Fast Hybrid Data Structure for a Large Alphabet K-Mers Indexing for Whole Genome Alignment

ROSTISLAV HŘIVŇÁK, PETR GAJDOŠ, AND VÁCLAV SNÁŠEL, (Senior Member, IEEE)

Department of Computer Science, FEECS, VŠB—Technical University of Ostrava, 70800 Ostrava, Czech Republic

Corresponding author: Rostislav Hřivňák (rostislav.hrivnak.st@vsb.cz)

This work was supported in part by the SGS Project, VŠB—Technical University of Ostrava, under Grant SP2021/94, in part by the Ministry of Health of the Czech Republic under Grant NU20-06-00269, and in part by the Celgene Research under Grant CZ-102.

ABSTRACT The most common index data structures used by whole genome aligners (WGA) are based on suffix trees (ST), suffix arrays, and FM-indexes. These data structures show good performance results as WGA works with sequences of letters over small alphabets; for example, four letters a, c, t, g for DNA alignment. A novel whole genome aligner, which we are developing, will work with distances between the label sites on DNA samples, which are represented as a sequence of positive integers. The size of alphabet $\sigma$ is theoretically unlimited. This has prompted us to investigate if there is a better structure that would improve search performance on large alphabets compared to the commonly used suffix-based structures. This paper presents the implementation of a highly optimized hybrid data structure based on a ternary search tree (TST) and hash tables, which perform much better when working with strings on large alphabets compared to the ST. Single core parallelism was achieved using advanced vector extensions (AVX) single instruction multiple data (SIMD) instruction set. When searching for short k-mers over an alphabet of 25,695 letters, our index search performance was up to 29 times better than the search performance of the reference ST. When the alphabet was compressed to approximately 1,300 letters, our index search performance was still up to 2.6 times better than the ST. The source code is available free on http://olgen.cz/Resources/Upload/Home/public/software/hds.zip under the MIT license.

INDEX TERMS Engineering management, engineering in medicine and biology, bioinformatics, computational and artificial intelligence, computer architecture, data structures, tree data structures, parallel processing, parallel algorithms.

I. INTRODUCTION Whole genome alignment (WGA), which is sometimes called mapping, is a process when two genomes, or their parts, are compared and the differences are found and annotated [1]–[4]. Retrieved information is then used by medical professionals to identify causes of illness or to precisely target treatment. Thanks to new long-read DNA scanning tools and methods, scientist can scan a patient’s whole genome very quickly. One of the tools available for such long-read genome scanning is a product from Bionano Genomics called Saphyr [5]–[7], which allows up to six human samples to be processed per day with 100 times coverage. The Saphyr device labels DNA with fluorescent markers markers, which are then scanned by the camera under the microscope. The positions of the labels are measured in base pairs from the start of the scanned DNA fragment so that the distances between these label sites can be calculated. This sequence of label site distances represents scanned DNA fragment.

In contrast, standard genome aligners [8], [9] works with a size-4 alphabet, where each letter represents one nucleotide. Most commonly used indexes are based on suffix data structures. MUMmer [10] aligner up to version 3 uses suffix tree (ST) [11], [12], and MUMmer version 4 uses a suffix array [13], [14]. STAR [15] aligner improves the speed of the suffix array as it does not compress it. The next data structures used by the aligners are variations of FM-indexes [16], [17], which are compressed variations of the suffix array. This data structure is used by the BOWTIE2 [18] or HISAT [19] aligners. The HISAT2 [20] aligner uses graph-based FM-indexes.

All of these data structures have one thing in common in that the time complexity of the search algorithm is always a

\[
\text{In this article, the terms sequence, string, and word denote the sequence of positive integer numbers, and the term letter will denote the element of such a sequence.}
\]
function of alphabet size $\sigma$. For example, ST nodes have up to $\sigma$ child nodes that must be iterated during the search operation. FM-indexes must sort subsets of indexed data during the Burrows-Wheeler transform decompression phase. This sort time complexity is also a function of alphabet size $\sigma$.

A completely different approach is to use the hash table to considerably speed up the lookup of the values. Various kinds of BLAST [21]–[23] or SSAHA [24] aligners uses hash tables to store all possible k-mer combinations and during the query execution, it can exploit the $O(1)$ time complexity of accessing the hash table value. This method is usable only on very small alphabets as the number of combinations that have to be stored grows exponentially with each letter in the alphabet. When the hash tables are used, there is a problem with implementing the approximate search algorithm for the data structure, which, in turn, leads to lower sensitivity of the aligner.

In this article, an alternative approach is presented using a ternary search tree (TST) which is a data structure for indexing multiple strings and then searching for their prefixes. If the functionality of the ST is replicated with the TST data structure, then all the $m$ suffix strings of the indexed string $S$ of length $m$ must be indexed in the TST. Values in the TST index can be the starting positions of the suffixes relative to the source string $S$. Values can also reference multiple source strings.

Another approach to replicating the functionality of the ST with the TST is to use the concept of k-mers. String $S$ of length $m$ can be split into $m-k+1$ k-mers of length $k$. The resulting k-mers are then used as keys in the TST index, and the values are k-mer starting offsets relative to source string $S$. When searching for any string of length $k$, the TST index now has the same functionality as the ST on string $S$ because TST contains all of the sub-sequences of length $k$ from input string $S$.

The TST data structure has the following advantages:

- It can store $n$ input strings $S_0, S_1, \ldots, S_n$, and just the index value must reference the position of the key sequence in the appropriate source string. This means that the ST index can have the same functionality as a generalized suffix tree (GST).

One disadvantage of the TST is that it has a higher memory consumption compared to the ST.

The TST index can be used as a base data structure for the so-called ternary search trees forest (TSTF). TST is used as a value in the hash table, and the key (hash) is calculated from the first $n$ characters (numbers) of the indexed string (k-mer). This operation splits one TST into many smaller trees and leads to considerably quicker search times, as the number of comparisons at the top of the TST is substituted by a constant time access to the hash table. The introduction of hash table leads to increased memory demands on the whole TSTF index. Another disadvantage of the TSTF is that the data for searching must be prepared by calculating the hash of the first $n$ key values using the same hash function that is used in the TST. Then, each search key must be placed into the collection of keys with the same hash. This enables a parallel search in the TST inside the TSTF. Search data preprocessing can be performed during the data loading or search data generation step of the genome mapping process so it is not part of the search operation itself. A challenge, which must yet be solved, is to implement an approximate search algorithm for the TSTF because of the hash table usage. The algorithm must return a set of TSTs from hash table that are close enough to a given key with given tolerance parameters.

II. TERNARY SEARCH TREE

This section will cover implementation details of the TST data structure. A TST [25] is a trie data structure made for indexing multiple strings and rapidly searching through them using string prefixes. The implementation supports the following features:

- Node Stop Creation Depth parameter
- Insert vector of keys and values
- Search for vector of keys

The Node Stop Creation Depth parameter is a positive number parameter that specifies the number of letters in the inserted string from which new nodes are not created and joined to the TST. Instead, letters from that position in the string are stored in the array (called residual data in our implementation) and linked to the last node in the TST on the proper branch. This optimization comes from the observation that branching in the TST happens at the start of words, and memory can be saved by not allocating unnecessary tree nodes. Branching at the top of the TST happens more frequently when the alphabet size and/or entropy of the input data grows.

Insert and search operations are both sped up using an AVX2 [26]–[28] SIMD instruction set. Single core parallelism is used when possible during the search and insert algorithms in the TST, yet some parts of the insertion and
search algorithm must be executed sequentially. The main reason for this is that program branching using the SIMD instruction set is impossible without the execution of both true and false code branches. Both code branches must be evaluated before merging the results together based on the condition. This can lead to accessing non-existent elements of an array or non-existent objects using a null pointer. Our implementation uses AVX2, i.e., 256-bit registers. One letter of the string is represented by the 32-bit integer, so the AVX2 instructions can process eight words in parallel.

During the parallel insertion phase, collisions may occur when two different letters from different words are inserted into the same tree node in the same step. This problem is solved by inserting all the keys into the target tree node sequentially before moving to the next letter or word. The letter from the last iterated word remains in the tree node. The algorithm will move to the next letter or word only if the currently processed letter of the word is equal to the key in the target node.

**TST insertion algorithm outline**

- Initialization
- While Word For Processing Exists
  - Update Letter Index Greater than Parameter Register
  - Update Current Node Index Register
  - Update Letter Register
  - Fill Compare Registers
  - Update Letter Register
  - Insert Indices
  - Update Word Register

**TST search algorithm outline**

- Initialization
- While Word For Processing Exists
  - Update Letter Index Greater than Parameter Register
  - Update Key Registers
  - Update Node Key Register
  - Search Residual Data
  - Compare Key Registers
  - Process Search Results
  - Update Word Register

Source data, words (keys) and values are passed in by reference to vectors (arrays). The algorithm uses several 256-bit state registers that are manipulated by the functions called from the main “while” loop. The algorithm runs as long as there are some words to process. TST nodes are stored in the memory-optimized array of bytes. Residual data are stored in the vectors, and pointers to the vectors are stored in the TST main byte array.

The following list describes the main 256-bit state registers:

- **Word Register** - indices of the currently processed word in the source vector.
- **Letter Register** - indices of currently processed letters in the currently processed word.
- **Current Node Index Register** - indices of the currently processed nodes in the main byte array.
- **Key Register** - filled with the key (letter) from the currently processed word.
- **Node Key Register** - filled with the key (letter) from the currently processed node based on the Current Node Index Register.
- **Compare Registers** - a set of three registers (less than, equal to, greater than) that are filled by comparing the Key Register with the Node Key Register.
- **Letter Index Greater than Parameter Register** - stores information if the letter index is greater than the Node Stop Creation Depth parameter, so the rest of the word should be stored in the residual data vector and not in the nodes.

A detailed function description is not provided as the algorithm itself is quite straightforward, and the main design points are provided.

The algorithm can be scaled up in several ways. Both operations, insert and search, can be scaled up on a single core level by adjusting the AVX part of the algorithm. If the alphabet can be reduced to 16-bit letters, then AVX2 256-bit registers can be divided into 16 16-bit segments, and up to 16 words can be processed in parallel. If central processing unit (CPU) with an AVX-512 instruction set is available, the algorithm can be adjusted to work with AVX-512 instructions, and up to 32 words in a 16-bit alphabet can be processed in parallel. The search operation can be further sped up by multi-threading using an OpenMP framework, for example.

**III. FOREST OF TERNARY SEARCH TREES**

As stated in section I, TSTF is an optimization that splits one TST into several smaller TSTs stored in a hash table. The hash function calculates the hash table key based on the first two values of the string that is indexed in the TSTF. This means that the insert and search operations in the TSTF consist of a data preparation step in which the hash is calculated and indexed strings are stored in collections - one string collection for its respective key hash. Then, strings from each collection are inserted into the TST under the matching hash table key.

Summary Hash Function (1), which was used for testing of the presented solution, sums the first two letters from the string.

\[
f_s(s_0, s_1) = s_0 + s_1 \tag{1}
\]

where \( s_0 \) is letter \( S[0] \) and \( s_1 \) is letter \( S[1] \) and both \( s_0 \) and \( s_1 \) are positive integers greater than zero. Hash functions also assume that the input string \( S \) has a length of at least two. Presented implementation of the TSTF uses a unsigned 32-bit integer as a hash table key data type. If \( f_s \) is greater than \( 2^{32} - 1 \) then the string is stored in the TST under the hash key 0.
IV. TEST METHODOLOGY, DATA AND PARAMETERS

This section describes how the data structures were tested, on which datasets, what parameters were selected, and why and how the measured results were evaluated.

A. GENERAL TESTING INFORMATION

All of the testing was conducted on a desktop class computer with AMD Ryzen 7 3700X 4.2 GHz, 32GB RAM 3200 MHz, and a NVMe SSD hard drive. Implementation was performed in C++ 17 and built using a mingw-64 GCC 9.3.0 compiler. Testing was executed on Linux Ubuntu running on a Windows Subsystem for Linux 2 (WSL2) in Windows 10.

An ST data structure was chosen as the reference data structure because it is the fastest data structure from the set of suffix-based data structures [29]. A slightly modified C++ implementation of the ST was used [30]. Parameters could not be set for the ST, and the only modification was the implementation of memory consumption measurement at the tree node level.

Human reference genome HG38 was used as test data. The genome was downloaded from the University of California Santa Cruz Genomics Institute (UCSC) in FASTA format [31]. It was labeled with a “cttag” marker sequence, which is commonly used by the Bionano Saphyr device. Any other marker sequence can be used, then the number of label sites will differ. Tests were performed with all 23 pairs of chromosomes of the HG38 genome. Total count of labels used for all tests is 534, 397.

As mentioned in sections II and III, TST and TSTF data structures had two important parameters that influenced their search performance and size. The first parameter is Node Stop Creation Depth, which was set to value 10 for all tests. This choice means that for 5-mers and 10-mers, tree nodes are always created for each letter of the inserted string. For 30-mers, it means that at least the first 10 letters of the string are stored in the tree nodes, and up to 10 letters are stored in residual data arrays. Value 10 was chosen based on the several experiments with the TST and TSTF data structures, but those are not presented here because of space constraints. The value of the Node Stop Creation Depth parameter should be changed based on the experiments with data structures in concrete application.

The second parameter is the hash function of the TSTF data structure. Here, summary hashing was chosen, as was mentioned in chapter III, because the TSTF data structure demonstrates the best search performance results with this hashing function.

B. TESTING METHODOLOGY

A string consisting of distances between the labels was coded by the alphabet coder to reduce the size of the alphabet. Alphabet coding is used only to reduce size of the alphabet to show performance difference between TSTF, TST and ST data structures. It is not aim of this article to study consequences of the alphabet size reduction on the genome alignment itself, but it can be expected that the quality of alignment will be lower due to information loss during the alphabet compression.

The next step was to transform the coded string into k-mers. K-mers lengths \( k = 5, 30, 50, \) and \( k = 100 \) were selected for testing. Please note that k-mer lengths were selected arbitrarily to show how search and memory performance of the data structures depends on it. Proper k-mer length selection is not deterministic process [32], [33] and depends on the kind of input data and specific genome aligner.

To best of our knowledge the optimal lengths of k-mers used varies from tens to lower hundreds of characters.

K-mers must be created with no step-over (i.e., k-mers of length 3 for sequence 1, 2, 3, 4, 5 are \{1, 2, 3\}, \{2, 3, 4\}, and \{3, 4, 5\}) because k-mers must cover a whole input string. During this step, the collection of k-mer offsets against the source string was calculated. This provides a reference to the source genome, and during the search phase, matches can be located in the source string. There are always \( S_{len} - k + 1 \) k-mers, where \( S_{len} \) is the length of the string and \( k \) is k-mer length. Total number of k-mers is very close to total number of labels in the input string.

The collection of k-mers and offsets can now be indexed in the TST, TSTF, and GST. The coded input string was indexed in the ST directly because it can find positions of any substring that occurs in the indexed string.

Search and indexing (insert) times were measured for all data structures. Indexing times can be found in the supplementary data and will not be the subject of further analysis as it is an one-time operation and does not have significant practical consequences. Please refer to the supplementary data section of this article to find out the values of the search performance measurement used to create the charts in section V. Search performance was measured by iterating through all of the k-mers in the input data set as it is commonly known that all of the k-mers must be found in the index (positive-search test). Then, all of the k-mers were modified in such a way that none of them were present in the index, and the search test was executed again (negative-search test). Negative search was always faster for all of the data structures presented in this article and will not be the subject of further analysis.

Memory consumption was measured at the tree nodes level of the data structures. Tree nodes were counted during the data indexing phase and then multiplied with the byte size of the node. For TSTF, the size of the hash table was added to the total size of the data structure.

V. TEST RESULTS

Figure 1 demonstrates how the search performance of the studied data structures depends on the alphabet size. The measured performance data were plotted for four sizes of k-mers (5, 30, 50 and 100).

For 5-mer and alphabet size \( \sigma = 25,695 \) (no alphabet coding) the results show that the presented TST and TSTF data structures are up to 29-times faster than the ST data structure. When alphabet size is reduced by coding to
Figure 1: Relationship of the data structure search performances on the alphabet sizes.

\[ \sigma = 1, 330, \] the TST and TSTF data structures are still 2.6-times faster than the ST. The search speed of TST and TSTF is roughly the same in both experiments, yet the search speed of ST gets much faster as the \( \sigma \) decreases. This fully agrees with the theory that the search performance of the ST strongly depends on the alphabet size, and the performance of the TST does not.

Figure 2 shows how the search performance of the studied data structures depends on the length of k-mers. Data are plotted for different sizes of the alphabet (1, 330, 4, 221, 10, 880, 14, 991, 25, 695).

We can see that ST performance is better than TSTF and TST for small alphabet sizes and large k-mer lengths and...
does not depend on k-mer length. This performance deficit of TSTF and TST is offset for bigger alphabet sizes.

Figure 3 demonstrate the relationship of the data structure sizes on alphabet size $\sigma$. The measured performance data were plotted for four sizes of k-mers (5, 30, 50 and 100).

Size of the ST data structure remains constant, but size of the TSTF and TST depends on the alphabet size slightly as the new tree nodes must be created for new letters which occurs in the input data set. This fully agrees with the theory.

Lastly, figure 4 shows how memory consumption of the data structures depends on the k-mer lengths. Data are plotted for different sizes of the alphabet (1, 330, 4, 221, 10, 880, 14, 991, 25, 695).

Again, size of the ST does not depend on k-mer length as there is one input string indexed in the data structure. For TSTF and TST we can see that memory consumption grows until Node Stop Creation Depth parameter stops creation of new tree nodes from certain length of k-mers. After that memory consumption remains constant.
VI. DISCUSSION
We have presented how a TST data structure can be used to implement a k-mer index on large alphabets. Its theoretical average search time complexity is \(O(\log n)\), where \(n\) is a total count of nodes in the TST; therefore, it is independent of alphabet size \(\sigma\). Combining TST with the hash table further improves the search speed by utilizing \(O(1)\) access time with the underlying TST. Subsequently, data structure indexing and search performance was further improved using single core parallelism with an SIMD AVX2 instruction set. Memory consumption was also improved by introducing the Node Stop Creation Depth parameter.

Search performance was much better than the performance of the ST, which was used as a benchmark. Implementing the TSTF can process up to 29 times more data than ST in the same amount of time for our testing set of data. This difference will be even greater for larger input sets with larger alphabet sizes. Of course, a better search performance is redeemed by higher memory consumption and preprocessing the data to be searched. Memory consumption is not a huge concern in this application as a whole genome (for example in FASTA format) is not indexed. The observed sizes of the TSTF index were in order of hundreds of megabytes when indexing the genome label site distances. Such memory requirements are miniscule in comparison to what computers are able to handle nowadays. Preprocessing the data for searching, which means calculates the hash function value of the lookup string and classifies the search strings by it, can be performed by the aligner during the search data preparation phase. This means that the aligner must be designed with this requirement in mind.

Further work can be divided into several areas. The first area is to technically improve the throughput of the TSTF during the search phase. This can be achieved by implementing data parallelism on multiple threads or CPU cores. It is also possible to scale up AVX2 to AVX512, although this requires the use of a CPU that supports the AVX512 instruction set. The second area is optimizing the TSTF data structure, which is basically optimizing the Node Stop Creation Depth parameter in conjunction with the hashing and alphabet coding functions. This area extends the data structure design to the design and optimization of the complete genome aligner software solution because those parameters not only affect performance (Node Stop Creation Depth parameter and hashing function), but also the quality of the alignment (alphabet coding function). The third area to implement a partial search algorithm, which would be a challenge because of the use of the hashing function.

**SUPPLEMENTARY DATA**
Section contains numerical data obtained during the performance testing of the data structures. Input set are 23 human chromosomes from HG38 human genome labeled with cttaag sequence. This results in total count of 534, 397 labels. Total count of k-mers used for indexing and searching can be calculated as \(K_{\text{tot}} = L_{\text{cttaag}} - K_{\text{len}} + 1\), where \(L\) is total count of labels and \(K_{\text{len}}\) is length of k-mers.

**REFERENCES**
[1] M. Anisimova, *Evolutionary Genomics*. Springer, 2019.
[2] C. N. Dewey, “Whole-genome alignment,” *Evol. Genomics*, pp. 237–257, 2012.
[3] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg, “Alignment of whole genomes,” *Nucleic Acids Res.*, vol. 27, no. 11, pp. 2369–2376, Jan. 1999.
[4] J. Armstrong, I. T. Fiddes, M. Diekhans, and B. Paten, “Whole-genome alignment and comparative annotation,” *Annu. Rev. Animal Biosci.*, vol. 7, no. 1, pp. 41–64, Feb. 2019.
[5] Bionano. (2021). *Bionano SAPHYR*. Accessed: Mar. 13, 2021. [Online]. Available: https://bionanogenomics.com/products/saphyris/.
[6] J. M. Shelton, M. C. Coleman, N. Herndon, N. L. E. T. Lam, T. Anantharaman, P. Sheth, and S. J. Brown, “Tools and pipelines for BioNano data: Molecule assembly pipeline and FASTA super scaffolding tool,” *BMC Genomics*, vol. 16, no. 1, pp. 1–16, Dec. 2015.
[7] S. Thankaswamy-Kosalai, P. Sen, and I. Nookaew, “Evaluation and assessment of read-mapping by multiple next-generation sequencing aligners based on genome-wide characteristics,” *Genomics*, vol. 109, nos. 3–4, pp. 186–191, Jul. 2017.

**TABLE 1.** Indexing times for different sizes of the alphabet \(\sigma\): all values are in milliseconds.

| \(\sigma\) | 5-mer | 10-mer | 50-mer | 100-mer | 5-mer | 10-mer | 50-mer | 100-mer |
|---------|-------|--------|--------|---------|-------|--------|--------|---------|
| TSTF    | ST    | TSTF   | ST     | TSTF    | ST    | TSTF   | ST     | TSTF   |
| 1,330   | 337   | 282    | 1,046  | 985     | 1,639 | 1,521  | 3,705  | 3,333  |
| 4,221   | 435   | 320    | 1,171  | 1,059   | 1,831 | 1,585  | 3,842  | 3,491  |
| 10,880  | 457   | 313    | 1,144  | 1,026   | 1,832 | 1,541  | 3,874  | 3,588  |
| 14,991  | 445   | 313    | 1,181  | 1,029   | 1,831 | 1,559  | 3,966  | 3,848  |
| 25,695  | 476   | 319    | 1,322  | 1,037   | 2,044 | 1,543  | 4,298  | 3,567  |

**TABLE 2.** Performance of data structures for different alphabet sizes \(\sigma\): all measured times are in milliseconds.

| \(\sigma\) | 5-mer | 10-mer | 50-mer | 100-mer | 5-mer | 10-mer | 50-mer | 100-mer |
|---------|-------|--------|--------|---------|-------|--------|--------|---------|
| TSTF    | ST    | TSTF   | ST     | TSTF    | ST    | TSTF   | ST     | TSTF   |
| 1,330   | 165   | 145    | 374    | 576    | 399   | 415    | 672    | 244    |
| 4,221   | 142   | 120    | 711    | 351    | 601   | 722    | 445    | 695    |
| 10,880  | 136   | 120    | 1,458  | 330    | 572   | 1,462  | 388    | 690    |
| 14,991  | 131   | 125    | 2,009  | 326    | 589   | 2,005  | 429    | 727    |
| 25,695  | 123   | 276    | 3,571  | 327    | 579   | 3,584  | 441    | 695    |
[8] A. Pang, J. Lee, T. Anantharaman, E. Lam, A. Hastie, and M. Borodkin, “Comprehensive detection of germline and somatic structural mutation in cancer genomes by bionano genomics optical mapping,” *J. Biomol. Technol.*, vol. 30, p. 59, 2019.

[9] B. N. Keel and W. M. Snelling, “Comparison of burrows-wheeler transform-based mapping algorithms used in high-throughput whole-genome sequencing: Application to illumina data for livestock Genomes I,” *Frontiers Genet.*, vol. 9, p. 35, Feb. 2018.

[10] G. Marçais, A. L. Delcher, A. M. Phillippy, R. Coston, S. L. Salzberg, and A. Zimin, “MUMmer4: A fast and versatile genome alignment system,” *PLOS Comput. Biol.*, vol. 14, no. 1, Jan. 2018, Art. no. e1005944.

[11] P. Weiner, “Linear pattern matching algorithms,” in *Proc. 14th Annu. Symp. Switching Automata Theory (SWAT)*, Oct. 1973, pp. 1–11.

[12] E. Ukkonen, “On-line construction of suffix trees,” *Algorithmica*, vol. 14, no. 3, pp. 249–260, Sep. 1995.

[13] U. Manber and G. Myers, “Suffix arrays: A new method for on-line string searches,” *SIAM J. Comput.*, vol. 22, no. 5, pp. 935–948, Oct. 1993.

[14] A. Ganguly, D. Gibney, S. Hooshmand, M. O. Külèkçi, and S. V. Thankachan, “FM-index reveals the reverse suffix array,” in *Proc. 31st Annu. Symp. Combinat. Pattern Matching (CPM)*, Wadern, Germany: Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020, pp. 1–14.

[15] A. Dobin, C. A. Davis, F. Schlesinger, J. Drenkow, C. Zaleski, S. Jha, P. Batut, M. Chaisson, and T. R. Gingeras, “STAR: Ultrafast universal RNA-Seq aligner,” *Bioinformatics*, vol. 29, no. 1, pp. 15–21, Jan. 2013.

[16] M. Burrows and D. Wheeler, “A block-sorting lossless data compression algorithm,” in *Digital SRC Research Report.*

[17] H. Li and R. Durbin, “Fast and accurate short read alignment with Burrows–Wheeler transform,” *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[18] B. Langmead and S. L. Salzberg, “Fast gapped-read alignment with Bowtie 2,” *Nature Methods*, vol. 9, no. 4, p. 357, 2012.

[19] D. Kim, B. Langmead, and S. L. Salzberg, “HISAT: A fast spliced aligner with low memory requirements,” *Nature Methods*, vol. 12, no. 4, pp. 357–360, Apr. 2015.

[20] D. Kim, J. M. Paggi, C. Park, C. Bennett, and S. L. Salzberg, “Graph-based genome alignment and genotyping with HISAT2 and HISAT-genotype,” *Nature Biotechnol.*, vol. 37, no. 8, pp. 907–915, Aug. 2019.

[21] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, “Gapped BLAST and PSI-BLAST: A new generation of protein database search programs,” *Nucleic Acids Res.*, vol. 25, no. 17, pp. 3389–3402, 1997.

[22] G. M. Boratyn, J. Thierry-Mieg, D. Thierry-Mieg, B. Busby, and T. L. Madden, “Magic-BLAST, an accurate RNA-seq aligner for long and short reads,” *BMC Bioinf.*, vol. 20, no. 1, pp. 1–19, Dec. 2019.

[23] C. Camacho, G. Coulouris, V. Avagyan, N. Ma, J. Papadopoulos, K. Bealer, and T. L. Madden, “BLAST+: Architecture and applications,” *BMC Bioinf.*, vol. 10, no. 1, pp. 1–9, Dec. 2009.

[24] Z. Ning, A. J. Cox, and J. C. Mullikin, “SSAHA: A fast search method for large DNA databases,” *Genome Res.*, vol. 11, no. 10, pp. 1725–1729, Oct. 2001.

[25] J. L. Bentley and R. Sedgewick, “Fast algorithms for sorting and searching strings,” in *Proc. 5th Annu. ACM-SIAM Symp. Discrete Algorithms, 1997*, pp. 360–369.

[26] Intel. (2021). *Intel Intrinsic Documentation*. Accessed: Mar. 13, 2021, [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[27] C. Pičič, J. M. Abaín, and J. C. Pichel, “Very fast tree: Speeding up the estimation of phylogenies for large alignments through parallelization and vectorization strategies,” *Bioinformatics*, vol. 36, no. 17, pp. 4658–4659, Nov. 2020.

[28] C. A. Darby, R. Gaddipati, M. C. Schatz, and B. Langmead, “Vargas: Heuristic-free alignment for assessing linear and graph read alignments,” *Bioinformatics*, vol. 36, no. 12, pp. 3712–3718, Jun. 2020.

[29] W.-K. Hon, T. W. Lam, W.-K. Sung, W.-L. Tse, C.-K. Wong, and S.-M. Yiu, “Practical aspects of compressed suffix arrays and FM-index in searching DNA sequences,” in *Proc. ALENEX/ANALC*, 2004, pp. 31–38.

[30] M. Cummings. (2021). *Generalized Suffix Tree*. Accessed: Mar. 13, 2021, [Online]. Available: https://github.com/murraycu/murrayc-suffix-tree/

[31] University of California Santa Cruz Genomics Institute. (2021). *Hg38 Fasta Source*. Accessed: Mar. 13, 2021, [Online]. Available: http://hgdownload.cse.ucsc.edugoldenpath/hg38/bigZips/

[32] R. Chikhi and P. Medvedev, “Informed and automated k-mer size selection for genome assembly,” *Bioinformatics*, vol. 30, no. 1, pp. 31–37, Jan. 2014.

[33] D. A. Durai and M. H. Schulz, “Informed kmer selection for de novo transcriptome assembly,” *Bioinformatics*, vol. 32, no. 11, pp. 1670–1677, 2016.

**ROSTISLAV HŘIVŇÁK** received the master’s degree in measurement and control engineering from the VŠB—Technical University of Ostrava, Czech Republic, in 2011. Since 2007, he has been working on different positions in private companies, such as Tieto and DELL or Oriflame Software, where he is currently a Senior Software Engineer. Since 2019, he has been a Post-graduate student in information technology with the VŠB—Technical University of Ostrava. His research interests include the big data, data structures, and data analysis, primarily in bioinformatics.

**PETR GAJDOŠ** received the master’s degree in software engineering and economics and the Ph.D. degree in informatics and applied mathematics from the VŠB—Technical University of Ostrava, Czech Republic, in 2003 and 2006, respectively. Since 2006, he has been working in the position of Assistant Professor with the VŠB—Technical University of Ostrava. In 2018, he finished habilitation thesis and became an Associate Professor. His research and development are aimed at the area of information retrieval, machine learning, data analysis, large-scale data processing, parallel programming, and computer graphics. Generally, he is focused on big data analysis and optimization problems in several projects dealing with bio-medical data analysis. He has authored or coauthored several refereed journal/conference papers, and book chapters.

**VÁCLAV SNÁŠEL** (Senior Member, IEEE) received the master’s degree in numerical mathematics from the Faculty of Science, Palacky University, Olomouc, Czech Republic, in 1981, and the Ph.D. degree in algebra and number theory from Masaryk University, Brno, Czech Republic, in 1991. He is currently a Full Professor with the VŠB—Technical University of Ostrava, Czech Republic. His research and development experience includes more than 30 years in the industry and academia. He works in a multidisciplinary environment involving artificial intelligence, social networks, conceptual lattice, information retrieval, semantic web, knowledge management, data compression, machine intelligence, and nature and bioinspired computing applied to various real-world problems. He has authored or coauthored several refereed journal/conference papers, books, and book chapters. He is the Chair of the IEEE International Conference on Systems, Man, and Cybernetics, Czechoslovak Chapter. He also served as an Editor/Guest Editor for several journals, such as Engineering Applications of Artificial Intelligence (Elsevier), Neurocomputing (Elsevier), and Journal of Applied Logic (Elsevier).

* * *