KATKA: A KRAKEN-like tool
with k given at query time

Travis Gagie[1], Sana Kashgouli[1], and Ben Langmead[2]

[1] Dalhousie University
[2] Johns Hopkins University

Abstract. We describe a new tool, KATKA, that stores a phylogenetic tree T such that later, given a pattern $P[1..m]$ and an integer k, it can quickly return the root of the smallest subtree of T containing all the genomes in which the k-mer $P[i..i+k-1]$ occurs, for $1 \leq i \leq m-k+1$. This is similar to KRAKEN's functionality but with k given at query time instead of at construction time.

1 Introduction

KRAKEN [14,13] is a popular tool that addresses the basic problem of determining where a fragment of DNA occurs in the Tree of Life, which arises for every sequencing read in a metagenomic dataset. KRAKEN takes a phylogenetic tree T and an integer k and stores T such that later, given a pattern $P[1..m]$, it can quickly return the root of the smallest subtree of T containing all the genomes in which the k-mer $P[i..i+k-1]$ occurs, for $1 \leq i \leq m-k+1$. For example, if T is the small phylogenetic tree shown in Figure 1, $k = 3$, and $P = \texttt{TAGACA}$, then KRAKEN returns

- 8 for `TAG` (which occurs in `GATTAGAT` and `GATTAGATA`),
- 6 for `AGA` (which occurs in `AGATACAT`, `GATTAGAT` and `GATTAGATA`),
- NULL for `GAC` (which does not occur in T),
- 2 for `ACA` (which occurs in `GATTACAT`, `AGATACAT` and `GATACAT`).

Notice that not all the genomes in the subtree returned for $P[i..i+k]$ need contain it: `AGA` does not occur in `GATTACAT` or `GATACAT`.

KRAKEN is widely used in metagenomic analyses, especially taxonomic classification, but there are some applications for which we would rather give k at query time instead of at construction time. For example, Nasko et al. [7] showed that "the [reference] database composition strongly influence[s] the performance", with larger k values generally working better as the database grows. When the representation of strains or species in the database is skewed, therefore, it may be hard to choose a single k that works well for all of them. In this paper we describe a new tool, KATKA, that allows k to be chosen at query time. We are still optimizing, testing and extending KATKA and will report experimental results in a future paper.
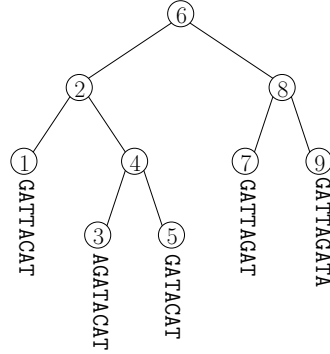
Fig. 1. A small phylogenetic tree.

2 Design

To simplify our presentation, in this paper we assume T is binary (although our approach generalizes to higher-degree trees). KATKA consists of three main components:

- a modified LZ77-index [6] for the concatenation of the genomes in T, in the order they appear from left to right in T and separated by copies of a special character $;
- a modified LZ77-index for the reverse of that concatenation;
- a lowest common ancestor (LCA) data structure for T.

Given $P[1..m]$ and k, we use the first and second indexes to find the leftmost and rightmost genomes in T, respectively, that contain the k-mer $P[i..i+k-1]$, for $1 \leq i \leq m-k+1$; we then use the LCA data structure to find the lowest common ancestor of those two genomes. Since the two indexes are symmetric and the LCA data structure takes only about 2 bits per vertex in T and has constant query time, we describe only the first index.

To build the index for the concatenation, we compute its LZ77 parse and consider the phrases and co-lexicographically sort the set of their maximal non-empty suffixes not containing $, and consider the suffixes of the concatenation starting at phrase boundaries and lexicographically sort the set of their maximal prefixes not containing $ (including the empty string ε after the last phrase boundary). We discard any of those maximal prefixes that do not occur starting at a phrase boundary immediately preceded by one of those maximal suffixes.

For our example, if the concatenation is

GATTACAT$AGATACAT$GATACAT$GATTAGAT$GATTAGATA ,

then its LZ77 parse is

G A T TA C AT$ AG ATA CAT$G ATACAT$GATT AGAT$ GATTAGATA ,

GATTAGATA ATA TA A   C G AG T GATT

```
GATTAGATA
      ATA
       TA                GATT
        A    C  G  T
                AG
             9                 | ε
                          7    | AGAT
                  1            | AT
                   5 3        | ATACAT
                     1        | ATTACAT
             1 3              | CAT
                        1     | TACAT
           1                  | TTACAT
```

Fig. 2. The grid we build for the concatenation in our example.

the co-lexicographically sorted set of maximal suffixes is

$$\mathtt{A, TA, ATA, GATTAGATA, C, G, AG, T, GATT},$$

and the lexicographically sorted set of maximal prefixes is

$$\varepsilon, \mathtt{AGAT, AGATACAT, AT, ATACAT, ATTACAT, CAT,}$$
$$\mathtt{GATTACAT, GATTAGATA, TACAT, TTACAT,}$$

but we discard $\mathtt{GATTACAT}$, $\mathtt{AGATACAT}$ and $\mathtt{GATTAGATA}$ because they do not occur starting at a phrase boundary immediately preceded by one of the maximal suffixes.

We build a grid with the number ℓ at position (x, y) if the genome at the ℓth vertex from the left in T is the first one in which there is a phrase boundary immediately preceded by the co-lexicographically xth of the maximal suffixes and immediately followed by the lexicographically yth of the maximal prefixes. Notice this grid will be of size at most $z \times z$ with at most z numbers on it, where z is the number of phrases in the LZ77 parse of the concatenation. Figure 2 shows the grid for our example.

We store data structures such that given strings α and β, we can quickly find the minimum number in the box $[x_1, x_2] \times [y_1, y_2]$ on the grid, where $[x_1, x_2]$ is the co-lexicographic range of the maximal suffixes ending with α and $[y_1, y_2]$ is the lexicographic range of the maximal prefixes starting with β. (For the index for the reversed concatenation, we find the maximum in the query box.) In our example, if $\alpha = \mathtt{G}$ and $\beta = \mathtt{AT}$, then we should find 1.

For example, we can store Patricia trees for the compact tries for the reversed maximal suffixes and the maximal prefixes, together with a data structure supporting fast sequential access to the concatenation starting at any phrase boundary. In the literature (see [8] and references therein), the latter is usually an augmented straight-line program (SLP) for the concatenation; if the genomes in T are similar enough, however, then in practice it could probably be simply
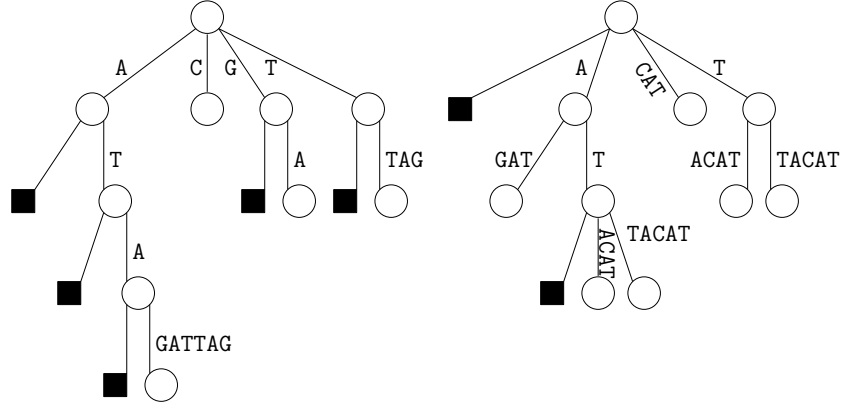
Fig. 3. The compact tries for the concatenation in our example.

a VCF file. (We note that we can reuse the access data structure for the index for the reversed concatenation, augmented to support fast sequential access also at phrase boundaries in the reverse of the concatenation.) Figure 3 shows the compact tries for our example, with each black leaf indicating that one of the strings in the set ended at the parent of that leaf.

Nekrich [12] recently showed how to store the grid in $O(z)$ space and support 2-dimensional range-minimum queries in $O(\log^\epsilon z)$ time, for any constant $\epsilon > 0$. For simplicity, we consider his data structure in our analysis even though we are not aware of any implementation yet.

3   Queries

Given a pattern $P[1..m]$ and an integer k, for every substring $P[i..j]$ of P with length at most k, we find and verify the locus for the reverse of $P[i..j]$ in the compact trie for the reversed maximal suffixes, and the locus for $P[i..j]$ in the compact trie for the maximal prefixes. (Patricia trees can return false positives when the sought pattern does not occur, so we must verify the loci by, for example, extracting their path labels from the SLP.)

By combining the searches for $P[i], P[i..i+1], \ldots, P[i..i+k-1]$, we make a total of $O(m)$ descents in the Patricia trees, each to a string-depth of at most k; we extract $O(m)$ substrings from the concatenation, each of length at most k and starting at a phrase boundary, to verify the loci. With care, this takes a total of $O(km)$ time in the worst case. When searching standard LZ77-indexes in practice, however, "queries often die in the Patricia trees" [10] — because of mismatches between characters in the pattern and the first characters in edge labels — which speeds up queries.

For each k-mer $P[i..i+k-1]$ in P and each way to split $P[i..i+k-1]$ into a non-empty prefix $P[i..j]$ and a suffix $P[j+1..i+k-1]$, we use a 2-dimensional

range-minimum query to find the minimum number in the box for $\alpha = P[i..j]$ and $\beta = P[j + 1..i + k - 1]$ in $O(\log^\epsilon z)$ time.

By the definition of the LZ77 parse, the first occurrence of $P[i..i+k-1]$ in the concatenation crosses or ends at a phrase boundary. It follows that, by taking the minimum of the minima we find, in $O(k \log^\epsilon z)$ time we find the leftmost genome in T that contains $P[i..i + k - 1]$. Repeating this for every value of i takes $O(km \log^\epsilon z)$ time.

By storing symmetric data structures for the reverse of the concatenation and querying them, we can find the rightmost genome in T that contains $P[i..i+k-1]$, for $1 \le i \le m-k+1$. With the LCA data structure for T, we can find the lowest common ancestor of the two genomes, which is the root of the smallest subtree of T containing all the genomes in which the k-mer $P[i..i + k - 1]$ occurs.

For our example, if $P = $ TAGACA and $k = 3$, then we find and verify the loci for

$$\text{T, A, AT, G, GA, GAT, A, AG, AGA, C, CA, CAG, A, AC, ACA}$$

in the compact trie for the reversed maximal suffixes, and the loci for

$$\text{A, AG, AGA, G, GA, GAC, A, AC, ACA, C, CA, A}$$

in the compact trie for the maximal prefixes.

For $P[1..3] = $ TAG, we look up the minimum number 7 in the box for $\alpha = $ T and the locus $\beta = $ AGAT for AG; since G has no locus in the compact trie for the maximal prefixes and GAT has no locus in the compact trie for the maximal reversed suffixes, we correctly conclude that the leftmost genome in T containing TAG is at vertex 7. A symmetric process with the index for the reversed concatenation tells us the rightmost genome in T containing TAG is at vertex 9, and then an LCA query tells us that vertex 8 is the root of the smallest subtree containing all the genomes in which TAG occurs.

Theorem 1. *Given a phylogenetic tree T whose g genomes have total length n, we can store T in $O(z \log n + g/\log n)$ space, where z is the number of phrases in the LZ77 parse of the concatenation of the genomes in T (separated by copies of a special character), such that when given a pattern $P[1..m]$ and an integer k, for $1 \le i \le m-k+1$ we can find the root of the smallest subtree of T containing all genomes in which the k-mer $P[i..i+k-1]$ of P occurs, in $O(km \log^\epsilon z)$ total time.*

Proof. The LCA data structure takes $2g + o(g)$ bits, which is $O(g/\log n)$ words (assuming $\Omega(\log n)$-bit words). An SLP for the concatenation with bookmarks permitting sequential access with constant overhead from the phrase boundaries in the parses of the concatenation and its reverse, takes $O(z \log n)$ space. For the concatenation, the Patricia trees and the instance of Nekrich's 2-dimensional range-minimum data structure take $O(z)$ space; for the reverse of the concatenation, they take space proportional to the number of phrases in its LZ77 parse, which is $O(z \log n)$. In total, we use $O(z \log n + g/\log n)$ space. As we have described, we make $O(m)$ descents in the Patricia trees, each to string-depth at

most k, and extract only $O(m)$ substrings, each of length at most k, from the concatenation and its reverse. The time is dominated by the $O(km)$ range-minimum queries, which take $O(\log^\epsilon z)$ time each.

4   Future Work

In addition to optimizing and testing KATKA, we are also investigating adapting it to work with maximal exact matches (MEMs) instead of k-mers. For example, if we store $O(z)$-space z-fast tries [2] for the Patricia trees then, for each way to split P into a non-empty prefix $P[1..i]$ and a suffix $P[i+1..m]$, we can find the loci of $P[1..i]$ reversed and $P[i+1..m]$ in $O(\log m)$ time. We can verify those loci in $O(\log n)$ time by augmenting the SLP to return fingerprints, without changing its $O(z \log n)$ space bound [3].

With an $O(z)$-space data structure supporting heaviest induced ancestor queries in $O\left(\frac{\log^2 z}{\log \log z}\right)$ time [1,4,5], in that time we can find the longest substring $P[h..j]$ with $h \leq i \leq j$ that occurs in T with $P[h..i]$ immediately to the left of a phrase boundary and $P[h+1..j]$ immediately to its right. Note $P[h..j]$ must be a MEM. With 2-dimensional range-minimum and range-maximum queries, we can find the indexes of the leftmost and rightmost genomes in which $P[h..i]$ occurs immediately to the left of a phrase boundary and $P[h+1..j]$ immediately to its right. We still use a total of $O(z \log n + g/\log n)$ space and now we use a total of $O\left(m\left(\frac{\log^2 z}{\log \log n} + \log n\right)\right)$ time.

Unfortunately, we may not find every MEM this way: it may be that, for some MEM $P[h..j]$ and every i between h and j, either $P[h..j]$ is not split into $P[h..i]$ and $P[i+1..j]$ by any phrase boundary or some longer MEM is split into $P[h'..i]$ and $P[i+1..j']$ by a phrase boundary. For any MEM we do not find, however, we do find another MEM at least as long that overlaps it.

A more serious drawback to this scheme is that it is probably quite impractical (for example, we are not aware of any implementation of a data structure supporting fast heaviest induced ancestor queries, either). Fortunately, there is probably a simple and practical solution — although possibly without good worst-case bounds — since the problems we are considering are similar to those covered in Subsection 7.1 of Navarro's [11] survey on wavelet trees.

Finally, we are investigating adapting results [9] using LZ77-indexes for document-listing, in order to find the number of genomes in T in which each k-mer of P occurs. It is easy to store a small data structure that reports the number of genomes in the smallest subtree for a k-mer, so we may be able to determine what fraction contain that k-mer.

References

1. Paniz Abedin, Sahar Hooshmand, Arnab Ganguly, and Sharma V Thankachan. The heaviest induced ancestors problem: Better data structures and applications. *Algorithmica*, pages 1–18, 2022.

2. Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Fast prefix search in little space, with applications. In *Proc. ESA*, 2010.

3. Philip Bille, Inge Li Gørtz, Patrick Hagge Cording, Benjamin Sach, Hjalte Wedel Vildhøj, and Søren Vind. Fingerprints in compressed strings. *Journal of Computer and System Sciences*, 86:171–180, 2017.

4. Travis Gagie, Paweł Gawrychowski, and Yakov Nekrich. Heaviest induced ancestors and longest common substrings. In *Proc. CCCG*, 2013.

5. Younan Gao. Computing matching statistics on repetitive texts. In *Proc. DCC*, 2022.

6. Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

7. Daniel J Nasko, Sergey Koren, Adam M Phillippy, and Todd J Treangen. newblock RefSeq database growth influences the accuracy of k-mer-based lowest common ancestor species identification. *Genome biology*, 19(1):1–10, 2018.

8. Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.

9. Gonzalo Navarro. Document listing on repetitive collections with guaranteed performance. *Theoretical Computer Science*, 772:58–72, 2019.

10. Gonzalo Navarro. Personal communication.

11. Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

12. Yakov Nekrich. New data structures for orthogonal range reporting and range minima queries. In *Proc. SODA*, 2021.

13. Derrick E Wood, Jennifer Lu, and Ben Langmead. Improved metagenomic analysis with KRAKEN 2. *Genome Biology*, 20(1):1–13, 2019.

14. Derrick E Wood and Steven L Salzberg. KRAKEN: ultrafast metagenomic sequence classification using exact alignments. *Genome Biology*, 15(3):1–12, 2014.