UCR - CS 010C

Rohan Behera

12/7/20

Assignment 4

Deliverables: Create a single PDF file that contains your answers to the questions. Then create a zip file that contains this PDF file along with all your code source files. Submit this zip file on iLearn.

Deadline: 12/11/2020 11:59 pm.

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 a double weight (distance between two cities).

   Implement the following functions in Graph:

   a. bool hasTripletClique(): 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), and (a, c)

```
bool Graph::hasTripletClique() const {
  if (nodes_.size() < 3) return false;
  // To do
    for (auto it1 = nodes_.begin(); it1 != nodes_.end(); ++it1) {
    std::string key1 = it1->first;
    std::set<Node *> node1_neighbours = it1->second->getNeighbors();
    for (std::set<Node*>::const_iterator it2 = node1_neighbours.begin(); it2 !=
node1_neighbours.end(); ++it2) {
      std::string key2 = (*it2)->getID();
      std::set<Node*> node2_neighbours = (*it2)->getNeighbors();
      for (std::set<Node*>::const_iterator it3 = node2_neighbours.begin(); it3 !=
node2_neighbours.end(); ++it3) {
        std::string key3 = (*it3)->getID();
        if (key1 == key3 ) {
          return true;
        }
        else return false;
      }
    }
  }
  return false;
}
```

   b. bool isConnected(): returns true if graph is connected

```
bool Graph::isConnected() const {
   return connected_;
}

/**
 * Checks if all the nodes in the graph is connected
 * @param s
 */
void Graph::dfs() {
   size_t max_nodes = getNoOfNodes();
 //  std::cout << "inside dfs " << max_nodes << std::endl;
   int ctr = 0;
   std::unordered_map<std::string, bool> visited_list(max_nodes);
   for(auto it : nodes_) {
      //it.second->setVisited(false);
      visited_list.emplace(it.first, false);
   }
   // std::cout << "First Node: " << nodes_.at(0)->getID() << std::endl;
 //  std::cout << " after dfs before connected check " << std::endl;
   Node* firstNode = nodes_.begin()->second;
   recursive_dfs(firstNode, visited_list);

   // if any of the value in the visited_list is false set connected_ = false, otherwise set to true
   int connected_ctr = 0;
   for (auto it = visited_list.begin(); it != visited_list.end(); ++it){
      if (it->second == true) {
         connected_ctr++;
      }
   }
   if (max_nodes == connected_ctr)
      connected_ = true;
   else connected_ = false;
   firstNode = NULL;
   visited_list.clear();
}
```

c. double getMinDistance(string city1,string city2): returns the shortest path distance between city1 and city 2. Hint: You may use Dijkstra Algorithm.

```
double Graph::getMinDistance(const std::string &nid1,
                             const std::string &nid2) const {
   assert(nodes_.size() >= 2);  // Must have at least 2 nodes
   // To do
   Node* src = nodes_.at(nid1);
   Node* dest = nodes_.at(nid2);

   int source_dist =  std::distance(nodes_.begin(),nodes_.find(nid1));
   int destination_dist =  std::distance(nodes_.begin(),nodes_.find(nid2));
```

```
    if (source_dist == destination_dist) {
      return 0;
    }

    int max_nodes = getNoOfNodes();

    bool* visited = new bool[max_nodes];
    // set source node with infinity distance
    // except for the initial node and mark
    // them unvisited.
    for(int i = 0; i < max_nodes; i++)
    {
      visited[i] = false;
    }
    // Distance of source vertex from itself is always 0
    double min_distance = 0;
    std::set<Node*> neighbours = src->getNeighbors();
    for (auto it = neighbours.begin(); it != neighbours.end(); ++it) {
      double new_dist =  std::distance(nodes_.begin(),nodes_.find((*it)->getID()));
      std::set<Edge*> adjacencyList =  src->getAdjacencyList();

      for (auto it = adjacencyList.begin(); it != adjacencyList.end(); ++it) {
        double weight = 0;
        if ((*it)->getNode()->getID() == dest->getID()) {
          weight = (*it)->getWeight();
          new_dist = new_dist+weight;
        }
      }
      if (new_dist < source_dist) {
        min_distance = new_dist;
      }
    }
    return min_distance;
  }
```

d. [extra credit] double getLongestSimplePath(): returns length of longest simple path (no cycle allowed)

```
  double Graph::getLongestSimplePath() const {
    assert(nodes_.size() >= 1);  // Must have at least 1 node
    // To do
    getLongestSimplePathHelper(nodes_.begin()->first);
    return 0.0;
  }

  double Graph::getLongestSimplePathHelper(const std::string &nid1) const {
    Node* src = nodes_.at(nid1);
```

```
    int source_dist =  std::distance(nodes_.begin(),nodes_.find(nid1));
    int max_nodes = getNoOfNodes();

    bool* visited = new bool[max_nodes];
    // set source node with infinity distance
    // except for the initial node and mark
    // them unvisited.
    for(int i = 0; i < max_nodes; i++)
    {
      visited[i] = false;
    }
    // Distance of source vertex from itself is always 0
    double longestDistance = 0;
    std::set<Node*> neighbours = src->getNeighbors();
    for (auto it = neighbours.begin(); it != neighbours.end(); ++it) {
      int new_dist =  std::distance(nodes_.begin(),nodes_.find((*it)->getID()));
      // find weight of the edge that connects source to it(neighbor)
      // how to find the edge between the two nodes
      std::set<Edge*> adjacencyList =  src->getAdjacencyList();
      //Edge* edge = adjacencyList. .find(dest->getID());
      for (auto it = adjacencyList.begin(); it != adjacencyList.end(); ++it) {
        double weight = 0;
        if ((*it)->getNode()->getID() == src->getID()) {
          weight = (*it)->getWeight();
          new_dist = new_dist+weight;
        }
      }
      if (new_dist > source_dist) {
        longestDistance = new_dist;
      }
    }
    return longestDistance;
  }
```

B. What is the big-Oh complexity of your functions above if graph has n nodes and m edges?
The isConnected() function uses depth-first search recursively so it's big-Oh complexity is O(e+v). The hasTripletClique() function uses two nested for loops to traverse through node1 and node2's neighbor list so it's big-Oh complexity is $O(v^3)$. The getMinDistance(city1, city2) function uses Dijkstra's Algorithm in an adjacency list, so the time complexity is O(e+v). The getLongestSimplePath() function's big-Oh complexity is O(e+v) because even though it uses an adjacency list, we are computing the longest path instead of the shortest path this time.

C. Test your functions. Write code to create a random graph of 100 nodes, with 500 random edges with weight 1.0, 500 random edges with weight 2.0 and 500 random edges with weight 3.0. (For function in A(d) use a smaller graph if too slow.) Measure the time of each function in nanoseconds or microseconds.

- Assume there can be at most 1 edge between 2 nodes.
- Assume there is no self-loop (edge from one node to itself).

Functions finished testing:

bool hasTripletClique():

```
start_time = std::chrono::high_resolution_clock::now();
std::cout << "Testing hasTripletClique() " << graph.hasTripletClique() << std::endl;
end_time = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end_time - start_time);
std::cout << "Time taken by hasTripletClique() in nanoseconds: " << duration.count() << " ns" <<
std::endl << std::endl;
tripleCliqueTime = duration.count();
```

bool is Connected():

```
start_time = std::chrono::high_resolution_clock::now();
std::cout << "The Graph is (1 means connected, 0 means not connected): " << graph.isConnected() <<
std::endl;
end_time = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end_time - start_time);
std::cout << "Time taken by isConnected() in nanoseconds: " << duration.count() << " ns" << std::endl;
isConnectedTime = duration.count();
```

double getLongestSimplePath():

```
start_time = std::chrono::high_resolution_clock::now();
std::cout << "Length of longest simple path: " << graph.getLongestSimplePath() << std::endl;
end_time = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end_time - start_time);
std::cout << "Time taken by getLongestSimplePath() in nanoseconds: " << duration.count() << " ns" <<
std::endl;
longSimplePathTime = duration.count();
```

double getMinDistance():

```
auto start_time = std::chrono::high_resolution_clock::now();
for (auto it : NODE_PAIRS) {
   std::cout << "The shortest path distance between city 1 and city 2 is: " <<
graph.getMinDistance(NODE_NAMES[it.first.index1], NODE_NAMES[it.first.index2]) << std::endl;
}
auto end_time = std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end_time - start_time);
std::cout << "Time taken by getMinDistance(string city1, string city2) in nanoseconds: " <<
duration.count() << " ns" << std::endl << std::endl;
getMinDistanceTime = duration.count();
```