

# Data Structures and Algorithms – Week 3 Code Samples

Lecturer: Arran Stewart

This zip file contains code samples we will refer to during week 3.

To understand the code samples well, you should make use of the textbook, which is *Data Structures and Problem Solving Using Java* (4th edition) by Mark Allen Weiss.

# Graph – adjacency matrix representation

```
1  import java.util.HashSet;
2  /**
3   *Graph.java is a class to assist with graph and network algorithms for CITS2200.
4   *Graphs may be directed or not, weighted or not.
5   *Graphs are simple: no loop edges (v,v) and no multiple edges between nodes.
6   *Vertices are labeled from 0 to number of vertices - 1.
7   *Edge weights must be positive integers (of value 1 if the graph is not weighted).
8   *Utility methods are provided to create random graphs.
9   *This implementation uses an adjacency matrix to represent the graph.
10  *author Tim French modified Rachel Cardell-Oliver
11  */
12
13  public class AdjacencyMatrixGraph implements GraphADT{
14
15      private int[][] edgeMatrix;
16      private int numberOfVertices;
17      private boolean directed;
18      private boolean weighted;
19
20      /**
21       *Constructs an graph with n vertices and no edges
22       *vertices are labelled 0 to n-1
23       *@param vertices the number of vertices (must be greater than 0)
24       *@param weighted true if the graph is to be weighted
25       *@param directed true if the graph is directed
26       */
27      public AdjacencyMatrixGraph(int vertices, boolean weighted, boolean directed){
28          if(vertices<1) numberOfVertices = 0;
29          else numberOfVertices = vertices;
30          edgeMatrix = new int[numberOfVertices][numberOfVertices];
31          this.weighted = weighted;
32          this.directed = directed;
33      }
34
35      /**
36       *Constructs an unweighted, undirected graph with n vertices and no edges
37       *vertices are labelled 0 to vertices-1
38       *@param vertices the number of vertices (must be greater than 0)
39       */
40      public AdjacencyMatrixGraph(int vertices){
41          this(vertices, false, false);
42      }
43
44
45      /**
46       *Adds an edge of the given weight to the graph.
47       *If the graph is undirected the reverse edge is also added.
48       *If the graph is unweighted the weight is set to one.
49       *If an edge already exists it is overwritten with the new weight.
50       *Only simple graphs allowed (no loop edges v,v)
51       *@param u the start of the edge
52       *@param v the end of the edge
53       *@param weight the weight of the edge
54       */
55      public void addEdge(int u, int v, int weight){
56          if (0 <= u && 0 <= v && u < numberOfVertices && v < numberOfVertices && u!=v) {
57              if(weight<1) weight = 0;
58              else if(!weighted) weight = 1;
59              edgeMatrix[u][v] = weight;
```

```

60         if(!directed) edgeMatrix[v][u] = weight;
61     }
62     else throw new RuntimeException("Vertex out of bounds");
63 }
64
65 /**
66  *Adds an edge of weight 1 to the graph.
67  *If the graph is undirected the reverse edge is also added.
68  *@param u the start of the edge
69  *@param v the end of the edge
70  */
71 public void addEdge(int u, int v){
72     this.addEdge(u,v,1);
73 }
74
75 /**
76  *Removes the edge from the graph (sets the edge weight to 0).
77  *If the graph is undirected the reverse edge is also removed.
78  *@param u the start of the edge
79  *@param v the end of the edge
80  */
81 public void removeEdge(int u, int v){
82     this.addEdge(u,v,0);
83 }
84
85 /**
86  * vertex IDs are 0 to numberOfVertices-1
87  *@return the weight of an edge.
88  *@param u the start of the edge
89  *@param v the end of the edge
90  */
91 public int getEdgeWeight(int u, int v){
92     if (0 <= u && 0 <= v && u < numberOfVertices && v < numberOfVertices)
93         return edgeMatrix[u][v];
94     else throw new RuntimeException("Vertex out of bounds");
95 }
96
97 /**
98  *@return true if u and v are adjacent
99  *@param u the start of the edge
100  *@param v the end of the edge
101  */
102 public boolean isAdjacent(int u, int v){
103     return (edgeMatrix[u][v] != 0);
104 }
105
106 /**
107  *@return an array list of the vertex ids of the neighbours of v
108  *@param v the vertice
109  */
110 public HashSet<Integer> getNeighbours(int v) {
111     int[] neighbours = (int[])edgeMatrix[v].clone();
112     HashSet<Integer> nlist = new HashSet<Integer>();
113
114     for (int i=0; i < numberOfVertices; i++) {
115         if (neighbours[i]>0) { nlist.add(i); }
116     }
117     return (nlist);
118 }
119
120 /**
121  *@return the number of vertices

```

```

122     */
123 public int getNumberOfVertices(){
124     return numberOfVertices;
125 }
126
127 /**
128  *@return true if the graph is directed
129  */
130 public boolean isDirected(){
131     return directed;
132 }
133
134 /**
135  *@return true if the graph is weighted
136  */
137 public boolean isWeighted(){
138     return weighted;
139 }
140
141 /**
142  *Count distinct edges in a directed or undirected graph
143  *Note this operation runs in time  $n^2$ .
144  *@return int number of non-zero entries in the edge matrix
145  */
146 public int getNumberOfEdges(){
147     int count = 0;
148     if (directed) {
149         for(int i = 0; i<edgeMatrix.length; i++)
150             for (int j=0; j<edgeMatrix[i].length; j++)
151                 if (edgeMatrix[i][j]>0) { count++; }
152     } else { //undirected graph
153         for(int i = 0; i<edgeMatrix.length; i++)
154             for (int j=i+1; j<edgeMatrix[i].length; j++)
155                 if (edgeMatrix[i][j]>0) { count++; }
156     }
157     return count;
158 }
159
160 /**
161  *This method should not be used to write graph algorithms
162  * use the graphADT operations instead
163  *Note this operation runs in time  $n^2$ .
164  *To find a single edge use getEdgeWeight.
165  *@return a clone of the edgeMatrix
166  */
167 private int[][] getEdgeMatrix(){
168     int[][] clone = new int[edgeMatrix.length][edgeMatrix.length];
169     for(int i = 0; i<edgeMatrix.length; i++)
170         clone[i] = (int[])edgeMatrix[i].clone();
171     return clone;
172 }
173
174 /**
175  *Utility method to create a random graph
176  *the graph is simple (no loop edges v,v)
177  *@return a random graph of the specified density
178  *@param numberOfVertices the number of vertices
179  *@param maxWeight the maximum edgeWeight for the graph.
180  *@param density between 0 and 1, the probability of an edge existing.
181  */
182 public static AdjacencyMatrixGraph randomGraph(int numberOfVertices, boolean weighted)
183     AdjacencyMatrixGraph g = new AdjacencyMatrixGraph(numberOfVertices, weighted, di

```

```

184         for(int i = 0; i<g.getNumberOfVertices(); i++){
185             for(int j = 0; j<(directed?g.getNumberOfVertices():i); j++){
186                 if(Math.random()<density && i!=j){
187                     if (!weighted) {
188                         g.edgeMatrix[i][j] = 1;
189                     } else {
190                         g.edgeMatrix[i][j] = (int)(Math.random()*(maxWeight-1))+1;
191                     }
192                     if(!directed) g.edgeMatrix[j][i] = 1;
193                 }
194             }
195         }
196         return g;
197     }
198
199     /**
200     *Creates a random, unweighted, undirected graph
201     *@return a random graph of the specified density
202     *@param numberOfVertices the nukmber of vertices
203     *@param directed true if the graph is to be directed
204     *@param density between 0 and 1, the probablity of an edge existing.
205     */
206     public static AdjacencyMatrixGraph randomGraph(int numberOfVertices, double density)
207     {
208         return randomGraph(numberOfVertices, false, false, density, 1);
209     }
210
211     /**
212     *Creates a random weighted, directed graph.
213     * The weights will be even distributed between 1 and
214     * the maximum edge weight (inclusive).
215     *@return a random graph of the specified density
216     *@param numberOfVertices the number of vertices
217     *@param directed true if the graph is to be directed
218     *@param density between 0 and 1, the probability of an edge existing.
219     *@param maxWeight the maximum edgeWeight for the Graph.
220     */
221     public static AdjacencyMatrixGraph randomGraph(int numberOfVertices, double density,
222     {
223         return randomGraph(numberOfVertices, true, true, density, maxWeight);
224     }
225
226     /**
227     *This method produces a representation of the
228     *graph that corresponds to the adjacency
229     *matrix used by the readFile method.
230     *@return a String representation of the graph,
231     */
232     public String toString(){
233         StringBuffer s = new StringBuffer(getNumberOfVertices()+"\n");
234         for(int i = 0; i<numberOfVertices; i++){
235             for(int j = 0; j<numberOfVertices; j++){
236                 s.append(edgeMatrix[i][j]);
237                 s.append("\t");
238             }
239             s.append("\n");
240         }
241         return s.toString();
242     }
243 }

```

# ArrayIntegerDeque

Implementation of a Deque using an array of integers.

```
1  import CITS2200.Overflow;
2  import CITS2200.Underflow;
3
4  /**
5   *A Class for basic operations of a double ended queue (DEQUE).
6   *Implements the local DequeADT
7   *@author Tim French
8   */
9
10 public class ArrayIntegerDeque implements DequeADT<Integer>{
11
12     int[] items;
13     int left;
14     int size;
15
16     public ArrayIntegerDeque(int maxSize){
17         items = new int[maxSize];
18         left = 0;
19         size = 0;
20     }
21
22
23     /**
24     *Adds an element to the right end of the queue.
25     *@param element the element to be added
26     *@throws overflow if the queue is full
27     */
28     public void pushLeft(Integer element) throws Overflow{
29         if(isFull()) throw new Overflow("Full!");
30         left = (left+items.length-1)%items.length;
31         items[left]=element;
32         size++;
33     }
34
35     /**
36     *Adds an element to the left end of the queue.
37     *@param element the element to be added
38     *@throws overflow if the queue is full
39     */
40     public void pushRight(Integer element) throws Overflow{
41         if(isFull()) throw new Overflow("Full!");
42         items[(left+size++)%items.length]=element;
43     }
44
45
46     /**
47     *Removes and returns the element on the left end of the queue.
48     *@return the leftmost element
49     *@throws Underflow if the queue is empty
50     */
51     public Integer popLeft() throws Underflow{
52         if(isEmpty()) throw new Underflow("Empty!");
53         int o = items[left++];
54         left = left%items.length;
55         size--;
56         return o;
57     }
```

```

58
59  /**
60  *Removes and returns the element on the right end of the queue.
61  *@return the rightmost element
62  *@throws Underflow if the queue is empty
63  */
64  public Integer popRight() throws Underflow{
65      if(isEmpty()) throw new Underflow("Empty!");
66      size--;
67      return items[(left+ size)%items.length];
68  }
69
70  /**
71  *Returns the element on the right end of the queue.
72  *@return the rightmost element
73  *@throws Underflow if the queue is empty
74  */
75  public Integer peekRight() throws Underflow{
76      if(isEmpty()) throw new Underflow("Empty!");
77      return items[(left+size-1)%items.length];
78  }
79
80  /**
81  *Returns the element on the left end of the queue.
82  *@return the leftmost element
83  *@throws Underflow if the queue is empty
84  */
85  public Integer peekLeft() throws Underflow{
86      if(isEmpty()) throw new Underflow("Empty!");
87      return items[left];
88  }
89
90  /**
91  *@return true is the queue is empty
92  */
93  public boolean isEmpty(){return size==0;}
94
95  /**
96  *@return true if the queue is full
97  */
98  public boolean isFull() {
99      return (size == items.length);
100  }
101
102  }

```

# Breadth-first search

```
1  /**
2   * perform a Breadth First Search over a directed graph
3   * @author Created by tim on 14/03/06, modified rachelcardell-oliver 2015
4   *
5   */
6
7
8  public class BFS {
9
10 /**
11  *Runs a BFS on a given directed, unweighted graph.
12  *@return an array listing the parent of each vertex
13  *in the spanning tree, or -1 is the vertex is not reachable from the start vertex.
14  *@param g the Graph to be searched
15  *@param startVertex the vertex on which to start the search
16  */
17  public int[] getConnectedTree(AdjacencyMatrixGraph g, int startVertex){
18      if (startVertex >= g.getNumberOfVertices()) {
19          throw new RuntimeException("Vertex out of bounds");
20      }
21      int[] queue = new int[g.getNumberOfVertices()];
22      int head = 0;
23      int tail = 0;
24      queue[tail++] = startVertex;
25      int currentVertex;
26      int[] visited = new int[g.getNumberOfVertices()]; //white = 0, grey = 1, black = 2
27      int[] spanningTree = new int[g.getNumberOfVertices()];
28      for(int i = 0; i<spanningTree.length; i++)spanningTree[i] = -1;
29      while(tail>head){
30          currentVertex = queue[head++];
31          visited[currentVertex] = 2;
32          for(int i = 0; i< g.getNumberOfVertices(); i++){
33              if (g.isAdjacent(currentVertex,i) && (visited[i] == 0)){
34                  visited[i] = 1;
35                  spanningTree[i] = currentVertex;
36                  queue[tail++] = i;
37              }
38          }
39      }
40      return spanningTree;
41  }
42
43 /**
44  *Runs a BFS on a given directed, unweighted graph to find the distances of vertices from
45  *@return an array listing the distance of each vertex from the start vertex of each, or
46  * -1 is the vertex is not reachable from the start vertex.
47  *@param g the Graph to be searched
48  *@param startVertex the vertex on which to start the search
49  */
50  public int[] getDistances(AdjacencyMatrixGraph g, int startVertex){
51      if(startVertex >= g.getNumberOfVertices()) {
52          throw new RuntimeException("Vertex out of bounds");
53      }
54      int[] queue = new int[g.getNumberOfVertices()];
55      int head = 0;
56      int tail = 0;
57      queue[tail++] = startVertex;
58      int currentVertex;
59      int[] visited = new int[g.getNumberOfVertices()]; //white = 0, grey = 1, black = 2
```



```

60     int[] distances = new int[g.getNumberOfVertices()];
61     for(int i = 0;i<distances.length;i++) distances[i] = -1;
62     distances[startVertex] = 0;
63     while(tail>head){
64         currentVertex = queue[head++];
65         visited[currentVertex] = 2;
66         for(int i = 0; i< g.getNumberOfVertices(); i++){
67             if(g.isAdjacent(currentVertex,i) && (visited[i] == 0)){
68                 visited[i] = 1;
69                 distances[i] = distances[currentVertex]+1;
70                 queue[tail++] = i;
71             }
72         }
73     }
74     return distances;
75 }
76
77 }

```

# Graph ADT

Interface for a Graph ADT.

```
1  import java.util.HashSet;
2
3  /**
4   * some operations to be provided by a graph implementation
5   * @author rachelcardell-oliver
6   *
7   */
8  public interface GraphADT {
9
10     /**
11      * @return the number of vertices in a graph
12      * vertices are labelled 0 to n-1
13      */
14     public int getNumberOfVertices();
15
16     /**
17      * Adds an edge of the given weight to the graph.
18      * If the graph is undirected the reverse edge is also added.
19      * If the graph is unweighted the weight is set to one.
20      * If an edge already exists it is overwritten with the new weight.
21      * Only simple graphs allowed (no loop edges v,v)
22      * @param u the start of the edge
23      * @param v the end of the edge
24      * @param weight the weight of the edge
25      */
26     public void addEdge(int u, int v, int weight);
27
28     /**
29      * get the weight of given edge of the graph
30      * @param u vertex (start)
31      * @param v vertex (end)
32      * @return weight >= 1 for weighted graph,
33      * @return 1 for an existing edge in an unweighted graph
34      * @return 0 if no edge exists
35      */
36     public int getEdgeWeight(int u, int v);
37
38     /**
39      * check whether u,v is an edge in the graph
40      * @param u vertex (start)
41      * @param v vertex (end)
42      * @return boolean true if u,v is an edge, false otherwise
43      */
44     public boolean isAdjacent(int u, int v);
45
46     /**
47      * get all the neighbours of a given vertex
48      * @param v vertex
49      * @return set of vertex IDs for the neighbours of v
50      */
51     public HashSet<Integer> getNeighbours(int v);
52
53
54
55 }
```

# Prim's algorithm

```
1  import java.util.PriorityQueue;
2  import java.util.HashSet;
3  import java.util.ArrayDeque;
4
5
6  public class PrimMST {
7
8      static final int INFTY = Integer.MAX_VALUE;
9      static final int NO_EDGE = -1;
10
11      /**
12       *Runs Prim algorithm on a given undirected, weighted graph.
13       *This version includes debug statements lines 58-62 for demo purposes.
14       *Comment these lines when not required.
15       *@return the weight of the minimum spanning tree or -1 if no such tree exists.
16       *@param g the Graph to be searched. Assume that an edge weight of -1 signifies that
17       **/
18      public static int[] getMinSpanningTree(AdjacencyMatrixGraph g) {
19
20          // Path represents an entry in the priority queue for Prim's algorithm.
21          class Path implements Comparable<Path>
22          {
23              public int      vertex;    //vertex in consideration
24              public int      cost;      //current cost for this vertex
25
26              public Path( int v, int c) {
27                  vertex = v;
28                  cost = c;
29              }
30
31              public int compareTo( Path rhs ) {
32                  double otherCost = rhs.cost;
33                  return cost < otherCost ? -1 : cost > otherCost ? 1 : 0;
34              }
35          }
36
37          //Prim's algorithm
38          int n = g.getNumberOfVertices();
39          //state variables for Prim
40          int[] visited = new int[n]; //white = 0, grey = 1, black = 2
41          int[] key = new int[n]; //edge costs
42          int[] p = new int[n]; //pi list of connected vertex
43
44          for (int i=1; i<n; i++) {
45              visited[i] = 0;
46              key[i] = INFTY;
47              p[i] = NO_EDGE;
48          }
49
50          PriorityQueue<Path> pri = new PriorityQueue<>(n); //create a priority queue with
51          int s = 0; //start vertex
52          visited[s] = 1; //grey
53          key[s] = 0;
54          p[s]=NO_EDGE;
55          pri.add(new Path(s,key[s]));
56
57          while(!pri.isEmpty()){
58              showArray("visited = ",visited);
59              showArray("key = ",key);
```

```

60         showArray("p = ",p);
61         showQueue("q = ",pri);
62         System.out.println();
63         int u = pri.remove().vertex;
64         HashSet<Integer> nn = g.getNeighbours(u);
65         for (Integer v : nn) {
66             if (visited[v]==0) {
67                 visited[v] = 1;
68                 key[v] = g.getEdgeWeight(u,v);
69                 p[v]=u;
70                 pri.add(new Path(v,key[v]));
71             } else if (visited[v]==1) {
72                 if (key[v] > g.getEdgeWeight(u,v)) {
73                     key[v] = g.getEdgeWeight(u,v);
74                     p[v] = u;
75                 }
76             }
77         }
78         visited[u] = 2; //black: u is done //is v in pseudo code?
79     }
80     return(p); //the connected edge for each v
81 }
82
83 public static void showArray(String msg, int[] arr) {
84     System.out.print(msg);
85     for (int i=0; i<arr.length; i++) {
86         if (arr[i]==INFTY) {
87             System.out.print("infty, ");
88         } else {
89             System.out.print(arr[i]+" ", );
90         }
91     }
92     System.out.println();
93 }
94
95 public static void showQueue(String msg, PriorityQueue q) {
96     System.out.print(msg);
97     //TODO
98     System.out.println();
99 }
100
101 /**
102  * demonstration of Prim's algorithm in action
103  * @param args
104  */
105 public static void main (String[] args) {
106     AdjacencyMatrixGraph g1 = new AdjacencyMatrixGraph(8,true,false); //weighted und
107     g1.addEdge(0,1,3);
108     g1.addEdge(1,2,1);
109     g1.addEdge(2,3,2);
110     g1.addEdge(2,5,4);
111     g1.addEdge(3,6,1);
112     g1.addEdge(3,5,3);
113     g1.addEdge(4,5,2);
114     g1.addEdge(5,6,2);
115     g1.addEdge(6,7,1);
116     g1.addEdge(7,5,6);
117
118     System.out.println(g1.toString()); //show the graph
119
120     int[] p = getMinSpanningTree(g1); //run Prim MST
121

```

```

122         //show the results
123         System.out.println("Prim MST result:");
124         for (int v=0; v<g1.getNumberOfVertices(); v++) {
125             if (p[v] != NO_EDGE) {
126                 System.out.println("edge="+ v+ " "+p[v]+" weight=" + g1.getEdgeWeight(
127             }
128         }
129     }
130 }
131 }

```