Algorithmen und Datenstrukturen II – Baby Namen

Von Lorenz Pfeifer, Hubertus Knobling, Jonathan Kirst

Zuerst trafen wir uns virtuell, um eine Aufgabenteilung zu besprechen. Dabei wurde uns klar, dass man die Aufgaben bei diesem Thema eher weniger gut aufteilen kann und so entschlossen wir uns, dass zunächst für 2 Tage jeder selbst versucht die Aufgabe zu lösen und wir nach dieser Zeit unsere Ergebnisse besprechen. Bei der ersten Besprechung fiel uns direkt auf, dass wir unerwarteterweise komplett andere Lösungswege hatten. Dabei hatte Jonathan zum Beispiel schon einen sehr performanten Algorithmus entwickelt, während Lorenz eine erweiterte Prüfung der ähnlichen Namen entwickelt hatte. Auch lösten wir die Aufgabe in verschiedenen Programmiersprachen (Java und PHP), wobei sich herausstellte, dass Java wohl überaschenderweise für uns in diesem Fall performanter ist. Wir entschieden uns daher, die Java Applikation weiterzuentwickeln und fügten somit unsere Ergebnisse zusammen.

Zum Levenshtein Algorithmus waren wir uns sehr schnell einig, dass wir diesen allein nicht ausreichend finden, um die Aufgabe sinnvoll zu lösen. Allerdings kann man die Aufgabe mit diesem sehr viel einfacher lösen. Daher nutzten wir diesen und implementierten noch zusätzliche Bedingungen, die beim Vergleichen zweier Namen zugreifen müssen:

- Wenn ein Name im anderen enthalten ist, kann man beide Name nahezu immer zusammenfassen (John, Johnathan, Johnny ,Johnie). Bei diesem Beispiel wäre Levenshtein allein ziemlich ungenau, wenn man auf eine Distanz von z.B. 2 bauen würde, da z.B. Johnathan zu John eine Distanz von 5 hat.
- Ein Name ist grundsätzlich nur dann zu einem anderen Namen gleich, wenn die ersten Buchstaben die gleichen sind. Uns fiel kein Beispiel ein, bei welchem Namen dies nicht der Fall ist.
- Zunächst hatten wir eine feste Anzahl der Levenshtein Distanz angegeben, da uns anschließend aber auffiel, dass dies nicht immer optimal ist, vor allem bei langen Worten entschieden wir uns, dies wie folgt zu ändern: Die bisherigen Fehler der Levenshtein DistanzMatrix müssen kleiner gleich der Anzahl der zu vergleichenden Buchstaben/5 sein.
- Um Duplikate zu vermeiden, wurde ein Boolean-Array geführt und bereits benutzte Namen wurden in diesem markiert.

Zur Optimierung der Laufzeit gibt es einige Abfragen, die es erlauben größere Codeabschnitte zu überspringen:

> Versetzte Laufvariablen (j=0; k=j+1)
> !alreadyUsedNames[j]
> wordIsContained

Da eine Sortierung (ORDER BY Occurences DESC) notwendig ist, wurde ein Comparator geschrieben und einem Treeset übergeben, welches somit die Strings der ähnlichen Namen in der passenden Reihenfolge abspeichert.

Als Levenshtein Distanz implementierten wir zwei verschiedene Funktionen, die ein wenig anders funktionieren. Wir nutzten danach die, die performanter bzw. schneller ist. Dies maßen wir mit System.nanoTime(). Wir haben dennoch beide in der App gelassen, sodass man diese besser vergleichen kann.