A Project Report

ON

Implementation of Graphs

BY

Rohan Agarwal - 1PI13CS124
Sameer Ravindra Kulkarni - 1PI13CS132
Anirudh Agarwal - 1PI13CS199

Guide
Prof. N.S.Kumar

PESIT-CSE
Bangalore

January 2016 – May 2016

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
PES INSTITUTE OF TECHNOLOGY
(an autonomous institute under VTU)
100 FEET RING ROAD, BANASHANKARI III STAGE,
BANGALORE-560085

Introduction :

A graph data structure consists of a finite (and possibly mutable) set of *vertices*, together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as *edges*, *arcs*, or *lines* for an undirected graph and as *arrows*, *directed edges*, *directed arcs*, or *directed lines* for a directed graph.

Operations (Graphs) :-

- Graph can support bfs and dfs traversals.
- Comparison of two graphs using bfs and dfs iterator
- Topological Sorting of given graph object
- Finding a node in graph
- All basic functionalities of a class
- Dijkstra's algorithm implementation

- Graph internally uses two hidden classes namely `Node` and `iterator`

Iterator :-

- It is hidden as inner class of Graph
- Is a generic iterator for both dfs and bfs
- Uses a single generic container as stack and queue
- Can support post and pre increment, is a forward iterator
- Also supports dereferencing , equals and not equals operations.

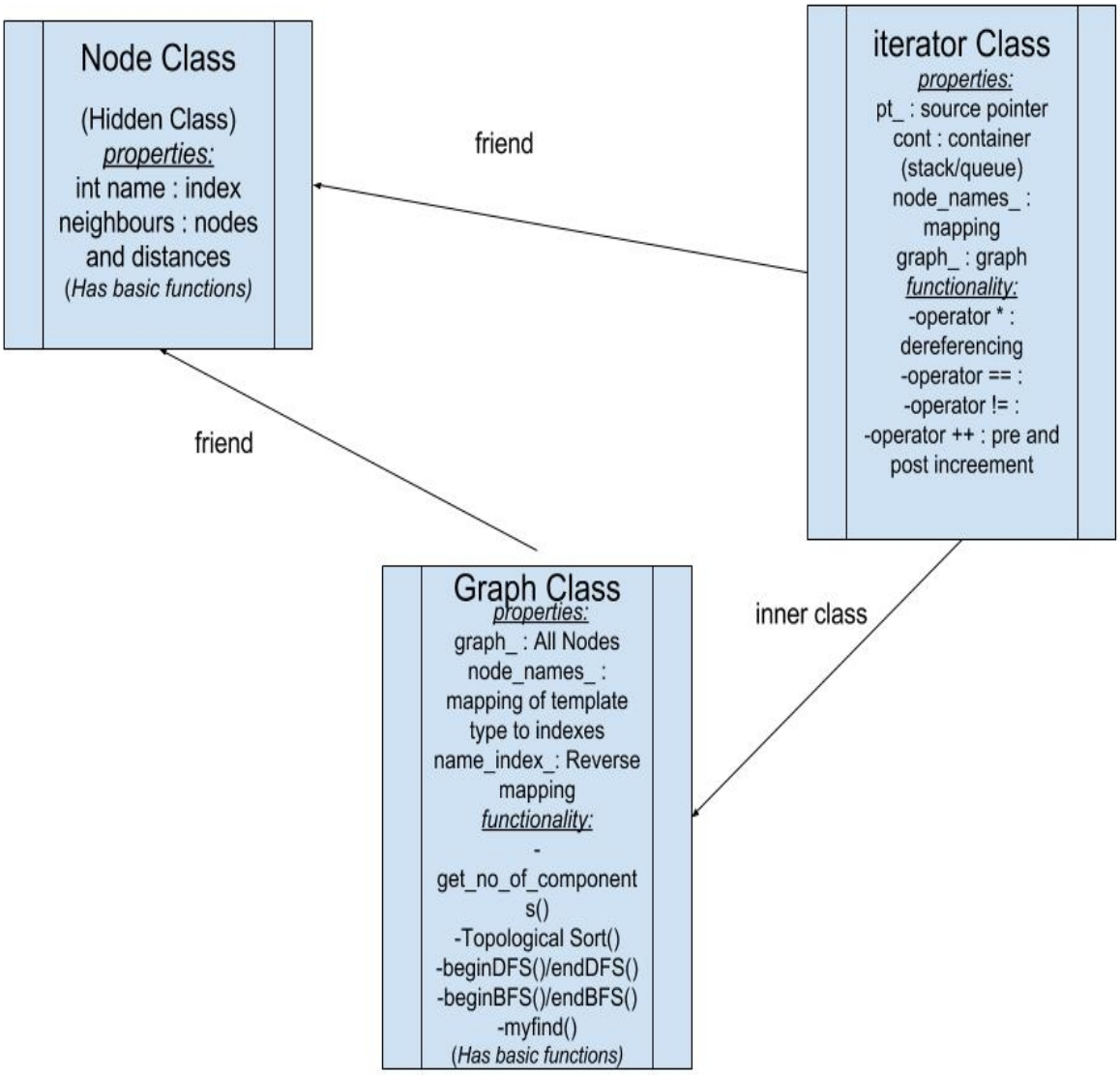
Node :-

- It is hidden as friend of Graph
- Contains name of node and neighbours along with their distances.
- Supports all basic functions of class

Client :-

- First needs to provide no of nodes in the graph
- Then a simple mapping of nodes i.e list of name of nodes
- An adjacency matrix needs to be provided after this to create a graph object.
- The graph object now created can be used to implement any of the features listed above.

Graph Structure :



Our implementation involves a graph class that will be exposed to the client to make graphs. Two hidden classes are used by graph class, `Node` and `iterator` classes, and their interrelationship is shown in the above diagram.

Implementation :

The Graph data structure is implemented using the Node class as the basic element. The Node class encapsulates the data provided by the user (eg. value of the node) and implementation specific details (eg. map of all the neighbours and an Identifier associated with each instance of Node class) within it. It defines all the six basic functions of a C++ class.
A single iterator class implements both Breadth First Search and Depth First Search traversals. The begin() and end() calls wrapped inside the function calls begin_bfs()/begin_dfs() and end_bfs()/end_dfs() by the user decide the type of traversal that takes place. The iterators of the class support post/pre increment, dereferencing, comparison operator thus implementing a forward iterator type.
Using the object of the Graph class the user can thus manipulate the graph with the functions provided by the class previously constructed.
Graph class functions

1. iterator begin_bfs(string str); / iterator begin_dfs(string str);
   => functions that calls make_itr(); function for BFS/DFS accordingly.
   => They initialize pointer to Node with the value passed as an argument.
2. iterator end_bfs(string str="$$$"); / iterator end_dfs(string str="$$$");
   => functions that calls make_itr(); function for BFS/DFS accordingly.
   => They initialize pointer to Node with the value passed or with a sentinel value indicating a dummy node.
3. iterator make_itr(int c, string str);
   => A wrapper method that returns that an object of BFS iterator or DFS iterator based on the function call begin_bfs()/begin_dfs() or end_bfs()/end_dfs().

4. vector<string> Graph::topo_sort(Graph::iterator,Graph::iterator)
   Returns the topological sort of the vertices iterated over in the range (of iterators) passed to it.
5. Graph::iterator myfind(Graph::iterator first, Graph::iterator last, string val)
   Returns iterator to the Node found in the range (of iterators) passed to it. If the given Node is not found it returns an iterator to the end.
6. int Graph::get_number_of_components(string city)
   Returns the number of components i.e. number of disconnected sub-graphs in the graph.
7. void Graph::dijkstra(string src, int N)
   Returns shortest paths from the Node passed as the argument to all the other Nodes in the graph.

Forward Iterator :

Forward Iterator is generic in the sense both dfs and bfs traversals can be done by using the same Iterator class. It has a generic container that can act as both stack and queue and an internal flag to indicate if the object is supposed to be a bfs iterator or a dfs iterator object.

Variables used by Iterator:

Node* pt_;

- Pointer to a Node instance

vector<Node> graph_temp_;

- A reference pointing to all the nodes in graph

bool* visited;

- To keep track of nodes during traversal

list<int> cont;

- Container for traversal

vector<string> node_names_;

- Mapping between names and index

iterator(Node sc, vector<Node> graph_temp, vector<string> node_names,int c);*

- *Constructor of iterator for dfs and bfs traversal*

*string operator *();*

- *Dereferencing operator*

bool operator==(const iterator& rhs);

- *Checks for equality*

bool operator!=(const iterator& rhs);

- *Checks for inequality*

iterator operator++();

- *Pre increment operator*

iterator operator++(int);

- *Post increment operator*

iterator begin_bfs(string str);

- *Iterator to given node (Source node)*

iterator end_bfs(string str="$$$");

- *Iterator to given node (Destination node)*

iterator begin_dfs(string str);

- *Iterator to given node(Source node)*

iterator end_dfs(string str="$$$");

- *Iterator to given node (Destination node)*

iterator make_itr(int c, string str);

- *Common method to return iterator object*

Test Results :

We have tested program on inputs varying from N=4 to N=100.
INPUT :

5 Mumbai Delhi Chennai Bangalore Belgaum 0 1 1 1 -1 1 0 1 1 -1 1 1 0 1 -1 1 1 1 0 -1 -1 -1 -1 -1 0	9 Mumbai Delhi Chennai Bangalore Belgaum Kanpur Jaipur Cochin Pune 0 4 0 0 0 0 0 8 0 4 0 8 0 0 0 0 11 0 0 8 0 7 0 4 0 0 2 0 0 7 0 9 14 0 0 0 0 0 0 9 0 10 0 0 0 0 0 4 0 10 0 2 0 0 0 0 0 14 0 2 0 1 6 8 11 0 0 0 0 1 0 7 0 0 2 0 0 0 6 7 0

Output :

- Operations related to graph class
  Created graph object g
  Copy constructor h<-g
  Move constructor i<-h
  Copy Assignment j<-i
  Move assignment k<-j

- BFS : Belgaum -> Chennai : Belgaum Mumbai Delhi

- DFS : Chennai -> *No destination : Chennai Bangalore Delhi Mumbai Belgaum

- Topo sort : Belgaum -> Chennai : Bangalore Mumbai Belgaum

- Compare graph :
  Compare graph : (g('Belgaum') == i('Belgaum') -> 1
  Compare graph : (g('Belgaum') == i('Bangalore') -> 0

- Dijkstra matrix
  Mumbai To Mumbai --> 0
  Mumbai To Delhi --> 4
  Mumbai To Chennai --> 12
  Mumbai To Bangalore --> 19
  Mumbai To Belgaum --> 21
  Mumbai To Kanpur --> 11
  Mumbai To Jaipur --> 9
  Mumbai To Cochin --> 8
  Mumbai To Pune --> 14