Assignment Three

Collin Drake
Collin.Drake1@Marist.edu

November 17, 2023

1 Undirected Graph

1.1 The Data Structure

A graph is a data structure made up of a collection of objects, denoted as vertices, and the edges or links that connect them. These vertices, or points, in a graph, contain three components: a unique identifier (ID), a flag, and a vector that contains pointers to the vertices connected to it by edges. There are two types of graphs, directed and undirected. Each kind defines how the edges interact between vertices. In a directed graph, each edge establishes a clear direction from one vertex to another. However, in an undirected graph, edges have no direction, showing symmetrical links between vertices. Graphs are commonly represented in two ways, either adjacency lists or Matrices. There are three fundamental operations that make up a graph: addVertex, addEdge, and findVertexByID.

Graph Constructor

```
Graphs::Graphs(){} //graph object constructor
```

Vertex Class

```
//This file creates the Vertex class
//For use with graphs
#include "Vertex.hpp"

Vertex::Vertex(string id){ //vertex class constructor
    this->id = id;
    this->processed = false;
}
```

1.1.1 Matrix

A matrix is simply a way to store or represent a graph using a two-dimensional array. In this representation, the rows and columns of a matrix represent the status of the edges between the vertices within a graph. Furthermore, the status of these edges are commonly denoted as 0 or 1, but in our case a "." communicates that there is no edge located between those vertices, while an "x" shows that there is.

Matrix

```
   void Graphs::Matrix(){ //graph object matrix
        cout<< "Matrix:" << "\n";
        //creates a 2d array and populates it with all dots
        string matrix[graph.size()+1][graph.size()+1];
        for(int row = 0; row < graph.size()+1; row++){
            for(int col = 0; col < graph.size()+1; col++){
                matrix[row][col] = ".";
            }
        }
        //populates matrix with edges
        for(int i = 0; i < graph.size(); i++){
            for(int j = 0; j < graph[i]->neighbors.size(); j++){
                string id2 = graph[i]->neighbors[j]->id;
                matrix[stoi(graph[i]->id)][stoi(id2)] = "x";
            }
        }
        //outputs matrix
        for(int row = 0; row < graph.size() + 1; row++){
            for(int col = 0; col < graph.size() + 1; col++){
                cout<< matrix[row][col] << "␣";
            }
            cout<< "\n";
        }
    }
```

1.1.2 Adjacency List

Similarly to a matrix, an adjacency list is a data structure used to represent or display a graph in a row and column fashion. An adjacency list outputs each vertex of a graph alongside its neighbors, which are displayed horizontally next to it. In our case, we have an undirected graph therefore, when outputting an adjacency list each edge is stored twice, separately in both vertices' neighbor arrays.

Adjacency List

```
   void Graphs::printAdjacencyList(){ //graph object adjacency list
        cout<< "Adjacency␣List:" << "\n";
        //outputs the graph object as an adjacency list
        for(int i = 0; i < graph.size(); i++){
            cout<< "[" << graph[i]->id << "]" << "␣";
            for (int j = 0; j < graph[i]->neighbors.size(); j++)
            {
                cout<< graph[i]->neighbors[j]->id << "␣";
            }
            cout<< "\n";
        }
    }
```

1.2 Asymptotic Analysis

Given an undirected graph containing both vertices and edges, three fundamental operations can be performed: adding a vertex, adding an edge, and searching for a vertex within the graph. All three functions can be characterized by their time complexities; O(1), O(v), and O(v). In our case, to keep track of all vertices in the graph, we will use a vector that contains pointers to each vertex object. When adding a vertex to the graph, the worst-case time complexity would be constant time, or O(1), as the new vertex would simply be added to the end of the graph object. The operation of adding an edge between two vertices would have a time complexity of linear time, or O(v) as you have to search for each vertex within our graph object using the algorithm linear search. Lastly, searching for a vertex within a graph is simply linear time, O(v), as we will use the linear search algorithm to find the target vertex within the graph.

2 Depth-First Search

2.1 The Algorithm

The depth-first search algorithm is an algorithm commonly used to repeatedly traverse a graph as far down as possible for each vertex until the target object is found. By using recursion, the depth-first search algorithm employs the computer's built-in stack. It does this by pushing each function call onto the stack until the target object is found, after which multiple pop operations are performed, unraveling the thread back to the original call, and returning the target object.

Depth-First Search

```
void Graphs::depthFirst(Vertex* fromVertex){
    //depth first search recursion traversal
    if(!fromVertex->processed){
        //check if processed, output, then process
        cout << "Visited:␣" << fromVertex->id << "\n";
        fromVertex->processed = true;
        //iterate through neighbors and recurse
        for(int i = 0; i < fromVertex->neighbors.size(); i++){
            Vertex* neighbor = fromVertex->neighbors[i];
            if(!neighbor->processed){
                depthFirst(neighbor);
            }
        }
    }
}
```

2.2 Asymptotic Analysis

Previously, we explored a stack's operations in detail, confirming that all of its operations run in constant time, O(1). In this scenario, where multiple vertices and edges are involved, a depth-first search traversal would run in O(v + e) time, considering both the vertices (v) and edges (e) of the graph.

3 Breadth-First Search

3.1 The Algorithm

The breadth-first search algorithm is a searching algorithm that explores a graph for a target object by traversing broadly before delving deeper. Unlike a depth-first searching algorithm, this method does not rely on recursion or a stack. Instead, it employs a queue along with basic "if" statements and loops.

Breadth-First Search

```
void Graphs::breadthFirst(Vertex* fromVertex){
     //breadth first search traversal
     //create a queue and add the first vertex
     Queue Q;
     Q.enqueue(fromVertex);
     fromVertex->processed = true;
     //loop through queue
     while (!Q.isEmpty())
     {
         //dequeue all vertex in the queue
         Vertex* nextVertex = Q.dequeue();
         cout << "Visited:␣" << nextVertex->id << "\n";
         //iterate through the neighbors of the dequeued vertex
         for(int i = 0; i < nextVertex->neighbors.size(); i++){
             Vertex* neighbor = nextVertex->neighbors[i];
             if(!neighbor->processed){
                 //if not processed enqueue the vertex
                 Q.enqueue(neighbor);
                 neighbor->processed = true;
             }
         }
     }
 }
```

3.2 Asymptotic Analysis

Previously, similar to a stack, we have detailed a queue's operations, validating that all of its operations run in constant time, O(1). As a result, the breadth-first search algorithm also maintains an O(v + e) time complexity.

4 Binary Search Tree

4.1 The Data Structure

A binary search tree is a collection of objects, or nodes, organized in a tree-like manner where the root node branches into two distinct sub-trees. Within a tree structure, there is a root node, branch nodes, and leaf nodes. Specifically, for each node in a binary search tree, all child nodes less than the given node are placed on the left, while child nodes greater than or equal to the given node are placed on the right. There are two fundamental operations that a binary search tree performs: inserting a node into the tree and searching the tree for a node.

Node Class

```
//This file creates the node class
 //For use with trees
 #include "Node.hpp"

 Node::Node(string val){ //Node class constructor
     this->val = val;
     this->left = nullptr;
     this->right = nullptr;
     this->parent = nullptr;
 }
```

BST Constructor

```
BST:: BST(){ //BST constructor
     //creates a BST and sets root to null
     root = nullptr;
     totalBSTSearch = 0;
 }
```

BSTInsert

```
void BST:: BSTInsert(string value){ //inserts a node into the BST
     //string to keep track of the path for each inserted node
     string path = "";
     //create new node
     Node* newNode = new Node(value);
     //create trailing and current pointers
     Node* trailing = nullptr;
     Node* current = root;

     //iterate through BST if current/root is filled
     while(current != nullptr){
         trailing = current;
         if(newNode->val.compare(current->val) < 0){
             //If the new value is less than the current go left (<)
             current = current->left; //L
             path.append("L");
         }
         else{
             //If the new value is greater than or equal to the current go right (>=)
             current = current->right; //R
             path.append("R");
         }
     }
     //set parent node to trailing
```

```
        newNode->parent = trailing;
        if(trailing == nullptr){
            //if there is no parent then the new node becomes the root node
            root = newNode;
            path.append("root␣node␣inserted");
        }
        else{
            //if there is a parent find out if new node goes left or right
            if(newNode->val.compare(trailing->val) < 0){
                //left (<)
                trailing->left = newNode;
            }
            else{
                //right (>=)
                trailing->right = newNode;
            }
        }
        //output path of each insert
        cout<< path << "\n";
    }
```

BSTree Search

```
   Node* BST:: TreeSearch(Node* node, string key, string path,
    int comparisons){ //lookup values in the BST
        comparisons++;
        if(node == nullptr || node->val == key){
            //return the retrieved value
            // output the path to find the target value and its comparison count
            cout<< path << "\n";
            cout<< comparisons << "\n";
            totalBSTSearch += comparisons;
            return node;
        }
        else if(key < node->val){ // <
            //recursive call move left
            path.append("L");
            return TreeSearch(node->left, key, path, comparisons);
        }
        else{ // >=
            //recursive call move right
            path.append("R");
            return TreeSearch(node->right, key, path, comparisons);
        }
    }
```

4.2 Asymptotic Analysis

When dealing with a binary search tree, two fundamental operations can be executed: inserting a node into the tree and searching the tree for a target node. Both functions can be characterized by their time complexity, O(log n) respectively. When adding a node or searching for a node in a binary search tree the time complexity is O(log n). This time complexity is a result of the binary search tree dividing the collection in half each time until the target node is found or the location for the new node is determined.

5 In-Order Traversal

5.1 The Algorithm

The in-order traversal algorithm is an algorithm used primarily for binary search trees; looking to the left sub-tree, then to the root, and then to the right sub-tree. In-order traversals are useful for outputting binary search trees in sorted order.

In-Order

```
       void BST:: InOrder(Node* node){ //output entire BST with an
       in-order traversal
        if(node == nullptr){
            return;
        }
        //recursively call with child node on the left
        InOrder(node->left);
        //output the value of each node
        cout << node->val << "\n";
        //recursively call with child node on the right
        InOrder(node->right);
    }
```

5.2 Asymptotic Analysis

In our case, when using an in-order traversal of a binary tree, the time complexity will be O(n), as each node is visited once. This means that as the input size grows, the time to execute the traversal will grow linearly.

6 Appendix

Graphs.cpp

```
//This file creates the Graphs class
#include "Graphs.hpp"

Graphs::Graphs(){} //graph object constructor

void Graphs::addVertex(string id){ //add vertex to graph object
    //creates a new vertex and adds it to the graph
    Vertex* newVertex = new Vertex(id);
    graph.push_back(newVertex);
}

void Graphs::addEdge(string id1, string id2) {
    //get pointers to each vertex
    Vertex* vertex1 = findVertexByID(id1);
    Vertex* vertex2 = findVertexByID(id2);
    //check if they were not found
    if (vertex1 == nullptr || vertex2 == nullptr) {
        return;
    }
    //add neighbor for both vertex
    vertex1->neighbors.push_back(vertex2);
    vertex2->neighbors.push_back(vertex1);
}


Vertex* Graphs::findVertexByID(string id){
//searches the graph object for the given vertex id
    //returns the pointer to the vertex within the graph object
    for(int i = 0; i < graph.size(); i++){
        if(graph[i]->id == id){
            return graph[i];
        }
    }
    return nullptr;
}

void Graphs::printAdjacencyList(){ //graph object adjacency list
    cout<< "Adjacency List:" << "\n";
    //outputs the graph object as an adjacency list
    for(int i = 0; i < graph.size(); i++){
        cout<< "[" << graph[i]->id << "]" << " ";
        for (int j = 0; j < graph[i]->neighbors.size(); j++)
        {
            cout<< graph[i]->neighbors[j]->id << " ";
        }
        cout<< "\n";
    }
}

void Graphs::Matrix(){ //graph object matrix
    cout<< "Matrix:" << "\n";
    //creates a 2d array and populates it with all dots
    string matrix[graph.size()+1][graph.size()+1];
    for(int row = 0; row < graph.size()+1; row++){
        for(int col = 0; col < graph.size()+1; col++){
            matrix[row][col] = ".";
        }
    }
    //populates matrix with edges
    for(int i = 0; i < graph.size(); i++){
        for(int j = 0; j < graph[i]->neighbors.size(); j++){
```

```
                string id2 = graph[i]->neighbors[j]->id;
                matrix[stoi(graph[i]->id)][stoi(id2)] = "x";
            }
        }
        //outputs matrix
        for(int row = 0; row < graph.size() + 1; row++){
            for(int col = 0; col < graph.size() + 1; col++){
                cout<< matrix[row][col] << "␣";
            }
            cout<< "\n";
        }
    }

    void Graphs::depthFirst(Vertex* fromVertex){
        //depth first search recursion traversal
        if(!fromVertex->processed){
            //check if processed, output, then process
            cout << "Visited:␣" << fromVertex->id << "\n";
            fromVertex->processed = true;
            //iterate through neighbors and recurse
            for(int i = 0; i < fromVertex->neighbors.size(); i++){
                Vertex* neighbor = fromVertex->neighbors[i];
                if(!neighbor->processed){
                    depthFirst(neighbor);
                }
            }
        }
    }

    void Graphs::breadthFirst(Vertex* fromVertex){
        //breadth first search traversal
        //create a queue and add the first vertex
        Queue Q;
        Q.enqueue(fromVertex);
        fromVertex->processed = true;
        //loop through queue
        while (!Q.isEmpty())
        {
            //dequeue all vertex in the queue
            Vertex* nextVertex = Q.dequeue();
            cout << "Visited:␣" << nextVertex->id << "\n";
            //iterate through the neighbors of the dequeued vertex
            for(int i = 0; i < nextVertex->neighbors.size(); i++){
                Vertex* neighbor = nextVertex->neighbors[i];
                if(!neighbor->processed){
                    //if not processed enqueue the vertex
                    Q.enqueue(neighbor);
                    neighbor->processed = true;
                }
            }
        }
    }

    void Graphs::resetProcessed(){
        //resets all the processed flags to false
        for(int i = 0; i < graph.size(); i++){
            graph[i]->processed = false;
        }
    }
```

BST.cpp

```
//This file creates the BST classes for insert, search, and in-order
```

```
traversal
 #include "BST.hpp"

 BST:: BST(){ //BST constructor
     //creates a BST and sets root to null
     root = nullptr;
     totalBSTSearch = 0;
 }

 void BST:: BSTInsert(string value){ //inserts a node into the BST
     //string to keep track of the path for each inserted node
     string path = "";
     //create new node
     Node* newNode = new Node(value);
     //create trailing and current pointers
     Node* trailing = nullptr;
     Node* current = root;

     //iterate through BST if current/root is filled
     while(current != nullptr){
         trailing = current;
         if(newNode->val.compare(current->val) < 0){
             //If the new value is less than the current go left (<)
             current = current->left; //L
             path.append("L");
         }
         else{
             //If the new value is greater than or equal to the current go right (>=)
             current = current->right; //R
             path.append("R");
         }
     }
     //set parent node to trailing
     newNode->parent = trailing;
     if(trailing == nullptr){
         //if there is no parent then the new node becomes the root node
         root = newNode;
         path.append("root node inserted");
     }
     else{
         //if there is a parent find out if new node goes left or right
         if(newNode->val.compare(trailing->val) < 0){
             //left (<)
             trailing->left = newNode;
         }
         else{
             //right (>=)
             trailing->right = newNode;
         }
     }
     //output path of each insert
     cout<< path << "\n";
 }

 Node* BST:: TreeSearch(Node* node, string key, string path, int comparisons){
 //lookup values in the BST
     comparisons++;
     if(node == nullptr || node->val == key){
         //return the retrieved value
         // output the path to find the target value and its comparison count
         cout<< path << "\n";
         cout<< comparisons << "\n";
         totalBSTSearch += comparisons;
         return node;
     }
```

```
        else if(key < node->val){ // <
            //recursive call move left
            path.append("L");
            return TreeSearch(node->left, key, path, comparisons);
        }
        else{ // >=
            //recursive call move right
            path.append("R");
            return TreeSearch(node->right, key, path, comparisons);
        }
    }

    void BST:: InOrder(Node* node){ //output entire BST with an in-order traversal
        if(node == nullptr){
            return;
        }
        //recursively call with child node on the left
        InOrder(node->left);
        //output the value of each node
        cout << node->val << "\n";
        //recursively call with child node on the right
        InOrder(node->right);
    }
```

7 References

7.1 Links

Below are the resources I have used to create simple, readable, and beautiful code.

- This website helped me with my in-order traversal class geeksforgeeks.com
- The textbook helped me with basic algorithm and data structure definitions: Algorithms textbook
- Your website helped me form and articulate descriptions for each data structure and algorithm used: Labouseur.com
- This link helped me reset my graph object: geeksforgeeks.com
- This link helped me with building the matrix: techiedelight.com
- This stack post helped me with my read graph function: stackoverflow.com
- This also helped me with my read graph function: stackoverflow.com
- type conversion from string to int: stackoverflow.com
- For detecting the end of the file: mathbits.com