Assignement 5 — Graph

**Part of the source code is included here, but all of it is in the zip folder*

Exercise 1

Use provided C++ skeleton to insert your code.

A.

Define Graph, which stores an undirected graph using adjacency list, where each node stores a CityName (string) and each edge has an integer weight (distance between two cities).

Implement the following functions in Graph:

a. bool IsThereTripletClique(): returns true if there are three nodes in the graph that are all connected to each other. E.g., a,b,c, with edges (a,b),(b,c),(a,c)
b. bool isGraphConnected(): returns true if graph is connected
c. int GetMinDistance(string city1,string city2): returns shortest path distance between city1 and city 2. Hint: You may use Dijkstra Algorithm.
d. [extra credit] int LongestSimplePath(): returns length of longest simple path (no cycle allowed)

B.

What is the big-Oh complexity of your functions above if graph has n nodes and m edges?

Here is the code for the isThereTripletClique() function:

```
bool Graph::IsThereTripletClique(Node* origin, Node* current,int depth){
  if(depth >3) return false;
  else if(depth == 3) return current == origin;
  else{
    for (vector<Edge*>::iterator it=current->adjacentsList()-
>begin(); it != current->adjacentsList()->end();it++)
      if(IsThereTripletClique(origin,(*it)->getNode(),depth+1))
        return true;
    return false;
  }
}

bool Graph::IsThereTripletClique(){
  for (map<string,Node* >::iterator it=graph->begin(); it != graph-
>end();it++)
    if(this->IsThereTripletClique(it->second,it->second,0))
      return true;
  return false;
```

The main function is the one with no parameters. It iterates over all the n nodes in the graph and calls the recursive helper function with three parameters (origin, current, depth). This loop will run on n iterations on the worst case, so its overall time complexity depends on the helper function performed n times. The first two branches of the function are executed in constant time, and the last one loops over the adjacent nodes of the current one and calls recursively the function with increased depth. The first thing to notice is that the depth of the recursion tree will never exceed 3, since it's the number of nodes connected that we are searching for. The main issue to evaluate the time complexity, is the degree d each node can have. We chose a boundary of $n-1$ for that number, since a node can't have more than $n-1$ neighbors. So, at each level of the three, each neighbor will perform a maximum of n-1 operations. That leaves us with n for the outer loop of the function, and $(n-1)*(n-1)*(n-1)$ operations. The time complexity of our function in the worst case is thus $O(n^4)$. Our space complexity is constant since the call stack depth is constant and freed for the other recursive calls, and we are allocating constant memory in each call.

Here is the code for the GetMinDistance function:

```
 int Graph::GetMinDistance(string city1, string city2){
   map<string,Node* >::iterator it1,it2;
   it1 = graph->find(city1);
   it2 = graph->find(city2);
   Node* start = (*it1).second;
   Node* end = (*it2).second;

   this->DijkstraShortest(start);

   return end->getMinDistance();
 }
void Graph::DijkstraShortest(Node* start){
   // even if the nodes are constructed with infity distance, we miust re run
 that part in case nodes are added
   // Iterate each node
   vector<Node*> unvisited_queue;
   map<string,Node* >::iterator it;
   for(it=graph->begin(); it != graph->end();it++) {
      Node* n = (*it).second;
      n->setMinDistance(INT16_MAX);
      n->setPredShort(0);
      unvisited_queue.push_back(n);
   }
   // start city to 0 distance
   start->setMinDistance(0);
   // process
   while(!unvisited_queue.empty()){
     // visit vertex with min_distance
     Node* currV = findMinAndPop(&unvisited_queue);
     // get adjacents
     vector<Edge* >* adjVect = currV->adjacentsList();
```

```
    for (Edge* adjE : *adjVect){
      Node* adjV = adjE->getNode();
      int edgeWeight = adjE->getWeight();
      int alternativeDistance = currV->getMinDistance() + edgeWeight;

      // if shorted path from start to adjV is found, update predShort and m
in_distance of adjV
      if(alternativeDistance < adjV->getMinDistance()){
        adjV->setMinDistance(alternativeDistance);
        adjV->setPredShort(currV);
      }
    }
  }

 }
```

For the GetMinDistance() function, Dijkstra's algorithm is determining the time and space complexity. For Dijkstra, we begin by creating a queue of unvisited nodes, that we fill by iterating over the n nodes. Thus, for now, it's $O(n)$ in both time and space. Then, we traverse all the items in the queue in a while loop, performing the following operations. We find the min of the queue and pop it, this function traverse the remaining elements in the queue, which will be $n-1, n-2, n-3 \dots 1$ until the queue is empty and erases the min value in the vector, and uses constant time and space otherwise. So that part runs overall in $n(n-1)/2 \sim O(n^2)$. The next part gets the neighbors of the current node and updates the min distance if needed. The update part is done in constant time and space, but there like for the other function, there is at most d degrees for a node, which is lower than n, se we can choose n as a boundary. That part of the while loop runs also in $O(n^2)$. It leaves the overall time complexity to $O(n^2)$, and the space complexity to $O(n)$, with the queue created with the n nodes.

```
bool Graph::isGraphConnected() {
    //Initialize connected nodes with the first node
    vector<string> visited = vector<string>();
    visited.push_back(this->graph->begin()->first);
    this->graph->begin()->second->setVisited(true);
    //traverse currently connected node
    for(size_t i = 0; i < visited.size(); i++){
      Node* node = this->graph->at(visited.at(i));
      //Check what nodes are connected to it to add them to connected nodes
      for(vector<Edge*>::iterator itn = node->adjacentsList()-
>begin();itn!=node->adjacentsList()->end();itn++) {
        if(!(*itn)->getNode()->isVisited()){
          visited.push_back((*itn)->getNode()->getPayload());
          (*itn)->getNode()->setVisited(true);
        }
      }
    }
    return visited.size() == this->graph->size();
 }
```

For the isGraphConnected() function, we initialize a vector of visited nodes, filled initially with the first element of the graph in the adjacency list. It will serve us to iterate only over the connected neighbors of our current node. We iterate over the visited vector, and for each node, we get its neighbors and if it hasn't been visited before, we reset its visited bool and add it to the visited vector, similarly to a Breadth First Search. That way we only explore the connected part of the graph, and we check at the end if the size of the visited vector is the same as the size of the graph. That means that in the worst-case scenario, our space complexity will be $O(n)$. The first operations are done in constant time, and the size of the visited vector is n in the worst case. In the loop iterating over this vector, we again get the d degrees (neighbors of the current node) and set their visited attribute to true, if not already visited. The number neighbors is bounded by again bounded by n. We end up with a worst time complexity of $O(n^2)$.

Here is the code for the longestSimplePath():

```
// longest distance you can travel (weights) to with no cycle
 int Graph::longestSimplePath() {
   int max_distance = 0;
   map<string,Node* >::iterator it;
   for(it=graph->begin(); it!=graph->end(); it++) {
      Node* n = (*it).second;
      int curr_distance = this->getMaxDistance(n, 0);
      if (curr_distance > max_distance)
          max_distance = curr_distance;
   }
   return max_distance;
 }

 int Graph::getMaxDistance(Node* n, int currDistance){
    n->setVisited(true);
    int max_distance = currDistance;
    for (Edge* adjE : *(n->adjacentsList())){
        Node* adjV = adjE->getNode();
        int pathDistance = currDistance;
        if(!adjV->isVisited())
            pathDistance = getMaxDistance(adjV, currDistance + adjE-
>getWeight());
        max_distance = max(pathDistance, max_distance);
    }
    n->setVisited(false);
    return max_distance;
 }
```

The algorithm is calculating the most weighted (longest in city distance) path in the graph, with no cycle. It picks every node in the graph, and calls the getMaxDistance from that node, with a starting distance of 0. Every instruction in the outer for loop is constant except the call to that function, thus calld n times. The getMaxDistance() finds all acyclic path for a given node and returns the max distance of all of the found path. It calls itself recursively on each degree of the current node but is taking account if the next node has been visited or not. It's thus hard to generalize its complexity because it depends highly on the average degrees that nodes have.

Since we call the getMaxDistance() function on each node of the graph, it ensures that our complexity will be $n * time\_complexity(getMaxDistance())$. If all node have the same average degree, then we would check $(d-1), (d-2), \dots, 1$ degree at every recursive call, since as we go deeper into the recursion tree, the nodes are more and more getting visited. Thus, the overall time complexity for this algorithm approach would be $O(n * d!)$ in the case that all nodes have the same degree.

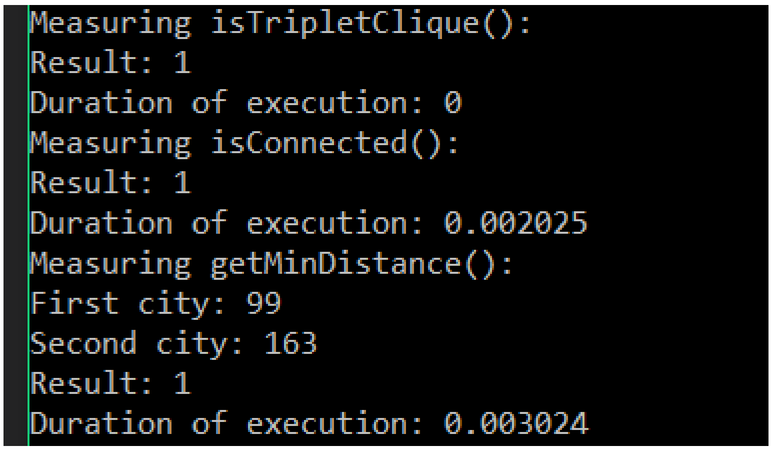
C.

Test your functions. Write code to create a random graph of 100 nodes, with 500 random edges with weight 1, 500 random edges with weight 2 and 500 random edges with weight 3. (For function in A(d) use a smaller graph if too slow.)

Measure the time of each function.

Few observations:

For n = 100 nodes and 3 times 500 edges with weight 1, 2 and 3, the first three algortihms executed too fast to execute to be able to measure their time. It might be due to the random arrangement of the graph, and so that there is a large cut of branches compared to theory. The findTripletClique could occur the first iteration and it's the same for the getMinDistance() function. The findLongestPath didn't completed because it's runtime wouldn't allow that, even for best case scenarios. So, we continued to test the 3 first algorithms on larger number of nodes and with max number of edges: $n * (n-1)/2$ for an undirected graph. For n=200 nodes, here is the result:


```
Measuring isTripletClique():
Result: 1
Duration of execution: 0
Measuring isConnected():
Result: 1
Duration of execution: 0.002025
Measuring getMinDistance():
First city: 99
Second city: 163
Result: 1
Duration of execution: 0.003024
```

The runtime of isConnected and getMinDistance is quite similar both in practice and in theory ($O(n^2)$), and the isTripletClique() happens to be the more biased because actually, with max number of edges, the algorithms exits on the first iteration, and thus is really fast, in fact for this configuration, runs in constant time. In fact, it's hard to find a configuration in which we can assess its runtime. For the longestPath algorithm, when we try with n=10 nodes and m=30 edges, the runtime is 0.01496s and when we try with n=10 nodes and m=45 edges, the runtime is 0.282237s. We have only multiplied the number of edges by 1.5, but the runtime multipled by 20 which clearly indicated a runtime that is not polynomial but factorial.