Indexing Variable Length Substrings for Approximate Matching

Gonzalo Navarro and Leena Salmela

November 13th 2008

Outline

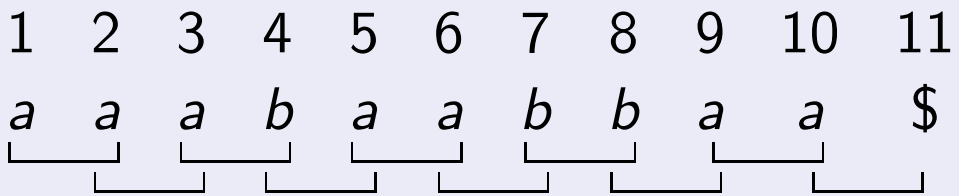
Outline

q-Gram Index

- For each q-gram that occurs in the text, store a list of all positions where the q-gram occurs.

Example

The 2-grams of the text *aaabaabbaa* are:



Thus the 2-gram index for the text is:

2-gram	positions
a\$	10
aa	1, 2, 5, 9
ab	3, 6
ba	4, 8
bb	7

Approximate Searching on the q-Gram Index

Given a pattern $P = p_1 \dots p_m$, find all substrings of the text that match the pattern with at most k differences (insertions, deletions, or mismatches).

- Take $k+1$ nonoverlapping q-grams of the pattern. For example:

$$\underbrace{p_1 \dots p_q} \quad \underbrace{p_{q+1} \dots p_{2q}} \quad \cdots \quad \underbrace{p_{kq+1} \dots p_{(k+1)q}}$$

- One of the q-grams must occur exactly in the text substring if the substring matches the pattern with at most k differences.
- Search for the q-grams in the q-gram index and verify around the positions where the q-grams occur.

Query Time Optimization

All q-grams are not equally frequent. We can speed up the search by choosing the least frequent non-overlapping q-grams of the pattern.

- For each substring of the pattern, calculate the number of verifications it triggers.
  - length of the substring $< q \implies$ find all q-grams starting with the substring
  - length of the substring $> q \implies$ use the prefix of the substring
- Use dynamic programming to figure out the optimal partition of the pattern into $k + 1$ pieces.
- Search for the pieces using the q-gram index and verify around the occurrences of the pieces.

Storing the q-Gram Index

- If we store the position lists as 32-bit integers, the space used by the index is approximately 4 times the text length in bytes.
- Storing differences instead of absolute values and encoding these with variable length integer coding saves space.

Example

2-gram index for the text *aaabaabbaa*

2-gram	positions	difference coded positions
a\$	10	10
aa	1, 2, 5, 9	1, 1, 3, 4
ab	3, 6	3, 3
ba	4, 8	4, 4
bb	7	7

Outline

Substring Index

- Choose the indexed substrings so that every position is indexed and the maximum frequency of the indexed substrings is minimized.
- If the index contains two substrings a and ax, a indexes only those positions where a is not followed by x.
- Storage as in q-gram index: Code position lists with differences and variable length integer coding.

Example

Substring index for text *aaabaabbaa*:

substring	positions
a	3, 6, 10
aa	1, 9
aab	2, 5
ba	4, 8
bb	7

Approximate Searching on the Substring Index

Searching for a pattern $P = p_1, \ldots, p_m$ with at most k differences:

- Search the index for the longest prefix of the pattern and verify the positions.
- If the longest prefix is $p_1, \ldots, p_i$ and the index contains a substring $p_1, \ldots, p_i, x$ for any x skip the next character of the pattern.
- Continue by searching the index for the longest prefix of the remaining pattern until $k + 1$ pieces of the pattern have been searched for.

Searching on the Substring Index: Example

Given the substring index for *aaabaabbaa*:

substring	positions
a	3, 6, 10
aa	1, 9
aab	2, 5
ba	4, 8
bb	7

Searching for *abaab* with 1 difference:

- The longest prefix of *abaab* found in the index is *a*, so we verify around positions 3, 6 and 10.
- The index contains the substring *aa* so we have to skip the second character.
- The longest prefix of *aab* found in the index is *aab* so we verify around positions 2 and 5.
- The index does not contain other substrings starting with *aab* so we do not need to skip a character. (Which is good as there are no characters left!)

Query Time Optimization

To be able to handle shorter patterns and to speed up the search for longer patterns, we can apply the same kind of optimization at query time as in the q-gram index.

- For each substring of the pattern, calculate the number of verification it triggers.
  - Find the longest prefix that is indexed in the substring index
  - If the whole pattern piece is indexed, include also those substrings that start with the pattern piece.
- Use dynamic programming to figure out the optimal partition of the pattern into $k + 1$ pieces.
- Search for the pieces in the substring index and verify around the occurrences of the pieces.

How to Construct the Substring Index?

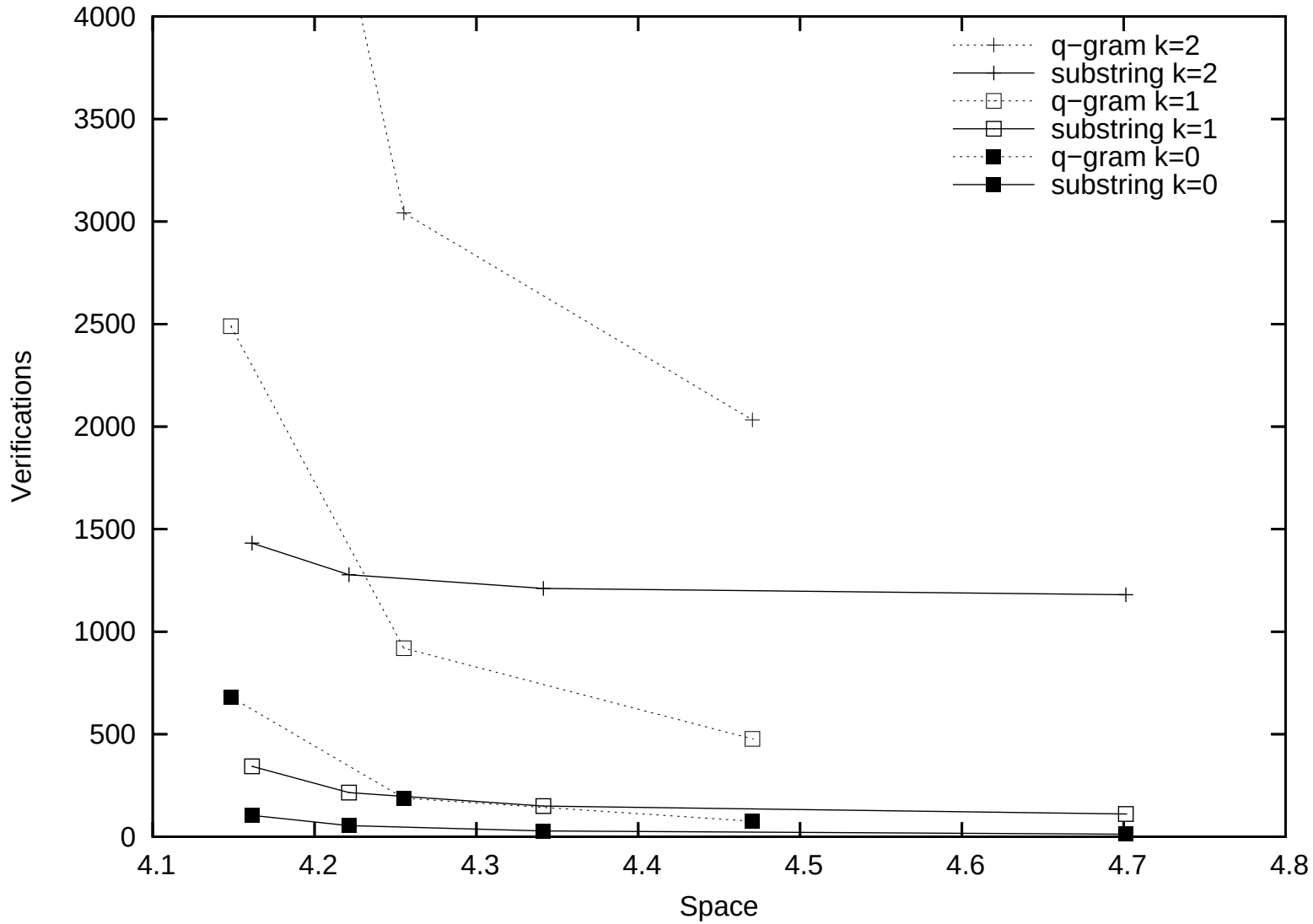
The current (brute force) implementation:

- Initialize the index by adding the empty string with a position list containing all the positions.
- Until a predefined number of substrings is indexed:
  - Take the string with the longest position list
  - Extend the string with the character c that most frequently follows it in the text and add this to the index.
  - Split the position list of the original string into two lists: One for those positions where the string is followed by c and the other for the rest of the positions.

Employing a suffix tree of the text should be more efficient.
(This is similar to generating fixed length codes for compression as presented by S.T. Klein and D. Shapira (SPIRE 2008).)

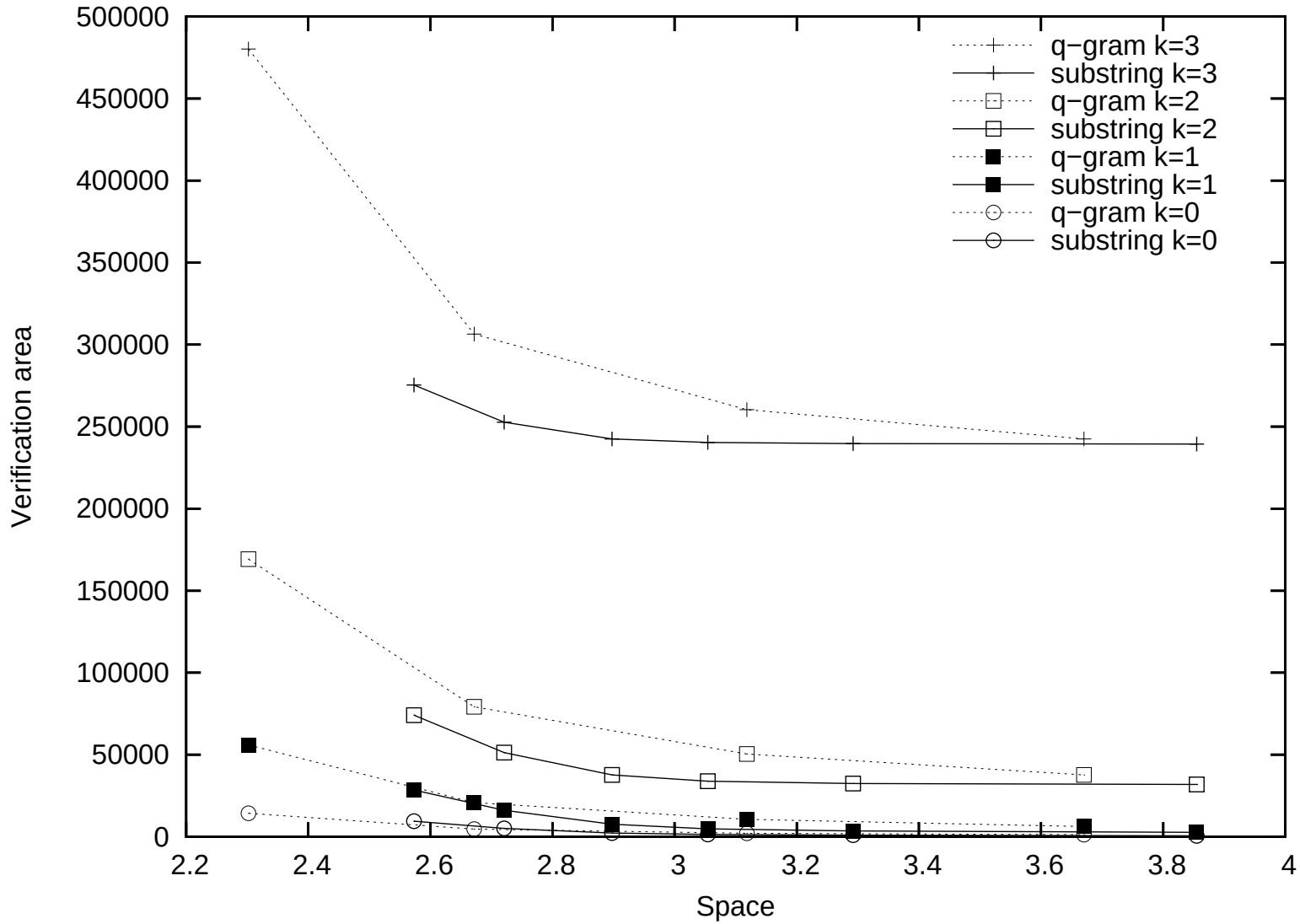
Outline

Verifications vs Space (Uncompressed Position Lists)



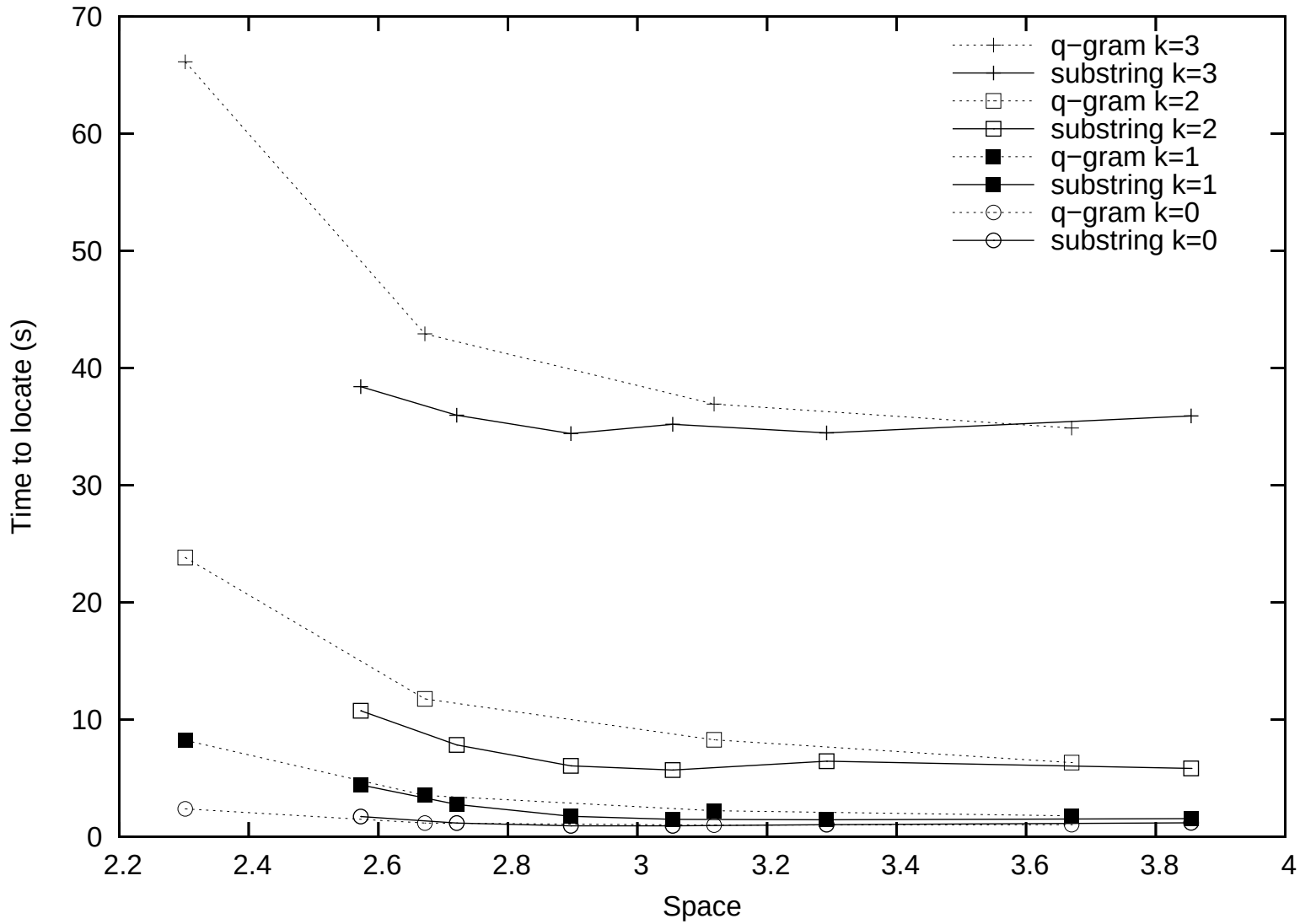
English text, $m = 20$

Verifications vs Space (Compressed Position Lists)



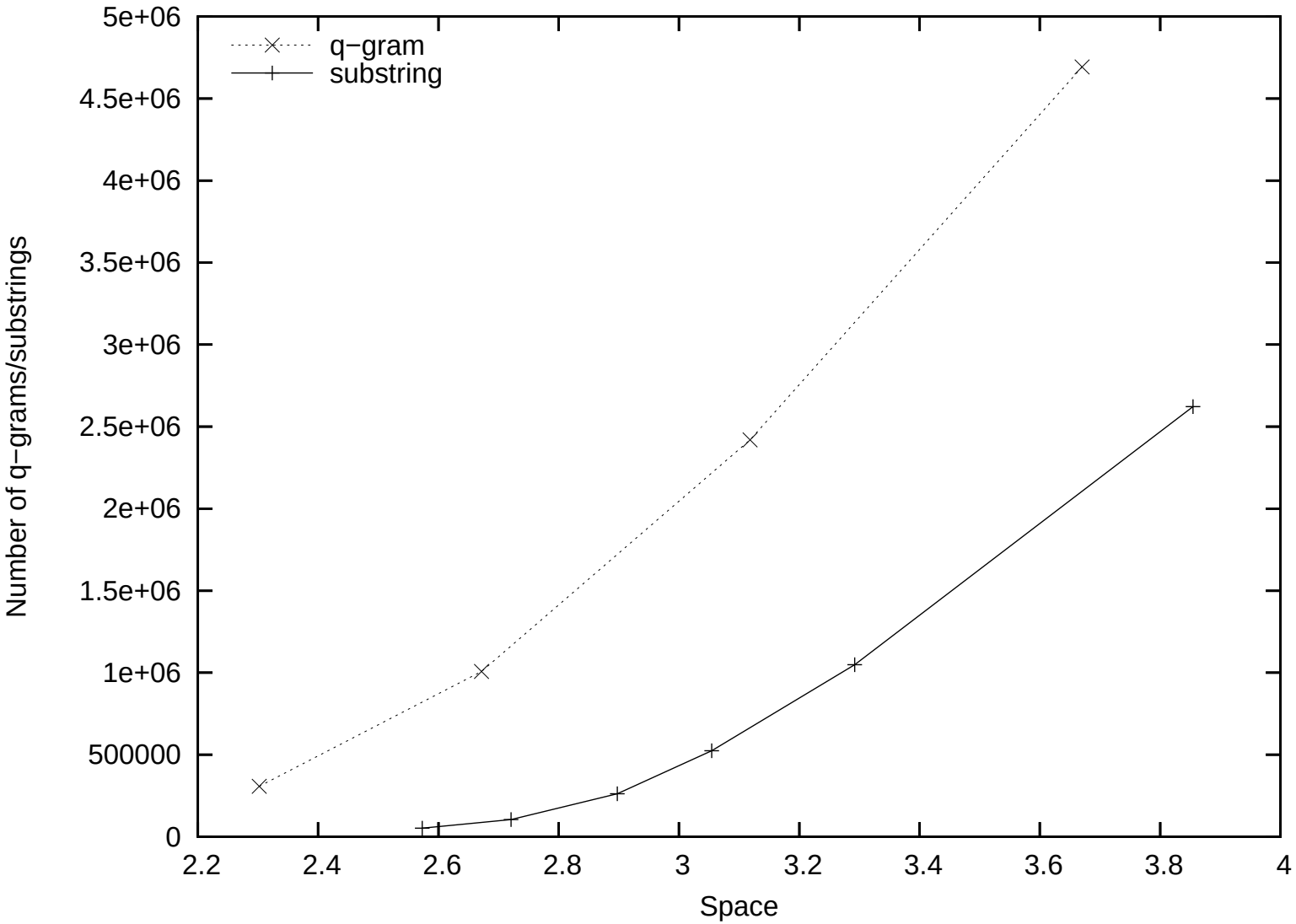
English text, $m = 20$

Time to Locate vs Space (Compressed Position Lists)



English text, $m = 20$

Number of q-grams (Substrings) vs Space



English text

Experiments on Other Data

Proteins and DNA

- q-gram index and substring index equally good

XML

- Worse than with English text (when compared to the q-gram index)

Sources

- Better than English text (when compared to the q-gram index)

Conclusions

- q-gram index: fixed length q-grams are indexed
- Substring index: variable length substrings indexed