

CSC 402 - Data Structures II, Fall, 2009

Digraph infrastructure and Dijkstra's algorithm, Dr. Dale E. Parson

http://en.wikipedia.org/wiki/Dijkstra's_algorithm

The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and extract minimum from Q is simply a linear search through all vertices in Q . In this case, the running time is $O(|V|^2 + |E|) = O(|V|^2)$.

For [sparse graphs](#), that is, graphs with far fewer than $O(|V|^2)$ edges, Dijkstra's algorithm can be implemented more efficiently by storing the graph in the form of [adjacency lists](#) and using a [binary heap](#), [pairing heap](#), or [Fibonacci heap](#) as a [priority queue](#) to implement extracting minimum efficiently. With a binary heap, the algorithm requires $O((|E| + |V|)\log |V|)$ time (which is dominated by $O(|E| \log |V|)$, assuming the graph is connected), and the [Fibonacci heap](#) improves this to $O(|E| + |V| \log |V|)$.

```
1 function Dijkstra(Graph, source):
2   for each vertex v in Graph: // Initializations
3     dist[v] := infinity // Unknown distance function from source to v
4     previous[v] := undefined // Previous node in optimal path from source
5   dist[source] := 0 // Distance from source to source
6   Q := the set of all nodes in Graph // All nodes in the graph are unoptimized - thus are in Q
7   while Q is not empty: // The main loop
8     u := vertex in Q with smallest dist[]
9     if dist[u] = infinity:
10      break // all remaining vertices are inaccessible from source
11    remove u from Q
12    for each neighbor v of u: // where v has not yet been removed from Q.
13      alt := dist[u] + dist_between(u, v)
14      if alt < dist[v]: // Relax (u,v,a)
15        dist[v] := alt
16        previous[v] := u
17  return dist[]
```

`/export/home/faculty/parson/AdvDataStructures/dijkstra`

```
1 /* digraph_baseclass.h -- Abstract base class declarations.
2
3   CSC402, Fall, 2009, Dr. Dale Parson.
4 */
5
6 #ifndef DIGRAPH_BASECLASS_H
7 #define DIGRAPH_BASECLASS_H
8 #include <iostream>
9 #include <string>
10 #include <set>
```

```
11 #include <map>
12 #include <vector>
13 using namespace std;
14
15 /* Class: digraph_baseclass
16
17 Abstract class that implements a directed graph.
18 This demonstration class consists strictly of
19 named nodes (vertices) and edges connecting those nodes.
20 It does not contain any application data.
21 Students could add application data as a template
22 type if desired. This class is abstract because operation
23 getPath must be implemented in a concrete subclass using an
24 algorithm such as Dijkstra's algorithm.
25 */
26 class digraph_baseclass {
27 public:
28 /**
29  * Inner class node is a concrete helper class
30  * representing a node (vertex) in a digraph. A node has
31  * a unique name and a set of outgoing edges.
32  */
33 class edge; // Forward declaration.
34 class node {
35 public:
36 /**
37  * Return the name of this node, which is unique
38  * within the graph.
39  */
40 virtual string getName() const;
41 /**
42  * Return a copy of the set of edges that are have
43  * this node as a source (outgoing edges).
44  */
45 virtual set<edge *> getEdges() const;
46
47 protected:
48 // Only digraph_baseclass should delete a node.
49 // A subclass should only extend the destructor.
50 friend class digraph_baseclass;
51 virtual ~node(); // Destructor deletes the outgoing edges.
52 node(string newname) { name = newname; }
53 string name;
54 set<edge *> edges;
55 private:
56 node(); // not implemented
```

```

57     node(const node &other) ;    // not implemented
58     node& operator=(const node& other); // not implemented
59 };
60 /**
61  * Inner class edge is a concrete helper class
62  * representing a directed edge in a digraph. An edge has
63  * a source node from the graph for its start, a sink
64  * node for its destination, and an integer cost.
65  * Negative costs are permitted, although some graph
66  * manipulation algorithms in subclasses cannot deal with negative
67  * costs. They must filter such costs.
68  */
69 class edge {
70 public:
71     /**
72     * Return the source node of this directed edge.
73     * An “enter” edge into some directed graphs may return
74     * null from getSource().
75     */
76     virtual node * getSource() const ;
77     /**
78     * Return the sink (destination) node of this directed edge.
79     * An “exit” edge from some directed graphs may return
80     * null from getSink().
81     */
82     virtual node * getSink() const ;
83     /**
84     * Return the cost of this directed edge.
85     * Cost defaults to 0 in graphs that do not support costed
86     * edges, and will be non-negative for subclasses that do not
87     * support negative costs.
88     */
89     virtual int getCost() const ;
90 protected:
91     friend class digraph_baseclass ;
92     friend class digraph_baseclass::node ;
93     // Only digraph_baseclass or a destructing node should delete an edge.
94     // A subclass should only extend the destructor.
95     virtual ~edge() {} // Default destructor does nothing.
96     edge(node *newsource, node * newsink, int newcost) {
97         source = newsource ;
98         sink = newsink ;
99         cost = newcost ;
100     }
101     node * source ;
102     node * sink ;

```

```

103     int cost ;
104 private:
105     edge() ; // not implemented
106     edge(const edge &other) ; // not implemented
107     edge& operator=(const edge& other); // not implemented
108 };
109 /**
110  * Make a new named node.
111  *
112  * Parameter newname is the name.
113  *
114  * Output parameter isNew returns true if the node is new, false
115  * if it already existed in the graph.
116  *
117  * Return a pointer to a node.
118  */
119 virtual node *makeNode(string newname, bool & isNew);
120 /**
121  * Make a new edge. If there is already an edge from source to sink,
122  * makeNode replaces its cost with the newcost.
123  *
124  * Parameter newsource is the source node. It may be NULL on an
125  * entry into the graph.
126  *
127  * Parameter newsink is the sink node. It may be NULL on an
128  * exit from the graph.
129  *
130  * Parameter newcost is the edge’s cost.
131  *
132  * Output parameter isNew returns true if the edge is new, false
133  * if it already existed in the graph, in which case its cost is set
134  * to newcost.
135  *
136  * Return a pointer to an edge. If either a non-NULL newsource or
137  * newsink is not a node in this digraph object, or if the cost is invalid
138  * for this graph subclass, or if both newsource and newsink are NULL,
139  * makeEdge sets isNew to false and returns a NULL pointer.
140  */
141 virtual edge *makeEdge(node *newsource, node *newsink, int newcost,
142     bool & isNew) ;
143 /**
144  * Return a path from a start node to a destination node as a vector
145  * of edges, where index [0] is the initial edge out of the start.
146  * Derived classes typically return the least expensive path in terms
147  * of cost.
148  *

```

```

149  * Parameter start is the start node for the path. A NULL start
150  * constitutes an entry into the graph (source == NULL).
151  *
152  * Parameter end is the destination node for the path. A NULL end
153  * constitutes an exit from the graph (sink == NULL).
154  *
155  * Return a vector, which will be empty if there is no path from
156  * start to destination. Note that if removeEdge or removeNode
157  * is used to remove edges or nodes from a graph after a call to
158  * getPath returns a pointer to such an edge or node, the caller
159  * should no longer use the values returned by that earlier call to
160  * getPath, because it will contain dangling pointers to deleted objects.
161  **/
162  virtual vector<edge *> getPath(node *start, node *end) const = 0 ;
163  /**
164  * getPaths invokes getPath for the supplied start node, which may be
165  * null for a graph entry, and every sink node as the end parameter.
166  * It returns a set of paths for each start-end pair of nodes
167  * using the start parameter, which may be an empty set. It returns an
168  * empty set if start is not in this graph.
169  **/
170  virtual set<vector<edge *>> getPaths(node *start);
171  /**
172  * Return a copy of the set of edges that enter this graph from
173  * the outside, i.e., they have NULL for the source node.
174  * Note to derived class implementors: They are stored within the
175  * entrynode field.
176  */
177  virtual set<edge *> getEntryEdges() const ;
178  /**
179  * Return a copy of the set of nodes in this graph.
180  */
181  virtual set<node *> getNodes() const ;
182  /**
183  * Return a node based on its name, or NULL if it is not in the graph.
184  */
185  virtual node * getNode(string nodename) ;
186  /**
187  * Dump a graphviz dot graph to ostream.
188  **/
189  void dump(ostream & ostream) const ;
190  /**
191  * Default 0-parameter constructor initializes an empty graph.
192  **/
193  digraph_baseclass() : entrynode(“”) {
194  // NOTE TO DERIVED CLASS IMPLEMENTORS: entrynode, which is a

```

```

195  // pseudonode outside the graph holding edges into the graph,
196  // is added by makeEdge only if an edge into the graph is added.
197  // nodeset.insert(&entrynode);
198  // nodemap[“”] = &entrynode ;
199  }
200  protected:
201  map<string, node *> nodemap ; // maps unique node name to node
202  set<node *> nodeset ; // set of nodes in this graph
203  // NOTE TO DERIVED CLASS IMPLEMENTORS: entrynode, which is a
204  // pseudonode outside the graph holding edges into the graph,
205  // is added by makeEdge only if an edge into the graph is added.
206  mutable node entrynode ; // Used on NULL-source entry into this graph.
207  // Derived classes should use entrynode when getPath’s start is NULL.
208  private:
209  // Not implemented:
210  digraph_baseclass(const digraph_baseclass &other) ;
211  digraph_baseclass& operator=(const digraph_baseclass& other);
212  };
213
214  #endif

1  /*  digraph_baseclass.cxx -- Method implementations for
2  digraph_baseclass.h.
3
4  CSC402, Fall, 2009, Dr. Dale Parson.
5  */
6
7  #include “digraph_baseclass.h”
8
9  string
10 digraph_baseclass::node::getName() const {
11     return name ;
12 }
13
14 set<digraph_baseclass::edge *>
15 digraph_baseclass::node::getEdges() const {
16     return edges ;
17 }
18
19 digraph_baseclass::node::~~node() {
20     set<edge *>::iterator eiter = edges.begin();
21     while (eiter != edges.end()) {
22         edge *nextedge = *eiter ;
23         eiter++ ;
24         delete nextedge ;
25     }

```

```

26 }
27
28 digraph_baseclass::node *
29 digraph_baseclass::edge::getSource() const {
30     return source ;
31 }
32
33 digraph_baseclass::node *
34 digraph_baseclass::edge::getSink() const {
35     return sink ;
36 }
37
38 int
39 digraph_baseclass::edge::getCost() const {
40     return cost ;
41 }
42
43 digraph_baseclass::node *
44 digraph_baseclass::makeNode(string newname, bool & isNew) {
45     node *result = 0 ;
46     if (newname == "") {
47         // Reserved string for entrynode, make sure it is in graph a priori.
48         nodeset.insert(&entrynode);
49         nodemap[""] = &entrynode ;
50     }
51     map<string, node *>::iterator oldnodeiter = nodemap.find(newname);
52     if (oldnodeiter == nodemap.end()) {
53         // This is a new node with no outgoing edges yet.
54         isNew = true ;
55         result = new node(newname);
56         nodemap[newname] = result ;
57         nodeset.insert(result);
58     } else {
59         isNew = false ;
60         result = oldnodeiter->second ;
61     }
62     return result ;
63 }
64
65 digraph_baseclass::edge *
66 digraph_baseclass::makeEdge(digraph_baseclass::node *newsource,
67     digraph_baseclass::node *newsink, int newcost, bool & isNew) {
68     digraph_baseclass::node *src = 0 ;
69     if (newsource == 0) {
70         if (newsink == 0) {
71             // It cannot be both an entry and exit edge.

```

```

72     isNew = false ;
73     return 0 ; // failure indicator.
74 }
75 nodeset.insert(&entrynode); // In case it is not already there.
76 nodemap[""] = &entrynode ;
77 src = &entrynode ;
78 } else {
79     set<node *>::iterator srciter = nodeset.find(newsource);
80     if (srciter == nodeset.end()) {
81         isNew = false ;
82         return 0 ; // It is not in this graph.
83     }
84     src = newsource ;
85 }
86 if (newsink != 0) {
87     set<node *>::iterator sinkiter = nodeset.find(newsink);
88     if (sinkiter == nodeset.end()) {
89         isNew = false ;
90         return 0 ; // It is not in this graph.
91     }
92     if (newsink == &entrynode) {
93         isNew = false ;
94         return 0 ; // It is not an actual node WITHIN this subgraph.
95     }
96 }
97 set<edge *>::iterator eiter = src->edges.begin();
98 while (eiter != src->edges.end()) {
99     if ((*eiter)->sink == newsink) {
100         // It is a repeat ;
101         (*eiter)->cost = newcost ;
102         isNew = false ;
103         return *eiter ;
104     }
105     eiter++ ;
106 }
107 isNew = true ;
108 edge *result = new edge(src, newsink, newcost);
109 src->edges.insert(result);
110 return result ;
111 }
112
113 set<vector<digraph_baseclass::edge *>>
114 digraph_baseclass::getPaths(node *start) {
115     set<vector<edge *>> result ;
116     digraph_baseclass::node *src = 0 ;
117     if (start == 0) {

```

```

118     src = &entrynode ;
119 } else {
120     set<node *>::iterator srciter = nodeset.find(start);
121     if (srciter == nodeset.end()) {
122         return result ; // It is not in this graph.
123     }
124     src = start ;
125 }
126 for (set<node *>::iterator niter = nodeset.begin() ;
127      niter != nodeset.end() ; niter++) {
128     node *end = *niter ;
129     vector<edge *> onepath = getPath(start, end);
130     if (onepath.size() > 0) {
131         result.insert(onepath);
132     }
133 }
134 return result ;
135 }
136
137 set<digraph_baseclass::edge *>
138 digraph_baseclass::getEntryEdges() const {
139     return(entrynode.edges);
140 }
141
142 set<digraph_baseclass::node *>
143 digraph_baseclass::getNodes() const {
144     return nodeset ;
145 }
146
147 digraph_baseclass::node *
148 digraph_baseclass::getNode(string nodename) {
149     map<string, node *>::iterator oldnodeiter = nodemap.find(nodename);
150     if (oldnodeiter != nodemap.end()) {
151         return oldnodeiter->second ;
152     }
153     return 0 ;
154 }
155
156 void
157 digraph_baseclass::dump(ostream & ostream) const {
158     ostream << "digraph digraph_baseclass {" << endl ;
159     // ostream << "node [shape = record]" << endl ;
160     set<node *>::iterator niter = nodeset.begin();
161     while (niter != nodeset.end()) {
162         node *nextnode = *niter ;
163         niter++ ;

```

```

164     /*** entrynode is now only added when needed
165     if (nextnode == &entrynode && nextnode->edges.size() == 0) {
166         // Pseudonode entry node is never used.
167         continue ;
168     }
169     ***/
170     ostream << " \'" << nextnode->getName()
171         << "\'" [label=\'"] << nextnode->getName() << "\'" ; << endl ;
172     set<edge *>::iterator eiter = nextnode->edges.begin();
173     while (eiter != nextnode->edges.end()) {
174         edge *nextedge = *eiter ;
175         eiter++ ;
176         string destname("");
177         if (nextedge->sink != 0) {
178             destname = nextedge->sink->getName();
179         }
180         ostream << " \'" << nextnode->getName()
181             << "\'" -> \'" << destname
182             << "\'" [label=\'"] << nextedge->cost << "\'" ; << endl ;
183     }
184 }
185 ostream << "}" << endl ;
186 }

```

```

1  /* digraph_dijkstra.h -- Concrete class that implements
2     method digraph_baseclass::getPath() using Dijkstra's
3     algorithm.
4
5     CSC402, Fall, 2009, Dr. Dale Parson.
6 */
7
8 #ifndef DIGRAPH_DIJKSTRA_H
9 #define DIGRAPH_DIJKSTRA_H
10 #include "digraph_baseclass.h"
11
12 /* Class: digraph_baseclass
13
14 Concrete class that implements the getPath() method of
15 abstract class digraph_baseclass using Dijkstra's algorithm,
16 see http://en.wikipedia.org/wiki/Dijkstra%27s\_algorithm.
17 This class also disallows negative costs for edges.
18 */
19 class digraph_dijkstra : public digraph_baseclass {
20 public:
21     /**
22      * Make a new edge as long as cost is non-negative. If cost is >= 0

```

```

23  * it invokes digraph_baseclass' makeEdge method, else it
24  * sets isNew to false and returns a NULL pointer.
25  *
26  * See digraph_baseclass::makeEdge.
27  **/
28  virtual edge *makeEdge(node *newsourc, node *newsink, int newcost,
29  bool & isNew);
30  /**
31  * Make a new node by calling digraph_baseclass::makeNode and then marking
32  * any current search cache as empty.
33  * See digraph_baseclass::makeNode.
34  */
35  virtual node *makeNode(string newname, bool & isNew);
36  /**
37  * Return a path from a start node to a destination node according
38  * to the digraph_baseclass::getPath() specification, and cache optimal
39  * path information in this object until the next mutator call.
40  * See digraph_baseclass::getPath().
41  **/
42  virtual vector<edge *> getPath(node *start, node *end) const ;
43  /**
44  * Default 0-parameter constructor initializes an empty graph.
45  **/
46  digraph_dijkstra() {
47  laststart = 0 ;
48  }
49 private:
50  struct dist {          // distance info for a node
51  // Default assignment and copy construction are OK.
52  int d ;              // distance to start of search
53  bool inf ;          // set to true on infinity
54  digraph_baseclass::node *n ; // node with this distance from start
55  dist(digraph_baseclass::node *target) {
56  d = 0 ;            // place holder
57  inf = true ;      // initialize to infinity
58  n = target ;
59  }
60  // This constructor should only be needed for array allocation.
61  // It should be over-written by a real value made using the above
62  // constructor.
63  dist() {          // needed for array allocation in priorityq
64  d = -1 ;        // place holder
65  inf = true ;    // initialize to infinity
66  n = 0 ;
67  }
68  } ;

```

```

69  static int compare(const dist *const left, const dist *const right);
70  // PriorityQueue comparator
71  mutable node *laststart ; // node for the most recent Dijkstra search
72  mutable map<node *, dist> distance ; // Dijkstra's distance from laststart
73  mutable map<node *, edge *> previous ; // predecessor in back path
74  // Not implemented:
75  digraph_dijkstra(const digraph_dijkstra &other) ;
76  digraph_dijkstra& operator=(const digraph_dijkstra& other);
77  };
78
79  #endif

```

1 /* digraph_dijkstra.cxx -- Method implementations for

```

2  digraph_dijkstra.h.
3
4  CSC402, Fall, 2009, Dr. Dale Parson.
5  */
6
7  #include "digraph_dijkstra.h"
8  #include "PriorityQueue.cxx"
9  #include <list>
10 using namespace std ;
11
12 digraph_baseclass::node *
13 digraph_dijkstra::makeNode(string newname, bool & isNew) {
14  laststart = 0 ; // invalidate any prior dijkstra search
15  return digraph_baseclass::makeNode(newname, isNew);
16  }
17
18 digraph_baseclass::edge *
19 digraph_dijkstra::makeEdge(digraph_baseclass::node *newsourc,
20 digraph_baseclass::node *newsink, int newcost, bool & isNew) {
21  if (newcost < 0) {
22  isNew = false ;
23  return 0 ;
24  } else {
25  laststart = 0 ; // invalidate any prior dijkstra search
26  return digraph_baseclass::makeEdge(newsource, newsink,
27  newcost, isNew);
28  }
29  }
30
31 int
32 digraph_dijkstra::compare(const digraph_dijkstra::dist *const left,
33 const digraph_dijkstra::dist *const right) {
34  if (left->inf) {

```

```

35     if (right->inf) { // both infinity
36         return 0;
37     }
38     return -1; // right is smaller, left has smaller priority
39 } else if (right->inf) { // right is bigger, left has bigger priority
40     return 1;
41 } else if (left->d < right->d) {
42     return 1;
43 } else if (left->d > right->d) {
44     return -1;
45 }
46 return 0;
47 }
48
49 vector<digraph_baseclass::edge *>
50 digraph_dijkstra::getPath(node *start, node *end) const {
51     vector<digraph_baseclass::edge *> result;
52     if (start == 0) {
53         if (end == 0) {
54             return result;
55         }
56         start = &entrynode;
57     }
58     // Following is Parson's derivation of pseudocode at
59     // http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
60     if (start != laststart) { // USE DIJKSTRA'S ALGORITHM:
61         PriorityQueue<dist> priq(compare, nodeset.size());
62         map<digraph_baseclass::node *, PriorityQueue<dist>::qhandle> qhmap;
63         laststart = start;
64         distance.clear(); // all distances are infinity
65         previous.clear(); // all previous entries are undefined
66         for (set<digraph_baseclass::node *>::iterator niter = nodeset.begin()
67             ; niter != nodeset.end(); niter++) {
68             digraph_baseclass::node *n = *niter;
69             dist startdist(n);
70             if (n == start) {
71                 startdist.inf = false; // start is at distance 0
72                 startdist.d = 0;
73             }
74             distance[n] = startdist;
75             qhmap[n] = priq.enqueue(startdist); // save the handle
76         }
77         cerr << endl; // DEBUG
78         while (priq.size() > 0) { // queue is not empty
79             bool isvalid;
80             dist cheapest = priq.dequeue(isvalid);

```

```

81     cerr << "DEBUG CHEAPEST FROM HEAP " << cheapest.n->getName() << " "
82     << cheapest.inf << " " << cheapest.d << endl;
83     if (!isvalid) {
84         cerr << "DEBUG ME! Failed dequeue on non-empty PriorityQueue!"
85         << endl;
86     }
87     if (cheapest.inf) {
88         break;
89     }
90     // This is the next closest node. Check its neighbors' distances.
91     set<digraph_baseclass::edge *> edges = cheapest.n->getEdges();
92     set<digraph_baseclass::edge *>::iterator eiter
93     = edges.begin();
94     while (eiter != edges.end()) {
95         digraph_baseclass::edge *e = *eiter;
96         eiter++;
97         digraph_baseclass::node *sinkn = e->getSink();
98         int newdist = e->getCost() + cheapest.d;
99         dist olddist = distance[sinkn];
100        if (olddist.inf || newdist < olddist.d) {
101            olddist.inf = false;
102            olddist.d = newdist;
103            distance[sinkn] = olddist;
104            previous[sinkn] = e;
105            /****
106            cerr << "DEBUGGING previous " << sinkn->getName() << " "
107            << e->getCost() << " " << e->getSource()->getName()
108            << endl;
109            *****/
110            priq.move(qhmap[olddist.n], olddist);
111        }
112    }
113 }
114 }
115 // Dijkstra has run, test whether there is a back path from end to start.
116 // Use a list so we can construct it backwards.
117 digraph_baseclass::node *cur = end; // Did we reach end node?
118 map<digraph_baseclass::node *, digraph_baseclass::edge *>::iterator itcur
119 = previous.find(cur);
120 if (itcur == previous.end()) {
121     // This destination node not reached from start in the graph.
122     return result;
123 }
124 list<digraph_baseclass::edge *> lresult;
125 while (cur != start) { // We should have a path back to start.
126     if (itcur == previous.end()) {

```

```

127     // A broken chain to start is a bug.
128     cerr << "BUG backlink from " << cur->getName() << " is NULL"
129     << endl ;
130     cerr.flush();
131     result.clear();
132     return result ;
133 }
134 digraph_baseclass::edge *backlink = itcur->second ;
135 lresult.push_front(backlink);
136 cur = backlink->getSource();
137 itcur = previous.find(cur);
138 }
139 list<digraph_baseclass::edge *>::iterator lriter = lresult.begin();
140 while (lriter != lresult.end()) {
141     result.push_back(*lriter);
142     lriter++ ;
143 }
144 return result ;
145 }

```

http://en.wikipedia.org/wiki/Bellman-Ford_algorithm

Bellman–Ford is in its basic structure very similar to [Dijkstra's algorithm](#), but instead of greedily selecting the minimum-weight node not yet processed to relax, it simply relaxes *all* the edges, and does this $|V| - 1$ times, where $|V|$ is the number of vertices in the graph. The repetitions allow minimum distances to accurately propagate throughout the graph, since, in the absence of negative cycles, the shortest path can only visit each node at most once. Unlike the greedy approach, which depends on certain structural assumptions derived from positive weights, this straightforward approach extends to the general case.

Bellman–Ford runs in $O(|V| \cdot |E|)$ time, where $|V|$ and $|E|$ are the number of vertices and edges respectively.

procedure BellmanFord(*list* vertices, *list* edges, *vertex* source)

// This implementation takes in a graph, represented as lists of vertices

// and edges, and modifies the vertices so that their distance and

// predecessor attributes store the shortest paths.

// Step 1: Initialize graph

for each vertex v in vertices:

if v is source **then** v .distance := 0

else v .distance := **infinity**

v .predecessor := **null**

// Step 2: relax edges repeatedly

for i **from** 1 **to** size(vertices)-1:

for each edge uv in edges: *// uv is the edge from u to v*

u := uv .source

v := uv .destination

```

        if  $u$ .distance +  $uv$ .weight <  $v$ .distance:
             $v$ .distance :=  $u$ .distance +  $uv$ .weight
             $v$ .predecessor :=  $u$ 
// Step 3: check for negative-weight cycles
for each edge  $uv$  in edges:
     $u$  :=  $uv$ .source
     $v$  :=  $uv$ .destination
    if  $u$ .distance +  $uv$ .weight <  $v$ .distance:
        error "Graph contains a negative-weight cycle"

```