Assignment Three

Ethan Morton

Ethan.Morton1@Marist.edu

November 15, 2024

1 Graphs

Template specializations are used to implement three types of undirected, non-weighted graphs: matrix, adjacency list, and linked objects. All graphs use some data structure to keep track of the vertex IDs and the edges between them. All graphs also use a lookup-table which allows you to look up the vertex in that data structure in constant time when given the vertex ID. The lookup-table is defined as unordered_map<string, int> id_lookup; for each implementation. The int represents the index used to find the vertex in a vector. This index auto-increments when a new vertex ID is added. This means the order of vertices in the vector is the same as which they appear in the graph file.

1.1 Matrix

The data structure for the matrix is vector<pair<string, vector<bool»>. To break it down, the outer vector represents each row in the matrix. The pair holds the vertex ID in the string, and the vector<bool> holds whether there is an edge to the vertex with that index in the vector. Initially, I was going to use vector<vector<uint8_t» and do bit manipulation myself to make the matrix memory efficient, but C++ does this optimization by default when using vector<bool>.

Since this is a non-weighted graph, each edge only needs to store whether it exists, hence the bool, as opposed to a numeric type. Additionally, since the graph is undirected, it only needs to store half of the edges.

The edge between 1 and 2 is the same as the edge between 2 and 1, so we only need to track this one time. To make this optimization, the number of columns in each row is equal to that row number + 1. We have to include the +1 since a vertex can have an edge with itself. This gives us a triangle-matrix instead of a square-matrix.

When doing this optimization, we also need to consider what happens when we add an edge and the vertex IDs are given in an order we don't want. Since half the matrix is missing, we would be indexing the vector outside of its range, which is undefined behavior (or maybe C++ throws an exception... I can't remember). We can simply get around this by forcing the row to be greater than or equal to the column, and swapping them if this is false.

1.1.1 Add Vertex

graph.hpp

```
void add_vertex(const string& id) {
    const int index = static_cast<int>(id_lookup.size());
    id_lookup[id] = index;
    pair<string, vector<bool>> p(id, vector<bool>(matrix.size() + 1));
    matrix.emplace_back(p);
}
```

Adding a vertex is constant time. It auto-increments the index based on the size of the lookup table, then inserts the ID-index pair into the table, then pushes the ID-row pair into the matrix.

1.1.2 Add Edge

graph.hpp

```
void add_edge(const string& id0, const string& id1) {
        uint8_t r = id_lookup[id0];
        uint8_t c = id_lookup[id1];
        //In an undirected graph, we only need to keep track of the edge in one spot.
        //This is done by forcing the ordering of the given vertex IDs to always be a specific way.
        //In this case, the row must not be less than the column.
        //This is why we can use the triangle shape shown in the Graph<Matrix> documentation.
        if (r < c) {
            std::swap(r, c);
        }
        matrix[r].second[c] = true;
}
```

Adding an edge is constant time. It gets the index from the lookup-table, forces the ordering of the index, then sets the boolean to true.

1.1.3 Print

graph.hpp

```
void print() const {
    printf(" ");
    //print X-axis vertex IDs
    for (const pair<string, vector<bool>>& pair : matrix) {
        if (pair.first.length() == 1) {
            printf("  %s", pair.first.c_str());
        } else {
            printf(" %s", pair.first.c_str());
```

```
        }
    }
    printf("\n");
    //Print each row
    for (const pair<string, vector<bool>>& pair : matrix) {
        //print vertex ID
        if (pair.first.length() == 1) {
            printf(" %s ", pair.first.c_str());
        } else {
            printf("%s ", pair.first.c_str());
        }
        //Print edges
        for (const bool col : pair.second) {
            if (col) {
                printf("1  ");
            } else {
                printf(".  ");
            }
        }
        printf("\n");
    }
}
```

The time complexity for printing the matrix is $O(\frac{n(n+1)}{2})$ or $O(n^2)$. Rows are printed in the order the appear in the graphs file. It should come out to look like a triangle.

1.2 Adjacency List

The adjacency list specialization is actually very similar to the matrix specialization. However, instead of storing whether there is an edge between each pair of vertices, it only stores the index of the vertices that each vertex has an edge with. The data structure for the adjacency list is vector<pair<string, vector<int»>. The difference between this and the matrix is the int instead of bool.

Like the matrix, the outer vector represents the rows. The string is the vertex ID, and the vector<int> is a list of the index of each vertex that this one has an edge with.

Also like the matrix, the adjacency list uses a memory-optimization thanks to the graph being undirected. We only need to store each edge once. When adding an edge, we can look at the indices given in the function arguments. We can make sure the first one is greater than or equal to the second one, and swap them if this is false. When we update the table, we are always indexing by the larger index, and inserting the smaller index into that vertex's adjacency list. Vertex 2 will reference vertex 1 as a neighbor, but not the other way around.

If we wanted to search whether vertex 1 is adjacent to vertex 2, this memory-optimization would slightly slow down the search time, but it would still remain $O(n)$. Fortunately we aren't doing this search, so we can do this optimization cost-free.

1.2.1 Add Vertex

graph.hpp

```
void add_vertex(const string& id) {
    id_lookup[id] = static_cast<int>(id_lookup.size());
    pair<string, vector<int>> p(id, vector<int>());
    adj_list.emplace_back(p);
}
```

Similar to the matrix specialization, adding a vertex is constant time, and it just inserts the ID-index pair into the table, and pushes the ID-list pair into the list.

1.2.2 Add Edge

graph.hpp

```
void add_edge(const string& id0, const string& id1) {
    int index0 = id_lookup[id0];
    int index1 = id_lookup[id1];
    //In an undirected graph, we only need to keep track of the edge in one spot.
    //This is done by forcing the ordering of the given vertex IDs to always be a specific way.
    //In this case, index0 must not be less than index1.
    //This way, the adjacency list is always indexed by the same number when given the same two vertices.
    if (index0 < index1) {
        std::swap(index0, index1);
    }
    adj_list[index0].second.emplace_back(index1);
}
```

Adding an edge is done in constant time. It forces the ordering of the index if necessary, then uses the larger index to index into the outer vector, then pushes the smaller index to the back of the inner vector.

1.2.3 Print

graph.hpp

```
void print() const {
    for (const pair<string, vector<int>>& row : adj_list) {
        //print vertex ID
        printf("%s |", row.first.c_str());
        //print vertex ID of adjacent vertices
        for (const int col : row.second) {
            printf(" %s,", adj_list[col].first.c_str());
        }
        printf("\n");
```

```
    }
}
```

1.3 Linked Objects

The linked objects specialization is a bit different from the other two. This one keeps a vector of vertices. The Vertex struct has string ID and vector<const Vertex*> neighbors. The pointers in neighbors are pointers to the other vertices in the graph's vector.

1.3.1 Add Vertex

graph.hpp

```
void add_vertex(const string& id) {
    id_lookup[id] = static_cast<int>(id_lookup.size());
    Vertex v = {id, vector<const Vertex*>()};
    vertices.emplace_back(v);
}
```

Similar to the other specializations, adding a vertex is done in constant time. The lookup table is updated and the vertex is added to the vertex vector.

1.3.2 Add Edge

graph.hpp

```
void add_edge(const string& id0, const string& id1) {
    Vertex* v0 = &vertices[id_lookup[id0]];
    Vertex* v1 = &vertices[id_lookup[id1]];
    v0->neighbors.emplace_back(v1);
    v1->neighbors.emplace_back(v0);
}
```

Adding an edge is done in constant time. Unlike the other two specializations, this one requires both vertices to be updated. The IDs are used to get the indices in the lookup table, and the addresses of those vertices are pushed into the other vertex's neighbors vector.

1.3.3 Breadth-First Traversal

graph.hpp

```
void print_breadth_first() const {
    if (vertices.empty()) {return;}
    unordered_set<const Vertex*> processed;
    queue<const Vertex*> q;
    const Vertex* v0 = &vertices[0];
    processed.insert(v0);
    q.push(v0);
    while (!q.empty()) {
        const Vertex* curr = q.front();
```

```
        q.pop();
        printf("%s, ", curr->id.c_str());
        for (const Vertex* n : curr->neighbors) {
            if (!processed.contains(n)) {
                q.push(n);
                processed.insert(n);
            }
        }
    }
    printf("\n");
}
```

Breadth-First traversal has a time complexity of $O(V+E)$ because it has to visit every vertex, and it needs to check every edge to see if a vertex has been processed. The function uses an unordered_set to keep track of all the pointers that have been processed. The algorithm starts by printing the vertex at index 0 in the vector. It uses a queue to keep track of which "layer" of vertices it's on. Once the first vertex is printed, it runs through all of its neighbors and prints those. The neighbors are stored in the queue, so in the next iteration, it goes through the items in the queue and prints it. A queue is used to print out "layers" at a time, where a layer represents the neighbors of the previous layer.

1.3.4 Depth-First Traversal

graph.hpp

```
void print_depth_first() const {
    unordered_set<const Vertex*> processed;
    dft(&vertices[0], &processed);
    printf("\n");
}

private:
void dft(const Vertex* v, unordered_set<const Vertex*>* processed) const {
    if (!processed->contains(v)) {
        printf("%s, ", v->id.c_str());
        processed->insert(v);
    }
    for (const Vertex* n : v->neighbors) {
        if (!processed->contains(n)) {
            dft(n, processed);
        }
    }
}
```

Depth-First traversal also has a time complexity of $O(V+E)$ for the same reason as breadth-first traversal. Like BFT, DFT uses an unordered_set to track the processed vertex pointers. BFT prints all the neighbors of a vertex before moving on. DFT uses the call stack to move from vertex to vertex with each call. It recursively prints the neighbor of the vertex, then its neighbor, then its neighbor, etc. It processes the next neighbor of a vertex once the function returns that printed the previous neighbor.

2 Binary Search Tree

This is just a standard binary search tree with functions to insert, search, and perform a depth-first traversal.

2.1 Insertion

binary_search_tree.cpp

```
void BinarySearchTree::insert(const string data) {
    printf("Inserting %s - Root", data.c_str());
    if (root == nullptr) {
        root = new Node(data);
        printf("\n");
        return;
    }
    Node* curr = root;
    while (curr != nullptr) {
        if (data < curr->data) {
            printf(",L");
            if (curr->left == nullptr) {
                curr->left = new Node(data);
                break;
            }
            curr = curr->left;
        } else {
            printf(",R");
            if (curr->right == nullptr) {
                curr->right = new Node(data);
                break;
            }
            curr = curr->right;
        }
    }
    printf("\n");
}
```

Insertion is a $O(log_2(n))$ operation because it has to do a binary search through the tree to find the location that the new node will be placed. It's actually slightly more because the tree likely won't be balanced. During the search, L or R is printed out to show the path of the search.

2.2 Depth-First Traversal

binary_search_tree.cpp

```
void traverse_helper(const Node* curr) {
    if (curr == nullptr) {return;}
    traverse_helper(curr->left);
```

```
    printf(" - %s", curr->data.c_str());
    traverse_helper(curr->right);
}

void BinarySearchTree::depth_first_traverse() const {
    printf("Depth-first traversal:");
    traverse_helper(root);
    printf("\n");
}
```

Depth-first traversal uses recursion to print out each element in the BST in order. It starts by making recursive calls to the left node, then when the function calls return, it starts printing out the nodes. Then it makes recursive calls to the right. This is how it prints from left to right.

2.3 Search

binary_search_tree.cpp

```
int BinarySearchTree::search(const string target) const {
    printf("Searching %s - Root", target.c_str());
    int comparisons = 0;
    const Node* curr = root;
    while (curr != nullptr) {
        comparisons++;
        if (target == curr->data) {
            return comparisons;
        } else if (target < curr->data) {
            printf(",L");
            curr = curr->left;
        } else {
            printf(",R");
            curr = curr->right;
        }
    }
    return -1;
}
```

This is a binary search function for the binary search tree. It has a time complexity of $O(log_2(n))$. The average number of comparisons observed for searching for the magicitems-find-in-bst.txt items is 11.952381. $log_2(666) = 9.379378$. The reason for this discrepancy is that the tree is unbalanced, and many searches require more than 10 comparisons. The way it searches is by iterating down a specific branch until it finds the target. If it hasn't found the target, it will move to the left if the target is less than the current node, or it will move to the right if the target is greater than or equal to the current node.

3 Putting it All Together

3.1 Verifying Graph Files

main.cpp

```cpp
int main(int argc, char* argv[]) {
    //Make a list of all the graph files to use
    vector<string> graph_files;
    if (argc == 1) {
        graph_files.emplace_back("./graphs1.txt");
    } else {
        for (int i = 1; i < argc; i++) {
            struct stat buffer;
            // File exists and is a regular file
            if (stat(argv[i], &buffer) == 0 && (buffer.st_mode & S_IFREG) != 0) {
                graph_files.emplace_back(argv[i]);
            } else {
                fprintf(stderr, "Error - invalid file path: %s\n", argv[i]);
                return 1;
            }
        }
    }

    //...
}
```

To make it easier to run the program with different graph files, I have made it so you can list the files in the arguments. See README.md for details.

3.2 Graphs

main.cpp

```cpp
int main(int argc, char* argv[]) {
    //...

    //Do graph stuff for each graph file
    string line;
    for (const string& file : graph_files) {
        std::ifstream graph_file(file);
        if (!graph_file.is_open()) {
            fprintf(stderr, "Unable to open file %s\n", file.c_str());
            return 1;
        }
        Graph<Matrix> graph_m;
```

```
        Graph<AdjList> graph_al;
        Graph<vector<Vertex>> graph_lo;
        bool init = false;
        int lineNum = 0;
        while (std::getline(graph_file, line)) {
            lineNum++;
            //ignore comments
            if (size_t pos = line.find("--"); pos != std::string::npos) {
                line = line.substr(0, pos);
            }
            //skip empty lines
            if (line.empty()) {continue;}

            //parse tokens
            std::istringstream iss(line);
            std::string command;
            iss >> command;
            if (command == "new") {
                std::string graph;
                iss >> graph;
                if (graph == "graph") {
                    if (init) {
                        print_graphs(&graph_m, &graph_al, &graph_lo);
                    }
                    graph_m = Graph<Matrix>();
                    graph_al = Graph<AdjList>();
                    graph_lo = Graph<vector<Vertex>>();
                    init = true;
                } else {
                    fprintf(stderr, "Error - unrecognized token: %s\nLine number: %d\n", graph.c_str(), lineNum);
                    return 1;
                }
            } else if (command == "add") {
                std::string type;
                iss >> type;
                if (type == "vertex") {
                    if (string vertex; iss >> vertex) {
                        graph_m.add_vertex(vertex);
                        graph_al.add_vertex(vertex);
                        graph_lo.add_vertex(vertex);
                    } else {
                        fprintf(stderr, "Error - failed to parse vertex id as int\nLine Number: %d\n", lineNum);
```

```
                        return 1;
                    }
                } else if (type == "edge") {
                    char dash;
                    if (string vertex0, vertex1; iss >> vertex0 >> dash >> vertex1 && dash == '-') {
                        graph_m.add_edge(vertex0, vertex1);
                        graph_al.add_edge(vertex0, vertex1);
                        graph_lo.add_edge(vertex0, vertex1);
                    } else {
                        fprintf(stderr, "Error - failed to parse vertex id as int\nLine Number: %d\n", lineNum);
                        return 1;
                    }
                } else {
                    fprintf(stderr, "Error - unrecognized token: %s\nLine Number%d\n", type.c_str(), lineNum);
                    return 1;
                }
            } else {
                fprintf(stderr, "Error - unrecognized token: %s\nLine Number %d\n", command.c_str(), lineNum);
                return 1;
            }
        }
        print_graphs(&graph_m, &graph_al, &graph_lo);
        graph_file.close();
    }

    //...
}
```

main.cpp

```
void print_graphs(const Graph<Matrix>* graph_m, const Graph<AdjList>* graph_al, const Graph<vector<Vertex>>* graph_lo) {
    printf("Graph with Matrix:\n");
    graph_m->print();
    printf("\nGraph with Adjacency List:\n");
    graph_al->print();
    printf("\nGraph with Linked Objects - Depth First:\n");
    graph_lo->print_depth_first();
    printf("\nGraph with Linked Objects - Breadth First:\n");
    graph_lo->print_breadth_first();
    printf("\n");
}
```

This chunk of code is responsible for going through each graph file and doing the following:

- Going line-by-line in the text file
- Parsing and tokenizing
- Handling the few possible tokens in a syntax tree

Here are the instructions handled in the syntax tree:

- "new graph" or reaching the end of the file calls the print_graphs function to print the 3 graphs.
- "add vertex #" calls the add_vertex function on the 3 graphs. # represents the vertex ID.
- "add edge # - #" calls the add_edge function on the 3 graphs. The #'s represent the vertex IDs.
- Unrecognized tokens will cause the program to return early with an error.

3.3 Binary Search Tree Insertion and Traversal

main.cpp

```
int main(int argc, char* argv[]) {
    //...

    //Read lines from magicitems.txt and insert into BST
    std::ifstream magicitems("../magicitems.txt");
    if (!magicitems.is_open()) {
        fprintf(stderr, "Unable to open magicitems.txt\n");
        return 1;
    }
    BinarySearchTree bst;
    while (std::getline(magicitems, line)) {
        bst.insert(line);
    }
    magicitems.close();
    printf("\n");

    //print items in order
    bst.depth_first_traverse();
    printf("\n");

    //...
}
```

Lines from magicitems.txt are read and inserted into the BST. Then depth_first_search is called on it.

3.4 Binary Search Tree Searching

main.cpp

```cpp
int main(int argc, char* argv[]) {
    //...

    //Read lines from magicitems-find-in-bst.txt and search BST
    std::ifstream file("./magicitems-find-in-bst.txt");
    if (!file.is_open()) {
        fprintf(stderr, "Unable to open magicitems-find-in-bst.txt\n");
        return 1;
    }
    int comparisons = 0;
    int count = 0;
    while (std::getline(file, line)) {
        int c = bst.search(line);
        if (c == -1) {
            fprintf(stderr, ", Not found after %d comparisons\n", c);
            return 1;
        }
        printf(", Found after %d comparisons\n", c);
        comparisons += c;
        count++;
    }
    file.close();
    printf("Average comparisons: %f\n", static_cast<float>(comparisons) / static_cast<float>(count));

    return 0;
}
```

Items are read from magicitems-find-in-bst.txt and passed into the BST's search function. Comparisons for each search are returned and averaged.