Algorithms for the description of molecular sequences

Vis, J.K.

Chapter 3

An Efficient Algorithm for the Extraction of HGVS Variant Descriptions from Sequences

Unambiguous sequence variant descriptions are important in reporting the outcome of clinical diagnostic DNA tests. The standard nomenclature of the Human Genome Variation Society (HGVS) describes the observed variant sequence relative to a given reference sequence. We propose an efficient algorithm for the extraction of HGVS descriptions from two sequences with three main requirements in mind: minimizing the length of the resulting descriptions, minimizing the computation time, and keeping the unambiguous descriptions biologically meaningful.

Our algorithm is able to compute the HGVS descriptions of complete chromosomes or other large DNA strings in a reasonable amount of computation time and its resulting descriptions are relatively small. Additional applications include updating of gene variant database contents and reference sequence liftovers.

The algorithm is accessible as an experimental service in the Mutalyzer program suite (`https://mutalyzer.nl`). The C++ source code and Python interface are accessible at: `https://github.com/mutalyzer/description-extractor`.

3.1 Introduction

The Human Genome Variation Society publishes nomenclature guidelines [den Dunnen et al., 2000] for unambiguous sequence variant descriptions in clinical reports, the literature and genetic databases. The Mutalyzer program suite [Wildeman et al., 2008] has been built to automatically check and correct these variant descriptions. As many complex variants are supported, the corresponding descriptions are not always straightforward to construct, justifying the need for the automatic extraction of HGVS descriptions by comparison of the sequence observed in an individual to the reference sequence specified in guidelines and databases. Here we approach this from an informatics perspective as a string comparison problem.

Consider two DNA strings: R, the reference string and S, the sample or observed string:

$$R = \texttt{ATGAT GATCAGATACAGTGTGATACAGGTAGTTAG ACAA}$$
$$S = \texttt{ATGATTTGATCAGATACA TGTGATACCGGTAGTTAGGACAA}$$

The string S can be rewritten in terms of string R by using the HGVS description:

`g.[5_6insTT;17del;26A>C;35dup]`

The *string-to-string correction problem* calculates the distance between two strings as measured by the minimum cost of a sequence of *edit operations* needed to transform one string into the other. The traditionally allowed edit operations [Wagner and Fischer, 1974] are exchanging one symbol of a string for another: a *substitution* indicated using > between symbols (`26A>C`), deleting a single symbol from a string: a *deletion* indicated using abbreviation `del` (`17del`), and inserting one symbol: an *insertion* using abbreviation `ins` (`5_6insTT`). There is a specific case: insertion of previous symbol(s) is described with HGVS term *duplication* using abbreviation `dup` (`35dup`). The string-to-string correction problem has been extended on in numerous occasions [Wagner and Lowrance, 1975, Tichy, 1984] usually allowing more powerful edit operations. Here, we solve another extension of this problem by defining additional edit operators especially suited to the HGVS nomenclature.

Formally, our extension can be defined as follows. Given two strings R and S over the finite alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, and a set of edit operators with their corresponding (non-negative) weights, calculate a sequence of edit operations that transforms a reference string R into a sample string S with a minimum cost with regard to the weights of the operations given in Table 3.1. The weights in Table 3.1 are based on the textual length of the HGVS nomenclature. Note that the length of the description of the position is dependent on the position, i.e., towards the end it takes more symbols to describe the position, therefore we will parameterize all weights making them independent of the positions.

Traditionally, the edit operations are defined on single symbols. To provide a more intuitive way of describing variants, we extend these operations in a natural way allowing use of substrings rather than individual symbols. Note that, in contrast to the insertion operator, the deletion operator on multiple symbols is not dependent on the length of the deleted substring, thereby creating an asymmetry between insertion and deletion.

In addition to the traditional edit operators we define two additional operators: *inversion* (HGVS abbreviation: `inv`) matches the *reverse complement* of the string and *transpositions*.

3.1.1 Transpositions

Here we define transpositions to be substrings which are copies of substrings found either elsewhere in the matched string or elsewhere in the same string. As we are interested in calculating concise descriptions, we will only consider insertions to be candidates for transpositions. This will produce favorable results especially in the case of long insertions that can be described as long transpositions as their weights are independent of the length of the inserted substrings. Furthermore, we allow some variants within these transpositions yielding *composite* transpositions, e.g.:

```
g.[5_6ins[GG;17_45;inv46_78]]
```

This composite transposition consists of three parts: a regular insertion of `GG`, a transposition of a substring of the reference sequence from position 17 to 45

Table 3.1: Edit operators for HGVS descriptions with their corresponding weights.

Operator	HGVS Description	Weight
Deletion (single)	p`del`	$x + 3$
Deletion (multiple)	$p_{\text{start}}\_p_{\text{end}}$`del`	$2x + 4$
Deletion/insertion (single)	p`delins`w	$x + 6 + \lvert w\rvert^{\dagger}$
Deletion/insertion (multiple)	$p_{\text{start}}\_p_{\text{end}}$`delins`$w$	$2x + 7 + \lvert w\rvert^{\ddagger}$
Insertion	$p_{\text{start}}\_p_{\text{end}}$`ins`$w$	$2x + 4 + \lvert w\rvert^{\ddagger}$
Inversion	$p_{\text{start}}\_p_{\text{end}}$`inv`	$2x + 4$
Substitution	pc_1`>`c_2	$x + 3$
Transposition	$p_{\text{start}}\_p_{\text{end}}$`ins`$[p_{\text{start}}\_p_{\text{end}}]$	$4x + 4$
Inverse transposition	$p_{\text{start}}\_p_{\text{end}}$`ins`$[p_{\text{start}}\_p_{\text{end}}$`inv`$]$	$4x + 7$

where x is the weight of a position description independent of the actual position.

† $w \in \Sigma^*$, with $|w| > 1$

‡ $w \in \Sigma^*$, with $|w| > 0$

followed by a transposition found on the reverse complement of the reference sequence, i.e., an inverse transposition. Note that the alternative would require the insertion of 62 nucleotides.

The remainder of this chapter is organized as follows. In Section 3.2 we introduce an algorithm to efficiently compute the HGVS description between two strings. Section 3.3 describes the experiments, followed by a discussion of the results in Section 3.4 and the conclusions in Section 3.5.

3.2 Methods

In order to automatically construct HGVS descriptions we propose an extraction algorithm. The three main requirements considered for this algorithm are:

1. The length of the descriptions — we try to minimize these;

2. The computational speed — in order to be practically useful we consider a maximum total computation time of 1 hour for chromosome 1 of the human genome on a desktop PC (3.4 GHz and 16 GB RAM). Although this specific timing criterion is arbitrary it serves as a indication for a responsive desktop environment;

3. The (biological) meaning of the descriptions — given that this algorithm is developed for genetic data, we want the descriptions to be as close as possible to the intuition of the people using them.

3.2.1 Extraction algorithm

A trivial way to describe the sample string in terms of the edit operations from the reference string, is to give the substitution of the whole reference string with the sample string by means of the deletion/insertion operator. This gives us an upper bound on the length of the description. We can stop recursively cutting the strings at the moment when the resulting description exceeds the trivial description or when we can decide that every possible description from this point on will result in a longer description.

The underlying idea of the extraction algorithm is to divide the string to be described into a sequence of unaltered regions and altered regions. The altered regions are then described according to the HGVS nomenclature. In order to minimize the length of the resulting descriptions, we apply a *greedy* approach by choosing the longest possible unaltered regions. Note that this is a heuristic which implies that it might be possible to find a more concise description by choosing a smaller unaltered region.

The algorithm is formulated recursively: given two strings R and S find the longest string that is a substring of R as well as S. Remove this string from

the problem, and continue recursively with both prefixes R_{pre} and S_{pre} and both suffixes R_{suf} and S_{suf}. The recursion ends when either of the two strings is empty or no common substring could be found, see Figure 3.1. In case of an empty reference string and a non-empty sample string, the corresponding variant is an insertion. When the sample string is empty and the reference string is not, the corresponding variant is a deletion. If no common substring could be found, depending on the length of both strings we deal with a substitution in case of a single nucleotide or a larger deletion/insertion.

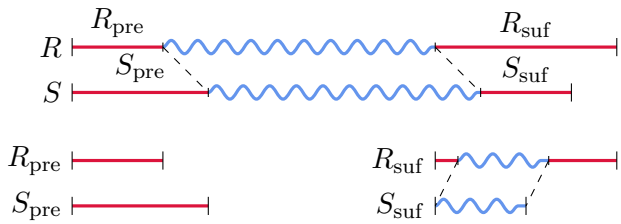

Figure 3.1: Graphical representation of the extraction algorithm with reference string R and sample string S, with the recursion showing a common substring in the suffixes (suf), but not in the prefixes (pre). The wavy lines denote the LCS during that iteration.

3.2.2 Finding the Longest Common Substring

In this section we explain the traditional approach for finding the longest common substring between two strings as an introduction to the more efficient version we present in Section 3.2.3.

The problem of finding the *longest common substring(s)* (LCS) between two (or more) strings is a well studied problem [Gusfield, 1997]. Traditionally, a dynamic programming approach for finding the LCS is used. Based on the recurrence relation (3.1), a table M is built containing at each position (i, j) the length of the longest common suffix between both prefixes.

Equation (3.2) is used to find the length of the longest common substring.

Together with the position (i, j) we can easily find the actual string.

$$M(S_{1..i}, R_{1..j}) = \begin{cases} M(S_{1..i-1}, R_{1..j-1}) + 1 & \text{if } S_i = R_j \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

$$\text{LCS}(S, R) = \max_{1 \le i \le |S|, 1 \le j \le |R|} M(S_{1..i}, R_{1..j}) \tag{3.2}$$

In order to illustrate the mechanisms of finding the LCS, we will present an example. Let $R =$ `AACACTTA`, and $S =$ `ACTAACACTT`. We construct M according to the recurrence relation (3.1) as shown in Table 3.2. We fill M from top to bottom, and from left to right. If the symbols on position (i, j) match, we look at position $(i - 1, j - 1)$ and extend the matched suffix. For instance, position $(3, 6)$ has 3, because position $(2, 5)$ has 2 and `T` matches `T`.

Table 3.2: Dynamic programming approach for finding the longest common substring. Here, the LCS is `AACACTT`, with length 7.

M	A	A	C	A	C	T	T	A
A	1	1	0	1	0	0	0	1
C	0	0	2	0	2	0	0	0
T	0	0	0	0	0	3	1	0
A	1	1	0	1	0	0	0	2
A	1	2	0	1	0	0	0	1
C	0	0	3	0	2	0	0	0
A	1	1	0	4	0	0	0	1
C	0	0	2	0	5	0	0	0
T	0	0	0	0	0	6	1	0
T	0	0	0	0	0	1	7	0

The number of rows in M corresponds to the length of S, while the number of columns corresponds to the length of R. By filling M we deduce the runtime and memory complexity of this algorithm: $O(|R| \cdot |S|)$. Usually $|R| \approx |S|$, giving a quadratic time behavior for this algorithm. We can easily reduce the required

amount of memory by storing only the current and previous row of table M, which gives us a memory bound of $O(\min(|R|, |S|))$.

Although this dynamic programming approach seems similar to the Smith-Waterman algorithm [Smith and Waterman, 1981] for local alignment, it is significantly different. In this phase of the extraction algorithm we focus only on finding the LCS. This permits us to use more powerful and non-local edit operators, i.e., inversions and transpositions which are not possible within the local alignment algorithm.

3.2.3 Finding the LCS more efficiently

In theory an instance of *generalized suffix trees* could be exploited giving us a linear bound on runtime. However, the implementations are impractical both in memory requirements as well as having large constants in the linear runtime. Instead, we will present an alternative LCS retrieval method based on the traditional dynamic programming approach in Section 3.2.2.

Although for application to chromosomal sequences we have to calculate the LCS of two large strings, we expect that these strings within one species would be very similar to each other. We expect the LCS of those strings to be very large compared to the length of the strings. Using this knowledge we propose to encode the strings into a higher alphabet. We split both string into substrings of length k, called k-mers, one string into non-overlapping k-mers and the other into overlapping k-mers. Using a k-mer representation is a well-known optimization for sequence alignment [Compeau et al., 2011].

The size of the table required is greatly reduced by the use of non-overlapping k-mers. It is, however, impossible to split both strings into non-overlapping k-mers, because it would impose a constraint on the starting position of the LCS to be found: only a LCS starting on a kth position can be found. By splitting one string into overlapping k-mers we remove this constraint while still reducing the table size.

In Table 3.3 we show the tables M_2 and M_3 constructed for the same example as given in Table 3.2 by using a modified version of the recurrence

relation (3.1):

$$M_k(S_{1..i}, R_{1..j}) = \begin{cases} M_k(S_{1..i-k}, R_{1..j-1}) + 1 & \text{if } S_i = R_j \\ 0 & \text{otherwise} \end{cases}, \tag{3.3}$$

where S_i and R_j are k-mers.

In order to calculate the value of position $(7, 2)$, we have to look at the position $(7 - 3, 2 - 1)$ to extend the k-mers matched so far. Equation (3.2), adapted in the natural way, can be used to extract the LCS based on k-mers. In this case it yields the LCS `AACACT` with length 6. Consequently, we have to extend the found LCS, possibly at both ends to find the actual LCS of length 7 as a post-processing step. In general, the actual LCS can be extended $k-1$ symbols to the left, and $k-1$ symbols to the right. This implies that for $k > 1$ the LCS can be found at a position in the M_k table with a sub-optimal value. To be precise: one less than the maximum value found using Equation (3.2). All these positions have to be considered for the LCS as well.

Table 3.3: Dynamic programming approach with overlapping and non-overlapping 2-mers and 3-mers.

M_2	AA	CA	CT	TA
AC	0	0	0	0
CT	0	0	1	0
TA	0	0	0	1
AA	1	0	0	0
AC	0	0	0	0
CA	0	2	0	0
AC	0	0	0	0
CT	0	0	3	0
TT	0	0	0	0

M_3	AAC	ACT
ACT	0	1
CTA	0	0
TAA	0	0
AAC	1	0
ACA	0	0
CAC	0	0
ACT	0	2
CTT	0	0

In comparison to the original table M, the M_k table is much smaller: $(|S| - k + 1) \times \lfloor |R|/k \rfloor$. If we can swap the roles of R and S freely, it is

advantageous to choose R to be the longest string. Again, we only have to store a part of this table: the current row and the k previous ones. All of these rows contain fewer elements than those in table M. The memory constraints remain approximately the same as for the original algorithm. The runtime, however, is greatly reduced for large values of k.

3.2.4 Choosing the size of the of k-mers

If we compare Table 3.2 and Table 3.3, it appears that we cannot find all arbitrary common substrings of R and S. For instance, the substring `ACT` starting in R at position 9 and in S at position 7 is not present in Table 3.3 due to an unfortunate misalignment in the non-overlapping k-mers. Moreover, all common substrings with a length less than k are not present at all. To be certain to find a common substring of length ℓ, k has to be at most $\lceil \ell/2 \rceil$. Therefore, we can consider k to be a guess for the expected length of the LCS between R and S.

To achieve the best performance of this algorithm, the initial value for k has to be chosen carefully. On one hand we like k to be as large as possible to reduce the runtime as much as possible. On the other hand k has to be small enough compared to the LCS between the two strings. In general we will not know the exact length of the LCS.

In case the algorithm returns no result, we lack the guarantee of the traditional approach, that there is no common substring between both strings. If k is chosen too large, the whole table will contain zeroes or ones and the algorithm fails to produce a result. To find the LCS, we have to reduce the value of k and run the algorithm again until a result is returned or the value of k falls below a certain threshold. In general, this threshold can be 1 which guarantees that there exists no common substring between both strings. However, this is impractical for large strings. Usually, the threshold can be set at the expected length of the LCS between two random strings over alphabet Σ, trivially bound by $2 \log_{|\Sigma|} n$ for strings of length n [Abbasi, 1997].

3.2.5 Adapting the extraction algorithm for inversions, transpositions and inverse transpositions

So far the extraction algorithm in Figure 3.1 handles only variants of the deletion, insertion, and substitution operations. To add support for the inversion operator, we have to run the LCS algorithm twice. First, the sample string is matched to the original reference string. Second, it is matched against the reverse complement of the reference string (in every instance of the recursion). If the LCS is found on the reverse complement, the algorithm picks this LCS and removes it from the solution. In the exceptional case of a tie between the length of a regular LCS and the length of a reverse complement LCS, the algorithm prefers the regular one, because of the higher weight associated with a reverse complement match. In the next step of the recursion a new decision will be made on whether to use the original or a reverse complement LCS independent of the current choice. Note that the complexity of the algorithm does not change essentially as we do twice the amount of work.

In order to find useful transpositions, we consider all insertions of a certain length. In practice, insertions of two base pairs will usually not be considered to be transpositions as all occurrences of two base pairs will be present elsewhere. With increasing length of the insertions the probability that these exact sequences are found elsewhere diminishes quickly. Therefore, if we are able to locate these sequences elsewhere, we can be confident that they are indeed transpositions. Instead of looking for the exact sequences, we use a modified recursive instance of the extraction algorithm to find transpositions with small variations. The main difference between the regular extraction algorithm and the modified algorithm proposed here is that deletions within a transposition are not meaningful, i.e., we just describe the actual insertions either as regions to be found elsewhere in the string or as regular insertions. Likewise, inverse transpositions are found by matching against the reverse complement string.

3.3 Experiments

We performed computer experiments to demonstrate the performance of our proposed algorithm both in terms of speed and the quality of its output. In the first experiment we will focus on the performance of the extraction algorithm on large DNA strings, i.e., whole human chromosomes. The second experiment aims to minimize the resulting descriptions in a real-life case study. The final experiment shows the biological quality of the resulting descriptions.

In all experiments we used a fixed initialization and reduction scheme for k when the algorithm fails to return a solution, as explained in Section 3.2.4. We initialize k to $|R|/4$; in case of no solution we reduce $k \leftarrow k/3$. This seems to be a good balance for maximizing k and miniziming the amount of re-runs for the LCS_k algorithm. On average 1 to 2 re-runs are sufficient.

For the transposition cut-off discussed in Section 3.2.5, we specify a threshold of 10% of the length of the inserted string. Any matched regions smaller than this length are considered to be uninteresting as transpositions and are described as regular deletions/insertions. Modifying this cut-off will greatly affect the runtime of the algorithm. Again, for our experiments, this cut-off strikes a good balance between runtime, description length, and biologically interesting patterns.

3.3.1 Performance on large DNA strings

To demonstrate the usefulness and speed of our proposed algorithm we used all chromosomal RefSeq sequences from human genome build NCBI36 (GCF_000001405.12), GRCh37 (GCF_000001405.13) and
GRCh38 (GCA_000001405.15) and performed three extraction experiments:

1. NCBI36 (sample) vs. GRCh37 (reference);
2. NCBI36 (sample) vs. GRCh38 (reference);
3. GRCh37 (sample) vs. GRCh38 (reference).

We extracted the HGVS descriptions of the differences of the respective sample sequences relative to the respective reference sequences per chromosome with

a total computation time of about 40 hours, see Figure 3.2.

As a preprocessing step we replaced all sequential occurrences of `N` with a single `N`. Large sequences of `N` are commonly found at the starts and ends of chromosomes (telomeres) and at their centers (centromeres). We particularly wanted to avoid transposition matching of sequences of `N` as they yield no information.

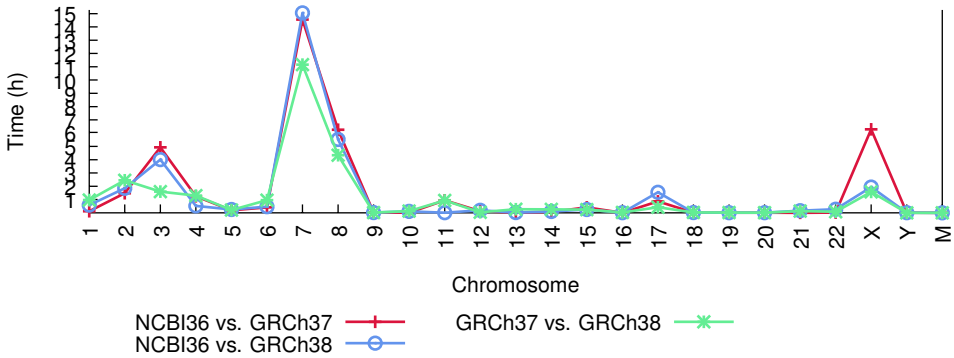

Figure 3.2: Performance of the extraction algorithm per chromosome on a desktop PC (3.4 GHz and 16 GB RAM).

In Figure 3.3 we observe that the maximum description length for any chromosome is about 1 MB. The descriptions can be calculated in at most 1 hour for most chromosomes except for chromosomes 5, 7, 8, and X. Here, we observe a large number of relatively small insertions which are just large enough to be considered for the transposition extraction. This process is very expensive in terms of calculation time, as a whole chromosome needs to be matched against a small string, eliminating the speed-up gained when using a large k.

There seems to be no obvious relation between the calculation time and the resulting description length; a longer calculation time does not always result in a more concise description. Again, small insertions seem to contribute most to this phenomenon. Often the expensive transposition extraction process is started, but the resulting description in terms of transpositions is, in the end, longer than the trivial description. This results in an increase in computation

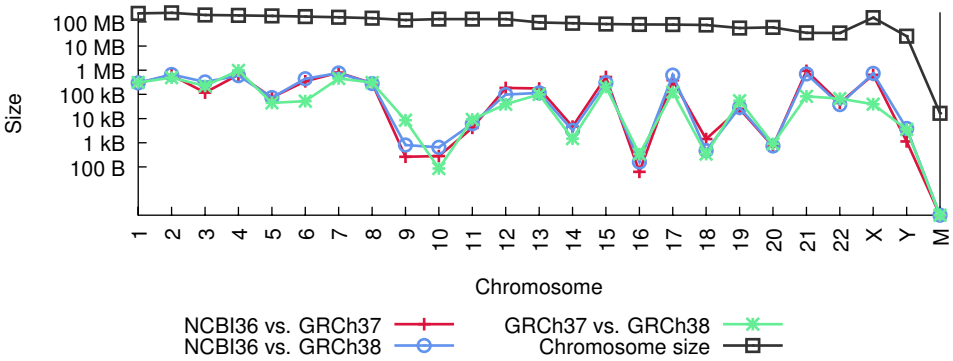

Figure 3.3: HGVS Description lengths of the extraction algorithm per chromosome.

time as well as in the description length.

We should mention that the case of description of one genome build relative to an other is a very artificial example. In each new version of the human genome information is added, i.e., gaps representing unsequenced regions have been replaced with regions that had not been sequenced before and assembly errors have been corrected. This results in multiple large insertions. Also, these descriptions yield no useful biological knowledge. However, we can give an estimate of the amount of information added with every new build.

Parallelization of the algorithm is trivially possible by using threads for each recursive call. The task of efficiently partioning the work over a fixed number of threads is non-trivial. The current recursive definition implies that many calls will terminate relatively quickly. The overhead of starting threads in these circumstances should be considered carefully. Our algorithm in its current form is ill-suited for distribution over multiple machines. Apart from the design of our algorithm we also have to provide an efficient way of accessing global data as the algorithm uses non-local operators. Moreover, one of our primary design criteria is the ability of efficiently generating variant descriptions within a desktop environment.

In Figure 3.4 we present the distribution of the HGVS operators from the description of Chromosome 2 (NCBI36 vs. GRCh37). This distribution is

representative for the distribution of the operators for most of the chromosomes in this experiment (note that Chromosome M has no variants).

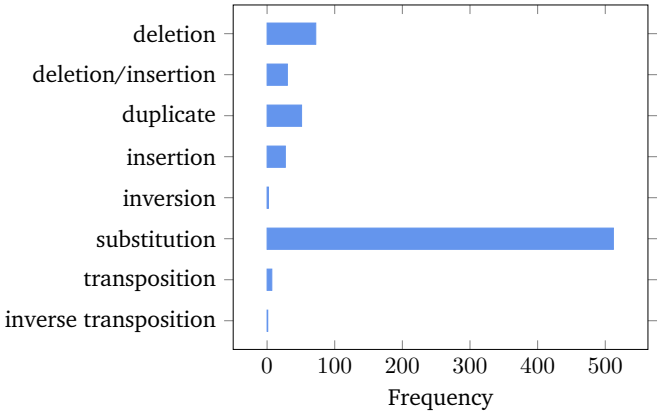

Figure 3.4: The distribution of HGVS operators for Chromosome 2 (NCBI36 vs. GRCh37).

The distribution in Figure 3.4 shows that almost 74% of all variants are substitutions. The insertion operators contribute most to the length of the descriptions (data not shown). The individual variants of composite transpositions are partitioned into their respective operators, e.g., the transposition `12_13ins[17_51;GC;50_99;CTCTG]` contains two transpositions, and two insertions.

3.3.2 Automated description extraction using sequences from a gene database

For this experiment we used the IMGT/HLA Database [Robinson et al., 2014] from which we extracted the sequence of $3{,}588$ HLA-B variant genes. For most of these sequences allele descriptions in HGVS-like format are provided using coding DNA numbering with RefSeq Gene reference sequence NG_023187.1 (see for example: `https://ebi.ac.uk/cgi-bin/ipd/imgt/hla/get_allele_hgvs.cgi?B*73:01`). As all sequences are between 500 and $1{,}000$ bp long, calculation time is not an issue and is in fact dominated by disk access times. For

this experiment it took about 1 hour to automatically generate all HGVS descriptions.

The original descriptions contain predominantly substitutions. For substitutions that are very close together it is often more concise to describe them using a deletion/insertion operator. The HGVS nomenclature forbids the occurrence of two adjacent substitutions. However, these are commonly found in the original descriptions. The original descriptions never have deletions at the beginning or end of the sequence while these variants are all captured by the automatic extraction process. Because of the missing deletions the resulting description length of the automated extracted description can be longer than its corresponding original one. If we disregard these deletions, the automatically derived description is either the same or of (much) smaller length. Finally, we have observed some irregularities in the original descriptions with regard to the HGVS nomenclature, e.g., `[960_961dupT;]` which contains two mistakes and an inaccuracy: (1) only one nucleotide is duplicated, so there is no need for a range of positions, (2) the nucleotide letter(s) do not have to be present for duplicates, and (3) as there is no variant following the duplicate no separating symbol (;) is needed. We have communicated the results of our description extraction with the curators of the IMGT/HLA Database.

3.3.3 Replacing reference sequences for gene variant databases

Gene variant database curators need to update gene-centered reference sequences (predominantly RefSeq Gene files) when new (improved) versions are generated following the release of a new genome build. The current algorithm can help: variant sequences can be generated from the original descriptions using the Mutalyzer Name Checker. These sequences can be compared with the new RefSeq Gene sequence leading to updated HGVS variant descriptions. These descriptions can replace the old ones in the database.

3.4 Discussion

In this section we introduce two additional qualities of automatically generated HGVS descriptions especially when used in genomic databases.

3.4.1 Compression

HGVS descriptions can be an attractive alternative for compressing DNA sequences, especially in a database containing sequences with high similarity that can be described using a single reference sequence. Often the standard reference genome can be used. All instances in the database can be stored by using their HGVS descriptions instead of their sequences and (optionally) one copy of the reference sequence. The difference between the original chromosome size and its corresponding description length is large: up to a million times smaller, see Figure 3.2. To give an impression of the overall compression efficiency, storing the complete human genome requires approximately 3 GB, while storing only the descriptions will take approximately 6 MB per instance giving a reduction 466 times. [Brandon et al., 2009] introduced a similar way of compressing genomic sequence data. They achieved similar results in terms of the compresssion factor as our method. As they focus on developing a compression algorithm, they used a binary encoding scheme for frequent partial variants. In this respect their algorithm differs from ours: we focus on the actual variants and we describe the complete variants in a human readable form.

Traditional compression techniques such as gzip will reduce the size of the human genome to approximately 800 MB. Apart from a much better compression rate, the HGVS format is human readable. Furthermore, many useful queries, e.g., determining the presence of a substitution, can be performed directly on the HGVS descriptions without the need for decompression.

3.4.2 Transitivity

In principle we could also transform descriptions generated using one specific reference string to descriptions versus other reference strings. This is a

potentially powerful operation for large genomic databases. It enables the conversion of entire databases to a new version of the reference genome in considerably less time than generating descriptions versus this new reference genome from scratch.

This transformation can be done by generating the HGVS description of the original reference string versus the new reference string, and then computing the new HGVS description for each instance by combining its description with the description of the reference genomes. This results in a linear (in terms of the description) amount of work for each instance. The actual implementation of the merging is beyond the scope of this chapter. However, to give an intuition for a possible implementation we provide a small example. Consider the HGVS descriptions `g.[5_14inv]` and `g.[3T>C;9G>C]`. The merging of non-overlapping variants is trivial. Positions might have to be offset based on the length of insertions and deletions in the prefix. For overlapping variants we can either construct the corresponding trivial deletion/insertion, i.e., `g.[5_14delinsCGACCGAT]` or alternatively split the inversion into two inversions separated by a substitution: `g.[10_14inv; 9G>C;5_8inv]`. Although the resulting description is a valid HGVS description, a more concise description might be found when running the extraction algorithm directly.

3.5 Conclusion

We introduced an algorithm to extract HGVS descriptions from raw DNA sequences with respect to reference sequences. We made this algorithm computationally efficient for highly similar strings by introducing an alternative version of the classic LCS algorithm using overlapping and non-overlapping k-mers. We showed that the combination of these algorithms is able to compute the HGVS descriptions of large DNA strings in a reasonable amount of computation time and that the resulting descriptions are relatively small. The HGVS descriptions yielded by the extraction algorithm are fully compliant with the Mutalyzer tool suite. The Name Checker tool can be used to generate the original sample string from the description.

We proposed to extend the HGVS nomenclature with the transposition

operator as it can greatly reduce the lengths of descriptions, while still being able to efficiently compute these transpositions.

In addition to having a canonical algorithm for generating HGVS descriptions we have shown that these descriptions are useful in genomic databases for their compression and transitivity properties. The automatic extraction of descriptions will be of great value to curators of existing databases: it makes updates using new versions of reference sequences or of the nomenclature and correction of HGVS descriptions very easy.

3.5.1 Future work

Nesting of variants has been proposed in an extension of the HGVS nomenclature [Taschner and den Dunnen, 2011]. Our extraction algorithm does not support nesting (with the exception of complex transpositions). A possible extension of the extraction algorithm could be made towards finding simple nested variants.

Breakpoint sequences observed with NGS sequencing technology also need to be described in sufficient detail to allow reconstruction of their sequence. The HGVS nomenclature committee is working on new guidelines involving junctions of more than one chromosome. The current version of our algorithm does not yet support transpositions involving more than one chromosome.

Other types of strings can be considered as well. We are mainly focussing on an extraction algorithm for amino acid sequences to describe changes in proteins using an altered set of edit operators.