Union Find

Outline

1 The Dynamic Connectivity Problem

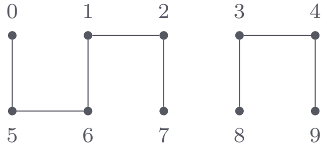
2 Union Find (UF)

3 Quick Find UF

4 Quick Union UF

5 Weighted Quick Union UF

The Dynamic Connectivity Problem



Union Find (UF)

UF

`int find(int p)`	returns the canonical site of the component containing site `p`
`int count()`	returns the number of components
`boolean connected(int p, int q)`	returns `true` if sites `p` and `q` belong to the same component, and `false` otherwise
`void union(int p, int q)`	connects sites `p` and `q`

Union Find (UF)

Program: `Components.java`

⇝ Standard input: n (int) and a sequence of pairs of integers representing sites

⇝ Standard output: number of components left after merging the sites that are in different components

```
>_ ~/workspace/dsa/programs

$ cat ../data/tinyUF.txt
10
4 3
3 8
6 5
9 4
2 1
8 9
5 0
7 2
6 1
1 0
6 7
$ java Components < ../data/tinyUF.txt
2 components
$ _
```

Union Find (UF)

Components.java

```java
import dsa.WeightedQuickUnionUF;
import stdlib.StdIn;
import stdlib.StdOut;

public class Components {
    public static void main(String[] args) {
        int n = StdIn.readInt();
        WeightedQuickUnionUF uf = new WeightedQuickUnionUF(n);
        while (!StdIn.isEmpty()) {
            int p = StdIn.readInt();
            int q = StdIn.readInt();
            uf.union(p, q);
        }
        StdOut.println(uf.count() + " components");
    }
}
```

Quick Find UF

Instance variables:

- ⇝ An array of component identifiers: `int[] id`
- ⇝ Number of components: `int count`

Quick Find UF

		id[]									
p	q	0	1	2	3	4	5	6	7	8	9
6	7	1	1	1	8	8	1	1	1	8	8

Quick Find UF

QuickFindUF.java

```java
package dsa;

import stdlib.StdIn;
import stdlib.StdOut;

public class QuickFindUF implements UF {
    private int[] id;
    private int count;

    public QuickFindUF(int n) {
        id = new int[n];
        for (int i = 0; i < n; i++) {
            id[i] = i;
        }
        count = n;
    }

    public int find(int p) {
        return id[p];
    }

    public int count() {
        return count;
    }

    public boolean connected(int p, int q) {
        return find(p) == find(q);
    }

    public void union(int p, int q) {
        int pID = find(p);
        int qID = find(q);
        if (pID == qID) {
            return;
        }
```

Quick Find UF

QuickFindUF.java

```
        for (int i = 0; i < id.length; i++) {
            if (id[i] == pID) {
                id[i] = qID;
            }
        }
        count--;
    }

    public static void main(String[] args) {
        int n = StdIn.readInt();
        QuickFindUF uf = new QuickFindUF(n);
        while (!StdIn.isEmpty()) {
            int p = StdIn.readInt();
            int q = StdIn.readInt();
            if (uf.connected(p, q)) {
                continue;
            }
            uf.union(p, q);
            StdOut.println(p + " " + q);
        }
        StdOut.println(uf.count() + " components");
    }
}
```

Quick Find UF

Operation	$T(n)$
`QuickFindUF(int n)`	n
`int find(int p)`	1
`int count()`	1
`boolean connected(int p, int q)`	1
`void union(int p, int q)`	n

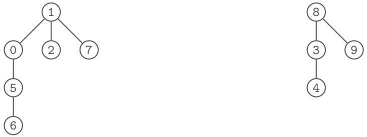
Quick Union UF

Instance variables:

- ⇝ An array of parent identifiers: `int[] parent`
- ⇝ Number of components: `int count`

Quick Union UF

		parent[]									
p	q	0	1	2	3	4	5	6	7	8	9
6	7	1	1	1	8	3	0	5	1	8	8



Quick Union UF

QuickUnionUF.java

```java
package dsa;

import stdlib.StdIn;
import stdlib.StdOut;

public class QuickUnionUF implements UF {
    private int[] parent;
    private int count;

    public QuickUnionUF(int n) {
        parent = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
        count = n;
    }

    public int find(int p) {
        while (p != parent[p]) {
            p = parent[p];
        }
        return p;
    }

    public int count() {
        return count;
    }

    public boolean connected(int p, int q) {
        return find(p) == find(q);
    }

    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
```

Quick Union UF

QuickUnionUF.java

```java
        for (int i = 0; i < id.length; i++) {
            if (id[i] == pID) {
                id[i] = qID;
            }
        }
        count--;
    }

    public static void main(String[] args) {
        int n = StdIn.readInt();
        QuickFindUF uf = new QuickFindUF(n);
        while (!StdIn.isEmpty()) {
            int p = StdIn.readInt();
            int q = StdIn.readInt();
            if (uf.connected(p, q)) {
                continue;
            }
            uf.union(p, q);
            StdOut.println(p + " " + q);
        }
        StdOut.println(uf.count() + " components");
    }
}
```

Quick Union UF

Operation	$T(n)$
`QuickUnionUF(int n)`	n
`int find(int p)`	tree height
`int count()`	1
`boolean connected(int p, int q)`	tree height
`void union(int p, int q)`	tree height

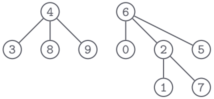
Weighted Quick Union UF

Instance variables:

- ⇝ An array of parent identifiers: `int[] parent`
- ⇝ An array of component sizes: `int[] size`
- ⇝ Number of components: `int count`

Weighted Quick Union UF

		parent[], size[]									
p	q	0	1	2	3	4	5	6	7	8	9
6	7	6	2	6	4	4	6	6	2	4	4
		1	1	3	1	4	1	6	1	1	1



Weighted Quick Union UF

WeightedQuickUnionUF.java

```java
package dsa;

import stdlib.StdIn;
import stdlib.StdOut;

public class WeightedQuickUnionUF implements UF {
    private int[] parent;
    private int[] size;
    private int count;

    public WeightedQuickUnionUF(int n) {
        parent = new int[n];
        size = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
        count = n;
    }

    public int find(int p) {
        while (p != parent[p]) {
            p = parent[p];
        }
        return p;
    }

    public int count() {
        return count;
    }

    public boolean connected(int p, int q) {
        return find(p) == find(q);
    }

```

Weighted Quick Union UF

WeightedQuickUnionUF.java

```java
    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP == rootQ) {
            return;
        }
        if (size[rootP] < size[rootQ]) {
            parent[rootP] = rootQ;
            size[rootQ] += size[rootP];
        } else {
            parent[rootQ] = rootP;
            size[rootP] += size[rootQ];
        }
        count--;
    }

    public static void main(String[] args) {
        int n = StdIn.readInt();
        WeightedQuickUnionUF uf = new WeightedQuickUnionUF(n);
        while (!StdIn.isEmpty()) {
            int p = StdIn.readInt();
            int q = StdIn.readInt();
            if (uf.connected(p, q)) {
                continue;
            }
            uf.union(p, q);
            StdOut.println(p + " " + q);
        }
        StdOut.println(uf.count() + " components");
    }
}
```

Weighted Quick Union UF

Operation	$T(n)$
`WeightedQuickUnionUF(int n)`	n
`int find(int p)`	$\log n$
`int count()`	1
`boolean connected(int p, int q)`	$\log n$
`void union(int p, int q)`	$\log n$