Program Structures and Algorithms

Name: Chakradhar Grandhi Durga. **NEU ID**: 002933727

ASSIGNMENT-4

4.1 WEIGHTED QUICK-UNION (PATH COMPRESSION)

Here is the code for the weighted quick union using path compression.

```
public int find(int p) {
    validate(p);
    int root = p;
    while(p != getParent(p))
        p = parent[p];
    // END
    if(this.pathCompression)
        this.doPathCompression(root);
    return p;
}
```

I have used the basic heuristic of union find for the find function. At the end this code checks if the path compression is applied. If not, path compression is applied to the code to make it optimal.

```
private void mergeComponents(int i, int j) {
    // FIXME make shorter root point to taller one
   if(find(i)==find(j))
       return;
    if(height[i] < height[j]){
        parent[i] = parent[j];
        height[j]= Integer.max(height[i],height[j])+1;
    }
    else{
        parent[j] = parent[i];
        height[i]= Integer.max(height[i],height[j])+1;
    }
    }
```

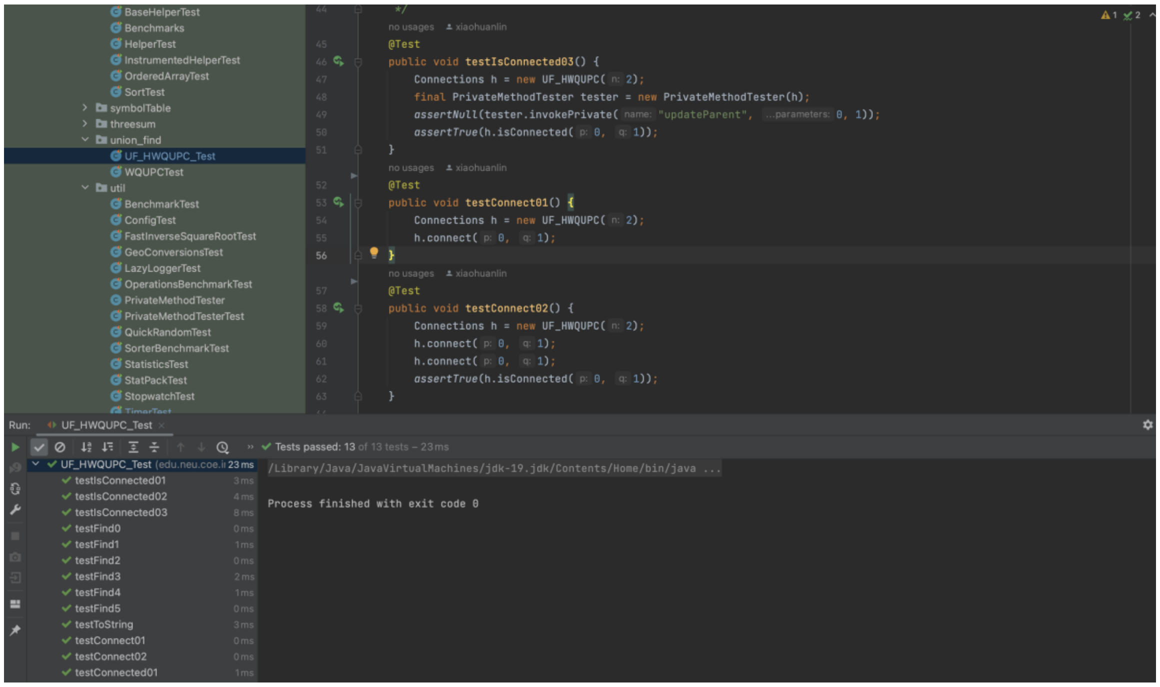
Merge components is the function used to join 2 trees and the logic here is to merge the path with shorter root to the taller one. The height is updated after merging the components and the new height will now be

$$\mathbf{max}\big(\boldsymbol{height}(\boldsymbol{node}\ \mathbf{1}),\ \boldsymbol{height}\ (\boldsymbol{node}\ \mathbf{2})\big) + \mathbf{1}.$$

```java
private void doPathCompression(int i) {
    // FIXME update parent to value of grandparent
    while(i!= getParent(i)){
        updateParent(i,getParent(getParent(i)));
        i = getParent(i);
    }
    // END
}
```

Do path compression if the actual function that performs the heuristic of the whole code of weighted union find with path compression.

Here are the test cases for the code. Here is the output



```
BaseHelperTest
Benchmarks
HelperTest
InstrumentedHelperTest
OrderedArrayTest
SortTest
> symbolTable
> threesum
v union_find
    UF_HWQUPC_Test
    WQUPCTest
v util
    BenchmarkTest
    ConfigTest
    FastInverseSquareRootTest
    GeoConversionsTest
    LazyLoggerTest
    OperationsBenchmarkTest
    PrivateMethodTester
    PrivateMethodTesterTest
    QuickRandomTest
    SorterBenchmarkTest
    StatisticsTest
    StatPackTest
    StopwatchTest
    TimerTest
```

```java
    */
    no usages  xiaohuanlin
    @Test
    public void testIsConnected03() {
        Connections h = new UF_HWQUPC( n: 2);
        final PrivateMethodTester tester = new PrivateMethodTester(h);
        assertNull(tester.invokePrivate( name: "updateParent", ...parameters: 0, 1));
        assertTrue(h.isConnected( p: 0,  q: 1));
    }
    no usages  xiaohuanlin
    @Test
    public void testConnect01() {
        Connections h = new UF_HWQUPC( n: 2);
        h.connect( p: 0,  q: 1);
    }
    no usages  xiaohuanlin
    @Test
    public void testConnect02() {
        Connections h = new UF_HWQUPC( n: 2);
        h.connect( p: 0,  q: 1);
        h.connect( p: 0,  q: 1);
        assertTrue(h.isConnected( p: 0,  q: 1));
    }
```

```
Run:  UF_HWQUPC_Test
✓ Tests passed: 13 of 13 tests – 23 ms
v ✓ UF_HWQUPC_Test (edu.neu.coe.i  23 ms
    ✓ testIsConnected01       3 ms
    ✓ testIsConnected02       4 ms
    ✓ testIsConnected03       8 ms
    ✓ testFind0               0 ms
    ✓ testFind1               1 ms
    ✓ testFind2               0 ms
    ✓ testFind3               2 ms
    ✓ testFind4               1 ms
    ✓ testFind5               0 ms
    ✓ testToString            3 ms
    ✓ testConnect01           0 ms
    ✓ testConnect02           0 ms
    ✓ testConnected01         1 ms

/Library/Java/JavaVirtualMachines/jdk-19.jdk/Contents/Home/bin/java ...

Process finished with exit code 0
```

.

You can see here from the above screenshot that all the test cases have passed successfully.

4.2 UNION-FIND FUNCTION

Here is the code that I have written for the union-find function taking random integers pairs from 0 to n-1 sites after taking n sites as an input.

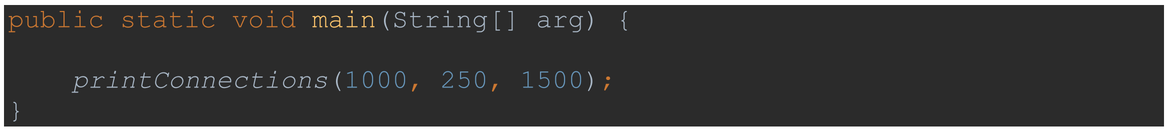
```
public static int count(int n){
    int pairs = 0;
    UF_HWQUPC uf = new UF_HWQUPC(n);
    Random random = new Random();
    while(uf.components()>1) {
        int i = random.nextInt(n);
        int j = random.nextInt(n);
        pairs++;
        if (!uf.isConnected(i, j)) {
            uf.connect(i, j);
            no_union++;
        }

    }
    return pairs;
}
```

the count function takes integer input n. it takes 2 random sites and connects them if the connection is not established. The **connect** function is used to establish the connection among the 2 sites. The counter **no_union** is used to calculate all the connections made. The function returns the total number of pairs generated.

```
public static void printConnections(int runs, int interval, int
upperBound) {
    for (int i = interval; i <= upperBound; i = i+interval) {
        int total = 0;
        for (int j = 0; j < runs; j++) {
            total+=count(i);
        }
        System.out.print("Sites : " + i + " Connections - " +
total/runs);
        System.out.println("Unions -" +no_union/runs);
        no_union=0;
    }
}
```

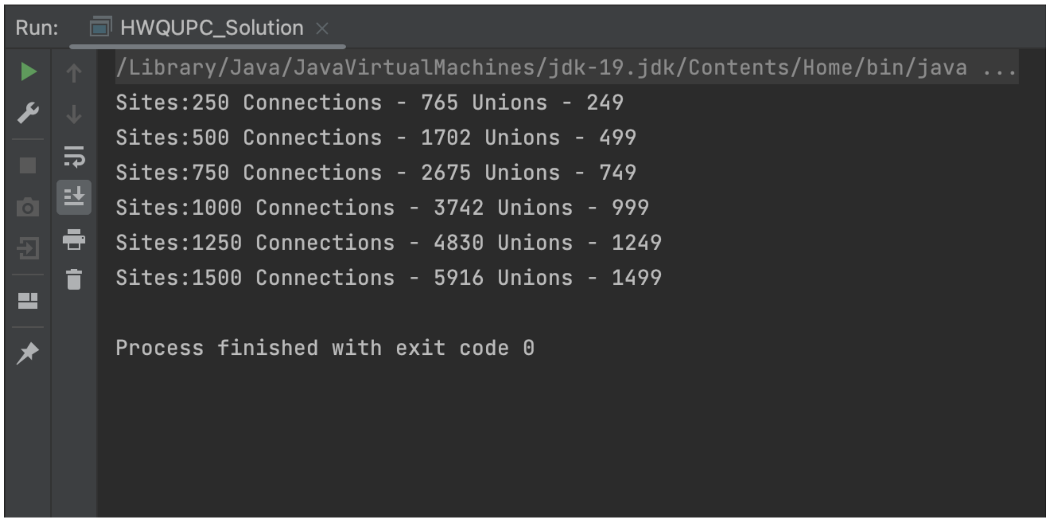
print connections is the function that takes the number of runs(to get the average), the interval(for calculation of sites at every interval), and the upper bound(to get data for a wide range of n).

we give out $\frac{\boldsymbol{pairs}}{\boldsymbol{runs}}$ to get the average number of pairs generated and $\frac{\boldsymbol{no\_unions}}{\boldsymbol{runs}}$ to get the average number of unions generated.



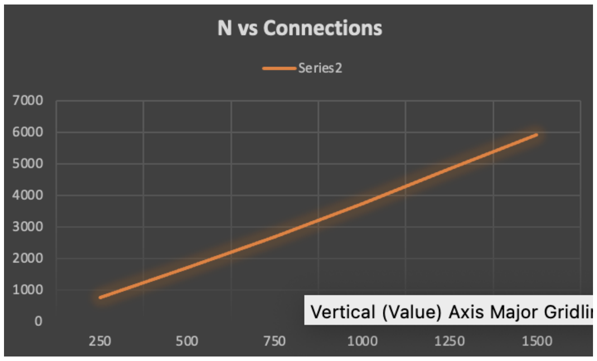
```java
public static void main(String[] arg) {

    printConnections(1000, 250, 1500);
}
```

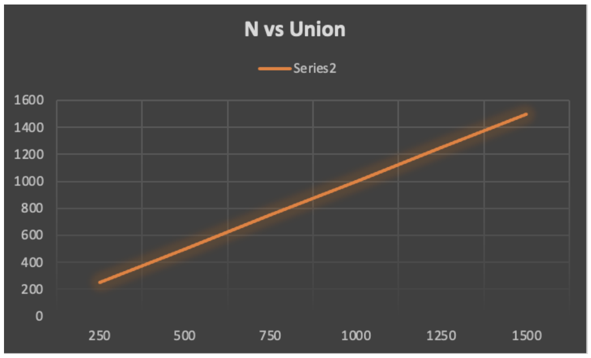
I wrote the main function which runs the above print connection to give the results and here are the results.



```
/Library/Java/JavaVirtualMachines/jdk-19.jdk/Contents/Home/bin/java ...
Sites:250 Connections - 765 Unions - 249
Sites:500 Connections - 1702 Unions - 499
Sites:750 Connections - 2675 Unions - 749
Sites:1000 Connections - 3742 Unions - 999
Sites:1250 Connections - 4830 Unions - 1249
Sites:1500 Connections - 5916 Unions - 1499

Process finished with exit code 0
```

I plotted all the results to find the relation between the number of sites and number of unions and here is how the graph looked





4.3 OBSERVATIONS AND CONCLUSIONS

If we observe the above graph the graph clearly follows the pattern

$$\boldsymbol{y = x - 1}$$

Here **y ⇒ unions**

x ⇒ number of sites

This clearly means that the tree is now a complete tree with **1 root** and **n-1 leaves** after path compression. That is the reason why we are getting unions as (sites-1).

And for

n= sites-1

the total number of connections would be

$$\approx \frac{\boldsymbol{n log(n)}}{\boldsymbol{2}}$$

The time complexity of Union Find is **O (log n)** without the use of the Union by Rank and Path Compression techniques. However, incorporating Path Compression can improve the time complexity to **O(kn),** where k is a constant. Path Compression has the added benefit of not adding extra time to the update process, as each node is directly connected to the root, reducing the length of the path.