CHAPTER 6

GRAPH THEORY

HOMEWORK

- Again, all homework is from the Exercises
  - No problems are from the Review Exercises
- Section 6.1, (p. 271), #5-10, 17-18, 22, 27-28, 46-48
- Section 6.2, (p. 281), #20-21, 28-38, 39, 41
- Section 6.3, (p. 296), #1-7
- Section 6.5, (p. 300), #1-3, 7-9, 13-14, 24-25
- Section 6.6, (p. 305), #1-7
- Section 6.7, (p. 311), #6-9, 18-24

A PROGRAMMING PROJECT

- The goal is to create a program that finds the shortest path between two vertices in a graph
- We will use C++
- I will break the program into two parts
- The first part will involve finding a way to store a graph in a program
- The second part will involve writing the code to find the shortest path

PART 1

- Part 1 (Design and Initialization)
- We will store the graph in a struct
- Our first step, then will be to create a struct that represents a graph
- The struct should have two parts
  - One should be the number of vertices in the graph
  - The other should be the adjacency matrix for the graph
- Adjacency matrices
  - This is a 2-dimensional array
  - Create it like this
    - int adj_matrix [20] [20]

MORE ABOUT THE STRUCT

- This program will deal with several graphs, so you can re-use your struct
- Put it right before main() so you can use it everywhere in your program
- Also, the graphs will not have loops or multiple edges
  - This means the adjacency matrix will contain only 0s and 1s
  - They will also be undirected

AN EXAMPLE OF THE STRUCT

- For the code examples that follow, I will assume the struct is

```
struct Graph
{
    int num_verts;
    int  adj_matx [100] [100];
};
```

- and the program creates a graph

```
Graph Fig6_p136;
```

PART 1 PROGRAMMING-THE CREATE-GRAPH FUNCTION

- Create three create_graph functions, one for each graph listed below
  - p. 274, Figure 6.2.1. p. 301, G2 in Fig 6.6.1. p. 282, #39
- You should name the functions create_graph_p274_Fig6_2_1, etc.
- The create_graph functions should look like the code on the next slide

PART 1 PROGRAMMING-THE CREATE-GRAPH FUNCTION

```
create_graph_Fig6_p136(graph gr)
{
    gr.num_verts = 10;
    init_adj_matx (gr);
    add_edge (gr, 0, 2);
    add_edge (gr, 2, 0);
    add_edge (gr, 1, 7);
    add_edge (gr, 7, 1);
}
```

- Change the add_edge() calls to whatever your graph requires

CODE TO INITIALIZE A TWO-DIMENSIONAL ARRAY

- Assume Fig6_p136.num_verts has been set to some value
- The code to initialize every element in the array to 0 is

```
for (int row=0;  row< Fig6_p136.num_verts;  row++)
        for (int col=0;  col< Fig6_p136.num_verts;  col++)
            Fig6_p136.adj_matx [row]  [col] = 0;
```

PART 1 PROGRAMMING-THE ADD_EDGE FUNCTION

- Create a function add_edge (graph, vertex1, vertex2)
- The function should "create" an edge from vertex1 to vertex2
  - Of course, it does this by putting a 1 into the adjacency matrix
- You should call it twice, because the graph is undirected
  - add_edge (Fig6_p136, vertex1, vertex2)
  - add_edge (Fig6_p136, vertex2, vertex1)

CODE TO PRINT NUMBERS IN A TWO-DIMENSIONAL ARRAY

```
for (int row=0;  row< Fig6_p136.num_verts;  row++)
{
      for (int col=0;  col< Fig6_p136.num_verts;  col++)
            cout << Fig6_p136.adj_matx [row]  [col] << “  “;
      cout << endl;
}
```

PART 1A (CALCULATE)

- Calculate and print the incidence matrix for each graph
  - You don't need to store the incidence matrix, though you can if you want to
- This should happen with the function

  calculate_and_print_incidence_matrix (graph)
- This function should do two things
  - First, it should calculate the incidence matrix
  - Second, it should print that incidence matrix

PART 1B (DISPLAY)

- You should "print" each of the graphs
- For a graph, the you should
  - Print the original adjacency matrix
  - Print an edge list
- Use a function for each of these
  - We already have the first function
  - You just have to write the second
- Both functions have one parameter: the graph

PART 1C (VERIFY)

- Double check with the pictures to make sure your code is correct!
- Your edge list should match the edge list found from the picture
- If they match, you have finished Part 1

PART 2-MODIFY THE PROGRAM SO THAT IT CALCULATES THE SHORTEST PATH

- Create a struct that represents a vertex
- The struct (call it Vertex) should contain
  - an int containing the length of the shortest path to the vertex
  - an int containing the number of the previous vertex in the shortest path
  - a boolean to indicate whether or not this vertex was visited
- Now create an array of Vertex structs
  - Dimensioning the array to 20 will be adequate

PART 2-MODIFY THE PROGRAM SO THAT IT CALCULATES THE SHORTEST PATH

- Create an array of unvisited vertices
  - This array will be an int array
  - You can also dimension this to 20
- You will also have to create two indices into this array
  - Name them back and front

PART 2

- To implement the shortest path algorithm
- You should create a function:
  - void find_shortest_path (graph gr, int start_vert, int end_vert)
- An example of the algorithm for the function will be given in class.

- **USING ANY ALGORITHM OTHER THAN THE ONE DESCRIBED IN CLASS WILL RESULT IN A ZERO GRADE FOR THIS ASSIGNMENT**

- What follows is pseudocode
- A lot of it looks like code, but you will have to convert it to actual C++

PART 2-INITIALIZATION

// Initialization

front = 0;

back = 1;

done = false;

Set all vertices as unvisited

Mark start_vertex as visited

Set the predecessor of start_vertex to –1 (a value indicating no predecessor)

Set the length of the path for start_vertex to 0

PART 2-THE ALGORITHM

```
Add start_vertex to unvisited vertex list
while (!done && unvisited vertex list is not empty)
        front_vertex = front vertex from unvisited vertex list
        while (!done && front_vertex has an unvisited neighbor)
                next_neighbor = next neighbor of front_vertex
                if (next_neighbor is not visited)
                        Mark next_neighbor as visited
                        Set the path length of next_neighbor to
                                1 + front_vertex's path length
                        Set the predecessor of next_neighbor to front_vertex
                        Add next_neighbor to back of unvisited vertex list
```

PART 2-CONDITION TO CONTINUE

```
if (next_neighbor = end_vertex)
        done = true;
```

PART 2-GET SHORTEST PATH FROM ARRAY

```
// Traversal is complete; add vertices to shortest path array
shortest_path [0] = end_vertex;
posn = 0;
while (vertex has a predecessor)
        vertex = predecessor of vertex;
        posn++;
        shortest_path [posn] = vertex;
```

PART 2-PRINT PATH

- // Print path
- Print vertices, from shortest_path [posn] down to shortest_path [0]
- Note: The vertices are in reverse order

HOMEWORK

- Again, all homework is from the Exercises
  - No problems are from the Review Exercises
- Section 6.1, (p. 271), #5-10, 17-18, 22, 27-28, 46-48
- Section 6.2, (p. 281), #20-21, 28-38, 39, 41
- Section 6.3, (p. 296), #1-7
- Section 6.5, (p. 300), #1-3, 7-9, 13-14, 24-25
- Section 6.6, (p. 305), #1-7
- Section 6.7, (p. 311), #6-9, 18-24