Union-Find Program User Manual

A Guide Written by Group 7

Daniel Vu
Collin Haines
Kevin Silver
Julio Diaz

CSCE 240
Columbia, South Carolina 29225

Table of Contents:

Getting Started

This manual serves as a guide for the Union-Find program. Union-Find is a program that is used to parse an input file containing data for a series of nodes, then compute the parent to child path between the nodes using a union-find algorithm and finally output the result. All inputs and outputs are handled by the argument in the command line; no user input is required after the command line input.

To start using the Union-Find Algorithm:

a. Open up Terminal.
b. Navigate to the directory where this file is located (e.g. 'Downloads/', '~/')
c. Compile all .cpp and .h files with the command:

```
make
```

d. Once compiling is complete run the program with the command:

```
./Aprog zin1.txt zoutx.txt zlogx.txt
```

e. To read the output and/or the log file, use the command:

```
cat zoutx.txt
cat zlogx.txt
```

Program Procedure

The program is executed from the command line with three arguments: the input file, the output file, and the log file. After initializing UnionFind, the primary operating class of the program, and opening the input and output streams, the main class assigns a variable *numberOfArcs* equal to the first integer value passed into the input file. Then `unionFind.addLink(a, b)` is executed *numberOfArcs* times.

Node values are stored in instances of the Node class, a container that stores the node's value as well as the value of its root. In this program, the value of a Node is referred to as the *currentValue*, and the value of the root of the Node is referred to as its *parentValue*.

After the first integer in the input file, the loop calls the addLink method of the UnionFind class, passing in as arguments the next two integers of the input file. This method initializes two integer variables, *smaller* and *larger*, which are used to assign the values *currentValue* and *parentValue* of a Node. The program first sorts the two values into *smaller* and *larger.* Then, the program searches for a Node with currentValue equal to *smaller* in the map *nodes*, which is located in UnionFind.h. If no such Node is found, then a Node with default values is created.

If the *currentValue* of *currentNode* is the default value (in this program, the default value of a node is *DUMMYX*, defined as a -1 integer), then the current and parent values of that node are changed to the smaller integer. Similarly, if the *currentValue* of the *parentNode* is the default value, then the current value of the node is changed to the larger integer, and the parent value is changed to the smaller value. This program assumes that the graphs are undirected, so a -> b is the same as b -> a. However, in order to simplify the input file, the program simply uses the larger value as the *currentValue* and the smaller value as the *parentValue* in all cases. The algorithm is still valid in this case, but it reduces the need to verify when an arc has been flipped in the input file.

If the *currentValue* variables of both of the nodes were already initialized, this implies that the Node already existed in the map. In this case, the smaller integer becomes the parent value of the new node, and the larger value becomes the node's current value.

Once the nodes have been initialized, we check to see if the *parentValu*e is equal to the *currentValue*. If the *parentValue* is equal to the *currentValue*, then the program assumes that this node is the root of the tree. It then runs the `dumpPaths()` method. This method traverses through paths and finds the connection to this node. If multiple paths chase to the same root, then a cycle is created. If the program is not at the top of the tree, then the program adds this new node to the tree.

Once the `unionFind.addLink()` calculations are completed, the `toString` method of the UnionFind class is called and outputs to the log file. The `toString` method uses an iterator and a for loop to crawl across the Nodes obtained from the input. A vector is initialized and then filled with the Node data reaching from front to back of a complete Node path, and this vector is then sent to the

`toStringPath` method of the UnionFind class, in which a single string is filled with each Node's contents. Each individual Node's data is obtained and formatted by the `toString` method of the Node class.

After all of the Node paths have been calculated, the steps involved in building the tree are flushed to the log file and the final tree is placed into the output stream. Once this is complete, all file streams are closed and the program terminates.

Input & Output

The Union-Find Program requires only one input file and produces one output file and one log file as set in the command line arguments.

Input File

The first line of the input file consist of a integer *n* that represents the number of arcs in the graph. Following this line will be *n* lines of data, each line consisting of a pair of integers. Each pair of integers (*a, b*) represents an arc from node *a* to node *b*. (See Figure 1.)

```
1  5
2    0   1
3    0   2
4    1   2
5    2   3
6    3   4
```

Figure 1. Example of Input File

Output File

Under normal execution, Union-Find produces an output file that contains an output of integers and arrows within parenthesis. In each parenthesis, arrows indicate the connection between a child to its parent node. The line of parenthesis represents the path that each initial node takes, ending with the root of the path. Putting these paths together makes up the tree. The output does what the program is intended to do and does not include the **redundant arcs to roots**. (See “Extra Credit”)

```
(  0 ->   0)
(  1 ->   0)(  0 ->   0)
(  2 ->   0)(  0 ->   0)
(  3 ->   2)(  2 ->   0)(  0 ->   0)
(  4 ->   3)(  3 ->   2)(  2 ->   0)(  0 ->   0)
```

Figure 2. Example Output File

Log File

The log file is the file with the most amount of information. It begins and ends with information about the time CPU usage and date.

```
TIME********************************************************************************
TIME CPU percent    0.00   0.00                    Wed Apr 23 17:59:20 2014
TIME beginning             0.00 u         0.00 s   Res:      565248
TIME beginning             0.00 u_t       0.00 s_t
TIME********************************************************************************
```

Figure 3. Log File Beginning

The next piece of data declares what file Union-Find uses as the input file, the filename of the output file and log file. It then shows what arc it is adding to the tree and then prints out the tree with the arcs and its path to the root. When a cycle is found, it shows the arc it was originally adding to the tree when the cycle was found. The next piece of information is the first path that node has to the root and then the second path to the root. It then shows the tree again with the path that is closest to the root of the tree. (See Figure 4.)

```
Main: Beginning execution
Main: infile 'zin1.txt'
Main: outfile 'zoutx.txt'
Main: logfile 'zlogx.txt'
UnionFind: BUILD TREE BY ADDING ARC(  1 ->   0)

UnionFind:
(  0 ->   0)
(  1 ->   0)(  0 ->   0)

UnionFind: BUILD TREE BY ADDING ARC(  2 ->   0)

UnionFind:
(  0 ->   0)
(  1 ->   0)(  0 ->   0)
(  2 ->   0)(  0 ->   0)

UnionFind: FOUNDCYCLE IN ADDING ARC(  2 ->   1)
UnionFind: PATH ONE (  2 ->   1)(  1 ->   0)(  0 ->   0)
UnionFind: PATH TWO (  2 ->   0)


UnionFind:
(  0 ->   0)
(  1 ->   0)(  0 ->   0)
(  2 ->   0)(  0 ->   0)
```

Figure 4. Log File Main Body

Extra Credit

By default, the Union-Find algorithm simply finds all paths from a child to the root of a tree. Because of this, various paths can have redundant subpaths. In order to display only the unique paths of a certain node, context must be kept so that the algorithm knows where it has gone. In our version of the Union-Find program, we had the algorithm run and give us strings of the paths. Then, using string manipulation, we cut out the redundant paths and only output the unique paths. (See Figure 5.)

Without String Manipulation

```
UnionFind: FOUNDCYCLE IN ADDING ARC(  8 ->   7)
UnionFind: PATH ONE (  8 ->   7)(  7 ->   3)(  3 ->   2)(  2 ->   1)(  1 ->   1)
UnionFind: PATH TWO (  8 ->   4)(  4 ->   1)(  1 ->   1)
```

With String Manipulation

```
UnionFind: FOUNDCYCLE IN ADDING ARC(  8 ->   7)
UnionFind: PATH ONE (  8 ->   7)(  7 ->   3)(  3 ->   2)(  2 ->   1)
UnionFind: PATH TWO (  8 ->   4)(  4 ->   1)
```

Figure 5. Extra Credit