NPTEL MOOC,JAN-FEB 2015
Week 5, Module 1

DESIGN AND ANALYSIS OF ALGORITHMS

Union-Find data structure

MADHAVAN MUKUND, CHENNAI MATHEMATICAL INSTITUTE
http://www.cmi.ac.in/~madhavan

Kruskal's algorithm, minimum cost spanning tree

- Process edges in ascending order of cost
- If an edge (u,v) does not create a cycle, add it to the tree
  - (u,v) can be added if u and v are in different components
  - After adding (u,v) these components get merged
- How can we keep track of components and merge them efficiently?

Union-Find data structure

- A set of elements S partitioned into subsets, or components, $\{C_1, C_2, \ldots, C_k\}$
  - Each s in S belongs to exactly one C_j
- Support the following operations
  - MakeUnionFind(S) — set up initial components, each s in S is a separate singleton component {s}
  - Find(s) — returns the component containing s
  - Union(C,C’) — merges the components C and C’

Naming the components

- Assign a label to each s in S to name its component
  - Two elements are in the same component if they have the same label
- Choice of labels is not important
- Easy option: use S itself as the set of labels
  - Initially, each s in S is assigned label s
  - After Merge(u,u’), change all labels u to u’ or vice versa

Naive implementation

- Assume S = {1,2,…,n}
  - Recall that this is our convention for nodes in a graph
- Set up an array Component[1..n]
  - MakeUnionFind(S): Set Component[i] = i, for all i
  - Find(i): Return Component[i]
  - Union(k,k'): For each i in 1..n, if Component[i] == k, update Component[i] = k'

Complexity …

- MakeUnionFind(S): Set Component[i] = i, for all i
  - O(n)
- Find(i): Return Component[i]
  - O(1)
- Union(k,k’): For each i in 1..n, if Component[i] == k, update Component[i] = k’
  - O(n)
- Sequence of m Union( ) operations: $O(m^2)$

Improved implementation

- As before, array Component[1..n]
- Also, for each component k, a list Members[k] of its members
- Array Size[k] records size of list Members[k]

Improved implementation

MakeUnionFind(S)

- Set Component[i] = i, for all i
- Initialize Members[i] = [i], Size[i] = 1, for all i

Find(i)

- Return Component[i]

Union(k,k’)

- For each i in Members[k], set Component[i] = k’
- Merge Members[k] and Members[k’]
- Update Size[k’] = Size[k] + Size[k’]

Why does this help?

- List Members[k] allows us to update a component in time proportional to its size
  - O(Size[k]) rather than O(n)
- How can we make use of Size[k] ?
  - Always merge smaller set into larger set
  - If Size[k] < Size[k’], relabel k as k’ else relabel k’ as k

Why does this help?

- Always merge smaller set into larger set
  - If Size[k] < Size[k’], relabel k as k’ else relabel k’ as k
- Individual merge operation could still take time O(n)
  - Both Size[k], Size[k’] could be about n/2
  - Need to do more careful “accounting”

Why does this help?

- For each element s, size of Component[s] at least doubles each time it is relabelled
- After a sequence of m Union( ) operations, at most 2m elements have been “touched”
  - Size of Component[s] is at most 2m
- Size of Component[s] grows as 1,2,4,…, so s is relabelled at most O(log m) times

Why does this help?

- After m Union( ) operations, at most O(m) elements have had their component updated, each at most O(log m) times
  - Recall that the list Members[k] allows us to update component k in time O(Size[k])
- Overall, m Union( ) operations take O(m log m) time
- Works out to O(log m) steps per Union( ) operation
  - Amortized complexity of Union( ) is O(log m)

Back to Kruskal's algorithm

- Sort edges E = $[e_1, e_2, \ldots, e_m]$ in ascending order
- MakeUnionFind(V)—each vertex j is labelled j
- Add edge $e_k = (u,v)$, if e_k does not create a cycle
  - Check that Find(u) != Find(v)
  - If so, merge components: Union(Find(u),Find(v))

Back to Kruskal …

- Tree has n-1 edges, so O(n) Union( ) operations
  - O(n log n) amortized cost overall
- Sorting the edges initially takes time O(m log m), but m is at most n^2, so equivalently O(m log n)
- Overall this gives O((m+n) log n), which is same as Prim's algorithm using heaps (to be done soon)

Summary

- Implement Union-Find using array Component[1..n], lists Member[1..n] and array Size[1..n]
- MakeUnionFind(S) is O(n)
- Find(s) is O(1)
- Amortized complexity of each Union(k,k’) is O(log m) over a sequence of m operations