COMPUTATIONAL BIOLOGY

Vol. 00 no. 00 2015
Pages 1–3

The "Famous" Protein

Or why the road to fame is easier for some

Naessens Tom [1,*]

[1]Department of Engineering, 2nd Master of Computer Science: Software Engineering

Received on 27/02/2015

ABSTRACT

Motivation: Fame is something people always have wanted to obtain. Some argue that becoming famous has something to do with talent, some think luck is the key and others think it has something to do with great mentorship and connections. This research tries to find another explanation in the basic building blocks of men: our DNA.

Results: We have found proof that one protein may attribute to the likeliness of someone to become famous: having a great name can make the road a lot easier. By searching for a list of 500 famous people's name, we have found a high correlation between fame and names encoded in the protein ENSP00000341887.

Contact: Tom.Naessens@UGent.be

1 INTRODUCTION

What do Daniel Craig, Kanye West, Hitler, Henry VIII and Darth Vader have in common? Indeed, they are famous. But there is more: a recent study has indicated that their names, among many other famous people's names, are all coded in one protein sequence taken from the human genome.

In 1990, scientists all around the world started working on one single goal to determine the complete sequence of what we are made of: our DNA. This project, the Human Genome Project, was concluded in April 2003. Long story short, DNA is transcribed into RNA, which is then translated into proteins. One protein is made up of long chains of amino acids, which can be represented by a 20 letter alphabet. For example, a short protein sequence from the before mentioned genome is GYLSFALSHDQWMGDDDAYLCIHEDQTVYIQPSH.

On April 7 1998, (Drosnin) released a book called "The Bible Code" which suggested that the Bible contains secret messages which predict the future. At first sight, the strand of letters of human proteins seems to be quite random, but when applying the same algorithm Drosnin applied to find secret messages in the Bible, one can easily find a lot of hidden messages encoded in these proteins as well. By repeating the exact same algorithm on all the proteins which have been annotated onto the human genome, we have found a protein with a very peculiar property which just cannot be a coincidence.

In the following sections, we first look into a detailed description of the methodology used to find these hidden messages. We then dive a bit deeper into the algorithmic techniques used to be able to process the information rapidly. Afterwards, we discuss remarkable results we have found and draw our conclusions in the last section of this paper. Readers only interested in the results may skip section 2 and 3 and proceed directly to the results in section 4.

Table 1. Example variations of protein sequence GSUELAF

Protein sequence	Start index	Step size
GSUELAF	1	1
GULF	0	2
SEA	1	2
GEF	0	3
SL	1	3
UA	2	3
GL	0	4
SA	1	4
UF	2	4
E	3	4
GA	0	5
SF	1	5
U	2	5
E	3	5
L	4	5

2 METHODOLOGY

We are searching for hidden messages using Drosnin's approach in the collection of proteins found in the human genome. To obtain the list of searchable proteins, we use the latest release of the human genome assembly, that is GRCh38, which consists of 99,436 proteins and 37,341,228 amino acids. All amino acids were represented by their corresponding one-letter characters, without making any extra assumptions.

If there is a secret message in those proteins, it has to be hidden. So we can conclude that the message or words will not occur literally in the protein sequences. That would just be too easy and coincidental. We therefore create a list of each single protein of all possible sequences by iterating over all the different step sizes with their corresponding start indexes. For example, if we have the following protein sequence: GSUELAF, we can list the variations of this protein in Table 1 on page 1.

To find any kind of hidden message in this list of processed protein sequenced, we first tried a dictionary approach with the English language. Therefore, we retrieved a list of English words from (3), consisting of 109,582 words. This word list does not only contain stems, but also includes inflected forms such as plural nouns and conjugated verbs to minimize the amount of filtering.

Now we have retrieved the necessary data, we check which words exist in which proteins by searching for a substring match of the

Table 2. Example result of "sea", "ale", "gulf" and "sue" in GSUELAF

Dictionary word	Start index	Step size
SEA	1	2
GULF	0	2
SUE	2	1
ALE	6	-1

word in all the variations of the protein and tried to find patterns of words which occurred together in a single protein. For example, our dictionary might contain "sea", "ale", "gulf" and "sue". For "sea" and "gulf", we have an exact match in the table listed on Table 1 on page 1. For "sue", we have an exact substring match in the protein itself. For "ale" however, we have no match at all.

Words can however also be coded backwards! Therefore, we simply reverse the words and try to find a substring match the same way as in the previous paragraph. "Sea", "gulf" and "sue" become "aes", "flug" and "eus", but these don't match anything. "Ale" however becomes "ela" which has a match in the original protein. We can then conclude "ale" is also included in the protein, but is coded backwards.

Now we have the words with their corresponding matching protein sequences (the full protein or the variation on the protein) and the start index and step size used to create the variation on the protein sequence, the last thing we have to do is to find the exact starting index in the original protein. This can be done with some simple mathematics. The substring search returns the occurrence of the first character, let's call this `substring_startindex`. For non reversed words, we can multiply this by the step size and add the start index of the according protein variation sequence, resulting in the start index in the original protein sequence. To find the start index of the reversed words, we have add the step size of the corresponding protein variation sequence multiplied by length of the word minus one. The step size is then also inverted.

So, for the example protein GSUELAF and the example dictionary of "sea", "ale", "gulf" and "sue", we obtain the result listed in Table 2 on page 2.

3 IMPLEMENTATION

3.1 Obstacles

The methodology described in the previous section can be implemented easily and naively for small dictionaries and proteins consisting of little amino acids, but for bigger datasets, we need a more robust, fast and memory-efficient solution. After running some basic tests without any optimizations, it became clear that there were two components which needed to be optimized. The first component was an efficient storage for the variations on a single protein. The longest protein has nearly 35991 amino acids, which results, when not limiting the step size to a certain number, into 647622055 variations of the same protein. These all need to be stored efficiently. The second component is that these large amounts of variations should be easily searchable for substrings, to find out if a variation actually contains a word from the dictionary.

Luckily, there is a solution which solves both problems at the same time. A generalized suffix tree is a variation of the standard suffix tree which builds a suffix tree for a set of strings. It takes $\Theta(n)$ time to build this tree and can be searched for all occurrences of a single string in the tree in asymptotically $\Theta(m + z)$, where m is the length of the combined strings searched for, and z the amount of occurrences in the generalized suffix tree.

3.2 Specific implementation

To avoid redoing work that has already been done, the SuffixTree implementation from (1) was used.

A summary of the specific implementation goes as follows: We read the data from two files: a cleaned up version of the GRCh38 file where the fasta headers have been reduced to a single ID and the dictionary of words we're searching for. We then iterate over the proteins, building a generalised suffix tree for all the variations of one protein. When this generalised suffix tree is built, we iterate over our dictionary and use our suffix tree to search rapidly for substrings. When one or more substrings are found, we calculate the corresponding start index in the original protein and print the result.

To create the variations of each word, the following Python code is used:

```
def proteinsplitter(word):
  yield (1, 1, word)

  for stepsize in range(2, len(word)-1):
    for startindex in range(0, stepsize):
      yield (startindex, stepsize,
        word[startindex:len(word):stepsize])
```

3.3 Optimisations

We can optimize both the search of dictionary words in the suffix tree, as the addition to the suffix tree of proteins by sorting both the proteins and the words in the dictionary. This ensures that both in the substring search of words and the addition of protein variations into the suffix tree we don't need to search or add from the root of the tree, but can skip some steps along the way.

All words with a wordlength smaller than three were filtered out of the dictionary, as these produced a lot of hits where nothing could be concluded from. The four biggest proteins (proteins longer than 30,000 amino acids) were also filtered due to memory constraints.

4 BENCHMARKING

The application was run on the University of Ghent (2) supercluster. For one total run, the list of proteins was split up into 400 batches, which were scheduled on the cluster. Due to data limitations, the words were only searched in one way and were not reversed. As the cluster was not entirely free, the walltime needed to finish these 400 jobs was one hour and four minutes. No job however took longer than twelve minutes, so given a free system, the walltime would be around twelve minutes.

5 RESULTS

A single-direction (without negative steps) run of the general wordlist (109,582 words) against all proteins resulted into 14 Gigabytes of data, 613,032,310 matches, equal to 15,982 unique words. This sheer amount of data was then aggregated per protein, per word and by word length to gather some statistics. A lot of data could be gathered from these results. Some highlights:

- ENSP00000387052 has the highest unique words per protein for its protein length, resulting in a factor of 2.57143 words per amino acid;
- ENSP00000352608 has the highest amount of unique words found in the protein, being 4,799 for a protein length of 5,038;
- ENSP00000368755 has the highest amount of words found in its protein sequence, being is 771,998 words for a length of 5,478 amino acids;
- The longest word found was "CENTRALEST", found in ENSP00000401204;
- "ELL" was found the most, being 5,406,823 times across all proteins.

But there was something highly peculiar. A lot of names popped up. Names from famous people. And, even more peculiar, they all came from one protein. As the general wordlist did not contain a lot of names, a new list was compiled out of 1,000 names (both surnames and given names) of 500 famous people. The algorithm was run, this time both normally and reversed, onto all proteins, searching only for these names of famous people.

Protein ENSP00000341887 yielded more than 131 unique names of famous stars. In table 3 on page 3, a table can be found of some examples of names of famous stars we have found in this protein. More examples (without indexes) are: "Carl Sagan", "Spider Man", "Henry VIII", "Walt Disney", "Will Smith", Princess "Diana", "Vin Diesel", "Elvis", "Steve Carell", and more.

6 CONCLUSION

By using a very simple algorithm, we can, in not even half an hour of time, find a single protein which seems to have an function more than thought before: the function for individuals to become famous (or notorious) when ones name is encoded in this protein.

Table 3. Famous people appearing in protein ENSP00000341887

Dictionary word	Start index	Step size
KANYE	6735	-1032
KANYE	978	768
KANYE	4678	828
+ 2 more		
WEST	1709	4
WEST	4888	12
+ 157 more		
HITLER	5318	-1000
HITLER	5265	-385
DANIEL	2844	189
CRAIG	6369	61
DARTH	818	438
VADER	970	-76
VADER	726	-15
VADER	5200	113
VADER	7067	127
VADER	1448	144
+ 14 more		
MERYL	7393	-491
MERYL	8152	-330
STREEP	6506	-308
STREEP	5928	-127
STREEP	1410	-11
STREEP	2180	546

REFERENCES

[1]Donner,J. (2013). Suffixtree.
[Drosnin]*a Drosnin,M.*The Bible Code. *Touchstone.*
[2]HPC (2014). Hpc ugent.
[3]International,S. (2015). Linguistics in sil.