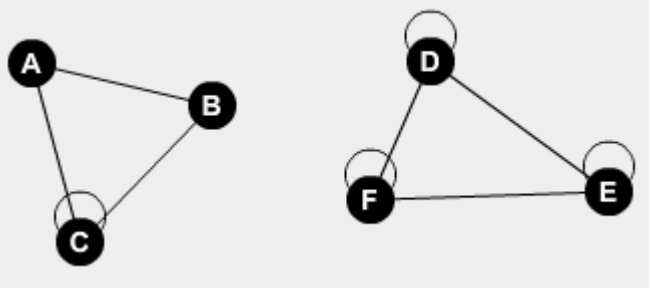
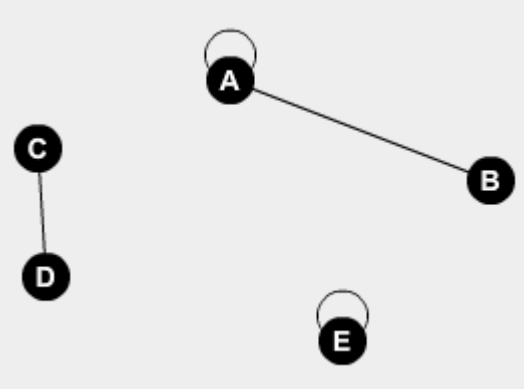
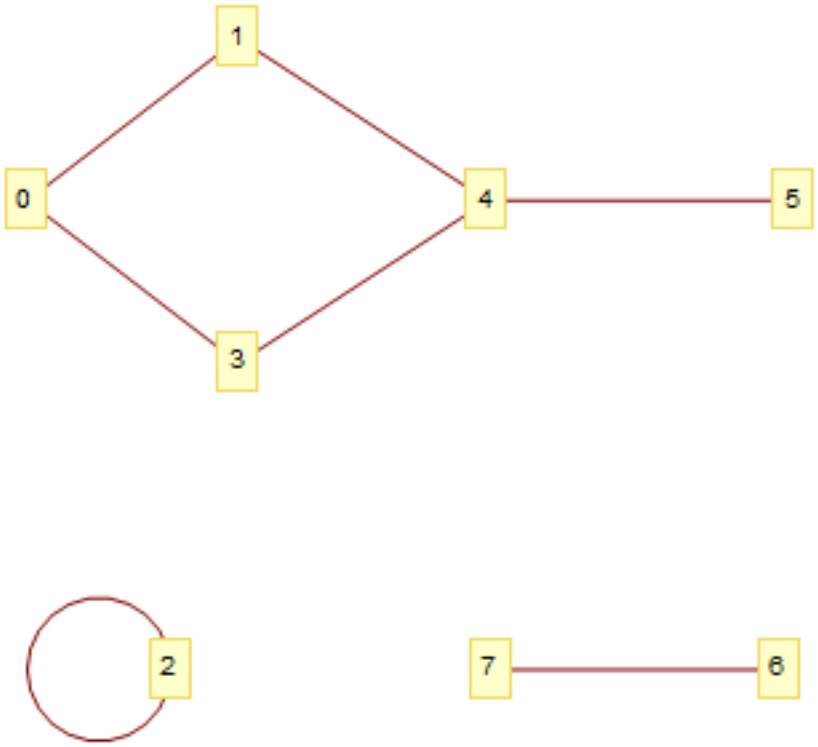
Graphs







DepthFirstScan

$$0, 1, 4, 5, 3$$

```
DepthFirstScan[
```

```
Graph[{0 -> 1, 0 -> 3, 1 -> 4, 2 -> 2, 3 -> 4, 4 -> 5,
6 -> 7}], 0, {"PrevisitVertex" -> (Print["Visiting", #] &)}]
```

BreadthFirstScan

$$0, 1, 3, 4, 5$$

```
BreadthFirstScan[
Graph[{0 -> 1, 0 -> 3, 1 -> 4, 2 -> 2, 3 -> 4, 4 -> 5,
6 -> 7}], 0, {"PrevisitVertex" -> (Print["Visiting", #] &)}]
```

Choice of algorithm

- Here, we could use both the matrix form or list form. Both will do. That is because we have twice as many vertices as edges, so approx. one path between each edge. For memory, my gut feeling suggests the list form as the matrix one has many zeros (and we have few vertices).
- Here, as we deal with many vertices(in the order of a 10^4), there are going to be many more paths, so better to use matrix form. Correction: List is still better. A matrix would have $1000 \times 1000 = 10^6$ items, and removing the edges would still leave 950000 zeros.
- List is better because we can easily look up a certain combination.

Theta n^2
On one hand, this is strange since all neighbours are in the same row, so it should be fast. But, the problem is actually in the recursion. Each time, we are going to look up neighbours' neighbours, which requires us to jump to different rows and go through them. We are going to load the same row more than once.

HashGraph

```
package kth.csc.inda;

import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Map;
import java.util.NoSuchElementException;
import java.util.Set;

/**
 * A graph with a fixed number of vertices implemented using adjacency
    maps.
 * Space complexity is &Theta;(n + m) where n is the number of vertices
    and m
 * the number of edges.
 *
 * @author [Name]
 * @version [Date]
 */
public class HashGraph implements Graph {
    /**
     * The map edges[v] contains the key-value pair (w, c) if there is an
        edge
     * from v to w; c is the cost assigned to this edge. The maps may be
        null
     * and are allocated only when needed.
     */
```

```
24      private final Map<Integer, Integer>[] edges;
25      private final static int INITIAL_MAP_SIZE = 4;
26
27      /** Number of edges in the graph. */
28      private int numEdges;
29
30      /**
31       * Constructs a HashGraph with n vertices and no edges. Time
             complexity:
32       * O(n)
33       *
34       * @throws IllegalArgumentException
35       *                 if n < 0
36       */
37      public HashGraph(int n) {
38          if (n < 0)
39              throw new IllegalArgumentException("n = " + n);
40
41          // The array will contain only Map<Integer, Integer> instances
                created
42          // in addEdge(). This is sufficient to ensure type safety.
43          @SuppressWarnings("unchecked")
44          Map<Integer, Integer>[] a = new HashMap[n];
45          edges = a;
46      }
47
48      /**
49       * A method that checks that v may exist in the array.
50       * @remark It does not check whether v has a value associated with it.
51       */
52      private void vBound(int v)
53      {
54          if(v >= edges.length || v < 0)
55          {
56              throw new IllegalArgumentException();
57          }
58      }
59      /**
60       * Add an edge without checking parameters.
61       */
62      private void addEdge(int from, int to, int cost) {
63          if (edges[from] == null)
64              edges[from] = new HashMap<Integer, Integer>(INITIAL_MAP_SIZE);
65          if (edges[from].put(to, cost) == null)
66              numEdges++;
67      }
68
69      /**
70       * {@inheritDoc Graph} Time complexity: O(1).
71       */
72      @Override
73      public int numVertices() {
74          return edges.length;
75      }
76
77      /**
```

```
78          * {@inheritDoc Graph} Time complexity: O(1).
79          */
80         @Override
81         public int numEdges() {
82             return numEdges;
83         }
84 
85         /**
86          * {@inheritDoc Graph}
87          */
88         @Override
89         public int degree(int v) throws IllegalArgumentException {
90             // TODO
91 
92             //how can v be out of bound?
93             vBound(v);
94 
95             Map<Integer,Integer> obj = edges[v];
96 
97             if(obj != null)
98             {
99                 return obj.size();
100            }
101            else
102            {
103                return 0;
104            }
105        }
106 
107        /**
108         * {@inheritDoc Graph}
109         */
110        @Override
111        public VertexIterator neighbors(final int v) {
112            // TODO
113            vBound(v);
114 
115            if (edges[v] == null || (edges[v]!=null && edges[v].keySet() ==
                   null))
116            {
117                return new VertexIterator() {
118 
119                    @Override
120                    public int next() throws NoSuchElementException {
121                        // TODO Auto-generated method stub
122                        throw new NoSuchElementException();
123                    }
124 
125                    @Override
126                    public boolean hasNext() {
127                        // TODO Auto-generated method stub
128                        return false;
129                    }
130                };
131            }
132            else
```

```
133         {
134             VertexIterator vi = new VertexIterator() {
135 
136             Iterator<Integer> it = edges[v].keySet().iterator();
137             @Override
138             public int next() throws NoSuchElementException {
139                 return it.next();
140             }
141 
142             @Override
143             public boolean hasNext() {
144                 return it.hasNext();
145             }
146         };
147 
148         return vi;
149         }
150 
151     }
152 
153     /**
154      * {@inheritDoc Graph}
155      */
156     @Override
157     public boolean hasEdge(int v, int w) {
158         // TODO
159         vBound(v);
160         vBound(w);
161 
162         Map<Integer, Integer> obj = edges[v];
163 
164         if(obj == null)
165             return false;
166         else
167         {
168             if(obj.containsKey(w))
169                 return true;
170             else
171                 return false;
172         }
173     }
174 
175     /**
176      * {@inheritDoc Graph}
177      */
178     @Override
179     public int cost(int v, int w) throws IllegalArgumentException {
180 
181         vBound(v);
182 
183         Map<Integer,Integer> obj = edges[v];
184 
185         int cost = NO_COST;
186 
187         if(obj != null && obj.containsKey(w))
188             cost=obj.get(w);
```

```
189
190         return cost;
191     }
192
193     /**
194      * {@inheritDoc Graph}
195      */
196     @Override
197     public void add(int from, int to) {
198         // TODO
199         this.add(from, to, NO_COST);
200     }
201
202     /**
203      * {@inheritDoc Graph}
204      */
205     @Override
206     public void add(int from, int to, int c) {
207         //kosten Ãmste uppdateras
208         // 0 ,0 en kant.
209         vBound(to);
210         vBound(from);
211
212         Map<Integer, Integer> obj = edges[from];
213         if(obj == null)
214         {
215             obj = new HashMap<Integer, Integer>();
216             obj.put(to, c);
217             numEdges++;
218         }
219         else if(obj.containsKey(to))
220         {
221
222             obj.put(to, obj.get(to) + c==NO_COST ? 0:c); // updating cost.
223             /*if(!hasEdge(to, from))
224             {
225                 numEdges++;
226             }*/
227         }
228         else
229         {
230             obj.put(to, c);
231             numEdges++;
232         }
233
234         edges[from] = obj;
235
236         //numEdges++;
237
238
239     }
240
241     /**
242      * {@inheritDoc Graph}
243      */
```

```
244     @Override
245     public void addBi(int v, int w) {
246         // TODO
247 
248         this.addBi(v, w, NO_COST);
249     }
250 
251     /**
252      * {@inheritDoc Graph}
253      */
254     @Override
255     public void addBi(int v, int w, int c) {
256         // TODO
257         // code duplication. reuse add method.
258         vBound(v);
259         vBound(w);
260 
261         this.add(v,w,c);
262         this.add(w,v,c);
263 
264         if(v==w)
265         {
266             //numEdges--;
267         }
268     }
269 
270     /**
271      * {@inheritDoc Graph}
272      */
273     @Override
274     public void remove(int from, int to) {
275         // TODO
276 
277         vBound(from);
278         vBound(to);
279 
280         Map<Integer, Integer> obj = edges[from];
281 
282         if(obj != null && obj.containsKey(to))
283         {
284             obj.remove(to);
285             numEdges--;
286 
287             if(hasEdge(to, from) && to==from)
288             {
289                 numEdges--;
290             }
291         }
292 
293     }
294 
295     /**
296      * {@inheritDoc Graph}
297      */
298     @Override
299     public void removeBi(int v, int w) {
```

```
300         // TODO
301
302         this.remove(v, w);
303         this.remove(w, v);
304
305
306
307     }
308
309     /**
310      * Returns a string representation of this graph.
311      *
312      * @return a String representation of this graph
313      */
314     @Override
315     public String toString() {
316         // TODO
317
318         if(edges == null  || (edges != null && numEdges==0))
319         {
320             return "{}";
321         }
322
323         StringBuilder sb = new StringBuilder();
324
325         sb.append("{");
326
327         for(int i = 0; i< edges.length; i++)
328         {
329
330
331             //VertexIterator vi = neighbors(i);
332             if(edges[i] != null && edges[i].keySet().size() > 0 )
333             {
334
335                 if(i>1 && edges[i-1]!=null)
336                 {
337                     sb.append(", ");
338                 }
339                 Set<Integer> set = edges[i].keySet();
340                 for(int key : set)
341                 {
342                     if(edges[i].get(key) != NO_COST)
343                     {
344                         sb.append("(" + i + ","+ key + "," +
                                edges[i].get(key) + ")");
345                     }
346                     else
347                     {
348                         sb.append("(" + i + ","+ key+")");
349                     }
350
351
352
353                 }
354
```

```
355                 }
356             }
357 
358             sb.append("}");
359 
360             return sb.toString();
361         }
362 }
```

Random Graphs

Different trials, n=1000..5000

Table 1: Results

HashGraph	MatrixGraph	HashGraph-MatrixGraph
705292	1670450	-965158
676144	966390	-290246
318982	938473	-619491
220044	1530869	-1310825
247960	1541543	-1293583

Result when n=1000

```
The graph consists of 907 components.
The longest one is located at 940 with the size of 94 items.
```

Conclusion

To sum up, HashGraph seems to be faster on average than MatrixGraph. There appears to be an increasing difference in speed as graphs get bigger (HashGraph will be the fastest one).

Rand class

```
1  package kth.csc.inda;
2  
3  import java.util.HashMap;
4  import java.util.Random;
5  import java.util.Timer;
6  import java.util.function.Consumer;
7  
8  
9  public class Rand {
10 
11     /**
12      * @param args
13      */
14     public static void main(String[] args) {
15         // TODO Auto-generated method stub
16         //System.out.println("dawd");
17 
18         System.out.println(Analyse(1000, true, false));
19 
20         System.out.println(Analyse(1000, false, false));
21         System.out.println(Analyse(2000, false, false));
```

```
22          System.out.println(Analyse(3000, false, false));
23          System.out.println(Analyse(4000, false, false));
24          System.out.println(Analyse(5000, false, false));
25
26          System.out.println("Break");
27
28          System.out.println(Analyse(1000, false, true));
29
30          System.out.println(Analyse(1000, false, true));
31          System.out.println(Analyse(2000, false, true));
32          System.out.println(Analyse(3000, false, true));
33          System.out.println(Analyse(4000, false, true));
34          System.out.println(Analyse(5000, false, true));
35
36
37
38      }
39
40      /**
41       * Analyses a graph.
42       * @param n The size
43       * @param showInfo Show detailed information about the graph.
44       * @param implementation See RandomGraph for comments.
45       * @return Time it took to execute
46       */
47      public static long Analyse(int n, boolean showInfo, boolean
            implementation)
48      {
49          Stopwatch sw = new Stopwatch();
50          Graph rand = RandGraph(1000, implementation);
51
52          sw.start();
53          int[] returnargs = maxComponents(rand);
54          sw.stop();
55
56          if(showInfo)
57          {
58              System.out.println("The graph consists of " + returnargs[2] +
                    " components.");
59              System.out.println("The longest one is located at " +
                    returnargs[1] + " with the size of " + returnargs[0] + "
                    items.");
60          }
61
62          return sw.nanoseconds();
63
64      }
65
66      /**
67       * random Generates a "random" graph with random edges (in total, n
              vertices and n edges.)
68       * @param n
69       * @param typeOfImplementation 0=HashGraph, 1=MatrixGraph
70       * @return
71       */
72      public static Graph RandGraph(int n, boolean typeOfImplementation)
```

```
73      {
74          Random rn = new Random();
75
76          Graph gr;
77
78          if(typeOfImplementation)
79              gr = new MatrixGraph(n);
80          else
81              gr = new HashGraph(n);
82
83          int pointer = 0; // at the start
84          boolean consumed = false;
85
86          for(int i = 0 ; (i < n) && (gr.numEdges() < n); i++)
87          {
88              for(int j = 0; j < (rn.nextInt(n) % 10) && (gr.numEdges() <
                    n); j++)
89              {
90                  gr.add(i, rn.nextInt(100));
91              }
92          }
93
94          if (gr.numEdges() < n)
95          {
96              while( n-gr.numEdges() > 0) {
97                  gr.add(n-1, rn.nextInt(10));
98
99              }
100         }
101
102         return gr;
103
104     }
105
106     public static int temp = 0;
107     /**
108      * Uses DFS to find a) the number of components b) where it is located
             and c) the size of the greatest component.
109      * @param g
110      * @return
111      */
112     public static int[] maxComponents(Graph g) {
113
114         int maxNum = 0;
115         int pos = 0;
116         int components = 0;
117         //int temp = 0;
118         //final int[] t = new int[g.numVertices()];
119
120         VertexAction printVertex = new VertexAction() {
121             @Override
122             public void act(Graph g, int v) {
123                 //System.out.print(v + " ");
124                 //t[v] += 1;
125                 temp++;
126
```

```
127             }
128         };
129         int n = g.numVertices();
130         boolean[] visited = new boolean[n];
131         for (int v = 0; v < n; v++) {
132             if (!visited[v]) {
133                 dfs(g, v, visited, printVertex);
134 
135                 // now we start on a new component
136                 if(temp >= maxNum)
137                 {
138                     maxNum = temp; // store the number of items in the
                        component
139                     pos = v; // the position in the array
140 
141                     temp = 0; // this is crucial
142                 }
143 
144                 components++;
145                 //System.out.println();
146             }
147         }
148 
149         return new int[] {maxNum, pos, components};
150     }
151 
152     /**
153      * Traverses the nodes of g that have not yet been visited. The nodes
            are
154      * visited in depth-first order starting at v. The act() method in the
155      * VertexAction object is called once for each node.
156      *
157      * @param g
158      *            an undirected graph
159      * @param v
160      *            start vertex
161      * @param visited
162      *            visited[i] is true if node i has been visited
163      */
164     private static void dfs(Graph g, int v, boolean[] visited,
165             VertexAction action) {
166         if (visited[v])
167             return;
168         visited[v] = true;
169         action.act(g, v);
170         for (VertexIterator it = g.neighbors(v); it.hasNext();)
171             dfs(g, it.next(), visited, action);
172     }
173 
174 }
```

Stopwatch

```
1  package kth.csc.inda;
2 
3  /**
```

```
4  * A simple Stopwatch utility for measuring time in milliseconds.
5  *
6  * @author  Stefan Nilsson
7  * @version 2011-02-07
8  */
9  public class Stopwatch {
10     /**
11      * Time when start() was called. Contains a valid time
12      * only if the clock is running.
13      */
14     private long startTime;
15
16     /**
17      * Holds the total accumulated time since last reset.
18      * Does not include time since start() if clock is running.
19      */
20     private long totalTime = 0;
21
22     private boolean isRunning = false;
23
24     /**
25      * Constructs a new Stopwatch. The new clock is not
26      * running and the total time is set to 0.
27      */
28     public Stopwatch() {}
29
30     /**
31      * Turns this clock on.
32      * Has no effect if the clock is already running.
33      *
34      * @return a reference to this Stopwatch.
35      */
36     public Stopwatch start() {
37         if (!isRunning) {
38             isRunning = true;
39             startTime = System.nanoTime();
40         }
41         return this;
42     }
43
44     /**
45      * Turns this clock off.
46      * Has no effect if the clock is not running.
47      *
48      * @return a reference to this Stopwatch.
49      */
50     public Stopwatch stop() {
51         if (isRunning) {
52             totalTime += System.nanoTime() - startTime;
53             isRunning = false;
54         }
55         return this;
56     }
57
58     /**
59      * Resets this clock.
```

```
60          * The clock is stopped and the total time is set to 0.
61          *
62          * @return a reference to this Stopwatch.
63          */
64         public Stopwatch reset() {
65             isRunning = false;
66             totalTime = 0;
67             return this;
68         }
69
70         /**
71          * Returns the total time that this clock has been running since
72          * last reset.
73          * Does not affect the running status of the clock; if the clock
74          * is running when this method is called, it continues to run.
75          *
76          * @return the time in milliseconds.
77          */
78         public long milliseconds() {
79             return nanoseconds() / 1000000;
80         }
81
82         /**
83          * Returns the total time that this clock has been running since
84          * last reset.
85          * Does not affect the running status of the clock; if the clock
86          * is running when this method is called, it continues to run.
87          *
88          * @return the time in nanoseconds.
89          */
90         public long nanoseconds() {
91             return totalTime +
92                 (isRunning ? System.nanoTime() - startTime : 0);
93         }
94
95         /**
96          * Tests if this clock is running.
97          *
98          * @return <code>true</code> if this clock is running;
99          *         <code>false</code> otherwise.
100         */
101        public boolean isRunning() {
102            return isRunning;
103        }
104
105        /**
106         * Returns a string description of this clock. The exact details
107         * of the representation are unspecified and subject to change,
108         * but this is typical: "25 ms (running)".
109         */
110        @Override
111        public String toString() {
112            return milliseconds() + " ms" +
113                (isRunning() ? " (running)" : " (not running)");
114        }
115
```

```
116     /**
117      * Unit test. Run with <code>java -ea Stopwatch</code>.
118      */
119     public static void main(String[] args) throws InterruptedException {
120         Stopwatch c = new Stopwatch();
121         assert !c.isRunning();
122         assert c.milliseconds() == 0;
123         assert c.toString().equals("0 ms (not running)");
124 
125         c.stop();
126         assert !c.isRunning();
127         assert c.milliseconds() == 0;
128 
129         c.reset();
130         assert !c.isRunning();
131         assert c.milliseconds() == 0;
132 
133         c.start();
134         String s = c.toString();
135         assert s.equals("0 ms (running)") || s.equals("1 ms (running)");
136         assert c.isRunning();
137         c.stop();
138         assert !c.isRunning();
139 
140         c.start();
141         assert c.isRunning();
142         c.reset();
143         assert !c.isRunning();
144         assert c.milliseconds() == 0;
145 
146         c.start();
147         assert c.milliseconds() < 2;
148         assert c.isRunning();
149         Thread.sleep(2);
150         assert c.isRunning();
151         assert c.milliseconds() > 0;
152         assert c.milliseconds() < 4;
153         assert !c.toString().equals("0 ms (running)");
154         Thread.sleep(10);
155         assert c.isRunning();
156         assert c.milliseconds() > 10;
157         assert c.milliseconds() < 14;
158 
159         c.stop();
160         assert !c.isRunning();
161         assert c.milliseconds() > 10;
162         assert c.milliseconds() < 14;
163 
164         c.stop();
165         assert !c.isRunning();
166         assert c.milliseconds() > 10;
167         assert c.milliseconds() < 14;
168 
169         c.start();
170         assert c.isRunning();
171         assert c.milliseconds() > 10;
```

```
172             assert c.milliseconds() < 14;
173             Thread.sleep(10);
174             assert c.isRunning();
175             assert c.milliseconds() > 20;
176             assert c.milliseconds() < 24;
177 
178             c.reset();
179             assert !c.isRunning();
180             assert c.milliseconds() == 0;
181             assert c.toString().equals("0 ms (not running)");
182         }
183     }
```