Quinn Roemer

CISP - 430

Assignment 11

5/3/2018

Part 0 - Recursive Stack Traversal Implementation

Description:

The goal for this section of the assignment was to create a program that modified the professors code to traverse graph 2 which was depicted in the assignment handout. This program was to output the traversal path and the resulting tree from the traversal. I ended up rewriting the professors code to enable me to gain a greater understanding on how the code works.

Source Code:

```
//Based on code by Professor Ross. Modified by Quinn Roemer.
#include <iostream>
#include <string>

using namespace std;

int graph[8][8] = {
    0, 1, 1, 1, 0, 0, 1, 0,          //A
    1, 0, 1, 0, 1, 0, 1, 0,          //B
    1, 1, 0, 0, 0, 1, 0, 1,          //C
    1, 0, 0, 0, 0, 1, 0, 0,          //D
    0, 1, 0, 0, 0, 0, 0, 1,          //E
    0, 0, 1, 1, 0, 0, 1, 1,          //F
    1, 1, 0, 0, 0, 1, 0, 1,          //G
    0, 0, 1, 0, 1, 1, 1, 0           //H
};
//    A B C D E F G H

// where I've been
bool visited[] = { false, false, false, false, false, false, false, false };

// the resulting tree.  Each node's parent is stored
int tree[] = { -1, -1, -1, -1, -1, -1, -1, -1 };

//Function Prototypes.
void traverse(int);
void printNode(int);
void printTree();

int main()
{
    cout << "Traversal path:" << endl;

    //Calling traverse: Starter node = A.
    traverse(0);
```

```cpp
    //Printing the resulting tree.
    printTree();
}

void traverse(int startValue)
{
    int nextNode = 0;

    visited[startValue] = true;

    //Printing the current node.
    printNode(startValue);

    //Finding an unvisited node to go to next.
    while (nextNode < 8)
    {
        if (visited[nextNode] == false && graph[startValue][nextNode] == 1)
        {
                //Recording the parent of this node.
                tree[nextNode] = startValue;

                //Recursively calling traverse.
                traverse(nextNode);
        }
        nextNode++;
    }

    //Debug line
    //cout << "Returning from node #" << startValue << endl;

}

void printTree()
{
    char letter = 'A';
    char parent;

    cout << "\nThe resulting tree:" << endl << endl;
    cout << "Node \t\t Parent" << endl;

    for (int i = 0; i < 8; i++)
    {
        parent = tree[i] + 'A';

        if (parent == '@')
        {
                parent = ' ';
        }

        cout << " " << letter << "\t\t   " << parent << endl;
```

```
        letter++;
    }
}

void printNode(int startValue)
{
    char letter = startValue + 'A';
    cout << letter << endl;
}
```

Output:

```
C:\WINDOWS\system32\cmd.exe
Traversal path:
A
B
C
F
D
G
H
E

The resulting tree:

Node             Parent
 A
 B                 A
 C                 B
 D                 F
 E                 H
 F                 C
 G                 F
 H                 G
Press any key to continue . . .
```

Part 1 - Recursive Stack Traversal Hand Execution

Description:

In this section of the assignment I was to perform a hand execution of the algorithm I implemented above. This hand execution was supposed to be a recursive function call box diagram. But in addition to that, I created several other diagrams to further detail how the algorithm works.

Note: Because the following diagrams are large they will take place on the next few pages.

Diagram 1:

Note: This diagram displays the graph used in the algorithm alongside the tree that resulted in running the algorithm.

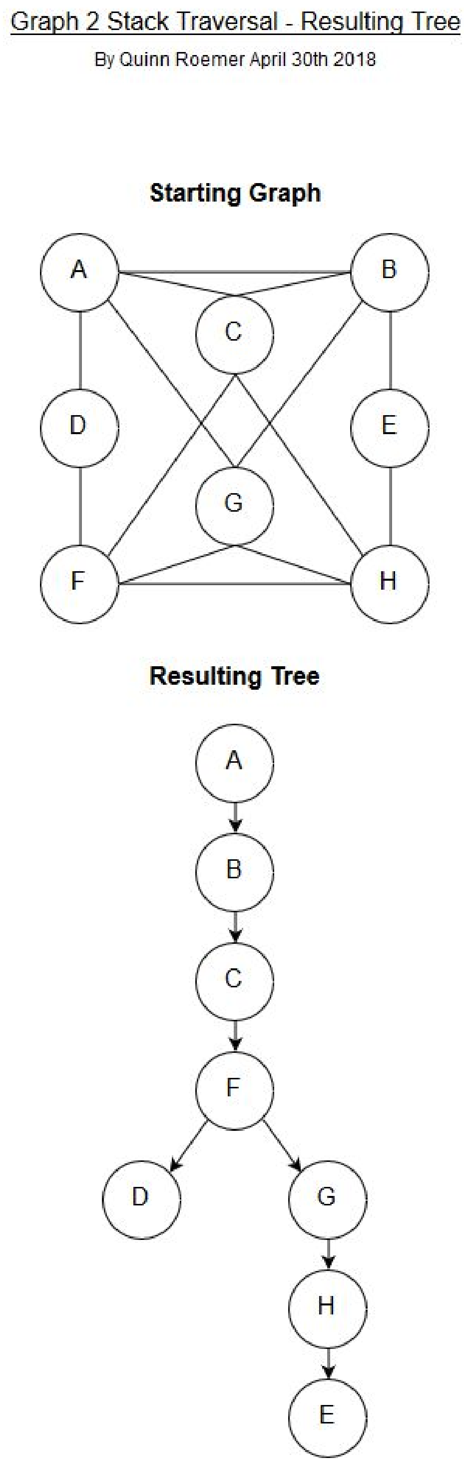

Diagram 2:

Note: This diagram display the visited array and how it was updated as the code ran. In addition, this diagram has an edge table for the graph.

Legend

Update Node

Graph 2 Stack Traversal - Resulting Arrays + Edge Table

By Quinn Roemer April 30th 2018

Traverse Call #1

Node	Visited
A	true
B	false
C	false
D	false
E	false
F	false
G	false
H	false

Traverse Call #2

Node	Visited
A	true
B	true
C	false
D	false
E	false
F	false
G	false
H	false

Traverse Call #3

Node	Visited
A	true
B	true
C	true
D	false
E	false
F	false
G	false
H	false

Traverse Call #4

Node	Visited
A	true
B	true
C	true
D	false
E	false
F	true
G	false
H	false

Traverse Call #5

Node	Visited
A	true
B	true
C	true
D	true
E	false
F	true
G	false
H	false

Traverse Call #6

Node	Visited
A	true
B	true
C	true
D	true
E	false
F	true
G	true
H	false

Traverse Call #7

Node	Visited
A	true
B	true
C	true
D	true
E	false
F	true
G	true
H	true

Traverse Call #8

Node	Visited
A	true
B	true
C	true
D	true
E	true
F	true
G	true
H	true

Edge Table

A → B → C → D → G

B → A → C → E → G

C → A → B → F → H

D → A → F

E → B → H

F → C → D → G → H

G → A → B → F → H

H → C → E → F → G

Diagram 3:

Note: This diagram displays the recursive function calls of the algorithm. Due to the large size of this diagram it will take place on the next two pages.

Part 2 - Recursive Stack Traversal Big Implementation

Description:

In the last section of this assignment I was to modify my code from part 0 to be able to traverse and output the resulting data from a much bigger graph. In addition to the usual output I had to keep track of the longest length and sequence of the longest branch in the resulting tree. To do this I implemented a counter that would increment at the beginning of traverse and decrement at the end of traverse. I used a stack to keep track of the sequence.

Source Code:

```
//Based on code by Professor Ross. Modified by Quinn Roemer.
#include <iostream>
#include <stdlib.h>
#include <fstream>
#include <string>
#include <stack>

using namespace std;

//Struct holds stacks so that they can be saved for later use..
struct stackSave {
    stack<int> savedStack;
    int size = 0;
};

//Graph Array.
int graph[100][100];

//Visited Array.
bool visited[100];

//The resulting tree is stored here.
int tree[100];

//Stack used to store longest sequence.
stackSave biggestStacks[100];
stack<int> sequence;

int maxSize[100] = { 0 };

int counter = 0;
int max = 0;
int saveOrder = 0;

//Function prototypes.
void loadGraph();
```

```
void traverse(int);
void printNode(int);
void printTree();
void saveStack();
void printStack();

int OutCount = 0;

int main()
{
    //Intilizing visited and tree.
    for (int count = 0; count < 100; count++)
    {
        visited[count] = false;
        tree[count] = -1;
    }

    //Intilizing the graph.
    loadGraph();

    cout << "Traversal path:" << endl << endl;

    //Calling traverse: Starter node = A;
    traverse(0);

    //Printing the resulting tree.
    printTree();

    //Printing the longest sequence.
    printStack();
}

void loadGraph()
{
    //Loading file.
    string line;
    int index = 0;

    ifstream myfile;

    //Opening file.
    myfile.open("bigGraph.txt");

    //Reading until end of file.
    while (myfile.peek() != EOF)
    {
        getline(myfile,line);

        for (int count = 0; count < 100; count++)
        {
```

```
                graph[index][count] = line[count] - '0';
            }

            index++;
        }

        //Closing file.
        myfile.close();
}

void traverse(int startValue)
{
        //Incrementing counter.
        counter++;

        //Pushing the startValue.
        sequence.push(startValue);

        //If counter is larger than the recorded max this code executes.
        if (counter > max)
        {
            //Calling saveStack and setting the new max.
            saveStack();
            max = counter;
        }

        int nextNode = 0;

        visited[startValue] = true;

        //Printing the current node.
        printNode(startValue);

        //Finding an unvisited node to go to next.
        while (nextNode < 100)
        {
            if (visited[nextNode] == false && graph[startValue][nextNode] == 1)
            {
                    //Recording the parent of this node.
                    tree[nextNode] = startValue;

                    //Recursively calling traverse.
                    traverse(nextNode);
            }
            nextNode++;
        }
        //Decrementing counter.
        counter--;

        //Removing top element from stack.
```

```
    sequence.pop();
}
void printTree()
{
    //This function prints the resulting tree.
    int parent;
    int counterLow = 0;
    int counterHigh = 50;

    cout << "\n\nThe resulting tree:" << endl << endl;
    cout << "Node\t\t Parent\t\tNode\t\tParent" << endl;

    for (int i = 0; i < 50; i++)
    {
        parent = tree[counterLow];

        if (parent == -1)
        {
                cout << " " << counterLow << "\t\t";
        }
        else
        {
                cout << " " << counterLow << "\t\t   " << parent;
        }

        parent = tree[counterHigh];

        if (parent == -1)
        {
                cout << " " << counterHigh << endl;
        }
        else
        {
                cout << " \t\t " << counterHigh << "\t\t  " << parent << endl;
        }

        counterHigh++;
        counterLow++;
    }
}

void printNode(int startValue)
{
    if (OutCount < 5)
    {
        printf("%3d ", startValue);
        OutCount++;
    }
    else
    {
```

```
        printf("\n%3d ", startValue);
        OutCount = 1;
    }
}

void saveStack()
{
    biggestStacks[saveOrder].size = sequence.size();
    biggestStacks[saveOrder].savedStack = sequence;
    saveOrder++;
}

void printStack()
{
    int size = -1;
    int place;
    biggestStacks;
    OutCount = 0;

    //Finding the biggest saved stack.
    for (int count = 0; count < 100; count++)
    {
        if (biggestStacks[count].size > size)
        {
                place = count;
                size = biggestStacks[count].size;
        }
    }

    //Printing the sequence and size.
    cout << "\nLongest sequence: size = " << biggestStacks[place].savedStack.size();
    cout << endl << endl;

    //Printing stack.
    while (!biggestStacks[place].savedStack.empty())
    {
        if (OutCount < 5)
        {
                printf("%3d ", biggestStacks[place].savedStack.top());
                OutCount++;
        }
        else
        {
                printf("\n%3d ", biggestStacks[place].savedStack.top());
                OutCount = 1;
        }
        biggestStacks[place].savedStack.pop();
    }
    cout << endl << endl;
}
```

Output:

(1 of 3)

```
C:\WINDOWS\system32\cmd.exe
Traversal path:

  0   1   4   2   6
  3   7   8  10   9
 14   5  12  16  15
 11  19  17  20  13
 22  18  21  26  25
 27  28  23  24  29
 32  34  31  33  35
 30  38  36  39  40
 43  37  41  42  44
 45  46  51  48  47
 52  54  49  50  53
 55  57  56  69  60
 58  59  64  65  61
 62  63  67  70  68
 66  71  73  72  75
 76  74  78  81  77
 79  82  80  83  84
 86  87  88  85  91
 89  90  92  93  95
 94  96  99  98  97

The resulting tree:

Node              Parent         Node             Parent
 0                                50               49
 1                  0             51               46
 2                  4             52               47
 3                  6             53               50
 4                  1             54               52
 5                  14            55               53
```

(2 of 3)

C:\WINDOWS\system32\cmd.exe

```
The resulting tree:

Node            Parent          Node            Parent
 0                               50              49
 1               0               51              46
 2               4               52              47
 3               6               53              50
 4               1               54              52
 5               14              55              53
 6               2               56              57
 7               3               57              55
 8               7               58              60
 9               10              59              58
 10              8               60              69
 11              15              61              65
 12              5               62              61
 13              20              63              62
 14              9               64              59
 15              16              65              64
 16              12              66              68
 17              19              67              63
 18              22              68              70
 19              11              69              56
 20              17              70              67
 21              18              71              66
 22              13              72              73
 23              28              73              71
 24              23              74              76
 25              26              75              72
 26              21              76              75
 27              25              77              81
 28              27              78              74
 29              24              79              77
 30              35              80              82
 31              34              81              78
 32              29              82              79
 33              31              83              80
 34              32              84              83
 35              33              85              88
 36              38              86              84
 37              43              87              86
 38              30              88              87
 39              36              89              91
 40              39              90              89
 41              37              91              85
 42              41              92              90
 43              40              93              92
 44              42              94              95
 45              44              95              93
 46              45              96              94
 47              48              97              95
 48              51              98              94
 49              54              99              96

Longest sequence: size = 98

 99  96  94  95  93
```

(3 of 3)

```
C:\WINDOWS\system32\cmd.exe

Longest sequence: size = 98

 99  96  94  95  93
 92  90  89  91  85
 88  87  86  84  83
 80  82  79  77  81
 78  74  76  75  72
 73  71  66  68  70
 67  63  62  61  65
 64  59  58  60  69
 56  57  55  53  50
 49  54  52  47  48
 51  46  45  44  42
 41  37  43  40  39
 36  38  30  35  33
 31  34  32  29  24
 23  28  27  25  26
 21  18  22  13  20
 17  19  11  15  16
 12   5  14   9  10
  8   7   3   6   2
  4   1   0

Press any key to continue . . .
```

Conclusion

I enjoyed this assignment. Being able to code and see the algorithm work in person is a great way to learn how it works in the first place. When I first started this assignment I was very confused on how the algorithm actually traverses the graph and generates output. However, after rewriting the code that I was provided I was able to figure out the general workings of the algorithm. Looking forward to the last assignment!