Dynamic Connectivity

1 Dynamic Connectivity

Given a set of N objects,

- Union command: connect two objects
- Find/connected query: is there a path connecting the two objects?

Connections are equivalency relations:

- Reflexive: p connected to p
- Symmetric: p connected to q, q connected to p
- Transitive: p $\rightarrow$ q $\rightarrow$ r, then p $\rightarrow$ r

Connected component: maximal *set* of objects that are mutually connected

1.1 Algorithm

Goal: Design efficient data structure for union-find

- Number of objects N can be huge
- Number of operations M can be huge
- Find queries and union commands may be intermixed

Union-find API:

```
public class UF {
   UF(int N)   //initialize with N objects

   //add connection b/w p and q
   void union(int p, int q)

   //are p and q connected?
   boolean connected(int p, int q)

   int find(int p)   //component identifier for p
   int count() //number of components
}
```

Dynamic-connectivity client:

```
Read in N objects from standard input.
Loop:
   Read in pair of integers from standard input
   If not connected, connect and print out pair
```

1.2 Quick-Find

Data structure:

- Integer array `id[]` of size N that tracks connected component ID for every object
- Interpretation: p and q are connected if the have the same id
- Union: To merge p and q, change all entries with `id == id[p]` to `id[q]`

Problem: plain union command is O(n) complexity for a single command.

1.3 Quick-Union

Data structure:

- Integer array `id[]` of size N
- Interpretation: `id[i]` is parent of i
- **Root** of i is `id[id[id[...id[i]...]]]`

Find: check if p and q have equivalent roots

Union: Set `id[p]` to `root(id[q])`

Root is a component with `id[i] == i`: is parent to itself

```
root(i):
    while (i != id[i]) i = id[i];
    return i;
```

Problem: O(n) worst-case for find *and* union due to complexity of root-finding

1.4 Improvements to Quick-Union

1.4.1 Weighting

- Modify quick-union to avoid tall trees
- Keep track of size of each tree
- Balance by linking root of smaller tree to root of larger tree

Maintain extra array `sz[i]` to count number of objects at tree rooted at i

```
Union(p, q):
    i = root(p); j = root(q);
    if sz[i] < sz[j]:
        id[i] = j; sz[j] += sz[i];
    else:
        id[j] = i; sz[i] += sz[j];
```

Find and union are both O($\log n$)

Proof. Find operation is O($\log n$)

1. Depth of x increases by 1 every merge.
2. Merge means tree T_1 containing x at least doubles since $|T_2| \geq |T_1|$
3. Size of tree containing x can double at most $\log n$ times

□

1.4.2 Path Compression

Just after computing the root of p, set the id of each examined node to point to that root.

Simpler one-pass variant: make every other node in the path point to its grandparent (halves path length)

```
root(i):
    while i != id[i]:
        id[i] = id[id[i]];
        i = id[i];
    return i;
```

1.4.3 Complexity

No algorithm that is completely linear, although weighted, single-pass compressed algorithm is very close in practice.

Algorithm	**Worst-case Time**
Quick-Find	MN
Quick-Union	MN
Weighted QU	$N + M \log N$
QU + Compression	$N + M \log N$
Weighted QU + Compression	$N + M \log^* N$

2 Applications of Union-Find

- Percolation
- Dynamic connectivity
- Least common ancestor
- Equivalence of FSA
- Kruskal's minimum spanning tree algorithm
- Hinley-Milner polymorphic type inference

2.1 Percolation

A model for many physical systems:

- $N \times N$ grid of sites
- Each site is open with probability p
- System **percolates** if top and bottom are connected by open sites

Likelihood of percolation depends strongly on the value of p. When N is large, theory guarantees a sharp threshold p^* such that $p > p^*$ almost certainly percolates and $p < p^*$ almost certainly does not percolate.

It is possible to model this p^* with a Monte Carlo simulation:

1. Initialize $N \times N$ grid to be blocked
2. Declare random sites open until connected to bottom
3. Vacancy percentage estimates p^*

To check whether an $N \times N$ system percolates, we initialize a list of structures for each site labeled 0 to $N^2 - 1$. Sites are in the same component if they are connected by open sites, and the system percolates if any site on the bottom row is connected to any site on the top row.

Clever trick: Introduce 2 virtual sites, initialize them to be connected to the entire top row and bottom row, respectively. Then the system percolates if the virtual top is connected to the virtual bottom.

To open a site, connect it to all neighboring open sites (quick-union algorithm).

With this simulation, we determine $p^* \approx 0.592746$.