root];
    }
    /**
     * Returns true if the the two sites are in the same component.
     *
     * @param p the integer representing one site
     * @param q the integer representing the other site
     * @return {@code true} if the two sites {@code p} and {@code q} are in
the same component;
     * {@code false} otherwise
     * @throws IllegalArgumentException unless
     *                                  both {@code 0 <= p < n} and {@code
0 <= q < n}
     */
    public boolean connected(int p, int q) {
        return find(p) == find(q);
    }
    /**
     * Merges the component containing site {@code p} with the
     * the component containing site {@code q}.
     *
     * @param p the integer representing one site
     * @param q the integer representing the other site
     * @throws IllegalArgumentException unless
     *                                  both {@code 0 <= p < n} and {@code
0 <= q < n}
     */
```

```java
    public void union(int p, int q) {
        // CONSIDER can we avoid doing find again?
        mergeComponents(find(p), find(q));
        count--;
    }
    @Override
    public int size() {
        return parent.length;
    }

    /**
     * Used only by testing code
     *
     * @param pathCompression true if you want path compression
     */
    public void setPathCompression(boolean pathCompression) {
        this.pathCompression = pathCompression;
    }
    @Override
    public String toString() {
        return "UF_HWQUPC:" + "\n  count: " + count +
                "\n  path compression? " + pathCompression +
                "\n  parents: " + Arrays.toString(parent) +
                "\n  heights: " + Arrays.toString(height);
    }

    // validate that p is a valid index
    private void validate(int p) {
        int n = parent.length;
        if (p < 0 || p >= n) {
            throw new IllegalArgumentException("index " + p + " is not 
between 0 and " + (n - 1));
        }
    }

    private void updateParent(int p, int x) {
        parent[p] = x;
    }

    private void updateHeight(int p, int x) {
        height[p] += height[x];
    }
    /**
     * Used only by testing code
     *
     * @param i the component
     * @return the parent of the component
     */
    private int getParent(int i) {
        return parent[i];
    }
    private final int[] parent;   // parent[i] = parent of i
    private final int[] height;   // height[i] = height of subtree rooted
at i
    private int count;  // number of components
    private boolean pathCompression;

    private void mergeComponents(int i, int j) {
      int rootX = find(i);
      int rootY = find(j);

      if (height[rootX] < height[rootY]) {
            parent[rootX] = rootY;
            height[rootY] += height[rootX];
      } else if (height[rootX] > height[rootY]) {
            parent[rootY] = rootX;
            height[rootX] += height[rootY];
      } else {
            parent[rootY] = rootX;
            height[rootX]++;
      }
    }
    /**
     * This implements the single-pass path-halving mechanism of path
compression
     */
    private void doPathCompression(int i) {
       while (i != parent[i]) {
         parent[i] = parent[parent[i]];
         i = parent[i];
       }
    }
}
```

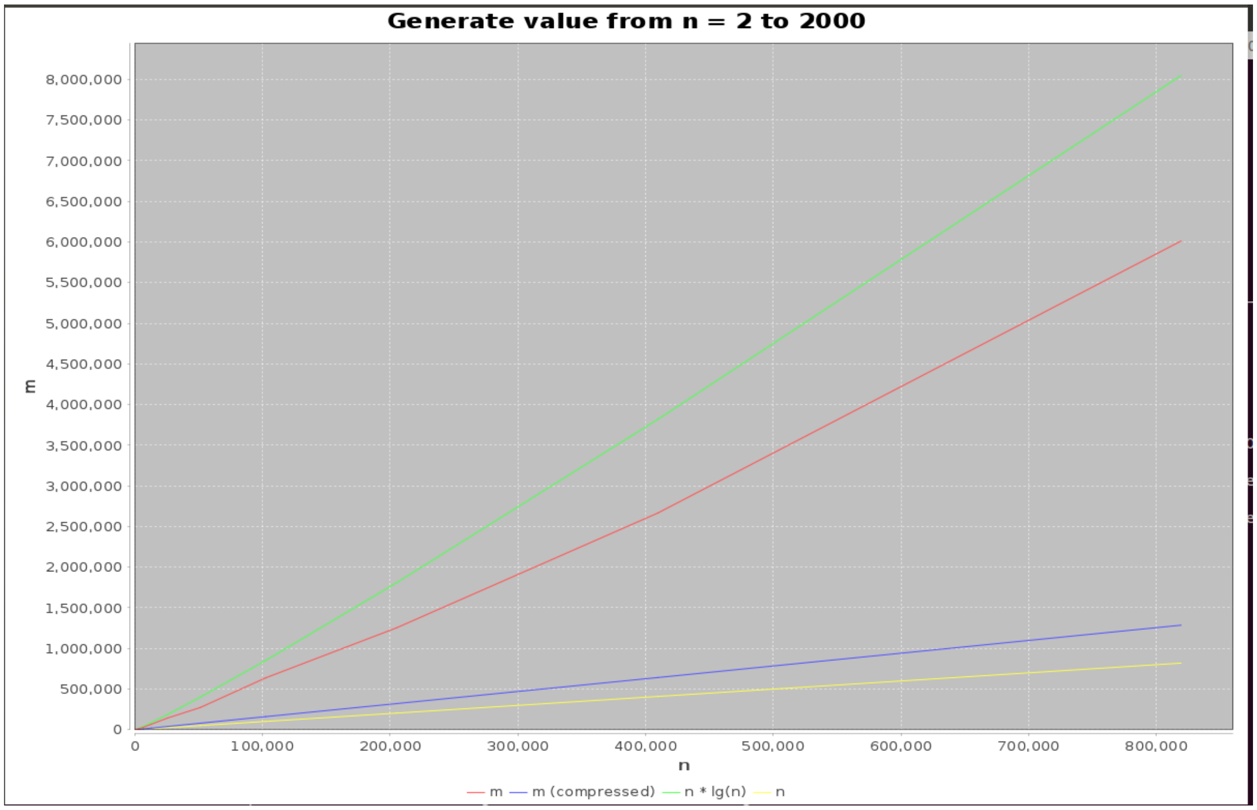
HWQUPC_Solution

```
package edu.neu.coe.info6205.union_find;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartFrame;
import org.jfree.chart.JFreeChart;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;

public class HWQUPC_Solution {

      private String frameTitle;
      private String graphTitle;
      private String xAxis;
      private String yAxis;
      private int width;
      private int height;

      private List<Group> groups;

      public static class Group {
            private String name;
            private List<Pair> pairs;

            public Group(String name, List<Pair> pairs) {
                  this.name = name;
                  this.pairs = pairs;
            }

            public String getName() {
                  return name;
            }
            public List<Pair> getPairs() {
                  return pairs;
            }
```

```
    }

    public static class Pair {
          private double x;
          private double y;

          public Pair(double x, double y) {
                this.x = x;
                this.y = y;
          }

          public double getX() {
                return x;
          }
          public double getY() {
                return y;
          }
    }

    public HWQUPC_Solution(
                String frameTitle,
                String graphTitle,
                String xAxis,
                String yAxis,
                int widht,
                int height) {
          this.frameTitle = frameTitle;
          this.graphTitle = graphTitle;
          this.xAxis= xAxis;
          this.yAxis = yAxis;
          this.width = widht;
          this.height = height;

          this.groups = new ArrayList<>();
    }

    public void addGroup(Group group) {
          groups.add(group);
    }

    public void plot() {
            XYSeriesCollection collection = new XYSeriesCollection();

            for (Group group : groups) {
                  String name = group.getName();

                  XYSeries series = new XYSeries(name);

                  List<Pair> pairs = group.getPairs();
                  for (Pair pair : pairs) {
                        double x = pair.getX();
                        double y = pair.getY();

                        series.add(x, y);
                  }

                  collection.addSeries(series);
            }

            JFreeChart chart = ChartFactory.createXYLineChart(
                        graphTitle,
                        xAxis,
                        yAxis,
                        collection);

            ChartFrame chartFrame = new ChartFrame(frameTitle, chart);
            chartFrame.setVisible(true);
            chartFrame.setResizable(false);
            chartFrame.setSize(width, height);
      }

      public static void main(String[] args) {
            int min = 2, max = 2000, step = 1, simulations = 10;

            List<Integer> inputs = IntStream.iterate(min, i -> i <= max, i
-> i + step)
                  .boxed()
                  .collect(Collectors.toList());

            List<Pair> pairs = new ArrayList<>(inputs.size());
            List<Pair> pairsCompressed = new ArrayList<>(inputs.size());
            List<Pair> logPairs = new ArrayList<>(inputs.size());
            List<Pair> linearPairs = new ArrayList<>(inputs.size());
            for (int input = 100; input < 1000000; input *= 2) {
                  double m = count(input, simulations, false);
                  double mCompressed = count(input, simulations, true);

                  double log2Value = log2(input);
                  double log = ((double) input) * log2Value / 2;

                  pairs.add(new Pair(input, m));
                  pairsCompressed.add(new Pair(input, mCompressed));
                  logPairs.add(new Pair(input, log));
                  linearPairs.add(new Pair(input, input));

                  System.out.println(input
                              + " m -> " + m
                              + " mCompressed -> " + mCompressed
                              + " -> log = " + log +
                              " -> log2Value = " + log2Value);
            }

            HWQUPC_Solution plotter = new HWQUPC_Solution(
                        "Union Find Relationship",
                        "Generate value from n = " + min + " to " + max,
                        "n",
                        "m",
                        1200,
                        800);

//          Group linerGroup = new Group("n - m", diffPairs);
            Group group = new Group("m", pairs);
            Group compressedGroup = new Group("m (compressed)",
pairsCompressed);
            Group logGroup = new Group("n * lg(n)", logPairs);
            Group linearGroup = new Group("n", linearPairs);

//          plotter.addGroup(linerGroup);
            plotter.addGroup(group);
            plotter.addGroup(compressedGroup);
            plotter.addGroup(logGroup);
            plotter.addGroup(linearGroup);

            plotter.plot();
```

```
      }

      private static double log2(int N) {
            double result = (Math.log10(N) / Math.log10(2));
        return result;
    }

      public static double count(
                  int n,
                  int simulations,
                  boolean pathCompression) {
        Random random = new Random();
        double total = 0;

        for (int i = 0; i < simulations; i++) {
            UF_HWQUPC uf = new UF_HWQUPC(n, pathCompression);

            double count = 0;
            while(uf.components() > 1) {
                  int a = random.nextInt(n), b = random.nextInt(n);
                  count++;

                  if(uf.connected(a, b)) {
                        continue;
                  }

                  uf.union(a, b);
            }

            total += count;
//            System.out.println("total increase to: " + total);
        }

        double average = total / (double) simulations;
//        System.out.println("average: " + average);
        return average;
    }

}
```

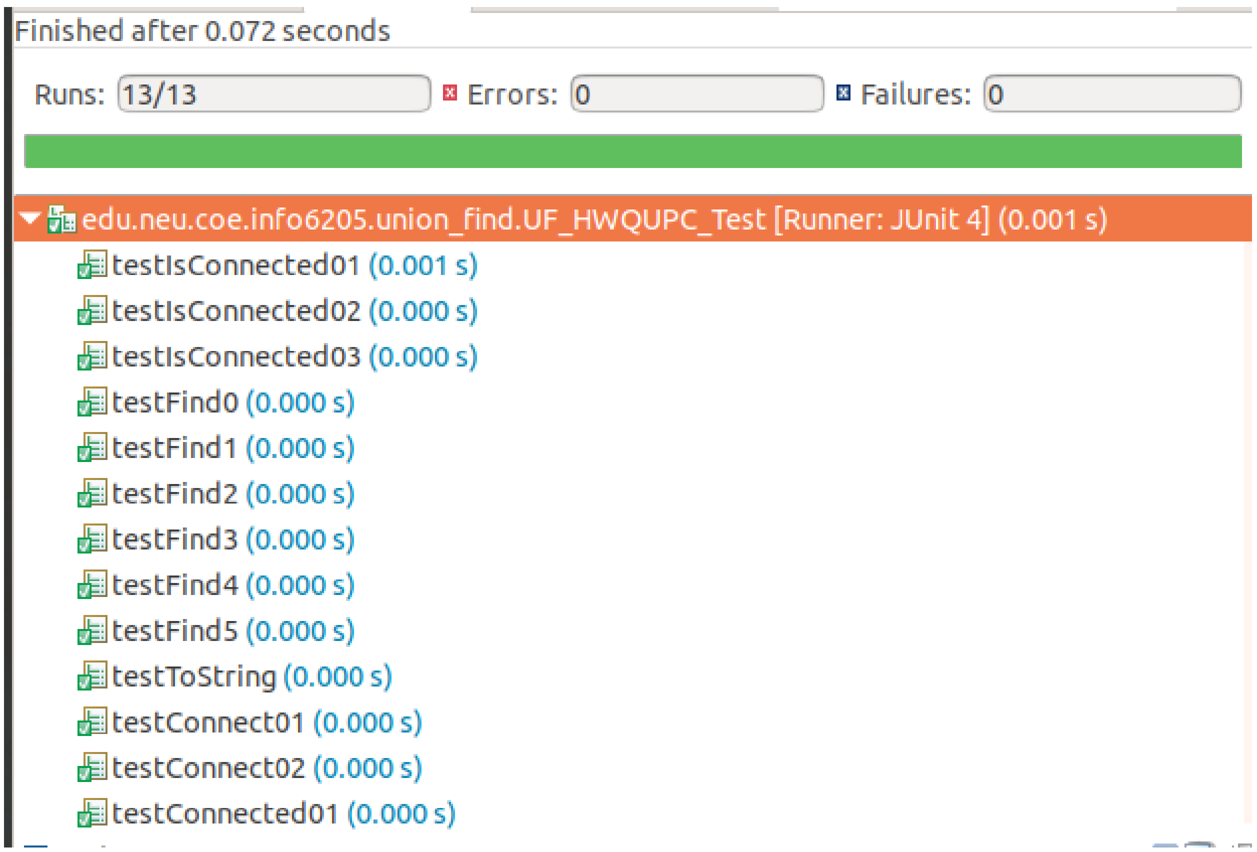
GRAPHICAL REPRESENTATION



UNIT TEST SCREENSHOTS

UF_HWQUPC_Test

Report



```
Finished after 0.072 seconds
Runs: 13/13    Errors: 0    Failures: 0

edu.neu.coe.info6205.union_find.UF_HWQUPC_Test [Runner: JUnit 4] (0.001 s)
    testIsConnected01 (0.001 s)
    testIsConnected02 (0.000 s)
    testIsConnected03 (0.000 s)
    testFind0 (0.000 s)
    testFind1 (0.000 s)
    testFind2 (0.000 s)
    testFind3 (0.000 s)
    testFind4 (0.000 s)
    testFind5 (0.000 s)
    testToString (0.000 s)
    testConnect01 (0.000 s)
    testConnect02 (0.000 s)
    testConnected01 (0.000 s)
```

Code

```
/*
 * Copyright (c) 2017. Phasmid Software
 */

package edu.neu.coe.info6205.union_find;

import edu.neu.coe.info6205.util.PrivateMethodTester;
import org.junit.Test;

import static org.junit.Assert.*;

public class UF_HWQUPC_Test {
```

```
@Test
public void testToString() {
    Connections h = new UF_HWQUPC(2);
    assertEquals("UF_HWQUPC:\n" +
            "  count: 2\n" +
            "  path compression? true\n" +
            "  parents: [0, 1]\n" +
            "  heights: [1, 1]", h.toString());
}

/**
 *
 */
@Test
public void testIsConnected01() {
    Connections h = new UF_HWQUPC(2);
    assertFalse(h.isConnected(0, 1));
}

/**
 *
 */
@Test(expected = IllegalArgumentException.class)
public void testIsConnected02() {
    Connections h = new UF_HWQUPC(1);
    assertTrue(h.isConnected(0, 1));
}

/**
 *
 */
@Test
public void testIsConnected03() {
    Connections h = new UF_HWQUPC(2);
    final PrivateMethodTester tester = new PrivateMethodTester(h);
    assertNull(tester.invokePrivate("updateParent", 0, 1));
    assertTrue(h.isConnected(0, 1));
}

/**
 *
```

```java
     */
    @Test
    public void testConnect01() {
        Connections h = new UF_HWQUPC(2);
        h.connect(0, 1);
    }

    /**
     *
     */
    @Test
    public void testConnect02() {
        Connections h = new UF_HWQUPC(2);
        h.connect(0, 1);
        h.connect(0, 1);
        assertTrue(h.isConnected(0, 1));
    }

    /**
     *
     */
    @Test
    public void testFind0() {
        UF h = new UF_HWQUPC(1);
        assertEquals(0, h.find(0));
    }

    /**
     *
     */
    @Test
    public void testFind1() {
        UF h = new UF_HWQUPC(2);
        h.connect(0, 1);
        assertEquals(0, h.find(0));
        assertEquals(0, h.find(1));
    }

    /**
     *
     */
    @Test
    public void testFind2() {
        UF h = new UF_HWQUPC(3, false);
        h.connect(0, 1);
        assertEquals(0, h.find(0));
        assertEquals(0, h.find(1));
        h.connect(2, 1);
        assertEquals(0, h.find(0));
        assertEquals(0, h.find(1));
        assertEquals(0, h.find(2));
    }

    /**
     *
     */
    @Test
    public void testFind3() {
        UF h = new UF_HWQUPC(6, false);
        h.connect(0, 1);
        h.connect(0, 2);
        h.connect(3, 4);
        h.connect(3, 5);
        assertEquals(0, h.find(0));
        assertEquals(0, h.find(1));
        assertEquals(0, h.find(2));
        assertEquals(3, h.find(3));
        assertEquals(3, h.find(4));
        assertEquals(3, h.find(5));
        h.connect(0, 3);
        assertEquals(0, h.find(0));
        assertEquals(0, h.find(1));
        assertEquals(0, h.find(2));
        assertEquals(0, h.find(3));
        assertEquals(0, h.find(4));
        assertEquals(0, h.find(5));
        final PrivateMethodTester tester = new PrivateMethodTester(h);
        assertEquals(3, tester.invokePrivate("getParent", 4));
        assertEquals(3, tester.invokePrivate("getParent", 5));
    }

    /**
     *
     */
    @Test
    public void testFind4() {
        UF h = new UF_HWQUPC(6);
        h.connect(0, 1);
        h.connect(0, 2);
        h.connect(3, 4);
        h.connect(3, 5);
        assertEquals(0, h.find(0));
        assertEquals(0, h.find(1));
        assertEquals(0, h.find(2));
        assertEquals(3, h.find(3));
        assertEquals(3, h.find(4));
        assertEquals(3, h.find(5));
        h.connect(0, 3);
        assertEquals(0, h.find(0));
        assertEquals(0, h.find(1));
        assertEquals(0, h.find(2));
        assertEquals(0, h.find(3));
        assertEquals(0, h.find(4));
        assertEquals(0, h.find(5));
        final PrivateMethodTester tester = new PrivateMethodTester(h);
        assertEquals(0, tester.invokePrivate("getParent", 4));
        assertEquals(0, tester.invokePrivate("getParent", 5));
    }
    /**
     *
     */
    @Test(expected = IllegalArgumentException.class)
    public void testFind5() {
        UF h = new UF_HWQUPC(1);
        h.find(1);
    }
    /**
     *
     */
    @Test
    public void testConnected01() {
        Connections h = new UF_HWQUPC(10);
//        h.show();
        assertFalse(h.isConnected(0, 1));
    }
}
```