Aufgabe 1: Superstar

Team-ID: 00922

Team-Name: Zweiundvierzig

Bearbeiter/-innen dieser Aufgabe:
Franz Miltz

25. November 2018

Inhaltsverzeichnis

1 Lösungsidee

Das Problem besteht darin einen Algorithmus zu finden, für den der Average Case bei stochastisch gleichverteilten Eingaben optimal ist. In anderen Worten: Würde man alle möglichen Anordnungen der Eingaben gleichzeitig betrachten, so wäre der Preis minimal.

Um das Problem zu modellieren wird offensichtlich ein Graph benötigt, bei dem die Knoten die Personen/Accounts sind und die Kanten beschreiben, wer wem folgt. Ein Graph ist ein Tupel aus einer endlichen Menge von Knoten und einer endlichen Menge Kanten. Eine Kante ist ein geordnetes Paar von Knoten (der Graph ist offensichtlich gerichtet). In unserem Fall kann eine Kante nur existieren oder eben nicht. Eine Markierung ist nicht notwendig. Außerdem kann es maximal von jedem Knoten zu jedem anderen Knoten genau eine Kante geben.

$$V = \{v_1, v_2, .., v_n\} \tag{1}$$
$$|V| =: N \tag{2}$$
$$E = \{(V_a, V_b), (V_c, V_d), ...\} \text{ mit } a, b, c, d \in [1, N] \subset \mathbb{N} \tag{3}$$
$$|E| \leq N \cdot (N - 1) \tag{4}$$

Das Ziel ist es nun so schnell wie möglich alle Knoten als Superstars auszuschließen. Man muss also versuchen, aus jeder Anfrage so viel wie möglich Information zu erhalten. Die folgenden Überlegungen sind essenziell:

1. Wenn Person A Person B folgt, können wir Person A als Superstar ausschließen.
2. Wenn Person A Person B nicht folgt, können wir Person B als Superstar ausschließen.

Wenn wir uns nun zwei Mengen, Kandidaten K und Nicht-Kandidaten $\bar{K}$, zur Hilfe nehmen, gilt Folgendes:

$$K \cup \bar{K} = V \tag{5}$$
$$K \cap \bar{K} = \emptyset \tag{6}$$
$$(v_1, v_2) \in E \Rightarrow v_1 \in \bar{K} \tag{7}$$
$$(v_1, v_2) \notin E \Rightarrow v_2 \in \bar{K} \tag{8}$$

Wenn man also ein Paar (v_1, v_2) betrachtet, wobei beide Knoten noch potentielle Superstars sind, kann man genau einen der beiden Knoten ausschließen, wenn man weiß, ob solch eine Kante existiert. Somit sollten erst möglichst viele Kandidaten über diesen Weg ausgeschlossen werden.

Sobald nurnoch ein Kandidat übrig ist, müssen alle Beziehungen dieses einzelnen Knotens mit allen anderen Knoten überprüft werden. Dabei ist es vollkommen egal, ob zunächst geprüft wird, ob alle Knoten diesem Knoten folgen, oder, ob diese Knoten keinem anderen Knoten folgt.

Mit diesem Ansatz wird nun sichergestellt, dass die optimalen Anfragen unter Berücksichtigung des zur Verfügung stehenden Wissens gestellt werden.

2 Umsetzung

Die Umsetzung erfolgte in Python 3 und der einzige `import` war `sys`, um auf die Attribute beim Ausführen in der Command-Line zuzugreifen. Die Sprache eignet sich insofern gut, als dass der Code relativ kurz ist und bereits Mengen (`set`) implementiert wurden, was angesichts der Lösungsidee sehr Vorteilhaft ist.

Das Programm kann genutzt werden, indem es mit dem Python3-Interpreter und einer Beispieldatei als Argument aufgerufen wird. Die Datei kann auf Linux-Systemen direkt ausgeführt werden. Der Output erfolgt durch den Output-Stream, d.h. direkt in die Konsole.

Es ist natürlich unmöglich stets die optimale Lösung zu finden, da es dem Programm nicht erlaubt ist auf die dafür notwendigen Informationen Rücksicht zu nehmen. Auf Grund der verwendeten Mengen und deren Interation ist das Programm **nicht deterministisch**. Das wird sich vor allem in den Beispielen äußern.

2.1 Graph

Zunächst wurde ein Graph implementiert. Dieser hat folgende Attribute:

- `n`: `int`, Anzahl der Knoten
- `v`: `set` mit Knoten als Elemente
- `e`: `set` mit Beziehungen `(v1, v2)` als Einträge
- `price`: `int`, Anzahl der Anfragen bzw. Preis in Euro

Hinzu kommen die Funktionen:

- `__init__(self, path)`: Konstruktor, setzt Attribute basierend auf Eingabedatei
- `request(self, node1, node2)`:
  Gibt `True` zurück, wenn es die Kante `(node1, node2)` gibt; inkrementiert den Preis

Diese Implementation ist nahezu minimal, aber vollkommen ausreichend für das Problem. Der eigentliche Algorithmus benötigt nur diesen Graphen, um Anfragen zu stellen.

2.2 Hauptteil

Zur Lösung des Problems, wurde eine Klasse mit folgenden Attributen erstellt:

- `g`: Der Graph
- `links`: `set`, die bereits abgefragten Kanten
- `candidates`: `set`, zu Beginn enthält es alle Elemente aus `g.v` (K)
- `not_candidates`: `set`, zu Beginn leer ($\bar{K}$)

Auffällig ist hier, dass die Kanten auch hier als `set` verwaltet werden, obwohl somit der tatsächliche Wert dieser Kanten verloren geht. Das ist möglich, weil unmittelbar nach jeder Anfrage an den Graphen bereits entschieden werden kann, welche Konsequenzen das Ergebnis hat, und dieser Wert später überflüssig wird.

Das Programm wird durch eine `run`-Funktion ausgeführt. Diese enthält eine Schleife, die immer wieder `request_best` aufruft, bis entweder jener Aufruf wahr zurückgibt oder keine Kandidaten mehr übrig sind. Weiterhin interessant ist die `request`-Funktion. Sie ist dazu da Anfragen an den Graphen zu stellen und das Erebnis auszuwerten. Dabei wird zuerst die Anfrage gestellt und wenn entweder `node1` ein Kandidat ist und jemandem folgt oder `node2` ein Kandidat ist, dem von `node1` nicht gefolgt wird, wird der jeweilige Knoten aus K entfernt und zu $\bar{K}$ hinzugefügt.

Nun ist es die Aufgabe der `request_best`-Funktion einen Funktionsaufruf von `request` durchzuführen, der möglichst optimal ist. Dabei wird zunächst eine Möglichkeit gesucht eine Anfrage zwischen zwei verschiedenen Kandidaten durchzuführen. Sollte das möglich sein, so gibt die Funktion `False` zurück. Wenn nicht, wird mit dem zweiten Teil, der `check_last`-Funktion fortgefahren. Hier wird für alle Knoten in $\bar{K}$ geprüft, ob eine zu dem letzten Kandidaten noch nicht geprüft wurde. Auch hier wird `False` zurückgegeben, wenn solch eine Anfrage möglich ist. Sollte auch nach diesem Muster keine mögliche Anfrage gefunden werden, kann wahr zurückgegeben werden, denn der letzte Knoten ist nun mit Sicherheit ein Superstar, denn er wurde weder bei den Anfragen, ob der Knoten jemandem folgt, noch bei den Anfragen, ob jemand dem Knoten folgt, aus der Menge der Kandidaten entfernt.

2.3 Ergebnis

Mit diesem Algorithmus liegt der optimale Fall bei $2(N-1)$ Anfragen/Euro, insofern ein Superstar exisitiert. Denn dann sind alle Anfragen, die Kandidaten ausschließen bereits Teil der unbedingt notwendigen Anfragen, die einen Superstar verifizieren. Sollte kein Superstar exisitieren, liegt der Best-Case natürlich bei N, da jede Kante genau einen Knoten ausschließen kann. Beide Fälle sind sehr unwahrscheinlich.

Der Worst-Case mit und ohne Superstar ist $3(N-1)$, da zunächst alle Knoten bis auf einen Kandidaten ausgeschlossen werden können und im Anschluss alle Verfikationskanten ($= 2(N-1)$) gerpüft werden. Die letzte Anfrage legt dann fest, ob der letzte Kandidat ein Superstar ist oder nicht.

3 Beispiele

In diesem Programm wurde kein Benchmark durchgeführt, was die benötigte Zeit berücksichtigt, da jene vernachlässigbar gering ist.

Weiterhin werde ich nur bei den Beispielen 1, 2 und 3 die Folge von Anfragen angeben, da es ansosnten zu viele werden. Es ist klar, dass beim erneuten Ausführen mit hoher Wahrscheinlichkeit ein anderer Preis zustande kommt. Somit sollten Sie sich nicht wundern, wenn sie die Ergebnisse nicht bzw. nicht sofort reproduzieren können. Leider ist es außerdem unmöglich für alle Eingaben alle Möglichkeiten auszuprobieren, da die Zahl jener Möglichkeiten mit $N!$ berechnet wird und somit schneller als exponentiell wächst.

3.1 Beispiel 1 (BwInf)

Befehl: `./superstar.py beispieldaten/superstar1.txt`
Ausgabe:

```
Selena -> Justin
Justin -> Hailey
Justin -> Selena
```

```
Hailey -> Justin

Total cost: EUR 4
Superstar found: Justin
```

Die ersten fünf Zeilen sind die durchgeführten Anfragen. „`Selena -> Justin`“ entspricht also „`Folgt Selena Justin?`“. In den letzten beiden Zeilen befinden sich der Preis für diese Untersuchung sowie das Ergebnis. In diesem Fall wurden vier Anfragen gestellt, es kostet also 4 Euro. Das Ergebnis ist, dass Justin der Superstar ist.

3.2 Beispiel 2 (BwInf)

Befehl: `./superstar.py beispieldaten/superstar2.txt`
Ausgabe:

```
Dijkstra -> Codd
Dijkstra -> Turing
Dijkstra -> Hoare
Dijkstra -> Knuth
Codd -> Dijkstra
Knuth -> Dijkstra
Hoare -> Dijkstra
Turing -> Dijkstra

Total cost: EUR 8
Superstar found: Dijkstra
```

Diese Ausgabe scheint optimal zu sein, da alle Anfragen den Superstar beinhalten.

3.3 Beispiel 3 (BwInf)

Befehl: `./superstar.py beispieldaten/superstar3.txt`
Ausgabe:

```
Sjoukje -> Rinus
Sjoukje -> Edsger
Edsger -> Jitse
Jitse -> Rineke
Jitse -> Peter
Jitse -> Pia
Jitse -> Jorrit
Sjoukje -> Jitse

Total cost: EUR 8
There is no superstar in this graph!
```

3.4 Beispiel 4 (BwInf)

Befehl: `./superstar.py beispieldaten/superstar4.txt`
Ausgabe:

```
Clara -> Oliver
Clara -> Sixten
...
Oscar -> Folke
Jack -> Folke

Total cost: EUR 160
Superstar found: Folke
```

Diese Ausgabe ist ebenfalls relativ gut, Preise von 220+ Euro sind bei diesem Graphen nicht abwegig.

3.5 Beispiel 5 (Zweiundvierzig)

Da das Programm nicht deterministisch arbeitet, wird es unmöglich einen Spezialfall zu finden, bei dem entweder immer die optimale Lösung gefunden wird, oder die Ausgabe immer dem Worst-Case entspricht. Somit kann sich an dieser Stelle kein Beispiel befinden, das eine solche Eigenheit das Programms aufzeigt.

4 Quellcode

Sämtliche Inputs und Outputs sowie Kommentare wurden aus dem Code entfernt, können aber in `superstar.py` gefunden werden.

4.1 Graph

```
class graph:
  def __init__(self, path):
      with open(path) as f:
          lines = f.readlines()
          self.v = set(lines[0].rstrip('\n').split('␣'))
          self.n = len(self.v)
          self.e = set()
          for l in [x.rstrip('\n').split('␣') for x in lines[1:]]:
              if len(l) == 2:
                  self.e.add((l[0], l[1]))
          self.price = 0
  def request(self, node1, node2):
      self.price += 1
      return (node1, node2) in self.e
```

4.2 Star-Finder

```
class star_finder:

    def __init__(self, g):
        self.g = g
        self.links = set()
        self.candidates = g.v.copy()
        self.not_candidates = set()

    def request(self, node1, node2):
        r = self.g.request(node1, node2)
        self.links.add((node1, node2))
        if r and node1 in self.candidates:
            self.candidates.remove(node1)
            self.not_candidates.add(node1)
        elif not r and node2 in self.candidates:
            self.candidates.remove(node2)
            self.not_candidates.add(node2)

    def run(self):
        while len(self.candidates) > 0:
            if self.request_best():
                break

    def request_best(self):
        if len(self.candidates) > 1:
            for i in self.candidates:
                for j in self.candidates:
                    if j != i:
                        self.request(i, j)
                        return False
        return self.check_last()

    def check_last(self):
        for c in self.candidates:
            for n in self.not_candidates:
                if n != c:
```

```
                    if (n,c) not in self.links:
38                      self.request(n, c)
                        return False
40                  elif (c, n) not in self.links:
                        self.request(c, n)
42                      return False
        return True
```

4.3 Hauptmethode

```
1 def superstar():
      path = sys.argv[1]
3     g = graph(path)
      sf = star_finder(g)
5     sf.run()

7 if __name__ == '__main__':
      superstar()
```