Cosc 201
Algorithms and Data Structures
Lecture 3 (3/3/2024)
Improving Union-Find

Brendan McCane
brendan.mccane@otago.ac.nz
and
Michael Albert

Recap

Make(n) Make a set of n vertices with no edges between them. Represent the set as an array `reps` of length n where `reps`[i] is the representative of the ith element.

Union(x, y) Connect x and y by an edge (do nothing if they are already connected by an edge)

Find(x) Find (and return) a representative of the group that x belongs to. If x and y are in the same group then we require that $Find(x) = Find(y)$.

Algorithm Union(x,y)

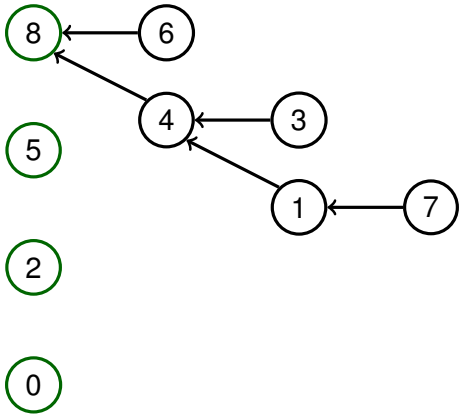
```
1: rx ← reps[x]
2: for i from 0 to n − 1 do
3:     if reps[i] = rx then
4:         reps[i] ← reps[y]
5:     end if
6: end for
```

Can we do better?

- For *Find* to work, every element needs to be able to identify their representative.
- But this doesn't have to be in one step.
- "Who's your representative?"
- "It's either that one, or whoever their representative is."
- Then $Union(x, y)$ can be:
  - Find the representatives rx of x and ry of y.
  - Make ry the representative of rx.

An example

Make(9)
Union(6, 8)
Union(3, 4)
Union(7, 1)
Union(7, 4)
Union(1, 8)



Pseudocode for *Find*(x)

Assumptions: An array, *reps*, contains the immediate representatives of the elements and x is a valid index in that array.

Outcome: Return the true representative of x by following through *reps* until an element is found that is its own representative.

```
1: while reps[x] ≠ x do
2:     x ← reps[x]
3: end while
4: return x
```

Iterative version

```
1: if reps[x] = x then
2:     return x
3: end if
4: return Find(reps[x])
```

Recursive version

Pseudocode for *Union*(x, y)

Assumptions: An array, *reps*, contains the immediate representatives of the elements. *Find* works.

Outcome: Set the new representative of x (and all the things in its set) to be the representative of y.

1: $reps[Find(x)] \leftarrow Find(y)$

Java for second version of union-find

```
public void make(int n) {
  reps = new int[n];
  for(int i = 0; i < n; i++) reps[i] = i;
}

public int find(int x) {
  while (x != reps[x]) x = reps[x];
  return x;
}

public void union(int x, int y) {
  reps[find(x)] = find(y);
}
```

Efficiency

Good news The cost of *Union* is the cost of *Find*.

Bad news The cost of *Find* is now (in the worst case), $O(n)$.

The problem is that the chain of representatives can get long. If we do $Union(0,1)$, then $Union(0,2)$, then $Union(0,3)$, etc. we get:

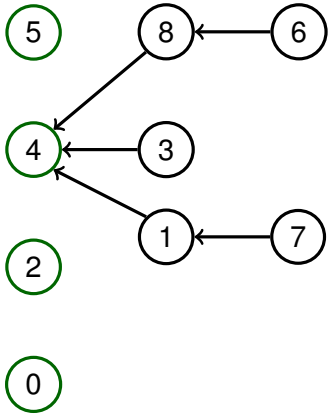
The challenge is to modify this approach to keep the complexity of *Find* under control.

The idea

- For a representative, let its rank be the length of the longest chain of ‘local representatives’ that reaches it.
- When forming a union - consider the two representatives involved. If one of them is larger rank, make it the local (and global) representative of the other.
- If they are of equal rank, make the second the representative of the first.
- Then the maximum of the two ranks can increase only if they were equal to begin with.

A modified example

Make(9)
Union(6, 8)
Union(3, 4)
Union(7, 1)
Union(7, 4)
Union(1, 8)



The code

- The main change in the code is to add a new array called `rank`.
- This records the rank of the representative elements at any time.
- The `union` operation does a “root union” on the corresponding representatives.
- In such a union, the element of larger rank retains its representative status.
- In case of a tie, we use the second element and increment its rank by one.

Java for third version of union-find

```
public void make(int n) {
  reps = new int[n];
  rank = new int[n];
  for (int i = 0; i < n; i++) {
    reps[i] = i;
    rank[i] = 0;
  }
  groups = n;
}

public int find(int x) {
  while (x != reps[x]) {
    x = reps[x];
  }
  return x;
}
```

```
public void union(int x, int y) {
  rootUnion(find(x), find(y));
}

private void rootUnion(int x, int y) {
  if (x == y) {
    return;
  }
  groups--;
  if (rank[x] > rank[y]) {
    reps[y] = x;
  } else if (rank[y] > rank[x]) {
    reps[x] = y;
  } else { // ranks are equal
    reps[x] = y;
    rank[y]++;
  }
}
```

The analysis

- The time required for a `find` operation is bounded by a constant times the length of the chain of representatives that needs to be followed.
- The maximum length of the chain is the rank of the representative.
- So, how long can the chain be?
- Or, put another way, if the rank of the representative is k, how large must the set be?
- If the minimum size of a set of rank k is n, then the maximum rank of set of size n is k.

Analysis continued

- For $k = 0$ we're talking about a 'bare point', so it must have size at least 1 (in fact, exactly 1).
- To form a representative of rank 1 we must take the union of two sets of rank 0. So the minimum size of a set of rank 1 is 2. Since we only ever add elements to sets, this can't get any smaller.
- For larger k – how did that set get formed? It must have been formed by the union of two sets of rank $k - 1$. So its size must be at least twice the minimum size of a set of rank $k - 1$.
- In other words the minimum size of a set of rank k is 2^k.
- Since $2^k \leqslant n$ this gives $k = O(\log n)$ and we have a logarithmic bound on the cost of `find` (and also of `union`).

Next time: Introducing *induction* which will allow us to make this argument formally.