**Dynamic Connectivity Problem**

The **dynamic connectivity** problem involves determining if there is a path between two objects in a set of objects that can be connected or disconnected along time. This problem is frequently addressed by using algorithms that supports the following operations:

* **Union:** connects two objects;
* **Find:** verify if two objects are connected;

**REAL LIFE APPLICATIONS:** social networks, image pixels, electrics circuits,

The **connections** between objects must attend to natural and intuitive properties named as **equivalence relations:**

* **Reflexive:** *p* is connected to *p.*
* **Symmetric:** if *p* is connected to *q*, then *q* is connected to *p*.
* **Transitive:** if *p* is connected to *q* and *q* is connected to *r*, then *p* is connected to *r.*

**Connected components:** maximal set of objects that are mutually connected.

A diagram of a diagram of a diagram

AI-generated content may be incorrect.

**Image 1 – Connected components.**

**Quick Find Algorithm**

The **quick find algorithm** is one way to solve the **dynamic connectivity** problem with fast **find** operations but slow **union** operations.

**Data Structure:** uses an *array* ***id****[]*, where ***id[****i****]*** stores the identifier of the group which element *I* belongs to.

*Two elements are connected if they have the same* ***id****.*

**Operations:**

* **Find/Connected:** checks if ***id[****p****]*** *==* ***id[****q****]***. (constant time-complexity: **O(1)**).
* **Union:** to connect *p* and *q*, changes all elements with ***id[****p****]*** to ***id[****q****]***. (linear time-complexity: **O(N)**).

**Quick Union Algorithm**

The **quick union algorithm** is another way of solving the **dynamic connectivity** problem with fast **union** operations but slow **find** operations.

**Data Structure:** uses an *array* ***id[]***, but now ***id[****i****]*** stores the parent of *I* in a tree structure. The **root** of an element is found by scaling up to parents until it finds an element that is its own parent (***id[****i****]*** == *i*).

*Two elements are connected if they have the same* ***root****.*

**Operations:**

* **Find/Connected:** compares the **roots** of *p* and *q*. (tree height proportional time-complexity: **O(N)** worst-case scenario.)
* **Union:** connects the **root** of *p* to the root of *q*. (tree height proportional time-complexity: **O(N)** worst-case scenario.)

**Improving Quick-Union**

There are faster ways to implement the **quick-union** algorithms:

**Weighted Quick-Union:**

* Avoid tall trees.
* Keep track of number of objects in each tree.
* Balance by linking root of smaller tree to root of larger tree.

**Module 2 Assignment**

**Percolation Problem:**

Given a N x N matrix, check if there exists a path of full-open nodes between any open node in first row and any open node in last row. **Model a Percolation system.**

*public class Percolation {*

*private static final int TOP = 0;*

*private final boolean[][] opened;*

*private final int size;*

*private final int bottom;*

*private int openSites;*

*private final WeightedQuickUnionUF qf;*

*public Percolation(int n) {*

*if (n <= 0) {*

*throw new IllegalArgumentException();*

*}*

*size = n;*

*bottom = size \* size + 1;*

*qf = new WeightedQuickUnionUF(size \* size + 2);*

*opened = new boolean[size][size];*

*openSites = 0;*

*}*

*public void open(int row, int col) {*

*checkException(row, col);*

*opened[row - 1][col - 1] = true;*

*++openSites;*

*if (row == 1) {*

*qf.union(getQuickFindIndex(row, col), TOP);*

*}*

*if (row == size) {*

*qf.union(getQuickFindIndex(row, col), bottom);*

*}*

*if (row > 1 && isOpen(row - 1, col)) {*

*qf.union(getQuickFindIndex(row, col), getQuickFindIndex(row - 1, col));*

*}*

*if (row < size && isOpen(row + 1, col)) {*

*qf.union(getQuickFindIndex(row, col), getQuickFindIndex(row + 1, col));*

*}*

*if (col > 1 && isOpen(row, col - 1)) {*

*qf.union(getQuickFindIndex(row, col), getQuickFindIndex(row, col - 1));*

*}*

*if (col < size && isOpen(row, col + 1)) {*

*qf.union(getQuickFindIndex(row, col), getQuickFindIndex(row, col + 1));*

*}*

*}*

*private void checkException(int row, int col) {*

*if (row <= 0 || row > size || col <= 0 || col > size) {*

*throw new IllegalArgumentException();*

*}*

*}*

*public boolean isOpen(int row, int col) {*

*checkException(row, col);*

*return opened[row - 1][col - 1];*

*}*

*public int numberOfOpenSites() {*

*return openSites;*

*}*

*public boolean isFull(int row, int col) {*

*if ((row > 0 && row <= size) && (col > 0 && col <= size)) {*

*return qf.find(TOP) == qf.find(getQuickFindIndex(row, col));*

*}*

*else throw new IllegalArgumentException();*

*}*

*private int getQuickFindIndex(int row, int col) {*

*return size \* (row - 1) + col;*

*}*

*public boolean percolates() {*

*return qf.find(TOP) == qf.find(bottom);*

*}*

*}*

**Notes:**

* **Public:** public access by other classes; **Private:** restrict access to its own class;
* **Static:** does not need an object’s instance to be accessed;
* **Final:** constant value;
* **Void:** non-returning method;
* **int/String/boolean/char/double:** specifies the data type of variable or returning method.