GeneralTreesUnionFind

August 7, 2020

1 General Trees & Union/Find Problem

https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/GenTreeIntro.html

1.1 Table of Contents

1.2 General Trees

- many organizations are hierarchical in nature
  - military, most businesses, governments, etc.
- binary tree is not adequate to represent organizations that have many many subordinates at lower level
- to represent these hierarchy of many arbitrary number of children, we use general trees
- general trees are trees whose internal nodes have no fixed number of children
- the following figure depicts a general tree

1.2.1 General Tree Definitions and Terminology

- a tree, T is a finite set of one or more nodes with one special node R, the root
- tree may have many **subtrees** rooted at some nodes that are children of R
  - subtrees are arranged from left to right
- a node's **out degree** is the number of children for that node
- a **forest** is a collection of one or more trees
- each node (except for root) has precisely one parent
  - a tree with n nodes must have $n - 1$ edges because each node, except the root, has one edge connecting that node to its parent

1.3 Implementation

- implementation of general tree is much harder compared to binary tree and is ignored

1.4 The Union/Find Problem

https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/UnionFind.html

1.4.1 Find: - determine if two objects are in the same set

```
- MST: given two nodes, are they in the same tree?
```

1.4.2 Union: efficiently merge two sets into one

```
- MST: merge two disjoint trees into one
```

- Kruskal's minimum spanning tree (MST) uses Union/Find technique
- what data structure can efficiently implement Union/Find operations?

1.5 Parent Pointer Trees

- a simple way to represent general tree
  - for each node store only a pointer to that node's parent
  - called **parent pointer representation**
- helps us precisely solve the Union/Find problem by offering two basic operations:
  1. determine if two objects are in the same set (the **FIND** operation)
     - follow the series of parent pointers from each node to its respective root
     - if both nodes have same root they belong to the same tree
     - helps if the height of the trees are shorter (or shortest possible)
  2. merge two disjoints sets together (intersection of disjoint sets is empty)
     - disjoint sets are united (the **UNION** operation)
     - perhaps by making one the parent of the other
     - goal is to keep the height shorter when merging
- this 2-step process goes by the name **UNION/FIND**

1.6 Parent Pointer Tree Implementation

- represented using a single array
- index represents node and the element stored represents its parent
  - a single array is used to implement a collection of trees
- use path compression and weighted union techniques
  - keep the height of the joined tree to as short as possible

[1]:
```
// a simplified demonstration of parent pointer tree
#include <iostream>
#include <vector>

using namespace std;
```

[2]:
```
// represent the above tree using parent pointer implementation
vector<int> parent(10, -1); //initialize parent vector of 10 nodes with -1
// can also initialize parent of a node at index i to itself
```

[3]:
```
parent[0] = 5;
parent[1] = 0;
parent[2] = 0;
parent[3] = 5;
parent[4] = 3;
```

```
//parent[5] = -1;
parent[6] = 5;
parent[7] = 2;
parent[8] = 5;
// parent[9] = -1
```

[3]: 5

[4]:
```
// recursive function to print path in reverse order from node to its root
void printPathReverse(vector<int> &parent, int node) {
    cout << char(node+65) << " ";
    if (parent[node] == -1) return;
    printPathReverse(parent, parent[node]);
}
```

[5]:
```
// print path to H
printPathReverse(parent, 7);
```

```
H C A F
```

[6]:
```
// recursive function to print path in from root to the given node
void printPath(vector<int> &parent, int node) {
    if (node == -1) return;
    printPath(parent, parent[node]);
    cout << char(node+65) << " ";
}
```

[7]:
```
// print path from root to to H
printPath(parent, 7);
```

```
F A C H
```

[8]:
```
// recursively find root without compressing path
int find(vector<int> &parent, int node) {
    if (parent[node] == -1) return node;
    return find(parent, parent[node]);
}
```

[9]:
```
// find root of H
cout << char(find(parent, 7)+65);
```

```
F
```

[10]:
```
// check parents of all the nodes in path to H;
// still the same as path has not been compressed
cout << char(parent[2]+65) << endl; // C
cout << char(parent[0]+65) << endl; // A
```

```
A
F
```

[10]: @0x10b9f7010

1.7 Do nodes J and H belong to the same tree?

[11]:
```cpp
// find root of J and H
cout << char(find(parent, 7)+65) << " " << char(find(parent, 9)+65) << endl;
```

```
F J
```

[12]:
```cpp
if (find(parent, 9) == find(parent, 7))
    cout << "Yes they belong to the same tree!";
else
    cout << "No they do not belong to the same tree!";
```

```
No they do not belong to the same tree!
```

1.8 Path Compression

- path compression technique can be used to create extremely shallow trees
- resets the parent of every node on the path from say X to R to R
- keeps the cost of subsequent FIND operations very close to constant
  - O(log$n$) in the worst case

[13]:
```cpp
// find root of node by compressing the path
// all the nodes in path to node will have their root changed to the root of
int findCompression(vector<int> &parent, int node) {
    if (parent[node] == -1) return node;
    parent[node] = findCompression(parent, parent[node]);
    return parent[node];
}
```

[14]:
```cpp
// find root of H and compress path
cout << char(findCompression(parent, 7)+65);
```

```
F
```

[15]:
```cpp
// check parent of H
cout << char(parent[7]+65) << endl;
```

```
F
```

[16]:
```cpp
// check parent of all the nodes in path to H
// path should be compressed making C and A's parents same as H's parents
cout << char(parent[2]+65) << endl; // 2->C
cout << char(parent[0]+65) << endl; // 0->A
```

```
F
F
```

[16]:
```
@0x10b9f7010
```

1.9 Weighted Union

- technique to join two sets by reducing their height
  - limits the total depth of the tree to $O(logn)$
- joins the tree with fewer nodes to the tree with more nodes
  - make the smaller tree's root point to the root of the bigger tree
- visualize weighted union here:https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/UnionFind.html

1.9.1 parent pointer tree implementation as ADT

[17]:
```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <sstream>
#include <iostream>
using namespace std;
```

[18]:
```cpp
// general Parent-Pointer Tree implementation for UNION/FIND
class ParPtrTree {
  private:
    vector<int> parents; // parent pointer vector
    vector<int> weights; // weights for weighted union
  public:
    // constructor
    ParPtrTree(size_t size) {
        parents.resize(size); //create parents vector
        fill(parents.begin(), parents.end(), -1); // each node is its own root␣
↪to start
        weights.resize(size);
        fill(weights.begin(), weights.end(), 1);// set all base weights to 1
    }

    // Return the root of a given node with path compression
    // recursive algorithm that makes all ancestors of the current node
    // point to the root
    int FIND(int node) {
        if (parents[node] == -1) return node;
        parents[node] = FIND(parents[node]);
        return parents[node];
    }
```

```cpp
    // Merge two subtrees if they are different
    void UNION(int node1, int node2) {
        int root1 = FIND(node1);
        int root2 = FIND(node2);
        // MERGE two trees
        if (root1 != root2) {
            // if weight of root1 is smaller;
            // root1 will point to root2
            if (weights[root1] < weights[root2]) {
                parents[root1] = root2;
                weights[root2] += weights[root1];
            }
            // root2 will point to root1
            else {
                parents[root2] = root1;
                weights[root1] += weights[root2];
            }
        }
    }

    // print representation of ParentPtrTree;
    // assuming nodes are A, B, C... as shown in the figure
    void print() {
        int w = 5, w1= 15;
        cout << setw(w1) << "parent nodes:";
        for (int i=0; i < this->parents.size(); i++) {
            if (parents[i] == -1)
                cout << setw(w) << "/";
            else
                cout << setw(w) << char(this->parents[i]+65);
        }
        cout << '\n' << setw(w1) << "parent ids:";
        for (int i=0; i < this->parents.size(); i++) {
            cout << setw(w) << this->parents[i];
        }
        cout << '\n' << setw(w1) << "node ids:";
        for (int i=0; i < this->parents.size(); i++) {
            cout << setw(w) << i;
        }
    }
};
```

1.9.2 Test ParPtrTree

- the following test code can be modified to test examples provided here: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/UnionFind.html

[19]:
```
// 10 disjoint sets: A-J mapped as 0-9
// A: 0, B: 1, ... J: 9
ParPtrTree ptr(10);
```

[20]:
```
ptr.print();
```

```
parent nodes:   /   /   /   /   /   /   /   /   /   /
  parent ids:  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1
    node ids:   0   1   2   3   4   5   6   7   8   9
```

[21]:
```
// union nodes (H) and (J)
ptr.UNION(7, 9);
ptr.print();
```

```
parent nodes:   /   /   /   /   /   /   /   /   /   H
  parent ids:  -1  -1  -1  -1  -1  -1  -1  -1  -1   7
    node ids:   0   1   2   3   4   5   6   7   8   9
```

[22]:
```
// union nodes (G) and (I)
ptr.UNION(6, 8);
ptr.print();
```

```
parent nodes:   /   /   /   /   /   /   /   /   G   H
  parent ids:  -1  -1  -1  -1  -1  -1  -1  -1   6   7
    node ids:   0   1   2   3   4   5   6   7   8   9
```

[24]:
```
// union nodes (A) and (J)
ptr.UNION(0, 9);
ptr.print();
```

```
parent nodes:   H   /   /   /   /   /   /   /   G   H
  parent ids:   7  -1  -1  -1  -1  -1  -1  -1   6   7
    node ids:   0   1   2   3   4   5   6   7   8   9
```

[25]:
```
ptr.UNION(1, 7);
ptr.print();
```

```
parent nodes:   H   H   /   /   /   /   /   /   G   H
  parent ids:   7   7  -1  -1  -1  -1  -1  -1   6   7
    node ids:   0   1   2   3   4   5   6   7   8   9
```

[26]:
```
ptr.UNION(6, 9);
ptr.print();
```

```
parent nodes:   H   H   /   /   /   /   H   /   G   H
  parent ids:   7   7  -1  -1  -1  -1   7  -1   6   7
    node ids:   0   1   2   3   4   5   6   7   8   9
```

1.10 Exercises

1. Tildes - https://open.kattis.com/problems/tildes
    - Hint: recursively update weights of intermediate nodes similar to find

- Union-Find - https://open.kattis.com/problems/unionfind

[ ]:
```

```