Algorithms and datastructures
Hashing

April 24, 2020

1 Introduction

You are responsible for teaching a programming course. Unfortunately, due to recent events, all activities at the university have to be replaced with remote alternatives. For your course, the students have to hand in the solutions for their programming assignments electronically. You suspect that some students might have plagiarized their code and would like to detect this automatically.

In this session, we will develop a simple system to detect this and will apply it on real student programming solutions. We will start with a naive implementation and improve this step by step to make it more efficient making clever use of hash functions.

You are free to restructure the code or to change method declarations if needed.

2 Representing source code as sets

The start code contains a main.cpp file with a *readDatafiles* method. This method expects a path to a folder as input and returns a std::vector<std::set<std::string>> with for each document in that folder a set of the unique words in that document. We also add bigrams (two sequential words). Note that this is a very simple way of transforming documents into sets as information about the frequency or order of the words is lost. We simply split the text based on whitespace even though this is probably not the best idea for source code.

3 Calculating the similarity between sets

To find similar solutions, we need to find similar sets. One way to calculate the similarity between sets is the Jaccard index. The Jaccard index is defined as the size of the intersection divided by the size of the union of the sample sets A and B.

$$J(A,B) = J(B,A) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

- Implement a method that calculates the Jaccard index for every pair of documents. The existing method works on a std::vector<std::set<T>>. This way we can use it for std::vector<std::set<std::string>> now and std::vector<std::set<**int**>> later. It returns a std::vector<std::priority_queue<std::pair<**double**, **int**>>> with for each document a priority queue that allows us to retrieve the most similar documents to that document. The priority queue consists of pairs (similarity score, other document). Pairs with a high similarity score will be returned first.
- Implement the showSummary method. This shows for each document which other documents are similar. The number between the brackets is the similarity score. Each document only needs to be shown once.

```
doc     Similar documents
------------------------------
11      109 (1)
23      261 (1)
28      158 (0.854545)
35      208 (1) 188 (1) 79 (1)
```

4 Optimization 1: replacing string comparisons by integer comparisons.

To calculate the Jaccard index, we need to compare two sets of strings. This is an expensive operation as string comparisons are expensive and each set can be large. Integers are much easier to compare than strings.

- Implement the replaceWithUniqueId method that replaces each unique string with a unique integer.
- How much faster is the jaccard method now ?

5 Optimization 2: replacing word IDs with hashcodes

Replacing each word with a unique id can be seen as a perfect hash function as there are no collisions in the mapping. The disadvantage is that we need some kind of datastructure to keep track of the already assigned IDs with the corresponding words. This might result in a large overhead if we have a very large number of different words.

- Implement the replaceWithHash method. This performs the same task as the replaceWithUniqueId method but now uses a hash function to do the mapping from string to integer. The hash-function is provided to the method as a template parameter, allowing you to easily try out different hash functions. You can find some hashfunctions in hashfunctions.h.
- The disadvantage of our hash approach is that we might have collisions. In this context, a collision means that two different words will be mapped to the same integer. The Jaccard index for two documents might then be higher, wrongly flagging two documents as plagiarized.

- Implement the findCollisions method that investigates how bad the collision problem is for our data and different hash functions. The hash function is again provided as a template parameter. A possible output of this method could be:

```
Collision found: "lege lijst" and "= lijst" both map to 1768584052
Collision found: "met een" and "is een" both map to 543516014
Collision found: "ploeg" and "if ploeg" both map to 1819239783
...
```

- Implement different hash functions to find one that works well for our data. You can find some inspiration online: http://www.cse.yorku.ca/~oz/hash.html or https://en.wikipedia.org/wiki/Jenkins_hash_function.

6 Optimization 3: Locality sensitive hashing

We replaced the expensive string comparisons with more efficient integer comparisons, either by replacing each word with a unique id or by replacing each word with its hashcode. Calculating the intersection and union of two very large sets is still an expensive operation even if the sets contain integers. Can we calculate the Jaccard index more efficiently ? As it turns out, we can again use hashing to do this.

Locality-sensitive hashing (LSH) is an algorithmic technique that hashes similar input items into the same "buckets" with high probability. In our case this means that we can create a hash of each document and that similar documents will result in similar hash codes. Note that this is the opposite from what we typically want of a hash function. If we use a hash function for a hash table for example, we try to hash similar keys to completely different hash codes. Also for cryptographic hash functions that are used to validate the authenticity of documents, you typically want that small changes result in very different hash codes.

Minhash is a locality-sensitive hashing technique that can be used to quickly **estimate** how similar two sets are. It is commonly used in search engines to find duplicate web pages.

Minhash uses k different hash functions. For each hash function and each document, it finds the element in the document that results in the lowest hash code. Each document is then represented by these k elements. We then check how many of these k minhashes match (y) and use y/k as an estimate for the Jaccard index.

Example

word	hash 1	hash 2
a	8	2
b	23	8
c	3	9
d	5	12
e	51	6
f	1	21

doc1={a,c,d,e} is mapped to {c, a} because "c" results in the lowest hash code for hash 1 and "a" results in the lowest hash code for hash 2.

doc2={a,b,d,f} is mapped to {f, a} because "f" results in the lowest hash code for hash 1 and "a" results in the lowest hash code for hash 2.

The Jaccard index is estimated to be 0.5. ($c \neq f, a = a$).

The more hash functions you use, the more accurate the estimation will be but of course the more expensive it is.

Our hash functions are defined as $h(x) = (ax + b)\%c$. Where the coefficients a and b are randomly chosen integers less than the maximum value of x. c is a prime number slightly bigger than the maximum value of x. Choosing k different hash functions thus involves choosing k random integers for a and b. You can find a list of possible prime numbers for c here: http://compoasso.free.fr/primelistweb/page/prime/liste_online_en.php. x is the element of the set and is here assumed to be an integer (the result of section 4 or 5).

- Implement the Minhashing approximation to find the similarities between sets.
- Experiment with different k values.
- How much faster is our Minhashing approach compared to our Jaccard method ?