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

1st Travis Gagie
Faculty of Computer Science
Dalhousie University
Halifax, Canada
travis.gagie@dal.ca

2nd Sana Kashgouli
Faculty of Computer Science
Dalhousie University
Halifax, Canada
sana.kashgouli@dal.ca

3rd Ben Langmead
Department of Computer Science
Johns Hopkins University
Baltimore, USA
blangme2@jhu.edu

***Abstract*—We describe a new tool, KATKA, that stores a phylogenetic tree T such that later, given a pattern P 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.**

***Index Terms*—metagenomics, taxonomic classification, phylogenetic trees, lowest common ancestor, LZ77-index**

I. Introduction

KRAKEN [9], [10] 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, 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 $P =$ 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 application for which we would rather give k at query time instead of at construction time. In this paper we describe a new tool, KATKA, which allows this. We are still optimizing and testing KATKA and will report experimental results in the full version of this paper.

II. 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:

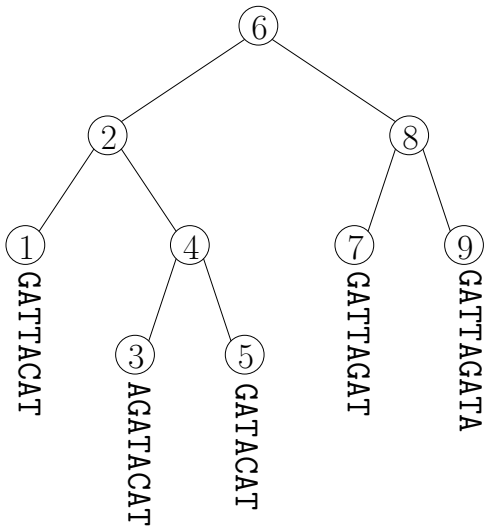


Fig. 1. A small phylogenetic tree.

- a modified LZ77-index [4] 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 contain $ (including the empty string ε after the last phrase boundary). We discard any of

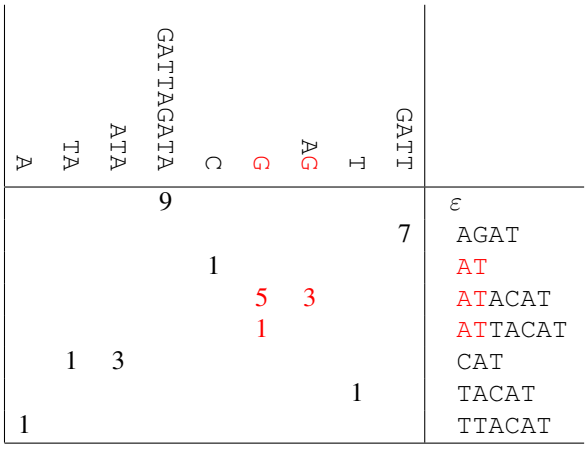


Fig. 2. The grid we build for the concatenation in our example.

those maximal prefixes that do not occur starting at a phrase boundary and immediately preceded by one of those maximal suffixes.

For our example, the concatenation is

GATTACAT$AGATACAT$GATACAT$GATTAGAT$GATTAGATA ,

its LZ77 parse is

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

the co-lexicographically sorted set of maximal suffixes is

A, TA, ATA, GATTAGATA, C, G, AG, T, GATT ,

and the lexicographically sorted set of maximal prefixes is

ε, AGAT, AGATACAT, AT, ATACAT, ATTACAT, CAT,
GATTACAT, GATTAGATA, TACAT, TTACAT,

but we discard GATTACAT, AGATACAT and 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 $\alpha =$ G and $\beta =$ 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 suffixes, together with a data structure supporting fast sequential access to the concatenation starting at any phrase boundary. In the literature (see [5] and references therein), the latter is usually an augmented straight-line program (SLP) for the concatenation, but if the genomes in T are similar enough then it could probably be simply 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.

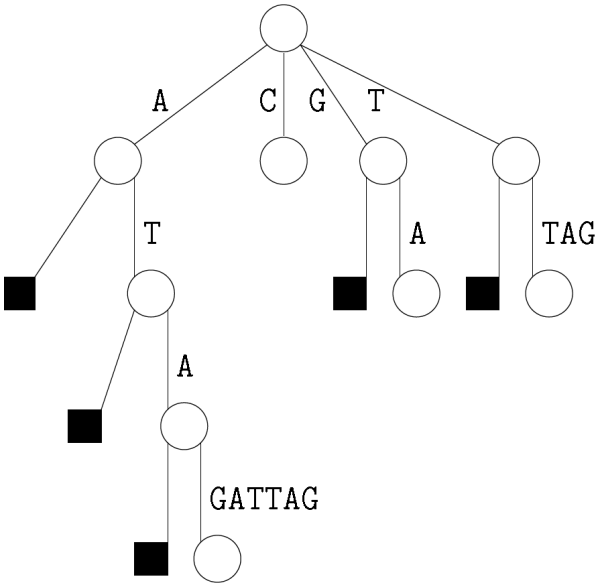
Nekrich [8] recently showed how to store the grid in $O(z)$ space and support 2-dimensional range-minimum queries in $O(\log^{\epsilon} z)$ time, for any constant $\epsilon > 0$, but we are not aware of any implementation yet. In our preliminary experiments, we have simply been storing a map that takes a non-empty path label from the first compact trie and a path label (which may be empty) from the second compact trie, and returns the minimum number in the box corresponding to the nodes with those path labels. (We omit pairs of path labels for which there is no number in the corresponding box.) Although this could take $\Omega(z^2)$ space in the worst case — when the compact tries are very skewed — it seems to work reasonably well in practice.

Figure 4 shows a grid representing the map for our example. In Figure 2, the box for the co-lexicographic range of maximal suffixes ending with G and the lexicographic range of maximal prefixes starting with AT contains a 1, a 3 and a 5, so for G and AT we store $\min(1, 3, 5) = 1$ in our map. This reduces a 2-dimensional range query to a lookup, at the cost of increasing the space in practice by a factor something like the average depth of the leaves in the compact tries.

Summing up, we store Patricia trees for the compact tries for the reversed maximal suffixes and the maximal prefixes, an SLP or VCF file or some other data structure supporting fast sequential access to the concatenation starting at the phrase boundaries), and a map from pairs of path labels in the compact tries to the minimum numbers in the corresponding boxes.

III. 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. 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" [7] — because



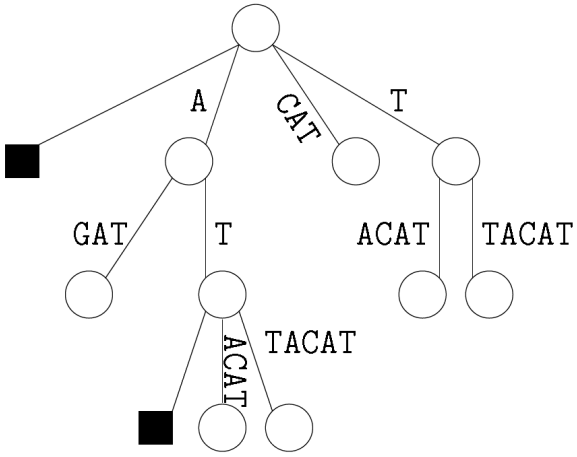


Fig. 3. The compact tries for the concatenation in our example.

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

Fig. 4. A grid representing the map we store for our example. For each non-empty path label in the first compact trie and each path label in the second one, we store the minimum number in the corresponding box on the grid in Figure 2.

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 look up in the map the minimum number in the box for $\alpha = P[i..j]$ and $\beta = P[j+1..i+k-1]$, which takes constant time because we already know the loci for the reverse of $P[i..j]$ and for $P[j+1..i+k-1]$. (We can either hash the substrings or encode them with their loci.) 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 with the map, in $O(k)$ time we find the leftmost genome in T that contains $P[i..i+k-1]$. Repeating this for every value of i also takes $O(km)$ 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 $P = \texttt{TAGACA}$ and $k = 3$, then we find and verify the loci for

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

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

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

in the compact trie for the maximal prefixes.

For $P[1..3] = \texttt{TAG}$, we look up the minimum number 7 in the box for $\alpha = \texttt{T}$ and the locus $\beta = \texttt{AGAT}$ 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.

As we noted in the introduction, we are still optimizing and testing KATKA and will report experimental results in the full version of this paper. For now, we sum up with the following theorem (using Nekrich's result to obtain good space bounds, at the cost of increasing the time bound by a $\log^{\epsilon} z$ factor):

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 $1 \leq i \leq m-k+1$ 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. □

IV. Future Work

In addition to optimizing and testing KATKA, we are also investigating adapting results [1]–[3] about using LZ77-indexes to find the longest common substring and the maximal exact matches of P and the genomes in T, in order to find their subtrees instead of the subtrees for k-mers. Finally, we are investigating adapting results [6] 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.

Acknowledgments

Many thanks to Nathaniel Brown, Younan Gao, Meng He, Finlay Maguire and Gonzalo Navarro, for helpful discussions.

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

[3] Younan Gao. Computing matching statistics on repetitive texts. In *Proc. DCC*, 2022.

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

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

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

[7] Gonzalo Navarro. Personal communication.

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

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

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