A simple grammar-based index for finding approximately longest common substrings[⋆]

Travis Gagie[1,3][0000−0003−3689−327X], Sana Kashgouli[1][0009−0008−1583−2914], and Gonzalo Navarro[2,3][0000−0002−2286−741X]

[1] Faculty of Computer Science, Dalhousie University, Halifax, Canada
[2] Dept. of Computer Science, University of Chile, Santiago, Chile
[3] CeBiB — Center for Biotechnology and Bioengineering, Chile

Abstract. We show how, given positive constants ϵ and δ, and an α-balanced straight-line program with g rules for a text $T[1..n]$, we can build an $O(g)$-space index that, given a pattern $P[1..m]$, in $O(m \log^{\delta} g)$ time finds w.h.p. a substring of P that occurs in T and whose length is at least a $(1-\epsilon)$ fraction of the longest common substring of P and T. The correctness can be ensured within the same expected query time.

Keywords: Grammar-based indexing · Approximately longest common substrings · alpha-balanced grammars

1 Introduction

Recent years have witnessed a sustained effort for indexing highly repetitive text collections within compressed space and supporting exact pattern matching [10, 11]. Exact pattern matching is however insufficient in some applications. In Bioinformatics, for example when storing repetitive collections formed by genomes of the same species, matching strings is rarely useful. Instead, one may be interested in finding long substrings of a string that appear in the sequence collections, to find for example conserved regions of a genome in a population.

The research on matching the longest possible substrings using these indices is scarce, however. A recent result [12] finds all the maximal exact matches (MEMs) of a pattern $P[1..m]$ in a text $T[1..n]$ that is indexed with a grammar. By building on an arbitrary (run-length) context-free grammar of size g, the index is of size $O(g)$ and finds all the MEMs in time $O(m^2 \log^{\delta} g)$, for any constant $\delta > 0$ (see also [6]). If the grammar is of a kind called locally consistent, the time improves to $O(m \log m(\log m + \log^{\delta} n))$. Other results (see [3, 12]) require larger indices.

In this paper we consider the simpler problem of finding one longest common substring between P and T (i.e., a longest MEM). Further, we are satisfied with a common substring whose length is at least $1-\epsilon$ times the longest one, for some fixed $0 < \epsilon < 1$. We show that, on α-balanced grammars [4, 14], this can be solved with high probability in time $O(m \log^{\delta} g)$ for any fixed constant $\delta > 0$. The correctness of the answer can be ensured in $O(m \log^{\delta} g)$ expected time.

[⋆] Funded in part by NSERC grant RGPIN-07185-2020; NSF/BIO grant DBI-2029552; NIH/NHGRI grant R01HG011392; and Basal Funds FB0001, ANID, Chile.

2 Preliminaries

Our index uses grammar-based compression, which compresses a text $T[1..n]$ by building and storing a context-free grammar that generates only T [9]. We focus in particular on *straight-line programs (SLPs)*, where each rule is of the form $X \rightarrow Y\,Z$, where Y and Z are terminals or nonterminals (called symbols). If T is repetitive, then it can be represented with an SLP of g rules, with $g \ll n$. Grammar-based indices [5] aim to use space linear in the grammar size while offering indexed searches for patterns $P[1..m]$, that is, enumerating all the positions in T where P occurs. Following Charikar et al. [4], we write $\langle X \rangle$ and $[X]$ to denote the string symbol X expands to and the length of that expansion, respectively. Our work builds on α-balanced SLPs, defined next. There exist practical constructions of small α-balanced grammars from repetitive texts [14].

Definition 1 ([4]). *For a constant $0 < \alpha \leq 1/2$, an SLP is said to be* α-balanced *if, for every rule $X \rightarrow Y\,Z$, it holds that*

$$\frac{\alpha}{1-\alpha} \leq \frac{[Y]}{[Z]} \leq \frac{1-\alpha}{\alpha}\,.$$

3 Data structure

Our data structure is built from an α-balanced SLP G. For each nonterminal X in this SLP, the structure stores a set of prefixes and suffixes of $\langle X \rangle$, of exponentially increasing lengths. Those are called prefix and suffix blocks, respectively.

Definition 2. *Let X be a symbol in G and fix a constant $0 < \epsilon < 1$. Then, for each $0 \leq k \leq \log_{1/(1-\epsilon)}[X]$, we call $\langle X \rangle[1..\lceil 1/(1-\epsilon)^k \rceil]$ a* prefix block *and $\langle X \rangle[[X]-\lceil 1/(1-\epsilon)^k \rceil+1..[X]]$ a* suffix block.

Precisely, given ϵ, consider the following sets:

$$\begin{aligned}
\mathcal{X} &= \{\langle X \rangle,\, X \text{ is a symbol in } G\},\\
\mathcal{B}_{\text{pref}} &= \{B,\, B \text{ is a prefix block of a symbol } X \text{ in } G\},\\
\mathcal{B}_{\text{suff}} &= \{B,\, B \text{ is a suffix block of a symbol } X \text{ in } G\}.
\end{aligned}$$

For every prefix block $B \in \mathcal{B}_{\text{pref}}$, we compute B's Karp-Rabin [8] hash $h(B)$ and the lexicographic range $[s_B, e_B]$ of the strings in $\mathcal{X}$ that are prefixed by B. We store each pair $(h(B), [s_B, e_B])$ in a perfect hash table H_{pref}, with $h(B)$ as the key and $[s_B, e_B]$ as the value. Symmetrically, for each suffix block $B \in \mathcal{B}_{\text{suff}}$, we compute B's Karp-Rabin hash $h(B)$ and the co-lexicographic range $[s_B, e_B]$ of the strings in $\mathcal{X}$ that are suffixed by B, storing each pair $(h(B), [s_B, e_B])$ in a perfect hash table H_{suff} with $h(B)$ as the key and $[s_B, e_B]$ as the value. The Karp-Rabin hash function $h(B)$ is designed to have no collision between substrings of T, which can be built in $O(n \log n)$ expected time [1]. With low probability, however, there may be collisions between substrings of a pattern P and blocks of T.

We now show that $|\mathcal{B}_{\mathrm{pref}}|$ and $|\mathcal{B}_{\mathrm{suff}}|$ are $O(g)$, and therefore our hash tables are of size $O(g)$ as well.

Lemma 1. *If $X \rightarrow Y\,Z$ is a rule in G, then only $O(1)$ prefix blocks $B \in \mathcal{B}_{\mathrm{pref}}$ are prefixes of $\langle X \rangle$ but not of $\langle Y \rangle$, and only $O(1)$ suffix blocks $B \in \mathcal{B}_{\mathrm{suff}}$ are suffixes of $\langle X \rangle$ but not of $\langle Z \rangle$.*

Proof. By Def. 1, we have

$$[X] \;=\; [Y] + [Z] \;\leq\; \left(1 + \frac{1-\alpha}{\alpha}\right) \cdot [Y] \;=\; \frac{[Y]}{\alpha}\,,$$

so the number of prefix blocks that are prefixes of $\langle X \rangle$ but not $\langle Y \rangle$ is, by Def. 2,

$$\log_{\frac{1}{1-\epsilon}} [X] - \log_{\frac{1}{1-\epsilon}} [Y] + O(1) \;=\; \log_{\frac{1}{1-\epsilon}} \frac{[X]}{[Y]} + O(1) \;\leq\; \log_{\frac{1}{1-\epsilon}} \frac{1}{\alpha} + O(1) \;=\; O(1)\,.$$

Symmetrically, because $[X] \leq [Z]/\alpha$, the number of suffix blocks that are suffixes of $\langle X \rangle$ but not of $\langle Z \rangle$ is $O(1)$. $\square$

Corollary 1. *The number of prefix and suffix blocks is $|\mathcal{B}_{\mathrm{pref}}| + |\mathcal{B}_{\mathrm{suff}}| = O(g)$.*

Proof. By Lemma 1, each symbol X of G, of which there are g, contributes $O(1)$ prefix blocks to $\mathcal{B}_{\mathrm{pref}}$ and $O(1)$ suffix blocks to $\mathcal{B}_{\mathrm{suff}}$. $\square$

The final component of our data structure is a discrete two-dimensional grid $\mathcal{G}$, with one row and one column per element of $\mathcal{X}$. Let

– $X \rightarrow Y\,Z$ be a rule in G,
– $\langle Y \rangle$ have co-lexicographic position i in $\mathcal{X}$, and
– $\langle Z \rangle$ have lexicographic position j in $\mathcal{X}$,

then we set a point at position (i, j) in the grid. We label this point with the position where $\langle Y \rangle$ ends inside an occurrence of $\langle X \rangle$ in T (i.e., if we choose the occurrence $T[a..b] = \langle X \rangle$, then the label of the point is $a + [Y] - 1$). The grid has g points, thus it can be represented in $O(g)$ space and answer range emptiness queries in $O(\log^{\delta} g)$ time, for any constant $\delta > 0$ [2].

Our whole data structure then comprises H_{pref}, H_{suff}, and $\mathcal{G}$, which add up to $O(g)$ space. We note that the values $[s_B, e_B]$ stored in H_{pref} are the lexicographic ranges of grid columns corresponding to strings in $\mathcal{X}$ prefixed with B, and those stored in H_{suff} are the co-lexicographic ranges of grid rows corresponding to strings in $\mathcal{X}$ suffixed with B.

4 Queries

Our searches build on a key result used in all grammar-based indices [5].

Lemma 2. *Let string S, of length $|S| > 1$, appear in T. Then, there is an index $1 \leq p < |S|$ and a point (i, j) in $\mathcal{G}$ such that*

– *i is the co-lexicographic range of a string $\langle Y\rangle \in \mathcal{X}$ suffixed by $S[1..p]$ and*
– *j is the lexicographic range of a string $\langle Z\rangle \in \mathcal{X}$ prefixed by $S[p+1..|S|]$.*

Proof. Note that S appears as a substring of the expansion of the initial symbol and, possibly, of others. If we order the rules $X \rightarrow Y\,Z$ so that Y and Z are listed before X, then the first time S appears as a substring of $\langle X\rangle$, it must appear as the concatenation of a nonempty suffix of $\langle Y\rangle$ and a nonempty prefix of $\langle Z\rangle$. The lemma then follows from the definition of $\mathcal{G}$. □

Now let L be the longest common substring of P and T and assume $|L| > 1$. Per Def. 2, let $k = \lfloor \log_{1/(1-\epsilon)} |L| \rfloor$. We note that

$$\left(\frac{1}{1-\epsilon}\right)^{k} > \left(\frac{1}{1-\epsilon}\right)^{\left(\log_{\frac{1}{1-\epsilon}} |L|\right)-1} = (1-\epsilon)\cdot|L|.$$

Thus, for our purposes, it suffices to find a substring of length $\ell = (1/(1-\epsilon))^k$ of L. By Lemma 2, there exists an index $1 \le p < |L|$ such that $L_Y = L[1..p]$ suffixes some $\langle Y\rangle \in \mathcal{X}$, $L_Z = L[p+1..|L|]$ prefixes some $\langle Z\rangle \in \mathcal{X}$, and there is a rule $X \rightarrow Y\,Z$ in G. Further, let $k_Y = \lfloor \log_{1/(1-\epsilon)} |L_Y| \rfloor$ and $k_Z = \lfloor \log_{1/(1-\epsilon)} |L_Z| \rfloor$. By the same argument above, it follows that

$$\left(\frac{1}{1-\epsilon}\right)^{k_Y} > (1-\epsilon)\cdot|L_Y| \text{ and } \left(\frac{1}{1-\epsilon}\right)^{k_Z} > (1-\epsilon)\cdot|L_Z|.$$

Therefore, it suffices to find a suffix of length $\ell_Y = \lceil (1/(1-\epsilon))^{k_Y} \rceil$ of $\langle Y\rangle$ and a prefix of length $\ell_Z = \lceil (1/(1-\epsilon))^{k_Z} \rceil$ of $\langle Z\rangle$ to form a substring of L of length $\ell_Y + \ell_Z > (1-\epsilon)\cdot(|L_Y| + |L_Z|) = (1-\epsilon)\cdot|L|$, because $L = L_Y \cdot L_Z$.

Per Def. 2, those suffixes $L'_Y = L_Y[|L_Y|-\ell_Y+1..\ell_Y]$ are suffix blocks, and those prefixes $L'_Z = L_Z[1..\ell_Z]$ are prefix blocks, and therefore they are stored in our hash tables. Thus, if we search H_{suff} for L'_Y and retrieve the associated range $[s_Y, e_Y]$, and search H_{pref} for L'_Z and retrieve the associated range $[s_Z, e_Z]$, we will find a point in the (row,column) range $[s_Y, e_Y] \times [s_Z, e_Y]$ of $\mathcal{G}$.

The correctness of Algorithm 1 stems from this discussion. A position of the common substring found is obtained by noticing that, when we assign ℓ in line 12, the string occurs at $P[p-\ell_Y+1..p+\ell_Z]$ and $T[t-\ell_Y+1..t+\ell_Z]$, where t is the label of any point in the grid range.

Since we do not know $|L|$ beforehand, the algorithm tries all the possible values for k_Y and k_Z, which yields a time complexity dominated by $O(m\log^2 m)$ range emptiness queries, that is, $O(m\log^2 m \log^\delta n)$ [2]. We note that, since the hashes are of Karp-Rabin type, we can precompute in $O(m)$ time the hash of every prefix, $h(P[1..p])$, and then we can compute in constant time the hash of every substring of P by operating with the modular inverses of the hashes [13]. If there is a collision we may find a false positive.

Note that Algorithm 1 will find only the empty string if $|L| = 1$, as we assumed $|L| > 1$. In case the algorithm returns zero, we must determine if $|L| = 1$ by checking if some symbol of P appears as a terminal in G; this is easily done with additional $O(m)$ time and $O(g)$ space.

Algorithm 1 The simple algorithm returning an approximation to the length of the longest common substring between T and $P[1..m]$.

```
1: $\ell \leftarrow 0$
2: for $p \leftarrow 1$ to $m$ do
3:     for $k_Y \leftarrow 0$ to $\lfloor \log_{1/(1-\epsilon)} p \rfloor$ do
4:         $\ell_Y \leftarrow \lceil (1/(1-\epsilon))^{k_Y} \rceil$
5:         $[s_Y, e_Y] \leftarrow$ search $H_{\text{suff}}$ for $P[p-\ell_Y+1..p]$
6:         if $[s_Y, e_Y]$ was found then
7:             for $k_Z \leftarrow 0$ to $\lfloor \log_{1/(1-\epsilon)}(m-p) \rfloor$ do
8:                 $\ell_Z \leftarrow \lceil (1/(1-\epsilon))^{k_Z} \rceil$
9:                 $[s_Z, e_Z] \leftarrow$ search $H_{\text{pref}}$ for $P[p+1..p+\ell_Z]$
10:                if $[s_Z, e_Z]$ was found then
11:                    if $\mathcal{G}$ has a point in $[s_Y, e_Y] \times [s_Z, e_Z]$ then
12:                        $\ell \leftarrow \max(\ell, \ell_Y + \ell_Z)$
13: return $\ell$
```

5 Faster queries

We can reduce the time complexity of Algorithm 1 by decreasing the number of combinations (k_Y, k_Z) we explore. The algorithm may try out $\Theta(\log^2 m)$ combinations per value of p, but several of those are redundant. For example, if the range $[s_Y, e_Y] \times [s_Z, e_Z]$ corresponding to the pair (k_Y, k_Z) is empty, then so is the range $[s'_Y, e'_Y] \times [s_Z, e_Z]$ corresponding to $(k_Y + 1, k_Z)$, as well as the range $[s_Y, e_Y] \times [s'_Z, e'_Z]$ corresponding to $(k_Y, k_Z + 1)$. It then suffices to explore *maximal* combinations (k_Y, k_Z). Further redundant work is done among values of p: we may be working on maximal combinations (k_Y, k_Z) that nevertheless yield shorter strings than one we had already obtained with a previous value of p.

To avoid redundant work, we will visit only the combinations (k_Y, k_Z) for which $\ell_Y + \ell_Z > \ell$; recall that ℓ is the maximum length $\ell_Y + \ell_Z$ obtained so far. Therefore, every time we find a nonempty range in $\mathcal{G}$, the value of ℓ increases. We say those combinations are *useful*. The other combinations, where either the searches in H_{pref} or in H_{suff} fail, or they succeed but the resulting range in $\mathcal{G}$ is empty, are *useless*. We will count useful and useless combinations separately.

Since there are only $O(\log^2 m)$ combinations (k_Y, k_Z), there exist $O(\log^2 m)$ different values $\ell_Y + \ell_Z$. Since the value of ℓ never decreases along the process, there are only $O(\log^2 m)$ situations in which a new value of $\ell_Y + \ell_Z$ can increase ℓ. This implies that the total number of useful combinations we visit is $O(\log^2 m)$.

To keep the number of useless combinations low, we will visit the space (k_Y, k_Z) in some suitable order. We first consider all the combinations where $k_Y \geq k_Z$, and then where $k_Z > k_Y$. We analyze the former case; the other is symmetric. We visit the values of k_Y in increasing order, and the values of k_Z in increasing order for each value of k_Y. Each new visited value k_Y is first combined with the smallest k_Z for which $\ell_Y + \ell_Z > \ell$. If this leads to a nonempty range in $\mathcal{G}$, then this is a useful combination, for which we have already accounted. The successive values of k_Z we try out from there are all useful, until we finally fail to find a nonempty range—and this then a useless combination— or until $k_Z > k_Y$. We do not consider further values $k_Z \leq k_Y$ in the first case because they will also fail to produce a nonempty range in $\mathcal{G}$.

Thus, each value of k_Y we visit leads to zero or more useful combinations possibly followed by a single useless one. We say that k_Y *succeeds* if it produces at least one useful combination; otherwise it *fails*. If k_Y succeeds, then the cost of its last useless combination, if any, can be charged to the useful ones it produced. Therefore we only need to count the number of values k_Y that fail. We will now show that a sequence of consecutive values of k_Y that fail has $O(1)$ combinations (all of them useless), and therefore their cost can also be charged to the preceding or following value of k_Y that succeeds. Only a sequence of all-failing values of k_Y cannot be accounted for in that way, but this can only be one sequence per value of p, adding up to $O(m)$ cost for the useless combinations.

The value of ℓ does not change across a sequence of failing values of k_Y. We never visit values $\ell_Y \leq \ell/2$: since $\ell_Z \leq \ell_Y$, they could not increase ℓ. A failing sequence of visited values k_Y then starts with some $\ell_Y > \ell/2$ and increments k_Y successively, combining it with nonincreasing values of k_Z. In this sequence, the first combination (k_Y, k_Z) we try for each k_Y, with the smallest k_Z that yields $\ell_Y + \ell_Z > \ell$, is useless, so we visit only that smallest value of k_Z per value of k_Y. We proceed increasing k_Y, always failing, until ℓ_Y exceeds ℓ, at which point the smallest value of k_Z that makes $\ell_Y + \ell_Z > \ell$ is 0. If such combination also fails, there is no point in continuing with larger values of ℓ_Y, because even combined with $k_Z = 0$ will not yield a useful combination. Since ℓ_Y is exponential in k_Y, there are only $O(1)$ values of k_Y that yield values $\ell/2 < \ell_Y \leq \ell$. Only $O(1)$ combinations are then tried along a sequence of failing values of k_Y.

Overall, we have $O(\log^2 m)$ steps charged to useful combinations and $O(m)$ to useless ones. Multiplied by the range emptiness time complexity, this yields $O(m \log^\delta g)$ total time. Note that we obtain a correct result only with high probability, because we check only that $h(L_Y)$ and $h(L_Z)$ match the hash values of the corresponding block prefixes and suffixes. To ensure correctness, we can store the nonterminal $X \rightarrow Y\,Z$ associated with the point connecting $\langle Y \rangle$ and $\langle Z \rangle$ in $\mathcal{G}$, so as to verify the correctness our answer in $O(m)$ time by extracting a suffix of $\langle Y \rangle$ and a prefix of $\langle Z \rangle$ in optimal time [7]. If our answer turns out to be incorrect (which happens with low probability) we can re-run the algorithm, this time verifying every potentially useful combination, in total time $O(m^2)$. We can thus ensure correct results by making our time $O(m \log^\delta n + m + n^{-c} m^2) = O(m \log^\delta n)$ in expectation (for any constant $c > 2$).

The construction time of our structure is dominated by the construction of the Karp-Rabin hash function with no collisions between blocks of T [13, Sec. 4].

Theorem 1. *Given positive constants ϵ and δ, and an α-balanced straight-line program with g rules for a text $T[1..n]$, we can build in $O(n \log n)$ expected time an $O(g)$-space index with which, given a pattern $P[1..m]$, in $O(m \log^\delta g)$ time we can find with high probability a substring of P that occurs in T and whose length is at least a $(1-\epsilon)$ fraction of the longest common substring of P and T. The correctness can be guaranteed with time still $O(m \log^\delta g)$, yet in expectation.*

References

1. Bille, P., Gørtz, I.L., Sach, B., Vildhøj, H.W.: Time–space trade-offs for longest common extensions. Journal of Discrete Algorithms **25**, 42–50 (2014)
2. Chan, T.M., Larsen, K.G., Pătraşcu, M.: Orthogonal range searching on the RAM, revisited. In: Proc. 27th ACM Symposium on Computational Geometry (SoCG). pp. 1–10 (2011)
3. Charalampopoulos, P., Kociumaka, T., Pissis, S.P., Radoszewski, J.: Faster Algorithms for Longest Common Substring. In: Proc. 29th Annual European Symposium on Algorithms (ESA). pp. 30:1–30:17 (2021)
4. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Transactions on Information Theory **51**(7), 2554–2576 (2005)
5. Claude, F., Navarro, G., Pacheco, A.: Grammar-compressed indexes with logarithmic search time. Journal of Computer and System Sciences **118**, 53–74 (2021)
6. Gao, Y.: Computing matching statistics on repetitive texts. In: Proc. 32nd Data Compression Conference (DCC). pp. 73–82 (2022)
7. Gasieniec, L., Kolpakov, R., Potapov, I., Sant, P.: Real-time traversal in grammar-based compressed files. In: Proc. 15th Data Compression Conference (DCC). p. 458 (2005)
8. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development **2**, 249–260 (1987)
9. Kieffer, J.C., Yang, E.H.: Grammar-based codes: A new class of universal lossless source codes. IEEE Transactions on Information Theory **46**(3), 737–754 (2000)
10. Navarro, G.: Indexing highly repetitive string collections, part I: Repetitiveness measures. ACM Computing Surveys **54**(2), article 29 (2021)
11. Navarro, G.: Indexing highly repetitive string collections, part II: Compressed indexes. ACM Computing Surveys **54**(2), article 26 (2021)
12. Navarro, G.: Computing MEMs on repetitive text collections. In: Proc. 34th Annual Symposium on Combinatorial Pattern Matching (CPM). p. article 22 (2023)
13. Navarro, G., Prezza, N.: Universal compressed text indexing. Theoretical Computer Science **762**, 41–50 (2019)
14. Ohno, T., Goto, K., Takabatake, Y., I, T., Sakamoto, H.: LZ-ABT: A practical algorithm for α-balanced grammar compression. In: Proc. 29th International Workshop on Combinatorial Algorithms (IWOCA). pp. 323–335 (2018)