Conner Sommerfield

CS210 – Data Structures

Professor Shen

November 13 2022

Program 4 Part B

Part B – Complexity Analysis (20 points)

Show that the upper bound of the time complexity of your citiesSortedByNumOf_Its_ReachableCities_byTrain implementation would be O(c2 + c * r), where:

1) c is the number of cities (nodes).

2) r is the number of direct self-driving routes (edges) between cities.

You would need to analyze the complexity of your implementation logic for the worst-case scenario as a function of the number of nodes and edges in the graph, then simplify it to the O notation.

Hints: Try to analyze the time complexity for each step of your algorithm, then sum the complexity from all steps, i.e., total time complexity = complexity of inserting nodes to construct the graph, finding the reachable cities from each city node in the graph, then sorting the city nodes by the city name, then by the number of

My main citiesSortedByNumOf_Its_ReachableCities_byTrain function is built with various steps of helper functions which are:

```
unordered_map<string, CityNode> graph =
ConnectedCities::createGraphSkeleton(cities);

    // fill each CityNode directRoute vector using trainRoute city pairs
    ConnectedCities::fillAdjacents(trainRoutes, graph);

    // fill each CityNode reachables (calls buildReachables and DFSHelper
helper functions)
    ConnectedCities::fillReachables(graph);

    // place nodes from hashmap into a vector
    vector<CityNode> answer = ConnectedCities::populateAnswerVector(graph);

    // sort vector by size of CityNode reachables and then by alphabetical
order
    ConnectedCities::sortByReachablePaths(answer);

    return answer;
```

We'll look at these functions individually to find our complexity

Helper function 1: createGraphSkeleton

```
unordered_map<string, CityNode>
ConnectedCities::createGraphSkeleton(vector<string>& cities) {
    unordered_map<string, CityNode> graph;
    for (string city : cities) {
        CityNode node(city); // call constructor for each city node to insert
into graph
        graph.insert({city, node}); // string is key, CityNode object is value
    }
    return graph; // use this graph throughout program
}
```

This function calls the constructor for each node and inserts the resulting CityNode into a hashmap. These two constant operations are performed **for each CityNode C, so our complexity here is O(C)**

Helper function 2: fillAdjacents

```
void ConnectedCities::fillAdjacents(vector<pair<string, string>>& trainRoutes,
unordered_map<string, CityNode>& graph) {

    // trainRoute looks like {SD, LA}, this means for node SD in our map, we
need to add LA as directRoute

    for (pair<string, string> trainRoute : trainRoutes) { // apply above
process to each trainRoute
        CityNode* nodePtr = &graph.at(trainRoute.first); // use first string
of pair to get pointer to node from graph
        nodePtr->addADirectRoutedCity(trainRoute.second); // add the second
string to that node's directRoutes
    }
}
```

This function declares a pointer to a CityNode in our graph and calls the function addDirectRoutedCity (O(1) amortized time for vector push_back function) of that CityNode **for each trainRoute R** in our list of train routes. That means **these constant operations are performed R times, and our complexity is O(R)**

Helper function 3: fillReachables

```
void ConnectedCities::fillReachables(unordered_map<string, CityNode>& graph) {

    // row will be each entry in our graph, LA: CityNode(LA) <- grab CityNode
at row.second at call buildReachables
    for (auto &row : graph) {
        CityNode* nodePtr = &row.second;
        ConnectedCities::buildReachables(nodePtr, graph);
    }
}
```

_____Before seeing buildReachables function_____

This function declares a pointer to a CityNode node in the graph and calls buildReachables for every row in the graph. Since the number of rows in the graph is determined by the number of CityNodes constructed in the createGraphSkeleton function, these operations are performed for **the number of city Nodes C**. However, we don't know the complexity of buildReachables, so let's find that to continue with this complexity analysis.

_____After seeing buildReachables function_____

We saw our buildReachables function has a complexity of r, and the buildReachables is called for each cityNode c, so our complexity here is O(r*c)

```
void ConnectedCities::buildReachables(CityNode* nodePtr, unordered_map<string,
CityNode>& graph) {
    unordered_set<string> visited; // will be passed to DFSHelper

    unordered_set<string> dummyset = ConnectedCities::DFSHelper(nodePtr,
graph, visited);
    nodePtr->setMemo(true); // if a route is explicitly found, its value is
valid

    for (string dummy : dummyset) {      // answer from DFSHelper given as
unordered_set, but the answer needs
        nodePtr->addReachableCity(dummy); // to be a vector, so we must copy
the values over
    }
}
```

_____Before seeing DFSHelper function_____

This function:

- Performs constant operation of declaring an unorderedset
- Calls the DFSHelper function ← We do not know the complexity of this function
- Calls the setMemo function which is a constant operation
- And for each string the dummyset, calls addReachableCity function of the current node which is a constant operation, We have to determine the complexity of DFSHelper to finish this function.

_____After seeing DFSHelper function_____

We determined that DFSHelper has a complexity of O(r). so we can now say that

- Calls the DFSHelper function ← this is an O(r) operation
- For each string in dummyset, call addReachableCity ← this will perform equal to the number of strings in dummyset which is returned by DFSHelper, so this is also an O(r) operation

This means our overall complexity will be of O(r)

*** I'm going to omit the comments so this is easier to read ***

```
unordered_set<string> ConnectedCities::DFSHelper(CityNode* nodePtr,
unordered_map<string, CityNode>& graph, unordered_set<string> visited)
{

    visited.insert(nodePtr->getCity());

    if (nodePtr->getReachableDummies().size() != 0 && nodePtr-
>getMemo() == true) {
        return nodePtr->getReachableDummies();
    }

    nodePtr->addReachableDummy(nodePtr->getCity());


    for (string child : nodePtr->getDirectRoutedCities())
        if (visited.count(graph.at(child).getCity()) == 0) {
            unordered_set<string> childReachables =
ConnectedCities::DFSHelper(&graph.at(child), graph, visited);
            for (string childReachable : childReachables) {
                nodePtr->addReachableDummy(childReachable);
            }
        }
        else {
           nodePtr->setMemo(false
    }

    for (string child : nodePtr->getDirectRoutedCities()) {
        if (graph.at(child).getMemo() == false) {
            nodePtr->setMemo(false);
        }
    }

    return nodePtr->getReachableDummies();
}
```

This function:

- Inserts the current node into the visited list; this is a constant operation
- In worst case, our memo conditional is skipped and we go on to the rest of the function (constant)
- Now we call the addReachableDummy function which performs in amortized O(1) time (vector push_back)
- Now we enter a for loop that will execute for each child (in other words, for each adjacent node) of the current node. **In the worst case, the current node will have each other node in the graph as a child, and it will have to call this recursive function on each**

one of those children that also have every other node as a child. This means that in the worst case, the recursive call will be called equal to the maximum number of edges in the graph, which would be equal to (c*(c-1))/2

- After returning our vector given from the recursive function, we will have to call the addReachableDummy function (determined to be a constant operation) for each child, and we determined the max number of children returned by the recursive function is c-1, so our worst case complexity here is O(c-1)
- We have another loop with constant operations proportional to the number of children so our worst case once more is O(c-1) complexity
- Then we have one last constant operation as our return statement.

So in our worst case, our function will have a complexity proportional to the max number of edges r of the graph.

Helper function 4: populateAnswerVector

```
vector<CityNode> ConnectedCities::populateAnswerVector(unordered_map<string,
CityNode>& graph) {
     vector<CityNode> answer;

     // for each CityNode in graph, place in answer vector
     for (auto &row : graph) {
        CityNode* nodePtr = &row.second; // row.second refers to each node of
graph
        answer.push_back(*nodePtr); // use pointer so you don't just insert a
copied node with no info
    }
    return answer;
}
```

This function simply declares a vector and calls the vector push_back function for each CityNode in the graph, so we have O(c) complexity.

Helper function 5: sortReachablePaths

```
void ConnectedCities::sortByReachablePaths(vector<CityNode>& cityNodes) {
    std::sort
    (
        cityNodes.begin(),
        cityNodes.end(),
        [](CityNode& a, CityNode& b)
        {
```

```
            int size1 = a.getReachableCities().size(); // store number of
reachable cities for each node
            int size2 = b.getReachableCities().size();

            if ( size1 == size2 )                  // if same number of
reachable cities
                {return a.getCity() < b.getCity();} // sort by the city name
aphabetically
            else
                {return size1 > size2;} // else sort by size of
reachableCities
        }
    );
}
```

This function simply calls the std::sort function given by C++ standard library which has **average case linearithmic (n log n) time, and our vector here contains all the city nodes c, so with n = c, we know this performs in clogc time.**

Conclusion

So our five functions have c + r + (c*r) + c + clogc time complexity. Let's just flip around the order so:

c + r + c + logc + (c*r) time complexity.

We know from our DFSHelper analysis that the worst case for r is (c*(c-1))/2 or (c^2-c)/2 or overall O(c^2) time complexity.

Our c^2 term will make the c and clogc terms negligible.

So instead of c + r + c + clogc (+ (c*r)), we can simplify this by writing c^2 + (c*r)

Therefore our time complexity is O(c^2 + c * r)