graph class

```
 1: #include <iostream>
 2: #include <map>
 3: #include <set>
 4: #include <queue>
 5: #include <limits>
 6: #include <algorithm>
 7: using namespace std;
 8:
 9: template <typename Node, typename Weight = int>
10: class graph {
11: public:
12:   // MUST BE INITIALIZED JUST AFTER CLASS DEFINITION ENDS.  See lines 126-127
13:   static const Weight INF;
14:
15:   /* Value Map can be used to track node weights, to track the
16:      neighbors of a particular node together with edge weights, or in
17:      general to keep track of a set of values associated with nodes
18:      (such as the nodes' distances from a single source)
19:   */
20:   typedef map<Node, Weight> VM;             // VM:   Value Map
21:   typedef typename VM::iterator VMI;        // VMI:  Value Map Iterator
22:   typedef typename VM::const_iterator CVMI; // CVMI: const Value Map Iterator
23:
24:   /* The Adjacency Map can be used for an undirected adjacency list,
25:      as each node has adjacencies that are represented with a Value
26:      Map.  For a directed graph, would keep two adjacency maps, one
27:      for incoming adjacencies and one for outgoing.
28:   */
29:   typedef map<Node, VM> AM;                 // AM:   Adjacency Map
30:   typedef typename AM::iterator AMI;        // AMI:  Adjacency Map Iterator
31:   typedef typename AM::const_iterator CAMI; // CAMI: const Adajacency Map Iter
32:
33:   /*
34:     The Parent Map is typically used to represent tree structures by
35:     having each node map to a "parent" node (with the parent mapping
36:     to itself by convention)
37:   */
38:   typedef map<Node, Node> PM;                // PM:   Parent Map
39:   typedef typename PM::iterator PMI;         // PMI:  Parent Map Iterator
40:   typedef typename PM::const_iterator CPMI;  // CPMI: const Parent Map Iterator
41:
42: private:
43:   AM in, out;           // Adjacency maps
44:   bool dir;
45:
46: public:
47:   graph(bool dir) : dir(dir) { }            // true=directed, false=undirected
48:   bool isDirected() const { return dir; }
49:
50:   /* Note:  adding an edge that already exists overwrites weight */
51:   void addEdge(Node from, Node to, Weight wt = 1) {
52:     out[from][to] = wt;
53:     if (dir) {
54:       in[to][from] = wt;  // reverse edge
55:       out[to];            // creates empty neighborhood if 'to' is new node
56:       in[from];           // creates empty neighborhood if 'from' is new node
57:     } else {
58:       out[to][from] = wt;  // reverse edge
59:     }
60:   }
61:
62:   bool hasEdge(Node from, Node to)  const {
63:     CAMI it = out.find(from);
64:     return (it != out.end() && it->second.find(to) != it->second.end());
65:   }
```

```
 66:
 67:   Weight getEdgeWeight(Node from, Node to) const {   // UNSAFE if no such edge
 68:     const VM& dm = out.find(from)->second;
 69:     return dm.find(to)->second;
 70:   }
 71:
 72:   VM empty;   // sentinel
 73:   const VM& getNeighbors(Node node, bool outward=true) const {
 74:     const AM &adj( (dir && !outward) ? in : out);
 75:     CAMI iter = adj.find(node);
 76:     if (iter != adj.end())
 77:       return iter->second;
 78:     else
 79:       return empty;
 80:   }
 81:
 82:   int getDegree(Node node, bool outbound=true) {
 83:     return getNeighbors(node,outbound).size();
 84:   }
 85:
 86:   void reverse() {
 87:     if (dir) in.swap(out);
 88:   }
 89:
 90:   set<Node> getNodes() const {
 91:     set<Node> nodes;
 92:     for (CAMI it = out.begin(); it != out.end(); ++it)
 93:       nodes.insert(it->first);
 94:     if (dir)
 95:       for (CAMI it = in.begin(); it != in.end(); ++it)
 96:         nodes.insert(it->first);
 97:     return nodes;
 98:   }
 99:
100:   // OPTIONAL METHOD.  ONLY NECESSARY WHEN MODEL NEEDS EDGE DELETIONS
101:   void deleteEdge(Node from, Node to) {        // UNSAFE if no such edge
102:     out[from].erase(to);
103:     if (dir)
104:       in[to].erase(from);
105:     else
106:       out[to].erase(from);
107:
108:     if (out[from].size() + in[from].size() == 0) {
109:       out.erase(from);
110:       in.erase(from);
111:     }
112:
113:     if (out[to].size() + in[to].size() == 0) {
114:       out.erase(to);
115:       in.erase(to);
116:     }
117:   }
118:
119:
120:   //  INSERT ADDITIONAL SUPPORT FOR ALGORITHMS HERE
121:
122:
123: };  // end of graph class
124:
125: // must initialize INF constant
126: template <typename Node, typename Weight>
127: const Weight graph<Node,Weight>::INF = numeric_limits<Weight>::max();
```

```
128:   //-----------------  Debug Utilities ------------------------
129:
130:   void printNeighbors(string node) const {
131:     cout << node << "'s ";
132:     if (isDirected()) cout << "outgoing ";
133:     cout << "neighbors are:" << endl;
134:     const VM &hNeigh(getNeighbors(node));
135:     for (CVMI it = hNeigh.begin(); it != hNeigh.end(); ++it) {
136:       cout << "  " << it->first << " with edge weight " << it->second << endl;
137:     }
138:
139:     if (isDirected()) {
140:       cout << node << "'s incoming neighbors are:" << endl;
141:       const VM &hNeigh(getNeighbors(node, false));
142:       for (CVMI it = hNeigh.begin(); it != hNeigh.end(); ++it) {
143:         cout << "  " << it->first << " with edge weight " << it->second << endl;
144:       }
145:     }
146:   }
147:
148:   void debugDump() const {
149:     cout << endl << "DEBUG DUMP" << endl;
150:     set<string> nodes = getNodes();
151:     typename set<string>::const_iterator it;
152:     for (it=nodes.begin(); it != nodes.end(); ++it) {
153:       cout << "Node " << *it << endl;
154:       printNeighbors(*it);
155:     }
156:   }
```

```
157:   //-----------------  Shortest Path Support ------------------------
158:
159:   /**
160:    *  result[node] equals the distance from source to node.
161:    *  graph::INF is used if node was unreachable.
162:    */
163:   VM compute_SP_distances(Node source) const {
164:     set<Node> nodes = getNodes();
165:     typename set<Node>::iterator it;
166:     VM estimate;
167:     for (it = nodes.begin(); it != nodes.end(); ++it)
168:       estimate[*it] = ( (*it == source) ? 0 : INF);
169:
170:     int V = nodes.size();
171:     bool changed = true;
172:     int passes = 0;
173:     while (changed && passes < V-1) {
174:       changed = false;
175:       for (it = nodes.begin(); it != nodes.end(); ++it) {
176:         if (estimate[*it] != INF) {
177:           const VM& neighbors = getNeighbors(*it);
178:           for (CVMI d = neighbors.begin(); d != neighbors.end(); ++d) {
179:             if (estimate[*it] + d->second < estimate[d->first]) {
180:               changed = true;
181:               estimate[d->first] = estimate[*it] + d->second;
182:             }
183:           }
184:         }
185:       }
186:       passes++;
187:     }
188:     return estimate;
189:   }
190:
191:   /**
192:    *  result[node] is the parent of node in shortest-path tree.
193:    *  Note:  result[node] = node for root and unreachable nodes.
194:    */
195:   PM compute_SP_parents(const VM& sp) const {
196:     PM parent;
197:     for (CVMI it = sp.begin(); it != sp.end(); ++it) {
198:       parent[it->first] = it->first;   // by default, be own parent
199:       const VM& neighbors = getNeighbors(it->first);
200:       for (CVMI d = neighbors.begin(); d != neighbors.end(); ++d) {
201:         if (sp.find(d->first)->second + d->second == sp.find(it->first)->second)
202:           parent[it->first] = d->first;
203:       }
204:     }
205:     return parent;
206:   }
```

```
207:   //------------------  Minimum Spanning Tree Support ------------------------
208:
209:   /* result[node] = parent of that node in MST, or self if root */
210:   PM computeMST() const {
211:     typedef multimap<Weight, Node> PQ;
212:     typedef typename PQ::iterator pqIter;
213:     typedef map<Node, pqIter>  Trace;
214:     PQ fringe;
215:     Trace locators;
216:     PM parent;
217:
218:     // initially, all nodes have priority infinity except a chosen root
219:     set<Node> nodes = getNodes();
220:     typename set<Node>::iterator it;
221:     Node root = *nodes.begin();
222:     locators[root] = fringe.insert(make_pair(0, root));
223:     parent[root] = root;
224:     for (it = ++nodes.begin(); it != nodes.end(); ++it) {
225:       locators[*it] = fringe.insert(make_pair(INF, *it));
226:       parent[*it] = *it;
227:     }
228:
229:     while (!fringe.empty()) {
230:       pqIter small = fringe.begin();
231:       Node removed = small->second;
232:       fringe.erase(small);
233:       locators.erase(locators.find(removed));
234:
235:       // check neighbors of removed
236:       const VM& neighbors = getNeighbors(removed);
237:       for (CVMI d = neighbors.begin(); d != neighbors.end(); ++d) {
238:         Node dest = d->first;
239:         Weight edgecost = d->second;
240:         typename Trace::iterator loc = locators.find(dest);
241:         if (loc != locators.end() && edgecost < loc->second->first) {
242:           // found better edge to dest;  remove/reinsert PQ entry
243:           fringe.erase(loc->second);
244:           locators[dest] = fringe.insert(make_pair(edgecost, dest));
245:           parent[dest] = removed;
246:         }
247:       }
248:     }
249:
250:     return parent;
251:   }
252:
253:   /* computes the total cost of the MST returned by computeMST */
254:   Weight computeMSTWeight(const PM& mst) const {
255:     Weight wt = 0;
256:     for (CPMI it = mst.begin(); it != mst.end(); ++it) {
257:       if (it->first != it->second)
258:         wt += getEdgeWeight(it->second, it->first);
259:     }
260:     return wt;
261:   }
```

```
262:   //-------------  Graph Traversals, Reachability, Connectivity  ---------------
263:
264:   /** Note:  unreachable nodes will not be included in PM **/
265:   PM bfs(Node start) const {
266:     queue<Node> itemsToVisit;
267:     PM visitedNodes; //first is the node visited; second is it's parent
268:     itemsToVisit.push(start);
269:     visitedNodes[start] = start;
270:     while (!itemsToVisit.empty()) {
271:       Node item = itemsToVisit.front();
272:       itemsToVisit.pop();
273:       VM neighbors = getNeighbors(item);
274:       for (VMI iter = neighbors.begin(); iter != neighbors.end(); ++iter) {
275:         if (visitedNodes.find(iter->first) == visitedNodes.end()) {
276:           visitedNodes[iter->first] = item;
277:           itemsToVisit.push(iter->first);
278:         }
279:       }
280:     }
281:     return visitedNodes;
282:   }
283:
284:   /** Is destination reachable from source? */
285:   bool reachable(Node source, Node to) const {
286:     PM tree = bfs(source);
287:     return (tree.find(to) != tree.end());
288:   }
289:
290:   //======= remainder of this section is for dfs and connected components ======
291:
292:   /* node's parent should be recorded BEFORE this call */
293:   void dfsUtil(Node node, Node from, PM& parent, vector<Node>& finished) const {
294:     if (parent.find(node) == parent.end()) {
295:       parent[node] = from;   // mark node as visited
296:       VM neighbors = getNeighbors(node);
297:       for (VMI iter = neighbors.begin(); iter != neighbors.end(); ++iter) {
298:         if (parent.find(iter->first) == parent.end()) {
299:           dfsUtil(iter->first, node, parent, finished);
300:         }
301:       }
302:       finished.push_back(node);
303:     }
304:   }
305:
306:   /**
307:    * Returns the DFS-tree and a vector of finish times.
308:    * Originally begins at node s.
309:    * If all nodes are not yet visit, restarts based upon order
310:    * from iterator restart to end.
311:    */
312:   template <typename Container>
313:   pair<PM, vector<Node> > dfsHint(Node s, const Container& order) const {
314:     PM parent;
315:     vector<Node> finished;
316:     typename Container::const_iterator walk(order.begin());
317:     dfsUtil(s, s, parent, finished);
318:     while (walk != order.end()) {
319:       if (parent.find(*walk) == parent.end())
320:         dfsUtil(*walk, *walk, parent, finished);
321:       ++walk;
322:     }
323:     return make_pair(parent, finished);
324:   }
325:
```

graph class

```
  /** Note:  unreachable nodes will not be included in PM **/
  PM dfs(Node start) const {
    return dfsHint(start, getNodes()).first;
  }

  /* pointer hop to root of tree */
  void traceRoot(Node n, PM& roots) const {
    if (roots[n] != n) {
      traceRoot(roots[n], roots);
      roots[n] = roots[roots[n]];
    }
  }

  PM computeComponentsUtil(const PM& tree) const {
    PM result(tree);
    for (CPMI it = tree.begin(); it != tree.end(); ++it)
      traceRoot(it->first, result);
    return result;
  }

  /* Result actually maps each node to the "captain" of its component */
  PM computeComponents() const {
    PM tree;
    if (!dir) {
      tree = dfsHint(out.begin()->first, getNodes()).first;
    } else {
      const_cast<graph*>(this)->reverse();
      vector<Node> finish = dfsHint(out.begin()->first, getNodes()).second;
      std::reverse(finish.begin(), finish.end());
      const_cast<graph*>(this)->reverse();
      tree = dfsHint(finish.front(), finish).first;
    }
    return computeComponentsUtil(tree);
  }
```

```
360:   //------------------  Network Flow Support ------------------------
361:   //
362:   // NOTE:  Need to include code for 'bfs' graph traversal function
363:
364:   // using Edmonds-Karp version of Ford-Fulkerson
365:   graph<Node,Weight> computeMaxFlow(Node from, Node to) const {
366:     graph<Node,Weight> flow(true);
367:     const set<Node>& nodes = getNodes();
368:     typename set<Node>::const_iterator a;
369:
370:     // make initial flow with all zero weights
371:     for (a=nodes.begin(); a != nodes.end(); ++a) {
372:       const VM& neigh(getNeighbors(*a));
373:       for (CVMI b = neigh.begin(); b != neigh.end(); ++b) {
374:         flow.addEdge(*a, b->first, 0);
375:       }
376:     }
377:
378:     bool improving;
379:     while (true) {
380:       graph<Node,Weight> residual(true);
381:       for (a=nodes.begin(); a != nodes.end(); ++a) {
382:         const VM& neigh(getNeighbors(*a));
383:         for (CVMI b = neigh.begin(); b != neigh.end(); ++b) {
384:           Weight capacity(getEdgeWeight(*a, b->first));
385:           Weight fwd(flow.getEdgeWeight(*a, b->first));
386:           if (capacity > fwd)
387:             residual.addEdge(*a, b->first, capacity-fwd);  // remaining capacity
388:           if (fwd > 0)
389:             residual.addEdge(b->first, *a, fwd);           // reverse edge
390:         }
391:       }
392:
393:       PM aug = residual.bfs(from);
394:       if (aug.find(to) == aug.end()) break;                // no augmenting path
395:       Weight bottle = INF;
396:       Node walk(to);
397:       while (walk != from) {
398:         Node parent = aug[walk];
399:         bottle = min(bottle, residual.getEdgeWeight(parent,walk));
400:         walk = parent;
401:       }
402:
403:       walk = to;
404:       while (walk != from) {
405:         Node parent = aug[walk];
406:         if (flow.hasEdge(walk, parent) && flow.getEdgeWeight(walk,parent)>0)
407:           flow.addEdge(walk, parent, flow.getEdgeWeight(walk,parent)-bottle);
408:         else
409:           flow.addEdge(parent, walk, bottle + flow.getEdgeWeight(parent,walk));
410:         walk = parent;
411:       }
412:     }
413:     return flow;
414:   }
415:
416:   // NOTE: invoked on the FLOW, perhaps as g.computeMaxFlow(from,to).flowVal()
417:   Weight flowVal(Node src) const {
418:     Weight total = 0;
419:     VM neigh(getNeighbors(src));
420:     for (CVMI it = neigh.begin(); it != neigh.end(); ++it)
421:       total += it->second;
422:     return total;
423:   }
```

```
424: //============================================================================
425: /*
426:  * Rest of this file shows sample usage.
427:  *
428:  * Since we are specifically interested in graph<string,int> and want to
429:  * save future typing, we provide convenient typedef for remainder of program.
430:  */
431:
432: typedef graph<string,int> G;   // from now on we can simply say 'G' for typename
433:
434: G exampleUndirected() {   // graph from past csci 314 midterm
435:   G g(false);
436:
437:   g.addEdge("A", "D", 1);
438:   g.addEdge("A", "E", 3);
439:   g.addEdge("A", "G", 9);
440:   g.addEdge("B", "D", 11);
441:   g.addEdge("B", "E", 14);
442:   g.addEdge("B", "H", 12);
443:   g.addEdge("C", "D", 6);
444:   g.addEdge("C", "F", 2);
445:   g.addEdge("C", "G", 13);
446:   g.addEdge("C", "H", 4);
447:   g.addEdge("D", "E", 5);
448:   g.addEdge("D", "G", 8);
449:   g.addEdge("D", "H", 7);
450:   g.addEdge("F", "H", 10);
451:   return g;
452: }
453:
454: G exampleDirected() {             // graph from CLRS page 589
455:   G g(true);   // directed graph
456:
457:   g.addEdge("s", "t", 6);
458:   g.addEdge("s", "y", 7);
459:   g.addEdge("t", "x", 5);
460:   g.addEdge("t", "y", 8);
461:   g.addEdge("t", "z", -4);
462:   g.addEdge("x", "t", -2);
463:   g.addEdge("y", "x", -3);
464:   g.addEdge("y", "z", 9);
465:   g.addEdge("z", "s", 2);
466:   g.addEdge("z", "x", 7);
467:   return g;
468: }
469:
470: G exampleConnectivity() {         // graph from CLRS page 553
471:   G g(true);   // directed graph
472:
473:   g.addEdge("a", "b");
474:   g.addEdge("b", "c");
475:   g.addEdge("b", "e");
476:   g.addEdge("b", "f");
477:   g.addEdge("c", "d");
478:   g.addEdge("c", "g");
479:   g.addEdge("d", "c");
480:   g.addEdge("d", "h");
481:   g.addEdge("e", "a");
482:   g.addEdge("e", "f");
483:   g.addEdge("f", "g");
484:   g.addEdge("g", "f");
485:   g.addEdge("g", "h");
486:   return g;
487: }
```

```
488:
489: G exampleFlow() {                  // graph from CLRS page 645
490:   G g(true);   // directed graph
491:
492:   g.addEdge("s",  "v1", 16);
493:   g.addEdge("s",  "v2", 13);
494:   g.addEdge("v1", "v2", 10);
495:   g.addEdge("v2", "v1", 4);
496:   g.addEdge("v1", "v3", 12);
497:   g.addEdge("v2", "v4", 14);
498:   g.addEdge("v3", "v2", 9);
499:   g.addEdge("v3", "t", 20);
500:   g.addEdge("v4", "v3", 7);
501:   g.addEdge("v4", "t", 4);
502:
503:   return g;
504: }
505:
506: void spTest(const G& g, string source) {
507:   cout << endl << "Shortest path distances from " << source << ":" << endl;
508:   G::VM sp = g.compute_SP_distances(source);
509:   G::PM parent = g.compute_SP_parents(sp);
510:   for (G::VMI it = sp.begin(); it != sp.end(); ++it)
511:     cout << "  " << it->first << " is at distance " << it->second
512:          << " via " << parent[it->first] << endl;
513:
514: }
515:
516: void mstTest(const G& g) {
517:   G::PM mst = g.computeMST();
518:   int mstCost = g.computeMSTWeight(mst);
519:   cout << endl << "Minimum Spanning Tree has weight " << mstCost << endl;
520:   cout << "Edges are:" << endl;
521:   for (G::PMI it = mst.begin(); it != mst.end(); ++it)
522:     if (it->first != it->second) {
523:       cout << "(" << it->first << "," << it->second <<") with weight "
524:            << g.getEdgeWeight(it->second, it->first) << endl;
525:     }
526: }
527:
528: void connectivityTest(const G& g) {
529:   cout << endl;
530:   if (g.isDirected()) cout << "Strongly ";
531:   cout << "Connected Components:" << endl;
532:   G::PM captains = g.computeComponents();
533:   for (G::PMI it = captains.begin(); it != captains.end(); ++it)
534:     cout << "  node " << it->first << " in component " << it->second << endl;
535:
536:   cout << endl << "testing reachability from " << captains.begin()->first
537:        << " to " << captains.rbegin()->first << endl;
538:   if (g.reachable(captains.begin()->first, captains.rbegin()->first))
539:     cout << "REACHABLE" << endl;
540:   else
541:     cout << "UNREACHABLE" << endl;
542:
543:   cout << endl << "testing reachability from " << captains.rbegin()->first
544:        << " to " << captains.begin()->first << endl;
545:   if (g.reachable(captains.rbegin()->first, captains.begin()->first))
546:     cout << "REACHABLE" << endl;
547:   else
548:     cout << "UNREACHABLE" << endl;
549: }
550:
551: void flowTest(const G& g, const string& from, const string& to) {
552:   cout << endl;
```

```
553:   G flow(g.computeMaxFlow(from,to));
554:   int value = flow.flowVal(from);
555:   cout << "Flow is " << value << endl;
556: }
```

```
557: int main() {
558:   G g(true);
559:   string source;
560:
561:   g = exampleUndirected();
562:   source = "H";
563:   g.printNeighbors(source);
564:   spTest(g, source);
565:   mstTest(g);
566:
567:   cout << endl << endl;
568:   g = exampleDirected();
569:   source = "s";
570:   g.printNeighbors(source);
571:   spTest(g, source);
572:
573:   cout << endl;
574:   g = exampleConnectivity();
575:   G::PM result = g.dfs("b");
576:   for (G::PMI it = result.begin(); it != result.end(); ++it)
577:     cout << "parent of " << it->first << " is " << it->second << endl;
578:
579:   // more complex dfsHint demonstration
580:   cout << endl;
581:   pair<G::PM, vector<string> > details = g.dfsHint("b", g.getNodes());
582:   for (int i=0; i<details.second.size(); i++)
583:     cout << "finish order " <<  details.second[i] << endl;
584:
585:   connectivityTest(g);
586:
587:   g = exampleFlow();
588:   flowTest(g, "s", "t");
589: }
590:
591: /***  test output
592:
593: H's neighbors are:
594:   B with edge weight 12
595:   C with edge weight 4
596:   D with edge weight 7
597:   F with edge weight 10
598:
599: Shortest path distances from H:
600:   A is at distance 8 via D
601:   B is at distance 12 via H
602:   C is at distance 4 via H
603:   D is at distance 7 via H
604:   E is at distance 11 via A
605:   F is at distance 6 via C
606:   G is at distance 15 via D
607:   H is at distance 0 via H
608:
609: Minimum Spanning Tree has weight 35
610: Edges are:
611: (B,D) with weight 11
612: (C,D) with weight 6
613: (D,A) with weight 1
614: (E,A) with weight 3
615: (F,C) with weight 2
616: (G,D) with weight 8
617: (H,C) with weight 4
618:
619:
620: s's outgoing neighbors are:
```

```
621:   t with edge weight 6
622:   y with edge weight 7
623:   s's incoming neighbors are:
624:   z with edge weight 2
625:
626:   Shortest path distances from s:
627:   s is at distance 0 via s
628:   t is at distance 2 via t
629:   x is at distance 4 via x
630:   y is at distance 7 via z
631:   z is at distance -2 via z
632:
633: parent of a is e
634: parent of b is b
635: parent of c is b
636: parent of d is c
637: parent of e is b
638: parent of f is g
639: parent of g is c
640: parent of h is d
641:
642: finish order h
643: finish order d
644: finish order f
645: finish order g
646: finish order c
647: finish order a
648: finish order e
649: finish order b
650:
651: Strongly Connected Components:
652:   node a in component a
653:   node b in component a
654:   node c in component c
655:   node d in component c
656:   node e in component a
657:   node f in component f
658:   node g in component f
659:   node h in component h
660:
661: testing reachability from a to h
662: REACHABLE
663:
664: testing reachability from h to a
665: UNREACHABLE
666:
667: Flow is 23
668:
669: */
```