Chair for Algorithms
and Data Structures
Prof. Dr. Hannah Bast
Claudius Korzen

Information Retrieval
WS 2017/2018

http://ad-wiki.informatik.uni-freiburg.de/teaching

UNI FREIBURG

Exercise Sheet 5

Submit until Tuesday, November 28 at **12:00pm (noon)**

This sheet is again about performance, so Python is not an option; see ES3 for more explanations. The code from the lecture for building a q-gram index from the entities in a given file is provided on the Wiki (in both Java and C++). Note that in the lecture, we padded the words on both sides, whereas for the exercise below you need padding on one side only.

Exercise 1

Write a program for fuzzy entity search with the following functionality:

1. Implement a method *buildFromFile* that builds a q-gram index from the entity names in a given entity file. The file format is as follows: one line per entity with tab-separated columns, with the entity id in the first column, the entity name in the second column, a popularity score in the third column, a short description of the entity in the fourth column, and additional information in the remaining columns; the first line contains a header with a short description of each column. Each id links to the related Wikidata entity (they are not suitable for using in the inverted lists; instead use continuous integer ids). The scores are needed for ranking; see item 5. The description are just for the output; see item 6. The use of the additional information is optional; see item 7. You can adapt the code from the lecture, see above.

Note: before computing the q-grams, normalize each string by lowercasing and removing all non-word characters (including whitespace) as shown in the lecture. Don't forget to later normalize the queries in the same way, see item 6 below.

2. Implement a method *mergeLists* that merges the inverted lists for a given set of q-grams. Pay attention to either keep duplicates in the result list or keep a count of the number of each id.

Note: it is ok, if you do this merging by simply concatenating the lists and then sort the concatenation. That is, you do not have to make use of the fact, that the lists are already sorted.

3. Implement a method *prefixEditDistance* that for two given strings x and y and a given integer δ, returns $\text{PED}(x, y)$ if it is $\leq \delta$, and $\delta + 1$ otherwise.

Note: the method must run in time $O(|x| \cdot (|x| + \delta))$, as explained in the lecture.

[$$t, $tu, tur, urn, rno, nov, ove, ver]

4. Implement a method *findMatches* that for a given string x, finds all entities y with $\text{PED}(x, y) \leq \delta$ for a given integer δ. First use the q-gram index to exclude entities y that do not have a sufficient number of q-grams in common with x, as explained in the lecture. Only for the remaining candidate entities should the PED be computed with the method from item 3. The method should record the number of these PED computations and the correctness of that number should be verified for the given test cases. The method should return: all y with $\text{PED}(x, y) \leq \delta$, and for each such y the value of $\text{PED}(x, y)$ and the score of y (from the file read by the method from item 1).

5. Implement a method *rankMatches* for ranking a given set of entities y, each with a score s and a PED value p. The entities should be sorted by (p, s). That is, all entities with PED $= 0$ should come before all entities with PED $= 1$, etc. And the entities with the same PED should be sorted by score (higher score first).

6. Implement a *main* function that builds a 3-gram index from a given file (given as a command-line argument), and then, in an infinite loop, lets the user type a query shows the top-5 matches according to the ranking from item 5, and using the method from item 4. Take $\delta = \lfloor |x|/4 \rfloor$, where x is the **normalized** query (see item 1). For each match, output the original entity name (not the normalized name) and its description. Optionally, also output the Wikdata and Wikipedia URL of the entity (which can easily be constructed from the additional information). If there are more than 5 matches, also print the total number of matches. Also print the time it took to process the query.

Note: you do not need a unit test for the main function, provided that most of the actual functionality is in the methods above (all of which should of course have a unit test).

7. Optionally, also use the synonym information that is provided as part of the input file (see item 1), namely in the following way: When the query matches one of the synonyms, count this as a match for the original entity name. If an entity name gets several matches in this way (for its original name or one of its synonyms), rank it by the best of the corresponding scores. The output should be just as for item 6, except when the position in the ranking is due to a synonym match: in that case, additionally output the corresponding synonym (not all synonyms, because that can be quite many for popular entities).

Note: "Optionally" means that you don't have to do this to get full points. It's relatively easy though and fun and improves the result quality considerably.

8. Add a row to the result table on the Wiki following the examples already given there.

As usual, you must implement the test cases provided in the TIP file on the Wiki and make sure that everything runs though on Jenkins without errors. You must also use the methods and names as described above (otherwise, it quickly becomes a nightmare for the tutors to correct your code and provide meaningful feedback). If you want to deviate from the given structure, please ask on the forum.

Add your code to a new sub-directory *sheet-05* of your folder in the course SVN, and commit it. Make sure that *compile*, *test*, and *checkstyle* run through without errors on Jenkins. And of course, commit the usual *experiences.txt*.