Oleg Šelajev, A51443
shelajev@gmail.com

Text Algorithms MTAT.03.190 - Home Exam

1. (15p) Describe the bit-wise parallel matching of regular expressions with errors (indels, substitutions). Write a maximum 2 page "extended" abstract about this topic.

A regular expression is a representation of regular language and thus has an equivalent NFA (nondeterministic finite automata) [1]. In order to find one we can employ Glushkov algorithm, which will provide us with an automata which has up-to *n^2* states and no epsilon-transitions.
An exact matching of pattern *P* on text *T*, given *G(P)* is a glushkov automata using one possible bit-wise parallel matching algorithm goes in the following fashion.
Let *S* denote a subset of states in the automata and *t(S)* - a set of states that are all possible children of state in *S*. For efficiency reasons and as we only need 1 bit of information per state to mark it as possible or not, we can use a bitstring of the length equal to the number of states in automata. If the automata is larger than a word on a current architecture, we can use an array of words, so t is space efficient and set operation on it are very fast since they can be implemented as binary operations on computer words: union = |, intersection = &, negation = ~, difference = ^.
Now due to construction of Glushkov automata, for every node all incoming edges are labeled with the same character. Meaning that we can efficiently precompute bit-vectors *B[a]* - which for every character a in the alphabet denotes a set of states that are reachable with an *a* labelled edge.
Now we set S to have only initial set and start reading text characters. For every character a new possible set of states is an intersection of current states' children and sets that are reachable by the read character edge, so we can it very efficiently:

S = t[S] & B[a], where *a* is a character read.

If *S* becomes 0, pattern does not match and if we reached the end of the text and *S* contains final states, then pattern matches the text.

Now we need to incorporate errors into this matching algorithm. Fortunately, it is quite straightforward to do. First of all we need to modify our automata: if we agree to have up-to *k* errors, we duplicate a automata *k+1* times and instead of having one set of states *S*, we have a list of sets *S, S1, S2, … SK*, where *Si* is set of reachable states with up-to *i* errors.
Then we process text as usually character by character and update possible states according to the following relations, assuming *a* is a character read:

- *S' = t[S] & B[a]*, no errors update
- *S'i = t[Si] & B[a] | S(i-1) | t[S'(i -1)] | t[S(i -1)],* and we repeat this for all possible *i*-s.

Let's dissect the last assignment: *t[Si] & B[a]* - is a no error update, possibly we have encountered an error before; *S(i-1) i*s responsible for deletion errors, *t[S'(i -1)]* - for insertions (we take all children of sets with up-to *i -1* errors); *t[S(i -1)]* - substitutions (delete current char and take all children, as we can substitute to any char).

Now if any of *Si* contains final states after all the text is processed, we have a match. A natural extension to this approximate pattern-match algorithm would be to add "error-free regions" where cannot have any errors. This can be added straightforwardly by modifying "*S(i-1) | t[S'(i -1)] | t[S(i -1)]*" part of update process. However that is out of the scope of this question.

References for an interested reader:

Gonzalo Navarro , Mathieu Raffinot, Fast and flexible string matching by combining bit-parallelism and suffix automata, Journal of Experimental Algorithmics (JEA), 5, p.4-es, 2000 (approximate search chapter)

2. (15p) Text indexing using BWT transform

1. Decode the following Burrows-Wheeler encoded sentence (a question).
yeseessy_rrrhhhlnittw_d__nnoguheeeiisee__si_?tt

Answer: 'where_is_there_dignity_unless_there_is_honesty?' © Cicero

2. Describe the decoding process.
To inverse a BTW knowing the last character in the message, in our case '?' we need to follow these steps:

1. Create an empty table for all rotations of the original string.
2. Until we fill the table insert transformed text as 0-th column and sort the rows of the table.
3. Output the row of the table that ends with the last character in the message.

Python code with which I reversed the message above is:

```
>>> a = 'yeseessy_rrrhhhlnittw_d__nnoguheeeiisee__si_?tt'
>>> from operator import add
>>> def inverse_bwt(m, last_char):
...   table = [''] * len(m)
...   for _ in m:
...     table = sorted(map(add, m, table))
...   return [r for r in table if r[-1]==last_char][0]
...
>>>
>>> print inverse_bwt(a, '?')
where_is_there_dignity_unless_there_is_honesty?
```

3. Describe a BWT based text index that allows to simulate suffix trees over the BWT encoded text. I.e. – how the BWT encoded text allows to make a compressed text index.

Be concise – brief and strict. Also pay attention to the illustration of the main concepts. Try to make at least one illustration/figure to describe this index.

First of all suffix trees can be easily simulated by suffix arrays[1]. The operation might not be very trivial, but it is quite straightforward. Now we need to show that there exist a method to efficiently

simulate suffix array operations given a BWT encoded text. Apparently there is a data structure for that. It is called FM-index for “Full text index in a Minute space” [2].
Basically FM-index consists of BWT encoding of text T and some additional information to power three operations, with which it can simulate suffix array:

- count(P) - performs a substring match of string P on the original text T.
- locate(P) - finds an index i where pattern P occurs in the original text T.
- extract(pos, length) - returns a substring of length length, starting from position pos in text T. Note that this operation is not necessary to simulate suffix array, but a trivial implementation is to decode full BWT and pick needed substring from a full text T, so it’s doable.

Additional information needed include an array C, where C[c] = sum([1 for a in T if a < c]) - how many times characters lexicographically less than c occur in text T; and a table O[c, k] = sum ([1 for a in BWT(T)[1 .. k] if a == c]), how many times character c occurs in a prefix of length L of the BWT encoded text. Additionally a mapping of indexes in BWT encoded text to the indexes of those characters in the original T.
Now we can show how operations count and locate are implemented.

Note that BWT transformation of a text is basically a sorted table of suffixes of the text. Thus all suffixes that start with a pattern P sit in a continuous range in this sorted table. To return the count we work the pattern char-by-char right to left and keep a range of BWT which corresponds to text that starts with a part of the pattern processed.

Check out this pythonic pseudocode for more details:

```
count(P, bwt, C, O) # where P is pattern, and bwt is BWT encoding of a text
  left, right = C[P[-1]] + 1, C[P[-1] + 1] + 1 # P[-1] + 1 is next char.
  if left >= right:
    return 0 # exit early
  for c in reversed(P[:-1]): # -1 as we processed rightmost char already
    left, right = C[c] + O[c, left -1] + 1, C[c] + O[c, right]
    if left >= right:
      return 0
  return right - left + 1
```

Figure 1. Pseudocode description of count operation, strings are 1 indexed.

We return as soon as we don’t have any positions where text T has a suffix that starts with a part of pattern P we processed so far. And if we processed the whole pattern, we just return the length of the range, because every index of this range corresponds to a position in the original text T, where suffix of T starts with the given pattern.
Now to implement *Locate* operation we just need to show that we can find index i in the original text that corresponds to an index j in BWT transform. We can store this information with the BWT transformation (which requires a linear additional space, but makes *locate* trivial).

References:

[1] Udi Manber and Gene Myers. 1990. Suffix arrays: a new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms* (SODA '90). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 319-327.
[2] Paolo Ferragina and Giovanni Manzini. 2005. Indexing compressed text. *J. ACM* 52, 4 (July 2005), 552-581. DOI=10.1145/1082036.1082039 http://doi.acm.org/10.1145/1082036.1082039

3. (10p) Practical assignment

Download the Exam_texts_corpus.tgz linked from course homepage. http://courses.cs.ut.ee/2010/text/uploads/Main/Exam_texts_corpus.tgz These are three books of jokes from Project Gutenberg.

NB! All files referenced here can be found on github or from the archive submitted.

a. Extract individual jokes, each into a separate file numbered 001.txt .. nnn.txt . Use regular expressions (perl, python, grep, awk, etc tools ...) for text parsing, provide the parser. (this does not have to be perfect, satisfy with respectable quality). Describe how many jokes. (i.e. extract all intro's, etc irrelevant parts)

All code and output files can be found in the github repo: https://github.com/shelajev/ut-text-algorithms-exam

First of all I renamed files, can be found in 'etc' dir and changed newlines to unix ones: \n.
pg12444.txt -> 1.txt
pg21084.txt -> 2.txt
pg29419.txt -> 3.txt

Book files are quite inconsistent, they don't follow a single format, thus I had troubles with parsing them. Especially weird is book 2, so I don't know how good the parsing was. The python code that parses books into individual files can be found here: http://pastebin.com/HVFi1Wi1
Here I'll just outline how it works, check out file https://github.com/shelajev/ut-text-algorithms-exam/blob/master/src/jokes.py for details:

1. manually mark first line and last lines that present any interest:
    a. start_line = ['PREFACE', 'INTRODUCTION', 'BUDGET OF FUN', 'TOASTER\'S HANDBOOK', 'JOKES FOR ALL OCCASIONS']
    b. end_line = ['INDEX', 'End of the Project Gutenberg']
2. read book files between those lines.
3. run regexp on the read lines:
    a. 1txt => **r'^[A-Z][A-Z]*\n\n.*?\n\n\n\n'**
        i. Joke name, two newlines, whatever joke text, four newlines.
    b. 2.txt => **r'[*]*\n\n.*?\n\n '**
        i. only 2 newlines, because of the books format.
        ii. * * * separators are ignored, because it's not consistent with other books :)
    c. 3.txt => **r'^[A-Z][A-Z \.]*\n\n.*?\n\n\n'**
        i. joke names end with a period in this book.
4. Whatever regexp matches is a joke, write it to a file.

5. Increment joke index.

Code also prints jokes into individual files and outputs number of jokes in books:
done with book 1, there were 684 jokes
bounds = (0, 684)
done with book 2, there were 668 jokes
bounds = (684, 1352)
done with book 3, there were 620 jokes
bounds = (1352, 1972)

Totalling to 1972 jokes.

b. Count word frequencies, extract lists of words sorted by frequency (all jokes; each book separately) and words "most specific" to each book. Provide top-25 lists. If you have a general text – pick some book or set of books from Gutenberg – then you can also identify jokes specific word list.

Top 25 words with frequencies in all 3 books:

```
the:16031
a:8575
to:6471
of:5986
and:5669
i:4718
in:4000
he:3951
you:3886
was:3341
his:2785
that:2713
it:2686
for:1967
s:1959
is:1842
said:1689
on:1621
t:1601
with:1543
an:1505
at:1461
as:1372
one:1330
but:1308
```

Top-25 lists for books are: https://github.com/shelajev/ut-text-algorithms-exam/blob/master/etc/book1.dict
https://github.com/shelajev/ut-text-algorithms-exam/blob/master/etc/book2.dict
https://github.com/shelajev/ut-text-algorithms-exam/blob/master/etc/book3.dict
Also you can find them in a table below:

the:7916 a:4374 to:3291 of:2987 and:2804 i:2295 in:2038 he:1929 you:1921 was:1636 that:1352 it:1297 his:1296 s:1147 is:1023 said:1010 for:980 t:821 one:789 on:768 an:748 with:744 at:692 man:666 but:654	the:4391 a:1881 to:1564 and:1370 of:1330 i:1308 you:1105 he:1047 in:1000 was:917 it:735 that:710 his:698 s:514 t:511 for:453 on:422 with:407 is:398 an:398 at:397 she:370 had:363 but:355 as:354	the:3724 a:2320 of:1669 to:1616 and:1495 i:1115 he:975 in:962 you:860 his:791 was:788 it:654 that:651 for:534 said:439 on:431 is:421 as:401 with:392 at:372 an:359 him:357 had:334 my:299 but:299

A general text to compare: Alice Adventures in Wonderland, available here: http://www.gutenberg.org/files/11/11.txt

I took 100 most frequent joke specific words and checked which of them are not in top 150 most frequent words in a general piece of text. Results is:

replied boy good old sir wife am young asked mr new yes has day man

We can see that it's mostly personalisations of joke characters: man, wife, mr, sir, boy; dialog verbs: asked, replied; or qualities: new, good, young. It makes sense that those words are much more frequent in jokes, which are usually short stories about someone or some item as opposed to a general text which is usually a narrative story about some actions during a longer timeline.

Code for analysis is available here: https://github.com/shelajev/ut-text-algorithms-exam/blob/master/src/freqs.py

c. Propose a way to group jokes by topic or content similarity. E.g. search for a list of words and then group by occurrences of the same (rare) words or semantically related topic. Pick an interesting joke and fetch most similar jokes.

We already know which words are common for jokes and which are not. So to determine if 2 jokes are similar we take all words that are in both of them and each of them contributes to a similarity rating a value 1.0 / frequency_of_this_word_across_all_three_books.
Thus words that present in one joke, but not another don't contribute anything (jokes are different at those words). And common words as 'a', 'the' contribute just a bit. However words that are rare (common frequency is low) contribute much more.

The code can be found here: https://github.com/shelajev/ut-text-algorithms-exam/blob/master/src/similarity.py

A sample output of similarity fetch is here: https://github.com/shelajev/ut-text-algorithms-exam/blob/master/etc/sample_similarity_output.txt

And it finds some random looking indexes of jokes similar to a given one:

```
>>> check_joke(9)
Checking joke 9, top similar jokes:
[(1783, 0.5148430114409129), (173, 0.5124272542151035), (255, 0.507759724914451),
(1831, 0.41399150011546115)]
…
[here would be jokes texts, but for brevity they are omitted, check out
sample_similarity_output.txt file to see them]
```