Suffix Classification Trees

Wojciech Wieczorek WOJCIECH.WIECZOREK@US.EDU.PL
Faculty of Computer Science and Materials Science
University of Silesia in Katowice
Żytnia 12, 41-205 Sosnowiec, Poland

Olgierd Unold OLGIERD.UNOLD@PWR.EDU.PL
Department of Computer Engineering
Wrocław University of Science and Technology
Janiszewskiego 11/17, 50-372 Wrocław, Poland

Łukasz Strąk LUKASZ.STRAK@US.EDU.PL
Faculty of Computer Science and Materials Science
University of Silesia in Katowice
Żytnia 12, 41-205 Sosnowiec, Poland

Editors: Olgierd Unold, Witold Dyrka, and Wojciech Wieczorek

Abstract

In this paper, a new method for generating acyclic word graph is proposed. The essential characteristics of the method are: the construction of the data structure in linear time with respect to the size of an input and gathering factor frequencies. Moreover, it has been shown through a computational experiment that the proposed approach surpasses—with respect to AUC score—similar grammatical inference algorithms on the sequences from a real biological dataset.

Keywords: Suffix Trees, Classification, Amyloidogenicity

1. Introduction

There are a few ways to divide inference algorithms. We may be faced with infinite data and output finite state acceptor (see García et al., 2008) or with finite data and output acyclic or cyclic finite state acceptor. Acyclic acceptors (see Rulot and Vidal, 1987) represent finite sets of sequences, while cyclic acceptors (see Coste and Fredouille, 2003) represent infinite sets of sequences. Since our aim is to get a classifier for biological data, namely amyloidogenic (as examples) and non-amyloidogenic sequences (as counter-examples), we are interested in generating acyclic acceptors based on finite samples.

In the present paper, the concept of a suffix classification tree (SCT) is proposed and an algorithm for its generation is devised. In essence, an SCT is a rooted tree whose edges are labeled by strings, while vertexes store factor frequencies, i.e., how many times a certain factor occurs in examples and how many times in counter-examples. What is more, every factor that can be found on input will be included in an obtained tree. Notice how inefficient would be a straightforward solution consisting in the examination of all factors over all sequences. In general, the genetic code specifies 20 standard amino acids, which makes an alphabet of cardinality 20. For example, in order to find the frequencies of all factors of length six in a set of 10 000 strings of length 30, we would have to perform about 10^{10} elementary instructions. Our main contribution is the method of building the desired data structure in the efficient, linear way by means of Ukkonen (1995) algorithm for suffix trees. Thanks to that, the same information can be gained using only about 10^7 instructions. We also propose the definition of a function that maps a new sequence (a string) to a value that denotes the chance that the string can be a new example. An SCT together with the evaluation function establish a classifier, its quality has been verified against the classical algorithms of a similar characteristic: error-correcting grammatical inference (Rulot and Vidal, 1987), alignment-based learning (van Zaanen, 2000), and automatic distillation of structure (Solan et al., 2005). We also included to experiments two state merging algorithms: Blue-fringe (Lang et al., 1998) and Traxbar (Lang, 1992).

This paper is organized into four sections. The next (second) section introduces the notion of suffix trees and also discusses a way to use them to get information about unknown sequences. The third section describes the experimental results. The last section summarizes results collected.

2. Suffix Classification Trees

The basic data structure which will be used for classification is an SCT (suffix classification tree). We will propose a method for its construction that is based on a suffix tree known from classical algorithms on strings (Gusfield, 1997). So let us start with the definitions of a suffix tree and a suffix classification tree.

2.1. Basic Definitions and Concepts

First, we will introduce basic definitions on strings.

Definition 1 *A string S is an ordered nonempty list of characters written contiguously from left to right. A substring of a string S is a string $S[i..j]$ that occurs in S from position i to position j. Prefix and suffix are special cases of substring. If m is the number of characters in string S, then $S[1..i]$ ($1 \leq i \leq m$) is the prefix of string S that ends at position i, and $S[i..m]$ ($1 \leq i \leq m$) is the suffix of string S that begins at position i.*

Substrings are also called *factors*. To simplify our notation, ith character of S will be denoted by $S(i)$. The list of all substrings of the string $S =$ *apple* would be *apple*, *appl*, *pple*, *app*, *ppl*, *ple*, *ap*, *pp*, *pl*, *le*, *a*, *p*, *l*, *e*; $S(1) = a$, $S[3..4] = pl$.

Definition 2 *Let Q be a string of length n. The number of occurrences of Q in a string S is equal to the number of different positions i for which $S[i..(i + n - 1)] = Q$.*

Definition 3 *A suffix tree T for an m-character string S is a rooted directed tree with exactly m leaves numbered 1 to m. Each edge is labeled with a substring of S. No two edges starting out of a node can have string-labels beginning with the same character. Except for the root, every internal node has at least two children. Finally, the string obtained by concatenating all the string-labels found on the path from the root to leaf i spells out suffix $S[i..m]$, for i from 1 to m.*

Since such a tree does not exist for all strings, we assume (as is true in Figure 1) that the last character of S appears nowhere else in the string (usually denoted \$). This ensures that no suffix is a prefix of another, and that there will be m leaf nodes, one for each of the m suffixes of S. Since all internal non-root nodes are branching, there can be at most $m-1$ such nodes, and $m + (m-1) + 1 = 2m$ nodes in total (m leaves, $m-1$ internal non-root nodes, 1 root).

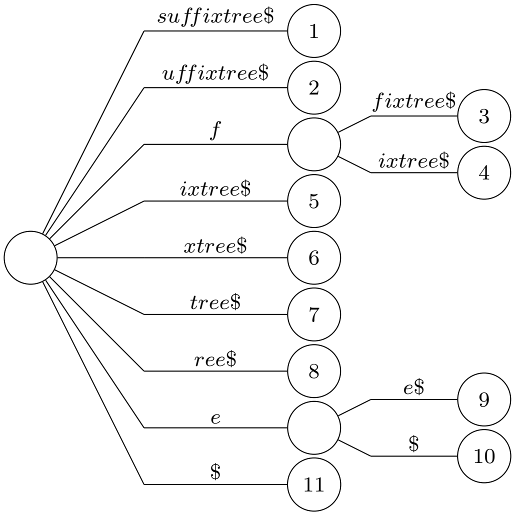


Figure 1: Suffix tree for string *suffixtree\$*. The root is the leftmost node, while leaves are numbered from 1 to 11.

A suffix tree for a string S of length m can be built in $O(m)$ time (Gusfield, 1997, chap. 6). It can be used to solve a large number of string problems that occur in text-editing, free-text search, computational biology and other application areas. Primary applications include: (i) finding the longest common substring in linear time, and (ii) finding an occurrence of any unknown string x in S or determine that x is not contained in S in $O(n)$ time (but with initial $O(m)$ time required to build the suffix tree for string S), where n is the length of x.

Definition 4 *A sample D will be an ordered pair $D = (D_+, D_-)$ where D_+, D_- are finite sets of strings with an empty intersection (have no common string). D_+ will be called the positive part of D (examples), and D_- the negative part of D (*counter-examples*).*

Definition 5 *A suffix classification tree T for a sample D is a rooted directed tree with each edge being string-labeled. No two edges starting out of a node can have string-labels beginning with the same character. Every string obtained by concatenating all the string-labels found on the path from the root to a node (the path-label of a node) spells out some substring $x = S[i..j]$, where S is an element of D (either D_+ or D_-). If x is the substring of an example or a counter-example, then there exists exactly one node with path-label x or with path-label uwy, where $x = uw$ and wy is the string-label of the last edge on the path.*

Each node, other than the root, stores two values (factor frequencies): the number (n_-) of occurrences of the path-label of that node in D_- and the number (n_+) of occurrences of the path-label of that node in D_+.

2.2. From Suffix Trees to Suffix Classification Trees

Let a sample be $D_+ = \{x_1, x_2, \ldots, x_r\}$ and $D_- = \{y_1, y_2, \ldots, y_s\}$ for $r, s \geq 1$. Create string $S = x_1 1 x_2 1 \cdots x_r 1 y_1 0 y_2 0 \cdots y_s 0 \$$, assuming that the characters 0, 1, and \$ (*special characters*) are not seen in the sample. Please observe that a suffix classification tree T_D for sample $D = (D_+, D_-)$ can be obtained from a suffix tree T_S for string S by using the below given procedure. In fact, as far as a shape is concerned, T_D is a subgraph of T_S with the same root. Notice also that in an implementation, instead of explicitly writing a substring on an edge of the tree, it is enough only to write a pair of indexes on the edge, specifying beginning and end positions of that substring in S. Since Ukkonen's algorithm (for creating T_S) has a copy of string S, it can locate any character in S using random access, i.e., in constant time.

In order to transform T_S into T_D do the following operations:

1. Let T be T_S.

2. For every node v in T initiate $n_+(v)$ and $n_-(v)$ with zero.

3. Let a current node u be the root of T and invoke the following recursive subroutine: for every edge (u, v) let ℓ be its string-label and process the edge according to the appropriate rule from (a) to (f).

    (a) If $\ell = \$$ then remove from T edge (u, v).

    (b) If ℓ does not contain any 0 or 1, then recursively invoke the subroutine with v as a new current node u.

    (c) If $\ell(1) = 0$ then $n_-(u) := n_-(u) + k$, next remove from T edge (u, v) along with t, where t is the subtree rooted at v and k is the number of leaves[1] in t.

    (d) If $\ell(1) = 1$ then $n_+(u) := n_+(u) + k$, next remove from T edge (u, v) along with t, where t is the subtree rooted at v and k is the number of leaves in t.

    (e) Let 0 be the first encountered special character and suppose it was at position i in ℓ; then label the edge with the prefix $\ell[1..(i-1)]$, $n_-(v) := n_-(v) + k$, remove t from T, where t is the subtree rooted at v and k is the number of leaves in t.

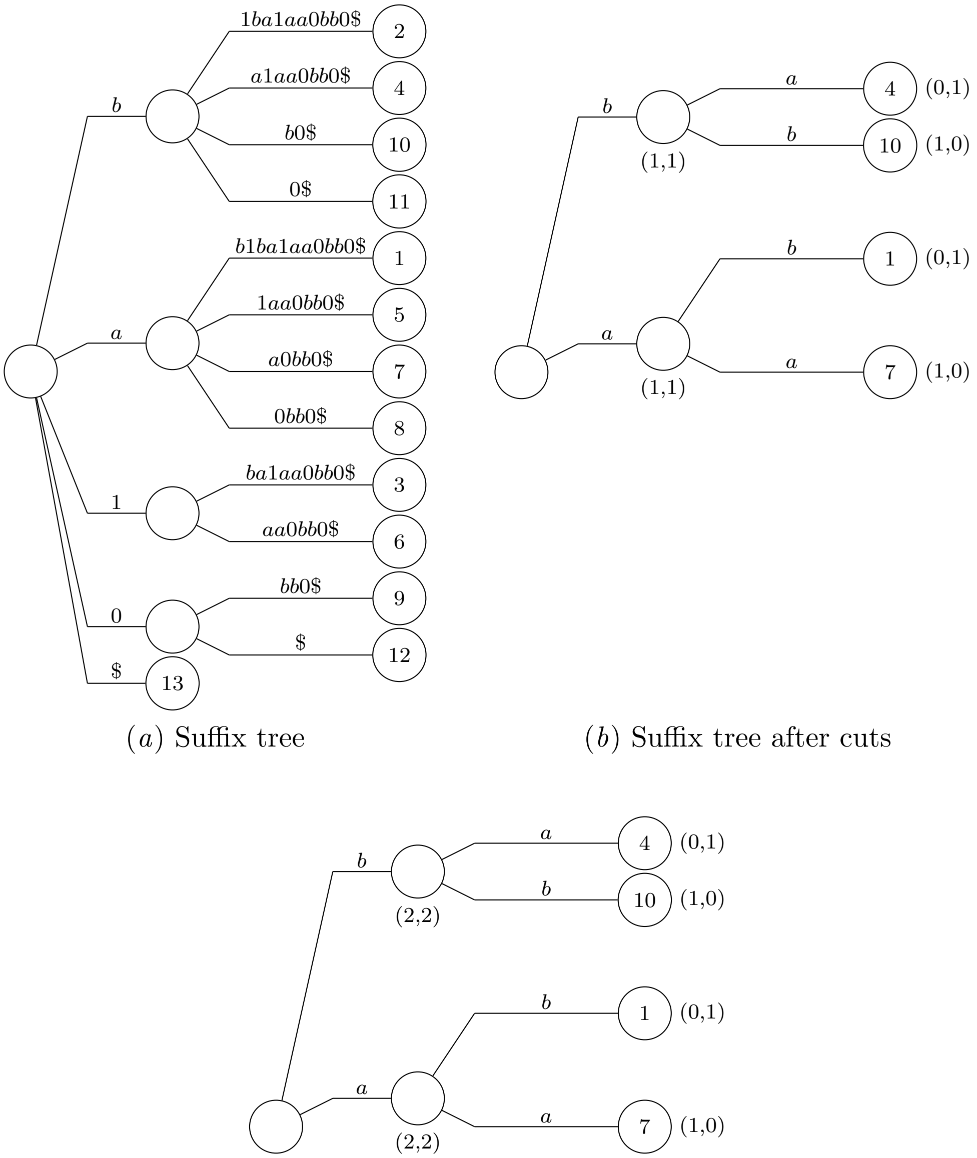
    (f) Let 1 be the first encountered special character and suppose it was at position i in ℓ; then label the edge with the prefix $\ell[1..(i-1)]$, $n_+(v) := n_+(v) + k$, remove t from T, where t is the subtree rooted at v and k is the number of leaves in t.

4. Traverse T in postorder fashion updating factor frequencies of every node (except the root) by incrementing n_+ and n_- using respective numbers from its sons (direct descendants).

5. Let T_D be T.

[1] A subtree with a single node (its root) has one leaf.

Let us look at an example. For sample $D = (\{ab, ba\}, \{aa, bb\})$ string $S = ab1ba1aa0bb0\$$ is created. Corresponding suffix tree is shown in Figure 2(a). After performing operations from 1 to 3, we got a tree shown in Figure 2(b). The next step (4) is to update $n_+(v)$ and $n_-(v)$ for every node v in the tree (except its root). The final suffix classification tree (step 5) is shown in Figure 2(c). We can read from it that, for instance, the substring b appears two times in counter-examples and two times in examples, while ab appears in no counter-example and exactly once in examples.



(a) Suffix tree

(b) Suffix tree after cuts

(c) Suffix classification tree

Figure 2: The steps of the SCT construction based on $D_+ = \{ab, ba\}$ and $D_- = \{aa, bb\}$.

An SCT is created in $O(n)$ time, where $n = \sum_{w \in D} |w|$, regardless of the length of any string w in sample D. Here are the reasons: T_S is built in $O(n)$ time and has size $O(n)$, while steps from 1 to 5 are bound by the size of the tree. Please observe also that finding the first 0 or 1 in a string-label can be performed in $O(1)$ time, provided a table with those lowest positions of special characters is maintained. For the text $ab1ba1aa0bb0\$$ from the example, it is enough to have the table $[3, 3, 3, 6, 6, 6, 9, 9, 9, 12, 12, 12]$, which—naturally—can be constructed once, in linear time.

2.3. Sequence Evaluation

Now we can proceed to the second part of our contribution, i.e., to the definition of a function $f \colon \Sigma^+ \times \mathcal{T} \times \mathcal{N} \to [0, 1]$ that maps a new string $S \in \Sigma^+$ to a real value that denotes the chance that the string can be regarded as an example. Σ is a fixed alphabet, $\mathcal{T}$ is the set of all SCTs over Σ, and $\mathcal{N}$ is the set of positive integer numbers. The idea is based on substring frequencies. For a given SCT and positive integer k, we check for all k-length substrings of S how many times they appear in the positive and negative part of a sample. This ratio determines our prediction.

The following is an algorithm for calculating $f(S, T, k)$:

```
1. Initiate $sum$ and $counter$ with 0, and $n$ with $|S|$.
2. For $position := 1, 2, \ldots, n - k + 1$ do:
    (a) $x = S[position..(position + k - 1)]$;
    (b) find in tree $T$ node $v$ with path-label $x$ or with path-label $uwy$, where $x = uw$
        and $wy$ is the string-label of the last edge on the path;
    (c) if such a node $v$ exists, then increment $counter$ by 1 and $sum$ by $n_+(v)/(n_+(v) + n_-(v))$.
3. Return $sum/counter$ (for $counter = 0$ return 0).
```

So, in other words, $f(S, T, k)$ determines a ratio that says whether in S are more substrings appearing in examples (a value closer to 1) or in counter-examples (a value closer to 0).

3. Referenced Methods and Results

The algorithm for generating suffix classification trees (SCT with $k = 3$) has been tested over a recently published amyloidogenic dataset (Wozniak and Kotulska, 2015). The dataset is composed of 1476 strings that represent protein fragments. 439 are classified as being amyloidogenic (examples), and 1037 as not (counter-examples). The shortest sequence length is 4, the longest is 83. Such a wide range of sequence lengths was an additional impediment to learning algorithms.

In order to compare our algorithm to other grammatical inference approaches, we took the methods mentioned in the introductory section as a reference: error-correcting grammatical inference (ECGI), alignment-based learning (ABL), automatic distillation of structure (ADIOS), and two Kevin Lang's state merging approaches (Blue-fringe and Traxbar).

The experiments were conducted following ten replication of ten-fold cross-validation (10×10 CV) and with 10 degrees of freedom test. This scheme, called corrected repeated k-fold CV test, was proved to to have excellent replicability (Bouckaert and Frank, 2004) and follows statistic

$$t_c = \frac{1/(k \cdot r) \sum_{i=1}^{k} \sum_{j=1}^{r} x_{ij}}{\sqrt{(1/(k \cdot r) + n_2/n_1)\hat{\sigma}^2}},$$

where x_{ij} is the difference of the performance quality between two compared algorithms on i-fold and j-run, n_1 is the number of instances used for training, n_2 the number of instances used for testing, $\hat{\sigma}^2$ is the variance of the n differences. For performing all pairwise comparisons Holm (1979) correction to p-value was applied.

In order to compare the binary classification we decided to compute Area Under the Receiver Operating Characteristic Curve (AUC). AUC is equivalent to two sample Wilcoxon rank-sum statistic (Hanley and McNeil, 1982). The value of the AUC score ranges from 0 to 1, with a score of 0.5 corresponding to random guess and a score of 1.0 indicating perfect separation of two classes (amyloids and non-amyloids). The AUC is prone to imbalanced dataset. Such a choice allowed us to deal with target scores either being probability estimates of the positive class (as an SCT with the f function) or non-thresholded measure of decisions (as returned by ABL and other methods).

Obtained results are summarized in Table 1 and Figure 3. Table 2 gives adjusted by Holm procedure p values for the comparison of the SCT as the control method with the remaining GI algorithms. There are statistically significant performance differences between SCT and all compared methods over AUC measure. Note that the new algorithm works in reasonable time (see Table 1), taking into account run-time efficiency of the other methods.

Table 1: Comparison of SCT with other GI methods in terms of averaged AUC with the standard deviation. The table is arranged in order of decreasing averaged AUC.

Method	AUC	CPU time [s]
SCT	0.805 ± 0.036	3.92 ± 0.62
ABL	0.597 ± 0.036	549.52 ± 182.10
Traxbar	0.578 ± 0.039	0.32 ± 0.06
Blue-fringe	0.576 ± 0.040	1.00 ± 0.14
ECGI	0.547 ± 0.024	51.29 ± 9.89
ADIOS	0.521 ± 0.041	14.01 ± 7.93

The results show that SCT is a method which outperforms the remaining methods as regards the AUC classification measure. All programs ran on an Intel i3-4010U, 1.7 GHz processor under Windows 10 operating system with 16 GB RAM. The computational complexity of all the algorithms is polynomially bound, however, the differences in running time were quite significant and our approach ranked among top three fastest methods. The algorithm for SCT construction[2] was written in the Python 3 programming language. The

[2] https://github.com/wieczorekw/wieczorekw.github.io/tree/master/SCT

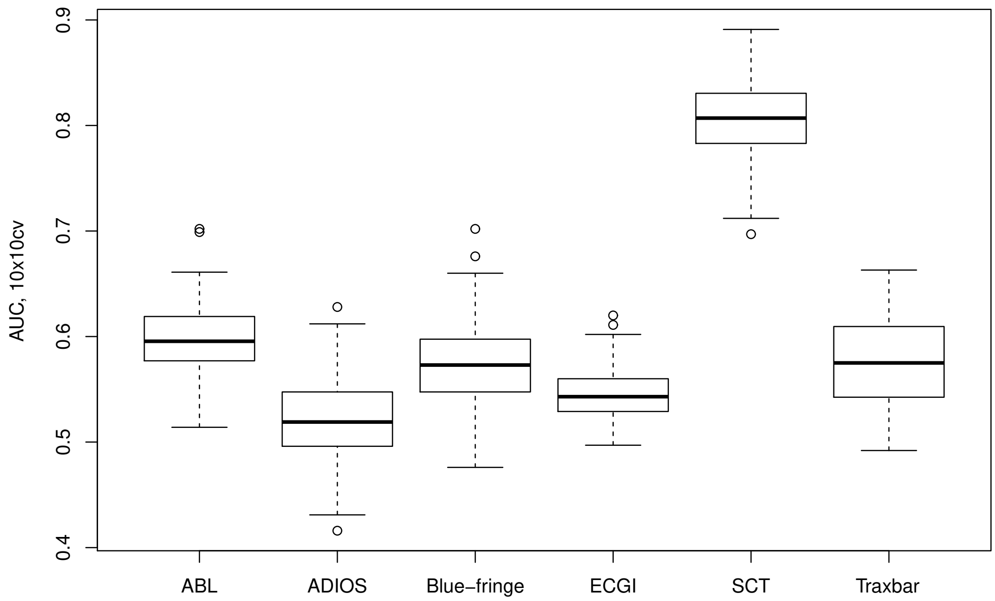


Figure 3: Performance comparison of ABL, ADIOS, Blue-fringe, ECGI, SCT, and Traxbar methods. Boxplots represent the AUC values obtained from 10 × 10 cross-validation.

Table 2: p-values for the comparison of SCT as a control method with the other methods. The initial level of confidence $\alpha = 0.05$ is adjusted by Holm procedure.

SCT versus	p-value adjusted by Holm
ABL	1.118275 e-51
Traxbar	1.821043 e-53
Blue-fringe	1.821043 e-53
ECGI	5.684131 e-67
ADIOS	4.933878 e-60

languages of implementation for five successive methods were: Python (for ABL and ECGI), Java (for ADIOS), and C (for Blue-fringe and Traxbar).

4. Conclusions

We proposed a new inference method called a suffix classification tree and applied it to a real bioinformatics task, i.e., classification of amyloidogenic sequences. The evaluation of generated SCT on an amyloidogenic dataset revealed its accuracy to predict amyloid segments. We showed that the new inference algorithm gives the best AUC in comparison to other automata or grammar learning methods that also are suitable for finite languages.

Acknowledgments

This research was supported by National Science Center, grant 2016/21/B/ST6/02158.

References

Remco R Bouckaert and Eibe Frank. Evaluating the replicability of significance tests for comparing learning algorithms. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 3–12. Springer, 2004.

François Coste and Daniel Fredouille. Unambiguous automata inference by means of state-merging methods. In *Machine Learning: ECML 2003, 14th European Conference on Machine Learning, Cavtat-Dubrovnik, Croatia, September 22-26, 2003, Proceedings*, pages 60–71, 2003.

Pedro García, Manuel Vázquez de Parga, Gloria I. Álvarez, and José Ruiz. Universal automata and nfa learning. *Theor. Comput. Sci.*, 407(1-3):192–202, November 2008. ISSN 0304-3975.

Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, New York, NY, USA, 1997. ISBN 0-521-58519-8.

James A Hanley and Barbara J McNeil. The meaning and use of the area under a receiver operating characteristic (roc) curve. *Radiology*, 143(1):29–36, 1982.

Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, pages 65–70, 1979.

K. J. Lang. Random DFA's can be approximately learned from sparse uniform examples. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, pages 45–52. ACM, 1992.

K. J. Lang, B. A. Pearlmutter, and R. A. Price. Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm. In *Proceedings of the 4th International Colloquium on Grammatical Inference*, pages 1–12. Springer-Verlag, 1998.

Héctor Rulot and Enrique Vidal. Modelling (sub)string-length based constraints through a grammatical inference method. In P. A. Devijver and J. Kittler, editors, *Proc. of the NATO Advanced Study Institute on Pattern Recognition Theory and Applications*, pages 451–459. Springer-Verlag, 1987.

Zach Solan, David Horn, Eytan Ruppin, and Shimon Edelman. Unsupervised learning of natural languages. *Proceedings of the National Academy of Sciences of the United States of America*, 102(33):11629–11634, 2005.

Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

Menno van Zaanen. ABL: Alignment-Based Learning. In *Proceedings of the 18th International Conference on Computational Linguistics (COLING)*, pages 961–967. Association for Computational Linguistics, Association for Computational Linguistics, 2000.

Pawel P. Wozniak and Malgorzata Kotulska. Amyload: Website dedicated to amyloidogenic protein fragments. *Bioinformatics*, 31(20):3395–3397, 2015. doi: 10.1093/bioinformatics/btv375.