Datenstrukturen und Algorithmen: Hausübung 7

Felix Schrader, 3053850
Jens Duffert, 2843110
Eduard Sauter, 3053470

30. November 2015

Aufgabe 1

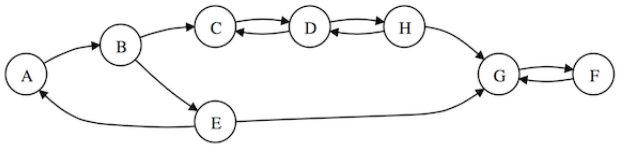
```
1  class AMDFGraph implements ADTUGraph{
2      // Die Knoten werden wie im Tiefendurchlauf geordnet.
3      var adjacencyMatrix, depthFirstList;
4      function create() {
5          // Die Adjazenzmatrix wird als verkettete Liste von
6          // verketteten Listen implementiert.
7          adjacencyMatrix = new LinkedList;
8          adjacencyMatrix.create();
9          // Auch die Knoten werden in einer verketteten Liste
10         // gespeichert.
11         depthFirstList = new LinkedList;
12         depthFirstList.create();
13     }
14     function nodeValue(u) {
15         return depthFirstList.retrieve(u);
16     }
17     function areAdjacent?(u, v) {
18         if(adjacencyMatrix.retrieve(u).retrieve(v) == 0 &&
19                 adjacencyMatrix.retrieve(v).retrieve(u) == 0) {
20             // Wenn die Eintraege in der Adjazenzmatrix mit
21             // Indizes u und v null sind, gibt es keine Kante
22             // zwischen u und v.
23             return false;
24         }
25         return true;
26     }
27     function edgeWeight(u, v) {
28         return adjacencyMatrix.retrieve(u).retrieve(v);
29     }
30     function sort() {
31         // Ordnet die Knoten in der Reihenfolge eines
32         // Tiefendurchlaufs.
33     }
34     function iterationStart() {
35         sort();
36         return depthFirstList.retrieve(0);
37     }
38     function iterationNext(u) {
39         if(u = depthFirstList.length() - 1) {
40             return NIL;
41         }
```

```
42          return u + 1;
43      }
44      function adjacentStart(u) {
45          sort();
46          for(i = 0; i < depthFirstList.length(); i++) {
47              if(adjacencyMatrix.retrieve(i).retrieve(u) != 0) {
48                  return i;
49              }
50          }
51          return NIL;
52      }
53      function adjacentNext(u, v) {
54          for(i = v + 1; i < depthFirstList.length(); i++) {
55              // Sucht beginnend bei v+1 nach einer Kante
56              // (Eintrag ungleich null in der Adjazenzmatrix).
57              if(adjacencyMatrix.retrieve(u).retrieve(i) != 0) {
58                  return i;
59              }
60          }
61          return NIL;
62      }
63      function insert(x) {
64          elements = depthFirstList.length();
65          // Das Element wird ans Ende von depthFirstList angefuegt.
66          depthFirstList.insert(elements, x);
67          // Die Adjazenzmatrix wird mit verketteten Listen
68          // gefuellt.
69          newColumn = new LinkedList;
70          newColumn.create();
71          adjacencyMatrix.insert(newColumn);
72          for(i = 0; i < elements; i++) {
73              // Die letzte Spalte der Matrix wird mit Nullen
74              // gefuellt (da es zu x keine Kanten gibt).
75              adjacencyMatrix.retrieve(i).insert(elements, 0);
76              // Es wird noch eine Zeile fuer x in die Matrix
77              // eingefuegt.
78              adjacencyMatrix.retrieve(elements + 1).insert(i, 0));
79          }
80      }
81      function remove(u) {
82          for(i = 0; i < depthFirstList.length(); i++) {
83              // Die zu u gehoerige Zeile in der Adjazenzmatrix wird
84              // geloescht.
85              adjacencyMatrix.retrieve(u).delete(i);
86          }
87          for(i = 0; i < depthFirstList.length() - 1; i++) {
88              // Die zugehoerige Spalte wird geloescht.
89              adjacencyMatrix.retrieve(i).delete(u);
90          }
91          // Der Knoten wird geloescht.
92          depthFirstList.delete(u);
93      }
94      function changeWeight(u, v, w) {
95          // Das Gewicht der Kante (u, v) wird in der
96          // Adjazenzmatrix geaendert.
97          adjacencyMatrix.retrieve(u).retrieve(v) = w;
98      }
99  }
```

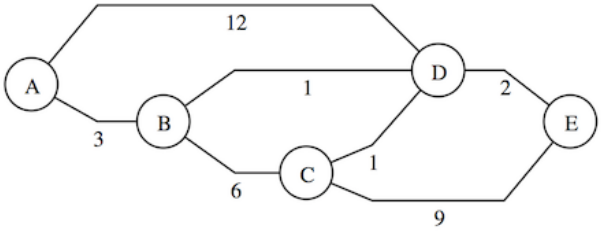
Aufgabe 3

a) Die *graphlib* von cpettit. Hier eine Zusammenfassung des Abschnittes “Graph Concepts” aus der Dokumentation:

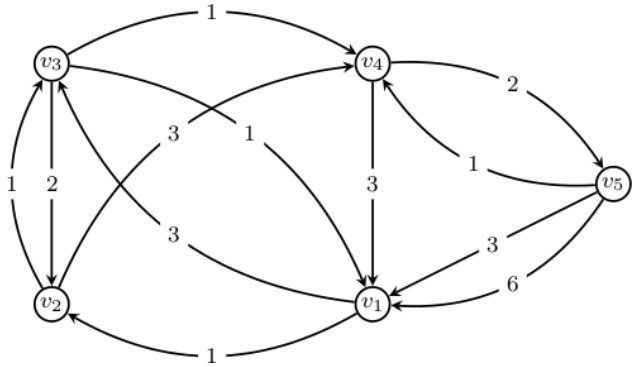
Directed Dies ist der Standard-Typ. Die Kanten sind gerichtet.



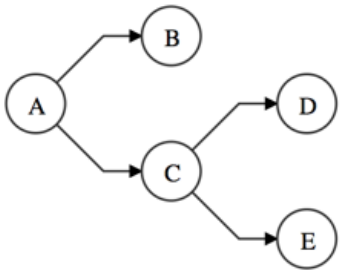
Undirected Diese sind analog zu denen in der Vorlesung.



Multigraph Hier können mehrere Kanten zwei gleiche Knoten verbinden. Nicht zu verwechseln mit einem “Hypergraphen”, bei dem eine Kante mehrere Knoten verbinden kann.



Compound Diese sind wie Bäume mit Wurzeln. Es lässt sich eine Kind-Elternteil Hirarchie auf den Knoten beschreiben.



b) *Der Tarjan Algorithmus*
Das Ziel dieses Algorithmus' ist es, die starken Zusammenhangskomponenten eines gerichteten Graphen zu bestimmen. Der folgende Code ist aus cpettitt's graphlib entnommen:

```
1  function tarjan(g) {
2    var index = 0,
3        stack = [],
4        visited = {}, // node id -> { onStack, lowlink, index }
5        results = [];
6
7    function dfs(v) {
8      var entry = visited[v] = {
9        onStack: true,
10       lowlink: index,
11       index: index++
12     };
13     stack.push(v);
14
15     g.successors(v).forEach(function(w) {
16       if (!_.has(visited, w)) {
17         dfs(w);
18         entry.lowlink = Math.min(entry.lowlink, visited[w].lowlink);
19       } else if (visited[w].onStack) {
20         entry.lowlink = Math.min(entry.lowlink, visited[w].index);
21       }
22     });
23
24     if (entry.lowlink === entry.index) {
25       var cmpt = [],
26           w;
27       do {
28         w = stack.pop();
29         visited[w].onStack = false;
30         cmpt.push(w);
31       } while (v !== w);
32       results.push(cmpt);
33     }
34   }
35
36   g.nodes().forEach(function(v) {
37     if (!_.has(visited, v)) {
38       dfs(v);
39     }
40   });
41
42   return results;
43 }
```

Der Algorithmus ähnelt der in der Vorlesung besprochenen Tiefensuche, Die Buchführung des Algorithmus ist jedoch abgewandelt. Es gibt noch die zusätzlich die Variablen

- `stack`
- `v.onStack`
- `v.index`
- `v.lowlink`

Es ist der Index die Reihenfolge, in denen die Knoten besucht wurden, diese Entspricht wieder genau der Tiefensuche, in `lowlink` ist jetzt jedoch auch die Information enthalten, welcher Index von einem Knoten aus erreichbar ist. Wenn ein Knoten a sich auf dem Stack unter einem anderen Knoten b befindet, dann gibt es einen Pfad $a \to b$. Dies impliziert auch, dass `a.index < b.index` ist. Falls umgekehrt `b.lowlink == a.lowlink` dann gibt es auch einen Pfad $b \to a$.

Diese Überlegung begründet Zeile 19-20 des Algorithmus: Wenn ein Knoten "grau" ist, also auf dem Stack, und der `lowlink` des Knoten auf dem Stack kleiner ist, dann haben wir einen Pfad zu- und von diesem Knoten gefunden, der `lowlink` des momentanen Knotens wird also herabgesetzt und der Knoten wird als Teil der starken Zusammenhangskomponente, in welcher der Knoten auf dem Stack ist, vermerkt. Salopp gesagt "sucht" der Algorithmus den niedrigsten Index eines anderen Knotens auf dem Stack der von diesem aus erreichbar ist. Wenn ein niedrigerer Index als `v.lowlink` gefunden wird, dann ist der Knoten Teil einer Zusammenhangskomponente welcher zurzeit in Bearbeitung ist.

Die Knoten werden also in Reihenfolge ihrer Besuchung auf den Stack getan. Wenn am Ende des Funktionsaufrufs kein Knoten mit geringerem Index gefunden wurde, dann werden alle Knoten über dem jetztigen auf dem Stack als Teil dessen starker Zusammenhangskomponente erachtet und vom Stack genommen. Die Knoten "bleiben" also nur dann auf dem Stack, wenn sie in einer Zusammenangskomponente eines anderen Knoten auf dem Stack sind.

Man kann den ersten besuchten Knoten einer starken Zusammenhangskomponente auszeichnen, da er den geringsten Index besitzt. Nur für einen solchen Knoten kann `lowlink == index` gelten. Wenn dies nun am Ende des Aufrufs von `dfs` gilt, dann sind alle Knoten über der Wurzel auf dem Stack in einer Starken Zusammenhangskomponente mit dieser. Die Zusammenhangskomponenten haben also alle ihren eigenen `index`, welcher durch den Index des ersten besuchten Knotens der Komponente gegeben ist.