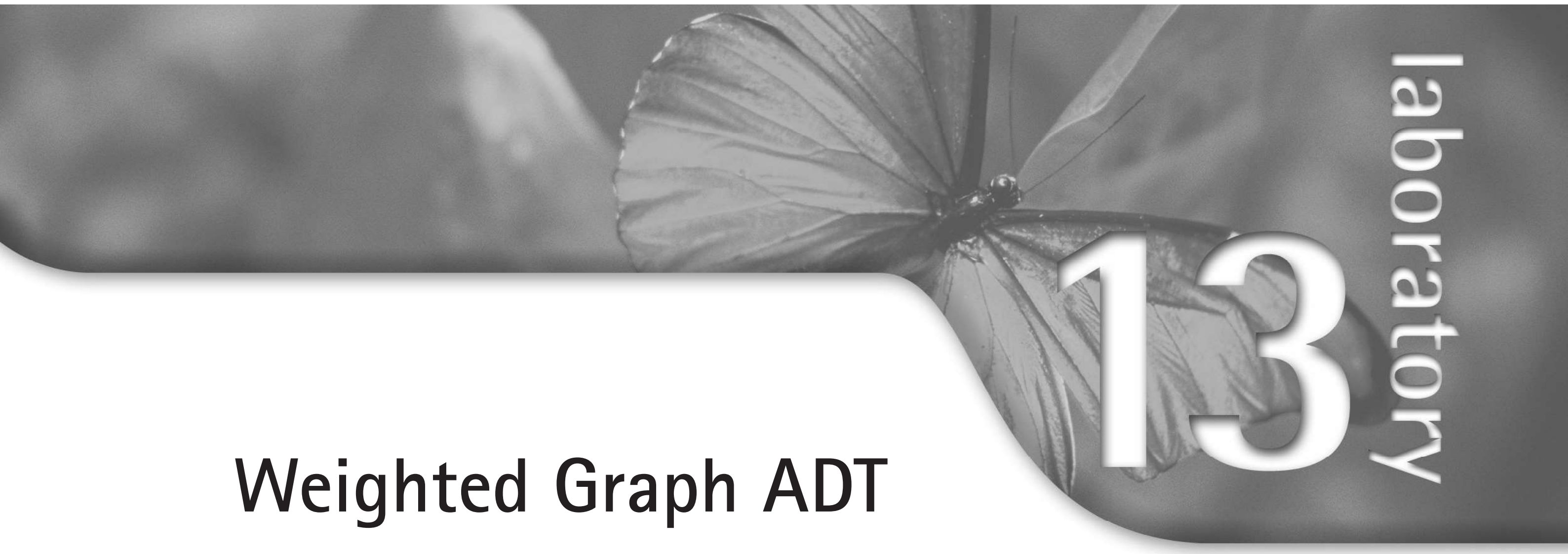


Weighted Graph ADT

objectives

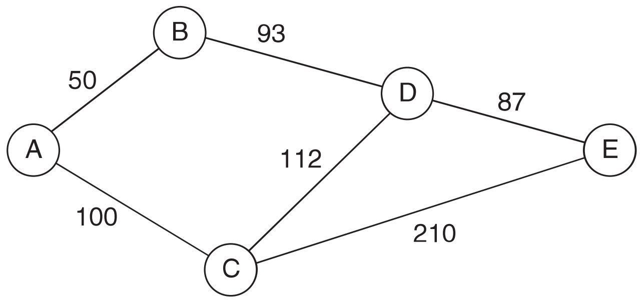
In this laboratory you will:

- Create an implementation of the Weighted Graph ADT using a vertex list and an adjacency matrix
- Develop a routine that finds the least costly (or shortest) path between each pair of vertices in a graph
- Add vertex coloring and implement a function that checks whether a graph has a proper coloring
- Investigate the Four-Color Theorem by generating a graph for which no proper coloring can be created using less than five colors

Overview

Many relationships cannot be expressed easily using either a linear or a hierarchical data structure. The relationship between the cities connected by a highway network is one such relationship. Although it is possible for the roads in the highway network to describe a relationship between cities that is either linear (a one-way street, for example) or hierarchical (an expressway and its off ramps, for instance), we all have driven in circles enough times to know that most highway networks are neither linear nor hierarchical. What we need is a data structure that lets us connect each city to any of the other cities in the network. This type of data structure is referred to as a **graph**.

Like a tree, a graph consists of a set of nodes (called vertices) and a set of edges. Unlike a tree, an edge in a graph can connect any pair of vertices, not simply a parent and its child. The following graph represents a simple highway network.



Each vertex in the graph has a unique **label** that denotes a particular city. Each edge has a **weight** that denotes the cost (measured in terms of distance, time, or money) of traversing the corresponding road. Note that the edges in the graph are **undirected**; that is, if there is an edge connecting a pair of vertices A and B, this edge can be used to move either from A to B or from B to A. The resulting **weighted**, **undirected graph** expresses the cost of traveling between cities using the roads in the highway network. In this laboratory, you focus on the implementation and application of weighted, undirected graphs.

Weighted Graph ADT

Data Items

Each vertex in a graph has a label (of type `char*`) that uniquely identifies it. Vertices may include additional data.

Structure

The relationship between the vertices in a graph is expressed using a set of undirected edges, where each edge connects one pair of vertices. Collectively, these edges define a symmetric relation between the vertices. Each edge in a weighted graph has a weight that denotes the cost of traversing that edge.

Operations

```
WtGraph ( int maxNumber = defMaxGraphSize )
     throw ( bad_alloc )
```

Requirements:
None

Results:
Constructor. Creates an empty graph. Allocates enough memory for a graph containing `maxNumber` vertices.

```
~WtGraph ()
```

Requirements:
None

Results:
Destructor. Deallocates (frees) the memory used to store a graph.

```
void insertVertex ( Vertex newVertex ) throw ( logic_error )
```

Requirements:
Graph is not full.

Results:
Inserts `newVertex` into a graph. If the vertex already exists in the graph, then updates it.

```
void insertEdge ( char *v1, char *v2, int wt )
     throw ( logic_error )
```

Requirements:
Graph includes vertices `v1` and `v2`.

Results:
Inserts an undirected edge connecting vertices `v1` and `v2` into a graph. The weight of the edge is `wt`. If there is already an edge connecting these vertices, then updates the weight of the edge.

```
bool retrieveVertex ( char *v, Vertex &vData ) const
```

Requirements:
None

Results:
Searches a graph for vertex `v`. If this vertex is found, then copies the vertex's data to `vData` and returns `true`. Otherwise, returns `false` with `vData` undefined.

```
bool getEdgeWeight ( char *v1, char *v2, int &wt ) const
     throw ( logic_error )
```

Requirements:
Graph includes vertices `v1` and `v2`.

Results:
Searches a graph for the edge connecting vertices `v1` and `v2`. If this edge exists, then returns `true` with `wt` returning the weight of the edge. Otherwise, returns `false` with `wt` undefined.

```
void removeVertex ( char *v ) throw ( logic_error )
```

Requirements:
Graph includes vertex `v`.

Results:
Removes vertex `v` from a graph.

```
void removeEdge ( char *v1, char *v2 ) throw ( logic_error )
```

Requirements:
Graph includes vertices `v1` and `v2`.

Results:
Removes the edge connecting vertices `v1` and `v2` from a graph.

```
void clear ()
```

Requirements:
None

Results:
Removes all the vertices and edges in a graph.

```
bool isEmpty () const
```

Requirements:
None

Results:
Returns `true` if a graph is empty (no vertices). Otherwise, returns `false`.

```
bool isFull () const
```

Requirements:
None

Results:
Returns `true` if a graph is full. Otherwise, returns `false`.

```
void showStructure () const
```

Requirements:
None

Results:
Outputs a graph with the vertices in array form and the edges in adjacency matrix form (with their weights). If the graph is empty, outputs "Empty graph". Note that this operation is intended for testing/debugging purposes only.

Laboratory 13: Cover Sheet

Name ______________________________ Date ____________________

Section ______________________________

Place a check mark in the *Assigned* column next to the exercises your instructor has assigned to you. Attach this cover sheet to the front of the packet of materials you submit following the laboratory.

Activities	**Assigned:** Check or list exercise numbers	**Completed**
Prelab Exercise		
Bridge Exercise		
In-lab Exercise 1		
In-lab Exercise 2		
In-lab Exercise 3		
Postlab Exercise 1		
Postlab Exercise 2		
Total		

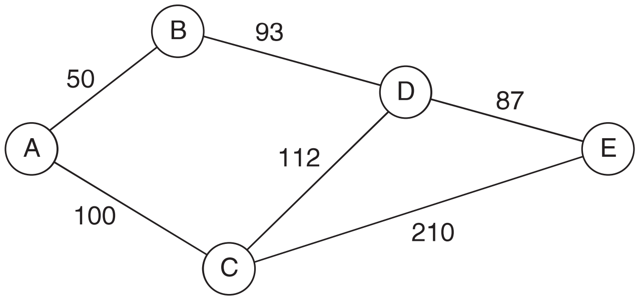
Laboratory 13: Prelab Exercise

Name ______________________________ Date ____________________

Section ______________________________

You can represent a graph in many ways. In this laboratory you will use an array to store the set of vertices and an **adjacency matrix** to store the set of edges. An entry (j,k) in an adjacency matrix contains information on the edge that goes from the vertex with index j to the vertex with index k. For a weighted graph, each matrix entry contains the weight of the corresponding edge. A specially chosen weight value is used to indicate edges that are missing from the graph.

The following graph yields the vertex list and adjacency matrix shown below. A '–' is used to denote an edge that is missing from the graph.



Vertex list		*Adjacency matrix*					
Index	*Label*	*From\To*	*0*	*1*	*2*	*3*	*4*
0	A	0	–	50	100	–	–
1	B	1	50	–	–	93	–
2	C	2	100	–	–	112	210
3	D	3	–	93	112	–	87
4	E	4	–	–	210	87	–

Vertex A has an array index of 0 and vertex C has an array index of 2. The weight of the edge from vertex A to vertex C is therefore stored in entry (0,2) in the adjacency matrix.

Step 1: Implement the operations in the Weighted Graph ADT using an array to store the vertices (`vertexList`) and an adjacency matrix to store the edges (`adjMatrix`). The number of vertices in a graph is not fixed; therefore, you need to store the maximum number of vertices the graph can hold (`maxSize`) as well as the actual number of vertices in the graph (`size`). Base your implementation on the following declarations from the file *wtgraph.h*. An implementation of the `showStructure` operation is given in the file *show13.cpp*.

```
const int defMaxGraphSize = 10,      // Default number of vertices
          vertexLabelLength = 11,    // Length of a vertex label
          infiniteEdgeWt = INT_MAX;  // "Weight" of a missing edge

class Vertex
{
  public:

    char label [vertexLabelLength];   // Vertex label
};

class WtGraph
{
  public:

    // Constructor
    WtGraph ( int maxNumber = defMaxGraphSize )
        throw ( bad_alloc );

    // Destructor
    ~WtGraph ();

    // Graph manipulation operations
    void insertVertex ( Vertex newVertex )            // Insert vertex
        throw ( logic_error );
    void insertEdge ( char *v1, char *v2, int wt )    // Insert edge
        throw ( logic_error );
    bool retrieveVertex ( char *v, Vertex &vData );
                                                      // Get vertex
    bool getEdgeWeight ( char *v1, char *v2, int &wt )
        throw ( logic_error );                        // Get edge wt.
    void removeVertex ( char *v )                     // Remove vertex
        throw ( logic_error );
    void removeEdge ( char *v1, char *v2 )            // Remove edge
        throw ( logic_error );
    void clear ();                                    // Clear graph

    // Graph status operations
    bool isEmpty () const;                            // Graph is empty
    bool isFull () const;                             // Graph is full

    // Output the graph structure — used in testing/debugging
    void showStructure ();

  private:

    // Facilitator functions
    int getIndex ( char *v );                  // Converts vertex label to
                                               // an adjacency matrix
                                               // index

    int getEdge ( int row, int col );          // Get edge weight using
    void setEdge ( int row, int col, int wt);  // Set edge weight using
                                               // adjacency matrix
                                               // indices
```

```
    // Data members
    int maxSize,            // Maximum number of vertices in the graph
        size;               // Actual number of vertices in the graph
    Vertex *vertexList;     // Vertex list
    int *adjMatrix;         // Adjacency matrix
};
```

Your implementations of the public member functions should use your `getEdge()` and `setEdge()` facilitator functions to access entries in the adjacency matrix. For example, the assignment statement

```
setEdge(2,3, 100);
```

uses the `setEdge()` function to assign a weight of 100 to the entry in the second row, third column of the adjacency matrix. The `if` statement

```
if ( getEdge(j,k) == infiniteEdgeWt )
   cout << "Edge is missing from graph" << endl;
```

uses this function to test whether there is an edge connecting the vertex with index `j` and the vertex with index `k`.

Step 2: Save your implementation of the Weighted Graph ADT in the file *wtgraph.cpp*. Be sure to document your code.

Laboratory 13: Bridge Exercise

Name ____ Date ________________________

Section ____

Check with your instructor whether you are to complete this exercise prior to your lab period or during lab.

The test program in the file *test13.cpp* allows you to interactively test your implementation of the Weighted Graph ADT using the following commands.

Command	Action
`+v`	Insert vertex `v`.
`=v w wt`	Insert an edge connecting vertices `v` and `w`. The weight of this edge is `wt`.
`?v`	Retrieve vertex `v`.
`#v w`	Retrieve the edge connecting vertices `v` and `w` and output its weight.
`-v`	Remove vertex `v`.
`!v w`	Remove the edge connecting vertices `v` and `w`.
`E`	Report whether the graph is empty.
`F`	Report whether the graph is full.
`C`	Clear the graph.
`Q`	Quit the test program.

Note that `v` and `w` denote vertex labels (type `char*`), not individual characters (type `char`). As a result, you must be careful to enter these commands using the exact format shown above—including spaces.

Step 1: Prepare a test plan for your implementation of the Weighted Graph ADT. Your test plan should cover graphs in which the vertices are connected in a variety of ways. Be sure to include test cases that attempt to retrieve edges that do not exist or that connect nonexistent vertices. A test plan form follows.

Step 2: Execute your test plan. If you discover mistakes in your implementation, correct them and execute your test plan again.

Test Plan for the Operations in the Weighted Graph ADT

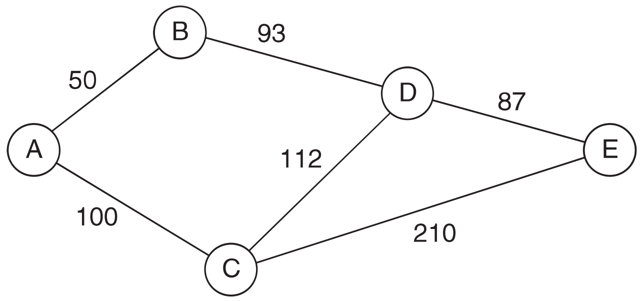
Test Case	*Commands*	*Expected Result*	*Checked*

Laboratory 13: In-lab Exercise 1

Name ______________________________ Date ____________________

Section ______________________________

In many applications of weighted graphs, you need to determine not only whether there is an edge connecting a pair of vertices, but whether there is a path connecting the vertices. By extending the concept of an adjacency matrix, you can produce a **path matrix** in which an entry (j,k) contains the cost of the least costly (or **shortest**) path from the vertex with index j to the vertex with index k. The following graph yields the path matrix shown below.



Vertex list			*Path matrix*				
Index	*Label*	*From/To:*	*0*	*1*	*2*	*3*	*4*
0	A	0	0	50	100	143	230
1	B	1	50	0	150	93	180
2	C	2	100	150	0	112	199
3	D	3	143	93	112	0	87
4	E	4	230	180	199	87	0

This graph includes a number of paths from vertex A to vertex E. The cost of the least costly path connecting these vertices is stored in entry (0,4) in the path matrix, where 0 is the index of vertex A and 4 is the index of vertex E. The corresponding path is ABDE.

In creating this path matrix, we have assumed that a path with cost 0 exists from a vertex to itself (entries of the form (j, j)). This assumption is based on the view that traveling from a vertex to itself is a nonevent and thus costs nothing. Depending on how you intend to apply the information in a graph, you may want to use an alternative assumption.

Given the adjacency matrix for a graph, we begin construction of the path matrix by noting that all edges are paths. These one-edge-long paths are combined to form two-edge-long paths by applying the following reasoning.

If there exists a path from a vertex j to a vertex m and
there exists a path from a vertex m to a vertex k,
then there exists a path from vertex j to vertex k.

We can apply this same reasoning to these newly generated paths to form paths consisting of more and more edges. The key to this process is to enumerate and combine paths in a manner that is both complete and efficient. One approach to this task is described in the following algorithm, known as Warshall's algorithm. Note that variables `j`, `k`, and `m` refer to vertex indices, *not* vertex labels.

```
Initialize the path matrix so that it is the same as the edge
matrix (all edges are paths). In addition, create a path with
cost 0 from each vertex back to itself.

for ( m = 0 ; m < size ; m++ )
  for ( j = 0 ; j < size ; j++ )
      for ( k = 0 ; k < size ; k++ )
          If there exists a path from vertex j to vertex m and
             there exists a path from vertex m to vertex k,
          then add a path from vertex j to vertex k to the path matrix.
```

This algorithm establishes the existence of paths between vertices but not their costs. Fortunately, by extending the reasoning used above, we can easily determine the costs of the least costly paths between vertices.

```
If there exists a path from a vertex j to a vertex m and
   there exists a path from a vertex m to a vertex k and
   the cost of going from j to m to k is less than entry (j,k) in
     the path matrix,
then replace entry (j,k) with the sum of entries (j,m) and (m,k).
```

Incorporating this reasoning into the previous algorithm yields the following algorithm, known as Floyd's algorithm.

```
Initialize the path matrix so that it is the same as the edge
matrix (all edges are paths). In addition, create a path with
cost 0 from each vertex back to itself.

for ( m = 0 ; m < size ; m++ )
  for ( j = 0 ; j < size ; j++ )
      for ( k = 0 ; k < size ; k++ )
          If there exists a path from vertex j to vertex m and
             there exists a path from vertex m to vertex k and
             the sum of entries (j,m) and (m,k) is less than entry
                 (j,k) in the path matrix,
      then replace entry (j,k) with the sum of entries (j,m)
               and (m,k).
```

The following Weighted Graph ADT operation computes a graph's path matrix.

```
void computePaths ()
```

Requirements:
None

Results:
Computes a graph's path matrix.

Step 1: Add the data member

```
int *pathMatrix;   // Path matrix
```

and the function prototype

```
void computePaths ();   // Computes path matrix
```

to the WtGraph class declaration in the file *wtgraph.h*.

Step 2: Implement the `computePaths` operation described above and add it to the file *wtgraph.cpp*.

Step 3: Replace the `showStructure()` function in the file *wtgraph.cpp* with a `showStructure()` function that outputs a graph's path matrix in addition to its vertex list and adjacency matrix. An implementation of this function is given in the file *show14.cpp*.

Step 4: Activate the "`PM`" (path matrix) test in the test program *test13.cpp* by removing the comment delimiter (and the characters "`PM`") from the lines that begin with "`//PM`".

Step 5: Prepare a test plan for the `computePaths` operation that includes graphs in which the vertices are connected in a variety of ways with a variety of weights. Be sure to include test cases in which an edge between a pair of vertices has a higher cost than a multiedge path between these same vertices. The edge CE and the path CDE in the graph shown earlier have this property. A test plan form follows.

Step 6: Execute your test plan. If you discover mistakes in your implementation of the `computePaths` operation, correct them and execute your test plan again.

Test Plan for the `computePaths` Operation

Test Case	*Commands*	*Expected Result*	*Checked*