Hemuppgift 7- Introduktion till Datalogi

Peter Boström

2009-03-19

Innehåll

1 Grafuppgifter

1.1 Rita grafer

Graferna finns bifogade längst bak.

Det går att konstruera en graf med 5 hörn, 4 kanter och 3 komponenter, förutsatt att kanten får gå från och till samma nod.

1.2 Graf med 8 hörn

Grafen finns bifogad längst bak.

1.2.1 Djupet först-sökning

DFS med start i hörnet 0:
0, 1, 4, 3, 5
Besöker 0.
Besöker 0:s första, 1
Besöker 1:s första obesökta: 4
Besöker 4:s första obesökta: 3
Besöker 3:s första obesökta: 5
Inga noder kvar med någon annan obesökt att besöka.

1.2.2 Bredden först-sökning

BFS med start i hörnet 0:
0, 1, 3, 4, 5
Besöker 0.
Besöker 0:s närmaste: 1, 3
Besöker 1:s närmaste som inte är besökta: 4
Besöker 3:s närmaste som inte är besökta: 5

1.3 Representera med närhetsmatris eller närhetslistor

1.3.1 1000 hörn, 2000 kanter, sparsam med minne

Närhetslistor, för att vara sparsam med minnet använder man närhetslistor så länge antalet kanter inte närmar sig antalet hörn i kvadrat. I detta fall hade det motsvarat 1 000 000 kanter. 2000 är väldigt mycket mindre, därför är listor att föredra.

1.3.2 1000 hörn, 50 000 kanter, sparsam med minne

Närhetslistor, då 50 000 fortfarande är väldigt mycket mindre än 1 000 000.

1.3.3 Avgöra om 2 hörn är grannar på konstant tid

Närhetsmatris, A[x][y] är alltid konstant.

1.4 Varför är DFS $\Theta(n^2)$ för en sammanhängande graf med n hörn som representeras med en närhetsmatris

DFS:en kommer att besöka ALLA noder. Från varje nod kommer den att leta igenom sina rader i tur och ordning efter en nod som inte är besökt, besöka den och dess undernoder, och så vidare, och sedan kommer den leta efter sin nästa obesökta nod. Dvs kommer den att gå igenom alla sina noder, för att se om de är grannar eller inte, och sedan kolla om de är besökta. I närhetsmatrisen kommer man behöva även gå igenom de noder som inte är grannar, eftersom man inte sparar de som är grannar, utan bara relationen mellan två stycken noder. Varje nod kommer på detta sätt behöva gå igenom alla sina noder, och finna att de antingen är grannar eller inte. n noder som i sin tur måste kolla mot n noder ger $Theta(n^2)$.

2 Programmering med grafer

Programmet slumpar fram grafer med samma antal kanter, slumpmässigt. Sedan djupet först-söker programmet tills alla hörn är besökta. Samtidigt samlar programmet upp information om antal kluster och hur många hörn som finns i största klustret.

2.1 Resultat

Storlek	Tid för DFS	Antal kluster	Största kluster
1000	20 ms	159 st	815 st
5000	36 ms	786 st	4034 st
10000	68 ms	1584 st	7968 st
20000	197 ms	3242 st	16027 st
40000	231 ms	6538 st	31744 st
80000	425 ms	12998 st	63673 st
100 000	1137 ms	16405 st	79474 st

2.2 Val av datastruktur

I det här fallet är valet av en närhetslista klart överlägset närhetsmatrisen. Eftersom antalet hörn är lika med antalet kanter så kommer varje nod vara kopplad till storleksordningen två kanter. Att gå igenom två kanter för att hitta vilka som gränsar till hörnet är klart överlägset att gå igenom lika många som antalet hörn, eftersom detta är storleksorningen upp till flera tusen gånger fler. Därför föreslås att problemet blir närmare linjärt än det är kvadratiskt, vilket (mer eller mindre) reflekteras i graferna.

A Källkod

GraphBuilder.java ListGraph.java ListGraphTest.java ListGraphVertexIterator.java

A.1 GraphBuilder.java

```
1  /**
2   *
3   */
4  package kth.csc.inda;
5
6  import java.util.Random;
7
8  /**
9   * @author lemming
10  * Denna äjkeln bygger grafer åt den som ähnder runt 0 grader.
11  */
12 public class GraphBuilder {
13     private static Random rand = new Random();
14
15     /**
16      * @param args ignored
17      */
18     public static void main(String[] args) {
19         final int n = 100000;
20         ListGraph lg = generateRandomGraph(n);
21
22         boolean [] visited = new boolean[n];
23
24         int maxNodes = 0;
25         int clusters = 0;
26
27         long ms = System.currentTimeMillis();
28
29         for(int i = 0; i < n; ++i)
30         {
31             if(visited[i] == true)
32                 continue;
33             int nodes = DFS(lg, i, visited);
34             if(maxNodes < nodes)
35                 maxNodes = nodes;
36             clusters++;
37         }
38         ms = System.currentTimeMillis() - ms;
39         System.out.println("Vertices/edges: " + n);
40         System.out.println("Clusters: " + clusters);
41         System.out.println("Largest cluster: " + maxNodes);
42         System.out.println("Cluster analysis: " + ms + " ms");
43     }
44
45     static private int DFS(UndirectedGraph g, int start, boolean[] visited)
46     {
47         int visitedNodes = 1;
48         for(VertexIterator it = g.adjacentVertices(start); it.hasNext();){
49             int i = it.next();
50             if(visited[i])
```

```
51                  continue;
52              visited[i] = true;
53              visitedNodes += DFS(g, i, visited);
54          }
55          return visitedNodes;
56      }
57
58      static private ListGraph generateRandomGraph(int size){ // Uppgift
            specifierar n noder och n kanter
59
60          ListGraph lg = new ListGraph(size);
61          int left = size;
62
63          // Eftersom size antas vara ganska stor är size mycket mindre än
                size^2 vilket ögr
64          // att vi kan öprva oss fram till om en kant passar eller inte.
65          do{
66              int x = rand.nextInt(size), y = rand.nextInt(size);
67              if(lg.areAdjacent(x, y)) //Already existed.
68                  continue;
69              lg.addEdge(x, y);
70              left--;
71          }while(left > 0);
72
73          return lg;
74      }
75
76  }
```

A.2 ListGraph.java

```
1   package kth.csc.inda;
2
3   import java.util.HashMap;
4   import java.util.HashSet;
5   import java.util.Iterator;
6   import java.util.Map;
7   import java.util.Set;
8
9   /**
10   * An undirected graph with a fixed number of vertices implemented using
11   * adjacency lists. Space complexity is O(V + E) where V is the number
12   * of vertices and E the number of edges.
13   *
14   * @author [Name]
15   * @version [Date]
16   */
17  public class ListGraph implements UndirectedGraph {
18      /** Number of vertices in the graph. */
19      private final int numVertices;
20
21      /** Number of edges in the graph. */
22      private int numEdges;
23
24      /**
25       * All vertices adjacent to v are stored in adjacentVertices[v].
```

```
26          * No set is allocated if there are no adjacent vertices.
27          */
28         private final Set<Integer>[] adjacentVertices;
29
30         /**
31          * Edge costs are stored in a hash map. The key is an
32          * Edge(v, w)-object and the cost is an Integer object.
33          */
34         private final Map<Edge, Integer> edgeCosts;
35
36         /**
37          * Constructs a ListGraph with v vertices and no edges.
38          * Time complexity: O(v)
39          *
40          * @throws IllegalArgumentException if v < 0
41          */
42         public ListGraph(int v) {
43             if(v < 0)
44                 throw new IllegalArgumentException();
45             numVertices = v;
46             numEdges = 0;
47
48             // The array will contain only Set<Integer> instances created
49             // in addEdge(). This is sufficient to ensure type safety.
50             @SuppressWarnings("unchecked")
51             Set<Integer>[] a = new HashSet[numVertices];
52             adjacentVertices = a;
53             for(int i = 0; i < numVertices; ++i){
54                 a[i] = new HashSet<Integer>();
55             }
56
57             edgeCosts = new HashMap<Edge, Integer>();
58         }
59
60         /** An undirected edge between v and w. */
61         private static class Edge {
62             // Invariant: v <= w
63             final int v;
64             final int w;
65
66             Edge(int v, int w) {
67                 if (v <= w) {
68                     this.v = v;
69                     this.w = w;
70                 } else {
71                     this.v = w;
72                     this.w = v;
73                 }
74             }
75
76             @Override
77             public boolean equals(Object o) {
78                 if (!(o instanceof Edge))
79                     return false;
80                 Edge e = (Edge) o;
81                 return v == e.v && w == e.w;
```

```
82          }
83
84          @Override
85          public int hashCode() {
86              return 31*v + w;
87          }
88
89          @Override
90          public String toString() {
91              return "(" + v + ", " + w + ")";
92          }
93      }
94
95      /**
96       * Returns the number of vertices in this graph.
97       * Time complexity: O(1).
98       *
99       * @return the number of vertices in this graph
100      */
101     public int numVertices() {
102         return numVertices;
103     }
104
105     /**
106      * Returns the number of edges in this graph.
107      * Time complexity: O(1).
108      *
109      * @return the number of edges in this graph
110      */
111     public int numEdges() {
112         return numEdges;
113     }
114
115     /**
116      * Returns the degree of vertex v.
117      * Time complexity: O(1).
118      *
119      * @param  v vertex
120      * @return the degree of vertex v
121      * @throws IllegalArgumentException if v is out of range
122      */
123     public int degree(int v) throws IllegalArgumentException {
124         if(v >= numVertices || v < 0)
125             throw new IllegalArgumentException();
126         return adjacentVertices[v].size();
127     }
128
129     /**
130      * Returns an iterator of vertices adjacent to v.
131      * Time complexity for iterating over all vertices: O(n),
132      * where n is the number of adjacent vertices.
133      *
134      * @param  v vertex
135      * @return an iterator of vertices adjacent to v
136      * @throws IllegalArgumentException if v is out of range
137      */
```

```
138     public VertexIterator adjacentVertices(int v) {
139         if(v >= numVertices || v < 0)
140             throw new IllegalArgumentException();
141         return new ListGraphVertexIterator(adjacentVertices[v].iterator());
142     }
143 
144     /**
145      * Returns true iff v is adjacent to w.
146      * Time complexity: O(1).
147      *
148      * @param  v vertex
149      * @param  w vertex
150      * @return true iff v is adjacent to w
151      * @throws IllegalArgumentException if v or w are out of range
152      */
153     public boolean areAdjacent(int v, int w) {
154         if(Math.max(v, w) >= numVertices || Math.min(v, w) < 0)
155             throw new IllegalArgumentException();
156         return adjacentVertices[v].contains(w);
157     }
158 
159     /**
160      * Returns the edge cost if v is adjacent to w and an edge cost
161      * has been assigned, -1 otherwise. Time complexity: O(1).
162      *
163      * @param  v vertex
164      * @param  w vertex
165      * @return true iff v is adjacent to w
166      * @throws IllegalArgumentException if v or w are out of range
167      */
168     public int edgeCost(int v, int w) throws IllegalArgumentException {
169         if(Math.max(v, w) >= numVertices || Math.min(v, w) < 0)
170             throw new IllegalArgumentException();
171         return (edgeCosts.containsKey(new Edge(v, w))) ? edgeCosts.get(new
                Edge(v, w)) : -1;
172     }
173 
174     /**
175      * Inserts an undirected edge between vertex positions v and w.
176      * (No edge cost is assigned.) Time complexity: O(1).
177      *
178      * @param  v     vertex position
179      * @param  w     vertex position
180      * @throws IllegalArgumentException if v or w are out of range
181      */
182     public void addEdge(int v, int w) {
183         if(Math.max(v, w) >= numVertices || Math.min(v, w) < 0)
184             throw new IllegalArgumentException();
185         if(adjacentVertices[v].add(w)){ // om det lades till en ny (det
                inte redan fanns en)
186             numEdges++;
187             adjacentVertices[w].add(v);
188         }
189 
190     }
191 
```

```
192     /**
193      * Inserts an undirected edge with edge cost c between v and w.
194      * Time complexity: O(1).
195      *
196      * @param  c edge cost, c >= 0
197      * @param  v vertex
198      * @param  w vertex
199      * @throws IllegalArgumentException if v or w are out of range
200      * @throws IllegalArgumentException if c < 0
201      */
202     public void addEdge(int v, int w, int c) {
203         if (c < 0)
204             throw new IllegalArgumentException();
205 
206         addEdge(v, w);
207         edgeCosts.put(new Edge(v, w), c);
208     }
209 
210     /**
211      * Removes the edge between v and w.
212      * Time complexity: O(1).
213      *
214      * @param  v vertex
215      * @param  w vertex
216      * @throws IllegalArgumentException if v or w are out of range
217      */
218     public void removeEdge(int v, int w) {
219         if(Math.max(v, w) > numVertices)
220             throw new IllegalArgumentException();
221         if(adjacentVertices[v].remove(w)){ // If it existed, remove from
                everywhere
222             numEdges--;
223             adjacentVertices[w].remove(v);
224             edgeCosts.remove(new Edge(v, w));
225         }
226     }
227 
228     /**
229      * Returns a string representation of this graph.
230      *
231      * @return a String representation of this graph
232      */
233     @Override
234     public String toString() {
235         if(numEdges == 0)
236             return "Vertices: 0, Edges: {}";
237 
238         StringBuilder sb = new StringBuilder();
239         sb.append("Vertices: " + numVertices + ", Edges: ");
240         sb.append('{');
241         for(int i = 0;i < numVertices; ++i){
242             for(Iterator<Integer> it = adjacentVertices[i].iterator();
                    it.hasNext();)
243             {
244                 Integer v = it.next();
245                 if(i > v)
```

```
246                     continue;
247                 sb.append('(');
248                 sb.append(i);
249                 sb.append(',');
250                 sb.append(v);
251                 int cost = edgeCost(i, v);
252                 if(cost != -1)
253                 {
254                     sb.append(',');
255                     sb.append(cost);
256                 }
257                 sb.append(')');
258                 sb.append(", ");
259             }
260         }
261 
262         sb.setLength(sb.length() - 2);
263         sb.append('}');
264         return sb.toString();
265     }
266 }
```

A.3 ListGraphTest.java

```
1  package kth.csc.inda;
2  
3  public class ListGraphTest extends AbstractGraphTest {
4      /**
5       * Returns an instance of an UndirectedGraph of the given size.
6       */
7      @Override
8      public UndirectedGraph createGraph(int size) {
9          return new ListGraph(size);
10     }
11 }
```

A.4 ListGraph.java

```
1  /**
2   *
3   */
4  package kth.csc.inda;
5  
6  import java.util.Iterator;
7  import java.util.NoSuchElementException;
8  
9  /**
10  * @author lemming
11  *
12  */
13 public class ListGraphVertexIterator implements VertexIterator{
14     private Iterator<Integer> iter;
15     public ListGraphVertexIterator(Iterator<Integer> it){
16         iter = it;
17     }
18     public boolean hasNext(){
19         return iter.hasNext();
```

```
20      }
21      public int next() throws NoSuchElementException{
22          return iter.next().intValue();
23      }
24  }
```