Aufgabe 1: Superstar

Team-ID: 00133

Team-Name: CubeFlo

Bearbeiter dieser Aufgabe:
Florian Rädiker (15 Jahre)

20. Oktober 2018

Inhalt

Lösungsidee

Ein Superstar muss genau zwei Bedingungen erfüllen: Er/Sie darf keinem folgen, aber alle anderen müssen dem Superstar folgen.
Grundsätzlich geht das Programm alle Mitglieder einer Gruppe durch und prüft jeweils mithilfe dieser zwei Bedingungen, ob es sich um den Superstar handelt.
Hinzu kommen zusätzliche Informationen, die beim Prüfen einer Bedingung entstehen. Diese grenzen die möglichen Superstars ein oder geben einen Hinweis auf einen möglichen Superstar, der dann als nächstes getestet wird. Dafür gibt es verschiedene Optimierungen, die im Folgenden aufgelistet sind.

Optimierungen

Optimierung 1: Non-Superstars

Bei jeder gestellten Anfrage „Folgt Y Z?“ ist bei wahrer Aussage bekannt, dass Y kein Superstar ist, da ein echter Superstar niemandem folgen würde. Bei falscher Aussage ist bekannt, dass Z kein Superstar ist, da Y Z nicht folgt, aber jeder würde dem echten Superstar folgen. Diese Informationen werden beim Prüfen einer Bedingung gesammelt.

Optimierung 2: Hinweis auf möglichen Superstar

Beim Prüfen beider Bedingungen gibt es bei falscher Aussage eine Person, die aufgrund der getesteten Bedingung der Superstar sein könnte und deshalb als nächstes getestet wird (anstatt des nächsten Gruppenmitglieds in der Reihe).
Beim Prüfen einer Bedingung wird der mögliche Superstar nacheinander in Bezug auf alle Mitglieder gemäß der Bedingung geprüft. Sollte es einen Bezug auf eine Person geben, und dieser Bezug widerlegt die Bedingung, so könnte die Person der Superstar sein.
Wenn geprüft wird, ob der mögliche Superstar niemandem folgt, und es wird eine Person gefunden, der der Superstar doch folgt, dann könnte diese Person der Superstar sein, da zumindest einer (nämlich der vermeintliche Superstar) dieser Person folgt.
Bei der anderen Bedingung könnte man eine Person finden, die dem möglichen Superstar nicht folgt. Damit ist die Bedingung widerlegt und die gefundene Person könnte der Superstar sein, da diese zumindest dem vermeintlichen Superstar nicht folgt (und der echte Superstar folgt ja niemandem).
Sollte solch ein Hinweis gefunden werden, wird die gefundene Person als nächstes überprüft.

Optimierung 3: Bedingungen mit unterschiedlichen Personen prüfen

Beim Prüfen einer Bedingung für einen möglichen Superstar wird dieser solange in Bezug auf alle Mitglieder gemäß der Bedingung getestet, bis entweder alle durch sind und die Bedingung damit erfüllt ist oder aber die Bedingung in Bezug auf ein Mitglied nicht erfüllt ist.
Diese Überprüfungen auf die einzelnen Mitglieder würden sich, wenn die Mitglieder sich in einer gewöhnlichen Liste befänden, meist nur auf die ersten paar Namen in der Liste beziehen, solange bis die Bedingung nicht erfüllt ist. Deshalb wird eine Liste verwendet, die bei Abbruch einer Iteration das nächste Mal an der Stelle anfängt, wo abgebrochen wurde. Dadurch werden alle Namen mal überprüft und es werden mehr Non-Superstars und Hinweise auf einen möglichen Superstar gefunden, da diese ja immer von den Mitgliedern, die in Bezug zum möglichen Superstar gesetzt werden, abhängig sind (siehe Optimierungen 1 und 2).

Optimierung 4: Anfragen cachen

Anfragen werden weder doppelt gezählt noch doppelt ausgeführt, da Werbetreibende ansonsten unnötige Ausgaben hätten. Hier wird das bequem über das Cachen der Funktion, die die Anfragen auswertet, gelöst (siehe „Quellcode > class TeeniGroup > Ergänzende Funktionen“. Echte Werbetreibende würden die Ergebnisse der Anfragen natürlich lokal zum Beispiel auf dem eigenen Server speichern, um diese auch zu späterer Zeit (sie könnten sich aber vielleicht etwas verändert haben) ohne Kosten erreichen zu können.

Optimierung 5: Optimale Reihenfolge beim Prüfen der Bedingungen

Die beiden Bedingungen für einen Superstar können in zwei verschiedenen Reihenfolgen geprüft werden. Wenn die eine Bedingung nicht erfüllt ist, muss die andere auch nicht getestet werden; dadurch werden Anfragen gespart. Also wird die Bedingung, die mit weniger Anfragen auskommt, zuerst getestet. Unter „Umsetzung > Optimale Bedingungsreihenfolge beim Prüfen ermitteln“ wird durch Ausprobieren ermittelt, welche Bedingungsreihenfolge weniger Anfragen verbraucht.

Optimierung 6: Reihenfolge der möglichen Superstars

Am Anfang der Suche nach dem Superstar muss ein Mitglied ausgewählt werden, das als erstes geprüft wird. Es existiert eine Liste aller Mitglieder, die in zwei Formen vorliegt: Einmal als spezielle Liste aus Optimierung 3, die die ganze Zeit unverändert mit allen Mitgliedern erhalten bleibt, und einmal als Liste mit noch nicht ausgeschlossenen Superstars, die zu Anfang natürlich alle Mitglieder enthält. Die spezielle Liste aus Optimierung 3 stellt beim Prüfen einer Bedingung zunächst das erste Mitglied der Liste bereit, aber wenn sich jetzt der echte Superstar ganz am Ende der Liste befindet, so dauert es eine ganze Weile, bis er gefunden wird. Deshalb wird immer, wenn beim letzten Prüfen kein Hinweis auf einen neuen Superstar gefunden wurde (Optimierung 2), was ja zu Beginn der Fall ist, der letzte Superstar aus der Liste der noch nicht ausgeschlossenen genommen. So bleibt Wahrscheinlichkeit, dass der echte Superstar lange versteckt bleibt, gering.

Optimierung 7: Feststellen, ob es einen Superstar gibt – durch Prüfen der zweiten Bedingung

Wenn die erste Bedingung (egal welche) positiv ausfällt, dann ist nun bekannt, dass es in dieser Gruppe entweder einen Superstar gibt (nämlich der, der gerade geprüft wird) oder keinen gibt. Wenn lediglich eine der beiden Bedingungen erfüllt ist, heißt das, dass alle Mitglieder (außer das, das gerade getestet wird), als Superstars ausgeschlossen wurden, da sie entweder alle dem möglichen Superstar folgen oder weil mindestens der mögliche Superstar ihnen nicht folgt (durch die zwei Bedingungen). Daher ist der mögliche Superstar, der gerade getestet wird, bei positiver zweiter Bedingung der echte Superstar, bei negativer gibt es keinen.

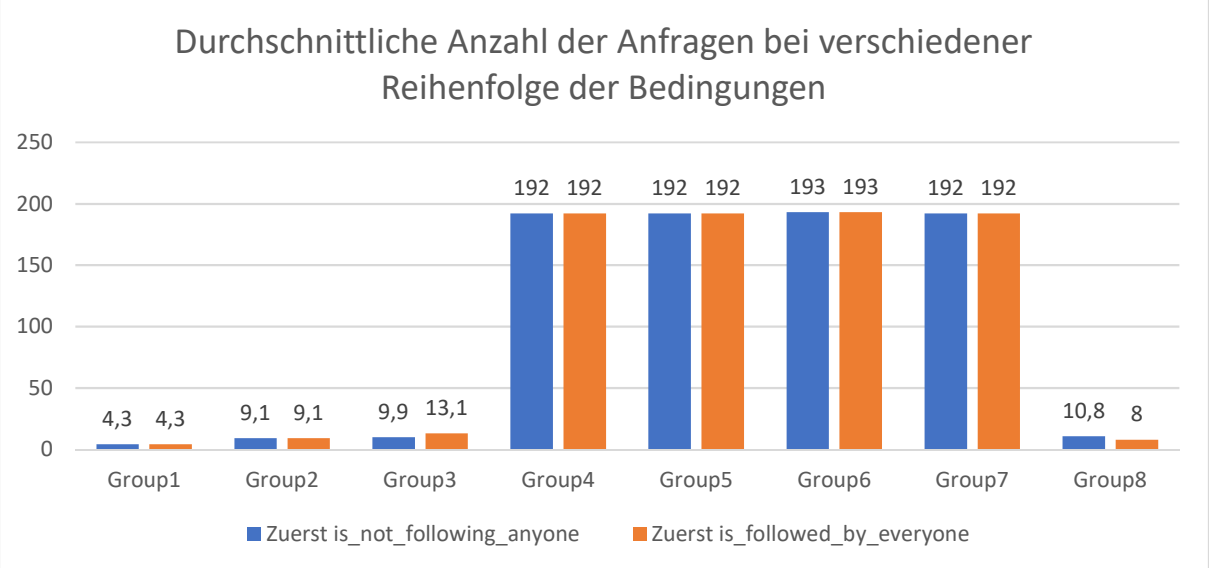
Umsetzung

Das Programm ist in Python 3 geschrieben.
Am wichtigsten ist die Klasse `TeeniGroup` im Modul `Aufgabe1/teenigroup.py`, die eine Gruppe des Netzwerks darstellt und alle Mitglieder und Verknüpfungen als Attribute speichert und hauptsächlich die Funktion `TeeniGroup.get_superstar` bereitstellt, die versucht, mit möglichst wenig Anfragen den Superstar zu finden. Außerdem gibt es zwei Funktionen, die die zwei Bedingungen prüfen, und zwar `is_not_following_anyone` und `is_followed_by_everyone`. Diese beiden Funktionen geben auch die Non-Superstars aus Optimierung 1 und den Hinweis auf einen möglichen Superstar – sofern einer gefunden wurde – aus Optimierung 2 zurück.
`TeeniGroup.get_superstar` läuft solange in einer `while`-Schleife, bis entweder eine Person beide

Bedingungen erfüllt, also Superstar ist, oder alle Mitglieder ausgeschlossen wurden (siehe Optimierung 7) und so kein Superstar in dieser Gruppe existiert.
Während des Durchlaufens der Mitglieder gibt es eine Liste, die die noch nicht ausgeschlossenen Mitglieder speichert, und von der alle Non-Superstars, die bei einer Prüfung durch die bedingungsprüfenden Funktionen gefunden werden, entfernt werden.
Bei jedem Schleifendurchlauf wird außerdem zu Beginn ein neues Mitglied ausgewählt, das nun auf die Bedingungen geprüft wird. Bei dieser Person handelt es sich um den Hinweis auf den Superstar, wenn es im letzten Durchlauf einen Hinweis auf den Superstar gab (Optimierung 2), oder aber die Person wurde bereits ausgeschlossen beziehungsweise es gab im letzten Durchlauf keinen Hinweis, dann wird das letzte Element der Liste mit den noch nicht ausgeschlossenen Superstars genommen (Optimierung 6).
Ein Attribut der Klasse (`TeeniGroup.names`) wird von den beiden bedingungsprüfenden Funktionen genutzt, um durch alle Mitglieder der Gruppe zu iterieren. Dabei handelt es sich um die spezielle Liste aus Optimierung 3, die hier in Form der Klasse `LoopIteratingList` (siehe Quellcode) implementiert ist.

Optimale Bedingungsreihenfolge beim Prüfen ermitteln

Nach Optimierung 5 sollte eine Reihenfolge beim Prüfen der Bedingungen gewählt werden, die möglichst wenig Anfragen verbraucht.
Um die zwei Reihenfolgen, in der die Bedingungen geprüft werden können, auszuprobieren, werden beide durch jeweils leichte Veränderung am Code mit der Funktion `TeeniGroup.get_mean` (siehe „Ergänzende Funktionen“ der `TeeniGroup`-Klasse im Quellcode) mit 100.000 Durchläufen von `get_superstar` auf die durchschnittliche Anzahl der Anfragen geprüft.
Hier werden auch zusätzliche Gruppen aus „Beispiele > Zusätzliche Beispieldaten“ verwendet.



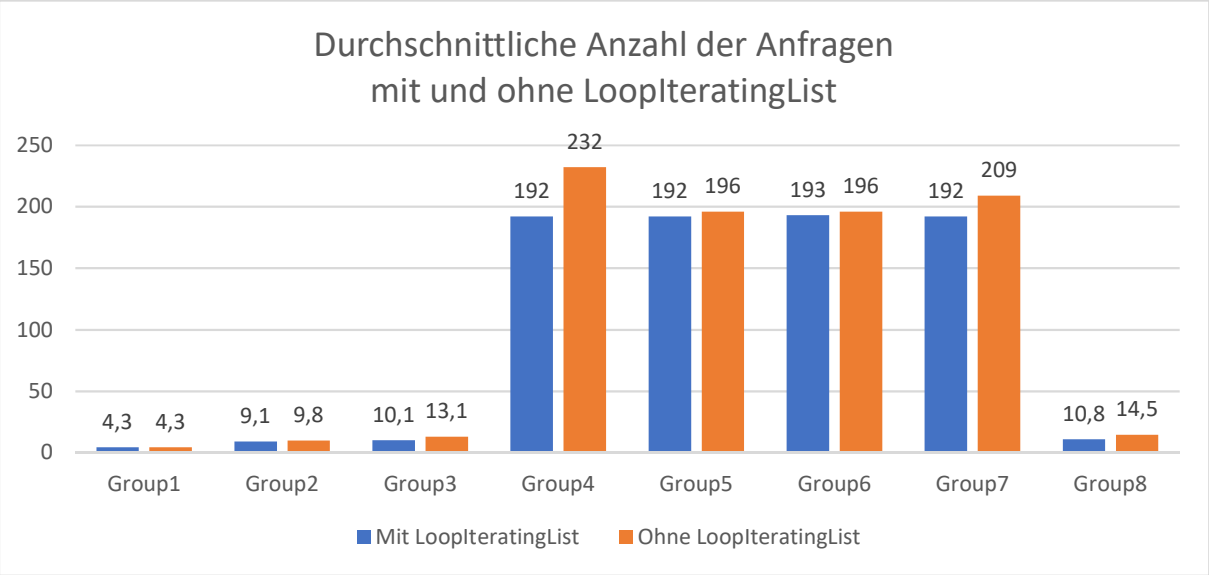
Die höheren Zahlen sind gerundet.

Ein wirklicher Unterschied ist nur in Group3 und Group8 zu beobachten; dort gibt es nämlich keinen Superstar. Unglücklicherweise tritt hier der Fall auf, dass einmal die erste und einmal die zweite Variante bessere Ergebnisse erzielt. Schaut man sich die Gruppen mal näher an, so gibt es in Group3 einen Schein-Superstar und in Group8 eine gleichmäßige Verteilung aller Verknüpfungen. In Group3 ist die erste Variante schneller, weil `is_not_following_anyone` schnell erkennt, dass „Edsger“ eben

doch „Jitse“ folgt, womit Schein-Superstars also schnell gefunden sind. In den Gruppen mit weniger Verknüpfungen ist eine Person schneller durch `is_followed_by_everyone` ausgeschlossen, weil es nicht so viele Personen gibt, die einer anderen folgen.
Grundsätzlich ist es also sinnvoll, beide Möglichkeiten einfach zugänglich zu machen; hier gibt es einen Parameter der Funktion `TeeniGroup.get_superstar` dafür. Dieser ist allerdings mit der ersten Variante vorbelegt, sodass `is_not_following_anyone` zuerst getestet wird, da ich Gruppen mit Schein-Superstars oder vielen Verknüpfungen für wahrscheinlicher halte. Superstars können schnell zu Schein-Superstars werden, und da offensichtlich viele Gruppen dazu tendieren, einen Superstar zu haben, gibt es wohl auch mehr Gruppen mit Schein-Superstars.

Auswirkungen der LoopIteratingList

Im Folgenden ist die Zahl der Anfragen mit und ohne `LoopIteratingList` geprüft worden, jeweils mit der Bedingung `is_not_followed_by_anyone` zuerst.



Die höheren Zahlen sind gerundet.

Bei manchen Gruppen wie Group5 und Group6 scheint die `LoopIteratingList` eine sehr geringe Auswirkung zu haben, bei manchen ist es aber sehr von Vorteil, die `LoopIteratingList` zu benutzen. Hauptsächlich wirkt sich die spezielle Liste ja auf die Non-Superstars (Optimierung 1) und die Hinweise auf mögliche Superstars (Optimierung 2) aus, weil mehr Personen in Bezug auf den möglichen Superstar, der gerade geprüft wird, gesetzt werden.
In Group7 beispielsweise ist es gut, auf diese Weise viele Personen zu testen, denn irgendwann ist auch mal „Folke“, der Superstar, dran, und dann wird er als nächstes als möglicher Superstar geprüft. Ohne `LoopIteratingList` würde das nicht so schnell passieren. Und in den Gruppen 5 und 6 gibt es einfach nicht so viele Personen auszuschließen, oft wird schon als zweite Person Folke getestet, wie in den Programmausgaben gut zu sehen ist (was man sich aber eigentlich auch denken kann). In Group4 folgt eine Person sehr vielen anderen, daher ist es sinnvoll, die `LoopiteratingList` einzusetzen, um vorzeitig Non-Superstars zu finden, da es generell viele Personen als Superstar zu testen gibt.

Beispiele

Zusätzliche Beispieldaten

Die Beispieldaten `superstar1.txt`, `superstar2.txt`, … werden in dieser Dokumentation mit Group1, Group2, … benannt. Um auch Sonderfälle abzudecken, existieren die Gruppen 5 bis 8; es handelt sich jeweils um Abwandlungen von `superstar3.txt` oder `superstar4.txt`, der Superstar bleibt auch gleich (also es gibt keinen bzw. „Folke“ ist der Superstar). Nur Group8 besitzt keinen Superstar und basiert auf den Mitgliedern von `superstar3.txt`. Die Bedeutung der Gruppen ist im Folgenden erläutert:

Name	Beschreibung	Schwierigkeit
Group5	Jeder (außer Folke) folgt nur einer Person, nämlich Superstar Folke	Ein Sonderfall mit nur den nötigen Verknüpfungen für einen Superstar
Group6	Die zufällige eine Hälfte folgt jeder anderen Person, die andere Hälfte folgt nur Folke (Folke folgt niemandem)	Ein ähnlicher Sonderfall wie in Group5, wobei aber doch recht viele Personen jemandem folgen und irritieren können
Group7	Jeder folgt Hanna UND Folke, einziges Indiz für Folke ist, dass Hanna Folke folgt, aber nicht umgekehrt	Es könnten viele Anfragen darauf verschwendet werden, einen falschen Superstar zu prüfen, solange, bis das einzige Indiz gefunden wird
Group8	Jeder folgt der Reihe nach einer anderen Person, so dass jeder einmal jemandem folgt und einmal von jemandem „gefolgt wird“	Es gibt – im Gegensatz zu Group3 – keine(n) Schein-Superstar(s), sodass dies einen möglichen entgegengesetzten Fall darstellt

Ein einfaches Beispiel

Dieses Beispiel soll die Funktionsweise und die Logik des Skripts weiter vertiefen, daher werden alle „Überlegungen“, die das Programm anstellt, aufgeführt, um den Vorgang innerhalb der Haupt-Funktion `TeeniGroup.get_superstar` möglichst transparent darzustellen. Die im folgenden genutzte Beispiel-Gruppe befindet sich übrigens in der Datei `beispiel1.txt`.

Die Beispiel-Gruppe besitzt 6 Mitglieder: Stefan, Marina, Klaus, Gertrud, Florian und Nils. Superstar ist Klaus. Die Verknüpfungen sehen so aus:

Jeder außer Klaus folgt Klaus
Stefan folgt Marina
Stefan folgt Gertrud
Marina folgt Stefan
Florian folgt Nils
Nils folgt Marina

Die Liste der Namen liegt in der oben genannten Reihenfolge vor. Es wird zuerst `is_not_following_anyone` geprüft (siehe „Umsetzung > Optimale Bedingungsreihenfolge beim Prüfen ermitteln“).

Im Moment enthält die Liste mit den noch nicht ausgeschlossenen Superstars alle in der oben genannten Reihenfolge.
Es wurde beim letzten Prüfen (das es noch nicht gibt) natürlich kein Hinweis auf einen möglichen

Superstar gefunden, also wird Nils, der letzte in den noch nicht ausgeschlossenen, geprüft und aus der Liste mit den möglichen Superstars entfernt.

Folgt Nils wirklich niemandem?

Der nächste in der Liste zum Testen von vorne ist Stefan (Optimierung 6)
FOLGT Nils Stefan? → NEIN

Stefan ist also kein Superstar, da Nils ihm nicht folgt (Optimierung 1)

Die nächste in der Liste ist Marina
FOLGT Nils Marina? → JA

Nils ist also kein Superstar, da er jemandem folgt.

Stefan wird aus den möglichen Superstars entfernt (Optimierung 1), die Liste sieht nun so aus: $[Marina, Klaus, Gertrud, Florian]$

Wie wäre es also mit Marina, da zumindest Nils ihr folgt? Marina wird, da sie jetzt getestet wird, aus der Liste der möglichen Superstars entfernt.

Folgt Marina wirklich niemandem?

Der nächste in der Liste ist Klaus
FOLGT Marina Klaus? → JA

Marina ist also kein Superstar.

Wie wäre es also mit Klaus? (Klaus wird aus der Liste der möglichen Superstars entfernt)

Folgt Klaus wirklich niemandem?

Die nächste in der Liste ist Gertrud
FOLGT Klaus Gertrud? → NEIN

Der nächste in der Liste ist Florian
FOLGT Klaus Florian? → NEIN

Der nächste in der Liste ist Nils
FOLGT Klaus Nils? → NEIN

Der nächste in der Liste ist Stefan
FOLGT Klaus Stefan? → NEIN

Die nächste in der Liste ist Marina
FOLGT Klaus Marina? → NEIN

Der nächste in der Liste ist Klaus.
Klaus kann sich nicht selbst folgen, er wird übersprungen

Alle Mitglieder wurden einmal getestet, Klaus folgt niemandem.
Also ist Klaus der Superstar, wenn ihm zusätzlich jeder folgt.

Folgt wirklich jeder Klaus?

Die nächste in der Liste ist Gertrud
FOLGT Gertrud Klaus? → JA

Der nächste in der Liste ist Florian
FOLGT Florian Klaus? → JA

Der nächste in der Liste ist Nils
FOLGT Nils Klaus? → JA

Der nächste in der Liste ist Stefan
FOLGT Stefan Klaus? → JA

Die nächste in der Liste ist Marina
FOLGT Marina Klaus? → JA (wurde aber schon getestet, wird also nicht doppelt gezählt, siehe Optimierung 4)

Der nächste in der Liste ist Klaus
Klaus kann sich nicht selbst folgen, er wird übersprungen

Alle Mitglieder der Gruppe folgen Klaus

Klaus ist der Superstar!

Beispiele der BwInf-Webseite

Die Beispieldaten werden nicht als Beispiele direkt genutzt, dies ist mit dem obigen Beispiel bereits passiert. Stattdessen wird auf die Besonderheiten der Gruppen eingegangen.

`superstar1.txt`

Hier ist Justin der Superstar. Diese sehr kleine Gruppe (3 Mitglieder) dient wohl zu Testzwecken am Anfang.

`superstar2.txt`

Hier ist Dijkstra (eine Anspielung auf Edsger W. Dijkstra, zumal auch die anderen Mitglieder bekannte Informatiker sind?) der Superstar. Besonders interessant sind in dieser Gruppe die vielen Verknüpfungen in einer kleinen Gruppe, die sozusagen als Vorbereitung auf Group4 dienen, die nach der folgenden Tabelle sehr viele Verknüpfungen besitzt.

Gruppe	Mitgliederanzahl	Verknüpfungsanzahl	Verknüpfungen pro Mitglied
Group1	3	3	1
Group2	5	10	2
Group3	8	8	1
Group4	80	3096	38,7

`superstar3.txt`

Diese Gruppe besitzt als einzige keinen Superstar, obwohl Edsger fast ein Superstar geworden wäre, wenn er nicht Jitse folgen würde. Dies ist insofern interessant, als dass man versuchen muss, trotz einem Schein-Superstar ein möglichst gutes Ergebnis zu erzielen.

`superstar4.txt`

Superstar ist Folke. Diese Gruppe besitzt immens viele Verknüpfungen (s.o.), was auf diese Lösung bezogen vor allem in Bezug auf Optimierung 2 schwierig ist, da es hier sehr viele „Hinweise“ auf Superstars aufgrund der vielen Verknüpfungen gibt. Andererseits gibt es dadurch auch viele Möglichkeiten, um eine Person als Superstar auszuschließen.

Beispiel des schlechtesten Falls

Dieses Beispiel wird möglichst einfach gehalten, um Übersichtlichkeit zu gewähren. Hier wird beschrieben, wie dieser schlechteste Fall zu Stande kommt und ein knapper Ablauf des Programms dargestellt.

Die Gruppe, von der im Folgenden ausgegangen wird, besitzt einen Superstar.
Um den Superstar zu finden, muss das Programm auf irgendeine Weise auf ihn aufmerksam werden und ihn dann im nächsten Durchgang prüfen. Dieses Auswählen kann durch zwei Dinge erreicht werden: Einmal kann ein Hinweis auf den Superstar beim vorherigen Prüfen eines vermeintlichen Superstars erlangt werden (Optimierung 2), einmal kann der Superstar der letzte in der Liste mit allen noch nicht ausgeschlossenen Superstars sein und wird deshalb als nächstes geprüft (Optimierung 6). Nach Optimierung 6 wird das letzte Mitglied der Liste mit den noch nicht ausgeschlossenen (also am Anfang allen) Personen als erstes geprüft. Daraus folgt, dass der Superstar im schlechtesten Fall auf gar keinen Fall hinten steht, da er dann als erstes geprüft werden würde. Um es weiter hinauszuzögern, sollte sich der echte Superstar im vorletzten Element der Liste befinden, da er hier auch vonseiten der `LoopIteratingList` so lange wie möglich nicht drankommt (die `LoopIteratingList` startet ja von vorne).
Der Superstar ist außerdem immer nur durch den Hinweis auf den möglichen Superstar an der Reihe, weil nur am Anfang das letzte Element der Liste mit den noch nicht ausgeschlossenen Mitgliedern genommen wird. Danach gibt es immer einen Hinweis. Deshalb ist der echte Superstar mit Prüfen dran, sobald die `LoopIteratingList` bei ihm angekommen ist. Bis es soweit ist, gibt es viele verschiedene Möglichkeiten, die im Prinzip beliebig sind und alle auf die gleiche Zahl an Anfragen hinauslaufen.
In der Beispiel-Gruppe befinden sich 4 Mitglieder mit den Namen von 1 bis 4. Der Superstar ist 1 und die Liste mit den möglichen Superstars und die `LoopIteratingList` sehen, wie oben beschrieben, so aus: $[4, 3, 1, 2]$. Alle Zahlen außer der eins können natürlich beliebig untereinander vertauscht werden. Im Folgenden ist die einfachste der Möglichkeiten aufgeführt, in der die `LoopIteratingList` schon bei der ersten Person (der 2, da dies das letzte Mitglied der Liste ist), den Superstar bereitstellt. Würde im folgenden Beispiel die 2 der 3 folgen, so würde erst noch die 3 geprüft. Da das nächste Element in der `LoopIteratingList` die 1 ist, würde statt „Folgt 2 1?“ „Folgt 3 1?“ geprüft, was keinen Unterschied macht, da man auch so mit der gleichen Anzahl an Anfragen auf die 1 kommt.

Das letzte Element ist die 2
Prüfen, ob 2 der Superstar ist…

Folgt 2 niemandem?

1) 2 folgt nicht 4
2) 2 folgt nicht 3
3) 2 folgt aber 1

Ist also 1 der Superstar? (denn zumindest 3 folgt der 1 ja)

Folgt 1 niemandem?

4) 1 folgt nicht 2
5) 1 folgt nicht 4
6) 1 folgt nicht 3

Folgt jeder der 1?

(1 überspringen, kann sich nicht selbst folgen)
(3) 2 folgt 1, bereits bekannt)
7) 4 folgt 1
8) 3 folgt 1

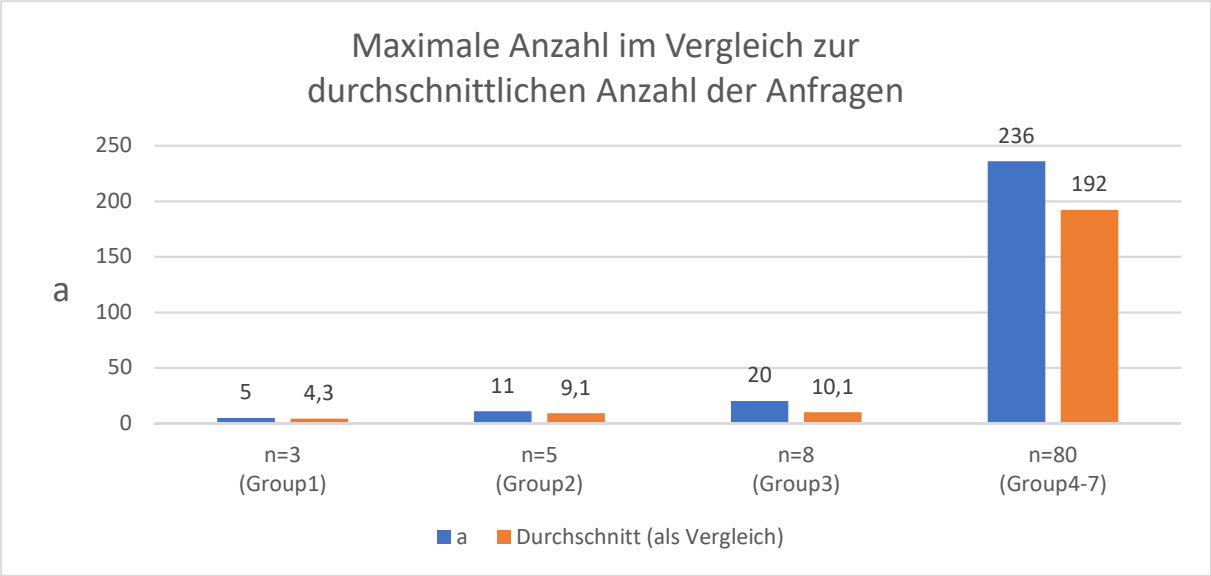
Abschätzen der maximalen Anzahl der Anfragen

n sei die Anzahl der Mitglieder einer Gruppe, a sei die maximale Anzahl an Anfragen.

Gruppen mit Superstar

Das Folgende bezieht sich auf das Beispiel für den schlechtesten Fall im vorherigen Abschnitt.
Da sich der Superstar im vorletzten Element der `LoopIteratingList` befindet, müssen $n-1$ Anfragen gestellt werden, bis der Superstar an die Reihe kommt. Das geschieht während des Prüfens eines anderen, falschen Superstars; als nächstes wird dann der echte Superstar geprüft. Für das Prüfen des echten Superstars sind $n-1$ Anfragen nötig, um sicherzustellen, dass der Superstar niemandem folgt (sich selbst kann er ja nicht folgen) und noch $n-2$ Anfragen, um sicherzustellen, dass jeder dem Superstar folgt. Es gibt eine Anfrage weniger, da ansonsten die Anfrage, mit der das Programm auf den Superstar kam, doppelt gezählt werden würde.
Durch Addition erhält man $a = (n-1) + (n-1) + (n-2) = 3n-4$ für $n > 1$, woraus $n \sim a$ (n ist proportional zu a) folgt.

Um dieses Ergebnis zu überprüfen, existiert die Methode `TeeniGroup.get_max` (siehe Quellcode), deren Ergebnisse mit den berechneten übereinstimmen.
Das Ergebnis ist im folgenden Diagramm dargestellt:



Bei allen Gruppen wird davon ausgegangen, dass es einen Superstar gibt, was in Group3 eigentlich nicht der Fall ist. Im Diagramm ist auch gut zu sehen, dass die tatsächliche durchschnittliche Anzahl an Anfragen prozentual keine starken Abweichungen aufweist. Dies ist wichtig, um zu wissen, dass in Gruppen gleicher Größe auch ungefähr gleiche Kosten zu erwarten sind.

Gruppen ohne Superstar

Gruppen ohne Superstar sind natürlich schwieriger abzuschätzen als Gruppen mit Superstar, weil es hier zum Beispiel kein festes Ziel gibt. Trotzdem habe ich bei ein paar Gruppengrößen versucht, einen möglichst schlechten Fall ohne Superstar zu generieren. Die Ergebnisse dieser Versuche stimmen mit der Formel für Gruppen mit Superstars überein und selbst wenn es nicht immer passen sollte, so ist es zumindest eine sehr gute Näherung (wobei zu beachten ist, dass Gruppen ohne Superstar sowieso die Ausnahme zu sein scheinen).

Quellcode

Bedeutung der Skripte

```
python3 aufgabe1-beispieldaten.py
```

Berechnet die Superstars der Beispieldaten der BwInf-Webseite und gibt die Superstars inklusive der mit `IS_FOLLOWING` gekennzeichneten Anfragen, wie in der Aufgabenstellung verlangt, aus.

```
python3 aufgbe1-cmd.py <group-path> [nodebug]
```

Berechnet den Superstar einer Gruppe in der Textdatei `<group-path>`. `nodebug` kann optional angehängt werden, um Debug-Ausgaben zu unterdrücken. Das Arbeitsverzeichnis wird automatisch auf das Verzeichnis des Skripts gesetzt, und weil die Beispieldaten im selben Ordner liegen, können sie über `superstar1.txt`, `superstar2.txt`, … direkt ausgewählt werden.

```
teenigroup.py
```

Modul, das die `TeeniGroup`-Klasse enthält. Siehe „Quellcode > class TeeniGroup“.

```
loop_list.py
```

Modul, das die `LoopIteratingList`-Klasse enthält. Siehe „Quellcode > class LoopIteratingList“.

class TeeniGroup

Diese Klasse aus dem Modul `Aufgabe1/teenigroup.py` stellt eine Gruppe des Netzwerks dar. Sie besitzt hauptsächlich die Funktion `TeeniGroup.get_superstar`, die nach dem unter „Lösungsidee“ beschriebenen Verfahren mit den zugehörigen Optimierungen arbeitet, um einen Superstar mit möglichst wenig Anfragen zu finden.

```
get_superstar(shuffle=True, first_condition="not_following",
              debug=False)
```

Versucht, einen Superstar in der Gruppe zu finden. Wenn `debug` `True` ist, werden zusätzliche Ausgaben (gestellte Anfragen etc.) gemacht. `first_condition` ist entweder `"not_following"` (zuerst `is_not_following_anyone`) oder `"followed"` (zuerst `is_followed_by_everyone`), was die erste Bedingung, die geprüft wird, bestimmt. Wenn `shuffle` `True` ist, wird die Liste mit Namen vorher gemischt (mit dem `random`-Modul)

Gibt ein `tuple` bestehend aus dem Namen des Superstars oder `None`, wenn es keinen gibt, und der Anzahl von gestellten Anfragen mithilfe der Funktion `TeeniGroup.get_request_amount` (siehe „Ergänzende Funktionen“) zurück.

Im Folgenden ist der Code der Funktion dargestellt, allerdings ohne Debug-Ausgaben bzw. die englischen Kommentare, wie sie in der originalen Datei vorhanden sind.

```
def get_superstar(self, shuffle=True, first_condition="not_following",
                        debug=False):
  # 'first_condition' und 'second_condition' sind die Funktionen, die später für
  # das Prüfen der Bedingungen benutzt werden. Die Reihenfolge wird hier gemäß
  # des Parameters angepasst (die Verwendung des selben Namens ist hier egal).
  if first_condition == "not_following":
      # Zuerst soll is_not_following_anyone geprüft werden
      first_condition = self.is_not_following_anyone
      second_condition = self.is_followed_by_everyone
  elif first_condition == "followed":
      # Zuerst soll is_followed_by_everyone geprüft werden
      first_condition = self.is_followed_by_everyone
      second_condition = self.is_not_following_anyone
  else:
      raise ValueError("Parameter 'first_condition' must be 'not_following' or "
                       "'followed'")

  # Der Zähler der Anfragen wird zurückgesetzt
  self.start_request_counting()

  names = list(self.names).copy()
  if shuffle:
      random.seed(time.time())
      # für unterschiedliche Ergebnisse wird die Liste mit Namen wahlweise gemischt
      # random.shufffle mischt eine übergebene list in-place.
      random.shuffle(names)
  # self.names wird von self.not_following_anyone und self.is_followed_by_everyone
  # genutzt. Zur LoopIteratingList siehe "Quellcode > LoopIteratingList" und
  # die dritte Optimierung unter "Lösungsidee".
  self.names = LoopIteratingList(names)
  # 'possible_superstars' wird von dieser Funktion genutzt, um alle noch nicht
  # ausgeschlossenen Superstars zu speichern
  possible_superstars = names.copy()
```

```
# speichert den Namen eines Mitglieds, wenn es beim letzten Prüfen einen Hinweis
# darauf gab, dass dieses Mitglied der Superstar sein könnte.
# Am Anfang gibt es das nicht, daher 'None'. Siehe "Optimierung 2"
next_possible_superstar = None

while len(possible_superstars) > 0:
    # Wenn beim letzten Testen ein Hinweis auf den Superstar gefunden wurde und
    # dieser noch nicht ausgeschlossen wurde, wird dieser als nächstes getestet.
    # possible_superstar speichert den Namen des Mitglieds, das im Moment getestet
    # wird
    if next_possible_superstar in possible_superstars:
        possible_superstar = next_possible_superstar
        possible_superstars.remove(next_possible_superstar)
    else:
        # ansonsten ein neues Mitglied aus der Liste nehmen und entfernen
        possible_superstar = possible_superstars.pop()

    # die erste Bedingung wird überprüft
    # first_condition_result ist ein bool (Bedingung erfüllt oder nicht)
    # next_possible_superstar: siehe Beschreibung oben, vor der while-Schleife
    # non_superstars: Liste aller Non-Superstars (Optimierung 1), die
    # ausgeschlossen wurden
    first_condition_result, next_possible_superstar, non_superstars =
                                        first_condition(possible_superstar, debug)

    # nur wenn die erste Bedingung wahr ist, muss auch die zweite geprüft werden
    if first_condition_result:
        # jetzt gibt es entweder einen Superstar oder es gibt keinen (Optimierung
        # 7), deshalb sind Hinweise unwichtig (mit "_")
        second_condition_result, _, _ =
                                        second_condition(possible_superstar, debug)

        if second_condition_result:
            # die zweite Bedingung ist auch wahr, also ist possible_superstar der
            # echte Superstar und er/sie wird zurückgegeben
            return possible_superstar, self.get_request_amount()
        else:
            # Die zweite Bedingung ist nicht erfüllt, also gibt es keinen Superstar
            return None, self.get_request_amount()
    # alle Non-Superstars werden entfernt, falls dies noch nötig sein sollte, weil
    # die erste Bedingung falsch war und so weiter getestet werden muss
    for no_superstar in non_superstars:
        if no_superstar in possible_superstars:
            possible_superstars.remove(no_superstar)
return None, self.get_request_amount()
```

`is_not_following_anyone(possible_superstar, debug=False)`

Überprüft, ob `possible_superstar` niemandem folgt. Wenn `debug` `True` ist, werden zusätzliche Ausgaben (gestellte Anfragen etc.) gemacht.

Gibt ein `tuple` zurück, bestehend aus:

- einem `bool`: `True`, wenn `possible_superstar` niemandem folgt, ansonsten `False`
- dem Namen einer anderen Person, die Superstar sein könnte, weil `possible_superstar` dieser folgt (natürlich nur, wenn die Bedingung nicht erfüllt wurde) (siehe „Lösungsidee > Optimierungen“).

- Einer Menge (`set`) von Non-Superstars, denen alle `possible_superstar` nicht folgt, also können sie keine Superstars sein (siehe „Lösungsidee > Optimierungen“).

Es wird durch alle Namen in `self.names` iteriert und für jeden dieser Namen sichergestellt, dass `possible_superstar` dieser Person nicht folgt. Sollte `possible_superstar` dieser Person doch folgen, könnte diese Person Superstar sein, da sie von jemandem gefolgt wird, und wird aus diesem Grund zusammen mit `False` und dem `set`, das die Non-Superstars enthält, zurückgegeben.

`is_followed_by_everyone(possible_superstar, debug=False)`

Überprüft, ob jeder `possible_superstar` folgt. Wenn `debug` `True` ist, werden zusätzliche Ausgaben (gestellte Anfragen etc.) gemacht.

Gibt ein `tuple` zurück, bestehend aus:

- Einem `bool`: `True`, wenn `possible_superstar` niemandem folgt, ansonsten `False`
- dem Namen einer anderen Person, die Superstar sein könnte, weil diese `possible_superstar` nicht folgt (natürlich nur, wenn die Bedingung nicht erfüllt wurde) (siehe „Lösungsidee > Optimierungen“).
- Einer Menge (`set`) von Non-Superstars, die alle `possible_superstar` folgen und somit keine Superstars sind (siehe „Lösungsidee > Optimierungen“).

Es wird durch alle Namen in `self.names` iteriert und für jeden Namen geprüft, dass diese Person `possible_superstar` folgt. Sollte eine der Person `possible_superstar` nicht folgen, wird `False` zurückgegeben, ansonsten `True`.

Ergänzende Funktionen

Im Folgenden sind einige wichtige Funktionen aufgeführt und sehr kurz erläutert, da sie nur teilweise mit der Lösung zusammenhängen.

`get_mean(count=10000)`	Gibt den Durchschnitt aus der Anzahl der Anfragen von `get_superstar`-Aufrufen (`count`-mal) zurück.
`get_max(count=10000)`	Gibt die größte benötigte Anzahl aus `count` Aufrufen von `get_superstar` zurück.
`__init__(names, followers)`	Initialisiert eine neue `TeeniGroup` mit den Mitgliedern in der `names` als `list` und den Personen, die anderen folgen im `dict` `followers`, für das gilt: „`key` folgt allen Personen in `value`“, wobei `value` eine `list` von Mitgliedernamen ist.
`@staticmethod from_textfile(path)`	Erstellt eine `TeeniGroup`, indem eine Textdatei unter `path` im auf der BwInf-Seite beschriebenen Format eingelesen wird.
`is_following( possible_follower, possible_followed_person, debug=False)`	Wenn `debug` `True` ist, werden zusätzliche Debug-Ausgaben gemacht. Gibt `True` zurück, wenn `possible_follwer` `possible_followed_person` folgt, ansonsten `False`. Außerdem wird der Zähler für die Anfragen um 1 erhöht, sofern diese Anfrage noch nicht gestellt wurde. Dies passiert in `TeeniGroup._is_following_raw` (siehe dort).
`@lru_cache(None) _is_following_raw( possible_follower, possible_follower_person)`	Wird von `TeeniGroup.is_following` aufgerufen und ist unbegrenzt gecacht. Diese Funktion existiert, um trotz Cache die in der Aufgabenstellung geforderte Ausgabe der Anfrage machen zu können. Dafür wird in `TeeniGroup.is_following` die Cache-Größe vor und nach dem

	Aufruf verglichen. Da diese Funktion nur einmal für bestimmte Parameter aufgerufen wird, wird hier der Zähler um 1 erhöht, sodass keine gleiche Anfrage doppelt gezählt wird (siehe „Lösungsidee > Optimierungen“).

`start_request_counting()`
 	
`get_request_amount()`
	Die erste Funktion setzt den Zähler für die Zahl der Anfragen zurück, die in `self._request_amount` gespeichert wird.
	Die zweite Funktion gibt die aktuelle Zahl der gesendeten Anfragen seit dem letzten Zurücksetzen zurück.

class LoopIteratingList

Diese Klasse aus dem Modul `Aufgabe1/loop_list.py` stellt eine besondere Liste dar, die von außen wie eine normale `list` behandelt wird, aber beim Iterieren immer da anfängt, wo das Iterieren beim letzten Mal aufgehört hat (z.B. durch eine Schleife, die mit `break` abgebrochen wurde, …). Dadurch ist es möglich, viele unterschiedliche Namen einer `TeeniGroup` zu testen, wie dies bei `TeeniGroup.is_not_following_anyone` und `TeeniGroup.is_followed_by_everyone` geschieht. So können noch mehr Personen als Superstars ausgeschlossen werden, weil jede/r mal drankommt.

Für die Funktionsweise werden die Methoden zum Iterieren – also `__next__` und `__iter__` überschrieben. Es existiert außerdem das Attribut `LoopIteratingList._next_position`, das den Index des Elements, das beim nächsten Mal zurückgegeben wird, speichert und bei jedem Aufruf von `__next__` um 1 erhöht bzw. wieder auf 1 gesetzt wird, sobald das Ende der Liste erreicht wurde. Wenn das Ende erreicht wurde, wird natürlich das Element mit Index 0 zurückgegeben, aber `_next_position` wird für das nächste Mal auf 1 gesetzt.
`__iter__` hat lediglich die Aufgabe, statt dem normalen built-in Iterieren die `__next__`-Methode sooft aufzurufen wie die Liste lang ist und die Ergebnisse mit `yield` zurückzugeben.