**Program Structures & Algorithms**

**Spring 2022**

**Assignment No.3**

Name: Roja Pinnamraju

(NUID):002925814

**Task:**

1.a) Implement height-weighted Quick Union with Path Compression.

I have implemented methods as follows:

a) doPathCompression(int i): updated parent to value of grandparent.

b) find(int p): return the root of p.

c) mergeComponents(int i , int j): added logic .

**2. Develop HWQUPC\_Solution**

Implemented HWQUPC\_Solution.java that generates a number of connections with a number of objects. I have created a main method that takes random values, calls count() method, and prints the number of connections.

**OUTPUT :**

**1)**

**A picture containing text

Description automatically generated**

**2)**

**Text

Description automatically generated**

**Relationship Conclusion :**

I conducted several runs for different n values to check the relation. In all the runs I could see there was an increase in the number of pairs required as the n

value increases. The number of pairs formed(m) increases vastly as the number of objects(n) increases.

* On plotting the graph, I could see there is almost a linearithmic relationship between a number of objects and connecting pairs.
* Time taken for components to reduce 1 will depend on the number of objects taken.

The relationship of the number of pairs needed to reduce components from n objects to 1 would be

*m = f(n) = 0.5 x n \* ln(n)*

**Evidence/Graph :**

For larger values of n, although not equal, the average number of pairs needed to reduce the components to 1 is close to *0.5 x n \* ln(n).*

In this union-find operation, we check if the pairs are connected or disconnected (n ln(n)). There are only two possibilities for each pair. Hence, the relationship between m and n is almost identical to *0.5 x n \* ln(n).*

Below are the results for the performed simulations:

|  |  |  |
| --- | --- | --- |
| **n** | **0.5\*nlogn** | **m** |
| 275 | 772 | 853 |
| 291 | 825 | 912 |
| 293 | 832 | 920 |
| 313 | 899 | 991 |
| 384 | 1143 | 1259 |
| 457 | 1399 | 1535 |
| 465 | 1428 | 1566 |
| 477 | 1471 | 1600 |
| 513 | 1601 | 1756 |
| 520 | 1626 | 1777 |
| 531 | 1666 | 1819 |
| 536 | 1684 | 1843 |
| 540 | 1699 | 1858 |
| 540 | 1699 | 1855 |
| 573 | 1820 | 1978 |
| 615 | 1975 | 2152 |
| 620 | 1993 | 2179 |
| 622 | 2001 | 2181 |
| 644 | 2083 | 2266 |
| 654 | 2120 | 2318 |
| 673 | 2191 | 2390 |
| 679 | 2214 | 2421 |
| 727 | 2395 | 2604 |
| 828 | 2782 | 3020 |
| 840 | 2828 | 3082 |
| 858 | 2898 | 3149 |
| 902 | 3069 | 3341 |
| 945 | 3237 | 3510 |
| 964 | 3312 | 3581 |
| 964 | 3312 | 3599 |

I have checked two plots to test the relationship between “n” and “m”. They are as follows

1) m vs n  
2) m vs 0.5\*n\*ln(n)

Coefficient of determination (R2) has been leveraged to identify the best fit among the below plots. But turns out that both the plots have similar R2 value.

As R2 value is not helping much here, I have plotted all the three parameters (m, n, 0.5\*n\*ln(n)) in a single plot for various observation points. From the plot below, it is clearly evident that “m” and “0.5\*n\*ln(n)\* are strongly correlated and would be the best fit for our data points.

**Code:**

package edu.neu.coe.info6205.union\_find;  
import java.util.\*;  
  
public class HWQUPC\_Solution {  
  
 public static void main(String[] args)  
 {  
 int[] testdata=new int[30];int out=0;  
 Random random = new Random();  
 for(int i=0; i<testdata.length;i++) testdata[i]=random.ints(200,1000).findFirst().getAsInt();  
 for(int i=0;i<testdata.length;i++) {  
 out=0;  
 for (int j = 0; j < 5000; j++) {  
 out += *count*(testdata[i]);  
  
 }  
  
 System.*out*.println(testdata[i]+" "+out / 5000);  
 // System.out.println("For "+out+" objects, number of connections ="+out);  
 }  
 }  
  
 public static int count(int i)  
 {  
 int randoms=0;  
 UF\_HWQUPC uf=new UF\_HWQUPC(i,true);  
 Random random= new Random();  
 while(uf.components()>1)  
 {  
 int a= random.ints(0,i).findFirst().getAsInt();  
 int b= random.ints(0,i).findFirst().getAsInt();  
 randoms++;  
 if(!uf.isConnected(a,b)){  
 uf.union(a,b);  
 }  
 }  
 return randoms;  
 }  
}

*/\*\*  
 \* Original code:  
 \* Copyright © 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;  
 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;  
  
 while(root!=parent[root]){  
  
 if(pathCompression)doPathCompression(root);  
  
 root=parent[root];  
  
 }  
 // *FIXME* // END   
 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  
 \* 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}  
 \*/* 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) {  
 // *FIXME 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) {  
 // *FIXME update parent to value of grandparent* // END  
 parent[i]=parent[parent[i]];  
 }  
}

**Unit test Results:**

**Graphical user interface, text

Description automatically generated**

Graphical user interface, text, application

Description automatically generated