CS490: Problem Solving in Computer Science

Lecture 8: Introductory Graph Theory III

Dustin Tseng
Mike Li

Wednesday January 16, 2006

- Union Find
- Minimum Spanning Tree

- Union Find
- Minimum Spanning Tree

Introduction

- union-find is a disjoint sets datastructure
- each element belongs to one set, identified by the “leader”
- the datastructure supports 2 operations:
- Find(x): given x, find x’s leader
- Union(x, y): merge x’s and y’s set together under one leader

Simple Implementation

- Let us label the elements by integers 1 to n.
- Moreover, assign each element a direct superior.
- A leader's direct superior would be itself.

```
int FIND( int x ) {
    if( uf[x] == x ) return x; // x is the leader
        return FIND( uf[x] );
}

void UNION( int x, int y ) {
    uf[FIND( x )] = FIND( y );
}
```

Simple Implementation

- Find() simply follows the uf[] links up until it reaches the leader
- Union() changes x's leader to y's, thus merging the two sets. (we could also have done it the other way)
- the running time is not that great here.
- Find() takes linear time.
- Union() calls Find() twice.
- Find() is the limiting factor $\rightarrow$ how do we improve?

Betteer Implementation

- We would like to reduce the steps needed to find x's leader as much as possible.
- One technique for doing is is called "path compression".
- suppose we have called Find(x) and got z.
- we could then change uf[x] to z, so next time, z is returned right away when Find(x) is called.

Better Implementation

```
int FIND( int x ) {
    if( uf[x] != uf[uf[x]] ) uf[x] = FIND( uf[x] );
        return uf[x];
}
```

- the first line checks whether the path has length at least 2
- if it does we set uf[x] to the leader

How is This Better

- This doesn't seem like an improvement
- Find(x) still takes linear time
- BUT, what about the next time?
- we need amortized analysis here(420)
- suppose we have n different people whose leader are all themselves
- then we execute k Union() and Find() operations in some unknown order
- our current implementation only require $O(k \log(n))$ time

Even Better Implementation

If we apply another technique called "uion by rank", we can reduce the running time to "almost linear".

- need an extra array
- each node will now have a rank, starting at 1
- rank[x] simply equals to the depth of the tree rooted at x
- remember that in Union() we have a choice of how to merge
- now with the rank, we can pick the shallower tree and point it to the deeper one.
- this prevents any tree from becoming deeper than $\log(n)$
- so by itself, untion by rank also guarantee a running time of $O(k \log{(n)})$ for k operations

Even Better Implementation

Here would be an implementation of union-by-rank. As a bouns, this returns true if there really is some merge, false if x and y are already in the same set.

```
bool UNION( int x, int y ) {
    int xx = FIND( x ), yy = FIND( y );
    if( xx == yy ) return false;

    // make sure rank[xx] is smaller
    if( rank[xx] > rank[yy] ) { int t = xx; xx = yy; yy = t; }

    // if both are equal, the combined tree becomes 1 deeper
    if( rank[xx] == rank[yy] ) rank[yy]++;

    uf[xx] = yy;
    return true;
}
```

Even Better Implementation

- Combining the two we will achieve $O((k + n) \log *(n))$
- $\log *(n)$ is defined as
  - $\log *(x) = 0$ if $x \leq 1$;
  - $\log *(x) = 1 + log * (log(x))$ o/w
- for a value of n less than 2^{64}, $\log *(x)$ will be less than 5.
- you can find a proof in chapter 21 in the textbook.

- Union Find
- Minimum Spanning Tree

Introduction

- a tree is an undirected, connected, acyclic graph.
- there is exactly one path between every pair of vertices in a tree
- a spanning tree of a given graph G=(V,E), is a tree T=(V, E') where E' is a subset of E.
- a minimum spanning tree is the spanning tree with the minimum cost

Kruskal's Algorithm

We will now introduce Kruskal's algorithm.

- this is a greedy algorithm
- starting with no edge in the spanning tree
- take the shortest edge and add it to the tree
- now take the rest of the edge in order of incresing length
- add them only if tree properties are preserved
- repeat until all vertices are connected

Kruskal's Algorithm

```
int uf[128];

struct Edge {              // a structure to represent an edge
    int u, v, w;           // the two endpoints and the weight
    bool operator<( const Edge &e ) const { return w < e.w; }
                           // a comparator that sorts by least weight
};

Edge edges[100000];        // the graph represented as a list of edges
int n, m;                  // the number of vertices and the number of edges

int kruskal() {

    sort( edges, edges + m );
    for( int i = 0; i < n; i++ ) uf[i] = i;

    int trees=n, sum=0; // the number of trees (parties), and the total weight

    for( int i = 0; i < m && trees > 1; i++ ) {
        if( UNION( edges[i].u, edges[i].v ) ) {
            trees--;    // use edge i in the tree
            sum += edges[i].w;
        }
    }
    return sum;
}
```

Union Find for MST

How is union find used here?

- first we sort the edge by increasing weight
- we do not need adjacenty list or matrix, a list of edges will be enough
- elements in union find are the vertices
- initially every vertex are independent
- when we examine a new edge, we check weather the endpoints are already in the same set
- if they are not then it is safe to use this edge

So why is this algorithm correct? What about its complexity?

What Else?

- References:
  - Frank and Igor's CS490 notes.
  - Cormen, Thomas H., et al. Introduction to Algorithms.
- homework help
- you will begin to present topics starting next week!