

CSC 402 - Data Structures II, Fall, 2010, Dr. Dale E. Parson

Assignments #5 & 6, Adjacency Matrix Implementations of Digraphs

The **Topological Sort** portion of this assignment is due via **gmake turnitin** by 11:59 PM on **Friday, December 3**. The **Unit-cost Shortest Path** portion of this assignment is due via **gmake turnitin** by 11:59 PM on **Saturday, December 11**. They will count as separate assignments. I will post my solution to Topological Sort and send an email on December 5, and **I will not accept late assignments after I post the solution.**

```
cd ~/AdvDataStructures
cp ~parson/digraphsASSN56.zip digraphsASSN56.zip
unzip digraphsASSN56.zip
cd ./digraphstruct
gmake clean build
```

At this point the program compiles OK, but attempting **gmake test** results in a diff error because you need to define two procedures within source file `digraph_test.cxx`. I have already written the I/O code within file `digraph_test.cxx` to read graphs from an input test file and write results to an output test file. I have also coded data structures to represent a directed graph as an *adjacency matrix*. To summarize, you will redefine two procedures to determine the topological sort of a graph (assignment 5) and to determine the Unit-cost Shortest Path from a node to other nodes in the graph (assignment 6). Last year's assignments used an *adjacency list* approach in the form of linked C++ objects for the graph and its nodes and edges. You will rewrite those procedures using the adjacency matrix data structure that is loaded by my input code.

See STUDENT comments for these two procedures in `digraph_test.cxx`:

```
static vector<int> getTopologicalSort(bool &isvalid)// assignment 5 due 12/3
static set<vector<orderedTriplet>> getPaths(string nodename) // assn 6 due 12/11
```

There are pointers to last year's solution code in those comments. Those solutions appear in `.cxx.txt` files within this directory. You can use them to see working examples of adjacency list code.

Once you have **getTopologicalSort** working, the temporary test file **topoTest.out** should show a valid topological sort. My solution from last year shows the following line:

```
v1, v2, v5, v4, v3, v7, v6,
Topological Sort: 7
```

That is not the only valid topological sort of this data set. There is a `graph.gif` file of this graph in the directory and at <http://bill.kutztown.edu/~parson/gif/graph.gif>. Inspect that to make sure that your topological sort is valid.

Turn this part in by the end of December 3.

Once you have **getPaths** working, inspect the test output to make sure that you have valid, least-cost paths similar to last year's output. The order of your path reporting may vary in the output, but the reported paths should be the minimal-cost paths, and none should be missing. Turn the final solution in by end of December 11.

In both cases **make sure to update your .ref files when you are convinced that your .out files are correct**. My makefiles sorts test output lines before applying diff:

```
./graphTest < topoTest.txt > topoTest.out 2>&1
sort < topoTest.out > topoTestSort.out
diff topoTestSort.out topoTestSort.ref > topoTestSort.dif
```

So you will need to cp topoTestSort.out topoTestSort.ref after you are certain that topoTest.out is correct.

I will be testing your solutions with additional test data files. Below is the initial listing of digraph_test.cxx.

digraphstruct/digraph_test.cxx

```
1  /*    digraph_test.cxx -- test driver for graph algorithms.
2
3      CSC402, Fall, 2009, Dr. Dale Parson.
4
5      Amended for CSC402, Fall 2010 -- This year's graph assignment
6      uses an adjacency matrix representation for the graph stored
7      in an N x N matrix that is allocated within this test driver,
8      without using any external C++ classes other than library
9      classes from iostream, STL, etc. The test input and output data
10     are the same as for ~parson/fall09AdvDataStructures/shorttestpath,
11     but the dta structures are entirely different, and self-contained
12     in this file. I supply the code for building the matrix. You must
13     write the code for using the matrix to determine topological sort
14     and shortest path determination.
15
16     In an adjacency matrix a node is an integer offset into the X and
17     Y dimensions of the matrix, and an edge is an entry in that matrix.
18     In this implementation an edge entry of 1 represents a unit-cost
19     edge from its node in the first dimension to the node in the dge
20     dimension; i.e., a value matrix[i][j] = 1 represents an edge from
21     the node i to node j with a cost of 1. A value matrix[i][j] = 0
22     represents NO EDGE from i to j; 0 is the *infinity* cost.
23     No other cost values are supported in the current assignment.
24
25     From Fall 2009:
26
```

```

27     This program reads graph definitions from cin until EOF.
28     Each line is EITHER
29         A distance B
30     where A and B are node names and distance is the integer cost
31         of an edge from A to B, OR
32         ! N
33     where N is a node name from which to compute paths, OR
34     .
35     A period is a sentinel value for end of file.
36 */
37
38 // THE FOLLOWING EXIT CODES CONSTITUTE THE RETURN VALUES FROM
MAIN:
39 static const int  SUCCESS   = 0 ; /* successful program execution */
40 static const int  USAGERR   = 1 ; /* invalid command line usage */
41 static const int  DATAERR  = 2 ; /* invalid input test data */
42
43 #include <iostream>
44 #include <fstream>
45 #include <string>
46 #include <map>
47 #include <set>
48 #include <vector>
49 using namespace std;
50
51 #include "StringToInteger.h"
52
53 /*
54     Function:      main
55
56     main is a test driver that loads the adjacency matrix and
57     runs the topological sort and shortest path procedures.
58 */
59
60 static const int MAXN = 100 ; // maximum size of N for NxN matrix.
61 static int matrix[MAXN][MAXN] ; // matrix[N][N] read from an input file
62 static int N = 0 ; // size of each dimension matrix[FROM][TO]
63 static string indexToName[MAXN] ; // indexToName[N] maps number to node name.
64 static map<string, int> nameToIndex ; // STL maps node name to number.
65
66 struct orderedTriplet { // used by getPaths to return FROM,TO,COST triplets
67     int from ; // source node for an edge
68     int to ; // destination node for an edge
69     int cost ; // cost of the edge in the matrix
70 };
71

```

```

72 // Forward declarations for STUDENT FUNCTIONS getTopologicalSort and
73 // getPaths, see below.
74 static vector<int> getTopologicalSort(bool &isvalid) ;
75 static set<vector<orderedTriplet> > getPaths(string nodename) ;
76
77 static int helpInsertNode(const string &nodename) {
78     // Places nodename in the graph, returns index into matrix.
79     // Returns -1 if there is no more room for a node,
80     // also reports this to cerr.
81     int result = -1 ;
82     map<string, int>::iterator finder = nameToIndex.find(nodename);
83     if (finder == nameToIndex.end()) {
84         // It is not already a node.
85         if (N == MAXN) {
86             cerr << "Error, no room to insert " << nodename
87                 << " into the matrix.";
88         } else {
89             result = N++ ;
90             // Static arrays come up initialized to zeroes, but here
91             // we redo that in case we later decide to reuse the matrix.
92             for (int i = 0 ; i < MAXN ; i++) {
93                 matrix[result][i] = 0 ; // no outgoing edges from node
94             }
95             indexToName[result] = nodename ;
96             nameToIndex[nodename] = result ;
97         }
98     } else {
99         result = finder->second ;
100     }
101     return result ;
102 }
103
104 int
105 main(const int argc, char *argv[]) {
106     int exitstatus = SUCCESS ;
107     string token1 ;
108     while (token1 != ".") {
109         string token2, token3 ;
110         cin >> token1 ;
111         if (token1 == ".") {
112             break ;
113         }
114         cin >> token2 ;
115         if (token1 != "!") {
116             // This is a NODE COST NODE LINE
117             bool isok, isNew ;

```

```

118     cin >> token3 ;
119     int cost = StringToInteger(token2.c_str(), isok);
120     if (!(isok) || cost != 1) {
121         cerr << "ERROR, invalid integer cost (not 1): "
122             << token2 << endl ;
123         exitstatus = DATAERR ;
124         continue ;
125     }
126     int indexof1 = helpInsertNode(token1);
127     int indexof3 = helpInsertNode(token3);
128     if (indexof1 > -1 && indexof3 > -1) {
129         matrix[indexof1][indexof3] = cost ;
130     }
131 } else {
132     // Print set of costs from node whose name is given in
133     // token2 to all other reachable nodes in the graph.
134     // Here we represent an edge as an ordered triplet of integers
135     // FROM,TO,COST in a struct orderedTriplet.
136     map<string, int>::iterator finder = nameToIndex.find(token2);
137     if (finder == nameToIndex.end()) {
138         cerr << "ERROR, no node named " << token2
139             << " for operator '!'" << endl ;
140         exitstatus = DATAERR ;
141         continue ;
142     }
143     set<vector<orderedTriplet> > pathset
144         = getPaths(token2);
145     set<vector<orderedTriplet> >::iterator psetiter
146         = pathset.begin();
147     for (; psetiter != pathset.end() ; psetiter++) {
148         vector<orderedTriplet> vedges = *psetiter ;
149         vector<orderedTriplet>::iterator vedgeiter
150             = vedges.begin();
151         if (vedgeiter != vedges.end()) { // not empty path
152             orderedTriplet nextedge = *vedgeiter ;
153             cout << indexToName[nextedge.from] << " " ;
154             int costsum = 0 ;
155             while (vedgeiter != vedges.end()) {
156                 cout << nextedge.cost << " "
157                     << indexToName[nextedge.to] << " " ;
158                 costsum += nextedge.cost ;
159                 vedgeiter++ ;
160                 if (vedgeiter != vedges.end()) {
161                     nextedge = *vedgeiter ;
162                 }
163             }

```

```

164         cout << " (Total cost is " << costsum << ".)" << endl ;
165     } else {
166         cerr << "Empty path reported for "
167             << token2 << endl ;
168         exitstatus = DATAERR ;
169     }
170 }
171 cout << endl ;
172 }
173 }
174 bool isvalid = true ;
175 vector<int> toposort = getTopologicalSort(isvalid);
176 if (isvalid) {
177     cout << endl << "Topological Sort: " << toposort.size() << endl << "\t" ;
178     for (int i = 0 ; i < toposort.size() ; i++) {
179         cout << indexToName[toposort.at(i)] << " , " ;
180     }
181     cout << endl ;
182 } else {
183     cout << endl << "Graph contains a cycle, no topological sort." << endl;
184 }
185 return(exitstatus);
186 }
187
188 // STUDENT Write getTopologicalSort() and getPaths() below.
189 // Function returns a valid topological sort of node indices in result
190 // and sets isvalid to true unless there is a cycle; if there is a cycle,
191 // return vector is empty and isvalid is set to false.
192 // See data structures and code for last year's topological sort of
193 // a linked-object graph representation in digraph_baseclass.h.txt and
194 // digraph_baseclass.cxx.txt. Note that digraph_baseclass::getTopologicalSort
195 // in digraph_baseclass.cxx.txt uses some additional STL data structures
196 // as local variables. You will have to use similar local variables,
197 // while adapting them to the adjacency matrix implementation used here.
198 /** WEISS PSEUDOCODE *****
199 void Graph::topsort()
200 {
201     Queue<Vertex> q ;
202     int counter = 0 ;
203     q.makeEmpty();
204     for each Vertex v
205         if( v.indegree == 0 )
206             q.enqueue( v );
207     while ( !q.isEmpty() )
208     {
209         Vertex v = q.dequeue( );

```

```

210         v.topNum = ++counter ; // Assign next number
211         for each Vertex w adjacent to v
212             if ( --w.indegree == 0 )
213                 q.enqueue( w );
214         }
215         if ( counter != NUM_VERTICES )
216             throw CycleFoundException();
217     }
218     See last year's code in digraph_baseclass.cxx.txt.
219     *****/
220 static vector<int> getTopologicalSort(bool &isvalid) {
221     isvalid = false ;
222     vector<int> result ;
223     // STUDENT CODE GOES BELOW.
224     return result ;
225 }
226
227 // STUDENT Write getPaths and getTopologicalSort() above.
228 // getPaths returns the set of paths from nodename to every node in
229 // the graph for which there is a path from nodename to that node.
230 // One destination may be nodename in a graph with a cycle.
231 // Each vector is a path of edge information in an orderedTriplet struct.
232 // The return value will be empty if there is no path from nodename to
233 // any other node.
234 // See data structures and code for last year's shortest path search of
235 // a linked-object graph representation in digraph_shortest_path.h.txt and
236 // digraph_shortest_path.cxx.txt. Note that digraph_baseclass::getPaths
237 // in digraph_baseclass.cxx.txt and digraph_shortest_path::getPath in
238 // digraph_shortest_path.cxx.txt use some fields of class digraph_shortest_path
239 // declared in digraph_shortest_path.h.txt, such as the "struct dist" and
240 // the mutable fields. You can redefine these fields as file-static
241 // variables below, *outside* of any function (i.e., *not* as local
242 // variables). Making them "static" keeps them invisible outside of
243 // this source file. We only need one copy of each of those previously
244 // "mutable" fields (don't make them mutable) because the current code
245 // manipulates only one graph at a time.
246
247 // I strongly recommend writing static getPath as a helper function for
248 // getPaths, patterned after the code in last year's implementation of
249 // getPath and getPaths.
250
251 // in digraph_baseclass.cxx.txt uses some additional STL data structures
252 // as local variables. You will have to use similar local variables,
253 // while adapting them to the adjacency matrix implementation used here.
254 static set<vector<orderedTriplet> > getPaths(string nodename) {
255     set<vector<orderedTriplet> > result ;

```

```

256 // STUDENT CODE GOES BELOW.
257 return result ;
258 }
259 // BELOW IS WEISS' PSEUDOCODE for getPath() for a single source-dest. path
260 /***** PSEUDOCODE FROM WEISS TEXTBOOK
261         O(V**2) unweighted shortest-path algorithm
262 void Graph::unweighted( Vertex s )
263 {
264     for each Vertex v
265     {
266         v.dist = INFINITY
267         v.known = false ;
268     }
269     s.dist = 0 ;
270     for( int currDist = 0; currDist < NUM_VERTICES; currDist++ )
271         for each Vertex v
272             if ( !v.known && v.dist == currDist )
273             {
274                 v.known = true ;
275                 for each Vertex w adjacent to v
276                     if ( w.dist == INFINITY )
277                     {
278                         w.dist = currDist + 1;
279                         w.path = v;
280                     }
281             }
282 }
283 *****/

```