37. Bundeswettbewerb Informatik

1. Runde

01.09.2018 - 26.11.2018

Aufgabe 1 - Superstar

Nico Jeske (48312)

Team: ITler (00140)

Ich versichere hiermit, die vorliegende Arbeit selbstständig und unter den Wettbewerbsregeln des Bundeswettbewerbs Informatik, ohne fremde Hilfe und ohne die Verwendung anderer, als der in den Quellen angegebenen Hilfsmitteln, angefertigt zu haben.

Nico Jeske, den 12.10.2018

1 Lösungsidee

1.1 Repräsentation als Graph

Das Soziale Netzwerk kann als gerichteter Graph modelliert werden. Hierbei repräsentiert ein Knoten V eine Person p im Sozialen Netzwerk. Die Edges E repräsentieren dann die Abonnements der jeweiligen Personen. Eine gerichtete Kante $(p_1, p_2) \in E$ würde z.B. aussagen, dass Person p_1 Person p_2 folgt. Zur besseren Vorstellung ist im folgendem die erste Beispieldatei als Graph dargestellt.

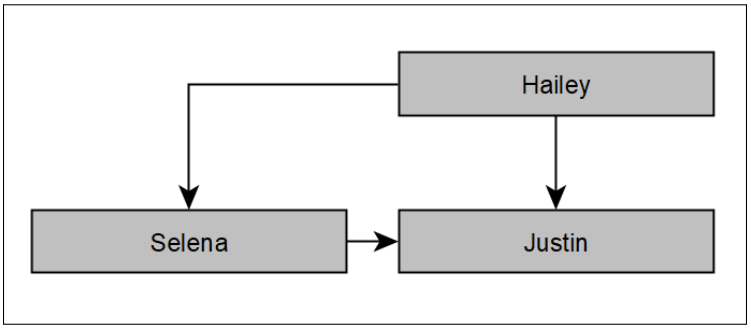

Figure 1: Darstellung der ersten Beispieldatei

1.2 Finden des Superstars

Das Ziel der Aufgabe besteht nun darin, wenn möglich einen Superstar $s \in V$ zu finden. Damit eine Person als Superstar gilt, muss sie folgende Bedingungen erfüllen:

1. $\forall p \in V \setminus s : (p, s) \in E$ gilt. Die Person also von allen anderen Personen der Gruppe gefolgt wird.
2. $\forall p \in V \setminus s : (s, p) \notin E$ gilt. Die Person also niemandem aus der Gruppe folgt.

Aus dem Graphen könnten nun also theoretisch alle Personen $p \in V$ durch gegangen werden und auf die Bedingungen für einen Superstar überprüft werden. In der Rahmen der Aufgabe ist der Werbeagentur der Graph jedoch nicht bekannt. Einzig und allein die Personen $p \in V$ sind bekannt. Die Abonnements $(p_1, p_2) \in E$ müssen jedoch einzeln in der Form "folgt Person p_1 Person p_2?", (gilt $(p_1, p_2) \in E$?) abgefragt werden. Dabei ist zu beachten, dass jede Frage einen Kostenpunkt von einem Euro aufweist. Die Anzahl der Anfragen sollte also minimiert werden.

Um nun einen möglichen Superstar s zu bestimmen, verwende ich folgende Methode:
Sei die Liste $C : (c_i | i \in C)$ die möglichen Superstars.
Sei die Menge Q eine Menge aller getätigten Abfragen $(p_1, p_2) \in E$.
Zu Beginn sind alle Personen mögliche Superstars, es gilt also $\forall p \in V : p \in C$.
Es wird ein Kandidat $c \in C$ als Startkandidat ausgewählt. Im Anschluss wird schließlich durch alle anderen Kandidaten $p \in C$ durchiteriert und geprüft, ob $(c, p) \in E$, also ob der Kandidat c dem Kandidaten p folgt. Wenn dies zutrifft ist klar, dass c kein Superstar sein kann, da er einer anderen Person folgt und somit die 2. Bedingung als Superstar nicht erfüllt. Sollte dies jedoch nicht zutreffen, c also nicht p folgt, steht fest, dass p kein Superstar sein kann, da ihm nicht alle Personen folgen können (Bedingung 1).
Im folgenden ist dieser Ansatz als Pseudocode dargestellt-

Algorithm 1 Superstar

Require: Personen V, Abonnements E

```
1: List C ← V                                   ▷ Mögliche Superstars
2: Set Q                                        ▷ Getätigte Anfragen
3: Candidate c ← C.get(0)                       ▷ Derzeitigen Kandidaten auswählen
4: querys ← 0                                   ▷ Anzahl getätigter Anfragen
5: while C.size < 1 do                          ▷ Bis nur ein Kandidat übrig bleibt.
6:     for each p ∈ C do                        ▷ Für alle anderen Kandidaten p ∈ C
7:         if c = p then                        ▷ Gleicher Kandidat: Continue
8:             continue
9:         end if
```

```
10:         if (c, p) ∈ Q then                          ▷ Schonmal abgefragt: Continue
11:             continue
12:         end if
13:         querys ← querys + 1
14:         Q.add((c, p) ∈ E)
15:         if (c, p) ∈ E then                          ▷ Der derzeitige Kandidat c folgt p
16:             C.remove(c)                             ▷ c ist kein Kandidat mehr
17:             c ← p                                   ▷ p wird der neue derzeitige Kandidat
18:             break
19:         else                                        ▷ Da p nicht von c gefolgt wird, kann p kein Superstar sein.
20:             C.remove(p)
21:         end if
22:     end for
23: end while
24: return Einziger übrig gebliebener Kandidat c und getätigte Anfragen Q
```

Nachdem nun die Kandidatenmenge auf einen einzigen Kandidaten runtergebrochen wurde, muss nun als letzter Schritt überprüft werden, ob dieser wirklich alle Bedingungen erfüllt. Somit werden also alle Beziehungen zwischen dem letzten Kandidaten c und den Personen im Sozialen Netzwerk überprüft, wobei bereits betätigte Anfragen in Q natürlich nicht wiederholt werden.

Im ersten Schritt wird hierbei zuerst kontrolliert, ob der Kandidat c wirklich keiner anderen Person folgt und somit der folgende Term zutrifft: $\forall p \in V : (c, p) \notin E$
Im zweiten Schritt wird dann schließlich geprüft, ob wirklich alle Personen $p \in V$ dem Kandidaten c folgen, also folgender Term zutrifft: $\forall p \in V : (p, c) \in E$

Wurden nun also beide Terme erfüllt, so ist der Kandidat c der Superstar der Gruppe. Wurden jedoch nicht beide Terme erfüllt, so besitzt die Gruppe keinen Superstar.

1.3 Minimierung der Kosten

Die hier hergeleitete Methode schafft es die Kosten durch die Anfragen sehr niedrig zu halten. Dies hat mehrere Gründe. Zuerst einmal wird zur Minimierung der Kosten auch jede getätigte Anfrage gespeichert, so dass eine Abfrage nicht unnötigerweise wiederholt wird. Des weiteren werden nur sehr wenige Anfragen gebraucht, um aus der gesamten Menge an Kandidaten einen einzigen Kandidaten herauszufiltern, welcher der Superstar der Gruppe sein kann. Um genau zu sein, werden für das filtern der Kandidatenliste nur $n-1$ Anfragen benötigt, wobei $n = |V|$ ist. Dies hat den Grund, dass bei dem Filtern der Kandidaten in jeder Iteration immer ein Kandidat entfernt wird.

Um nun im letzten Schritt zu bestätigen, dass der letzte übrig gebliebene Kandidat auch wirklich der Superstar der Gruppe ist müssen nun natürlich noch weitere Abfragen gemacht werden, die sich auch nicht vermeiden lassen, sich aber aufgrund der Speicherung der getätigten Abfragen auf maximal $2*(n-1)$ beläuft.

Schlussfolgernd lässt sich also sagen, dass das finden des Superstars in der Gruppe maximal $3 * (n-1)$ Anfragen finden lässt, die kosten sich also auf maximal $3 * (n-1)$ belaufen, meist aber unter diesem Maximum liegen.

2 Implementation

Die Aufgabe wird in Java implementiert. Der Aufbau des Netzwerkes wird aus der Eingabedatei mit einem BufferedReader eingelesen. Aus diesem wird schließlich mit Hilfe der Google Guava Bibliothek ein MutableGraph erstellt, welcher das Soziale Netzwerk wie in der Lösungsidee beschrieben als Graph darstellt. Dieser Graph wird dann schließlich der Funktion *solve* übergeben, welche den in der Lösungsidee beschriebenen Algorithmus anwendet.

Zum Speichern der getätigten Anfragen wird hierbei eine Hashmap verwendet, bei der der Schlüssel eine Person p repräsentiert und der dazugehörige Wert eine Liste an Personen $s \in P$, zu denen die Abfrage $(p, s) \in E$ gestellt wurde.

Die Kandidaten und die Personen des Netzwerks werden in einer simplen Liste gespeichert.

Der Algorithmus zum finden des Superstars an sich hält sich genaustens an den in der Lösungsidee beschriebenen Pseudocode und benötigt keine genauere Betrachtung.

3 Beispiele

Datei	Superstar	Kosten
superstar1.txt	Justin	4€
superstar2.txt	Dijkstra	9€
superstar3.txt	N/A	N/A
superstar4.txt	Folke	181€

4 Quellcode

Listing 1: Einlesen des Sozialen Netzwerks

```
/**
   * Creates a relationship graph from the input file.
   * The persons are the nodes and their relationship the directed edges
   *
   * @param br the file with the relations
   * @return A relation-graph from the given relations
   * @throws IOException              Error while reading the file
   * @throws IllegalArgumentException Error while analyzing the file
   */
  private static MutableGraph<Person> initialize(@NotNull BufferedReader br) throws
  IllegalArgumentException, IOException {
    debug("Loading persons...");
    StringTokenizer stringTokenizer = new StringTokenizer(br.readLine());
    int numberOfPeople = stringTokenizer.countTokens();
    //Hashmap with name -> person
    Map<String, Person> people = new HashMap<>();

    if (numberOfPeople <= 1)
      throw new IllegalArgumentException("The network must consist of more than one person.");


    debug("Number of people=" + numberOfPeople);
    debug("Creating graph...");
    MutableGraph<Person> relationGraph = GraphBuilder.directed().expectedNodeCount(numberOfPeople)
                                         .build();

    //Adds the persons to the graph
    while (stringTokenizer.hasMoreTokens()) {
      String name = stringTokenizer.nextToken();
      Person newPerson = new Person(name);
      people.put(name, newPerson);
      relationGraph.addNode(newPerson);
      debug("Created person '" + name + "'");
    }

    //Adds the relations to the graph
    String currentLine;
    while (!Strings.isNullOrEmpty(currentLine = br.readLine())) {
      StringTokenizer relationTokens = new StringTokenizer(currentLine);

      if (relationTokens.countTokens() != 2)
        throw new IllegalArgumentException("Invalid relationship specification in the input file");

      String relationFromString = relationTokens.nextToken();
      String relationToString = relationTokens.nextToken();
      Person relationFromPerson = people.get(relationFromString);
      Person relationToPerson = people.get(relationToString);

      //saves the newly created edge
      relationGraph.putEdge(relationFromPerson, relationToPerson);
    }

    debug("Created graph with " + relationGraph.nodes().size() + " persons and "
            + relationGraph.edges().size() + " relationships");
    return relationGraph;
  }
```

Listing 2: Finden des Superstars

```
/**
   * Finds the superstar in a given relationship graph
   *
   * @param relationGraph relationship graph
   */
  private static void solve(MutableGraph<Person> relationGraph) {
    //query counter for the invoice
    int querys = 0;

```

```
10      //DEBUG
11      int lastQueryCount;
12
13
14      //Allready checked relationships.
15      Map<Person, List<Person>> checkedRelations = new HashMap<>();
16      relationGraph.nodes().forEach(person -> checkedRelations.put(person, new ArrayList<>()));
17
18      //All persons in the network are possible candidates
19      List<Person> candidates = new ArrayList<>(relationGraph.nodes());
20      //All persons in the network
21      List<Person> people = new ArrayList<>(relationGraph.nodes());
22
23      //Graph of the inquired relationships
24      MutableGraph<Person> askedGraph = GraphBuilder.directed().build();
25      //Copy over the persons
26      people.forEach(askedGraph::addNode);
27
28      //First Candidate
29      Person person = candidates.get(0);
30
31      debug("Finding candidate");
32
33      //Sort out the candidates until there is only one candidate left:
34      //1. Go through all the other candidates where the relationship status has not yet been checked.
35      //2.1. If there is a relationship between the current candidate and another candidate,
36      //     the current candidate cannot be a superstar and is removed while the algorithm
37      //     is repeated for the other candidate.
38      //2.2. Otherwise, if there is no relationship between the current candidate and another candidate,
39      //     then the other candidate cannot be a superstar, since not all of them follow him and are
40      //     therefore deleted from the candidate list.
41      while (candidates.size() > 1) {
42        for (int i = 0; i < candidates.size(); i++) {
43          Person testedPerson = candidates.get(i);
44
45          //Same person
46          if (testedPerson == person)
47            continue;
48
49          //Already checked
50          if (checkedRelations.get(person).contains(testedPerson))
51            continue;
52
53          //debug("Test relation between " + person + " and " + testedPerson);
54          querys++;
55          checkedRelations.get(person).add(testedPerson);
56          //Has a relationship -> remove current candidate. new candidate = testedPerson
57          if (relationGraph.hasEdgeConnecting(person, testedPerson)) {
58            candidates.remove(person);
59            askedGraph.putEdge(person, testedPerson);
60            person = testedPerson;
61            break;
62          } else {
63            //No relationship -> Remove tested Person
64            candidates.remove(testedPerson);
65          }
66        }
67      }
68
69      //Getting the only possible candidate
70      Person lastCandidate = candidates.get(0);
71      debug("Found candidate " + lastCandidate + " with " + querys + " querys.");
72      //info("Letzter Kandidat  f  r  den Superstar ist " + lastCandidate);
73
74      lastQueryCount = querys;
75      debug("Check that the candidate doesnt follow anyone.");
76
77      //-------------------------------------------------------
78      //     - Check if all conditions are met for candidate -
79      //-------------------------------------------------------
80
81      //Testing if the last candidate follows no one
82      for (Person p : people) {
83        if (p == lastCandidate)
84          continue;
85
86        if (checkedRelations.get(lastCandidate).contains(p))
87          continue;
88
89        //debug("Test relation between " + lastCandidate + " and " + p);
90        querys++;
91        checkedRelations.get(lastCandidate).add(p);
92        if (relationGraph.hasEdgeConnecting(lastCandidate, p)) {
93          error(lastCandidate + " ist kein Superstar, da er " + p + " folgt.");
94          return;
95        }
96      }
```

```
97      debug("Completed check with " + (querys - lastQueryCount) + " querys");
98
99
100     lastQueryCount = querys;
101     debug("Check that really everyone follows the candidate");
102     //Check that really everyone follows the candidate
103     for (Person p : people) {
104       if (p == lastCandidate)
105         continue;
106
107       if (checkedRelations.get(p).contains(lastCandidate))
108         continue;
109
110       querys++;
111       checkedRelations.get(p).add(lastCandidate);
112       if (!relationGraph.hasEdgeConnecting(p, lastCandidate)) {
113         error(lastCandidate + " ist kein Superstar, da ihm nicht alle folgen.");
114         error("Es ist kein Superstar in dieser Gruppe vorhanden.");
115         return;
116       } else {
117         askedGraph.putEdge(p, lastCandidate);
118       }
119     }
120     debug("Completed check with " + (querys - lastQueryCount) + " querys");
121
122     //Last check for safety
123     int connnectionsToCandidate = askedGraph.inDegree(lastCandidate);
124     if (connnectionsToCandidate != askedGraph.nodes().size() - 1) {
125       error(lastCandidate + " ist kein Superstar, da ihm nicht alle folgen? (Unexpected Error)");
126       error("Es ist kein Superstar in dieser Gruppe vorhanden.");
127       return;
128     }
129
130     info(lastCandidate + " ist der SUPERSTAR der Gruppe.");
131     info("Es wurden " + querys + " Anfragen ben tigt. Kosten: " + querys + "    ");
132   }
```

5 Quellen

- Guava: https://github.com/google/guava
- minlog: https://github.com/EsotericSoftware/minlog