CIS7 Unit 11 Rev1 Lab: Graphs in C++ Programming.

In this lab activity, we will use C++ programming to examine the adjacency list and matrices of 2-D graphs as explained in Chapter 16. See Chapter 16 notes for more information on graphs, vertices, and matrix.

In the below graph 1, 5 vertices are displayed: 0, 1, 2, 3, 4



Graph 1: Undirected simple graph

* Refer to the above Graph 1 and complete the following tasks:
* Identify the vertices and the edges. {1,2,3,4,0) and {{0,1},{0,4},{1,0},{4,0},{1,4},{4,1},{1,3},{3,1},{2,1},{1,2},{2,3},{3,2},{3,4},{4,3}}
* Create an adjacency list. (0,1),(0,4),(1,3),(1,2),(1,4),(2,1),(2,3),(4,1),(4,0),(4,3)
* Create an adjacency matrix

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| v | 0 | 1 | 2 | 3 | 4 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 2 | 0 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 0 | 1 |
| 4 | 1 | 1 | 0 | 1 | 0 |

<bits/stdc++.h> in C++

**<bits/stdc++.h>** in C++ is a header file that includes every standard library. Ideally, it is used for finding algorithm or reduce development work.

Disadvantages of bits/stdc++

* **bits/stdc++.h** is ***not a standard header file of GNU C++ library***. So, ***if you try to compile your code with some compiler other than GCC it might fail***.
* Using it would include a lot of unnecessary stuff and increases compilation time.
* This header file is not part of the C++ standard, is therefore ***non-portable***, and should be avoided.
* Moreover, even if there were some catch-all header in the standard, you would want to avoid it in lieu of specific headers, since the compiler has to actually read in and parse every included header (including recursively included headers) every single time that translation unit is compiled.

Advantages of bits/stdc++

* Using this file is a good idea in programming competition, when you want to reduce the time wasted in doing chores; especially when your rank is time sensitive.
* This also reduces all the chores of writing all the necessary header files.
* You don’t have to remember all the STL of GNU C++ for every function you use.

std::vector::push\_back

***Adds a new element*** ***at the end of the vector***, ***after its current last element***. The value content is copied (or moved) to the new element.

This effectively ***increases the container size by one***, which causes an ***automatic reallocation of the allocated storage space if -and only if- the new vector size surpasses the current vector capacity***.

* Input the below C++ program in to IDE, run the program and answer the following questions:



* Verify the output of the program. Does it match with Exercise 1? Explain your answer.

The data seems to create an accurate adjacency list using the inputted valyes

* Review the program code and identify the section of the code that produces the adjacency list for the graph. Explain the functionality of the code in the overall program.

The printGraph function seems to create the adjacency list by iterating through the edges to find the correct values to output.

* Given the graph, can a developer create a C++ program without evaluating the adjacency vertices?

No, the edges are needed to find adjacent vertices in the program, but they can be located initially by inspecting the graph manually.

Data Structure: struct

A **data structure** is ***a group of data elements grouped together under one name***. These data elements, known as **members**, can ***have different types and different lengths***. Data structures can be declared in C++ using the following syntax:

struct type\_name {

member\_type1 member\_name1;

member\_type2 member\_name2;

member\_type3 member\_name3;

.

.

} object\_names;

Where **type\_name** is a ***name for the structure type***, **object\_name** can be ***a set of valid identifiers for objects that have the type of this structure***. Within braces {}, there is a list with the data members, each one is specified with a type and a valid identifier as its name.

Example:



This declares a ***structure type***, called *product*, and defines it having ***two members***: ***weight and price***, each of a different fundamental type. This declaration creates a new type (product), which is then used to declare three objects (variables) of this type: apple, banana, and melon. Note how once product is declared, it is used just like any other type.

Right at the end of the **struct** definition, and before the ***ending semicolon (;)***, the optional field object\_names can be used to directly declare objects of the structure type. For example, the structure objects apple, banana, and melon can be declared at the moment the data structure type is defined.

Class in C++

A **class** is used to ***specify the form of an object and it combines data representation and methods for manipulating that data into one neat package***. The data and functions within a class are called **members of the class**. A **class** ***definition starts with the keyword class followed by the class name***; and the ***class body, enclosed by a pair of curly braces***. A class definition ***must be followed either by a semicolon or a list of declarations***. For example, we defined the Box data type using the keyword class as follows –



The keyword **public** ***determines the access attributes of the members of the class that follows it***. A ***public member can be accessed from outside the class*** ***anywhere within the scope*** ***of the class object***. You can also specify the members of a class as **private or protected**.

* Refer to the below graph, Graph 2, and complete the following tasks:



**Graph 2: Directed simple graph**

* Identify the vertices and edges of Graph 2. {0,1,2,3,4,5}, (2,0),(1,2),(2,1),(2,3),(4,5), (5,4),(0,1)
* Determine the adjacency list and matrix for Graph 2.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| v | 0 | 1 | 2 | 3 | 4 | 5 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 | 0 | 0 |
| 3 | 1 | 1 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 1 |
| 5 | 0 | 0 | 0 | 0 | 1 | 0 |

* Input the below C++ program in IDE, run the program and answer the following questions:

#include <iostream>

using namespace std;

// Data structure to store Adjacency list nodes

struct Node {

int val;

Node\* next;

};

// Data structure to store graph edges

struct Edge {

int src, dest;

};

class Graph

{

// Function to allocate new node of Adjacency List

Node\* getAdjListNode(int dest, Node\* head)

{

Node\* newNode = new Node;

newNode->val = dest;

// point new node to current head

newNode->next = head;

return newNode;

}

int N; // number of nodes in the graph

public:

// An array of pointers to Node to represent

// adjacency list

Node \*\*head;

// Constructor

Graph(Edge edges[], int n, int N)

{

// allocate memory

head = new Node\*[N]();

this->N = N;

// initialize head pointer for all vertices

for (int i = 0; i < N; i++)

head[i] = nullptr;

// add edges to the directed graph

for (unsigned i = 0; i < n; i++)

{

int src = edges[i].src;

int dest = edges[i].dest;

// insert in the beginning

Node\* newNode = getAdjListNode(dest, head[src]);

// point head pointer to new node

head[src] = newNode;

// Uncomment below lines for undirected graph

/\*

newNode = getAdjListNode(src, head[dest]);

// change head pointer to point to the new node

head[dest] = newNode;

\*/

}

}

// Destructor

~Graph() {

for (int i = 0; i < N; i++)

delete[] head[i];

delete[] head;

}

};

// print all neighboring vertices of given vertex

void printList(Node\* ptr)

{

while (ptr != nullptr)

{

cout << " -> " << ptr->val << " ";

ptr = ptr->next;

}

cout << endl;

}

// Graph Implementation in C++ without using STL

int main()

{

// array of graph edges as per above diagram.

Edge edges[] =

{

// pair (x, y) represents edge from x to y

{ 0, 1 }, { 1, 2 }, { 2, 0 }, { 2, 1 },

{ 3, 2 }, { 4, 5 }, { 5, 4 }

};

// Number of vertices in the graph

int N = 6;

// calculate number of edges

int n = sizeof(edges)/sizeof(edges[0]);

// construct graph

Graph graph(edges, n, N);

// print adjacency list representation of graph

for (int i = 0; i < N; i++)

{

// print given vertex

cout << i << " --";

// print all its neighboring vertices

printList(graph.head[i]);

}

return 0;

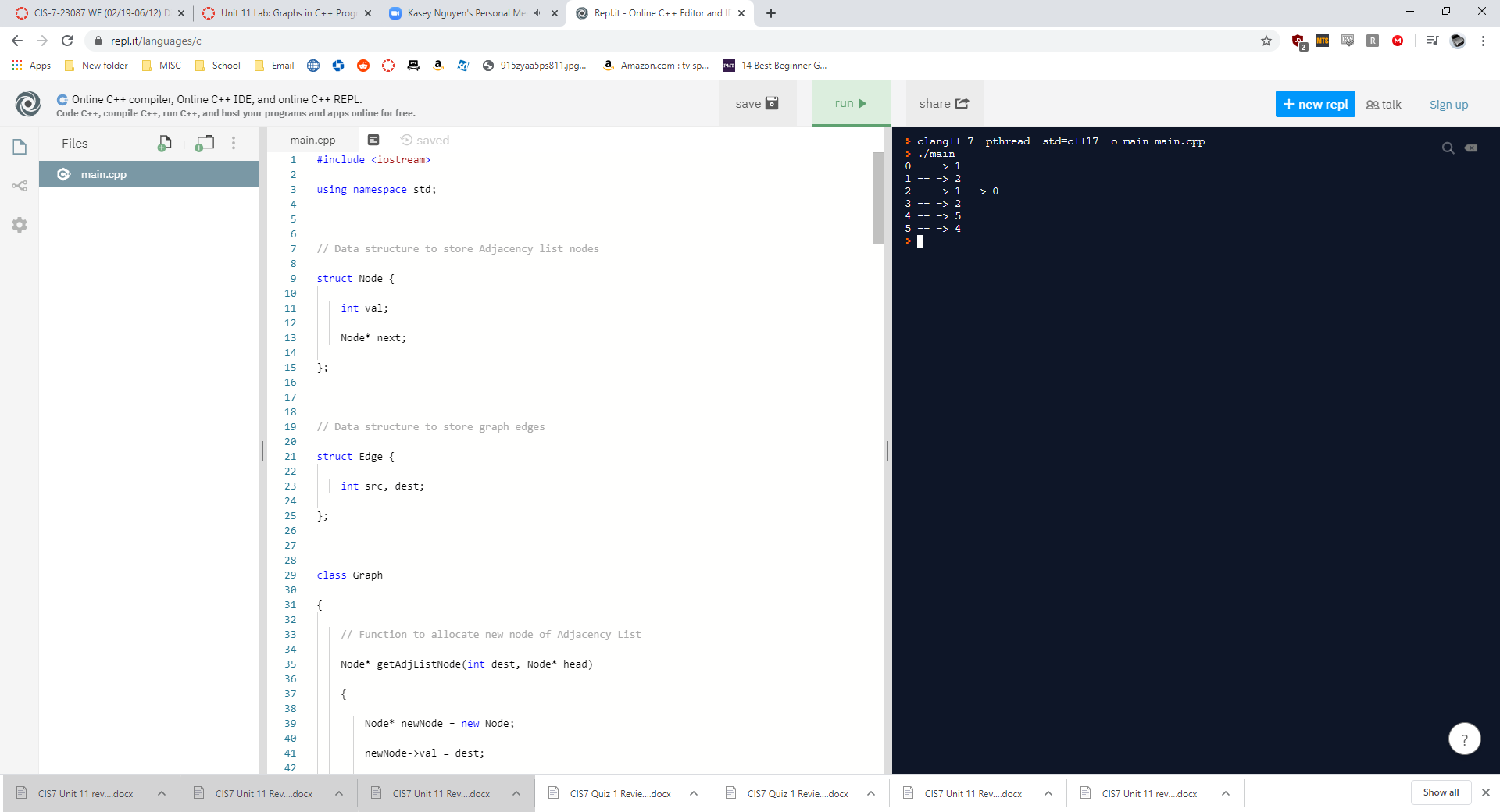
}

* Review above program of Graph 2 and determine the areas of the program that produce the representation of the graph components: vertices, edges and adjacency.

The class graph has a constructor that can add the edges, vertices, and adjacency using iterations using the for loop to find 0-5, with n representing the number of vertices.

* Compare the program adjacency list to your result in Exercise 3 Question B.

The list is formed properly in this program as well, and each vertex has its proper adjacent vertices listed.



* Can a developer create this program without pre-determining the adjacency list?

It appears that this program does not work without predetermining an adjacency list, as the program has no way of knowing what your graph looks like.

* Compare to a simple graph program, such as Exercise 2 program, what areas of Exercise 4 program produce edge directions? Explain the code functionality in the overall program.

There is an edges[i].src and edges[i].dest that iterate through each n (vertex) and establish an edge given a pre-coded “graph”

* Refer to the below graph, Graph 3, and complete the following tasks:



**Graph 3: Directed Weighted Graph**

* Identify the vertices and edges in Graph 3. V={0,1,2,3,4,5) E={{0,1},{1,2},{2,0},{2,1},{3,2},{4,5},{5,4}
* Derive an adjacency list and a matrix for Graph 3.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| v | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | 6 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 7 | 0 | 0 | 0 |
| 2 | 5 | 4 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 10 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 3 |
| 5 | 0 | 0 | 0 | 0 | 1 | 0 |

* Input the following C++ program in IDE, run the program and answer the following questions.

#include <iostream>

using namespace std;

// Data structure to store Adjacency list nodes

struct Node {

int val, cost;

Node\* next;

};

// Data structure to store graph edges

struct Edge {

int src, dest, weight;

};

class Graph

{

// Function to allocate new node of Adjacency List

Node\* getAdjListNode(int value, int weight, Node\* head)

{

Node\* newNode = new Node;

newNode->val = value;

newNode->cost = weight;

// point new node to current head

newNode->next = head;

return newNode;

}

int N; // number of nodes in the graph

public:

// An array of pointers to Node to represent

// adjacency list

Node \*\*head;

// Constructor

Graph(Edge edges[], int n, int N)

{

// allocate memory

head = new Node\*[N]();

this->N = N;

// initialize head pointer for all vertices

for (int i = 0; i < N; ++i)

head[i] = nullptr;

// add edges to the directed graph

for (unsigned i = 0; i < n; i++)

{

int src = edges[i].src;

int dest = edges[i].dest;

int weight = edges[i].weight;

// insert in the beginning

Node\* newNode = getAdjListNode(dest, weight, head[src]);

// point head pointer to new node

head[src] = newNode;

// Uncomment below lines for undirected graph

/\*

newNode = getAdjListNode(src, weight, head[dest]);

// change head pointer to point to the new node

head[dest] = newNode;

\*/

}

}

// Destructor

~Graph() {

for (int i = 0; i < N; i++)

delete[] head[i];

delete[] head;

}

};

// print all neighboring vertices of given vertex

void printList(Node\* ptr, int i)

{

while (ptr != nullptr)

{

cout << "(" << i << ", " << ptr->val

<< ", " << ptr->cost << ") ";

ptr = ptr->next;

}

cout << endl;

}

// Graph Implementation in C++ without using STL

int main()

{

// array of graph edges as per above diagram.

Edge edges[] =

{

// (x, y, w) -> edge from x to y having weight w

{ 0, 1, 6 }, { 1, 2, 7 }, { 2, 0, 5 }, { 2, 1, 4 },

{ 3, 2, 10 }, { 4, 5, 1 }, { 5, 4, 3 }

};

// Number of vertices in the graph

int N = 6;

// calculate number of edges

int n = sizeof(edges)/sizeof(edges[0]);

// construct graph

Graph graph(edges, n, N);

// print adjacency list representation of graph

for (int i = 0; i < N; i++)

{

// print all neighboring vertices of vertex i

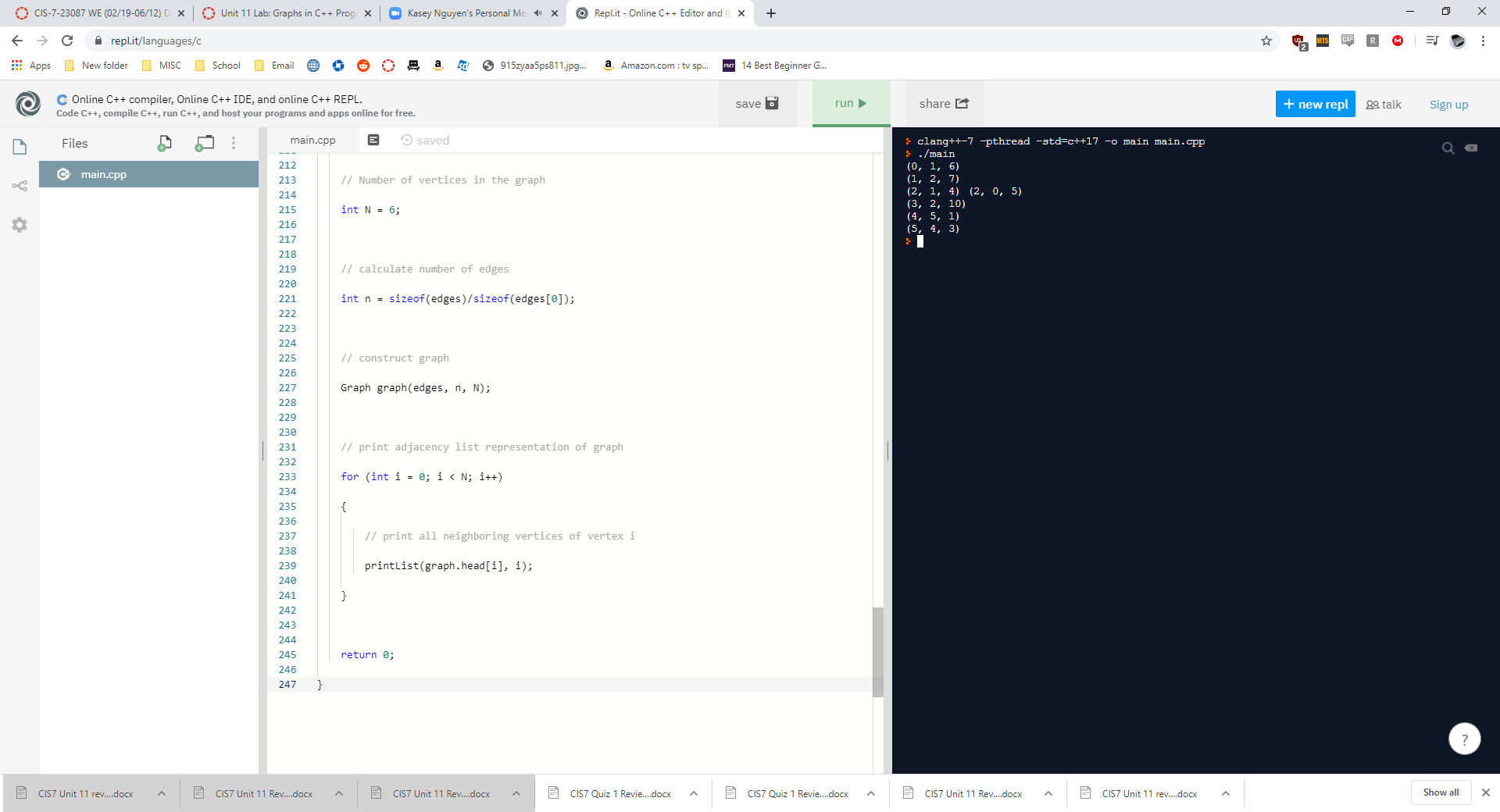
printList(graph.head[i], i);

}

return 0;

}

* Compare the program adjacency list to your result in Exercise 5 Question B.

The adjacency list 

Seems to line up with the expected values.

* Can a developer create this program without pre-determining the adjacency list or a matrix?

No, without a reference point that pre determines the adjacencies or some data that allows the adjacencies to be calculated, the program will have nothing to work with.

* Compare to a unweighted directed graph program, such as Exercise 4 program, what areas of Exercise 6 program produce the weighted edges? Explain the code functionality in the overall program.

The weighted edges are found in the parentheses under the main section, when the Edges are defines in an array. Each weight is assigned as a 3rd variable to the edge, with the first two being the connected vertices. The hinged variable allows the weights to be calculated properly.

Save you document and upload it on Canvas (Unit 10 Lab) for submission.

Shutdown the lab computers.