Querying N-grams Over Dynamic Datasets

Tim Contois
Alex Lassalle
Josh Pikovsky
Andrew Gramigna

Introduction

N-grams are sequences of n contiguous components of a language string, where the components can be anything from words to syllables depending on the requirements of the particular implementation. N-grams can take the form of unigrams (n-gram of size 1), bigrams (size 2), trigram, or just the value of n-gram for larger sizes. Many data mining and natural language processing processes involve examining the n-grams for a given text. For example, considering the phrase "Tim only eats pizza" the bigrams would be: "Tim only", "only eats", "eats pizza", and the trigrams would be "Tim only eats" and "only eats pizza". Language models based on n-grams are commonly used in fields like computational linguistics and probability and can assist with many common language processing tasks like spelling correction and speech recognition.

Searching for n-grams is an important problem that has many applications spanning language processing, information retrieval, security and digital forensics. Consider the motivating example of TimBook, a popular social networking site with a large user base that provides its users with a chat feature. The owners of the site are contacted by the National Security Agency, and informed that there is an imminent threat to the country's security by individuals that are suspected to be users of their site. The potential security threat has some specific details associated with it, and the NSA provides TimBook with a list of flagged terms that could signal knowledge or intent of the crime and subpoenas the list of users that have used those terms. The engineers at TimBook now have a short time frame to search the massive database of conversations spanning the many years since the chat features' inception for the relevant terms. A possible solution for this would be developing and applying an algorithm that could efficiently query a document for known n-grams.

While there are several approaches to efficiently querying large and dynamic datasets for n-grams, string searching (or string matching) is an interesting and relevant class of algorithms that could be applied to this problem. In these algorithms, one or more strings are found and

matched in another, presumably larger text. The solution proposed by this paper builds on the concept string matching algorithms by implementing finite automata, and preprocessing data into specific data structures that could allow for fast querying.

Contest

The basis for this paper came from the SIGMOD 2017 Programming Contest, in which teams from degree-granting institutions were invited to compete in a programming contest, with the top two teams receiving prizes. The task for the 2017 contest was to create an efficient way to query n-grams over a dynamic dataset. The contest's page asks contestants to

"...filter a stream of documents using a set of n-grams of interest and, for each document, return where in it one of the n-grams is found. The input to the task will have two parts: first, an initial set of n-grams, which may be processed and indexed; second, a series of queries (documents) and n-gram updates (insertions or deletions), arbitrarily interleaved. For each n-gram insertion or deletion, the set of n-grams of interest is updated accordingly. For each new query (document) arriving, the task is to return as fast as possible the n-grams of the currently up-to-date set that are found in the document. These should be presented in order of their first appearance in the document" [1].

Essentially, the task is to pass through documents, and return n-grams from a given dataset as efficiently as possible. The contest page provides the n-gram dataset, and a test harness with which the submissions are evaluated. The tests require three operations:

Query: A query requires a response of n-grams that were found in the input document. For each document, the query returns a line which has all extracted n-grams. If none are found, -1 is returned.

Add: An add operation requires the addition of a new n-gram to the set of n-grams of interest to. The dataset should remain unmodified if the n-gram is already contained in it.

Delete: A delete operation modified the n-gram dataset by removing a given n-gram. If the n-gram does not exist, the set will remain unmodified.

Given the SIGMOD dataset, and the requirements for these queries, the task was to find an efficient and original solution.

Naive Solution

Given the problem of finding n-grams of interest within a large document, a naive solution would store a list of n-grams of interest, and then query the document by searching the document for each n-gram individually.

```
1. ngrams = [ ]
2. def add (ngram):
3.         ngrams.add (ngram);
4. def query (document):
5.         results = [ ]
6.         for ngram in ngrams:
7.                 if ngram in document:
8.                         results.add (ngram)
9.         return results
```

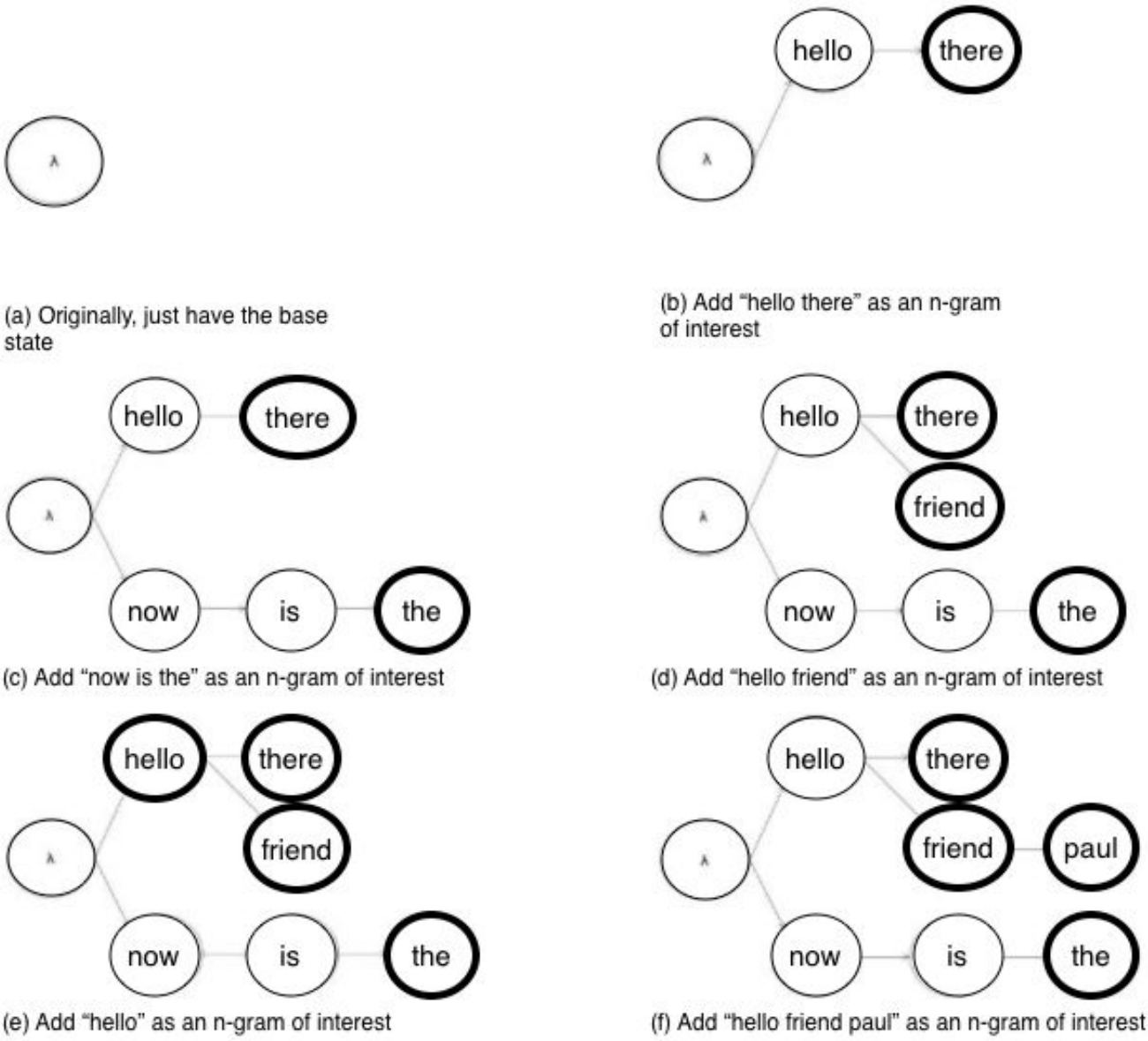
Figure 1: Pseudocode for a naive solution

One major drawback of this algorithm is that it searches the document once for each n-gram in the list of n-grams of interest. In situations with thousands of n-grams of interest, the naive solution will have to search the document thousands of times. Even with the use of a fast string matching algorithm to find each n-gram, such as Boyer-Moore (see Related Work), the dependence on the number of n-grams hinders the performance of the naive algorithm.

Implementing Finite Automata

We developed an algorithm (referred to as "TAJ") that only scans through the query document one time. To make one scan possible, TAJ preprocesses the n-grams into a deterministic finite automaton (DFA). In this DFA, each word of an n-gram corresponds to a state. Before any n-grams are added, the DFA consists of only the base state. As n-grams are added, a transition is added from the base state to the state corresponding to the first word of that n-gram. If multiple n-grams have the same first word, they share the state for that word rather than duplicating it. For example, the n-grams "hello" and "hello there" would both use the same state corresponding to "hello." This same pattern applies if n-grams begin with multiple words in common, such as "hello there friend" and "hello there paul."

It is also necessary to store the states that have output. To illustrate, consider the n-gram "sigmod programming contest." Only the "contest" state would have output. If the n-gram "sigmod programming" is added later, the "programming" state would be added to the set of output states, but no new states would need to be created.



(a) Originally, just have the base state

(b) Add "hello there" as an n-gram of interest

(c) Add "now is the" as an n-gram of interest

(d) Add "hello friend" as an n-gram of interest

(e) Add "hello" as an n-gram of interest

(f) Add "hello friend paul" as an n-gram of interest

Figure 2: Graphical illustration of TAJ's states. States with output are represented by making the state's border bold.

To query a document, TAJ only looks at transitions from a specific set of states. TAJ looks from whatever state(s) the previous word left it in and always looks from the base state. The transitions from the base state consists of the first word of each n-gram of interest (with no duplicates). Each state has a corresponding Python dictionary storing the transitions from that state. The dictionary maps words (the key) to their corresponding state (the value).

```
1.  s_id = 1              # Stores the next available state id. 0 refers to the base state λ
2.  transitions = {}      # Dictionary of dictionaries:  state_id  → {word → next_state}
3.  outputs = set()       # Stores states that correspond to the last word of an n-gram
4.  def add_ngram (ngram):
5.      prev_state = 0     # prev_state is originally the base state λ
6.      for w in ngram:
7.          if prev_state in transitions:
8.              ts = transitions[prev_state]
9.          else:
10.             ts = {}
11.             transitions[prev_state] = ts
12.         if w in ts:
13.             prev_state = ts[w]
14.         else:
15.             ts[w] = s_id
16.             prev_state = s_id
17.             s_id += 1
18.     outputs.add (prev_state)     # Add the last state of the new n-gram to outputs
19. def query (document):
20.     results = [ ]
21.     cur_states = [0]
22.     for w in document:
23.         new_states = [0]
24.         for s in cur_states:
25.             if s in transitions:
26.                 ts = transitions[s]
27.                 if w in ts:
28.                     new_id = ts[w]
29.                     new_states.append (new_id)
30.                     if new_id in outputs:
31.                         results.append (new_id's output)
32.         cur_states = new_states
33.     return results
```

Figure 3: Pseudocode for TAJ

In most scenarios while querying, TAJ will only be looking from a small number of current states in each iteration of the loop. TAJ always looks from the base state, which corresponds to looking for the first word of an n-gram of interest. If the word in the previous iteration is part of an n-gram, then TAJ also searches from the state the previous word brought it to. For example, consider a setup where the n-grams are "hello," "there," and "hello friend" and the query is "hello class." In the first iteration of the loop, TAJ searches from only the base state and finds a transition to "hello." In the next iteration of the loop, TAJ must search from both the base state (to see if "class" is the start of a new n-gram) and the "hello" state (to see if there is an n-gram

starting with “hello class”). In the worst case, TAJ must search from n different states where n is the number of words in the longest n-gram (for a proof see Runtime Analysis). The overall query runtime complexity for TAJ depends on the length of the query document and the number of words in the largest n-gram. For comparison, the naive solution depends on the length of the query document and the number of n-grams. In a situation with many n-grams all of relatively short length, TAJ can greatly outperform the naive approach.

Sorting with Buckets

The SIGMOD contest required the list of n-grams found in the document to be returned in a specific order. The resulting list of n-grams should be sorted first on the position of the first word of the n-gram in the document and secondly on the position of the last word in the n-gram if multiple words have the same starting position. To avoid sorting the n-grams in the correct order at the end, buckets can be added to the design. The number of buckets is made equal to n where n is the number of words in the longest n-gram. With this strategy, one bucket can safely be emptied in each iteration of the loop through the document. When a new n-gram is found, it is placed in the correct bucket corresponding to its start location. Within the same bucket (same starting index), no sorting is needed because shorter n-grams are placed in the bucket first. Once n iterations have passed without emptying a particular bucket, it is safe to empty that bucket because it corresponds to n-grams beginning with a word n positions earlier (which is no longer relevant because the longest n-gram of interest is length n). With this strategy, the output results never need to be sorted.

Evaluation

The SIGMOD contest ran each team’s algorithm on 4 different workloads (Small, Medium, Large, and X-Large).

Each team received starter code with the naive implementation an n-gram searching algorithm. This code looped over each n-gram of interest and searched the query document for that n-gram. This naive approach took approximately 46 seconds on the small workload and timed out on all the others. TAJ brought the small workload time down to under 2 seconds and no longer timed out on the Medium and Large workloads.

Table 1: Comparing the time in seconds for the Naive algorithm, TAJ, and the fastest algorithm on the leaderboard

Algorithm	Small*	Medium	Large**	X-Large
Naive	46s	timeout	timeout	timeout
TAJ	1.9s	45.9s	39s	timeout
Leader	0.03s	0.5s	0.7s	3.6s

*The Small test that was released began with two thousand n-grams initially and included a work document (adds/queries/deletes) that was 17MB.
**The large test set was at least 175k n-grams initially and had a work document of at least 93MB.

Runtime Analysis

m = length of query document

x = number of n-grams

n = number of words in longest n-gram

y = number of characters in longest n-gram

Table 2: Comparing the preprocessing and search time of mentioned algorithms

Algorithm	Preprocessing Time	Searching Time
Naive	$O(x)$	$O(m \cdot x)$
TAJ	$O(x \cdot n)$	$O(m \cdot n)$
Commentz-Walter	$O(x \cdot y)$	$O(m \cdot n)$

For the naive algorithm, the preprocessing step just involves putting each initial n-gram of interest in a list. In the worst case, the searching with the naive algorithm does not find any of n-grams in the query document and at worst it needs to search the entire document as many times as n-grams in the list giving us $O(m \cdot x)$. For TAJ preprocessing, at worst it adds x n-grams, all of the longest length (n), giving us $O(x \cdot n)$. In the worst case search, TAJ looks for possible matches to our n-grams or parts of n-grams from n different states throughout the entire document. This is better illustrated in the lemma below. For the Commentz-Walter algorithm, the preprocessing time is $O(x \cdot y)$, and the searching time is $O(m \cdot n)$ [8], meaning TAJ actually has a lower preprocessing time complexity than Commentz-Walter because n is always less than or equal to y. The searching time for TAJ and Commentz-Walter are equivalent in terms of complexity. Both TAJ and Commentz-Walter are faster than the naive solution as long as the

number of n-grams is greater than the length of the longest n-gram, which is almost always the case.

Lemma: TAJ worst case search

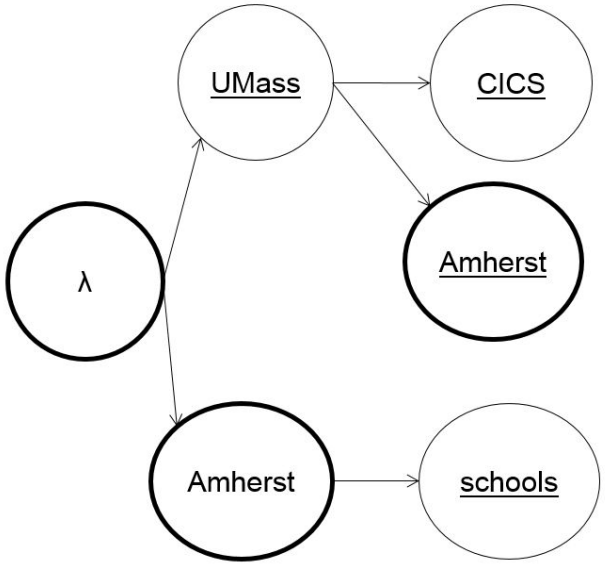


Figure 4: States being searched from when querying "students" are in bolded circles

First the algorithm queries "UMass" and it finds a transition from the base state to it. Next, it looks for the next word in the query from both the base state and the UMass state, which in this case is "Amherst". It is looking to see if "Umass Amherst" or "Amherst" are part of an n-gram of interest. It turns out they both are, which is the worst case scenario. Now the algorithm reaches the figure above where it searches for "students" from both Amherst states in the n-grams of interest. At worst Amherst appears n times, as the first, second, … , all the way to the nth word in our n-grams of interest. The algorithm must still check from the base state as well every time. It is clear from this diagram that the maximum number of states TAJ can look from is at most n+1. Graphically, consider the "columns of the DFA". In this example the columns are 0: "λ", 1: "UMass, Amherst", and 2: "CICS, Amherst, schools." There can be at most one state TAJ searches from in each column of diagram (and there are n+1 columns).

Related Work

There has been other work related to finite state automaton based searches and string matching. For string matching, each state would represent a partial match. For example the partial matches for "cats" are "", "c", "ca", "cat", and "cats." The start and accept case would be 0 character and m character prefix (for string length m). The transition from state+character to state would be the longest prefix of the original pattern and a suffix of the state+character state we've just seen. The

disadvantage of the character matching approach is that it takes $O(m^3)$ time for strings of length m to build the state table. The SIGMOD competition does not count preprocessing time.

"A linguistic knowledge discovery tool: Very large ngram database search with arbitrary wildcards" by Satoshi Sekine relates to a similar problem of querying in a large database of n-grams. This paper described a new search tool for a very large set of n-grams, which can be queried with an arbitrary number of wildcards. These searches are very fast and are designed to be a useful tool for natural language processing researchers. While recreating their design is impractical for the short time frame of the contest, a technique like this would create an n-gram database that is optimized for efficient wildcard queries. This paper also proved to be an interesting perspective on the many other strategies that could be used to speed up the queries.

Robert Boyer and J Moore developed what is still considered the fastest string-matching algorithm [3]. To determine if some query string *query* contained some known target string, *target*, the naive approach to string-matching would be to compare the first character of *query* with the first character of *target*. If the characters match then we check the next characters of the two strings. If there isn't a match we increase the starting position in *query* and again compare it with the first character of *target*. This algorithm has been shown to be quadratic [5].

The Boyer-Moore algorithm approaches the string matching problem from the opposite direction, that is from right to left. When the last character *target* does not match corresponding character in *query* there is no need to compare the first 1,2, ... *target.length-1* characters. We can slide down our position in *query* using their described shift functions which rely on values from preprocessing. The longer our target string the larger this shift will be and the more characters we do not need to compare with. If the last character of t*arget* does match the *target.length* character in *query* then we move on to check the second to last character in *query* and continue comparing until we can verify we do have a match.

The Boyer-Moore algorithm works carefully to avoid unnecessary comparisons. The worst-case complexity of the algorithm is $\boldsymbol{\Theta}(nm)$ and modifications made since the paper's release have reduced the worst-case time to be linear. We did not implement the Boyer-Moore algorithm directly because of the nature of our problem would require a new search for every n-gram. The algorithms we found more relevant such as the Commentz-Walter algorithm do rely on the fast string matching Boyer-Moore.

Further research into related work has found that string searching can be improved by implementing structures using suffixes. *"Online construction of suffix trees" [6]* by Esko Ukkonen presents the modern iteration of the suffix tree algorithm, an efficient search algorithm based on a method of preprocessing that builds substring indexes. By creating a trie-like data

structure that stores all suffixes of a string, occurrences of patterns can be more easily found. By taking a DFA for the suffixes of the tree, a suffix automaton can be created.

An additional strategy for string searching with indexes can be found in "*Suffix Arrays: a new method for on-line string searches"* [7], in which authors Manber and Myers propose an alternative technique that involves using an array data structure that contains the sorted suffixes of a string. This is an alternative to suffix trees that is described to be three to five times more space efficient. Using suffix arrays, a string search of type "Is W a substring of A?" can be answered in time O(len(W)+log(len(A)). Using a structure like this for string searching ends up being very useful for online queries for extremely long strings, like genetic sequences which can range in the order of one or more million symbols. While suffix trees are a useful and fundamental strategy for string searching, they are space expensive, and suffix arrays practically solve that problem by using arrays with bucket sorting. The authors write that their approach "...distills the nature of a suffix tree to its barest essence: a sorted array coupled with another to accelerate the search." [7]. By using an array, a string search algorithm can greatly decrease its query time and space requirements, albeit at the cost of a somewhat slower construction.

Future Work

The most specific area of future work would be to convert the code of our system to another language. Our algorithm was written in the high-level Python, and as such may have been slower than it could have been had it been written in a different language. Many of the contest solutions were written in C++, a language that is compiled rather than interpreted and therefore always faster than Python. Converting our code base to C++ would be the most obvious and intuitive improvement that could be made in the future. With C++ or another lower-level language, knowledge of the specific task could be leveraged to optimize certain parts of the algorithm.

Additionally, instead of Python data structures like dictionaries, the system could alternatively be represented as tables in a database. This could be a useful approach if the problem necessitated persistent storage of the collected n-grams. The Python script is on a run-by-run basis, and does not persistently store n-grams between runs. By using a database system, the n-grams could be stored cumulatively and continually used in queries. We found the Python data structures to be faster than using a database (such as SQLite) for SIGMOD purposes, but the states, transitions, and outputs can similarly be stored in a database and used in queries.

Another area of future work would be to examine other contributions to the programming competition, and to see if we can use some of the optimizations implemented by other teams. Per contest rules, the best 5 submissions will release their code and post a write-up of their algorithm. Looking through this information, there may be elements of their code that could

improve TAJ or vice versa. We could also further research the various string indexing techniques referenced in the related work, and try using those to speed up the algorithm.

Conclusion

While we were unable to join the leaderboard before the end of the contest in early May, the problem proposed was an interesting and difficult challenge, and we continued to work on it in search of an original and efficient solution. The focus of our solution was based on preprocessing the n-grams into data structures that could be more readily queried for results. Implementing a deterministic finite state automata proved to be a powerful improvement to the that allowed for the searching of multiple patterns at the same time. By using buckets, sorting the results could be eliminated from the process. Using these finite state machines and preprocessing techniques, we were able to develop TAJ, an algorithm that could speedily query large text files for a list of n-grams.

We were able to analyze our algorithm against a naive implementation, and the Commentz-Walter search, an algorithm we discovered as a comparable state based string search approach. Our algorithm runtime is significantly faster than the naive implementation, and is equivalent in runtime complexity to Commentz-Walter. In terms of our technical evaluation, our algorithm performed fairly well in the competition while the tests were still available to the public. In addition, our implementation was run in a higher level language (Python) which could have contributed to its slower speed. As mentioned in the future work, an implementation in C++ could be significantly faster by optimizing lower-level details.

The problem of finding a set of n-grams in a large dataset is an important problem with many useful applications across several subfields of computing. Our algorithm presents a possible solution to this problem by implementing finite state automata and string searching preprocessing techniques with verifiably fast results.

References

[1] http://sigmod17contest.athenarc.gr/

[2] "ICS 161: Design and Analysis of Algorithms Lecture Notes for February 22, 1996." Finite Automata and String Matching. N.p., n.d. Web. 03 Apr. 2017.

[3] Boyer, Robert S., and J. Strother. Moore. A Fast String Searching Algorithm. Ft. Belvoir: Defense Technical Information Center, 1976. Web.

[4] Sekine, Satoshi. "A linguistic knowledge discovery tool: Very large ngram database search with arbitrary wildcards." *22nd International Conference on on Computational Linguistics: Demonstration Papers*. Association for Computational Linguistics, 2008.

[5] Knuth, D.E., Morris, J.H., and Pratt, V.R. Fast pattern matching in strings. TR CS-74-440, Stanford U., Stanford, Calif., 1974.

[6] Ukkonen, E. (1995), "On-line construction of suffix trees" (PDF), *Algorithmica*, **14** (3): 249–260,

[7] Manber, Udi; Myers, Gene (1990). *Suffix arrays: a new method for on-line string searches*. First Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 319–327.

[8] Commentz-Walter B. (1979) A string matching algorithm fast on the average. In: Maurer H.A. (eds) Automata, Languages and Programming. ICALP 1979. Lecture Notes in Computer Science, vol 71. Springer, Berlin, Heidelberg