Terence Henriod

Generated by Doxygen 1.7.6.1

Fri Nov 22 2013 14:08:22

Contents

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 WeightedGraph::Vertex Class Reference

`#include <WeightedGraph.h>`

Public Member Functions

- void setColor (char newColor)
- void setLabel (const string &newLabel)
- char getColor () const
- string getLabel () const

Private Attributes

- string label
- char color

3.1.1 Detailed Description

A fundamental elemental element of a graph. Contains data pertaining to a label and color of a vertex contained in a graph.

label The label of the vertex. Used as the key of the vertex. color The color the vertex currently has.

3.1.2 Member Function Documentation

3.1.2.1 char **WeightedGraph::Vertex::getColor (** **)** const `[inline]`

3.1.2.2 string **WeightedGraph::Vertex::getLabel (** **)** const `[inline]`

3.1.2.3 void **WeightedGraph::Vertex::setColor** (char *newColor*) `[inline]`

3.1.2.4 void **WeightedGraph::Vertex::setLabel** (const string & *newLabel*) `[inline]`

3.1.3 Member Data Documentation

3.1.3.1 char **WeightedGraph::Vertex::color** `[private]`

3.1.3.2 string **WeightedGraph::Vertex::label** `[private]`

The documentation for this class was generated from the following file:

- /nfs/home/thenriod/Desktop/PA11/Doxy11/WeightedGraph.h

3.2 WeighedGraph Class Reference

`#include <WeightedGraph.h>`

3.2.1 Detailed Description

An implementation of a weighted graph that utilizes an adjacency matrix and a path matrix. Supports graph coloring, although it does not support any kind of graph coloring algorithms other than checking for a valid color. Contains an inner vertex class.

maxSize The current maximum size capacity of the graph (in terms of number of vertices). size The current size (number of vertices) of the graph. vertexList An array containing the vertices currently contained in the graph. adjacencyMatrix A 1-D array to be mapped as a 2-D array. In this implementation, the 1-D array is blocked out in terms of the maximum row size, and only the required portions of each block are used. The helper functions are designed to accomodate for this non- contiguous array use. Each valid entry pertains to an edge weight between two vertices, based on how the particular index was mapped to. pathMatrix This array is mapped in a similar manner to the adjacency matrix, except it contains shortest path (least cost) data.

The documentation for this class was generated from the following file:

- /nfs/home/thenriod/Desktop/PA11/Doxy11/WeightedGraph.h

3.3 WeightedGraph Class Reference

`#include <WeightedGraph.h>`

Classes

- class Vertex

Public Member Functions

- WeightedGraph (int maxNumber=MAX_GRAPH_SIZE)
- WeightedGraph (const WeightedGraph &other)
- WeightedGraph & operator= (const WeightedGraph &other)
- ∼WeightedGraph ()
- void computePaths ()
- void insertVertex (const Vertex &newVertex) throw (logic_error)
- void insertEdge (const string &v1, const string &v2, const int wt) throw (logic_-error)
- bool retrieveVertex (const string &v, Vertex &vData) const
- bool getEdgeWeight (const string &v1, const string &v2, int &wt) const throw (logic_error)
- void removeVertex (const string &v) throw (logic_error)
- void removeEdge (const string &v1, const string &v2) throw (logic_error)
- void clear ()
- bool areAllEven () const
- bool hasProperColoring () const
- bool isEmpty () const
- bool isFull () const
- void showShortestPaths ()
- void showStructure () const

Static Public Attributes

- static const int MAX_GRAPH_SIZE = 10
- static const int INFINITE_EDGE_WT = INT_MAX
- static const int NOT_FOUND = -1
- static const int LOOP_COST = 0
- static const char DEFAULT_COLOR = 'R'

Private Member Functions

- void setEdge (const int row, const int col, const int wt)
- void setPath (const int row, const int col, const int cost)
- int getEdge (const int row, const int col) const
- int getIndex (const string &v) const
- int getPath (const int row, const int col) const

Private Attributes

- int maxSize
- int size
- Vertex ∗ vertexList
- int ∗ adjacencyMatrix
- int ∗ pathMatrix

3.3.1 Constructor & Destructor Documentation

3.3.1.1 WeightedGraph::WeightedGraph (int *maxNumber* = MAX_GRAPH_SIZE)

WeightedGraph

The default constructor for the WeightedGraph ADT. Initializes an empty graph of the given parameterized size.

Parameters

maxNumber	The number of vertices the WeightedGraph can contain when full. By default the value MAX_GRAPH_SIZE (defined in WeightedGraph.h)is given.

Precondition

1. There is available memory to instantiate a WeightedGraph object.
2. The WeightedGraph is given a valid identifier.

Postcondition

1. An empty WeightedGraph object will be instantiated.

3.3.1.2 WeightedGraph::WeightedGraph (const WeightedGraph & *other*)

WeightedGraph

The default constructor for the WeightedGraph ADT. Initializes an empty graph of the given parameterized size.

Parameters

maxNumber	The number of vertices the WeightedGraph can contain when full. By default the value MAX_GRAPH_SIZE (defined in WeightedGraph.h)is given.

Precondition

1. There is available memory to instantiate a WeightedGraph object.
2. The WeightedGraph is given a valid identifier.
3. Weightedgraph other is a valid instantiation of a WeightedGraph object.

Postcondition

1. A WeightedGraph object equivalent to the given parameter WeightedGraph other will be instantiated.

3.3.1.3 WeightedGraph::∼WeightedGraph ( )

WeightedGraph

The destructor for the WeightedGraph ADT. Ensures all dynamic memory is returned.

Precondition

1. ∗this was a properly instantiated WeightedGraph object.

Postcondition

1. All dynamic memory will be returned.
2. ∗this will be destructed.

3.3.2 Member Function Documentation

3.3.2.1 bool WeightedGraph::areAllEven ( ) const

areAllEven

Determines if each vertex in the graph has an even degree, and then returns true if each vertex does have an even degree and false otherwise.

Returns

hasEulerianChain The truth value of whether or not the graph has all even degree vertices (also indicates if there is a closed chain that uses all paths once). true if all vertices do have even degree, false otherwise.

Precondition

1. ∗this is a valid instantiation of a WeightedGraph.

Postcondition

1. The WeightedGraph will remain unchanged.
2. The truth of all verticex having even degree is returned.

1. Every vertex is iteratively checked.
2. The number of edges each vertex has is checked.
3. If any vertex is found to have an odd degree the check is halted.
4. Return true if all vertices have even degree, return false otherwise. TODO refine this algorithm if possible.

3.3.2.2 void WeightedGraph::clear ( )

clear

Empties the WeightedGraph of data.

Precondition

1. A valid WeightedGraph object has been instantiated.

Postcondition

1. The WeightedGraph will be emptied.

1. The size of the data set is reduced to zero.

3.3.2.3 void WeightedGraph::computePaths ( )

computePaths

Computes the shortest paths between the vertices of a graph and stores them in the graph's path matrix.

Precondition

1. ∗this is a valid instantiation of a WeightedGraph
2. The adjacency matrix is in a valid state.

Postcondition

1. The shortest (lowest cost) paths between all vertices will be computed.
2. The path costs will be stored in the path matrix.
3. No record of what the shortest paths actually are will be kept, only the resulting cost.

Floyd's Algorithm is used as follows:

1. Every vertex is iteratively considered as a "mutual" neighbor.
2. Every vertex is iteratively considered for every given "mutual" vertex as a "start" vertex.
3. Every other vertex is then iteratively considered for given "mutual" and "start" vertices, as an "end" vertex.
4. A check is made for the existence of paths between the start vertex and the mutual vertex and between the mutual vertex and the end vertex.
5. If such paths exist, then the cost of traversing the two paths described in the previous step (the "indirect" path) is compared with the cost of the path directly from the start vertex to the end vertex.
6. If the indirect path is found to be cheaper than the direct one, then the direct one is replaced with the indirect one on the path matrix.

3.3.2.4 int **WeightedGraph::getEdge** (const int *row,* const int *col*) const `[private]`

getEdge

Returns the edge weight at the given position at the specified location in the adjacency matrix.

Parameters

row	The row index to be searched in the path matrix.
col	The column index to be searched in the path matrix.

Returns

weight

Precondition

1. ∗this is a valid instantiation of a WeightedGraph.
2. row and col are valid positions in the adjacency matrix.

Postcondition

1. The WeightedGraph and all its members will remain unchanged.
2. The index of the desired entry in the 1-D representation of the adjacency matrix is returned.

1. The indices that correspond to a 2-D matrix can be converted using the formula: (row-index ∗ number-of-columns) + column-index.

3.3.2.5 bool **WeightedGraph::getEdgeWeight** (const string & *v1*, const string & *v2*, int & *wt*) const throw (logic_error)

getEdgeWeight

Finds the edge wieght corresponding to the edge between two vertices of the given labels. The weight is then passes back by reference. true is retruned to indicate that the edge (or at least the two vertices that define it) exist in the graph. Otherwise, false is returned.

Parameters

v1	A label for the first vertex used to define an edge.
v2	A label for the second vertex used to define an edge.
wt	The parameter used to pass back a weight of a sought edge.

Returns

result The truth value of whether an edge could be found in a graph, true if the edge does exist in the graph, false otherwise. Also indicates if the reference parameter int wt contains valid or undeterminate data.

Precondition

1. ∗this is a valid instantiation of a WeightedGraph.
2. The sought vertices v1 and v2 (and therefore the edge between them) should exist in the graph.

Postcondition

1. If the edge exists in the graph, then its weight is copied into the int wt reference parameter to be passed back and true is returned.
2. Otherwise, wt will be in an undetermined state and false is returned.

1. The vertex list is searched for verticex with lables matching the two given ones.
2. If found, their indices are used to locate the edge wait in the adjacency matrix.

3. The edge weight is copied to the reference parameter wt.
4. If either of the vertices is not found, then an exception is thrown to indicate that an invalid edge was sought.

Exceptions

logic_error	Used to indicate that at least one of the given vertex labels does not exist in the graph, and therefore the edge does not exist.

3.3.2.6 int WeightedGraph::getIndex (const string & *v*) const [private]

getIndex

Retrieves the index of a vertex in the vertex list for use as a coordinate in an adjacency matrix.

Parameters

v	A string label used a a key for finding a desired vertex.

Returns

result The integer index of the sought vertex in the vertex list.

Precondition

1. ∗this is a valid instantiation of a WeightedGraph.

Postcondition

1. The WeightedGraph will remain unchanged.
2. The index of the sought vertex in the vertex list is returned.
3. If the vertex is not found, NOT_FOUND (defined in WeightedGraph.h) is returned to signal that a vertex with a label matching v was not found.

1. A linear search is conducted to find a vertex in the vertex list with a label matching the given parameter string v.

3.3.2.7 int WeightedGraph::getPath (const int *row,* const int *col*) const [private]

getPath

Returns the path cost at the given position at the specified location in the path matrix.

Parameters

row	The row index to be searched in the path matrix.
col	The column index to be searched in the path matrix.

Returns

pathCost

Precondition

1. ∗this is a valid instantiation of a WeightedGraph.
2. row and col are valid positions in the adjacency matrix.

Postcondition

1. The WeightedGraph and all its members will remain unchanged.
2. The index of the desired entry in the 1-D representation of the adjacency matrix is returned.

1. The indices that correspond to a 2-D matrix can be converted using the formula: (row-index ∗ number-of-columns) + column-index.

3.3.2.8 bool WeightedGraph::hasProperColoring ( ) const

hasProperColoring

Returns the truth value of whether or not the graph has a valid vertex coloring. true is returned if no vertex is adjacent to another vertex of same color, returns false otherwise.

Returns

hasValidColoring The truth value of whether or not the graph has a valid vertex coloring.

Precondition

1. ∗this is a valid instantiation of a WeightedGraph.
2. The vertices are all colored with valid colors.

Postcondition

1. The WeightedGraph will remain unchanged.
2. If the graph has a valid vertex coloring, true is returned. Otherwise, false is returned.

1. Every vertex is checked to see if it has the same color as any neighbors.
2. If a pair of neighbors are found to have the same color, the check is halted.
3. If no pair of neighbors are found to have the same color, then true is returned.
4. If any pair of neighbors is found to have the same color, false is returned. TODO refine this algorithm if possible.

3.3.2.9 void **WeightedGraph::insertEdge** (const string & *v1*, const string & *v2*, const int *wt*) throw (logic_error)

insertEdge

Inserts an undirected edge of the given weight between the given vertices.

Parameters

v1	The label of the first vertex in the pair defining the edge.
v2	The label of the second vertex in the pair defining the edge.
wt	The weight of the edge to be updated or added.

Precondition

1. $*$this is a properly instantiated WeightedGraph.
2. The graph contains at least two vertices.
3. The vertices that define the edge should be present in the graph.

Postcondition

1. The weighted edge will be added to the graph. That is, the adjacency matrix will contain the updated/new edge weight in the two matrix locations pertaining to the given vertices (because this is not a digraph).
2. If either of the given vertex parameters are not found, then an exception of type logic_error is thrown to indicate such.

1. The vertex list is searched for both vertices and their index is recorded.
2. The found indices are used to locate the appropriate locations in the adjacency matrix.

3. The adjacency matrix is then updated with the appropriate edge weight.
4. If either of the vertices is not found in the graph, then an exception of type logic_error is thrown to indicate that the edge cannot be added between vertices that do not exist in the graph.

Exceptions

logic_error	This exception is used to report that the edge cannot be added between vertices that do not exist in the graph.

3.3.2.10 void **WeightedGraph::insertVertex (** const Vertex & *newVertex* **)** throw (logic_error)

insertVertex

Adds a vertex to the graph if there is room in the list. An exception is thrown if parameter Vertex newVertex cannot be added. No edges are added, just a vertex.

Parameters

newVertex	A new vertex to be added to the graph.

Precondition

1. A valid WeightedGraph object has been instantiated.
2. The WeightedGraph has not been filled to maximum capacity.

Postcondition

1. If possible, Vertex newVertex is added to the end of the vertexList array member.
2. If the graph is full, an exception of type logic_error is thrown to indicate that the graph is full.

1. If the vertexList array member is not full, then the size of the list is increased by one and the vertex is added in the last spot in the list.
2. A row of "non-edges" is then added to the adjacencyMatrix array member.
3. An edge with the weight of a loop cost is added where the new vertex intersects with itself in the matrix.
4. A column of "non_edges" is also appended to the matrix.
5. If the vertexList was full, then an exception of type logic_error is thrown to indicate that the WeightedGraph is full.

Exceptions

logic_error	Indicates that an attempt to add a Vertex to a full WeightedGraph was made

3.3.2.11 bool WeightedGraph::isEmpty ( ) const

isEmtpy

Returns the truth of the emptiness of the graph, true if empty, false otherwise.

Returns

empty The truth of the graph being empty.

Precondition

1. ∗this is a valid instantiation of a WeightedGraph.
2. The size member of ∗this is in a valid state.

Postcondition

1. The truth of the emptiness of the graph is returned.

3.3.2.12 bool WeightedGraph::isFull ( ) const

isFull

Returns the truth of the fullness of the graph, true if full, false otherwise.

Returns

full The truth of the graph being full.

Precondition

1. ∗this is a valid instantiation of a WeightedGraph.
2. The size member of ∗this is in a valid state.

Postcondition

1. The truth of the fullness of the graph is returned.

3.3.2.13 WeightedGraph & WeightedGraph::operator= (const WeightedGraph & *other*)

operator=

The overloaded assignment operator. Clones the data in the given parameter Weighted-Graph other into ∗this.

Parameters

other	A WeightedGraph to whose data will be cloned.

Returns

∗this A reference to this for multiple assignments on the same line.

Precondition

1. Both ∗this and other are valid WeightedGraph objects.

Postcondition

1. ∗this will be an equivalent object to other. All data in ∗this will be a clone of that in other.

1. If the given parameter other is ∗this, no action is taken.
2. If WeightedGraph other is of different size, ∗this is resized.
3. The data in this is then made equivalent to that of other, size and all.

3.3.2.14 void WeightedGraph::removeEdge (const string & *v1*, const string & *v2*) throw (logic_error)

removeEdge

Removes an edge from the graph by giving it an "infinite" weight. (The "infinite" weight is the maximum integer value defined in climits)

Parameters

v1	A label corresponding to the first vertex that defines an edge.
v2	A label corresponding to the second vertex that defines an edge.

Precondition

1. ∗this is a valid instantiation of a WeightedGraph.
2. Vertices with labels corresponding to the parameters v1 and v2 should exist in the graph.

Postcondition

1. If the edge exists in the graph, it is "removed" by giving it an "infinite" weight.
2. If either of the given vertex labels are not present in the vertex list, an exception is thrown to indicate that an invalid edge was sought.

1. The insertEdge function is called to "remove" the edge by giving it an infinite weight. This will prevent duplication of code.
2. If the either of the vertices that define the edge can't be found, then an exception is thrown to indicate such.

Exceptions

logic_error	Used to indicate that an invalid edge was chosen because at least one of the given vertex labels does not correspond to an existing vertex.

3.3.2.15 void WeightedGraph::removeVertex (const string & *v*) throw (logic_error)

removeVertex

Removes a vertex from the vertex list and resizes the adjacency matrix to remove any edges that were associated with the vertex.

Parameters

v	A label corresponding to the vertex to be removed.

Precondition

1. ∗this is a valid instantiation of a WeightedGraph.
2. The given label parameter should correspond to a vertex contained in the vertex list of the graph.

Postcondition

1. The vertex corresponding to the given label will be removed.
2. Data corresponding to edges shared by the removed vertex and other vertices will be removed and the adjacency matrix will be resized appropriately.

   3. The new size of the graph is updated.
   4. If the given label does not correspond to a given vertex, an exception is thrown to indicate this.

1. The sought vertex is first located in the vertex list and its index is saved.
2. The size of the graph is decremented.
3. The vertex list is then condensed by shifting all elements in the list after the removed vertex over.
4. The rows of the adjacency matrix that appear after the row corresponding to the removed vertex are moved up to replace the "removed" row.
5. The columns that appear after the one corresponding to the removed vertex are shifted left in order to replace the "removed" column.
6. If the given label is not found in the graph, an exception is thrown, indicating this.

Exceptions

logic_error	Used to indicate that an attemp to remove a vertex that is not present in the graph was made.

3.3.2.16 bool WeightedGraph::retrieveVertex (const string & *v*, Vertex & *vData*) const

retriveVertex

Searches the vertex list for a particular vertex label. Passes the vertex back by reference if it is found.

Parameters

v	A string used to identify a sought vertex.
vData	An object of type Vertex intended to contain the data of the sought vertex if the vertex with a label matching the parameter label is found.

Returns

result The truth value of whether the sought vertex was found. If the vertex was found and vData contains that vertex's information, then true is returned. - Otherwise, false is returned and vData is in an undetermined state.

Precondition

1. ∗this is a valid instantiation of a weighted graph.
2. vData is a properly constructed Vertex object.

3. A vertex with a label matching the one given as a parameter should exist in the graph.

Postcondition

1. If a vertex with a matching label is found, then its data is copied into the given reference parameter Vertex vData and true is returned. Otherwise, false is returned and vData will be in an undetermined state.

1. The vertex list is searched for a vertex with a label matching the given parameter.
2. If such a vertex is found, its data is copied to parameter Vertex vData and true is returned.
3. If a vertex with such a label is not found, then parameter Vertex vData will be in an undetermined state and false is returned.

3.3.2.17 void **WeightedGraph::setEdge** (const int *row*, const int *col*, const int *wt*) `[private]`

setEdge

Given appropriate adjacency matrix coordinates, sets the weight of the edge between two vertices.

Parameters

row	The row index of the first vertex defining the edge.
col	The column index of the second vertex defining the edge.
wt	The weight the sought edge will be given.

Precondition

1. ∗this is a valid instantiation of a WeightedGraph.
2. Parameters row and col are valid indices corresponding to a location in the adjacency matrix.

Postcondition

1. The location in the adjacency matrix representing the edge from the row index vertex to the column index vertex is given the parameterized weight.

1. The indices that correspond to a 2-D matrix can be converted using the formula: (row-index ∗ number-of-columns) + column-index.
2. The resulting index is used to set the edge's weight.

3.3.2.18 void **WeightedGraph::setPath** (const int *row*, const int *col*, const int *cost*) [private]

setPath

Given appropriate adjacency matrix coordinates, sets the cost of the path between two vertices.

Parameters

row	The row index of the first vertex defining the path.
col	The column index of the second vertex defining the path.
cost	The cost the sought path will be given.

Precondition

1. ∗this is a valid instantiation of a WeightedGraph.
2. Parameters row and col are valid indices corresponding to a location in the path matrix.

Postcondition

1. The location in the path matrix representing the edge from the row index vertex to the column index vertex is given the value of the parameter cost.

1. The indices that correspond to a 2-D matrix can be converted using the formula: (row-index ∗ number-of-columns) + column-index.
2. The resulting index is used to set the path's cost.

3.3.2.19 void **WeightedGraph::showShortestPaths** ( )

showShortestPaths

Computes and displays the graphs path matrix.

Precondition

1. ∗this is a valid instantiation of a WeightedGraph
2. The adjacency matrix is in a valid state.

Postcondition

1. The WeightedGraph will remain unchanged.
2. The shortest (lowest cost) paths between all vertices will be re-computed.
3. The shortest path matrix will be displayed.

1. The shortest paths are computed and stored in the pathMatrix data member.
2. Then the path matrix is displayed one row at a time.
3. If the graph is actually empty, this is stated.

3.3.2.20 void WeightedGraph::showStructure ( ) const

showStructure

Outputs a graph's vertex list and adjacency matrix. This operation is intended for testing/debugging purposes only.

PROVIDED BY THE LAB MANUAL PACKAGE

Precondition

1. ∗this is a valid instantiation of a WeightedGraph.

Postcondition

1. ∗this will remain unchanged.
2. The vertex list an adjacency matrix for the graph will be displayed.
3. The weighted graph will remain unchanged.

1. A vertex list is displayed, both labels and colors
2. Then the adjacency matrix is displayed one row at a time.
3. If the graph is actually empty, this is stated.

3.3.3 Member Data Documentation

3.3.3.1 int∗ WeightedGraph::adjacencyMatrix `[private]`

3.3.3.2 const char WeightedGraph::DEFAULT_COLOR = 'R' `[static]`

3.3.3.3 const int WeightedGraph::INFINITE_EDGE_WT = INT_MAX `[static]`

3.3.3.4 const int WeightedGraph::LOOP_COST = 0 [static]

3.3.3.5 const int WeightedGraph::MAX_GRAPH_SIZE = 10 [static]

3.3.3.6 int WeightedGraph::maxSize [private]

3.3.3.7 const int WeightedGraph::NOT_FOUND = -1 [static]

3.3.3.8 int* WeightedGraph::pathMatrix [private]

3.3.3.9 int WeightedGraph::size [private]

3.3.3.10 Vertex* WeightedGraph::vertexList [private]

The documentation for this class was generated from the following files:

- /nfs/home/thenriod/Desktop/PA11/Doxy11/WeightedGraph.h
- /nfs/home/thenriod/Desktop/PA11/Doxy11/WeightedGraph.cpp

Chapter 4

File Documentation

4.1 /nfs/home/thenriod/Desktop/PA11/Doxy11/config.h File - Reference

Defines

- #define LAB12_TEST1 1
- #define LAB12_TEST2 1
- #define LAB12_TEST3 1

4.1.1 Define Documentation

4.1.1.1 #define LAB12_TEST1 1

WeightedGraph class configuration file. Activate test #N by defining the corresponding LAB12_TESTN to have the value 1.

4.1.1.2 #define LAB12_TEST2 1

4.1.1.3 #define LAB12_TEST3 1

4.2 /nfs/home/thenriod/Desktop/PA11/Doxy11/test12.cpp File - Reference

```
#include <iostream>#include <cstring>#include <cctype>×
#include "WeightedGraph.h"#include "config.h"
```

Functions

- void print_help ()
- int main ()

4.2.1 Function Documentation

4.2.1.1 int main ( )

4.2.1.2 void print_help ( )

4.3 /nfs/home/thenriod/Desktop/PA11/Doxy11/WeightedGraph.cpp File Reference

Class implementations for the Weighted Graph ADT.

`#include "WeightedGraph.h" #include <cassert>`

Defines

- #define NDEBUG

4.3.1 Detailed Description

Class implementations for the Weighted Graph ADT.

Author

Terence Henriod

Lab 11: Weighted Graph ADT

Version

Original Code 1.00 (11/14/2013) - T. Henriod

4.3.2 Define Documentation

4.3.2.1 #define NDEBUG

4.4 /nfs/home/thenriod/Desktop/PA11/Doxy11/WeightedGraph.h - File Reference

Class declarations for the WeighedGraph ADT and the Vertex inner class. Utilizes both an adjacency matrix and a path matrix.

```
#include <stdexcept>#include <iostream>#include <climits>×
#include <string>
```

Classes

- class WeightedGraph
- class WeightedGraph::Vertex

Variables

- const string UNLABELED = "UNLABELED"

4.4.1 Detailed Description

Class declarations for the WeighedGraph ADT and the Vertex inner class. Utilizes both an adjacency matrix and a path matrix.

Author

Terence Henriod

Lab 11: WeightedGraph ADT

Version

Original Code 1.00 (11/14/2013) - T. Henriod

4.4.2 Variable Documentation

4.4.2.1 const string UNLABELED = "UNLABELED"