Datenstrukturen und Algorithmen: Hausübung 7

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

3. Dezember 2015

Aufgabe 1

```
1  class AdjMatGraph implements ADTUGraph{
2      // Die Knoten werden wie im Tiefendurchlauf geordnet.
3      var adjacencyMatrix, nodeList;
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         nodeList = new LinkedList;
12         nodeList.create();
13     }
14     function nodeValue(u) {
15         return nodeList.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 iterationStart() {
31         return nodeList.retrieve(0);
32     }
33     function iterationNext(u) {
34         if(u = nodeList.length() - 1) {
35             return NIL;
36         }
37         return u + 1;
38     }
39     function adjacentStart(u) {
40         for(i = 0; i < nodeList.length(); i++) {
41             if(adjacencyMatrix.retrieve(i).retrieve(u) != 0) {
```

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

In Tabelle 1 wurden die Laufzeiten und Speicher der ADTLinkedList aus der Vorlesung verwendet. Laufzeiten über $\mathcal{O}(1)$ kommen hier durch Iterationen über alle Knoten und die Laufzeit $\mathcal{O}(|V|)$ von delete hervor.

Funktion	Laufzeit	Speicher
create	$\mathcal{O}(1)$	$\mathcal{O}(1)$
nodeValue	$\mathcal{O}(1)$	$\mathcal{O}(1)$
areAdjacent?	$\mathcal{O}(1)$	$\mathcal{O}(1)$
edgeWeight	$\mathcal{O}(1)$	$\mathcal{O}(1)$
iterationStart	$\mathcal{O}(1)$	$\mathcal{O}(1)$
iterationNext	$\mathcal{O}(1)$	$\mathcal{O}(1)$
adjacentStart	$\mathcal{O}(\|V\|)$	$\mathcal{O}(1)$
adjacentNext	$\mathcal{O}(\|V\|)$	$\mathcal{O}(1)$
insert	$\mathcal{O}(\|V\|)$	$\mathcal{O}(1)$
remove	$\mathcal{O}(\|V\|^2)$	$\mathcal{O}(1)$
changeWeight	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Tabelle 1: Laufzeiten und Speicherbedarf der Funktionen des ADTÜGraph

Aufgabe 2

a) In dieser Aufgabe soll der minimale Spannbaum von Punkt A durch das Prim-Verfahren bestimmt werden. Dies wurde grafisch dargestellt. Es wurde darauf verzichtet, alle Wege bei der Bestimmung farbig zu markieren.
Der Startpunkt ist A. Von A ist es möglich drei Wege zu gehen. 1.Weg zu D (Wertung 2), 2.Weg zu C (Wertung 9) und 3.Weg zu E (Wertung 9). In der Abbildung 1 links oben ist dies zu sehen. Es wird der Weg mit der geringsten Wertung genommen und somit der zu D. Von D aus ist es

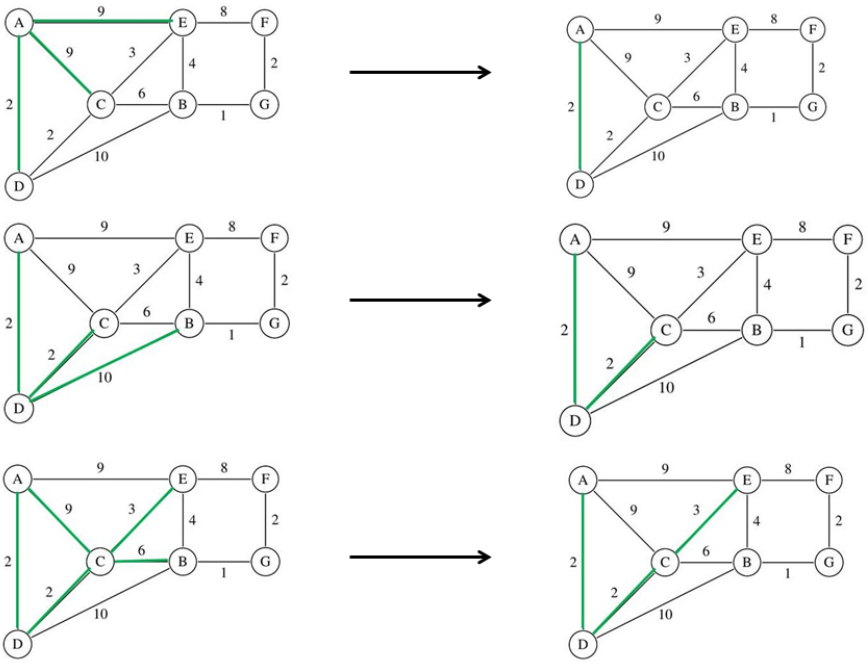


Abbildung 1: Grafik für Verfahren von Prim

möglich zu C (Wertung 2) und zu B (Wertung 10) zu gelangen. Also sind jetzt die Wege zu E, C (je Wertung 9) und zu C (Wertung 2) und B (Wertung 10) zur Verfügung. Auch in diesem Fall wird der Weg mit der geringsten Wertung genommen. Dies wird mit allen weiteren Wegen gemacht. Das Ergebnis des minimalen Spannbaum ist in Abbildung 2 rechts unten zu sehen.

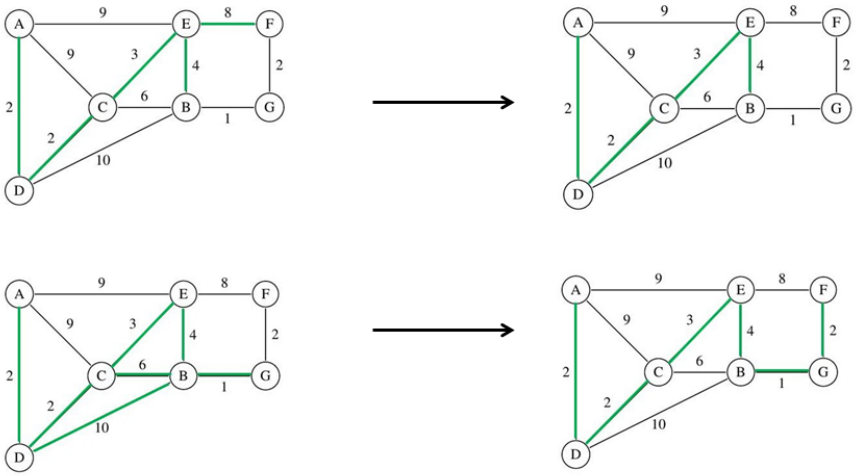


Abbildung 2: Grafik für Verfahren von Prim

Dieses lautet:

$$A \to D \to C \to E \to B \to G \to F$$

Dieser hat einen Wert von:

$$2 + 2 + 3 + 4 + 1 + 2 = 17$$

b) Es soll nun mit dem Dijkstra Algorithmus der kürzestes Weg von A zu F bestimmt werden. Dabei wurden die ? in den Kästen wo noch keine Informationen vorhanden sind weggelassen.
1.Schritt

	A	B	C	D	E	F	G	PriorityQueue
$dist_1$	0			2	9			D_2
$pred_1$	Nil			A	A			

2.Schritt

	A	B	C	D	E	F	G	PriorityQueue
$dist_1$	0	12	4	2	9			D_2
$pred_1$	Nil	D	D	A	A			

3.Schritt

	A	B	C	D	E	F	G	PriorityQueue
$dist_1$	0	10	4	2	9			D_2
$pred_1$	Nil	C	D	A	A			

4.Schritt

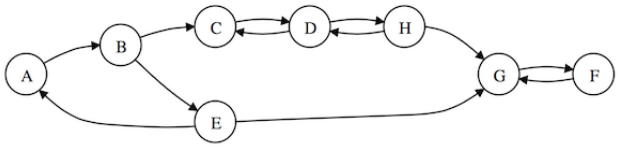
	A	B	C	D	E	F	G	PriorityQueue
$dist_1$	0	10	4	2	7	15		D_2
$pred_1$	Nil	C	D	A	C	E		

Dies Ergebnis ist nicht effektiv da der kürzestes Weg 14 (Aufgabenteil a) ist und nicht 15 wie mit diesem Algorithmus

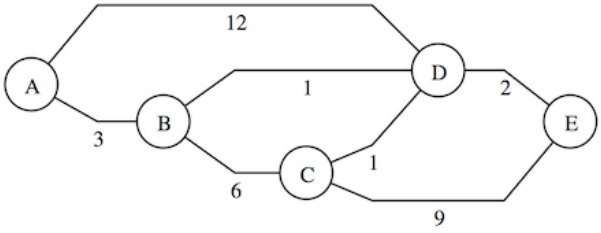
Aufgabe 3

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

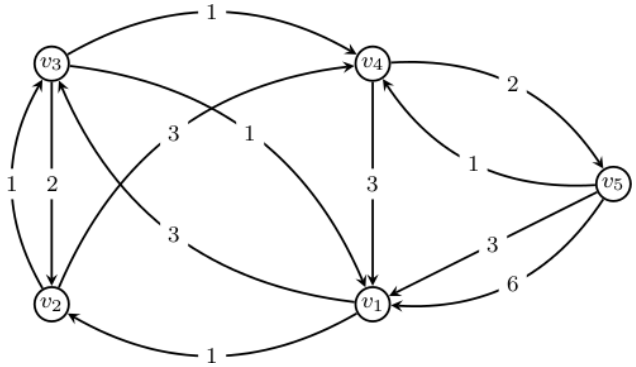
Directed Dies ist der Standard-Typ. Die Kanten sind gerichtet.



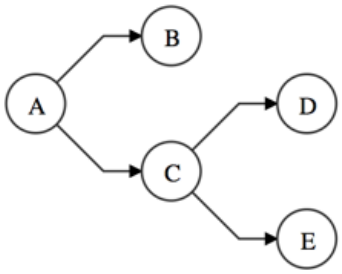
Undirected Diese sind analog zu denen in der Vorlesung.



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, dieser entspricht wieder genau dem 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.