# Edgar Khachatryan's solution documentation

Since my goal was to find a way to quickly search for a phrase in a lot of documents, I realized that preprocessing files and storing them in some kind of data structure is necessary, since storing documents' content in it gives us the possibility to improve the performance of searching at the expense of memory usage. So I came to the conclusion of using the data structure "dictionary", since searching for an item in it is very efficient. This was the key idea in my research.

At first, my algorithm was to search in a dictionary for the first token in the given phrase, then read the line from the text document in every match, tokenize and check the next tokens. At first sight, this was a good idea, since it's significantly improved performance in many test cases. But this method has a huge disadvantage: if we search for a phrase where the first token is a stop-word, the algorithm's performance becomes very poor, since it's tokenizing every single line where this word appears. It could be a case where the first word is "the", which appears on almost every line of each document! So, I started to think about a way to improve it somehow. My next idea was to search for a token which appears least in documents and check for its neighbours. For example, if we are searching for a phrase "is a good idea", where "is" appears like 130.000 and "idea" appears 1.000 times, we could tokenize 1.000 lines, not 130.000, which will improve algorithm efficiency.

But is it the most optimal way of using a dictionary here? Of course not! I did some research and found a data structure called "inverted index". This gave me a new idea. What if we store a dictionary's values in a set, instead of a list and intersect these results? This can lead us to a lot fewer lines to tokenize and check, which will significantly improve the efficiency of the algorithm. Now about the same example of the phrase "is a good idea". Intersection of these sets can lead us to lines, where all these 4 tokens appear together. So, instead of reading and tokenizing 1.000 lines, like in the above version of the algorithm, it will read and tokenize about 10 lines! Of course, this method has a disadvantage: every single token being searched can be a stop-word. But I haven't found more optimal solutions in the given time. Some technical improvements can be made to my algorithm to make it faster (for example, passing a line from a text document to 4 different functions to tokenize it. I could write 1 instead of these 4 manually.), but I preferred to focus on other aspects of this task.

Tokenizing files and storing them on a data structure also helped in solving other subtasks, like lowercase/uppercase characters, punctuation. Also, tokenizing a line from a document helped in solving the second task, where there is a missing middle word in the searched phrase.

This method solved all the subtasks (of course, with some drawbacks), except for the subtasks "Modifying word endings, using synonyms". To solve the word ending problem, I decided to use stemming in tokenization logic. I also have been thinking about using lemmatization, but this method was significantly slower, because there is a lot of text. I did some research to find a way to solve the synonyms' problem. I found a method called "word2vec" which will help in solving it, but I couldn't find a way to integrate this into the algorithm above, so I decided not to do it and so, the synonyms' problem hasn't been solved.

There are some test cases in my .ipynb, which show advantages and disadvantages of my algorithm. For testing, I haven't used some certain techniques. I tested the algorithm manually with some different test cases.