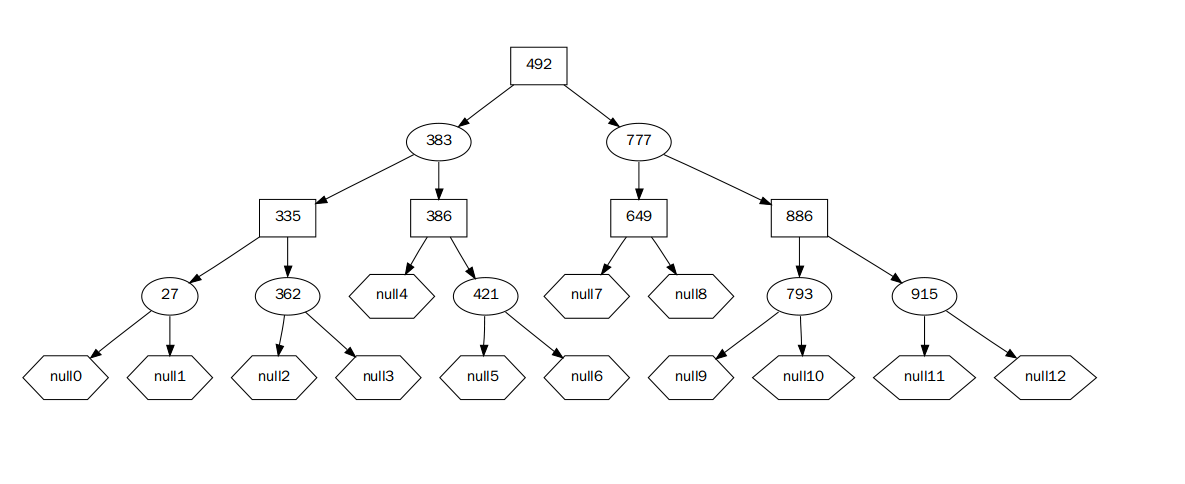
**AG44 Report**



Composition groupe:

Bai Yuhao

Miao YiWen

UTBM Département Informatique

AG44 Semestre A18

**Part I The principle of code**

In our project, we have defined 3 classes: ***Class Vertex, class Edge and class Graph.***

**Class Vertex**

There are 3 attributes in this class, they are: *int id, string nom, int visited.*

***id*** is the identifier of every object of class Vertex. In the class Graph, we put a list of Vertex into a list which is called ListVertex. The id of every Vertex is also the their location in the list.

***nom*** permits us to give a name to every Vertex, it is not necessary. In the class Graph, for each function, we provide 2 ways to use. For example, if the user wants to traverse the Graph, he has to enter the starting Vertex as the parameter of the traverse function, and now, we permit him to enter both the id of Vertex and the nom of Vertex.

***visited*** is a parameter which is used when we traverse the Graph. visited==0 means that the Vertex hasn’t been visited, and visited==1 means that the Vertex has been visited but hasn’t been quit. And if visited==2, that means the vertex has been completely visited (we have visited this Vertex and we have quit this Vertex)

There are 3 constructors in this class, they are: ***Vertex (int id, string nom), Vertex (int id), Vertex (string nom)***

The first one has two parameters, it let us enter the id and nom of the object. And the second one has only one parameter. We only have to enter the id of the object. The function will automatically create the nom of the Vertex according to the ASCII. And the third one can also be used. Each time we use this function in the function main, we will automatically create a id for it.

**Class Edge**

There are 5 attributes in this class, they are:***Vertex src, Vertex dst, int weight, int id, bool visited.***

***src*** and ***dst*** are the source and the destination of the edge and the weight is the weight of this edge.

***id*** is the identifier of every object of class Edge. In the class Graph, we put a list of Edge into a list which is called ListEdge. The id of every Edge is also the their location in the list.

***visited*** is a parameter which is used when we traverse the Graph. visited==0 means that the Edge hasn’t been visited, and visited==1 means that the Edge has been visited.

There are 1 constructors in this class, it is: ***Edge (Vertex src, Vertex dst, int weight);***

It has three parameters, Vertex src, Vertex dst and int weight. And when we add these edges into the graph, they will automatically get a id.

**Class Graph**

There are attributes as follow:

***int nbEdge*** : show the number of Edges in the Graph

***int nbVertex*** : show the number of Vertex in the Vertex

***int nbTopSortVertex*** : this attribute is used when we use the function DFS, to store the order of Vertex visited

***bool directed*** : to show if the graph is directed,0 = undirected, 1 = directed

***bool struct\_type*** : to show the type of the graph,0 = liste, 1 = matrix

***int connectedComponent :*** to store the number of connected components of the graph if we traverse the graph

***Vertex\* listVertex*** : it’s the list of Vertex stored in the graph

***Edge\* listEdge*** : it’s the list of Edge stored in the graph

***Vertex\* TopSortVertex*** : it records the topology order of the graph

***Edge\* sortedEdge*** : it records the edges sorted

***int\*\* matrixAdj*** : it’s the adjacent matrix of the Graph, it records the relations of the Vertex and the weight of edges

***vector< vector<int> > listAdj*** : it’s the adjacent list of the Graph, it records the relations of the Vertex and the weight of edges

There are several functions in the Graph :

***Graph()***;

This is the constructor of the Graph, and the Graph can contain 20 Vertex and 50 Edges by default.

***Graph(int a, int b);***

This is the constructor of the Graph, but with 2 parameters a and b, the first one is the number of the Vertex and the second one is the number of the Edges, so the Graph can contain a Vertex and b Edges by default.

***~Graph();***

This is the destructor of the Graph.

***void clear();***

This function is used to clear the Graph. It will clear all the Vertex and the Edges in the Graph.

***bool isEmpty();***

We use this function to determine if the Graph is empty. If the nbEdge==0 and nbVertex==0, it means that the graph is empty.

***InsertVertex(Vertex \*v);***

We use this function to add Vertex in the list listVertex. Everytime we use this function, the attribute nbVertex = nbVertex+1 automatically.

***InsertEdge(Edge \*e);***

We use this function to add Edge in the list listEdge. Everytime we use this function, the attribute nbEdge = nbEdge+1 automatically.

***DeleteVertex(int id);***

We use this function to delete Vertex in the list listVertex. Everytime we use this function, we delete the Vertex that we want to delete, and we delete all the edges which have the relation with this Vertex and then we use the last Vertex to replace the location of the Vertex deleted ,we modify the matrix and the attribute nbVertex = nbVertex-1 automatically. And in this function, we use the id of the Vertex to find the Vertex that we want to delete.

***DeleteEdge(Edge \*e);***

We use this function to delete Edge in the list listEdge. Everytime we use this function, we delete the Edge that we want to delete. We use the id of the Vertex to find the Vertex that we want to delete. We set the weight of the edge that we delete to 0.

***void file2graph();***

This function permit us to make the graph. We decide what we want to build by writing in the file. directed=0 means undirected graph and directed= 1 maens directed graph. And struct\_type=0 means we show list and struct\_type=1 means we show matrix. In this function, we call void graph\_o\_matrix() or void graph\_n\_matrix() or void graph\_o\_list() or void graph\_n\_list() in different case.

***void graph\_o\_matrix();***

This function is used to make directed matrix.

***void graph\_n\_matrix();***

This function is used to make undirected matrix.

***void graph\_o\_list();***

This function is used to make directed list.

***void graph\_n\_list();***

This function is used to make undirected list.

***int readFile(string filename);***

We use this function to read the file. The first line in the file is the number of the Vertex. The second line in the file is to determine directed(o) or no-directed(n). The third line in the file is to determine the type of the file. M means matrix and l means list. If we use the matrix, the matrix has n rows and n columns. We use a space to separate different element in the matrix. And the number of each element means the weight of the edge. If we use list, we have n rows. And the form is as follow:<a,b> <c,d>. It means that the certain vertex has a edge with vertex a and the weight of the edge is b, and it also has a edge with vertex c, and the weight is d. We also use a space to separate each element.

***void visit\_toWhite(Vertex \*v);***

This function allows to visit Vertex and set Vertex.visited to 0, which means that the Vertex hasn’t been visited.

***void visit\_toGray(Vertex \*v);***

This function allows to visit Vertex and set Vertex.visited to 1, which means that the Vertex has been visited but not quit.

***void visit\_toBlack(Vertex \*v);***

This function allows to visit Vertex and set Vertex.visited to 2, which means that the Vertex has been totally visited.

***void visitEdge\_0(Edge \*e);***

This function allows to visit Edge and set Edge.visited to 0, which means that the Edge hasn’t been visited.

***void visitEdge\_1(Edge \*e);***

This function allows to visit Edge and set Edge.visited to 1, which means that the Edge hasn been visited.

***int findEdge(int i,int j);***

This function allows to find the Edge that we want by entering the id of src and dst.

***Vertex\* BFS(Vertex\* bfs);***

This function returns a pointor (a list) which contain the Vertex that need to be traversed. In this function, firstly, we find the location of the Vertex that we are now in, and then we calculate the number of element in the list bfs. Secondly, we put the adjacent Vertex in the list bfs if this Vertex hasn't benn visited. Finally,we quit this Vertex that means this Vertex has been totally visited and we remove this Vertex from the list bfs.

***void BFSTraverse(int id);***

This function is used to traverse all the Graph by creating a list bfs and using the function. First of all, we create a list bfs to contain the Vertex that we have reached. To initialize, we set all the vertex unvisited. And then we find the location of the start Vertex. We put the start Vertex to the list, we research from the start point until find all the Vertex in the same component, and if the list is not empty, we continue to research, and each time BFS() returns the new list bfs. If the list bfs is empty, we have researched all the Vertex in this component, but maybe we haven't traversed all the points in this Graph. We use a loop make sure that we will traverse all the Vertex in the Graph and we can also calculate the number of connected Components in this graph.

***int DFS(int id,int date);***

This function returns a int called date who record the order of traverse. First of all, we find the location of the Vertex that we are now in and we set its visited to 1 which means that we have entered this Vertex but haven’t quit. And then we find the adjacent Vertex of this Vertex, when we enter a new Vertex, date = date+1. We use recurrence to visit all the Vertex in the same component. When we quit this Vertex, date = date+1.

***void DFSTraverse(int id);***

Like BFSTraverse (int id), first of all, we set all Vertex unvisited to initialize. And then we find the location of the start Vertex. We put the start Vertex to the list, we research from the start point until find all the Vertex in the same component by using recurrence. But when the the function finish, we just find all the Vertex in the same component, but maybe we haven't traverse all the points in this Graph. We use a loop make sure that we will traverse all the Vertex in the Graph and we can also calculate the number of connected Components in this graph.

***void shortestPath\_BF(int id);***

It’s a algorithm created by BelLnam and Ford. First of all, we find the location of the start Vertex, and then we create a matrix called belford to record the shortest distance to other Vertex. The first line of the Matrix is the distance and the second line of the Matrix record the source Vertex of this Vertex. Firstly, we visit the start Vertex and we record the distance of its adjacent Vertex. And secondly we use a loop of n-1 time, and we calculate if the path is shorter each time. After n-1 loop, we can have the final result. In each loop, we copy the matrix belford in the last loop to find the difference. And then we use another loop to calculate the new distance. If the new distance is shorter, we replace the shortest distance. Finally we print our results.

***void shortestPath\_DJ(int id);***

It’s a algorithm created by Dijkstra. First of all, we find the location of the start Vertex, and then we create a matrix called djks to record the shortest distance to other Vertex. The first line is record if the Vertex is in the shortest path,and second line of the Matrix is the distance and the third line of the Matrix record the source Vertex of this Vertex. Firstly, we visit the start Vertex and we record the distance of its adjacent Vertex. And secondly we use a loop of n-1 time, and we calculate if the path is shorter each time. After n-1 loop, we can have the final result. In each loop, we copy the matrix djks in the last loop to find the difference. We find the shortest path in current loop and we record the location and find the next shortest path now but we should renovate all the distance. If the new distance is shorter, we replace the shortest distance. Finally we print our results.

***bool union\_find(Vertex v1,Vertex v2);***

To determine if we can reach v2 as we start from v1 by going through the edges which have been added into the minimum spanning tree. So we decide to use the algorithm of BelLnam-Ford but there is a difference. We can only go through the edges which have been added into the minimum spanning tree! So at the end of this function,if the distance between v1 and v2 is INT\_MAX, it means we can't reach v2 if we start from v1.

***void sortEdge();***

This function allos us to sort the edge accrording to their weight. And we can get a list called sortedEdge with the Edges sorted.

***void MST\_KR\_id();***

It’s a algorithm created by Kruskal to make the minimum spanning tree. First we have to determine if this graph is connected, if not, we can’t k make the minimum spanning tree for it. We use the number of connected component to determine. We can do this only if the number connected component of this graph is 1. If the number connected component of this graph is 0,it means we haven't traversed this graph to calculate it, so we must traverse (BFS,DFS) to calculate connected component before this step. And if the number connected component of this graph is not 0 or 1, we can do this because the graph is not connected.

First of all we set all the vertex and edges unvisited. And then we can add edges in the order of list sortedEdge. If one of the Vertex (src or dst) is not in the MST, we can add this edge to the MST so that we make sure everytime we add at least a new Vertex into the MST. We use the function sortEdge to determine if both src and dst are in the MST. If we add a edge to MST, we set this edge and its src and dst to visited. Finally we will get the sum of weight.

***void MST\_PR\_id();***

It’s a algorithm created by Prim to make the minimum spanning tree. First we have to determine if this graph is connected, if not, we can’t k make the minimum spanning tree for it. We use the number of connected component to determine. We can do this only if the number connected component of this graph is 1. If the number connected component of this graph is 0,it means we haven't traversed this graph to calculate it, so we must traverse (BFS,DFS) to calculate connected component before this step. And if the number connected component of this graph is not 0 or 1, we can do this because the graph is not connected.

First of all we set all the vertex and edges unvisited. we make a loop for n-1 tines and each time we find the shortest edge related to the Vertex in the MST. We find the current shortest edge and at least one of the Vertex(src or dst) of this edge is not visited so that we make sure everytime we add at least a new Vertex into the MST. After that, we put this edge to the MST and we set the dst and src of the edge visited. Finally we will get the sum of weight.

***void TopSort();***

To show the topologic order by using this function.

***void show\_id();***

To show the adjecent matrix or the adjecent list by using this function.

**Part II The difficulties and the solutions**

**BFS**

When we tried to realize algorithm BFS, we found that it’s different from the DFS. In DFS, we can use recurrence to traverse all the Vertex in the same component. But in BFS, we can’t use it. We have to create a container to contain all the Vertex that have been visited but haven’t been quit.

So we decide to create a new pointer who points to a queue called bfs when we start to visited the start Vertex in the function BFSTraverse().

And this pointer is also the parameter of the function BFS(). Every time we execute the function BFS(), we enter the first Vertex in the queue bfs, and we visit this Vertex and we find all its adjacent Vertex which haven’t been visited and we put these Vertex into the end of the queue bfs. And when we find all its adjacent Vertex, we quit this Vertex and we delete it from the queue.

And then the function BFS() also return the pointer. So we can use the sentence bfs = BFS(bfs) to receive and store the new queue and use it for the next step.

We use a while loop to control the number of steps. When the first element bfs[0] is null, which means that the queue is empty and we have traversed all the Vertex, we quit this loop.

**unionFind**

When we make the minimum spanning tree, we add the edges into the tree. But we have to determine if the two Vertex(src and dst) of this edges haven’t been already added to the tree. We can’t add this edge if both Vertex have been added because it will cause a repeat.

We have many ways to determine. We can use the knowledge of equivalence class and we tried to realize it but we find that it’s difficult and we have to define a new class for it, so we have abandoned this method.

We found that we can use the algorithm BelLnam and Ford to realize it. If the distance of the Vertex src and Vertex dst equal to INT\_MAX (infinite) before we add this edge to the tree, we can add this edge to the tree.

So we add a bool attribute called visited into the class Edge. Every time we add an edge into to the tree, we set it attribute visited=True, which means that the edge has been already added into the edge. And if we want to add a new edge, we calculate the distance of its src and dst by going through all the edges which have been added.

**readFile**

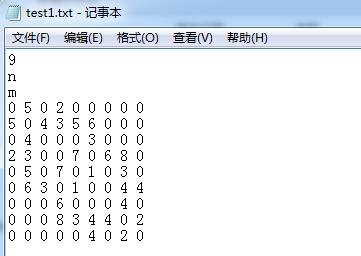
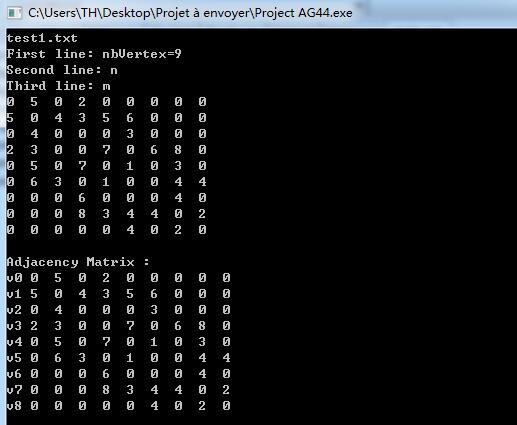
In this function, we can read the properties of the adjacency matrix or the adjacent list in the document. But we have not learned how to open a document for reading before. There are also many differences in reading, which are read by line or by byte. The hardest part is how to determine when to wrap and determine if the number of vertices in a line is eligible.

**Part III Date test and complexity**

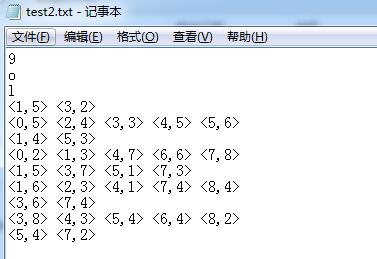
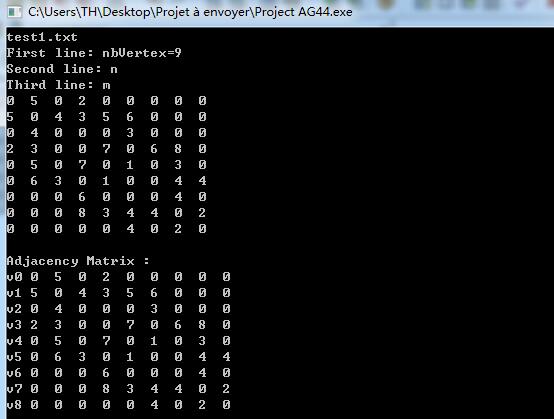
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Adjacency matrix** | | | | | | | | | | | | | |
| Delete node | | DFS | | BFS | | | Prim | | **Kruskal** | | **Dijkstra** | | **Floyd** |
| O(n) | | O(n^2)。 | | O(n^3)。 | | | O(n) | | O(eloge) | | O（n^2） | | O(n^3) |
| **Adjacent list** | | | | | | | | | | | | | |
| Delete node | DFS | | BFS | | Prim | | | **Kruskal** | | **Dijkstra** | | **Floyd** | |
| O(n) | O(n+e) | | O(n+e) | | O(elog2n) | | | O(eloge) | | O（n^2） | | O(n^3) | |
| n: Number of vertrices | | | | | |
| e：Number of edges | | | | | |

For the test, we create two document:

Test1: no directed matrix



Test2: directed list



**Part IV Qualitative balance of work**

Miao Yiwen codes the cardinal structure of class Vertex, class Edge and class Graph and build the relation among them. She writes the function readFile() and the function file2graph() in the class Graph. She is also in charge of the test file.

Bai Yuhao has done the achievement of the algorithm for example BFS traverse, DFS traverse, shortestPath of BelLnam-Ford and Dijkstra, union\_find and the MST of Kruskal and Prim.