EXPONENTIAL LENGTH SUBSTRINGS IN PATTERN MATCHING

Vlad-Adrian Ulmeanu
Department of Computer Science
Polytechnica University of Bucharest
Bucharest, Romania 060042
vlad.ulmeanu01@gmail.com

Radu Iacob
Department of Computer Science
Polytechnica University of Bucharest
Bucharest, Romania 060042
radu.iacob@upb.ro

ABSTRACT

This work describes a hash-based mass-searching algorithm, finding (count, location of first match) entries from a dictionary against a string s of length n. The presented implementation makes use of all substrings of s whose lengths are powers of 2 to construct an offline algorithm that can, in some cases, reach a complexity of $O(n \log^2 n)$ even if there are $O(n^2)$ possible matches. If there is a limit on the dictionary size m, then the precomputation complexity is $O(m + n \log^2 n)$, and the search complexity is bounded by $O(n \log^2 n + m \log n)$, even if it performs in practice like $O(n \log^2 n + \sqrt{nm} \log n)$. Other applications, such as finding the number of distinct substrings of s for each length between 1 and n, can be done with the same algorithm in $O(n \log^2 n)$.

***K*eywords** Hashing · Trie · Exponential length substrings

1 Introduction

Some of the most popular applications of today's industry (Intrusion Detection Software that scan the network flow for malicious packets: Snort [1], Suricata [2], Antiviruses: ClamAV [3]) rely on regex matching complex rules to perform their tasks. Generally, the regex matchers are the bottleneck in such applications, and optimizations for them are sought after (Hyperscan [4], Google RE2 [5]).

Recently, speedups have been achieved ([6], Kargus [7]) on Deep Packet Inspection algorithms by preconditioning the regex match over a file: extract keywords from the rule, and see if they can be found in the packet (multi-string pattern matching). If a satisfying subset of them is found, then the more expensive regex may be tried to be matched.

This paper presents a different approach to the stated classic problem of pattern matching a dictionary against a string. Mainly, the presented algorithm's applications are for finding the count, or location of the first match of the words in the dictionary. Later on, other applications are shown, such as counting the number of distinct substrings, computing the substring frequency distribution, or an approximate online YES/NO matching of the dictionary words. While this latter solution may particularly return false positives, it may be used as a pre-filter for a regex matcher, since it cannot return false negatives.

The problem of mass-searching in a string for the existence of multiple substrings has been explored many times in the past: Aho-Corasick [8], Suffix trees [9], arrays [10] or automata [11] are some of the most well-known algorithms. However, the usage of polynomial hashes [12] was often overlooked while trying to solve this problem.

We want to write an offline algorithm for the following problem:

Input

- A string s of length n.
- A dictionary $ts = \{t_1, t_2, .., t_{sz(ts)}\}$.

Output

- For each string t in the dictionary, we want to answer how many times it is found in s.
- We could also ask for the position of the fist occurrence of each t in s, but the paper mainly focuses on the first question.

Other remarks

- The function sz refers to the length of either a set or a string.
- We also know the total number of characters in ts: $m = sz(t_1) + sz(t_2) + .. + sz(t_{sz(ts)})$.

The most common hash-based algorithm uses the fact that there could be only $O(\sqrt{m})$ different string sizes in a dictionary of size m. Therefore, by doing only $O(\sqrt{m})$ Rabin-Karp rolling hashes, we can get everything we need to answer the queries.

However, this simpler method generates $O(n\sqrt{m})$ hashes, which raises multiple problems:

- Lower difficulty of finding a collision because of the large amount of hashes generated.
- Generating a hash has a costly constant. As a result, implementations either take too long to run in practice, or give a wrong answer since a small modulo value was used to save time.

Our presented algorithm uses only $O(n \log n + m)$ hashes, which negates the presented issues, and enables a viable approach to this problem. A discussion regarding our algorithm's collision resistance exists in section 5.2.

The work will present the algorithm's summary in section 2, its base complexity in section 3, an optimization (section 4) and its resulting improvement in section 6, as well as the complexity when the dictionary size is bounded in section 7.

Also, we show an experimental complexity bound (section 8.1), and a runtime comparison between this algorithm and other popular ones (section 8.2). Finally, some applications are shown in section 9, along with the conclusion (section 10) and implementation references (section 11).

Some used notations follow:

- If t is a string, then $t[i:j] = t_i t_{i+1}..t_{j-1}$. $t[i..j] = t[i:j+1]$.
- If t is a string, t_{-1} is its last character. Similarly, if t were an array, t_{-1} would be its last element.
- A substring of s can be obtained by removing some characters from the beginning of s, and some characters from the back of s.
- A DAG is a Directed Acyclic Graph.
- The hash function is used in shortened form as H.

2 Algorithm Summary

We will take all substrings from s which have their lengths equal to a power of 2.

For example, for $s = aybabtu$, there are 17 substrings of interest:

- a
- y
- b
- a
- b
- tu
- ay
- yb
- ba
- ab
- bt
- $ayba$
- $ybab$
- $babt$
- $abtu$
- t
- u

The number of substrings of interest is $O(n \log n)$. There are $O(\log n)$ powers of 2 smaller or equal to n. For each considered power of 2, there are at most n possible positions in s where one could start the substring.

We will now construct a DAG in which each substring we have considered earlier becomes a node. We will add an edge between every pair of substrings that are one after another in s, but only if the first substring's length is strictly greater than the second substring's length. The DAG will also have a starter node that has an edge to every other node in the graph.

Since each DAG node (excluding the starter) may have at most $O(\log n)$ outgoing edges (there are $O(\log n)$ substrings starting exactly after the end of another substring), building the graph will take $O(n \log^2 n)$.

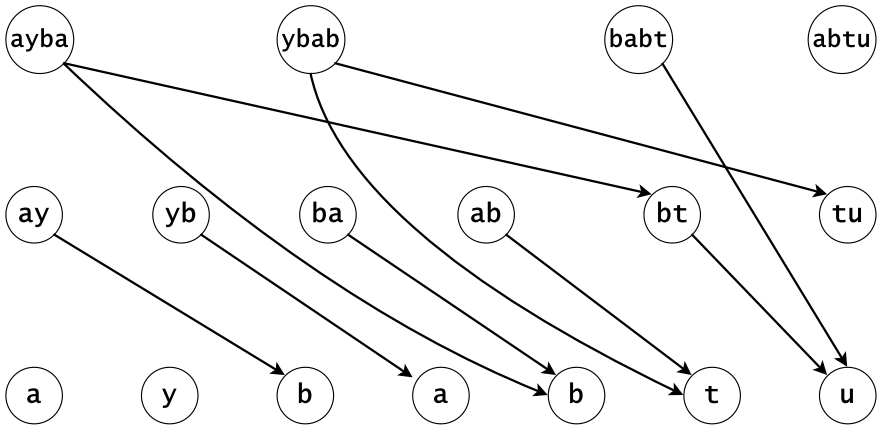


Figure 1: For $s = aybabtu$, the DAG will look like this.

If we want to see how many times a string t appears in s, we would break it down into a chain of substrings of 2-exponential length in strictly decreasing order (the chain's length is $O(\log n)$), and count how many times the chain occurs in the DAG.

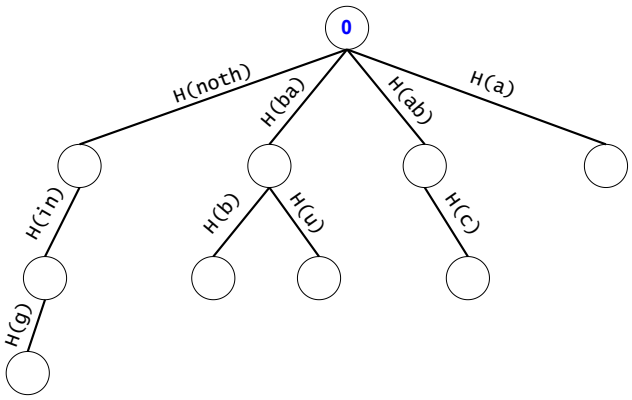
- For example, searching for $t = ybabt$ would result in breaking it down into $\{t[1..4], t[5..5]\}$ and counting the number of occurrences of the chain $ybab \rightarrow t$ in the DAG.

- While doing a search in the DAG, we will never need to traverse the edge $ay \rightarrow ba$, since any t could contain at most one substring of length 2, neither would we ever need to traverse $ay \rightarrow babt$, because we would instead traverse $ayba \rightarrow bt$.

All nodes will contain the hash for the value of the substring that is in it. This way, all search-related comparisons will take $O(1)$, and will generally have a high probability to be correct (section 5.2).

Also, all the substring chains will be transformed into hash chains.

Now, we will complement the DAG search by including a trie, which will be built with the dictionary strings' hash chains.



(a)

```
id = 0: starter node
id = 1: a
id = 2: ay
id = 3: ayba
id = 4: y
id = 5: yb
id = 6: ybab
id = 7: b
id = 8: ba
```

```
id =  9: babt
id = 10: a
id = 11: ab
id = 12: abtu
id = 13: b
id = 14: bt
id = 15: t
id = 16: tu
id = 17: u
```

(b)

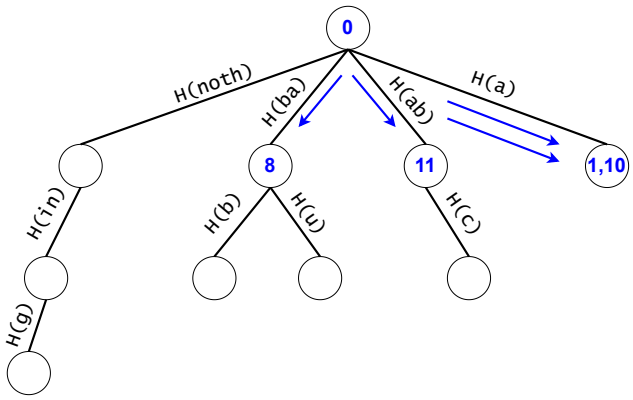
Figure 2: Complementary trie for $s = aybabtu$, and $ts = \{bau, abc, a, nothing, bab, ba\}$

The actual search will go through the trie and find help in the DAG.

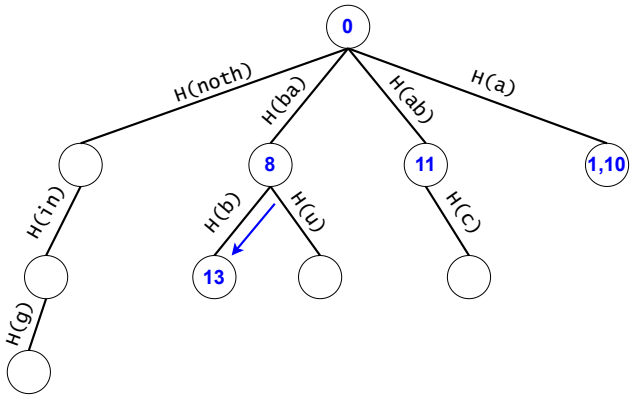
- Consider one "token" as a DAG node that is residing within a trie node. At the start of a search, there is only one token in the root of the trie, for the starter node of the DAG.

- During the search, each token will "divide" itself to its children in the trie. For each outgoing edge (generically $tkn \rightarrow oth$) a token has in the DAG, we will check if the trie node (with tkn in it) has an outgoing edge labeled with $H(oth)$.

- Recursively continue the search for each child from the trie node which has at least one token in it.



(a)



(b)

Figure 3: For the first figure, the token with the id 0 will look at its children in the DAG (all the nodes in this case) and see which ones can also be found on outgoing edges: node ba with the id 8, node ab with the id 11, and two tokens with the same value of a, 1 and 10. No DAG child of the id 0 is named $noth$, so we won't propagate anything through the trie edge labeled $noth$.

Further propagation is possible only from the token with the id 8, which has in the DAG a child with the value of b (second figure).

At the end of the search, consider for each hash chain its terminal node in the trie. The answer for each specific query is equal to the number of tokens in that trie node (for example 1 for ba, corresponding to the token with the id of 8).

$$\{bau(0), abc(0), a(2 : \{1, 10\}), nothing(0), bab(1 : \{13\}), ba(1 : \{8\})\}.$$

3 Complexity Calculation

We want to compute the complexity of the trie search. If the number of tokens in the trie is NT, and the complexity for dividing a token is DT, then the complexity is $O(NT \cdot DT)$.

The following snippet shows the pseudocode used for dividing the tokens found in a trie node.

```
for DAGnode in trieNode.tokens:
   for DAGson in g[DAGnode]:
      if hash[DAGson] in trieNode.sons:
         trieSon = trieNode.sons[hash[DAGson]]
         trieSon.tokens.append(DAGson)
```

Every token has an associated DAG node, so it has $O(\log n)$ children. For each of the children, we need to check if the trie node has the same hash on one of its outgoing edges as the child. Consider the trie node to have a hash table attached, so the checking phase is theoretically $O(1)$.

Therefore, the complexity for dividing a token can be:

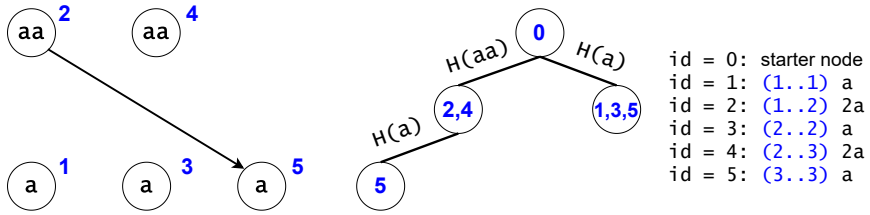
$$DT = O(\log n \cdot 1) = O(\log n).$$

The only exception for the previous statement takes place when dividing the starter node's token. Since it has $O(n \log n)$ children, dividing it will take $O(n \log n)$ instead.

Statement: Even if some tokens from different trie nodes may share the same DAG node attributed to them (i.e. the tokens have the same id), every token can be uniquely mapped to a substring from s (substring value and starting position matter).

Proof: If two tokens would describe the same substring of s (value-wise, not necessarily position-wise), they would both be found in the same trie node, since the described substrings result from the concatenation of the labels on the trie paths from the root.

Now, since the two tokens are in the same trie node, they can either have different DAG nodes attached (so different ids), meaning they map to different substrings (i.e. different starting positions), or they belong to the same DAG node, so one of them is a duplicate: contradiction, since they were both propagated from the same father. □



(a) (b)

Figure 4: For example, if $s = aaa, ts = \{a, aa, aaa\}$: the token numbered 5 can be found twice in the trie search, but while one occurrence accounts for $a[1..3]$ ($H(aa) \longrightarrow H(a)$), the other one accounts for $a[3..3]$ ($H(a)$). Tokens found in the same trie node, such as 2 and 4 account for different substrings: $a[1..2]$, and $a[2..3]$.

There can only be as many tokens in the trie as the number of possible substrings (considering two of them different if they don't have the same starting position). We also have to count the token in the starter node:

$$1 + \frac{n(n+1)}{2} \geq NT \Longrightarrow NT \in O(n^2).$$

Therefore, the total complexity of the trie search is $O(n \log n + n^2 \log n) = O(n^2 \log n)$.

We are now interested in finding a way of decreasing the number of tokens that have to be held by the trie.

4 DAG Compression

Consider the subtree of a node from a DAG to be formed of all nodes which can be reached from it using a DFS.

We will use the following optimization in order to decrease the number of nodes in the DAG: *if two nodes have exactly the same subtree, unite them.*

In order to preserve the $O(n \log^2 n)$ precomputation time for building the DAG, we will confer each node's subtree a hash value. Afterwards, we will unite two nodes iff they share the same subtree hash value.

Let $sbh_{i,j}$ be the associated subtree hash for the node representing the substring $s[i \,..\, i + 2^j - 1]$.

Then:

$$sbh_{i,j} = H(s[i \,..\, \min(n,\, i + 2^j - 1 + 2^j - 1)]) = H(s[i \,..\, \min(n,\, i + 2^{j+1} - 2)]).$$

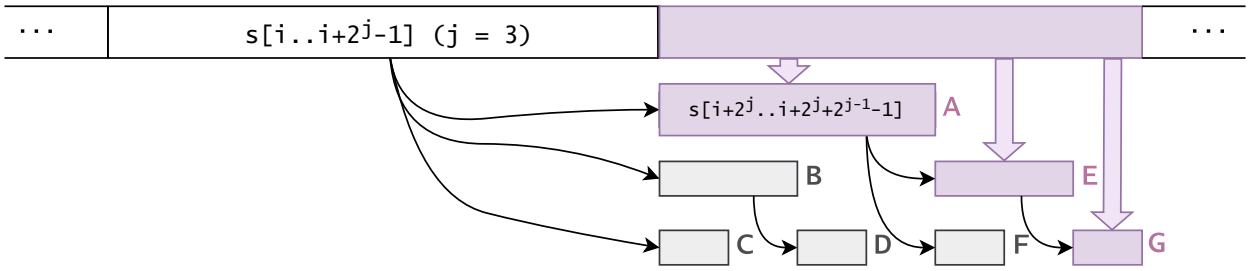


Figure 5: Since all the values of the nodes in the subtree of $s[i \,..\, i + 2^j - 1]$ are kept in the next $2^j - 1$ characters (or how many are left) $\Rightarrow$ assign $sbh_{i,j}$ to be the combined hash of $s[i \,..\, i + 2^j - 1]$ and $s[i + 2^j \,..\, \min(n, i + 2^j - 1 + 2^j - 1)]$.

We calculate the subtree hashes, then sort all of them in $O(n \log n)$ (achievable in practice with a high constant if radix sort is employed), and unite any two consecutive positions in the resulting array that have the same subtree hash.

When uniting, we always keep the node whose substring begins furthest to the left. This will be helpful for other applications, such as finding the leftmost occurrence of every string in the dictionary.

We will now introduce a leverage metric for each node in the DAG: lev_{node}.

- Before uniting anything, we have $lev_{node} = 1 \; \forall \, node \in$ DAG.
- When uniting two nodes $a \leftarrow b$, we aggregate their leverages: $lev_a \leftarrow lev_a + lev_b$, $lev_b \leftarrow 0$.

Statement: At the end of the DAG compression phase, let the weight of a chain of nodes in the DAG $ch = \{node_1, node_2, .., node_{sz(ch)}\}$ to be:

$$w_{ch} = \min(lev_{node} \mid node \in ch) = lev_{node_1}.$$

If the hash chain bound to the substring t is found k times in the DAG under the names $\{ch_1, ch_2, .., ch_k\}$, then the number of occurrences of t is:

$$\sum_{i=1}^{k} w_{ch_i} = \sum_{i=1}^{k} \min\left(lev_{node} \mid node \in ch_i\right)$$

Suppose that the weight of the chain ch_i is its contribution to the number of occurrences of t in s.

Proof: If $1 \leq i < j \leq sz(ch)$, then $node_j$ is in $node_i$'s subtree.

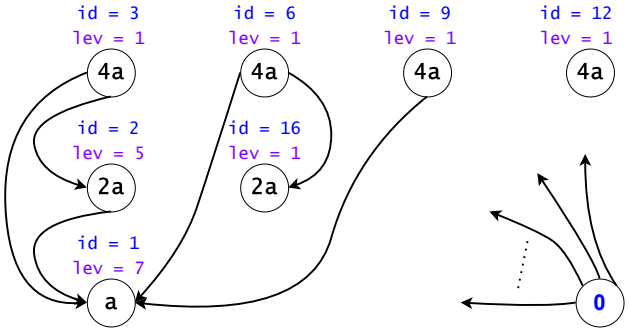
The more we go into the chain, the less restrictive the compression requirement becomes (i.e. fewer characters need to match to unite two nodes).

If we united $node_i$ with another node x, then there must be a node y in x's subtree that we can unite with $node_j$. Therefore, $lev_{node_i} \leq lev_{node_j}$.

The difference between lev_{node_1} and lev_{node_2}, lev_{node_3}, ..., $lev_{node_{sz(ch)}}$ is due to other unite operations that took place, which do not interest us when calculating w_{ch}.

So w_{ch} is actually lev_{node_1}, or $\min(lev_{node} \mid node \in ch)$. □

Let na represent $aa..a$ (n times).



(a)

```
id = 0: starter node
id = 1: (1..1) a
id = 2: (1..2) 2a
id = 3: (1..4) 4a
id = 4: (2..2) a
id = 5: (2..3) 2a
id = 6: (2..5) 4a
id = 7: (3..3) a
id = 8: (3..4) 2a
id =  9: (3..6) 4a
id = 10: (4..4) a
id = 11: (4..5) 2a
id = 12: (4..7) 4a
id = 13: (5..5) a
id = 14: (5..6) 2a
id = 15: (6..6) a
id = 16: (6..7) 2a
id = 17: (7..7) a
```

(b)

Figure 6: For example, compressing the DAG for $s = 7a$ will result in this.

- The number of occurrences for $t = 2a$ would be $\min(\{5\}) + \min(\{1\}) = 6$.
- $t = 3a$: $\min(\{5, 7\}) = 5$.
- $t = 5a$: $\min(\{1, 7\}) \cdot 3 = 3$.
- $t = 6a$: $\min(\{1, 5\}) + \min(\{1, 1\}) = 2$.
- $t = 7a$: $\min(\{1, 5, 7\}) = 1$.

5 Implementation details

We will discuss details from the main implementation [21] in this section.

5.1 Implicitness and other design choices

The precomputation phase, taking $O(n \log^2 n + m)$, can be split into:

- computing and sorting the subtree hashes.
- compressing the duplicate hashes, as well as updating the DAG node leverages and ids (what is the new alias of a deleted DAG node).
- computing hash chains from the dictionary.
- building the trie from the hash chains.

The search phase, whose complexity we will further reduce in section 7, consists of recursively performing the trie search with help from the DAG.

Even if we have presented two data structures, a DAG and a compressed trie, an efficient implementation would only need to generate the trie. The DAG may be implicitly referred to during the search phase.

The implicit representation of the DAG can achieved by allowing the DAG ids (such as those found in figure 2) to carry more information. Considering k to be the smallest integer such that $2^k \geq n$, let the $n + 1 - 2^p$ DAG nodes that refer to substrings of s of size 2^p have the ids $2^k p$, $2^k p+1$, .., $2^k p+n-2^p$. Let the id of the starter node to be -1 by convention.

Albeit the ids no longer form a continuous sequence, the penalty incurred by having slightly more cache misses is unnoticeable. The gains are much more important: by just having an id, we can figure out the position of its associated substring in s, its DAG node's hash, as well as its DAG children's hashes.

Radix sort is preferred when sorting hashes (for example, when checking for duplicate subtree hashes). The high constant is favourable to the additional $\log$ factor. Also, in order to reduce the number of system calls, trie nodes are allocated in batches.

5.2 Collision Resistance

We want to pick a hashing scheme such that:

- the probability of any collision existing in a test is very small.
- the scheme is resistant to known collision attacks. We consider that the attacker knows our scheme.
- the scheme performs the computations relatively quickly.

For a given string s, we have to generate $\lceil 2n \log_2 n \rceil$ hashes to build and compress its associated DAG. A dictionary of size m would warrant the creation of at most another m hashes. Since we have only $\lceil 2 \log_2 n \rceil$ different hash lengths ($1, 2, 4, ..$ for the hashes of the DAG nodes, and $1, 3, 7, ...$ for the subtree node hashes), we have a lot of hashes of the same length, so we want to use rolling hashes (and therefore a polynomial hashing scheme) for speed reasons.

Let the polynomial hash of a string u of length n, in base B, over the finite field $\mathbb{Z}/\mathbb{Z}M$ be:

$$H_{B,M}(u) \equiv \sum_{i=0}^{n-1} B^{n-1-i} conv(s_i) \pmod{M}$$

Suppose that we are working only with lowercase Latin letters, and that `conv(c) = c - ord('a') + 1`.

Some design choices that would enable known collision attacks that we are trying to prevent are:

- Picking a modulo that is a power of two, broken by generating s as a Thue-Morse sequence. [13]
- Picking a modulo M so small that randomly generating $O(\sqrt{M})$ hashes is feasible, resulting in a high likelihood of a collision between a generated pair. (birthday attack)
- Picking a large, but composed modulo. Vulnerable to the composed birthday attack: suppose that $M = M_1 \cdot M_2$, where both M_1 and M_2 are primes. Perform a birthday attack as if the modulo was only M_1. Obtain two strings (of the same, preferably small length) u and v. Now, perform a birthday attack as if the modulo was M_2, but with the modified alphabet $\Sigma' = \{u, v\}$, and the modified base $B' = B^{sz(u)}$. We will obtain another two strings w and x that collide modulo M_2. Since the letters in Σ' are the same modulo M_1, w and x will also collide modulo M_1.
- Fixed base: the tree attack. We try to collide two strings u, v of the same length n. We try to fit the difference between the polynomial hashes of u and v with coefficients $c_i \in \{-1, 0, 1\}$ such that $\sum_{i=0}^{n-1} (B^{n-1-i} \mod M) \cdot c_i = 0$, a sufficient condition for a collision if a solution is found. [14]

A survey that covers more attacks: [15]

We have picked a large prime modulo M. For each run of the algorithm, we will randomly pick a base from $\{27, .., M-1\}$.

In order to aide computation speed, M has been chosen to be the largest Mersenne prime that can fit in 8 bytes, $M61 = 2^{61} - 1$. Multiplying two numbers modulo $M61$ can be done with only some multiplies, adds, shifts and ternary operators.

Now, the probability for any two different strings u, v of the same length n to have the same hash modulo M:

$$P\Big(H_{B,M}(s) \equiv H_{B,M}(t) \pmod{M}\Big) =$$

$$P\Big(\sum_{i=0}^{n-1} B^{n-1-i} \cdot (conv(s_i) - conv(t_i)) \equiv 0 \pmod{M}\Big) \leq \frac{n-1}{M}$$

Since now the left-hand side is a non-zero $n-1$-degree polynomial in B over $\mathbb{Z}/\mathbb{Z}M$, therefore it cannot have more than $n-1$ roots, no matter what value may B take. M is prime, so this inequality is also the base-case for the DeMillo-Lipton-Schwartz-Zippel lemma [16].

Our design uses two different, and randomly independently chosen bases B_1, B_2 for each run.

For a value of $m \leq n \log_2 n$ that we will explore in section 7.2, we would generate at most $3n \log_2 n$ hashes per test. The hashes and their relationship can be split as:

- $\lceil n \log_2 n \rceil$ subtree hashes, that are only used to compress the DAG, and do not interact with the other hashes. However, we sort them in the chosen implementation, so we have to assume that $\binom{\lceil n \log_2 n \rceil}{2}$ hash comparisons take place.
- $\lceil n \log_2 n \rceil$ hashes for the DAG node values.
- At most $m \leq n \log_2 n$ hashes formed from the dictionary. These interact only with the hashes for the DAG node values, so another $\binom{\lceil 2n \log_2 n \rceil}{2}$ hash comparisons will take place.

Supposing that we have an array of independently, randomly chosen 16-byte values $otp_1, .., otp_n$, and a pseudorandom function $f : \{0,1\}^{128} \to \{0,1\}^{64}$, then the hash for an string u is computed as follows:

```
h1 = H[B1, M](u)
h2 = H[B2, M](u) #obtain h1 and h2, generally by rolling.
H128 = h1 * 2**64 + h2 #concatenate the two hashes.
H128 = H128 ^ otp[len(u)] #xor with the corresponding one-time pad.
H64 = f(H128) #finally, the usable 64-bit hash is the output of the PRF.
```

We are interested in remembering the hashes in an 8-byte format, since they would be used by many data structures in various implementations, and not filling up the cache/TLB entries so quickly does provide a noticeable speedup in running time.

We now want to calculate the collision probability for the hashes of two strings of different lengths. Since they are xor-ed with different one time-pads, the probability of a collision of two 128-bit values is 2^{-128}. If they do not collide post xor-ing, the outputs from the PRF may collide with a probability of 2^{-64}:

$$p_{diff} = 2^{-128} + (1 - 2^{-128}) \cdot 2^{-64} \simeq 2^{-64}$$

We also want to calculate the collision probability for the hashes of two strings of the same length n. Now, the probability of collision pre-PRF is at most $((n-1)/M)^2$:

$$p_{same} \leq \left(\frac{n-1}{M}\right)^2 + \left(1 - \left(\frac{n-1}{M}\right)^2\right) \cdot 2^{-64}$$

Assuming $n = 10^5$ means $((n-1)/M)^2 < 2^{-88}$. $p_{same} < 2^{-88} + (1 - 2^{-88}) \cdot 2^{-64} \simeq 2^{-64}$.

In both cases, the PRF probability dominates the sum. We can approximate both collision probabilities with 2^{-64}.

The probability of collision for the subtree hashes is:

$$p_{sub} \simeq 1 - (1 - 2^{64})^{\binom{\lceil n \log_2 n \rceil}{2}} \simeq 1 - (1 - 2^{64})^{2^{39.663}} \simeq 1 - e^{-2^{39.663-64}} \simeq 2^{39.663-64} \simeq 2^{-24.337}$$

The probability of collision for the other hashes can be computed similarly, but with an exponent 4 times as big: $p_{oth} \simeq 2^{-22.337}$.

Then the total collision probability per test is:

$$p_c = 1 - (1 - p_{sub})(1 - p_{oth}) \simeq 2^{-22}$$

If speed/memory can be sacrificed, we can forgo the PRF and have $p_{diff} = 2^{-128}$, $p_{same} < 2^{-88}$. $p_c < 2^{-46}$, assuming that $p_{diff} = 2^{-88}$.

We used the pseudo-random number generator `xoshiro256++` ([17], implementation: [18]) to act as a PRF: if we want to compute the PRF's output for the input x, seed the initial state of the PRNG with x in conjunction with `splitmix64` ([19], used implementation: [20]), then use the PRNG only once.

6 Corner Case Improvement

We will now attempt to prove that the complexity of the trie search after the DAG compression for $s = na$ is $O(n \log^2 n)$.

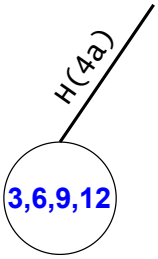


Figure 7: A trie node's "father edge" unites it with its parent in the trie.

Statement: If a trie node's "father edge" is marked as $2^x a$, then there can be at most 2^x tokens in that trie node.

Proof: If the "father edge" is marked as $2^x a$, then any token in that trie node can only progress for at most another $2^x - 1$ characters in total.

- For example, in figure 7, a token can further progress along its path for at most another 3 characters, possibly going through $H(2a)$ and $H(a)$.

Since all characters are a here, the only possible DAG subtree configurations accessible from this trie node are the ones formed from the substrings $\{0a, 1a, 2a, 3a, ..., (2^x - 1)a\}$, so 2^x possible subtrees (figure 8).

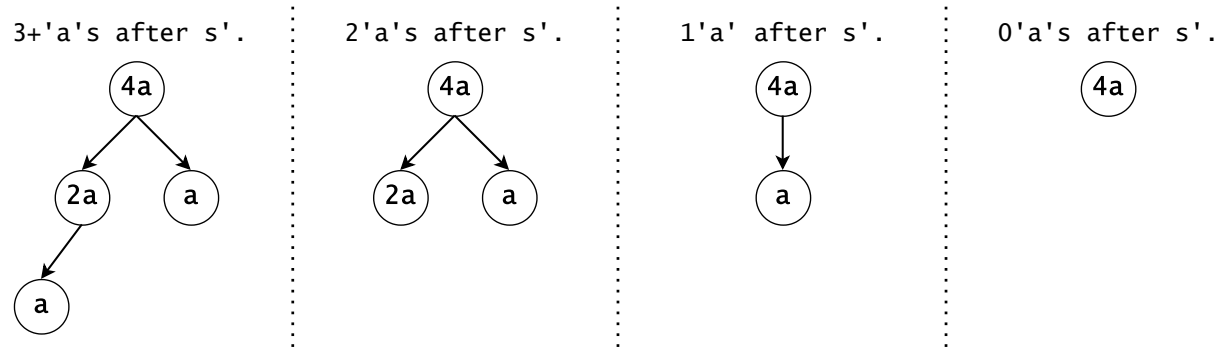


Figure 8: For example, consider a token's associated DAG node with the value of $4a$, and s' to be the substring that led to that trie node (i.e. the concatenation of the labels on the trie path from the root).

Recall that a token corresponds to a DAG node. If we have more than 2^x tokens in the trie node, then there must exist two tokens who belong to the same DAG subtree configuration, meaning that their nodes would have been united during the compression phase. Therefore, we can discard one of them.

So the number of tokens in a trie node cannot be bigger than the number of distinct subtrees that can follow from the substring that led to that trie node (in this case 2^x). □

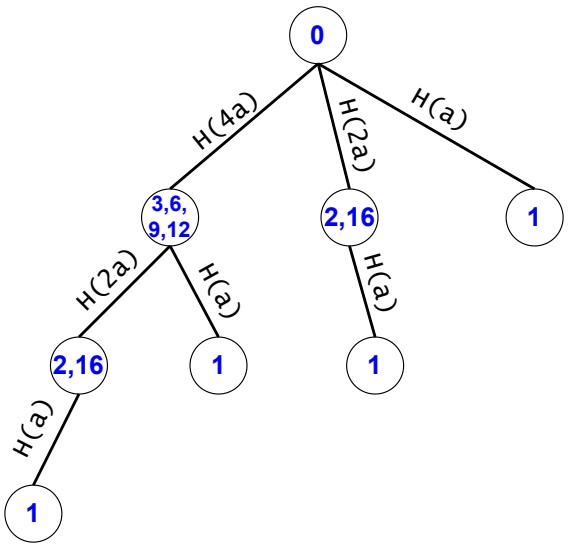


Figure 9: For example, the trie search after the DAG compression for $s = 7a$.

Let $trie(n)$ be the trie built from the dictionary $\{1a, 2a, 3a, .., na\}$.

Let $cnt_n(2^x)$ be the number of nodes in $trie(n)$ that have their "father edge" labeled as $2^x a$.

- For example, in figure 9, $cnt_7(1) = 4$, $cnt_7(2) = 2$, and $cnt_7(4) = 1$.

Pick k such that $2^{k-1} \leq n < 2^k$. Then, $trie(n) \subset trie(2^k - 1)$, since $trie(2^k - 1)$ can be obtained from $trie(n)$ by adding the following strings in the dictionary: $\{(n+1)a, (n+2)a, .., (2^k - 1)a\}$.

Therefore, no matter what s has been chosen to build the DAG, if we would use $trie(2^k - 1)$ as its associated trie, it will have at least as many tokens in it at the end of the search rather than if the trie had been $trie(n)$.

Statement: We will prove by induction that $trie(2^k - 1)$ has:

$$cnt_{2^k-1}(2^x) = 2^{k-1-x} \ \forall x \in [0, k-1].$$

Proof: If $k = 1$, then $trie(1)$ has only one outgoing edge from the root node labeled $1a \implies cnt_1(2^0) = 1 = 2^{1-1-0}$.

We will now suppose that we have proven this statement for $trie(2^{k-1} - 1)$ and we will try to prove it for $trie(2^k - 1)$.

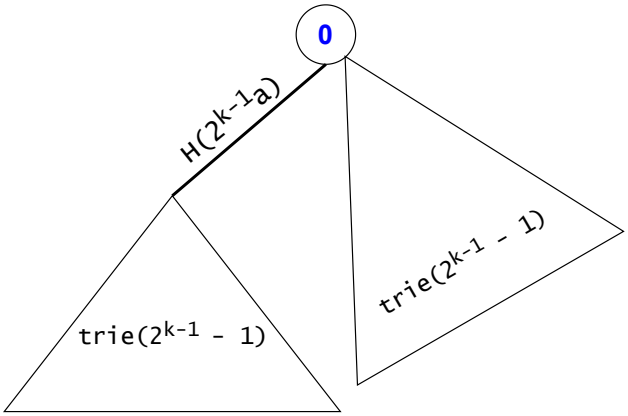


Figure 10: The expansion from $trie(2^{k-1} - 1)$ to $trie(2^k - 1)$.

As a result:

- $cnt_{2^k-1}(1) = 2 \cdot cnt_{2^{k-1}-1}(1) = 2^{k-2} \cdot 2 = 2^{k-1}$,
- $cnt_{2^k-1}(2) = 2 \cdot cnt_{2^{k-1}-1}(2) = 2^{k-2}$,
- ...
- $cnt_{2^k-1}(2^{k-2}) = 2 \cdot cnt_{2^{k-1}-1}(2^{k-2}) = 2^1$,
- $cnt_{2^k-1}(2^{k-1}) = 1 = 2^0$. $\square$

We will now count the number of tokens for $trie(2^k - 1)$:

$$cnt \leq \sum_{i=0}^{k-1} cnt_{2^k-1}(2^i) \cdot 2^i = 2^{k-1} \cdot 2^0 + 2^{k-2} \cdot 2^1 + .. + 2^0 \cdot 2^{k-1} = 2^{k-1} \cdot k.$$

So there are $O(k \cdot 2^{k-1})$ tokens in $trie(2^k - 1)$. Since $2^{k-1} \leq n < 2^k \Rightarrow k - 1 \leq \log_2 n < k \Rightarrow k \leq \log_2 n + 1$.

Also, $2^{k-1} \leq n$.

Therefore, $O(k \cdot 2^{k-1}) \subset O((\log_2 n + 1) \cdot n) = O(n \log n) \Rightarrow \forall\ s = na$, its associated trie search will generate $O(n \log n)$ tokens.

Since the complexity for dividing a token is $O(\log n)$, the trie search complexity for $s = na$ is $O(n \log^2 n)$.

We will now try to fit the entire program's complexity in $O(n \log^2 n)$ if $s = na$ and $ts = \{1a, 2a, 3a, .., na\}$.

Recall that any string in the dictionary is defined by its hash chain of length $O(\log n)$, so we don't really need to give the strings in the dictionary in complete form. Using a compressed form, such as $3a2b5c$ (meaning $aaabbccccc$), then figuring out the hash chain in $O(r + \log^2 n)$ is a better option. (r is the length of the compressed string) [22]

Reading the uncompressed dictionary will take $O(n \cdot (1 + \log^2 n)) = O(n \log^2 n)$. We have shown that both the precomputation and trie search phases both take $O(n \log^2 n)$, so the final complexity is $O(n \log^2 n)$.

Another way of achieving this complexity for this edge case would consist in reading the strings in the dictionary directly in hash chain form.[23]

Therefore, any standard method that has to read the uncompressed dictionary will do worse than the presented algorithm if $s = na$, the uncompressed dictionary length is $O(n^2)$ and its compressed dictionary length is $O(n)$.

If the standard method being used is suffix based, we can provide an almost-full dictionary with $O(1)$ missing entries that could have been found in s, and another $O(1)$ entries that can't be found in s, so that the method still has to go through the $O(n^2)$ dictionary characters to figure out which queries are missing or extra.

If s has a period of length p, and every character in the period would be distinct (i.e. we discussed $p = 1$ for $s = na$, $p = 2$ would be $s = ababab...$), and the dictionary would consist of every substring that can be found in s ($O(np)$ of them), then the running time would be $O(pn \log^2 n)$.

There are p times as many nodes in the trie. From each node of length 2^k there can still only be O(2^k) different continuing subtrees, since no matter where we pick the substring, the continuing patter is identic. (for $bcab$: ϵ, b, bc, bca). Therefore, the running time compared to $p = 1$ would be $O(p)$ slower. If p is small enough, we can still consider the running time to be $O(n \log^2 n)$.

Also, similar running times can be achieved is s is a periodic string plus a constant number of added characters, i.e. $abab...abc$, since it would still have $O(n)$ distinct substrings.

7 Dictionary Size Limit

Generally, a limit for the total number of characters that can be found in the dictionary is given (m). We will now attempt to find an upper bound for the search phase complexity if such a constraint exists. Implicitly, we wish to bound the total number of generated tokens during the search phase.

7.1 No DAG compression

We will first try to find an upper bound if no DAG compression is performed. If we would like to search for some strings of length l each, then we can generate tokens corresponding to at most $n + 1 - l$ substrings of s. Ideally, we would want to generate the corresponding tokens for all $n + 1 - l$ substrings while searching for only one string of length l. This is possible if $s = na \; \forall \, l \in [1, n]$.

As a consequence, the upper bound if no DAG compression is performed is given by the case $s = na$.

We now have to model a function $NT : \mathbb{N}^* \to \mathbb{N}, NT(m) =$ if the dictionary character count is at most m, what is the maximum number of tokens that we can generate?

If t' cannot be found in s, let t be its longest prefix that can be found in s. Then, no matter if we would have t or t' in any ts, the number of generated tokens would be the same. As a result, we would prefer adding t, since it would increase the size of the dictionary by a smaller amount. This means that in order to maximize $NT(m)$, we would include only substrings of s in ts.

For $s = na$, the substrings that we want to put in the dictionary are $\{1a, 2a, .., na\}$.

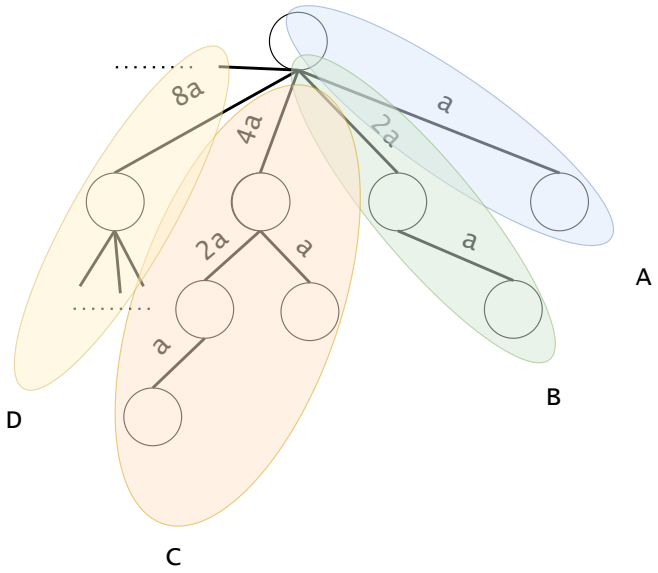


Figure 11: Consider the "full trie" to be the complementary trie filled with all of the options that we want to put in ts.

Also, consider it segmented into sub-tries, each sub-trie being rooted in one of the full trie root's children. (here the segments being colored areas from figure 11: A, B, C, D, ..)

Consider a sub-trie as finished if by adding any other substring to ts, the number of tokens offered by that sub-trie doesn't increase anymore.

- For example, A is finished if $\{a\} \subset ts$, B when $\{3a\} \subset ts$, C when $\{5a, 7a\} \subset ts$, D when $\{9a, 11a, 13a, 15a\} \subset ts$, ...

Consider starting exploring a sub-trie as adding a substring in ts that increases its token contribution from 0.

Statement: In order to maximize $NT(m)$, we should explore and finish the sub-tries in the order of their magnitude, without starting to explore other sub-tries midway.

Proof: We start with ts empty.

We can make two types of changes to it:

- "Add" a string.
- "Upgrade" a string that is already in ts. For example, if $m = 2$, we can choose $ts = \{2a\}$. When we increase m to 3, a good option would be to "upgrade" $2a$ in the dictionary to $(2+1)a$, since the $n-1$ tokens generated by $2a$ would be preserved, while adding another $n-2$ tokens.

Whenever we do the "add" operation, we want to momentarily maximize the slope of the NT function: $\triangle NT/\triangle m$.

If possible, upgrading is always better than adding:

Let $ST(k)$ be a sub-trie containing the substrings $\{(2^k)a, .., (2^{k+1}-1)a\}\, \forall k \geq 0$. Let $0 \leq k < k'$:

- If we add from $ST(k)$, $\triangle NT/\triangle m \geq n + 1 - (2^{k+1}-1)/2^{k+1} - 1$.
- If we add from $ST(k')$, $\triangle NT/\triangle m \leq n + 1 - (2^{k'})/2^{k'}$, which is lower than the minimum slope that we would get by adding from $ST(k)$.

So the optimal order of operations when filling ts would look like:

```
k = 0
while can fill ts:
   if ST(k) finished: k += 1
   else if ST(k) not explored: add a string from ST(k).
   else:
      if can upgrade string from ST(k): upgrade from ST(k).
      else: add a string from ST(k).
```

Therefore, we should explore and finish $ST(0)$, $ST(1)$, .. in this order. □

In order to finish a sub-trie, we need to include in ts all substrings that end in a leaf of that sub-trie, i.e. all substrings of odd length that are found in that sub-trie. We also do not need to include any other substrings to finish that sub-trie, since any node in it can be found on the path from the sub-trie root to a particular leaf.

Therefore, the optimal way to put strings in ts would be:

- $1a$
- $3a$
- $5a, 7a$ (one specific order would guarantee better results midway, but we are not concerned with that)
- $9a, 11a, 13a, 15a$
- ...

Now, for certain values of m, we know upper bounds for $NT(m)$:

m	**NT(m)**
1	n
$1+3$	$n+(n-1)+(n-2)$
$1+3+(5+7)$	$n+...+(n-6)$
$1+3+(5+7)+(9+11+13+15)$	$n+...+(n-14)$
...	
$1+3+5+7+...+(2^k-1)=4^{k-1}$	$n+(n-1)+...+(n+1-(2^k-1))$

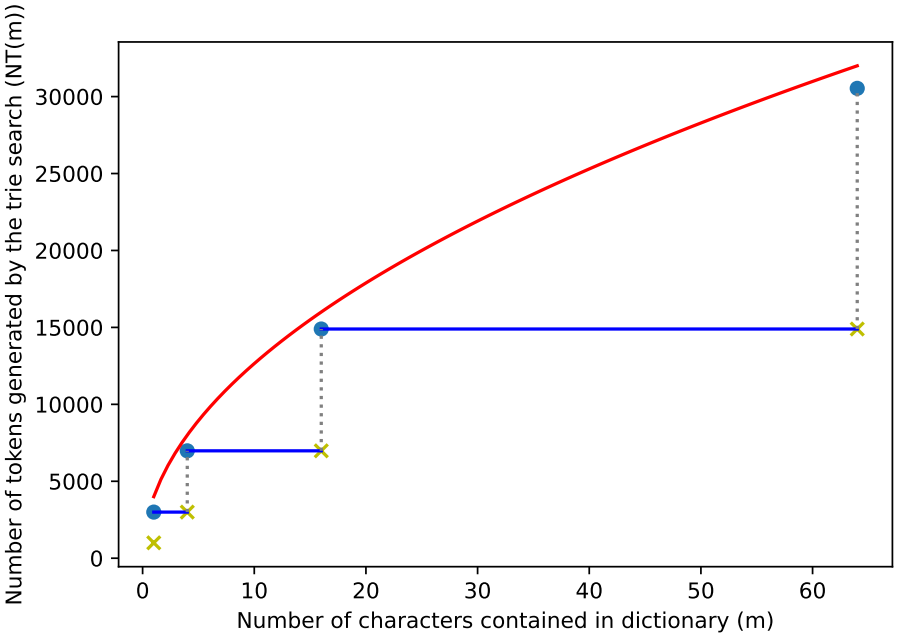


Figure 12: The data in the table is represented by goldenrod crosses in this plot.

We will now try to plot how the graph of NT would look in the worst case situation. We know that if $m \leq m'$, then $NT(m) \leq NT(m')$, so in the worst case we will just assume that:

$$NT(4^{k-2}+1) = NT(4^{k-2}+2) = \ldots = NT(4^{k-1}) \; \forall \, k \geq 2$$

In figure 12, the above presented worst case is drawn with blue lines. Even more, we will assume that the goldenrod crosses are as well on the blue lines (represented here as blue dots).

Now, if we bound the following inequality, we bound the whole NT function.

$$\forall m = 4^{k-1}, \; NT(m) \leq n + (n-1) + ... + (n+1-(2^{k+1}-1)).$$
$$\implies NT(m) \leq n \cdot (1+2^{k+1}) = n + n \cdot 2^{k+1}.$$

But if $m = 4^{k-1} \implies 2^{k+1} = 4\sqrt{m} \implies NT(m) \leq n + 4n\sqrt{m}$.

So the whole NT function is bounded by $NT(m) \leq n + 4n\sqrt{m}$ - the red line in figure 12. $NT(m) \in O(n\sqrt{m})$.

The total complexity of the search part is now also bounded by $O(n\sqrt{m}\log n)$, so the total program complexity becomes $O(m + n\log^2 n + \min(n\sqrt{m}\log n, n^2 \log n))$.

7.2 With DAG compression

We want to prove that $NT \in O(n \log n + m)$. We will suppose that $NT(m) \leq n + n \log_2 n + m$. If we would want to invalidate this claim, we should (at least):

- Overcome the function's bias of $n + n \log_2 n$, and
- Have at least for a moment (i.e. at least whilst appending one more string to ts) the slope of the function $\triangle NT / \triangle m > 1$.

We have already proven that the worst case for defining NT's bound will occur if ts is formed only from substrings of s.

We will use the following definitions:

- if $sz(t) = 2^{k_1} + 2^{k_2} + .. + 2^{k_{-1}}$, with $k_1 > k_2 > .. > k_{-1}$, then let $get(t, i) = g(t, i) = t[2^{k_1} + 2^{k_2} + .. + 2^{k_{i-1}} : 2^{k_1} + 2^{k_2} + .. + 2^{k_i}], i \geq 1$.
  - For example, if $t = ababc$, $get(t, 1) = abab$, $get(t, 2) = c$.
- if $sz(t)$ is a power of 2, then $sub(s, t) =$ the number of nodes from s' DAG post-compression whose values are equal to t.
  - For example, $sub(abababc, ab) = 2$.
- if t is a substring of s, $sz(t)$ is a power of 2, and we know t's position in s ($t = s[i..j]$), then $shade(s, t) = sh(s, t) = s[i.. \min(n, j + j - i)] = s[i.. \min(n, 2j - i)]$.
  - For example, if $s = asdfghjkl$, $sh(s, sdfg) = sdfg + hjk$, $sh(s, a) = a$, $sh(s, jk) = jk + l$, $sh(s, kl) = kl$.
- if x is a nonnegative integer, then $popc(x) = pc(x) =$ the number of bits equal to 1 in x's representation in base 2.

If we want to add $t \in ts$, t being a substring of s, then:

- $sub(s, get(t, 1)) = 1$, or
- $sub(s, get(t, 1)) > 1$.

A) If $sub(s, get(t, 1)) = 1$:

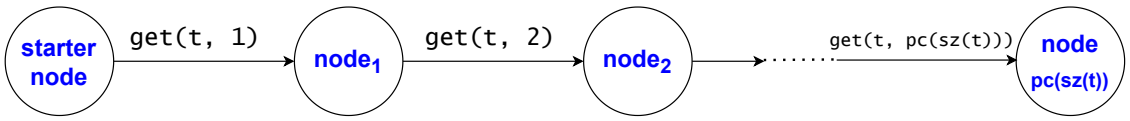


Figure 13: t's way into the trie will look like this.

After adding t into ts:

- $node_1$ will have at most another $sub(s, get(t, 1)) = 1$ token in it.
- $node_2$ will have at most another $\min(sub(s, get(t, 1)), sub(s, get(t, 2))) = 1$ token in it.
- ...
- $node_{pc(sz(t))}$ will have at most another token in it.

$$\Rightarrow \frac{\triangle NT}{\triangle m} \leq \frac{popc(sz(t))}{sz(t)} \leq 1.$$

We cannot hope to do any better than the bound's slope in this case, so we don't even have to worry about the bias here.

B) If $sub(s, get(t, 1)) > 1$, then let $t_1, t_2, .., t_{-1}$ be the substrings from s for which $get(t_i, 1) = get(t, 1)$.

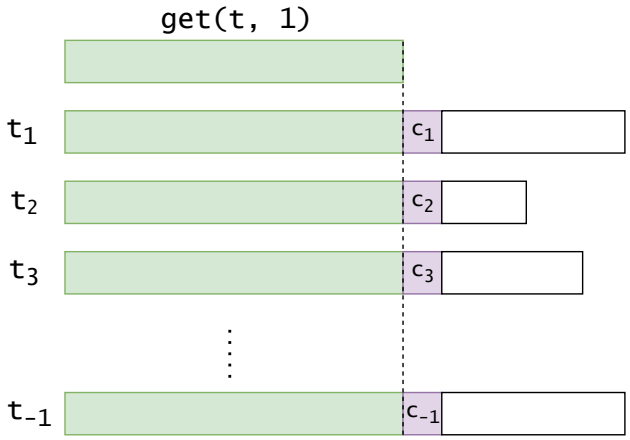


Figure 14: The first sz(get(t, 1)) characters have to be identical. However, $sh(g(t_i, 1)) \neq sh(g(t_j, 1))$. If they would be equal, t_i and t_j would be united post-compression, meaning that t_j's existence wouldn't affect the number of added tokens by searching for t.

The earlier the difference occurs between any t_i and t_j, the more copies of $get(t, 1)$ can be crammed into s.

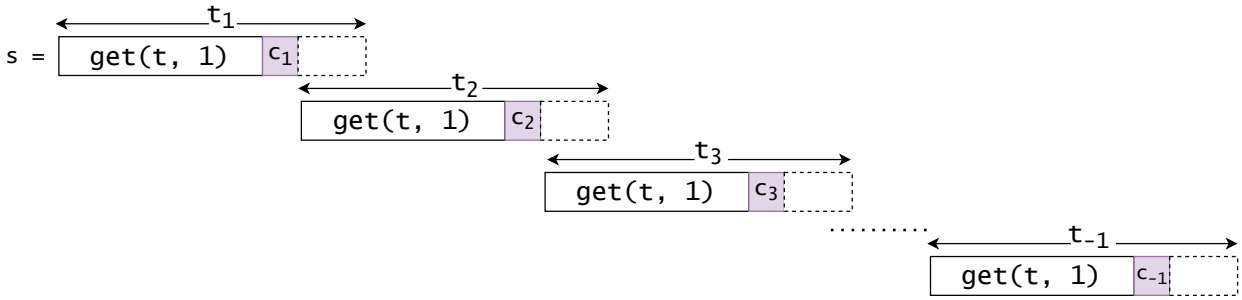


Figure 15: Since $|\ \Sigma\ |$ has been chosen to be n, we can produce the difference as early as possible with the characters $\{c_1, c_2, .., c_{-1}\}$ pairwise distinct.

We will suppose that none of $c_1, c_2, .., c_{-1}$ can be found anywhere else in s. We will also suppose that $NT(m)$ will be bounded in the worst case scenario if s is built like in figure 15:

$$s = +_{i=1}^{n/(sz(g(t,1))+1)}(g(t, 1) + c_i)$$

Let $get(t, 1)$ be a "period" of s. Let $2^M = sz(get(t, 1))$. We are interested in how many $c_i \in sh(g(t, 1))$.

Ba) if there are at least 2 cs in $sh(g(t, 1))$, then let the first 2 be c_{i+1} and c_{i+2}.

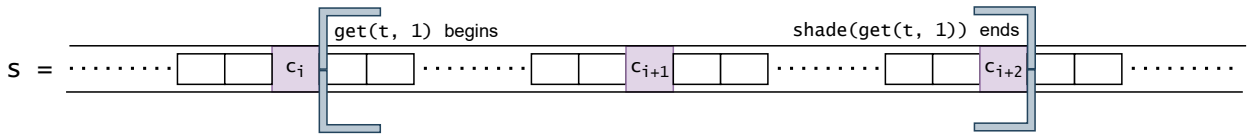


Figure 16: We want to prove that $c_{i+1} \in g(t, 1)$. Worst case, t immediately begins after c_i, and $sh(g(t, 1))$ ends exactly in c_{i+2}.

Since $sz(g(t, 1)) = 1 + sz(sh(g(t, 1))) - sz(g(t, 1))$, and the distance $dist[c_i..c_{i+1}) = dist(c_{i+1}..c_{i+2}] \Rightarrow g(t, 1)$ ends as early as $dist[c_i..c_{i+1}) + 1$ from c_i, that is exactly in c_{i+1}.

Knowing that $c_{i+1} \in g(t,1)$, and c_{i+1} is unique in $s \Rightarrow sub(s, get(t,1)) = 1$, same as case A).

Bb) if there is exactly one $c_i \in sh(g(t,1))$.

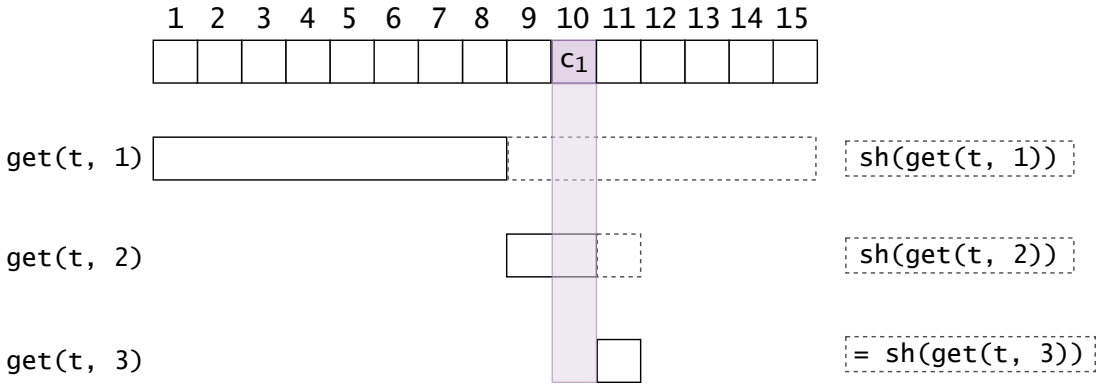


Figure 17: Then t will begin in a "period", and for some time $get(t,j)$ won't touch c_i, but $sh(g(t,j))$ will touch it.

We will have the following course of actions:

- $c_i \in sh(g(t,1)) \setminus g(t,1)$
- $c_i \in sh(g(t,2)) \setminus g(t,2)$
- ...
- $c_i \in sh(g(t,j)) \setminus g(t,j)$
- Then either:
  - $j = popc(sz(t))$, meaning t ended without $c_i \in t$.
  - $c_i \notin sh(g(t,j+1)) \setminus g(t,j+1) \Rightarrow c_i \in g(t,j+1) \Rightarrow sub(s, g(t,j+1)) = 1$, so same as case A).

Therefore, we are only interested in $g(t,1), .., g(t,j)$ for which $c_i \in sh(g(t,j)) \setminus g(t,j) \, \forall z \in [1..j]$. After we exhaust these variants, we either have finished t, or go in case A).

Let $d[0..M]$ be an array. d_j represents the sum of contributions of each substring t that has $g(t,1)$ completely in a certain "period" of s (with $2^j = sz(g(t,1))$). The contribution of t represents how many tokens can it generate before any part of him gets out of that "period". (t's parts are $g(t,1), g(t,2), ...$, and getting out means having some $g(t,z) \ni c_i$).

Bb1) if $sz(g(t,1)) = 1$, then t such that $c_i \in sh(g(t,1)) \setminus g(t,1)$, since $sh(g(t,1)) = g(t,1)$, and the complement would be empty. Therefore, $d_0 = 0$.

Bb2) if $sz(g(t,1)) = 2$, then $\exists 1$ variant for t such that $c_i \in sh(g(t,1)) \setminus g(t,1)$:

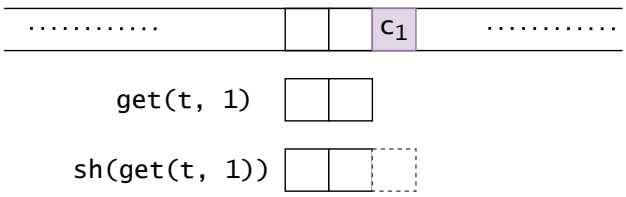


Figure 18: $g(t,1)$ has to end immediately before c_i.

We have in the DAG $\frac{n}{2^M+1}$ nodes valued with $g(t,1)$ that cannot be compressed:

- If a substring t' would be introduced in ts, with $g(t', 1) = g(t, 1)$, at most another $n/2^M$ tokens would be added. (considering that the next character after t' is c_i. Adding another t'' with the same properties won't increase NT anymore)
- If we have other t's with $g(t', 1) = g(t, 1)$ found inside the period, then $c_i \notin sh(g(t, 1))$. We will treat this in case Bc).

$$d_1 = \frac{n}{2^M}.$$

Bb3) if $sz(g(t, 1)) = 4$:

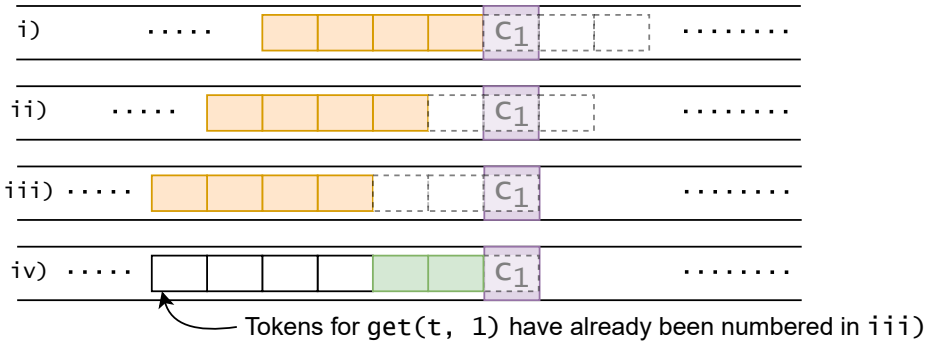


Figure 19: For i) and ii), we cannot add a chunk that has a length equal to a power of 2 without touching c_i. For iii) we choose not to add one. Case iv) is the same as iii) only that we have chosen to add another chunk.

$$d_2 = \frac{n}{2^M}(3 + 1).$$

If $sz(g(t, 1)) = 2^z$, then the shade extends for another 2^{z-1} characters. It begins directly over c_i, and shifts back to the beginning of s. Therefore, there are $2^z - 1$ ways to choose the starting position of the largest chunk (of size $sz(g(t, 1))$).

Bb4) if $sz(g(t, 1)) = 8$:

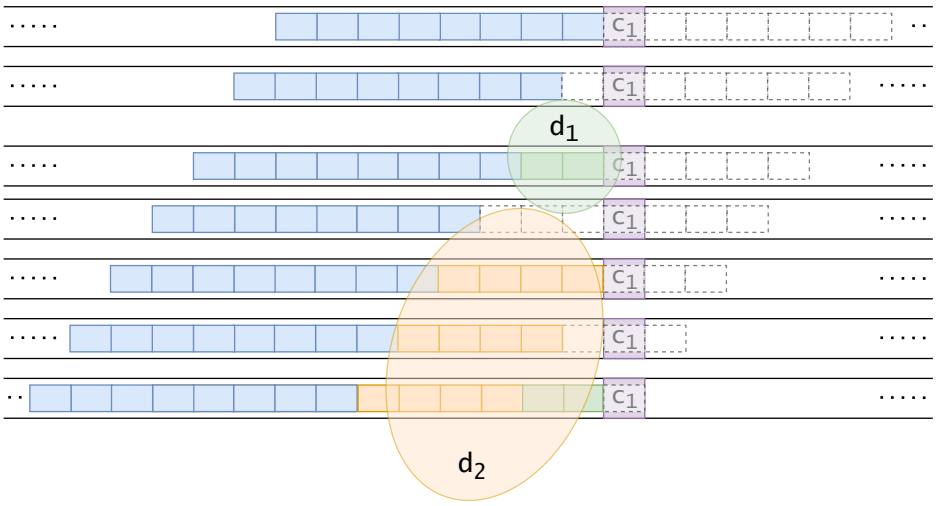


Figure 20: $d_3 = n/2^M(2^3 - 1) + d_2 + d_1 + d_0$.

When we compute d_z, we must include the combinations created by $d_{z_1}, d_{z-2}, .., d_0$, which account for ways of continuing the first chunk.

$\mathrm{d}_0 = 0$.
$\mathrm{d}_z = \frac{n}{2^M}(2^z - 1) + d_{z-1} + d_{z-2} + .. + d_0 \; \forall \; z \in [1..M-1]$.
$\mathrm{d}_M = \frac{n}{2^M}$.

$\mathrm{d}_1 < \frac{n}{2^M} \cdot 2^1$.
$\mathrm{d}_2 < \frac{n}{2^M}(2^2 + 2^1)$.
$\mathrm{d}_3 < \frac{n}{2^M}(2^3 + 2^2 + 2 \cdot 2^1)$.
...
$\mathrm{d}_z < \frac{n}{2^M}(1 \cdot 2^z + 1 \cdot 2^{z-1} + 2 \cdot 2^{z-2} + 4 \cdot 2^{z-3} + ... + 2^{z-2} \cdot 2^1) = \frac{n}{2^M}[2^z + 2^{z-1}(z-1)]$

$$\Rightarrow d_z < \frac{n}{2^M} 2^{z-1}(z+1).$$

$$\Rightarrow \sum_{i=1}^{M} d_i = \frac{n}{2^{M+1}} \sum_{i=1}^{M-1} 2^i i + \frac{n}{2^M} \sum_{i=1}^{M-1} 2^{i-1} + 1.$$

$$\frac{n}{2^M} \sum_{i=1}^{M-1} 2^{i-1} + 1 = \frac{n}{2^M}(2^{M-1} - 1) + 1 < \frac{n}{2^M} 2^{M-1} = \frac{n}{2}.$$

From the appendix A, we can bound $\sum_{i=1}^{M-1} 2^i i < 2^M M$.

$$\Rightarrow \sum_{i=1}^{M} d_i < \frac{n}{2^{M+1}} 2^M M + \frac{n}{2} = \frac{nM}{2} + \frac{n}{2} \leq \frac{n}{2} + \frac{n}{2} \log_2 n.$$

So the NT that can result from the chains of chunks from a specific "period", so that each chunk's shade covers a c_i while they don't is $< \frac{n}{2} + \frac{n}{2} \log_2 n$.

Bc) if there is no $c_i \in sh(g(t,1))$, then t is completely compressed. The ts that enter this case live in a different world, encased entirely within a "period", since $sh(g(t,1))$ doesn't exit it. The different world can be modeled recursively to have the same characteristics as the general world: a "period", and unique delimiting characters.

The length of the smaller "period" must be at most $1/2 \cdot 2^M$. Otherwise, its shade would exit its "world". Buna

After recursively making the "periods" smaller, the total NT bias from both cases Bb) and Bc) is:

$$NT < \frac{n}{2} + \frac{n}{2} \log_2 n + \frac{n}{4} + \frac{n}{4} \log_2 \frac{n}{2} + \frac{n}{8} + \frac{n}{8} \log_2 \frac{n}{4} + ... < n + n \log_2 n.$$

Therefore, after going over the bias of $n + n \log_2 n$, we have to go in the case Ba), meaning that the function NT's slope would be limited onwards to 1.

$$\Rightarrow NT(m) \in O(n \log n + m).$$

Now, the total complexity of the algorithm is $O(n \log^2 n + m \log n)$, meaning that if $m \in O(n \log n)$, then the total complexity becomes $O(n \log^2 n)$.

8 General Usage

Through practically running the algorithm, we aim to:

- See if there may be a tighter bound for the NT function, compared to the one shown in section 7.2;
- Profile the algorithm. Understand the overhead caused by the precomputation phase, or handling the trie.

8.1 Better NT bound

We want to generate a suite of tests. We fix the size of s as n, then we variate m, trying to see how high $NT(m)$ can get.

m can take values between 1 and $n(n+1)(n+2)/6$ (i.e. including all substrings of s into ts).

We have chosen the following 29 probability distributions, which we will use to randomly generate s.

Probability Distribution
$\mid \Sigma \mid = i, p_1 = p_2 = .. = p_i = 1/i \ \forall i \in [1..26]$. uniform distribution for variable sizes of the alphabet.
$\mid \Sigma \mid = 2, p_1 = 0.99, p_2 = 0.01$
$\mid \Sigma \mid = 2, p_1 = 0.9, p_2 = 0.1$
$\mid \Sigma \mid = 5, p = \{0.6, 0.2, 0.1, 0.05, 0.05\}$

To generate some s, we first choose some probability distribution from the ones just listed, then we randomly pick each character forming s following the probability array.

We then fix a value of m and try to populate ts with substrings of s. We have two ways of populating: randomly choosing the substrings to add to ts, or adding them in a greedy manner (resembling the way of choosing the substrings to add in ts in section 7.1).

Experimentally, the greedy method gave higher values for $NT(m)$:

```
remainingLengths = {1, 2, .., n}
hashes = {}
ts = {}
while m > 0:
    r = remainingLengths.min()
    while r <= n:
        remainingLengths.erase(r)
        for i in [1..n-r+1]: #now add all substrings of length r.
            j = i + r - 1
            if hash(s[i..j]) not in hashes:
                if m < r:
                    return ts #we can no longer add characters to ts, we have finished filling it.
                m -= r
                hashes.insert(hash(s[i..j]))
                ts.insert(s[i..j])
        r = r * 2 + 1
return ts
```

We know a point that has to be on the graph of the NT function. If $\mid \Sigma \mid = n$, and all characters of s are pairwise distinct, then if we were to include all substrings of s into ts, we would generate all of the possible tokens during the search (compression wouldn't help). Therefore:

$$NT(n(n+1)(n+2)/6) = n(n+1)/2.$$

We want the NT function to depend on both n and m, something like $NT(m) = ct \cdot n^{\alpha} m^{\beta}$. The point we want to fit on NT's graph would lead to $3\beta + \alpha = 2$.

The easiest idea we could try would be $\beta = 1/2 \Rightarrow \alpha = 1/2 \Rightarrow NT(m) = ct\sqrt{nm}$ (i.e. replacing the arithmetic mean from section 7.2 with the geometric mean). We now have to figure the constant ct:

$$NT\Big(\frac{n(n+1)(n+2)}{6}\Big) = ct\sqrt{\frac{n^2(n+1)(n+2)}{6}} = \frac{n(n+1)}{2}.$$

$$\Rightarrow ct^2 \frac{n+2}{6} = \frac{n+1}{4} \Rightarrow ct^2 = \frac{3}{2} \cdot \frac{n+1}{n+2} < \frac{3}{2} \Rightarrow ct < 1.225.$$

We will also add the bias $O(n \log n)$ from section 7.2.

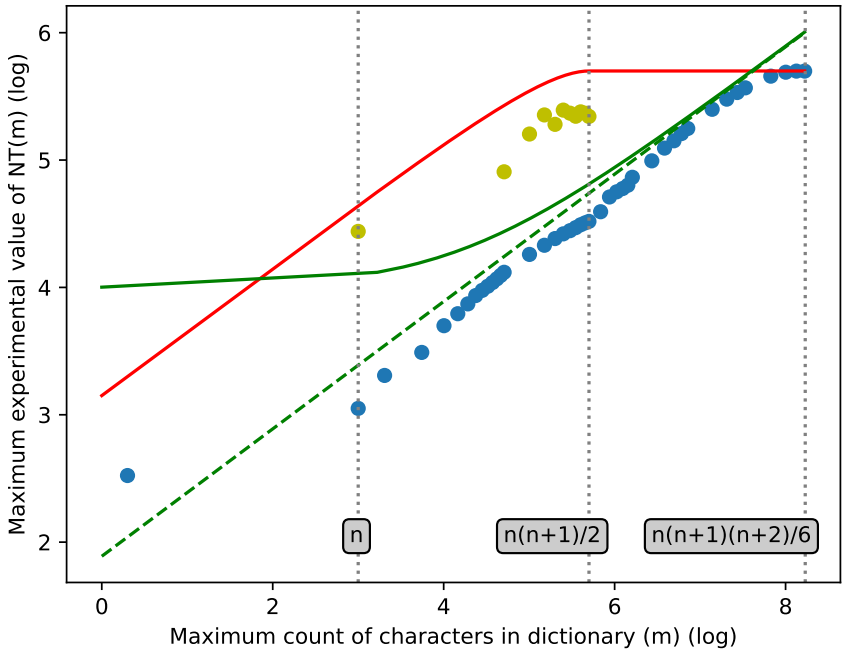


Figure 21: We have fixed $n = 1000$. Both axes are in logarithmic (base 10) scale.

- The blue dots represent the maximum NT experimentally obtained for some m (with compression).
- The goldenrod dots represent the maximum NT experimentally obtained without compression (the $p = \{1\}$ distribution was specifically left out).
- The red line represents the NT function's bound if no compression is done - $O(\min(n\sqrt{m}, n^2))$.
- The green line represents the experimental bound for NT: $n \log_2 n + 2 \cdot 1.225 \cdot \sqrt{nm}$.
- The dotted green line represents the unbiased bound: $2 \cdot 1.225 \cdot \sqrt{nm}$.

We haven't been able to breach the experimental bound $NT(m) \leq n \log_2 n + 2 \cdot 1.225 \cdot \sqrt{nm}$ (for neither $n = 1000$ or 10^5), meaning that it may be possible for the algorithm complexity to be $O(n \log^2 n + m + \sqrt{nm} \log n)$.

If $m \in O(n \log^2 n)$, it may be possible for the whole algorithm complexity to be $O(n \log^2 n)$.

8.2 Algorithm Comparison

We want to test how our algorithm fares on specific tests against other popular implementations of the problem.

We will test implementations of:

- The presented algorithm, offline, with DAG compression. [24]
- The Aho-Corasick automaton. [25]
- The Suffix automaton. [26]
- The Suffix tree. [27]
- The Suffix array. [28]

The following folder [29] contains instructions on how to generate the tests used in this section.

From our complexity analysis and profiling, we know that:

- Precomputation takes a noticeable chunk of time, $O(n \log^2 n)$, with a big constant.

- We can expect the search phase to take $O(\sqrt{nm}\log n)$.

As a result, we want to generate tests that have a small value for n, and a big value for m.

The following batches were run on the Polygon servers (`Intel i3-8100, 6MB` cache), with a 15s time limit per test, and a 1 GB memory limit per process.

n	**m**	**type**	**this(s)**		**AC(s)**		**suffTree(s)**		**suffAuto(s)**		**suffArray(s)**	
			Med	*Max*	*Med*	*Max*	*Med*	*Max*	*Med*	*Max*	*Med*	*Max*
10^3	10^8	00	3.0	3.3	6.1	6.9	0.3	0.3	0.5	0.6	0.6	1.0
10^4	$2 \cdot 10^8$	01	8.4	9.0		TL	1.5	1.8	1.9	2.1	2.7	4.0
10^4	$2 \cdot 10^8$	11	7.0	7.4		TL	1.0	1.4	1.4	1.5	3.1	3.5
10^4	$2 \cdot 10^8$	20	5.1	5.4		RE	0.2	0.5	1.7	1.8	1.5	1.6
10^4	$2 \cdot 10^8$	21	5.3	5.6		TL	0.4	0.6	1.0	1.1	3.1	3.2
10^5	$2 \cdot 10^8$	00	9.3	10.4		RE	1.0	5.4	1.5	2.9	1.6	8.9
10^5	$2 \cdot 10^8$	10	7.7	8.2		RE	0.5	6.5	1.2	3.1	2.0	8.9
10^5	$2 \cdot 10^8$	20	5.3	5.4		RE	0.2	6.3	2.1	4.2	2.5	8.9

Each testing batch is characterized by the median time ("*Med*") an algorithm scored on a test from the batch, and by the maximum time ("*Max*"). *T*L signifies that the median/maximum test from the batch finished in over 15s. *R*E signifies that `std::bad_alloc` was thrown for the median/maximum test from the batch.

One testing batch contains a fixed n and m, and one generated test for each distribution showed in section 8.1 (so 29 tests in total).

In order to efficiently test for large values of m, all words in the dictionary are guaranteed to be found at least once in s, meaning that all dictionary words t can be instead given in the form of two integers l, r: $s[l..r] = t$.

One test can be generated in the following ways. The first digit of the $type$ decides how s is built:

- 0 randomly adds characters according to the chosen distribution.
- 1 randomly chooses a "period" like in section 7.2, then adds some random characters decided by the distribution before repeating again the "period".
- 2 randomly picks two small strings u, v, then builds them up in a Fibonacci manner: $u, v \leftarrow v, u + v$. Stop when $sz(v) \geq n$, then let s be the first n characters of v. The large number of squares in s would make our algorithm's compression phase more efficient.

The last digit of the $type$ decides how ts is built:

- 0 uses the generator from the snippet in section 8.1.
- 1 uses the same generator as 0, but the way of choosing the next substring length changes from $r = 2r + 1$ to $r = r + 1$.

The following batches were run on the department's server (`Intel Xeon E5-2640 2.60GHz, 20 MB` cache), with no set time limit, and a 4 GB memory limit per process. *ML* signifies that the process went over the memory limit.

n	**m**	**type**	**distribution**	**this(s)**	**suffArray(s)**	**suffAuto(s)**	**suffTree(s)**
10^5	$2 \cdot 10^9$	01	1	15.9	37.5	9.4	15.7
10^5	$2 \cdot 10^9$	21	15	17.7	17.1	14.8	2.0
10^5	$2 \cdot 10^9$	01	27	30.4	23.3	10.4	23.2
$5 \cdot 10^5$	$2 \cdot 10^9$	01	27	40.1	32.4	12.1	26.0
$5 \cdot 10^5$	$5 \cdot 10^9$	01	27	ML	74.3	29.3	64.1
$5 \cdot 10^5$	10^{10}	21	3	104.9	132.5	61.9	11.7

We can notice that under some not very restrictive requirements (small n, big m), the runtime gap between the presented algorithm and other well written implementations of popular algorithms is of decent size, sometimes it even being in our favour (especially against Aho-Corasick).

If m is considerably large, our algorithm may even do better than a suffix implementation. This usually happens when s is built in a Fibonacci manner, allowing the compression to be more efficient than expected.

9 Other Applications

9.1 Number of distinct substrings

Inherently, the presented algorithm performs badly for such a task, since it has to assume that the trie will contain any hash chain associated to a substring of s. If $\mid \Sigma \mid = n$, then the DAG compression would do nothing and we would end up generating one token for each substring for a total runtime of $O(n^2 \log^2 n)$.

Greedy algorithms that can be applied directly on the DAG generally fail for strings that generate DAGs with 4 or more levels (for example, attribute a score to each node in the compressed DAG, initially 0. Do an inverse propagation: for each node, add its score plus one to each of its parents. The final result is the sum of all scores plus the number of distinct nodes in the DAG).

However, the trie search can be optimized for such a task. If a trie node (different from the root) has only one token in it, we shouldn't propagate it any further, because we know that the associated string of the token (u) is unique in s. As a consequence, every substring v of s that has u as a prefix is also unique.

If we know the following information about the token:

- How long is the string on the trie node's "father edge" (for example, if the token resided in the trie node at the end of the chain $* \xrightarrow{abba} * \xrightarrow{bc} *$, we would be interested in $sz(bc) = 2$). Let this be X.
- Where does the string on the trie node's "father edge" end in s (for example, if $s = dabba\mathbf{bc}da$, bc would end exactly at the 7th position, indexing from 1. Let this be Y.

Then the contribution of the token and its would-be subtree nodes in the trie is $1 + \min(X - 1, n - Y)$.

Let $NT'(n)$ be the maximum number of tokens (not counting the root token) that can exist in the associated trie of a string of length n. We will only count the tokens that end up propagating (i.e. are not the only ones in their respective nodes). We want to prove that $NT'(n) \leq n + n \log_2 n$.

The proof closely resembles the one shown in section 7.2. If we went down a path in the trie enough that our current prefix would contain one of the unique characters c_i, then we wouldn't care about that path anymore, because $\triangle NT / \triangle m \leq 1$. Similarly, here we don't care about that path because c_i makes the current prefix unique in the whole string, so we stop propagating.

As a result, we can bound $NT'(n)$ by borrowing the bias from section 7.2 and ignoring the slope. The complexity of the trie search would be $O(NT'(n) \log n) = O(n \log^2 n)$. The total time complexity of the algorithm would also be $O(n \log^2 n)$. [30]

Computing the substring frequency distribution ($\forall i \in [1, n]$, we want $d_i =$ how many distinct substrings of s are of length i?) in $O(n \log^2 n)$ can be now done with little effort. For every token that is alone in a trie node, let Z be the length of the substring formed with the strings on the chain. For $* \xrightarrow{abba} * \xrightarrow{bc} * \Rightarrow Z = 6$. The contribution of the token and its would-be subtree would be represented by adding 1 to all elements in the interval $d_{[Z..Z+\min(X-1,n-Y)]}$, doable in $O(1)$ with partial sums. [31]

9.2 Approximate search

If we are tasked to solve an easier variant of the problem: check $\forall t \in ts$ if t appears in s - we can provide a fast online approximate solution. If this solution says that some t doesn't appear in s, it is always correct. If it says t appears in s, it may be wrong.

- Use a more aggressive DAG compression optimization: *if two nodes have the same value, unite them*. This optimization isn't correct, since some previously non-existent hash chains can be created post-compression.
- However, if we now want to find t in s, there is at most 1 hash chain in the compressed DAG describing t. Binary search through all DAG node values to find the head of the hash chain in $O(\log(n \log n))$, then go through the chain in $O(\log^2 n)$. Total query complexity is $O(sz(t) + \log^2 n)$. [32]

9.3 Others

- Finding the first occurrence of a substring in s. For each node in the compressed DAG, store not only its leverage, but the lowest index which starts a chain which ends there as well. The DAG is originally constructed so that indexes with a lower value begin first.
- An online variant is possible, even if it is more complicated. Build the trie as the questions come along. When solving a query, firstly go through the trie. If the current query has a common prefix with a previous query, it isn't worth recomputing the common path again. Use the tokens left at the end of the longest common path to continue the search. Its complexity is worse, since we may attempt to divide the same tokens found in a currently terminal node from the trie multiple times. ([33] - implements count, and first occurrence queries)
- Finding all the occurrences for the strings in the dictionary. Without DAG compression, figure out the ending positions of the occurrences from the tokens in the terminal nodes in the trie. Its trie search complexity is $O(\min(n\sqrt{m} \log n,\ n^2 \log n))$, since it cannot use compression. [34]

10 Conclusion

The presented algorithm is useful for pattern matching a dictionary against a string: finding the number of occurrences, finding the position of the first occurrence, along with other applications - counting the number of different substrings, computing the substring frequency distribution, just finding the occurrences, or doing an approximate search for *YES/NO* queries.

If there is a limit for the dictionary's size in characters (m), it can achieve a complexity of $O(n \log^2 n + m \log n)$. Experimentally, a complexity of $O(n \log^2 n + m + \sqrt{nm} \log n)$ may be possible.

In practice, for a value of m comparable to n, the algorithm is slower than its standard counterparts, but if m is large enough, it performs better than an Aho-Corasick implementation. If m is even larger, it may even perform better than some suffix implementations on some cases.

Also, it has an edge when all of the string's $O(n^2)$ substrings can be represented only as $O(n)$ distinct ones, and the input is given in compressed form, or directly in hash chains (for example $s = aa..a$, $ababab..$, $abcabcabc..$ or $abab..abc$). For the edge cases, the running complexity of the search phase is $O(n \log^2 n)$ (the constants being 1, 4, 9, 9 for the examples).

In contrast, if any standard method were to just read the $O(n)$ elements of the dictionary, they would require $O(n^2)$ time to do so, while this algorithm just needs $O(n \log n)$ hashes to build the respective chains for the queries.

11 Implementation References

- Offline algorithm without DAG compression. [34]
- Offline algorithm. Uncompressed input reader. [21]
- Class for reading the compressed form of the dictionary. [22]
- Online algorithm. Also supports queries for position of first occurrence. [33]
- Offline algorithm modified to directly read hash chains. Showcases runtime speeds for $s = aa..a$, $ababab..$, $abcdabcdabcd...$, $abab...abc$, $aaaa..aabaaa$. [23]

- Number of distinct substrings. [30]
- Substring frequency distribution. [31]
- Approximate solution for *YES/NO* queries. [32]

Acknowledgments

We would like to thank Robert Popovici, Daniel Mitra, Luca Furtună and Matei Barbu for proofreading this paper, and offering very useful feedback. We would also like to thank the team behind CSES for providing a great problem set, and the team behind Polygon for easing the process of testing various implementations.

A DAG compression bound inequality

We want to bound $\sum_{i=1}^{N} 2^i i$.

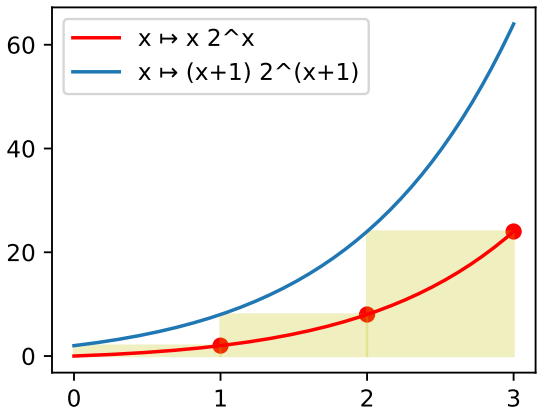


Figure 22: The area contained in the rectangles represent the sum that we are trying to bound. The area contained under the graph of $x \mapsto 2^x x$ is smaller than the area we want to bound, but if we shift the graph of the function to the left by one unit, the new area is bigger than the one we want to bound.

$$\Rightarrow \int_0^N 2^{x+1}(x+1)dx > \sum_{i=1}^{N} 2^i i.$$

$$\int_0^N 2^{x+1}(x+1)dx = \int_1^{N+1} 2^x x dx = \frac{1}{\ln 2}\int_1^{N+1} (2^x)' x\, dx = \frac{1}{\ln 2}\left(2^x x \,|_1^{N+1} - \int_1^{N+1} 2^x\, dx\right)$$

$$= \frac{1}{\ln 2}\left(2^{N+1}(N+1) - \frac{2^{N+1}}{\ln 2} - 2 + \frac{2}{\ln 2}\right) < 2^{N+1}(N+1).$$

References

[1] Snort.

[2] Suricata.

[3] Clamav.

[4] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A fast multi-pattern regex matcher for modern cpus. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 631–648, USA, 2019. USENIX Association.

[5] Google re2.

[6] Byungkwon Choi, Jongwook Chae, Muhammad Jamshed, KyoungSoo Park, and Dongsu Han. Dfc: Accelerating string pattern matching for network applications. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 551–565, USA, 2016. USENIX Association.

[7] Kargus.

[8] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, jun 1975.

[9] Esko Ukkonen. On-line construction of suffix treess. *Algorithmica*, 1995.

[10] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, page 319–327, USA, 1990. Society for Industrial and Applied Mathematics.

[11] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, and R. McConnell. Building the minimal dfa for the set of all subwords of a word on-line in linear time. In Jan Paredaens, editor, *Automata, Languages and Programming*, pages 109–118, Berlin, Heidelberg, 1984. Springer Berlin Heidelberg.

[12] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[13] Jean-Paul Allouche and Jeffrey Shallit. The ubiquitous prouhet-thue-morse sequence. In C. Ding, T. Helleseth, and H. Niederreiter, editors, *Sequences and their Applications*, pages 1–16, London, 1999. Springer London.

[14] Abraham D. Flaxman and Bartosz Przydatek. Solving medium-density subset sum problems in expected polynomial time. In *Proc. 22nd Symposium on Theoretical Aspects of Computer Science — STACS 2005*, volume 3404 of *Lecture Notes in Computer Science*, pages 305–314. Springer-Verlag, 2 2005.

[15] Daniel Rutschmann. On the mathematics behind rolling hashes and anti-hash tests.

[16] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, oct 1980.

[17] David Blackman and Sebastiano Vigna. Scrambled linear pseudorandom number generators. *ACM Trans. Math. Softw.*, 47(4), sep 2021.

[18] Xoshiro256++ implementation.

[19] Guy L. Steele, Doug Lea, and Christine H. Flood. Fast splittable pseudorandom number generators. *SIGPLAN Not.*, 49(10):453–472, oct 2014.

[20] Splitmix64 implementation.