Assignment Three – LaTeX Graphs

Patrick Tyler
Patrick.Tyler1@marist.edu

November 17, 2023

1 Graph Overview

Graphs are an important structure representing connected items. They are made of vertices and connected through edges. Graphs can be implemented in various ways such as an adjacency list, a matrix, or linked objects. Each has space or time advantages depending on the operation. Linked objects are usually favored for graph traversals because they offer an ergonomic way to traverse nodes. Adjacency lists store a graph of a collection of vertices that each have a collection neighbors (the vertices that this vertex can reach). Matrices store a graph as a vertex count by vertex count 2D array of marking which, in an unweighted graph, are usually just stored as booleans ticked on if an connection from the outer vertex exists to the inner vertex. Linked objects may store information about themselves such as id and a collection of their neighbors; they are usually exposed by just a signal origin pointer which is link to the rest of its connected component; if there are disconnected components (vertices that are not reachable starting at the origin), they may be reached by some other way and the graph may implement some other mechanism to reach them.

1.1 Inserting to a Graph

Adding an item, often called vertex, to graph can be implemented in constant time[1]. Depending on the use case of the graph it can be useful to sore additional information about the vertices such as mappings to id's to its vertex or index and a collection of all vertices themselves in the case of disconnected components. In the below code example, a vertex is inserted into an existing graph to form semantic adjacency list, matrix, and linked object structures. There could be some optimizations if the length of the graph is known ahead of time.

[1] The code example is O(n) because each matrix item existing in the graph need be resized. Also, vector push back operations may be O(n) if the internal capacity of the array is exceeded and therefore the items need to be copied to another bigger array. However, most of the time, especially with the amount of vertices in the graphs1.txt test file, push backs mostly occur in constant time because only the current size counter need be incremented.

```
        void addVertex(string* key){
            Vertex* newVertex = new Vertex;
            newVertex->id = (*key);
            newVertex->neighbors = (*new vector<Vertex*>);
            if( origin == nullptr){
                origin = newVertex;
            }
            vertices->push_back(newVertex);
            keysToIndex.insert({(*key), this->nodeCount});
            keysToVertex.insert({(*key), newVertex});
            nodeCount++;

            // Resize vectors to fit all nodes
            adjacencyList->resize(nodeCount);
            matrix->resize(nodeCount);
            // Matrix linear operation
            for(auto& neighbors : (*matrix)){
                neighbors.resize(nodeCount);
            }
        }
```

Adding the connections, often called edges, to a graph can also be implemented in constant time. The indices or linked objects of the two vertices can be provided directly or obtained in constant time with mappings. Then edges can be added through pushing to a collection in the case of adjacency lists and linked objects or indexed and marked in the case of the matrix.

```
        void addEdge(string* key1, string* key2){
            // Getting indexes for keys
            int indexKey1 = keysToIndex[(*key1)];
            int indexKey2 = keysToIndex[(*key2)];


            // Adding edge on matrix
            (*matrix)[indexKey1][indexKey2] = true;
            (*matrix)[indexKey2][indexKey1] = true;

            // adding edge on adjacency list
            (*adjacencyList)[indexKey1].push_back((*key2));
            (*adjacencyList)[indexKey2].push_back((*key1));

            // Getting Vectors for keys
            Vertex* vectorKey1 = keysToVertex[(*key1)];
            Vertex* vectorKey2 = keysToVertex[(*key2)];


            vectorKey1->neighbors.push_back(vectorKey2);
            vectorKey2->neighbors.push_back(vectorKey1);
        }
```

```
};
```

Girl: *I do not where this relationship is going. It feels undirected.*
Guy: *Great! I knew we had a mutual connection.*
Girl: *What? That's not what I meant at all.*
Guy: *You see I thought you meant that because undirected gra-*
Girl: *Nevermind, it's just you. I'm leaving.*

---

2 Traversing graphs

As mentioned before, linked objects are the preferred structure for graph traversals in most cases. This may be problematic when graphs have disconnected components, but that too can ultimately be solved. The order in which the vertices within a graph are processed are dictated by the search method. The time complexity of traversal should be $O(|v| + |e|)$ or the sum of the vertices and edge cardinality. This is because each vertex is set from unseen to seen and each edge is considered once (and only traversed when the vertex is unseen). In the case of disconnected components, there may be additional complexity in the implementation; however, the time complexity order remains the same because at worst there will only need to be $|v|$ more is seen checks with the same amount of edge checks.

2.1 Depth First Search

Depth first search processes the vertices by going deep before branching out. Implementations of this traversal generally use a stack which obtains the order by pushing each vertex and its neighbors onto the stack marking and processing each and getting the next vertex to process by popping it out of the stack until all vertices have been processed. The following code usings the call stack instead of separate stack structure.

2.2 Breadth First Search

Breadth first search (BSF) processes the vertices by branching out before going deep. Implementations of this traversal generally use a queue which obtains the order by enqueuing each vertex and its neighbors onto the queue marking and processing each and getting the next vertex to process by dequeuing it out of the queue until all vertices have been processed.

Joke redacted for this section.

---

3 Binary Search Tree

Binary search trees (BST) are a special type of graph where the vertices or usually just called nodes have at most two children, and are arranged such that the left and right subtrees all follow an ordering. Usually, this ordering dictates that all elements in the left subtree of a node are strictly less than and all elements in the right subtree of a node are greater than or equal to.

3.1 Path

The "path" of a node is relative to the root. It represents the outcomes of the comparisons of the node values down the tree. For instance, a node whose path from the root is L R R would be less than the root node and greater than or equal to the next two nodes down the tree. Below is insertion and search code which displays the path of the node. Finding a node by its value / its correct position in the tree are very similar tasks. The program starts at the root and then travels down the tree pick the left or right node to consider next until it finds the node with the desired value in the case of a search or until it finds the node that is a null pointer in the case of an insert. Both ideally work in $O(logn)$ where n is the number of elements in the BST; however, depending on the insertion order it can at worst be $O(n)$ when the data structure forms more of a stick rather than a tree. Practically, look up times can be expected to more closely resemble the logarithmic order.

```
    void cInsert(T data, int maxDataLength){
        string DELIMINATOR = " ";
        string runningPath = rightPad(data + ": ", maxDataLength + 2);
        BinNode<T>* newNode = new BinNode<T>;
        newNode->data = data;
        newNode->left = nullptr;
        newNode->right = nullptr;

        // empty case
        if(head == nullptr){
            head = newNode;
            cout << runningPath << "HEAD" << endl;
            return;
        }

        // trailing node to get previous after termination
        BinNode<T>* trailingNode;
        BinNode<T>* consideredNode = head;
        // terminates when left or right is null
        while(consideredNode != nullptr){
            trailingNode = consideredNode;
            if(data >= consideredNode->data){
                runningPath += "R" + DELIMINATOR;
                consideredNode = consideredNode->right;
            } else {
                runningPath += "L" + DELIMINATOR;
                consideredNode = consideredNode->left;
```

```
                }
            }
            // recheck check side
            // if comparison is expensive can store a bool from
            //  the while instead of rechecking
            if(data >= trailingNode->data){
                trailingNode->right = newNode;
            } else {
                trailingNode->left = newNode;
            }

            cout << runningPath << endl;
        }
```

```
        int cSearch(T data, int maxDataLength){
            string DELIMINATOR = " ";
            string runningPath = rightPad(data + ": ", maxDataLength + 2);
            auto consideredNode = head;
            // start at one for the first while check
            int comparisonCount = 1;
            while(consideredNode != nullptr && consideredNode->data != data){
                comparisonCount++;
                if(data >= consideredNode->data){
                    runningPath += "R" + DELIMINATOR;
                    consideredNode = consideredNode->right;
                } else {
                    runningPath += "L" + DELIMINATOR;
                    consideredNode = consideredNode->left;
                }
            }

            cout << rightPad(to_string(comparisonCount), 2) << " comparisons for " << runningPath << endl;

            return comparisonCount;
        }
```

3.2 In-order Traversal

In-order traversals are one of three in the common BST family of traversals, and they follow the pattern of left then root then right. In other words, the right subtree of a node is processed then the root is processed then the right subtree. The time complexity of this operation is $O(n)$ because it only needs to consider each node once. The code below produces the desired order because the left subtree is called to be processed then once that entire call stack finishes the root is processed (printed out) then the right subtree.

```
        void cInOrderTraversal(BinNode<T>* node){
            if(node == nullptr) { return; }
            cInOrderTraversal(node->left);
            cout << node->data << endl;
            cInOrderTraversal(node->right);
        }
```

Why don't we teach these computer science topics to kids? *It's too graphic.*

4 Miscellaneous Code Notes

Display code and file reading code should be commented well and, as always, is available in main.cpp file for this assignment.