A Succinct Four Russians Speedup for Edit Distance Computation and One-against-many Banded Alignment

Brian Brubach[1]
Department of Computer Science, University of Maryland, College Park, MD 20742, USA
bbrubach@cs.umd.edu
https://orcid.org/0000-0003-1520-2812

Jay Ghurye[2]
Department of Computer Science, University of Maryland, College Park, MD 20742, USA
jayg@cs.umd.edu
https://orcid.org/0000-0003-1381-4081

Abstract

The classical Four Russians speedup for computing edit distance (a.k.a. Levenshtein distance), due to Masek and Paterson [15], involves partitioning the dynamic programming table into k-by-k square blocks and generating a lookup table in $O(\psi^{2k}k^2|\Sigma|^{2k})$ time and $O(\psi^{2k}k|\Sigma|^{2k})$ space for block size k, where ψ depends on the cost function (for unit costs $\psi = 3$) and $|\Sigma|$ is the size of the alphabet. We show that the $O(\psi^{2k}k^2)$ and $O(\psi^{2k}k)$ factors can be improved to $O(k^2 \lg k)$ time and $O(k^2)$ space. Thus, we improve the time and space complexity of that aspect compared to Masek and Paterson [15] and remove the dependence on ψ.

We further show that for certain problems the $O(|\Sigma|^{2k})$ factor can also be reduced. Using this technique, we show a new algorithm for the fundamental problem of one-against-many banded alignment. In particular, comparing one string of length m to n other strings of length m with maximum distance d can be performed in $O(nm + md^2 \lg d + nd^3)$ time. When d is reasonably small, this approaches or meets the current best theoretic result of $O(nm + nd^2)$ achieved by using the best known pairwise algorithm running in $O(m + d^2)$ time [17, 22] while potentially being more practical. It also improves on the standard practical approach which requires $O(nmd)$ time to iteratively run an $O(md)$ time pairwise banded alignment algorithm.

Regarding pairwise comparison, we extend the classic result of Masek and Paterson [15] which computes the edit distance between two strings in $O(m^2/\log m)$ time to remove the dependence on ψ even when edits have arbitrary costs from a penalty matrix. Crochemore, Landau, and Ziv-Ukelson [8] achieved a similar result, also allowing for unrestricted scoring matrices, but with variable-sized blocks. In practical applications of the Four Russians speedup wherein space efficiency is important and smaller block sizes k are used (notably $k < |\Sigma|$), Kim, Na, Park, and Sim [13] showed how to remove the dependence on the alphabet size for the unit cost version, generating a lookup table in $O(3^{2k}(2k)!k^2)$ time and $O(3^{2k}(2k)!k)$ space. Combining their work with our result yields an improvement to $O((2k)!k^2 \lg k)$ time and $O((2k)!k^2)$ space.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases edit distance, banded alignment, one-against-many alignment, genomics, method of the Four Russians

Digital Object Identifier 10.4230/LIPIcs.CPM.2018.13

[1] Supported in part by NSF awards CCF-1422569 and CCF-1749864 as well as the NIH, grant R01-AI-100947 to Mihai Pop.
[2] Supported in part by the NRL, award N00173-16-2-C001 to Mihai Pop.

Acknowledgements The authors wish to thank advisors Mihai Pop and Aravind Srinivasan.

1 Introduction

Edit distance (a.k.a. Levenshtein distance) is one of the most natural and ubiquitous measures of similarity between two strings. In the most common variant, *unit cost*, it counts the minimum number of edits needed to transform one string into another. Here, we use the Levenshtein definition of *edits* which include insertions, deletions, or substitutions of a single character. However, in some cases edit operations may be assigned differring costs from a penalty matrix and additional operations (e.g. inversions or transpositions) may be considered. Computing this distance is a fundamental problem with applications in many areas such as computation biology, natural language processing, and information theory.

The most well known algorithms use dynamic programming to solve the problem in $O(m^2)$ time where m is the length of the strings. The only improvement to this has been the Four Russians algorithm [15], running in $O(m^2/\log m)$ time. While the conditional hardness results, such as [3], suggest this is unlikely to be improved further for arbitrary strings even on small alphabets [5].

The problem of comparing a string against a large set of sequences is of central importance in domains such as computational biology, information retrieval, and databases. The banded alignment variant (a.k.a. the d differences approximate string matching problem), in which we only report the distance when it is at most some parameter d is also highly relevant. It's useful in numerous settings wherein we only care about finding small distances or the maximum distance between any two strings in known to be small. In gene clustering for example, solving this problem is a key subroutine in many *greedy* clustering heuristics wherein we iteratively choose a cluster center and form a cluster by recruiting all strings which are within some small maximum distance d of the center [6]. With the development of faster and cheaper DNA sequencing technologies, metagenomic sequencing datasets can contain over 1 billion sequences [7].

Another area of research surrounding the Four Russians speedup is how to apply it in practice. While the theoretical result uses a block size of $\log m$, such a large block size is impractical due the size of the lookup table exceeding hardware constraints. For the unit cost version, [13] showed how to drastically reduce the required space, especially for large alphabets, by avoiding redundant string comparisons. We show that our approach can be combined with theirs to reduce the space (and preprocessing time) requirement even further.

1.1 Related Work

The edit distance problem is extremely well-studied and the following related work is by no means exhaustive. We focus primarily on the aspects most related to this paper: pairwise comparison, the Four Russians speedup, and one-against-many comparison. For simplicity, we describe all results in the context wherein all strings have length exactly equal to m.

The most well-known approach for computing the edit distance between a pair of strings of length m uses dynamic programming and requires $O(m^2)$. This was later improved to $O(m^2/\log m)$ in 1980 using the Four Russians speedup [15, 14] and [8] achieved $O(m^2/\log m)$ for unrestricted scoring matrices. The Four Russians speedup, originally proposed for matrix multiplication, has been adapted to many problems besides edit distance including: RNA folding [10], transitive closure of graphs [20], and matrix inversion [4]. On the negative side, [3] recently showed that no algorithm for edit distance can do better than $m^{2-\epsilon}$ time unless the Strong Exponential Time Hypothesis (SETH) is false and [5] extended this to

include strings on a binary alphabet. They accomplished this by reducing a satisfiability problem to edit distance and showing that a subquadratic algorithm for edit distance implies a subexponential algorithm for satisfiability. However, if we fix a maximum distance d and only care about reporting the exact distance when it's less than d, we call this the *banded alignment* problem. This problem has seen improvements to $O(md)$ time [9] and the current best algorithm takes only $O(m + d^2)$ time [17, 22].

One-against-many edit distance comparison involves comparing a single string to a set of n other strings. Here, we consider only the banded alignment version of the problem wherein we seek to find the distance to all strings within maximum distance d. This problem can be solved in $O(nm + nd^2)$ or $O(nmd)$ time by iteratively applying the pairwise banded alignment algorithms discussed above. Heuristic approaches may run much faster in practice by exploiting properties of the input strings such as prefix similarity and storing the set of strings in a clever data structure such as a trie or BK-tree [9]. However, little theoretical progress has been made. A popular approach to this problem in the context of spell checkers employs Levenshtein automata and/or transducers [21, 16, 12]. Assuming d is a fixed constant, these algorithms run in $O(nm)$ time. However, in practice they consider extremely small values of d (at most 3 or 4) and their runtime appears to grow exponentially in d. In the context of gene clustering in computational biology, [6] show that all pairs banded alignment can be performed in $O(n^2m)$ time under the assumption that all strings are extremely similar. They also use an extension of the Four Russians speedup to one-against-many banded alignment, but our approach to this problem requires no assumptions on the input strings.

The Four Russian speedup is well-studied in context of the regular expression membership problem where the goal is to determine if a particular string matches a given regular expression. Myers[18] showed that for a regular expression of length P and a string of length m, the exact regular expression membership problem (no mismatches are allowed) can be solved in $O(mP/\log m)$ time using the Four Russian speedup compared to the naive $O(mP)$ runtime. Wu, Manber, and Myers [23] extended this result for approximate regular expression membership problem where the goal is to check if a string is within an edit distance d from the given regular expression. They showed that approximate regular expression matching problem can be solved in $O(mP/\log_{d+2} m)$ time.

Space efficiency is also a major concern in practical applications of the Four Russians speedup since the entire lookup table must be stored in main memory. Thus, block sizes as small as $k = 4$ or 5 may be used. The classical approach for the unit cost variant uses $O(3^{2k}k|\Sigma|^{2k})$ space. Kim, Na, Park, and Sim [13] showed how to remove the dependence on the alphabet size, generating a lookup table in $O(3^{2k}(2k)!k^2)$ time and $O(3^{2k}(2k)!k)$ space. This offers a significant improvement, for example, when $|\Sigma| = 20$ for protein sequences or $|\Sigma| = 26$ for the English language.

1.2 Preliminaries

For simplicity of presentation, we assume all strings have equal length m. However, the results extend easily to the case where strings have different lengths. We assume the lookup table is any data structure that can perform lookups and insertions in $O(k)$ time for blocks which are identified by distinct keys of length $O(k)$.

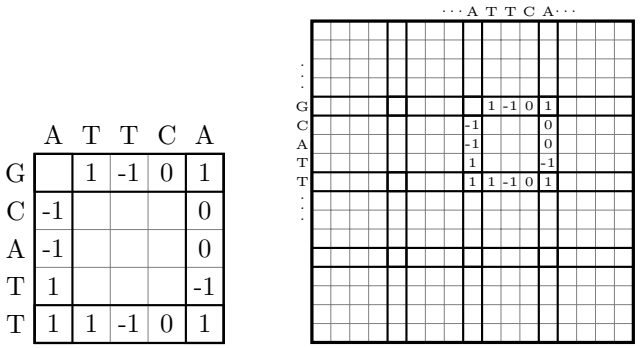

Figure 1 Example of classic Four Russians. **Left:** a single block. Notice that for any input in the upper left corner, we can sum that value with one path along the edges of the block to recover the value in the lower right corner. Note that the offset value in the lower right corner may be different for the row and column vectors overlapping at that cell. In this case, the lower right cell is one more than its left neighbor and one less than its above neighbor. **Right:** the full dynamic programming table divided into sixteen 5×5 blocks. Note that the offset values in the example block may not correspond to the optimal alignment of the two substrings shown since they depend on the global alignment between the two full length strings.

1.2.1 The classical Four Russians speedup

In the classical Four Russians speedup of edit distance computation due to [15, 14], the dynamic programming table is broken up into square *blocks* of size k-by-k as shown in the right of Fig. 1. These blocks are tiled such that they overlap by one column/row on each side (for a thorough description see [11]).

The high level idea of the Four Russians speedup is to precompute all possible solutions to a *block function* and store them in a lookup table. The block function takes as input the two substrings to be compared in that block and the first row and column of the block itself in the dynamic programming table. It outputs the last row and column of the block. We can see in Figure 1 that given the two strings and the first row and column of the table, such a function could be applied repeatedly to compute the lower right cell of the table and therefore, the edit distance.

There are several tricks that reduce the number of inputs to the block function to bound the time and space requirements of the lookup table. For example, when the edits have unit cost, the input row and column for each block can be reduced to vectors in $\{-1, 0, 1\}^k$. These *offset vectors* encode only the difference between one cell and the next (see Fig. 1) which is known to be in in $\{-1, 0, 1\}$. It has also been shown that the upper left corner does not need to be included in the offset vectors. This bounds the number of possible row and column inputs at 3^k each [15]. More generally, when edit costs are derived from a penalty matrix, the number of row/column inputs is bounded by ψ^k where ψ is the number of possible offset values and depends on the penalty matrix.

1.3 Our Contributions

We show a new way to store and query block functions. For a given pair of strings corresponding to a k-by-k block in the dynamic programming table, we store an entry in the lookup table using only $O(k^2 \lg k)$ time and $O(k^2)$ space. We show how to query this entry in $O(k)$ time. By contrast, the classical approach requires $O(\psi^{2k} k^2)$ time and $O(\psi^{2k} k)$ space, where ψ is the number of possible offset values and depends on the costs of edits, to store a

lookup entry for a pair of strings since it computes the function for all possible row/column offset vectors and $O(k)$ time per query. Thus, we improve the time and space complexity of that aspect by a factor of at least ψ^{2k}/k and remove the dependence on ψ. This result is stated in Theorem 1.

▶ **Theorem 1.** *Given two strings corresponding to a k-by-k block, we can store a lookup entry using $O(k^2 \lg k)$ time and $O(k^2)$ space such that given any values for the first row and column of the block, we can compute the last row and column of the block in $O(k)$ time.*

We demonstrate the power of our technique for block functions by designing an algorithm for the fundamental problem of one-against-many banded alignment. In particular, comparing one string of length m to n other strings of length m where we only need to report distances within a maximum distance threshold d can be performed in $O(nm + md^2 \lg d + nd^3)$ time. When d is reasonably small, this improves on the common, naive approach which requires $O(nmd)$ time to iteratively run an $O(md)$ time pairwise banded alignment algorithm. It also approaches the best theoretic result of $O(nm+nd^2)$ achieved by using the best known pairwise algorithm running in $O(m + d^2)$ time [17, 22]. We note that the author of [17], describes the $O(m + d^2)$ time algorithm as "impractical" and "primarily of theoretical interest". We are somewhat more optimistic, observing that our algorithm blends neatly with approaches such as [6] for comparing genetic sequences and as discussed in Section 4.3 can be implemented in a way that exploits the prefix similarity occurring in practice.

▶ **Theorem 2.** *Performing banded alignment with maximum distance d between a string of length m and n other strings also of length m can be done in $O(nm + md^2 \lg d + nd^3)$ time.*

We extend the classic result of [15] which computes the edit distance between two strings in $O(m^2/\log m)$ time to remove the dependence on ψ even when edits have costs derived from a penalty matrix. Here, the number of entries in the lookup table does not depend on the penalty matrix. We acknowledge that [8] also achieves the same $O(m^2/\log m)$ running time on unrestricted scoring matrices. However, there are some differences between our approach and theirs which may make one or the other more advantageous in different settings. Most notably our approach adheres more closely to the classic Four Russians speedup and uses a uniform block size which is necessary for our one-against-many algorithm. Uniform block sizes also allow our technique to be combined easily with the space-efficient approach in [13] and the gene clustering technique in [6] since both rely on splitting the dynamic programming table into uniform size blocks. In the case of [6], this is crucial to exploiting the prefix similarity among highly conserved genomic sequences. On the other hand, the blocks in [8] vary in size in a clever way to take advantage of the compressibility of the strings being compared. This yields a faster running time for pairwise comparison of strings with small entropy, $O(hn^2/\log n)$, where $h \leq 1$ is the entropy of the text.

▶ **Theorem 3.** *Given a penalty matrix for edit operations, the edit distance between two strings can be computed in $O(m^2/\log m)$ time.*

In practical applications of the Four Russians speedup wherein space efficiency is important and smaller block sizes k are used (notably $k < |\Sigma|$), [13] showed how to remove the dependence on the alphabet size for the unit cost version, generating a lookup table in $O(3^{2k}(2k)!k^2)$ time and $O(3^{2k}(2k)!k)$ space. Combining their work with our result yields an improvement to $O((2k)!k^2 \lg k)$ time and $O((2k)!k^2)$ space.

▶ **Theorem 4.** *For a block size k, a lookup table can be generated in $O((2k)!k^2 \lg k)$ time and $O((2k)!k^2)$ space such that we can find the unit cost edit distance between two strings of length m in $O(m^2/k)$ time.*

2 Storing and querying the block function

Here, we consider the crucial subroutine in our algorithms and prove Theorem 1. For a block size k, we first show how to store a lookup entry for any two strings of length k in $O(k^2 \lg k)$ time and $O(k^2)$ space. Then, we show how, given two strings of length k and the first row and column of the block, we can compute the last row and column in $O(k)$ time by querying the corresponding lookup entry. Notice that in contrast to the classical Four Russians speedup, the information we precompute and store for a block function is based only on the two strings being compared. Thus, we avoid having to store an entry for each of the 3^{2k} possible input vectors considered in [15] (For unit costs, they encode rows/columns as offset vectors in {-1, 0, 1} since the values in adjacent cells differ by at most 1, yielding 3^k possible inputs each for the row and column vectors).

2.1 Notation

We start by defining some notation, illustrated in Figure 2. Let $U = \{u_1, u_2, \ldots, u_{2k-1}\}$ be an ordered set of the cells in the first row and column of the block and let $V = \{v_1, v_2, \ldots, v_{2k-1}\}$ be an ordered set of the cells in the last row and column of the block. For both sets, the ordering starts with the upper right corner and ends in the lower left corner. Thus, both u_1 and v_1 correspond to the upper right corner, u_k corresponds to the upper left corner, v_k corresponds to the lower right corner, and both u_{2k-1} and v_{2k-1} correspond to the lower left corner.

For each pair of cells (u, v), we store the least cost $c_{u,v}$ of any path through the block from u to v. If no such path exists, we set $c_{u,v} = \infty$ and if u and v correspond to the same cell, we set $c_{u,v} = 0$. Note that $c_{u,v}$ is not necessarily based on the optimal alignment within the entire block. It corresponds to an alignment of the subset of the block with u as the upper left corner and v as the lower right. Also, recall that this block will be part of a larger dynamic programming table and the path through the block corresponding to the best global alignment may not be the same as the path corresponding to the best local alignment within the block.

We can think of this set of costs as a complete, weighted bipartite graph $G = \{U, V, U \times V\}$ with weights $c_{u,v}$ on the edges. We also use c_u and c_v to denote the values stored in the corresponding cells of the block within the dynamic programming table. When we query a block function for two strings, the c_u values (input row and column) will be given as input and our goal will be to compute the c_v values (output row and column). Thus, if we consider the values stored in the cells after the full dynamic programing table has been computed, we have that $c_v = \min_{u \in U}(c_u + c_{u,v})$.

2.2 Storing lookup entries

For every pair of substrings we wish to query eventually, our lookup table will simply store the cost c_{uv} for every edge in the graph G defined by comparing those substrings. These cost values will be stored in a $|V| \times |U|$ matrix M with a row for each $v \in V$ and a column for each $u \in U$. Cell M_{ji} will contain $c_{u_i v_j}$. We now show that computing G and storing M for any pair of substrings of length k can be done in $O(k^2 \lg k)$ time.

▶ **Lemma 5.** *Given a pair of strings of length k, we can compute all $c_{u,v}$ in $O(k^2 \lg k)$ time.*

Proof. Note that each $c_{u,v}$ can be seen as the weight of the shortest path in a grid graph of dimension $k \times k$. Thus the algorithm of [19] can be applied. That algorithm requires $O(k^2 \lg k)$ preprocessing time and can then compute each of the $O(k^2)$ $c_{u,v}$ entries in $O(\lg k)$ time. This leads to an overall running time of $O(k^2 \lg k)$. ◀

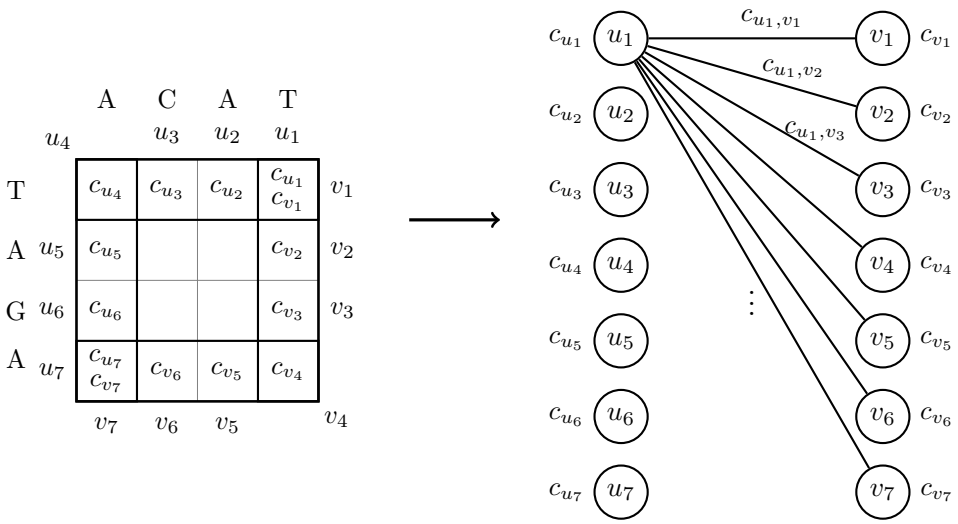

Figure 2 Illustration of how the dynamic programming table is represented as a bipartite graph of least cost paths. **Left:** The dynamic programming table for a block comparing the strings "ACAT" and "TAGA" with all u, v, c_u, and c_v labeled. **Right:** The bipartite graph representation. Note that this will be a complete, weighted bipartite graph with costs $c_{u,v}$ for all pairs in $U \times V$.

For completeness, we also state the simple fact that the space requirement for an entry is $O(k^2)$.

▶ **Lemma 6.** *Given a pair of strings of length k, storing the entry requires $O(k^2)$ space.*

Proof. The proof follows directly from the fact that we are simply storing the edges of a complete, weighted bipartite graph $G = \{U, V, U \times V\}$ with $|U| = |V| = 2k - 1$. ◀

2.3 Querying a block function

Given the two substrings and the input row and column vectors, we now show how to use our lookup entry matrix M to compute the output row and column (a.k.a all c_v for $v \in V$) in $O(k)$ time.

▶ **Lemma 7.** *Given the input row and column vectors and the $O(k) \times O(k)$ lookup entry matrix M, we can compute the output row and column in $O(k)$ time using the SMAWK algorithm [2].*

Proof. Let $\vec{w}$ be the vector of all c_u values generated from the input row and column vectors. Scaling each column of M by the corresponding cell in $\vec{w}$ gives us a new matrix M' wherein the minimum value in each row j is our desired output value $c_{v_j} = \min_{u \in U}(c_u + c_{u,v_j})$. It is known that M' is totally monotone [1, 19] and thus we can find row minima in $O(|U|) = O(k)$ time using the classic SMAWK algorithm [2]. Note that we need not explicitly generate M' since the value of any cell we wish to query can be computed from M and $\vec{w}$ as $M'_{ji} = M_{ji} + \vec{w}_i$. ◀

The proof of Theorem 1 follows from Lemmas 5, 6, and 7.

2.4 Alternatives to query a block function without SMAWK

While our algorithm for banded alignment in Section 3 uses larger block sizes than the typical pairwise Four Russians approach, in many cases, the blocks will be small enough for SMAWK to be inefficient in practice. As such, we introduce a simpler query algorithm here and briefly discuss the potential for future work to speed up the query function in practice.

This simpler query algorithm achieves a slightly worse asymptotic running time of $O(k \lg k)$ and can be described as follows. Recall that our goal is to find the minimum value of each row in the totally monotone matrix M' with $|U|$ columns and $|V|$ rows. We first find the minimum value in row $|V|/2$ and let *mincol* be the column containing that cell. We then perform the same operation recursively on two submatrices of M'. The first submatrix includes all rows up to $|V|/2$ and all columns up to (and including) *mincol*. The second includes the rows after $|V|/2$ and columns from *mincol* to $|U|$. We do not claim this simpler algorithm is a novel approach to finding row minima and include it merely to illustrate possible alternatives to SMAWK.

▶ **Lemma 8.** *The algorithm described here runs in $O(k \lg k)$ time and outputs the correct result.*

Proof. For the running time, note that each recursive call nearly partitions the columns of M' (pairs of submatrices overlap at single columns), resulting in $O(|U|) = O(k)$ time spent at each level of recursion. Since we split the rows in half at each level, there will be $O(\lg |V|) = O(\lg k)$ levels total, giving a final running time of $O(k \lg k)$.

The correctness follows directly from the properties of totally monotone matrices also utilized in the analysis of SMAWK. ◀

Looking to the future, we note that neither SMAWK nor the algorithm in this section leverage all of the specific properties of the matrix M'. For example, M' is not an arbitrary totally monotone matrix. It comes from M, a matrix which we can afford to spend k^2 time preprocessing, scaled by $\vec{w}$, a vector with the property that adjacent cells differ by at most 1 in the unit cost setting.

3 One-against-many comparison

3.1 Extending the Four Russians approach to banded alignment

For our algorithm for one against many banded alignment, we use the extension to banded alignment from [6] which simplifies both the analysis and practical implementation. The extension uses a slightly different block function and way of tiling blocks to cover the relevant diagonal "band" of the dynamic programming table. The blocks now overlap on a square of size $d + 1$ at the upper left and lower right corners. We will call these overlapping regions *overlap squares*. The block function still takes as input the two substrings to be compared. The set U contains only the first row and column of the the upper left overlap square and V contains only the first row and column of the lower right overlap square as well as the difference between the upper left corners of the two overlap squares.

Thus, we can move directly from one block to the next, storing a sum of the differences between the upper left corners. In this case, reaching the final lower right cell of the table requires an additional $O(d^2)$ operation to fill in the last overlap square, but this adds only a negligible factor to the running time.

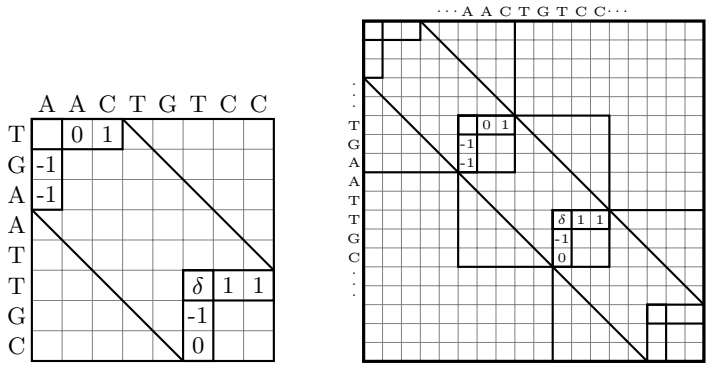

Figure 3 Example of our approach to the Four Russians speedup. **Left:** a block for maximum edit distance $d = 2$. The output δ represents the offset from the upper left corner of one block to the upper left corner of the next block. Note that we only need to consider a diagonal band of the block itself. **Right:** using these blocks to cover the diagonal band of the dynamic programming table in the context of banded alignment.

3.2 Our algorithm

We start with some notation and definitions. For a string s, let $s_{i,i+k}$ be a length k substring starting at index i of s. We define two types of block comparisons, *identities* and *differences*, based on the strings being compared. An identity comparison is between the substring $s_{i,i+k}$ and another substring that is identical to one of the substrings $s_{j,j+k}$ for $j \in \{i-d, i-d+1, \ldots, i, \ldots, i+d\}$. All other comparisons are difference comparisons. In other words, identity comparisons between two strings will come from long common subsequences between the two strings. Difference comparisons will come from the locations where an edit occurs. Note that we can stop comparing two strings once we've encountered more than d differences among their prefixes. Let S be a set of strings and let p be the single string we wish to compare to all strings in S.

The algorithm can be summarized as follows. We first compute and store lookup entries for all possible identity comparisons for each block in p. We then perform pairwise comparisons between p and each string in S. A pairwise comparison is computed as follows. For each block, we first query the lookup table using the corresponding substrings. If we find an entry (similarity comparison), we query it as described in Section 2. Otherwise (difference comparison), we perform standard banded alignment on the two strings with the first row and column of the table initialized to the values of the input row and column of the block. If at any time during a pairwise comparison the distance accumulated exceeds d, then we immediately halt and move on to the next pair.

We divide the analysis into three parts: the time to compute and store the lookup table, the time to query the lookup table during pairwise comparison, and the time to compute the block function for difference comparisons.

▶ **Lemma 9.** *The time to compute and store the lookup table for all block identity comparisons in a single string p of length m and max distance d is $O(md^2 \lg d)$.*

Proof. Let the block size $k = 2d$. Then p will be divided into $m/d - 1$ blocks. For any given block, let $p_{i,i+k}$ be the substring of p corresponding to that block. Then, for every $j \in \{i-d, i-d+1, \ldots, i, \ldots, i+d\}$, we need to store the comparison between $p_{i,i+k}$ and $p_{j,j+k}$. We need not compare $p_{i,i+k}$ to any substrings outside this range since that would imply an alignment of distance greater than d. Thus, for each block we need to store lookups

for at most $2d + 1 = O(d)$ different identity comparisons. Computing the lookup entry for each comparison takes $O(k^2 \lg k) = O(d^2 \lg d)$ time by Theorem 1. Putting it all together, we have $O(m/d \cdot d \cdot d^2 \lg d) = O(md^2 \lg d)$. ◀

▶ **Lemma 10.** *Excluding the time to compute block functions for difference comparisons, the time to compare a string p of length m to n other strings using the precomputed lookup table is $O(nm)$.*

Proof. Each pairwise comparison involves computing $m/d - 1$ block functions. If a block corresponds to an identity comparison querying the block function takes $O(k) = O(d)$ time by Theorem 1. Otherwise, if it's a difference comparison block, the only time will come from checking the lookup table which we've assumed takes $O(d)$ time. It follows that the running time for each pairwise comparison is $O(m)$ and comparing p to all n strings requires $O(nm)$ time. ◀

▶ **Lemma 11.** *The time needed to compute block functions for difference comparisons between p and all n other strings is $O(nd^3)$.*

Proof. Notice that each edit is present in at most two overlapping blocks. It follows that for a given pair of strings, the number of block queries corresponding to differences can be at most $2(d + 1) = O(d)$ since we will halt a comparison if the distance ever reaches $d + 1$ or more. Thus, the running time to compute the full dynamic programming for difference blocks for all n pairwise comparisons is $O(n \cdot d \cdot d^2) = O(nd^3)$. ◀

The proof of Theorem 2 follows from combining Lemmas 9, 10, and 11.

4 Extensions and applications

In this section, we briefly show how the results of Section 2 can be applied to other settings in which the Four Russians speedup is used for computing string edit distance.

4.1 Comparing two arbitrary strings with a penalty matrix

As with the classical Four Russians, when the alphabet size is constant, we can choose the block length k to be an appropriate logarithmic function of the string length m such that the lookup table can be computed efficiently. For an alphabet Σ, there are $|\Sigma|^{2k}$ pairs of string of length k. By Theorem 1, each pair requires $O(k^2 \lg k)$ time to compute the lookup entry regardless of the costs of the edits. Thus, the preprocessing for $k = (\log_{|\Sigma|} m)/2$ takes $O(m \log^2 m \log \log m)$ time. Since the total number of blocks in the dynamic programming table is $O(m^2/k^2)$ and computing each block function from the lookup table takes $O(k)$ time by Theorem 1, the running time to compute the distance using the lookup table is $O(m^2/\log m)$. This completes the proof of Theorem 3.

4.2 Improved space-efficiency

The approach in Section 2 can be combined with the work of [13] to achieve the improved time and space bound in Theorem 4 for computing the lookup table. Notice that Theorem 1 gives a time and space bound for each pair of substrings for which we need to compute a block function. Specifically, each pair of strings contributes $O(k^2 \lg k)$ time and $O(k^2)$ space. As a complement, [13] showed how to encode strings in such a way that we reduce the number of redundant string comparisons. There, the number of strings compared is reduced to $O((2k)!)$. Theorem 4 follows from these simple observations.

4.3 Exploiting prefix similarity in one-against-many comparison

Since the one-against-many banded alignment algorithm in Section 3 uses the same extension to banded alignment as [6], it can be combined with other techniques from that paper. In particular, they divide all of the strings in the database S into blocks and store the blocks in a trie-like data structure. This allows them to exploit prefix similarity of the strings of S and further improve the running time in practice. Additionally, that uses *lazy computation*, the technique of computing and storing the lookup table on-the-fly rather than precomputing it to heuristically avoid comparing substrings which don't actually appear in the dataset. In the context of Theorem 2, that could potentially reduce the md^3 factor.

5 Conclusion and future directions

In this paper, we provided an approach to storing and querying block functions in the Four Russians speedup for edit distance computation using less time and space than the original method. We demonstrated how this approach can lead to an algorithm for the one-against-many banded alignment problem. Finally, we showed how our approach can easily be combined with prior work to gain additional improvements such as space-efficiency.

The problems of comparing two similar strings and one-against-many comparison of highly similar strings have applications in variety of domains. For example, searching a query sequence against the database of multiple sequence within a certain similarity threshold is one of the basic tasks in designing database management systems. In the case of document plagiarism detection, the task is to compare two documents which are assumed to be highly similar to each other. In the case of computational biology, sequence similarity detection is a ubiquitous task in most analysis. Although there have been efficient algorithms proposed in literature, they are not very easy or practical to implement on a routine basis. Our algorithm may bridge this gap and be easier to implement while yielding similar theoretical bounds.

There are many questions and potential future directions following this work. One natural question is whether the techniques in this paper can be applied to other problems yielding a Four Russians speedup. In many cases, such as boolean matrix multiplication, the answer is no. However, problems more closely related to edit distance may yield some improvement. Regarding the specific problems in this paper, the $O(nd^3)$ term in the one-against-many result can likely be improved to $O(nd^2)$ to match [17] and doing so using practical techniques would be a nice addition to this work. Similarly, improving the constant factors in the query by using a more specialized algorithm than SMAWK (even an asymptotically worse algorithm) could enhance the practical applications of our approach. On the hardness side, which of these results are tight?

References

1 Aggarwal and J. Park. Notes on searching in multidimensional monotone arrays. In *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science*, pages 497–512, Oct 1988.

2 Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter Shor, and Robert Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(1):195–208, Nov 1987.

3 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *Proceedings of the Forty-seventh Annual ACM Symposium on Theory of Computing*, STOC '15, pages 51–58, New York, NY, USA, 2015. ACM.

4 Gregory Bard. Matrix inversion (or lup-factorization) via the method of four russians, in $\theta(n^3/logn)$ time. *LMS J. Comput. Math*, 1:14, 2008.

5 Karl Bringmann and Marvin Kunnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *Proceedings of the 2015 IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS)*, FOCS '15, pages 79–97, Washington, DC, USA, 2015. IEEE Computer Society.

6 Brian Brubach, Jay Ghurye, Mihai Pop, and Aravind Srinivasan. Better greedy sequence clustering with fast banded alignment. In Russell Schwartz and Knut Reinert, editors, *17th International Workshop on Algorithms in Bioinformatics (WABI 2017)*, volume 88 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:13, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

7 J Gregory Caporaso, Christian L Lauber, William A Walters, Donna Berg-Lyons, James Huntley, Noah Fierer, Sarah M Owens, Jason Betley, Louise Fraser, Markus Bauer, et al. Ultra-high-throughput microbial community analysis on the illumina hiseq and miseq platforms. *The ISME journal*, 6(8):1621–1624, 2012.

8 Maxime Crochemore, Gad M. Landau, and Michal Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM J. Comput.*, 32(6):1654–1673, 2003.

9 J. W. Fickett. Fast optimal alignment. *Nucleic Acids Res.*, 12(1 Pt 1):175–179, Jan 1984.

10 Yelena Frid and Dan Gusfield. A simple, practical and complete o-time algorithm for rna folding using the four-russians speedup. *Algorithms for Molecular Biology*, 5(1):13, 2010.

11 Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press, 1997.

12 Ahmed Hassan, Sara Noeman, and Hany Hassan. Language independent text correction using finite state automata. In *In Proceedings of the Third International Joint Conference on Natural Language Processing*, pages 913–918, 2008.

13 Youngho Kim, Joong Chae Na, Heejin Park, and Jeong Seop Sim. A space-efficient alphabet-independent four-russians' lookup table and a multithreaded four-russians' edit distance algorithm. *Theor. Comput. Sci.*, 656:173–179, 2016. doi:10.1016/j.tcs.2016.04.028.

14 William J. Masek and Michael S. Paterson. How to compute string-edit distances quickly. In *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison*, pages 337–349. Addison-Wesley Publ. Co., Mass., 1983.

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

16 Stoyan Mihov and Klaus U. Schulz. Fast approximate search in large dictionaries. *Comput. Linguist.*, 30(4):451–477, 2004.

17 Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, Nov 1986.

18 Gene Myers. A four russians algorithm for regular expression pattern matching. *Journal of the ACM (JACM)*, 39(2):432–448, 1992.

19 Jeanette P. Schmidt. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM J. Comput.*, 27(4):972–992, 1998.

20 Claus-Peter Schnorr. An algorithm for transitive closure with linear expected time. *SIAM Journal on Computing*, 7(2):127–133, 1978.

21 Klaus Schulz and Stoyan Mihov. Fast string correction with levenshtein-automata. *International Journal of Document Analysis and Recognition*, 5:67–85, 2002.

22 Esko Ukkonen. Algorithms for approximate string matching. *Inf. Control*, 64(1-3):100–118, 1985.

23 Sun Wu, Udi Manber, and Eugene Myers. A subquadratic algorithm for approximate regular expression matching. *Journal of algorithms*, 19(3):346–360, 1995.