```
//==============================================================
// graph.cpp
// This file contains excerpts from the Graph, Vertex
// and Edge classes.
//==============================================================

//==============================================================
// VERTEX
//==============================================================

class Vertex
{
public:
  int id;
  Vertex* pred;
  List<Vertex> adj_list;
  char color;
  int disc;
  int fin;
  int key;

  bool    operator<     (const Vertex &x) {return this->key <= x.key;}
  bool    operator>     (const Vertex &x) {return this->key > x.key;}
  bool    operator==    (const Vertex &x) {return this->id == x.id;}

  Vertex& operator= (const Vertex& v);

          Vertex        (int name);
          Vertex        (Vertex* v);
          ~Vertex       (void);
private:
  friend ostream& operator<< (ostream& os, const Vertex& v)
  {
    os << "Vertex ID: " << v.id << endl;
    return os;
  }
};

//==============================================================
// Default Constructor
//==============================================================
Vertex::Vertex(int name)
{
  id = name;
  pred = NULL;
  color = ’w’;
  disc = INT_MAX;
  fin = 0;
  key = INT_MAX;
}

//==============================================================
// Assignment Operator
//==============================================================
Vertex& Vertex::operator= (const Vertex& v)
{
  this->id = v.id;
  this->pred = v.pred;
  this->adj_list = v.adj_list;
  this->color = v.color;
  this->disc = v.disc;
  this->fin = v.fin;
  return *this;
}
//=================================================================
// EDGE
//=================================================================
class Edge
{
public:
  int u;    //start Vertex
  int v;    //end Vertex
  int weight;     //weight (u,v)

  Edge()
  {
    u = -1;
    v = -1;
    weight = 0;
  }

  ˜Edge()
  {
  }
};
//=================================================================
// GRAPH
//=================================================================
class Graph
{
public:
          Graph          (string filename);
          Graph          (const Graph& g);
          ˜Graph         (void);
  Graph   operator=      (const Graph& g);
  void    dfs            (void);
  bool    cycle          (void);
  void    Prim           (int root);


private:
  List<Vertex>  graph;
  List<Edge>    edges;
  bool          cycles = false;

  void    dfs_visit       (Vertex &u, int timee);
  bool    pqHelp          (Vertex s, MinPriorityQueue<Vertex> pq);
  int     findEdge        (Vertex u, Vertex v);
};

//=================================================================
//default constructor
//Pre-Condition:
//  -file with matrix representation of a graph
//Post-Condition:
//  -a graph
//=================================================================
Graph::Graph  (string filename)
{
  ifstream file;
  file.open(filename);                              //open file
  string line;
  getline(file, line);
  istringstream buffer(line);                 //read in number of
  int num_vert;                                         //vertices
  buffer >> num_vert;
  for (int i = 0; i < num_vert; i++)
  {
    Vertex *v = new Vertex(i);
    graph.append(v);                //append all vertices to graph
  }
  for (int j = 0; j < num_vert; j++)               //iterates rows
  {
    int srch_pt = 0;
    getline(file, line);
    for (int k = 0; k <num_vert; k++)
    {
      int space = line.find(" ", srch_pt);   //read in connections
      int weight;
      istringstream buffer (line.substr(srch_pt, space-srch_pt));
      buffer >> weight;
      srch_pt = space+1;

      if (weight != 0)
      {
        Edge *e = new Edge();                    //creates edges
        e->u = graph[j]->id;
        e->v = graph[k]->id;
        e->weight = weight;
        edges.append(e);
        graph[j]->adj_list.append(graph[k]);
      }
    }
  }
  //cout << graph;
  file.close();
}

//==============================================================
//dfs - depth first search
//Pre-Conditions
//  -graph
//Post-Conditions
//  -traversed graph
//==============================================================
void Graph::dfs (void)
{
  if (graph.length()==0)                //throw error if empty
    throw EmptyError();
  for (int i = 0; i < graph.length(); i++)
  {
    graph[i]->color = ’w’;                 //set all to white
  }
  int timee = 0;
  //cout << timee << endl;
  for (int i = 0; i < graph.length(); i++)
  {
    if (graph[i]->color == ’w’)
    {
      dfs_visit(*(graph[i]), timee);            //visit vertex
    }
  }
}
//==============================================================
//dfs_visit - depth first search
//==============================================================
void Graph::dfs_visit (Vertex& u, int timee)
{
  timee += 1;
  u.disc = timee;
  u.color = ’g’;                                 //discover
  cout << "Visiting: " <<u << endl;
  for (int i = 0; i < u.adj_list.length(); i++)
  {
    if (u.adj_list[i]->color == ’g’)
    {
      cycles = true;
    }
    if (u.adj_list[i]->color == ’w’)
    {
      u.adj_list[i]->pred = &u;              //set predecessor
      dfs_visit(*u.adj_list[i], timee);//visit adjacent vertices
    }
  }
  u.color = ’b’;               //all adjacent vertices visited
  timee += 1;
  u.fin = timee;
}

//==============================================================
//Prim’s algorithm -- we tried but it’s not happening
//Pre-Conditions
//  -undirected weighted graph
//Post-Conditions
//  -MST of graph
//==============================================================
void Graph::Prim (int root)
{
  MinPriorityQueue<Vertex> pq;                 //create min pq
  for (int i = 0; i < graph.length(); i++)
  {
    if (graph[i]->id != graph[root]->id)
    {
      graph[i]->key = INT_MAX;
      graph[i]->pred = NULL;
      pq.insert(graph[i]);             //insert vertices into pq
    }
  }
  graph[root]->key = 0;
  pq.insert(graph[root]);                   //insert root into pq
  cout << pq << endl;

  while (!pq.empty())
  {
    Vertex *u = pq.extractMin();          //find minimum weight
    cout << "MIN: " << u->id << endl;
    for (int j = 0; j < u->adj_list.length(); j++)
    {
      Vertex *v = u->adj_list[j];
      //cout << "weight of " << *v <<" " <<  v->key << endl;
      if (v->key > u->key && findEdge(u,v) < v->key)
      {

        v->pred = u;                      //update pred and key
        v->key = findEdge(u,v);
        //cout<<"reset key of "<<v->id<<"to "<<v->key<<endl;
      }
    }
  }
  for (int k = 0; k < graph.length(); k++)
  {
    cout << k << " weight: " << graph[k]->key <<graph[k]->pred <<endl;
  }
}
```