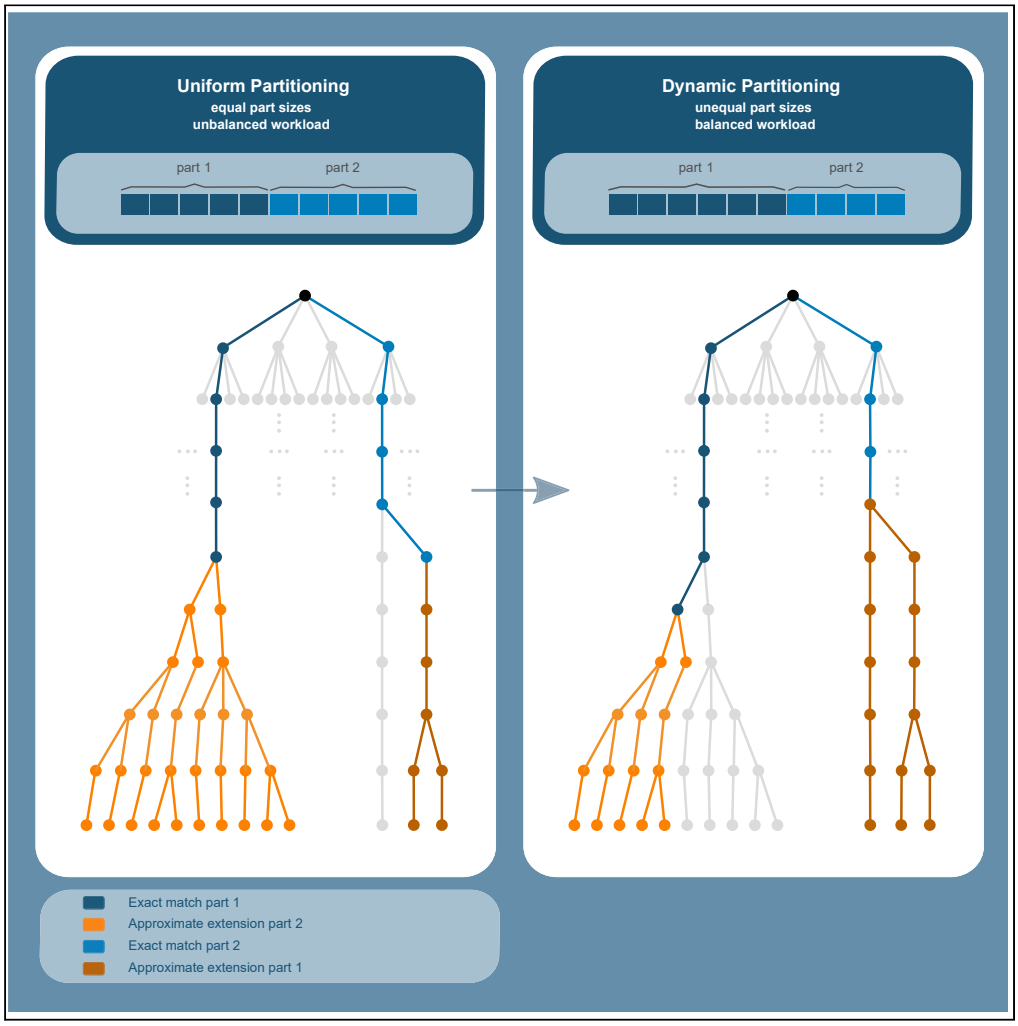
iScience

CellPress
OPEN ACCESS

Article

Dynamic partitioning of search patterns for approximate pattern matching using search schemes



Luca Renders,
Kathleen Marchal,
Jan Fostier

jan.fostier@ugent.be

Highlights

Dynamic partitioning of search patterns reduces search space and runtime

Memory interleaving of bit vector representations of the BWT reduces runtime

Avoiding redundancy inherent to the edit distance metric reduces the search space

Our software tool Columba outperforms the state of the art by a factor of 3.5

Renders et al., iScience *24*, 102687
July 23, 2021
https://doi.org/10.1016/j.isci.2021.102687

iScience

CellPress
OPEN ACCESS

Article

Dynamic partitioning of search patterns for approximate pattern matching using search schemes

Luca Renders,[1] Kathleen Marchal,[1,2] and Jan Fostier[1,3,*]

SUMMARY

Search schemes constitute a flexible and generic framework to describe how all approximate occurrences of a search pattern in a text can be found efficiently. We propose an algorithm for the dynamic partitioning of search patterns which can be universally applied to any kind of search scheme and demonstrate that this technique significantly reduces the search space. We present Columba, a software tool written in C++, in which a multitude of search schemes are implemented. We discuss implementation aspects such as memory interleaving of Burrows-Wheeler transform representations and the reduction of redundancy that is inherently associated with the edit distance metric. Ultimately, we demonstrate that Columba has superior performance to the state of the art. Using a single CPU core, Columba is able to retrieve all occurrences of 100,000 Illumina reads and their reverse complements within a maximum edit distance of four in the human genome in less than 3 min.

INTRODUCTION

Sequence alignment algorithms are central to many bioinformatics applications. Given a query pattern P (e.g., a read) and a search text T (e.g., a reference genome or a collection of genomes), the basic task is to identify occurrences of P in T. Due to the presence of sequencing errors and natural variation, one is often interested in "approximate pattern matching": finding occurrences of P in T within a certain Hamming distance (allowing only substitutions) or Levenshtein/edit distance (allowing substitutions, insertions, and deletions).

Most bioinformatics tools (e.g., BLAT [Kent, 2002], BLAST [Altschul et al., 1990], BWA [Li and Durbin, 2009], etc.) use "lossy" approximate pattern matching algorithms: they rely on heuristics to quickly identify some (but not necessarily all) approximate matches of P in T. By sacrificing some sensitivity, significant gains in performance can be obtained. In this paper, we focus on "lossless" algorithms that are guaranteed to retrieve all approximate matches of P in T. Recent algorithmic developments on "bidirectional indexes" and "search schemes" promise to decrease the performance gap between lossy and lossless approximate pattern matching algorithms.

Regular full-text substring indexes (e.g., suffix trees [Gusfield, 2007], enhanced suffix arrays [Abouelhoda et al., 2004], FM indexes [Ferragina and Manzini, 2000]) allow unidirectional pattern matching. In this work, we focus on the FM index (Ferragina and Manzini, 2000), which is based on the Burrows-Wheeler transform (BWT) of the search text T. It allows for the identification of exact occurrences of a query pattern P in T with a time complexity of $O(|P| + n_{occ})$ with $|P|$ the length of P and n_{occ} the number of occurrences of P in T. It does so by matching pattern P character by character in reverse order, from right to left. Baseline lossless approximate pattern matching with up to k errors is performed by exploring branches of the FM index that could potentially be matches. Branches are explored character by character for as long as they remain within the maximum allowed Hamming/Levenshtein distance of k errors with the corresponding part of query pattern P. When the number of errors exceeds the value of k, the branch is abandoned (called "backtracking"), and the search procedure continues with the next feasible branch that is yet unexplored. The problem is that (a) the number of branches to explore increases rapidly with the number of errors allowed and (b) the vast majority of branches that are explored turn out not to be matches. In other words, the search space, i.e., the feasible region of the full-text index in which occurrences of P could be found, is very large, rendering the backtracking procedure computationally unfeasible even for modest values of k.

[1]Department of Information Technology, Ghent University - imec, Ghent 9052, Belgium

[2]Department of Plant Biotechnology and Bioinformatics, Ghent University, Ghent 9052, Belgium

[3]Lead contact

*Correspondence: jan.fostier@ugent.be

https://doi.org/10.1016/j.isci.2021.102687

Bidirectional indexes (affix trees [Maaβ, 2000], affix arrays [Strothmann, 2007], and bidirectional FM indexes [Lam et al., 2009]) augment the functionality of their unidirectional counterparts by allowing to extend queries in either direction: a pattern P can be searched by starting at any position of P and extending the match either to the left or to the right in arbitrary order. Lam (Lam et al., 2009) was the first to note that this added functionality could significantly accelerate lossless approximate pattern matching in full-text indexes by leveraging the classical "pigeonhole principle": by partitioning P in $k+1$ non-overlapping parts with k the maximum number of allowed errors, it immediately follows that one part must be error free. By first performing an exact search for each of the $k+1$ parts and then extending that match with an approximate search, the search space is reduced significantly and computational gains are obtained. Indeed, the initial exact match avoids the computationally costly exploration of the densely branched region near the root of the search tree.

Kucherov et al. generalized these ideas and introduced the notion of "search schemes" (Kucherov et al., 2014). Search schemes define how a pattern P is matched using a bidirectional full-text index such that unsuccessful branches are discarded as quickly as possible, reducing the search space and hence runtime. We adopt their notation. Pattern P is partitioned into p (potentially uneven) parts P_i ($i = 1 \ldots p$). A *search* $\mathcal{S} = (\pi, L, U)$ is a triplet of arrays of size p where π is a permutation over $\{1, 2, \ldots, p\}$ that defines the order in which the parts P_i are processed. It must satisfy the connectivity property in that sense that a partial match can only be extended in a contiguous manner, either to the left or to the right. The arrays L and U define the lower and upper bound to the cumulative number of errors when each part is processed. The core idea is to only gradually increase the number of allowed errors when more parts of P are matched against the full-text index, significantly reducing the search space near the dense root of the search tree. To cover all possible error distributions over the length of a pattern, multiple "searches" are required that collectively form a search scheme. Kucherov et al. proposed a number of efficient search schemes for up to $k = 4$ errors using $p = k+1$ or $p = k+2$ parts. For example, for $k = 2$ errors and $p = k+1 = 3$ parts, they proposed a search scheme with three searches: $\mathcal{S}_1 = (123, 000, 022)$; $\mathcal{S}_2 = (321, 000, 012)$; and $\mathcal{S}_3 = (213, 001, 012)$. In the $\mathcal{S}_1$ search, exact matching is first performed for the leftmost part P_1. Next, this exact match is extended to the right, thus processing parts P_2 and P_3, using a backtracking procedure that allows up to two errors. In the $\mathcal{S}_2$ search, exact matching is first performed for the rightmost part P_3 and extended to the left by first allowing up to a single error in P_2 and then two errors in P_1. Indeed, occurrences of P with two errors in the middle part were already covered by search $\mathcal{S}_1$. Finally, search $\mathcal{S}_3$ first involves an exact matching of P_2, which is then extended to the left allowing a single error and finally to the right with at least one and at most two errors.

Kianfar et al. (Kianfar et al., 2018) further extended this work and used integer linear programming to generate additional search schemes for the Hamming distance metric. Additionally, they show that related work on lossless approximate pattern matching by Vroland et al. on 01*0 seeds (Vroland et al., 2016) can also be expressed as search schemes. Therefore, search schemes represent a flexible framework for lossless approximate pattern matching in which a multitude of algorithmic ideas can be expressed. For reference, all search schemes used in this paper are provided in Table S1.

In this paper, we propose an algorithm for the "dynamic partitioning" of a search pattern P based on its sequence content. We demonstrate that this technique reduces the search space and runtime by up to 28% and 25%, respectively, compared to a uniform partitioning of P. Dynamic partitioning can be universally applied to any kind of search scheme to boost its performance. We implemented this idea in Columba, an open-source tool written in standard C++11. Columba is almost 3.5 times faster than Bwolo (Vroland et al., 2016) for the task of identifying all occurrences of 101 bp Illumina reads in the human reference genome within an edit distance of $k = 4$ errors. Columba is available at https://github.com/biointec/columba under AGPL-3.0 license.

RESULTS AND DISCUSSION

Benchmarks

In all benchmarks, all occurrences of both strands of search patterns up to an edit distance (allowing substitutions and/or indels) of $k = \{1, 2, 3, 4\}$ were identified in the human reference genome (GRCh38) (Schneider et al., 2017). Non-ACGT characters (e.g., Ns) in the reference sequence were replaced by a randomly chosen nucleotide. Chromosomes were concatenated into a single string. It is therefore possible that a pattern is mapped across the boundary of adjacent chromosomes. Such matches can easily be filtered during post-processing.

We sampled two sets of each 100,000 Illumina HiSeq 2000 reads (101 bp) from a whole genome sequencing data set (accession no. ERR194147). The first set of reads was used only to determine suitable parameters (as described further) for the different partitioning strategies, for the different search schemes, and for different values of k. All benchmark results were obtained using the second set of reads, demonstrating that the empirically derived parameters generalize well to other data sets with similar characteristics. We also provide results for 100,000 search patterns of length 50 bp, randomly sampled from long Pacific Biosciences reads (accession no. SRR1304331).

All results were obtained using a single core of a 24-core AMD EPYC 7451 CPU running at a base clock frequency of 3.1 GHz. Each simulation was repeated 20 times. We report both the average wall clock time as well as the standard deviation. We report the fraction of search patterns with at least one match as well as the total number of non-redundant occurrences. We label an occurrence as redundant when its starting position is at most $2k+1$ nucleotides away from another match with an equal or lower edit distance.

Dynamic partitioning of search patterns reduces the search space

As noted by Kucherov et al. (Kucherov et al., 2014) the partitioning of P into equally sized parts (called "uniform" partitioning) is not necessarily optimal. This is because different searches might enumerate a different number of strings during the search procedure. Using the search scheme for $k=2$ errors from the introduction section as an example, it is clear that search $\mathcal{S}_1$ will on average be associated with a larger search space than $\mathcal{S}_2$ and $\mathcal{S}_3$ because $\mathcal{S}_1$ allows for two errors in the second part that is processed, whereas $\mathcal{S}_2$ and $\mathcal{S}_3$ allow only for a single error. By increasing the size of P_1, the number of exact matches of P_1 will generally decrease. Consequently, search $\mathcal{S}_1$ will enumerate fewer candidate occurrences. Conversely, by decreasing the size of P_2 and P_3, the search space to explore by $\mathcal{S}_2$ and $\mathcal{S}_3$ will increase. Due to the asymmetry between searches, the decrease in search space can be larger than the increase, thus improving the overall performance of the search scheme. In their paper, Kucherov et al. propose a dynamic programming algorithm to find the optimal part sizes, using a model that assumes a random search text T and a random search pattern P. In this work, we focus on the human genome as a reference genome and on search patterns that have a small edit distance to some subsequence of this reference genome. The assumption of randomness is not valid as the human genome has, unlike random sequences, a very complex repeat structure. Hence, we established optimal part sizes empirically using the first set of reads, as described in the previous section. We call this partitioning the "optimal static" partitioning. These optimal part sizes depend on characteristics of the search patterns (length, error rate, and distribution) as well as the characteristics of the reference genome itself (repeat structure).

Even though these part sizes may be optimal on average, they are not necessarily optimal for each individual search pattern P. For example, depending on the sequence content of P itself, some parts of P might have a very low number of exact matches (or even no matches), whereas other parts might have a very high number of exact matches. In turn, this will again translate into an uneven workload among searches. By reducing the size of the parts with few exact matches while increasing the size of parts with many exact matches, it is again possible to achieve a global reduction in workload. We propose an algorithm for the "dynamic partitioning" of P. The ideas are illustrated in Figure 1. Each part is initialized by matching a single character. The position of that character is taken as the middle position of the corresponding uniformly sized parts, except for the first and last part where we take, respectively, the first and last character of P. During each next step, the part with the highest number of exact occurrences (and which can still be extended) is selected, and it is extended by a single character in the direction (left or right) of the adjacent part that has the fewest number of exact occurrences. This procedure is repeated until all characters of P are assigned to some part. Intuitively, this greedy algorithm attempts to partition P such that each part is associated with an equal number of exact matches. The actual partitioning that is obtained depends on the sequence content of P itself and will therefore differ between search patterns, hence the name "dynamic partitioning". The overhead imposed by dynamic partitioning is very small: the SA/BWT ranges of the exact matches of the parts of P (see STAR methods: Bidirectional FM index) that emerge as a by-product of the procedure are stored and are re-used during the execution of the search schemes. Indeed, efficient search schemes consist of searches that first involve the exact matching of some part of P. Note that the dynamic partitioning algorithm in general extends parts both to the left and right and therefore takes full advantage of the functionality offered by the bidirectional FM index.

In its basic form, the dynamic partitioning algorithm yields parts with a roughly equal number of exact matches, similar to what is expected on average from uniform partitioning. However, in order to obtain the best results,

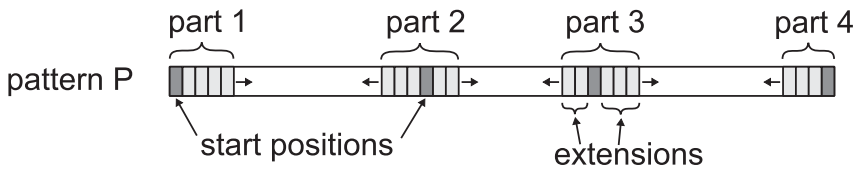


Figure 1. Dynamic partitioning of a search pattern *P* with four parts.
Each part is initialized by matching a single character (dark gray squares). The part with the largest number of exact occurrences is extended by a single character (light gray squares), either to the left or to the right.

one should aim to balance the relative number of exact matches among parts such that they correspond to what is expected on average from the optimal static partitioning. The dynamic partitioning algorithm is easily adapted to this task as follows: (1) we initialize the first character of each part as the center position of the optimal static partitioning (again, except for the first/last part); (2) we assign per-part weights and balance the weighted number of exact matches among parts. Intuitively, the first modification allows certain parts to "grow" more than others while the weights take into account the expected relative workload among searches. Again, we obtained these weights empirically using the first set of reads.

Table 1 shows the impact of different partitioning strategies for different values of the maximum allowed edit distance $k = \{1, 2, 3, 4\}$, using the search schemes proposed by Kucherov et al. with $k+1$ parts (Kucherov et al., 2014). Besides runtime, the number of nodes visited in the search tree as well as the number of computed elements in the banded edit distance matrix are shown. The former equals the number of times a partial match is extended by a single character c (in either direction). In practice, this involves expensive random memory access, and it is therefore a clear indication of intrinsic performance, regardless of the quality of implementation.

For $k = 1$ errors, the search scheme consists of two searches which are symmetric with respect to each other. Hence, the uniform and optimal static partitioning strategies both partition P into two equally sized parts and both show equal performance. The dynamic partitioning strategy reduces the number of nodes visited by 11.6%, demonstrating its ability to reduce the search space. However, because the overhead resulting from the dynamic partitioning procedure itself is larger than the gains from this smaller search space, the runtime increases slightly.

The search scheme for $k = 2$ errors is not symmetric; thus, we expect the optimal static partitioning to differ from the uniform partitioning. For $|P| = 101$, we find optimal part sizes of 41, 29, and 31, respectively. Table 1 shows that applying optimal static partitioning reduces both the number of nodes visited and the runtime by 9.8%. The use of dynamic partitioning yields a notable reduction in search space and runtime of, respectively, 28.2% and 20.4%, both with respect to uniform partitioning. Clearly, the ability to partition patterns based on their sequence content results in a significant computational gain.

For $k = 3$ errors, the search scheme is only asymmetric in the lower bounds. We find that the optimal part sizes are uniform for $|P| = 101$; hence, optimal static partitioning and uniform partitioning yield identical results. However, dynamic partitioning again reveals superior performance with a runtime reduction of 17.1%.

The search scheme for $k = 4$ is highly asymmetric. It consists of eight searches, three of which start from an exact match of part P_1. Therefore, it is beneficial to increase the size of P_1, such that the number of exact matches of this part is reduced. We find optimal part sizes of 27, 20, 15, 19, and 19 that yield a reduction in runtime of 11.1%. Dynamic partitioning, however, is able to further reduce runtime by 24.7%, while the number of nodes visited is reduced by 28.3%. Also for other search schemes, we found that dynamic partitioning reduces the search space and runtime to a similar extent (Tables S2–S6). The same holds for the search patterns sampled from Pacific Biosciences reads (Table S7). We conclude that dynamic partitioning can be universally applied to boost the performance of search schemes.

Memory interleaving of bit vectors reduces runtime

To extend a partial match by a single character c, the FM index relies on $\mathrm{occ}(c, p)$ queries on the BWT to return the number of occurrences of character c in the prefix $\mathrm{BWT}[0, p)$ (see STAR methods: Bidirectional FM index). Given our focus on DNA sequences, which have alphabet size $|\Sigma| = 4$, we realize this using four

Table 1. Comparison of different partitioning strategies when identifying all occurrences of both strands of 100,000 Illumina reads in the human reference genome for different values of the maximum allowed edit distance k using the search schemes by Kucherov et al. with $k+1$ parts.

Partitioning strategy	Wall clock time ± SD		No. of nodes visited (search space)		No. of matrix elements computed	
k = 1, non-redundant matches = 959,844, reads mapped 93.6%						
Uniform	6.73 ± 0.18 s		29,212,674		45,697,288	
Optimal static	6.73 ± 0.18 s	(+0.0%)	29,212,674	(+0.0%)	45,697,288	(+0.0%)
Dynamic	7.23 ± 0.14 s	(+7.4%)	25,829,974	(− 11.6%)	39,391,011	(− 13.8%)
k = 2, non-redundant matches = 2,329,746, reads mapped 96.0%						
Uniform	22.32 ± 0.15 s		104,476,617		339,010,141	
Optimal static	20.14 ± 0.12 s	(− 9.8%)	94,289,677	(− 9.8%)	282,148,209	(− 16.8%)
Dynamic	17.76 ± 0.13 s	(− 20.4%)	74,978,120	(− 28.2%)	228,389,750	(− 32.6%)
k = 3, non-redundant matches = 4,606,995, reads mapped 97.0%						
Uniform	55.19 ± 0.44 s		276,027,753		1,068,947,201	
Optimal static	55.27 ± 0.40 s	(+0.1%)	276,027,753	(+0.0%)	1,068,947,201	(+0.0%)
Dynamic	45.73 ± 0.39 s	(− 17.1%)	212,965,491	(− 22.8%)	816,519,456	(− 23.6%)
k = 4, non-redundant matches = 7,997,221, reads mapped 97.7%						
Uniform	215.37 ± 0.96 s		1,154,125,425		4,941,009,260	
Optimal static	191.45 ± 1.46 s	(− 11.1%)	1,012,354,507	(− 12.3%)	4,272,355,550	(− 13.5%)
Dynamic	162.14 ± 0.99 s	(− 24.7%)	827,521,618	(− 28.3%)	3,607,583,286	(− 27.0%)

The percentage values indicate the relative increase or decrease with respect to uniform partitioning. See also Tables S2–S7.

bit vectors B_c with constant time rank support. Then, $\mathrm{occ}(c,p) = B_c.\mathrm{rank}_1(p) = \sum_{0 \le i < p} B_c[i]$, i.e., the number of 1-bits in the first p positions of B_c. For 64-bit CPU architectures, the rank9 algorithm (Vigna, 2008) has attractive properties. Briefly, a bit vector of size n bits is stored in $n/64$ 64-bit words. For every p that is a multiple of 512 bits, the pre-computed "first-level count" $\mathrm{rank}_1(p)$ is stored as a 64-bit value. Each first-level count is associated with seven additional pre-computed "second-level counts" that contain values $\mathrm{rank}_1(p+64k) - \mathrm{rank}_1(p)$, for $1 \le k \le 7$. These second-level counts are stored as seven 9-bit values within a 64-bit word. Rank operations $\mathrm{rank}_1(p)$ for arbitrary p can then be answered in constant time by adding three contributions: (1) the appropriate first-level and (2) second-level counts; (3) a popcount(w) instruction to count the number of 1-bits in w, the word of the bit vector that contains position p and for which the bits at positions p and higher were masked to zero. The memory overhead of the first- and second-level counts amounts to only 25% of the bit vector data.

Vigna (Vigna, 2008) proposed to store the first- and second-level counts in an "interleaved" manner: the two words that hold corresponding first- and second-level counts are located next to each other in memory. When loading the cache line that contains first-level count information, the second-level count information is retrieved as well, thus reducing the number of cache misses. Gog and Petri (Gog and Petri, 2014) proposed to additionally interleave the pre-computed count information with the bit vector data itself. In that case, all information to answer a rank query is stored on either a single cache line or two adjacent cache lines.

In the context of search schemes (or more generally: backtracking algorithms), the search tree is explored by extending a partial match with each character $c \in \Sigma$. Hence, within a short time duration, different calls to $\mathrm{occ}(c,p)$ are made for fixed p and for all $c \in \Sigma$. In order to maximally fill a cache line (64 bytes or eight words on typical x86 architectures) with relevant information, we propose to interleave the data related to different bit vectors B_c as shown in Figure 2. In the case of DNA sequences, all four calls to $\mathrm{occ}(c,p)$ with $c = \{A, C, G, T\}$ can then be answered using 12 words of data. By using 64-byte aligned vectors to store the interleaved bit vector data and pre-computed counts, we guarantee that only two cache lines are required for four occ calls. For the same task, when using four non-interleaved bit vectors, at least four cache lines would be required when these individual bit vectors are stored using the scheme by Gog

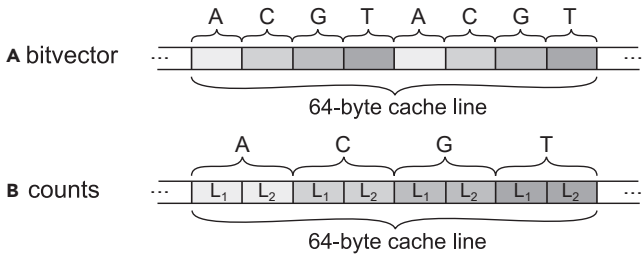


Figure 2. Interleaved storage of bit vectors B_c and associated first- and second-level counts for $\Sigma = \{A, C, G, T\}$.
Four calls to occ(c, p) for fixed p and all $c \in \Sigma$ require the data of only two cache lines.

and Petri while eight cache lines would be required using Vigna's storage scheme. For large genomes, due to the fact that p takes unpredictable values, most of these cache lines have to be retrieved from main memory, a task that requires ~100 ns (equivalent to 200–300 CPU cycles) on modern CPU architectures. Thus, reducing the number of cache lines that have to be retrieved from memory improves performance.

We compare our proposed memory storage scheme for bit vectors to the scheme proposed by Vigna. For the sake of simplicity, we refer to these methods as "interleaved" and "non-interleaved", respectively, even though also in Vigna's scheme, first- and second-level counts are interleaved. Table 2 shows the runtime for both approaches for the different partitioning strategies using the search scheme by Kucherov et al. for $k = 4$ errors and $k + 1 = 5$ parts. Runtime is considerably reduced by 35%–37%. Note that the memory storage scheme is independent from the partitioning scheme, and hence, both techniques can be combined. Collectively, they reduce runtime from 342 s (uniform partitioning with non-interleaved bit vectors) to 162 s (dynamic partitioning with interleaved bit vectors), a reduction of 53%.

Reducing redundancy for the edit distance metric reduces the search space

The use of the edit distance metric inherently results in a certain degree of redundancy. An occurrence of a pattern P may be reported multiple times with slightly different start and/or end positions in T. For example, if P occurs as an exact match in T and k errors are allowed, then $\sum_{i=1\ldots k} 4i = 2k(k+1)$ redundant occurrences O will be found for which the alignments between P and O have leading and/or trailing gaps in P and/or O. Hence, the degree of redundancy increases rapidly with the number of errors allowed.

As searches of search schemes align a pattern P part by part, the same redundancy occurs when dealing with partial matches of P. Consider a search $\mathcal{S} = (\pi, L, U)$. If all partial occurrences up to part $P_{\pi[i]}$ with an edit distance of at most $U[i]$ are reported, then an alignment procedure for part $P_{\pi[i+1]}$ will be started for each such occurrence, even though some of them may be redundant. In STAR methods, we provide techniques to reduce redundant computations and thus improve performance.

We compare an optimized implementation, which implements the ideas from STAR methods, to a naive implementation. In this naive implementation, an alignment procedure is started for each partial occurrence, even though some of them may be redundant. In Table 3, both approaches are compared for different values of k. For $k = 1$, the difference in runtime, number of nodes, and matrix elements is negligible. For $k = 2$, the search space is reduced by 26%, leading to a reduction in runtime by 13%. For $k = 3$, even larger reductions of respectively 46% (search space) and 33% (runtime) are found. This is to be expected, as the redundancy grows quadratically with k. For $k = 4$, the optimized implementation reduces the search space by 63% and the runtime by 52%. We conclude that for higher values of k, an optimized implementation to handle redundancy associated with edit distance metric computations is essential to achieve high computational performance.

Columba has higher computational performance than other state-of-the-art tools

We benchmarked the different search schemes that have been proposed in literature in our implementation (called Columba). Kucherov et al. proposed schemes with $k + 1$ and $k + 2$ parts (Kucherov et al., 2014). Kianfar et al. generated optimal search schemes for the Hamming distance metric (Kianfar et al., 2018). For $k = 3$ and $k = 4$ errors, they contain searches that already allow one or two errors in the first part of P that is matched. Hence, we found that these search schemes are not competitive when the edit distance metric is

Table 2. The wall clock time for mapping both strands of 100,000 Illumina patterns using interleaved and non-interleaved bit vectors for $k = 4$ errors and the search scheme by Kucherov et al. with $k + 1 = 5$ parts.

Bit vectors	Uniform partitioning		Optimal static partitioning		Dynamic partitioning	
Non-interleaved	342.02 ± 2.71 s		295.15 ± 2.93 s		249.49 ± 2.19 s	
Interleaved	215.37 ± 0.96 s	(− 37.2%)	191.45 ± 1.46 s	(− 35.1%)	162.14 ± 0.99 s	(− 35.0%)

used. From the same research group, Pockrandt et al. derived an alternative search scheme for $k = 4$ errors referred to as Man_{Best} (Pockrandt, 2019). We also include the search schemes based on the pigeonhole principle as well as those based on the 01*0 seeds (Vroland et al., 2016; Kianfar et al., 2018). For the latter, a dedicated tool called Bwolo was developed. In other papers, Bwolo was found to be the fastest available method for the edit distance metric (Kianfar et al., 2018; Pockrandt, 2019). For all search schemes, dynamic partitioning of search patterns was used with optimal parameters. We also compared our implementation to the implementation of search schemes in SeqAn 3. In our hands, the runtime was found to be orders of magnitude larger than our implementation. Therefore, no results for SeqAn 3 are reported.

Table 4 lists the running times. For efficient search schemes, Columba shows superior performance to Bwolo: for $k = 4$ errors and using the Kucherov $k + 1$ search scheme, Columba is almost four times faster than Bwolo. Particularly, even when Bwolo's 01*0 search strategy is used within Columba, a significant performance difference is revealed. Bwolo uses a unidirectional search index, whereas Columba relies on a bidirectional search functionality. Therefore, Bwolo has to perform in-text verification to validate candidate matches, whereas Columba performs the matching entirely within the index structure itself. Nevertheless, as also noted in (Kianfar et al., 2018), in-text verification can in certain cases be faster.

Interestingly, in our implementation and using our data set, the original $k + 1$ search schemes proposed by Kucherov et al. yield the highest performance. Nevertheless, the performance difference with the Man_{Best}, Kucherov $k + 2$, and 01*0 schemes is relatively small, and other data sets may yield different conclusions. Remarkably, the performance differences between these four efficient search schemes are about as large as the performance differences between partitioning strategies. Therefore, it might be worthwhile to further investigate novel partitioning strategies as well as novel search schemes.

We also benchmarked the runtime of Columba with uniform partitioning, non-interleaved bit vectors, and a naive edit distance implementation (i.e., without any of the improvements proposed in this paper) for $k = 4$

Table 3. Comparison of a naive and optimized strategy for handling the redundancy associated to the edit distance metric for different values of k.

Edit distance redundancy	Wall clock time ± SD		No. of nodes visited (search space)		No. of matrix elements computed	
$k = 1$, non-redundant matches = 959,844, reads mapped 93.6%						
Naive	7.19 ± 0.10 s		26,102,601		39,697,328	
Optimized	7.23 ± 0.14 s	(+0.6%)	25,829,974	(− 1.0%)	39,391,011	(− 0.8%)
$k = 2$, non-redundant matches = 2,329,746, reads mapped 96.0%						
Naive	20.50 ± 0.18 s		101,087,547		297,657,004	
Optimized	17.76 ± 0.13 s	(− 13.4%)	75,164,043	(− 25.6%)	227,670,435	(− 23.5%)
$k = 3$, non-redundant matches = 4,606,995, reads mapped 97.0%						
Naive	67.72 ± 0.52 s		393,769,104		1,389,115,919	
Optimized	45.73 ± 0.39 s	(− 32.5%)	212,965,491	(− 45.9%)	816,519,456	(− 41.2%)
$k = 4$, non-redundant matches = 7,997,221, reads mapped 97.7%						
Naive	335.67 ± 1.94 s		2,247,115,246		8,250,593,095	
Optimized	162.14 ± 0.99 s	(− 51.7%)	827,521,618	(− 63.2%)	3,607,583,286	(− 56.3%)

Both strands of 100,000 Illumina reads are mapped, using the search schemes by Kucherov et al. with $k + 1$ parts, dynamic partitioning, and interleaved bit vectors.

Table 4. The runtime for different search schemes that we support in our tool Columba versus a state-of-the-art tool Bwolo for different values of *k*.

Search scheme/tool	$k = 1$	$k = 2$	$k = 3$	$k = 4$
Pigeonhole principle	7.13 ± 0.04 s	24.13 ± 0.15 s	144.88 ± 0.99 s	722.21 ± 5.72 s
Kucherov $k+1$	7.23 ± 0.14 s	17.76 ± 0.13 s	45.73 ± 0.39 s	162.14 ± 0.99 s
Kucherov $k+2$	7.23 ± 0.09 s	20.93 ± 0.12 s	58.61 ± 0.35 s	195.52 ± 1.57 s
01*0 principle	7.22 ± 0.09 s	18.50 ± 0.10 s	68.67 ± 0.38 s	241.17 ± 2.07 s
Kianfar	7.11 ± 0.06 s	17.05 ± 0.11 s	152.54 ± 1.19 s	1994.92 ± 19.69 s
Man_{Best}	NA	NA	NA	192.56 ± 1.35 s
Bwolo	12.68 ± 0.37 s	35.87 ± 0.25 s	123.60 ± 0.33 s	602.14 ± 0.77 s

For all search schemes, dynamic partitioning was used.

and using the search scheme by Kucherov et al. with $k+1=5$ parts. This resulted in a wall clock time of 560.64 s, almost 3.5 times slower than the runtime of 162.14 s of Columba with all improvements in place (see Table 1).

Note that Columba, like most alignment tools, can easily take advantage of multi-core and/or multi-CPU architectures by aligning reads on different cores in parallel.

SUMMARY

In this paper, we made three contributions, which together form Columba. We proposed an algorithm for the "dynamic partitioning" of a search pattern *P* based on its sequence content. We demonstrate that for the task of mapping 100 bp Illumina reads to the human reference genome, this technique reduces the search space and runtime by up to 28% and 25%, respectively, compared to a uniform partitioning of *P*. Dynamic partitioning can be universally applied to any kind of search scheme to boost its performance. Additionally, we proposed a new strategy to interleave bit vector representations of the BWT in memory that is specifically tailored to search schemes. This leads to fewer cache misses and reduces runtime by 35%. Again, this technique can be universally applied to all search schemes. Finally, we provided an analysis and discussion on how to maximally avoid redundancy that is inherently associated with edit distance computations. Avoiding this redundancy reduces the search space by up to 63% and the runtime by 52%. Collectively, these techniques reduce the runtime with a factor of almost 3.5. Columba supports different search schemes: the pigeonhole-based schemes, the search schemes with $k+1$ and $k+2$ parts as proposed by Kucherov et al. (Kucherov et al., 2014), the schemes proposed by Kianfar et al. (Kianfar et al., 2018), and the 01*0 seeds by Vroland et al. (Vroland et al., 2016). We demonstrated that our implementation is almost four times faster than Bwolo (Vroland et al., 2016) for the task of identifying all occurrences of 100 bp Illumina reads in the human reference genome within an edit distance of $k=4$ errors. Columba is available at https://github.com/biointec/columba under AGPL-3.0 license.

LIMITATIONS OF THE STUDY

This work considers only lossless approximate pattern matching algorithms. Columba is not compared against state-of-the-art tools that use lossy approximate pattern matching algorithms. Only the edit distance (i.e., Levenshtein) metric is considered and not the Hamming distance metric. Columba cannot find occurrences under a more generic scoring scheme with arbitrary match, mismatch, and gap scores/penalties. Search schemes have been proposed in literature for up to $k=4$ errors. It is still an open research question how efficient search schemes for higher values of *k* can be designed. We have benchmarked Columba using only patterns of length 101 (Illumina) and length 50 (Pacific Biosciences). Performance and relative performance differences with respect to other tools may vary for other pattern lengths. We have benchmarked only using the human reference genome as a search text.

STAR★METHODS

Detailed methods are provided in the online version of this paper and include the following:

- KEY RESOURCES TABLE
- RESOURCE AVAILABILITY
  - Lead contact

SUPPLEMENTAL INFORMATION

Supplemental information can be found online at https://doi.org/10.1016/j.isci.2021.102687.

ACKNOWLEDGMENTS

The authors received no specific funding for this work.

AUTHOR CONTRIBUTIONS

L.R. and J.F. designed and implemented the algorithms. L.R. performed all benchmarks. K.M. and J.F. supervised the study. All authors have written and approved the manuscript.

DECLARATION OF INTERESTS

The authors declare no competing interests.

Received: March 19, 2021
Revised: May 17, 2021
Accepted: May 28, 2021
Published: July 23, 2021

REFERENCES

Abouelhoda, M.I., Kurtz, S., and Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithms *2*, 53–86. https://doi.org/10.1016/S1570-8667(03)00065-0.

Altschul, S.F., Gish, W., Miller, W., Myers, E.W., and Lipman, D.J. (1990). Basic local alignment search tool. J. Mol. Biol. *215*, 403–410. https://doi.org/10.1016/S0022-2836(05)80360-2.

Burrows, M., and Wheeler, D. (1994). A block-sorting lossless data compression algorithm. Technical report. Digit. Syst. Res. Cent. https://doi.org/10.1109/DCC.1997.582009.

Ferragina, P., and Manzini, G. (2000). Opportunistic data structures with applications. In Proceedings 41st Annual Symposium on Foundations of Computer Science, pp. 390–398. https://doi.org/10.1109/SFCS.2000.892127.

Gog, S., and Petri, M. (2014). Optimized succinct data structures for massive data. Softw. Pract. Exp. *44*, 1287–1314. https://doi.org/10.1002/spe.2198.

Gusfield, D. (2007). Algorithms on Strings, Trees, and Sequences : Computer Science and Computational Biology (Cambridge Univ. Press). https://doi.org/10.1017/CBO9780511574931.

Kent, W.J. (2002). BLAT – the BLAST-like alignment tool. Genome Res. *12*, 656–664. https://doi.org/10.1101/gr.229202.

Kianfar, K., Pockrandt, C., Torkamandi, B., Luo, H., and Reinert, K. (2018). Optimum search schemes for approximate string matching using bidirectional fm-index. arXiv, 1711.02035.

Kucherov, G., Salikhov, K., and Tsur, D. (2014). Approximate string matching using a bidirectional index. In Combinatorial Pattern Matching, A.S. Kulikov, S.O. Kuznetsov, and P. Pevzner, eds. (Springer International Publishing), pp. 222–231. https://doi.org/10.1007/978-3-319-07566-2_23.

Lam, T., Li, R., Tam, A., Wong, S., Wu, E., and Yiu, S. (2009). High throughput short read alignment via bi-directional BWT. In IEEE International Conference on Bioinformatics and Biomedicine, pp. 31–36. https://doi.org/10.1109/BIBM.2009.42.

Li, H., and Durbin, R. (2009). Fast and accurate short read alignment with Burrows–Wheeler transform. Bioinformatics *25*, 1754–1760. https://doi.org/10.1093/bioinformatics/btp324.

Maaß, M.G. (2000). Linear bidirectional on-line construction of affix trees. In Combinatorial Pattern Matching, R. Giancarlo and D. Sankoff, eds. (Springer Berlin Heidelberg), pp. 320–334. https://doi.org/10.1007/s00453-003-1029-2.

Masek, W.J., and Paterson, M. (1980). A faster algorithm computing string edit distances. J. Comput. Syst. Sci. *20*, 18–31. https://doi.org/10.1016/0022-0000(80)90002-1.

Pockrandt, C.M. (2019). Approximate String Matching: Improving Data Structures and Algorithms. Ph.D. thesis (Freien Universität Berlin). https://doi.org/10.17169/refubium-2185.

Pockrandt, C., Ehrhardt, M., and Reinert, K. (2017). EPR-dictionaries: a practical and fast data structure for constant time searches in unidirectional and bidirectional FM-indices. Lect. Notes Comput. Sci. *10229*. https://doi.org/10.1007/978-3-319-56970-3_12.

Schneider, V., Graves-Lindsay, T., Howe, K., Bouk, N., Chen, H.C., Kitts, P., Murphy, T., Pruitt, K., Thibaud-Nissen, F., Albracht, D., et al. (2017). Evaluation of GRCh38 and de novo haploid genome assemblies demonstrates the enduring quality of the reference assembly. Genome Res. *27*. https://doi.org/10.1101/gr.213611.116.

Strothmann, D. (2007). The affix array data structure and its applications to RNA secondary structure analysis. Theor. Comput. Sci. *389*, 278–294. https://doi.org/10.1016/j.tcs.2007.09.029.

Vigna, S. (2008). Broadword implementation of rank/select queries. In Experimental Algorithms, C.C. McGeoch, ed. (Springer Berlin Heidelberg), pp. 154–168. https://doi.org/10.1007/978-3-540-68552-4_12.

Vroland, C., Salson, M., Bini, S., and Touzet, H. (2016). Approximate search of short patterns with high error rates using the 01*0 lossless seeds. J. Discrete Algorithms *37*, 3–16. https://doi.org/10.1016/j.jda.2016.03.002.

STAR★METHODS

KEY RESOURCES TABLE

REAGENT or RESOURCE	SOURCE	IDENTIFIER
Deposited data		
Illumina sequencing data	EBI ENA	ERR194147
Pacific Biosciences sequencing data	EBI ENA	SRR1304331
Software and algorithms		
Columba	This paper	https://github.com/biointec/columba
Bwolo	(Vroland et al., 2016)	https://bioinfo.lifl.fr/olo/

RESOURCE AVAILABILITY

Lead contact

Further information and requests for resources should be directed to and will be fulfilled by the lead contact, Jan Fostier jan.fostier@ugent.be.

Materials availability

This study did not generate new unique reagents.

Data and code availability

The data sets used in this study are derived from publicly available data. The first data set consists of 100,000 randomly sampled reads from an Illumina sequencing experiment (EBI ENA, accession no. ERR194147). The sampled reads are available at: https://github.com/biointec/columba/releases/download/example/sampled_illumina_reads.fastq. The second data set consists of 100,000 subreads of length 50 randomly sampled from a Pacific Biosciences sequencing experiment (EBI ENA, accession no. SRR1304331). The sampled version is available at: https://github.com/biointec/columba/releases/download/example/sampled_pacbio_seeds.fastq.

The code generated during this study is available at https://github.com/biointec/columba.

METHOD DETAILS

Bidirectional FM index

We use standard notation on strings, zero-based array indexing and half-open intervals [.,.). The Burrows-Wheeler transform BWT$[0, n)$ of a text $T[0, n)$ of size n that ends with a unique (and lexicographically smallest) sentinel character \$ is defined as BWT$[i] = T[\text{SA}[i] - 1]$ if SASA$[i]>0$ and BWT$[i]$ = \$ otherwise (Burrows and Wheeler, 1994). SA denotes the suffix array, an array of size n over integer values that indicate the starting positions of the suffixes of T in lexicographic order. We need support for occ(c, i) queries on the BWT that return the number of occurrences of character c in the prefix BWT$[0, i)$. We realize this through $|\Sigma|$ bit vectors with constant-time rank support. Here, $|\Sigma|$ denotes the size of the alphabet (e.g., $|\Sigma| = 4$ for DNA sequences). Exact pattern matching is then performed by matching character by character, from right to left. Let $[i, j)$ denote the interval over the suffix array for which the corresponding suffixes have P as a prefix. The suffix array interval $[i', j')$ whose suffixes have cP as a prefix can then be computed by $i' = C(c) + \text{occ}(c, i)$ and $j' = C(c) + \text{occ}(c, j)$. Here, $C(c)$ denotes the number of characters in BWT$[0, n)$ strictly smaller than c. These are pre-computed and stored in a small array of size $|\Sigma|$. As occ(c, i) queries can be performed in constant time, exact matching of a pattern P takes $O(|P|)$ time. The size of the obtained interval $[i, j)$ denotes the number of occurrences of P in T. The actual positions of the occurrences in T are found using the suffix array. The full-text index that comprises a BWT representation and auxiliary tables is called the FM index and may occupy as little as 2-4 bits of memory per character for DNA sequences (Ferragina and Manzini, 2000).

A *bidirectional* FM index is realized by also storing BWT_r, the Burrows-Wheeler transform of the reverse of T. By keeping track of both the range $[i, j)$ over the BWT as well as the range $[i', j')$ over BWT_r in a synchronized

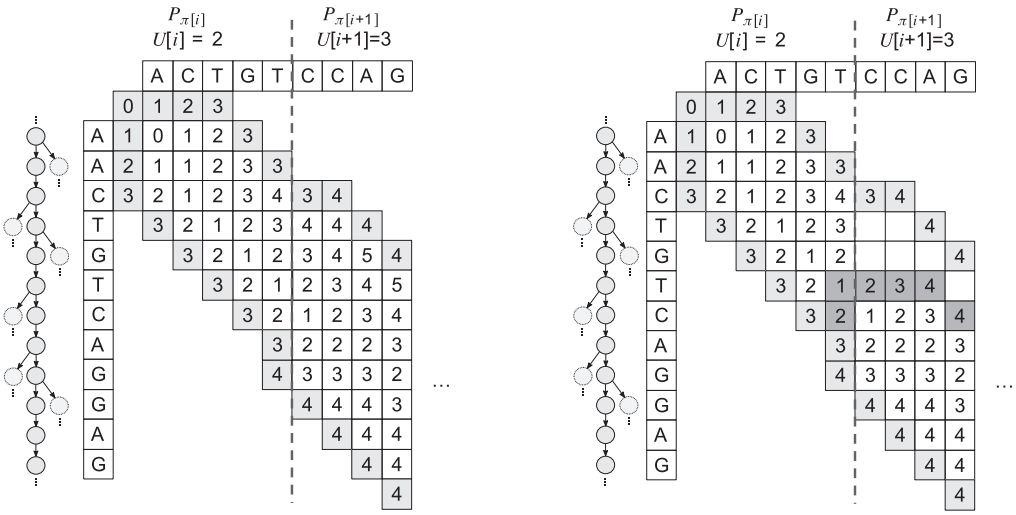


Figure 3. A banded alignment matrix (AM) for which the allowed edit distance increases between parts

Left: The gray-shaded cells are set during initialization, whereas the white-shaded cells are completed during the execution of the search procedure. Search pattern P is depicted horizontally, whereas a branch of the index is depicted vertically. Right: the AM for part $P_{\pi[i+1]}$ is initialized around the uppermost cluster center of the final column of part $P_{\pi[i]}$.

manner (Lam et al., 2009), one can extend a pattern P to both cP as well as Pc. By replacing the 'Occ' data structure with a 'Prefix-Occ' structure, this can be realized in constant time (Pockrandt et al., 2017).

Reducing redundancy for the edit distance metric

Consider a search $\mathcal{S} = (\pi, L, U)$. If all partial occurrences up to part $P_{\pi[i]}$ with an edit distance of at most $U[i]$ are reported, then an alignment procedure for part $P_{\pi[i+1]}$ will be started for each such occurrence, even though some of them may be redundant. If the search procedure does not change direction in between parts $P_{\pi[i]}$ and $P_{\pi[i+1]}$, one can achieve this by increasing the width of the banded alignment matrix (AM) from $2U[i]+1$ to $2U[i+1]+1$ as exemplified in Figure 3 (left).

This approach can be improved upon by taking into account the actual edit distance values observed in the final column of the AM of part $P_{\pi[i]}$. It may then be possible to reduce the width of the band of the next part $P_{\pi[i+1]}$. For this, we introduce the notion of a *cluster*. We remind the reader that adjacent cells on a row or column of the AM differ by a value of at most one (see Lemma 3 of (Masek and Paterson, 1980)). A cluster $\{s, c, e\}$ ($s \leq c < e$) at column j of the AM contains all elements $\mathrm{AM}[i, j]$ with $s \leq i < e$ for which it holds that $\mathrm{AM}[i, j] = \mathrm{AM}[c, j] + |c - i|$. Cell $\mathrm{AM}[c, j]$ is called the center of the cluster and can be thought of as a local minimum. Figure 4 (left) illustrates the clusters of a column of the AM.

Lemma 1.

Consider the edit distance alignment of strings X and Y and consider the clusters of some column of the AM. For each optimal alignment path that passes through one or more cells of cluster $\{s, c, e\}$ and that does not pass through the center, an alternative optimal alignment path exists that passes through the center of that cluster.

Proof Case 1: the optimal alignment path passes through cell $\mathrm{AM}[i, j]$, with $s \leq i < c$ (see Figure 4 right). By definition of a cluster, it holds that $\mathrm{AM}[i, j] = \mathrm{AM}[c, j] + (c - i)$. Assume that $\mathrm{AM}[c, k]$ is the leftmost cell on row c through which the optimal alignment path passes. Because the edit distance can only increase along an alignment path, $\mathrm{AM}[c, k] \geq \mathrm{AM}[i, j]$, hence $\mathrm{AM}[c, k] \geq \mathrm{AM}[c, j] + (c - i)$ (1). Because also $\mathrm{AM}[c, k] \leq \mathrm{AM}[c, j] + (k - j)$ (2), it follows from (1) and (2) that $(k - j) \geq (c - i)$. The optimal alignment path between matrix cells $\mathrm{AM}[i, j]$ and $\mathrm{AM}[c, k]$ entails exactly $(k - j) - (c - i)$ gaps. It then follows that $\mathrm{AM}[c, k] \geq \mathrm{AM}[i, j] + (k - j) - (c - i)$ and hence, $\mathrm{AM}[c, k] \geq \mathrm{AM}[c, j] + (k - j)$. Together with inequality (2), it follows that $\mathrm{AM}[c, k] = \mathrm{AM}[c, j] + (k - j)$. Therefore, an alternative optimal alignment path runs from the origin to $\mathrm{AM}[c, j]$; from $\mathrm{AM}[c, j]$ to $\mathrm{AM}[c, k]$ and then proceeds in an identical manner as the original path.

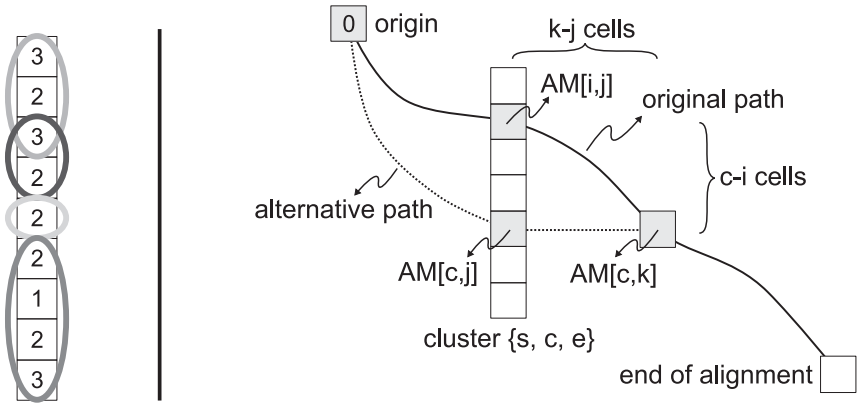


Figure 4. Illustrations related to Lemma 1.

Left: The clusters of a column of the AM are encircled. Note that a cell can be part of two adjacent clusters and that a cluster can consist out of only a single cell. Right: Illustration of proof of Lemma 1, case 1.

Case 2: the optimal alignment path passes through cell AM$[i,j]$, with $c<i<e$. Because of the definition of a cluster, it holds that AM$[i,j]$ = AM$[c,j] + (i - c)$. The value AM$[i,j]$ can thus always originate from AM$[c,j]$ through a vertical path, as the difference in rows between these two cells is exactly $(i - c)$. Again, an alternative optimal alignment path exists from the origin to cell AM$[c,j]$; from AM$[c,j]$ to AM$[c,i]$; and then proceeds in an identical manner as the original path. ∎

We leverage Lemma 1 to save computations: when part $P_{\pi[i]}$ is processed, we identify the uppermost cluster center in the last column of the alignment matrix of part $P_{\pi[i]}$ that has a value between $L[i]$ and $U[i]$. The cells above this cluster center can be ignored: even when they take part in some optimal alignment path, there will always be an alternative optimal path that passes through the center cell (lemma 1). By ignoring these cells, the banded alignment matrix for the next part may be initialized with a reduced width and the number of edit cells to be computed in the next part is reduced. In Figure 3 (right), this is exemplified.

In case the search changes direction in between parts $P_{\pi[i]}$ and $P_{\pi[i+1]}$ the same redundancy problem arises. By reporting *all* approximate (partial) matches with an edit distance of at most $U[i]$, multiple, possibly redundant alignments will be started in the other direction. Again, this can be mitigated using the cluster concept. Two cases exist. First, assume part $P_{\pi[i]}$ is either the leftmost or rightmost part of pattern P. This implies that the partial matches will no longer be extended in the current direction. Instead of reporting *all* partial occurrences with an edit distance of at most $U[i]$, we report only the partial match associated to the cluster center with the lowest value. If multiple such centers exist, then the uppermost one is reported, corresponding to the shortest (partial) match. In the second case, if part $P_{\pi[i]}$ is neither the leftmost or rightmost part of pattern P, partial matches are first extended in the *other* direction before resuming the extension in the *present* direction. In this case, we cannot *a priori* assume that an optimal alignment path of the entire pattern P (if such alignment exists) will pass through the cluster center with the lowest value. However, due to lemma 1, we know that such optimal alignment path (or an alternative, equivalent path) *will* pass through *one* of the cluster centers. Therefore, when part $P_{\pi[i]}$ has been processed, we extend (in the other direction) *only* the FM-range corresponding to the *deepest* point in present branch with the knowledge that this partial match will ultimately turn out to have an edit distance between the lowest and highest cluster center values. This information is valuable when checking the $L[i+1]$ and $U[i+1]$ bounds during the processing of part $P_{\pi[i+1]}$.