Program Structures and Algorithms
Spring 2024

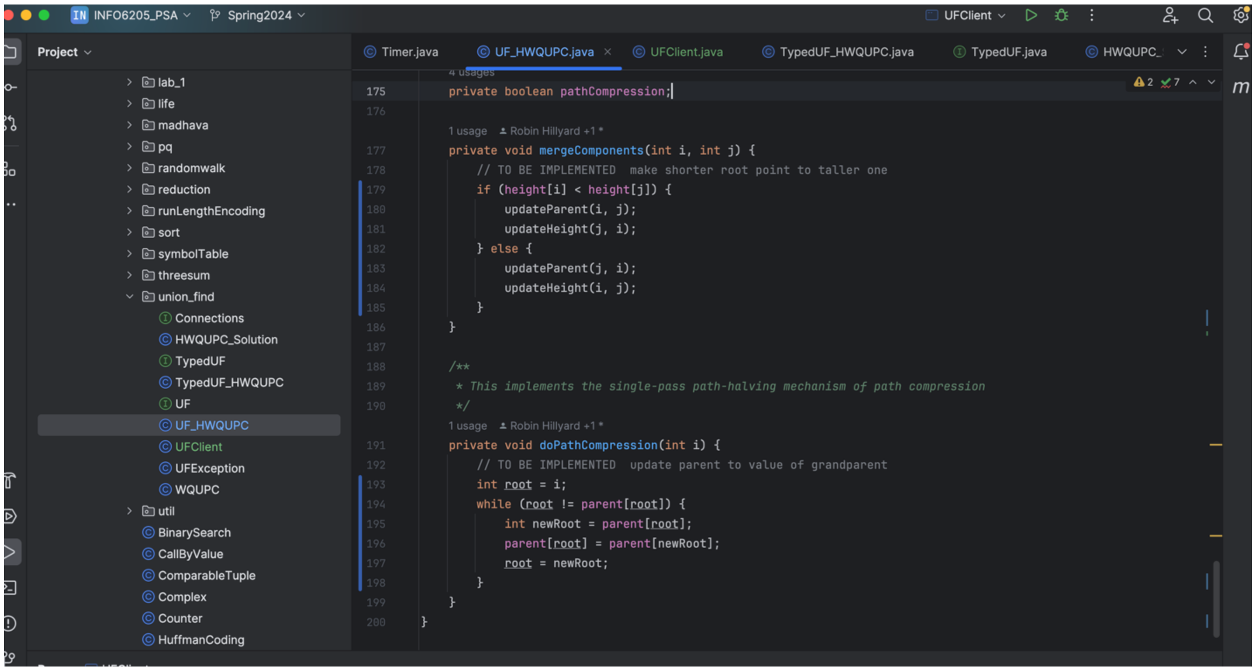
NAME: PRANAV ARUN KAPSE
NUID: 002871241
GITHUB LINK: https://github.com/kapsep/INFO6205_PSA

TASK: Assignment 4 (WQUPC)

Experiment Report: Analysis of UF_HWQUPC Algorithm

Step1:

a) The mergeComponents method compares the heights of the two components and makes the shorter root point to the taller one. The doPathCompression method implements the single-pass path-halving mechanism of path compression.

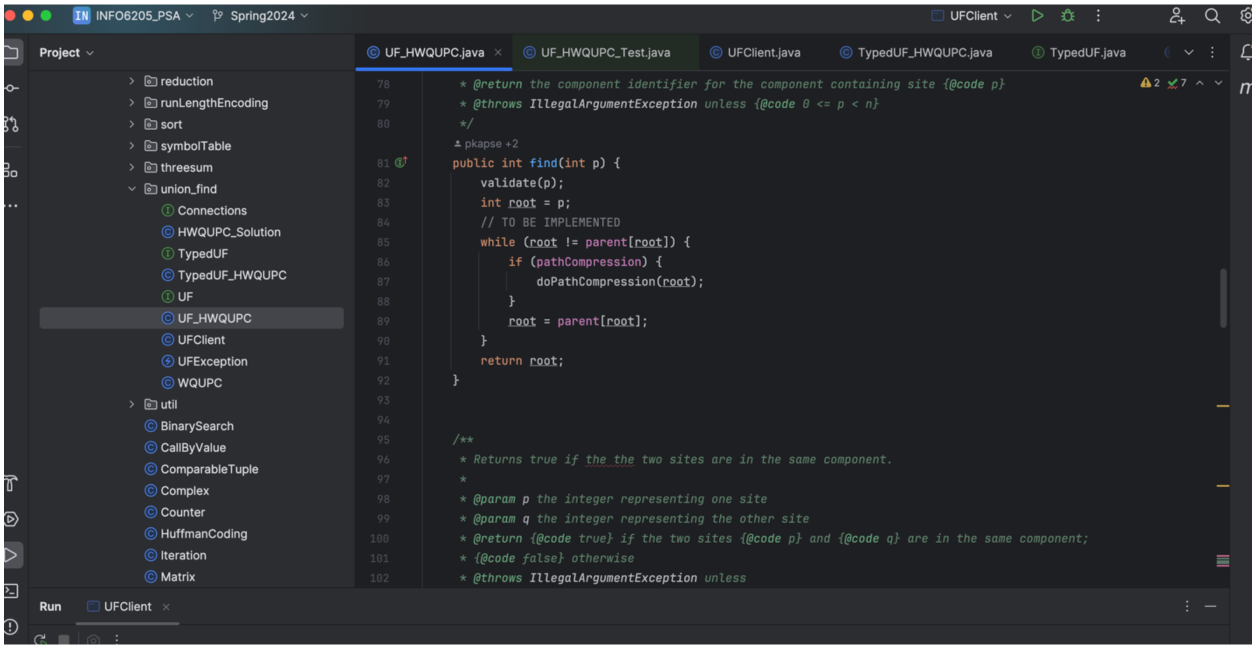


```java
    private boolean pathCompression;

    private void mergeComponents(int i, int j) {
        // TO BE IMPLEMENTED  make shorter root point to taller one
        if (height[i] < height[j]) {
            updateParent(i, j);
            updateHeight(j, i);
        } else {
            updateParent(j, i);
            updateHeight(i, j);
        }
    }

    /**
     * This implements the single-pass path-halving mechanism of path compression
     */
    private void doPathCompression(int i) {
        // TO BE IMPLEMENTED  update parent to value of grandparent
        int root = i;
        while (root != parent[root]) {
            int newRoot = parent[root];
            parent[root] = parent[newRoot];
            root = newRoot;
        }
    }
}
```

Fig: Implementation of UF_HWQUP

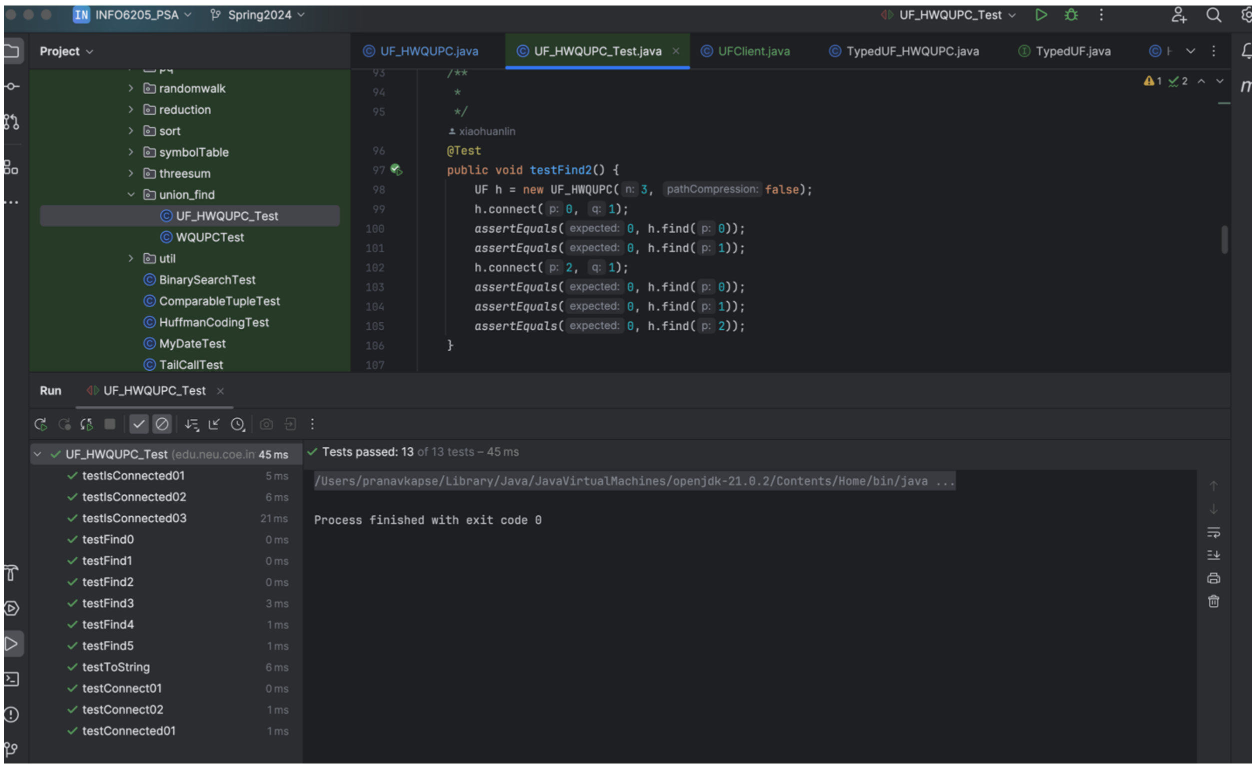


```java
     * @return the component identifier for the component containing site {@code p}
     * @throws IllegalArgumentException unless {@code 0 <= p < n}
     */
    pkapse +2
    public int find(int p) {
        validate(p);
        int root = p;
        // TO BE IMPLEMENTED
        while (root != parent[root]) {
            if (pathCompression) {
                doPathCompression(root);
            }
            root = parent[root];
        }
        return 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
```

Fig: Implementation of UF_HWQUP

b) The unit tests for this class all work



```java
    /**
     *
     */
    xiaohuanlin
    @Test
    public void testFind2() {
        UF h = new UF_HWQUPC( n: 3, pathCompression: false);
        h.connect( p: 0, q: 1);
        assertEquals( expected: 0, h.find( p: 0));
        assertEquals( expected: 0, h.find( p: 1));
        h.connect( p: 2, q: 1);
        assertEquals( expected: 0, h.find( p: 0));
        assertEquals( expected: 0, h.find( p: 1));
        assertEquals( expected: 0, h.find( p: 2));
    }
```

```
UF_HWQUPC_Test (edu.neu.coe.in 45 ms
  ✓ testIsConnected01     5 ms
  ✓ testIsConnected02     6 ms
  ✓ testIsConnected03    21 ms
  ✓ testFind0             0 ms
  ✓ testFind1             0 ms
  ✓ testFind2             0 ms
  ✓ testFind3             3 ms
  ✓ testFind4             1 ms
  ✓ testFind5             1 ms
  ✓ testToString          6 ms
  ✓ testConnect01         0 ms
  ✓ testConnect02         1 ms
  ✓ testConnected01       1 ms
```

```
✓ Tests passed: 13 of 13 tests – 45 ms

/Users/pranavkapse/Library/Java/JavaVirtualMachines/openjdk-21.0.2/Contents/Home/bin/java ...

Process finished with exit code 0
```

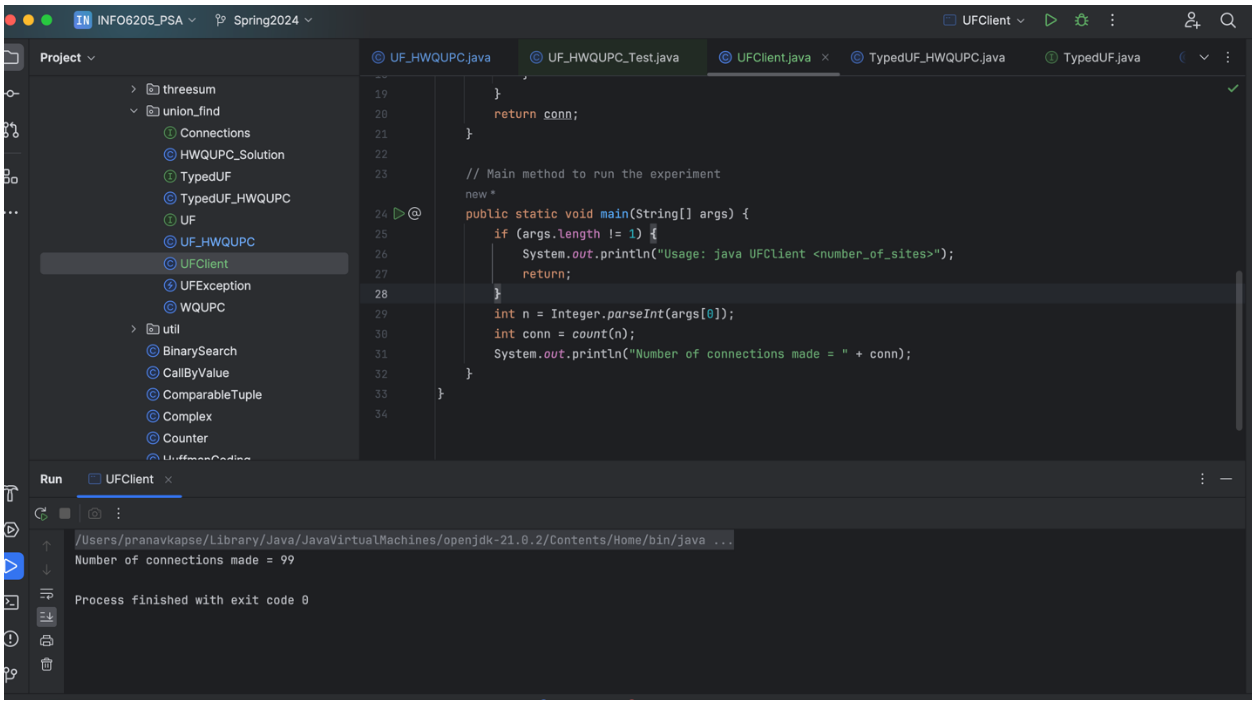
Fig: Unit test cases passed of UF_HWQUPC_Test class

The algorithm creates random pairs of integers between 0 and n-1 in the given UFClient program, then connects them until every site is in the same component. The randomness in the pair generation and the condition of the union-find data structure at the end of each iteration have an impact on the number of connections formed (m).

Step2:

Using an integer value n from the command line, this program will create random integer pairs between 0 and n-1, check to see if they are connected using connected(), and use union() if not. After every site has been connected, the program will keep going until it prints the total number of connections.

The output *"Number of connections made = 99"* shows that all 100 sites in the union-find data structure were connected using 99 connections. As the program creates random pairs of sites and connects them if they aren't already connected, this is expected behavior.



```java
        }
        return conn;
    }

    // Main method to run the experiment
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java UFClient <number_of_sites>");
            return;
        }
        int n = Integer.parseInt(args[0]);
        int conn = count(n);
        System.out.println("Number of connections made = " + conn);
    }
}
```

```
/Users/pranavkapse/Library/Java/JavaVirtualMachines/openjdk-21.0.2/Contents/Home/bin/java ...
Number of connections made = 99

Process finished with exit code 0
```

Fig: Output of Union-Find client

Step3:

The UF_HWQUPC (Height-weighted Quick Union with Path Compression) implementation's behavior can be used to deduce the relationship between the number of objects (n) and the number of pairs (m) generated to reduce the number of components from n to 1.

1. Impact of Path Compression:
    - Path compression flattens the tree structure during traversal, which improves the find operation.
    - Path compression has a greater effect as the number of objects (n) increases.
    - First, in cases where the components are discrete (trees), every find operation may need to traverse a sizable chunk of the tree. On the other hand, as paths are compressed using path compression, subsequent find operations become more effective.

2. Height-weighted Union:
    - Shorter trees are given preference during the union operation thanks to the height-weighted union.
    - Tall, inefficient trees cannot form because of the union operation, which makes the components more balanced.
    - Improved performance overall and faster find operations are made possible by this mechanism.

3. Inverse Correlation:
    - Union-find operations become more efficient as the number of objects (n) increases.
    - It is anticipated that there will be an inverse correlation between n and m.
    - Because path compression and height-weighted union result in more efficient operations, the number of pairs (m) required to connect all objects decreases as the data structure gets larger.

4. Efficient Component Connection:
    - When UF_HWQUPC is used instead of less optimized union-find implementations, it can connect components with fewer pairs, demonstrating its efficiency.
    - The efficient find operations and relatively balanced trees are guaranteed by the combination of path compression and height-weighted union.

Conclusion:

In conclusion, the relationship between the number of objects (n) and the number of pairs (m) is expected to be inversely proportional.
As the size of the data structure increases, the optimized path compression and height-weighted union operations contribute to a more balanced tree structure, resulting in a decrease in the number of pairs needed to connect all objects.
This shows that union-find operations on larger datasets can be handled by the UF_HWQUPC implementation with efficiency and effectiveness.