IITM-CS6024 : Algorithmic Approaches to Computational Biology — Date: Nov 16

Assignment 4 — Due Date : Nov 26, 23:55

Roll No: BS17B011 — Name: G PRASHANT

- Use LaTeX to write-up your solutions, and submit the resulting single pdf file at GradeScope by the due date (Note: **No late submissions** will be allowed, other than one-day late submission with 10% penalty or four-day late submission with 30% penalty! Within GradeScope, indicate the page number where your solution to each question starts, else we won't be able to grade it!).
- Collaboration is encouraged, but all write-ups must be done individually and independently, and mention your collaborator(s) if any. Same rules apply for codes uploaded to HackerRank (i.e., write your own code; we will run plagiarism checks on codes).
- If you have referred a book or any other online material for obtaining a solution, please cite the source. Again don't copy the source *as is* - you may use the source to understand the solution, but write-up the solution in your own words.

1. (11 points) [Repetitive BWT]: Let BWT and BWM refer to the Burrows-Wheeler Transform and Matrix respectively. While answering biological questions below, please cite original sources (not Wikipedia!) in proper format.

   (a) (1 point) Reconstruct the string S whose BWT(S) = $nco\$toovican$.

SOLUTION

We are given BWT(S) = $nco\$toovican$. We perform inverse BWT to reconstruct the original string S. This starts with generating the first column of BWM(S) matrix, which can be obtained by lexicographically sorting BWT(S).

$$BWM(S)_{:,1} = \begin{bmatrix} \$ \\ a \\ c_1 \\ c_2 \\ i \\ n_1 \\ n_2 \\ o_1 \\ o_2 \\ o_3 \\ t \\ v \end{bmatrix}$$

Following the procedure for BWT, we get the first row of BWM(S) as follows:

$$BWM(S)_{1,:} := \$\ c\ o\ n\ v\ o\ c\ a\ t\ i\ o\ n$$

In the process, we obtain the full matrix BWM(S), which is therefore given by:

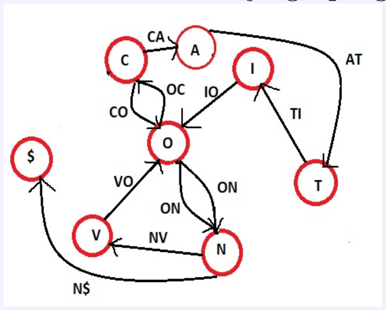
$$\begin{array}{l}
\$convocation \\
ation\$convoc \\
cation\$convo \\
convocation\$ \\
ion\$convocat \\
n\$convocatio \\
nvocation\$co \\
ocation\$conv \\
on\$convocati \\
onvocation\$c \\
tion\$convoca \\
vocation\$con
\end{array}$$

Therefore, $S := convocation$

(b) (2 points) Find a string T distinct from the above string S such that BWM(T) agrees with BWM(S) on its 2^{nd} column (and possibly other columns).

SOLUTION

If the 2^{nd} column of the matrices BWM(S) and BWM(T) are the same, then obviously, the 1^{st} columns will also be the same. It will be the sorted version of the former. For the first 2 columns of the BWM to be equal for string S and T, they should have the same 2-mer composition. We can find if there are multiple Eulerian paths in the de Bruijn graph generated by string S\$.



We can prove that the string T also starts with c. So, we traverse the de Bruijn graph starting from c by taking an alternate path, and we get *catioconvon*\$

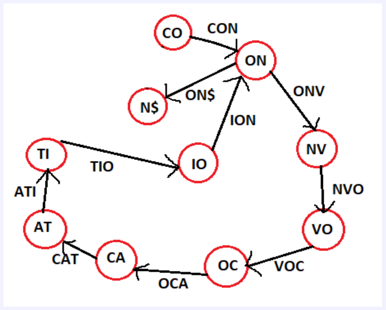
Therefore, $T := catioconvon$

Likewise, by constructing a de Bruijn graph of k-mers, we can generate other strings that have the first k columns identical in the BWM when compared to that of S.

(c) (2 points) Find a string T' distinct from the above string S such that BWM(T') agrees with BWM(S) on its 2^{nd} and 3^{rd} columns (and possibly other columns).

SOLUTION

Similar to previous question, we can construct a de Bruijn graph corresponding to 3-mers, as shown below:



From the graph, it is evident that there is only one path that starts from *co*, that gives the string S. Hence, it is not possible to generate a distinct string T' such that BWM(S) and BWM(T') have the same entries in the 2^{nd} and 3^{rd} columns.

(d) (2 points) BWT is used to compress genomes with repeats (i.e., repetitive DNA sequences). What are the different types of repeats found in genomes, and which type is best suited for a BWT-based compression?

SOLUTION

There are two types of repeats commonly found in the genome[10], they are:

- Tandem Repeats: This type of repeat occurs when a specific pattern in the DNA sequence is repeated multiple times, and the repetitive fragments are adjacent to each other. For example, a 3 nucleotide tandem repeat can be (ATG)(ATG)(ATG)(ATG), where the fragment "ATG" is repeated 4 times. These repeats can occur due to gene duplications, or more generally, they are attributed to evolution[5].

- Interspersed Repeats: Unlike tandem repeats, interspersed repeats occur when the repetitive units are dispersed across the genome, that are commonly induced by transposable elements (transposons)[8].

For BWT-based compression, Tandem repeats is best suited. After Burrows-Wheeler Transformation, all the characters that are a part of the repeating unit in the original string are grouped together, and this enables us to implement run-length encoding for further compression. The length of the grouped subtrings are higher when tandem repeats are present, thereby enabling maximum compression. Also, BWT is a one-to-one unique transformation technique that can be used for identification of tandem repeats in the genome, as reported by several studies[6][7].

(e) (2 points) We've already seen that repeats cause some issues with *de novo* genome assembly. Please list similar or other distinct issues that reference-based genome assembly using read mapping suffer from due to repetitive DNA sequences.

SOLUTION

The presence of repeats in the genome sequence lead to ambiguity and computational challenges. Often, aligners map the reads to multiple locations due to the presence of repeats in the genome, and the best matching alignment position is assigned to the read. This, however might not always lead to correct alignments. Incorrect alignments of repetitive fragments can affect downstream analyses such as variant calling procedures to identify SNPs, insertions, deletions, and other kinds of mutations[9].

(f) (2 points) Which part of a human chromosome contains the most number of repeats, and why? (To answer the "why", you can report evolutionary implications or functional roles/consequences of these repeats.)

SOLUTION

Telomeres are regions in the human chromosome that contain the most number of repetitive DNA sequences. Telomeric regions in the DNA are found in the ends of each chromosome, where they form a cap. They are required for replication and stability and consist of around 300-8,000 precise tandem repeats of the sequence $CCCTAA/TTAGGG$[4]. It was shown that the $(TTAGGG)_n$ repeat, which is primarily responsible for maintaining the stability of the chromosome, is evolutionarily conserved across several vertebrate species[3]. This repetitive portions in the telomeres serve as binding sites for proteins like TERF1, which is responsible for telomere length regulation[11]. Such GT-rich repeats are a characteristic of most eukaryotic telomeres.

2. (10 points) [A FITTING CODE]: A basic understanding of global/local sequence alignment can help you solve other useful variants of sequence alignment like the ones explored in this question.

(a) (7 points) Please mention here briefly how you would solve the fitting alignment problem, and implement it at the contest: https://www.hackerrank.com/cs6024-assignment-4-1. Please read the contest page for the fitting alignment problem definition and instructions for the output format.

SOLUTION

For the fitting alignment problem, the algorithm is similar to that of global alignment, except that we do not impose penalties for gaps in the beginning and the end of the larger sequence. In other words, we aim to align the smaller string to a substring of the larger string, thereby allowing gaps (mostly deletions) preceding and trailing the substring. We assign the smaller string across the rows and the larger string across the columns of the DAG matrix. The recurrence relation is as follows.

$$DAG(i,j) = \max \begin{cases} D(i-1,j-1) + Score(w_{i-1}, v_{j-1}) \\ D(i-1,j) - 1 \\ D(i,j-1) - 1 \end{cases} \tag{1}$$

where,

$$D(i,0) = -i \ \forall i \in \{0,1,2,\cdots n_2\}$$

and

$$D(0,j) = 0 \ \forall j \in \{0,1,2,\cdots n_1\}$$

i.e., the first row of the DAG matrix are zeros.
Here,

$$Score(w_{i-1}, v_{j-1}) = \max \begin{cases} 1 & w_{i-1} = v_{j-1} \\ -1 & \text{otherwise} \end{cases} \tag{2}$$

where the strings w (smaller string of length n_2), v (larger string of length n_1) are zero-indexed.

If the backtracking starts from

$$(n_2, j^*) = (n_2, \max_j \arg\max_j D(n_2, j))$$

i.e., the latest position corresponding to the maximum entry in the last row (maximum index of maximum entry) of the DAG matrix.

The order of preference was given as match/mismatch > insertion > deletion. While tracing backwards, the order is reversed.

(b) (3 points) What changes would you make to your code to solve the overlap alignment problem that is important for *de novo* genome assembly?

An overlap alignment of strings $v = v_1...v_n$ and $w = w_1...w_m$ is a global alignment of a suffix of v with a prefix of w. An optimal overlap alignment of strings v and w maximizes the global alignment score between an i-suffix of v and a j-prefix of w (i.e., between $v_i...v_n$ and $w_1...w_j$) among all i and j.

SOLUTION

For the overlap alignment problem, the approach is very similar to that of the fitting alignment problem explained in the previous part. The same recurrence relation is followed. The following initializations are imposed:

$$D(i, 0) = 0 \ \forall i \in \{0, 1, 2, \cdots n_2\}$$

and

$$D(0, j) = 0 \ \forall j \in \{0, 1, 2, \cdots n_1\}$$

i.e., the first row and first column of the DAG matrix are zeros.

For backtracking, we choose the terminal position, corresponding to the maximum entry of the last row and last column combined. The position for backtracking is given by

$$(k^*, l^*) = \arg\max_{(k,l)} \max_{i,j}(D(n_2, j) \cup D(i, n_1))$$

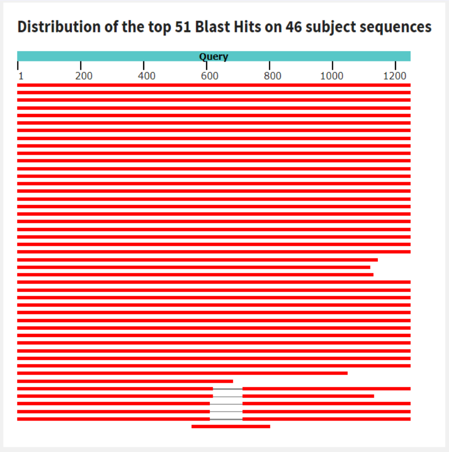
We therefore start from (k^*, l^*), which is the index corresponding to the maximum entry in the union of last row and last column, and backtrack to reach the base case, i.e, when the first row or column is reached.

3. (9 points) [Bla(s)ting thru' sequences]: Having completed a major part of the CS6024 course, you have become a popular sought-after bioinformatician by local health care officials. One morning, you receive a call from an IITM hospital staff that she has collected a viral sample from a patient with grave symptoms and sequence of a gene from that viral sample (which is attached as a sequence in FASTA format in the course moodle). She would like you to use sequence alignment to answer her queries.

   (a) (3 points) Which viral species does the sample belong to? Which other viral species is it closest to? Describe briefly how and why you came to your conclusions. Please use the NCBI BLAST toolset.

SOLUTION

The given sample belongs to viral species *Nipah henipavirus* (Generic Name: Nipah Virus). A highest match of 100% occurred between the given sample and the matrix protein (M) gene of Nipah Virus. All the 46 hits that were generated corresponded to Nipah Virus (one of which was a synthetic construct), and no other viral species closely resembled the given sample. A graphical summary of the BLAST results is shown below:



The BLAST algorithm uses a heuristic local alignment algorithm that compares the query sequence with a large database of genomic sequences. The highly statistically significant hits are identified and displayed. As mentioned, all the hits were in the same organism, Nipah Virus. This justifies the inferences given above.

(b) (3 points) What score and statistical significance fields of the NCBI BLAST output are you using to interpret the results? Provide a concise and clear definition of these score and statistical significance values, and mention the scoring matrix underlying the reported score.

SOLUTION

The two fields, *Total Score* and *E-value* (for statistical significance) were used to interpret the results. For the above example, the Total Score (for the best alignment was 2266), with an E-value of 0. The score represents the alignment score returned by the BLAST algorithm. Higher the Score, better the alignment. The Expect value (E-value) describes the expected number of hits to see by chance when searching a database of a particular size. An E-value of close to zero (less than a threshold value) suggests that the match is statistically significant. The scoring matrix underlying the reported score is governed by the match and mismatch score, which were +1 and -2, respectively. The Gap Cost was chosen to be linear (determined by mismatch score, which is -2).

(c) (3 points) You now discover a yet another sequence alignment tool called BLAT. Analyse the viral sample sequence using BLAT and interpret the results. How are the results from this tool for the above questions different? And why?

SOLUTION

The sequence sample was given as input to the BLAT tool[2] against the genomes of all the available organisms. The top 3 hits were found in the genomes of *S. cerevisiae* (35 tiles), SARS-CoV-2 (21 tiles) and Ebola Virus (13 tiles). The match with the Yeast genome might indicate viral insertion. Likewise, the match with other viral species like SARS-CoV-2 and Ebola might indicate the presence of homologous sequences. The possible reasons for the results being different with BLAST are stated below:

- The genome assembly of Nipah Virus is not present in the BLAT database, and hence there is no way that the algorithm can identify the true source of the sequence.
- The BLAT tool is predominantly used to find homology across sequences. It rapidly aligns the query sequences by missing divergent or shorter sequence alignments. It uses an algorithm which is not the same as that of BLAST.

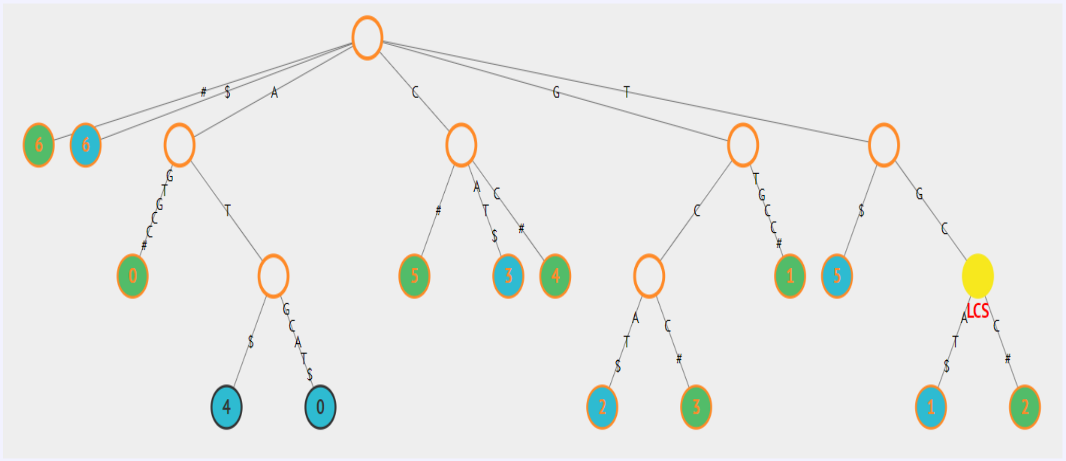
4. (10 points) [Covering ST and SA]: Let ST and SA refer to suffix tree and suffix array respectively.

   (a) (2 points) Briefly describe an algorithm that uses ST to find the longest substring shared by two strings.

SOLUTION

Consider the problem of finding the longest common substring of the two strings A and B, with $|A| = m$, $|B| = n$. Suffix trees are best suited for matching patterns in a string, and in this case, we first build a suffix tree of the string $C = A\#B\$$, where $\#$ and $\$$ are the terminal characters of the strings A and B, respectively. The leaves of this suffix tree represent indexed suffixes that correspond to one of $A\#$ and $B\$$. The internal nodes of the tree are substrings of either of A, B or both (we label them accordingly). The nodes that are substrings of both A and B have two leaves, one of which is a suffix of $A\#$, and the other is a suffix of $B\$$. To find the largest common subtring (LCS), we have to traverse through (depth first) the suffix tree to identify the internal node with common substring that has maximum depth.

For example, consider A = "$AGTGCC$", B = "$ATGCAT$". The generalized suffix tree of C = "$AGTGCC\#ATGCAT\$$" is shown below:



The yellow internal node represents LCS, which is "TGC". The algorithm takes $\mathcal{O}(m+n)$ in both time and space.
(The above algorithm was adopted from Ben Langmead's (JHU) lecture notes. The suffix tree was constructed using the tool VISUALGO)

(b) (2 points) Briefly describe an algorithm that uses ST to find the shortest substring of one string that does not appear in another string?

SOLUTION

The algorithm for finding the shortest uncommon substring (that occurs in string B but not A) is quite similar to the one previously described, where a generalized suffix tree of the string $C = A\#B\$$ is constructed. We perform a breadth-first search (BFS) to determine the first internal node that has leaves which are suffixes to only string B (i.e., nodes that labeled as "B"). The path from the root node to this internal node gives the shortest uncommon substring. Taking the same example as the previous part, "AT" is the shortest substring that occurs in B but not in A. A disadvantage of this algorithm is that it cannot identify all such shortest uncommon substrings ("CA", for example).

(c) (3 points) We saw in class how a simple depth-first traversal of a ST yields the equivalent SA, provided the outgoing edges at every node of the ST are (lexicographically) sorted. Given a ST whose edges at each node are not necessarily sorted, what is the most efficient algorithm you can think of to convert this ST to a lexicographically sorted ST? Specifically, what sorting algorithm would you use at each node and what is its per-node as well as overall running time?

SOLUTION

Given an unsorted suffix tree (ST), we can perform a depth first traversal, and as we encounter an internal node, we can sort the edges of the node by comparing the first character of the edge labels. For the purpose of sorting, we can use radix sort. Since the edges emerging from a particular node have different first characters, the radix sort algorithm will return the sorted edges in the very first step.

In general, the time complexity of a radix sort algorithm is $\mathcal{O}(k(n + |\Sigma|))$, where k is the maximum length of n strings, with a character set given by Σ.

Let m be the length of the original string S, using which the suffix tree ST was constructed, which has $m + 1$ leaves (including the \$ sign). We know that each node has at most $|\Sigma|$ children, and we only have to sort the first characters ($k = 1$). Then, the per node sorting complexity is given by $\mathcal{O}(|\Sigma| + |\Sigma|) = \mathcal{O}(|\Sigma|)$.

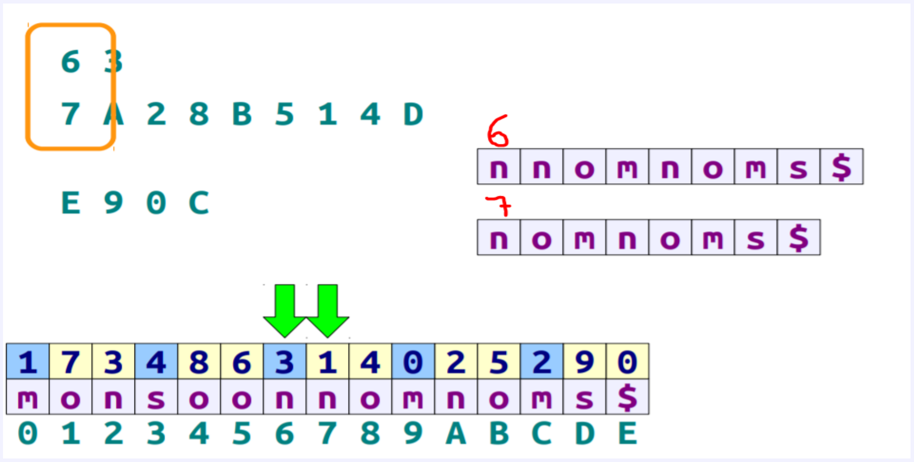
Also, for a suffix tree with $m + 1$ leaves, there has to be at most m internal nodes. Hence, the total time complexity for sorting a suffix tree is $\mathcal{O}(m|\Sigma|)$.

(d) (3 points) The DC3 algorithm for SA construction (recursively) sorts suffixes starting at positions that are **not** multiples of 3 in the first step, and uses the rank of these "sampled" suffixes to sort/merge the remaining suffixes that are at multiples of 3 in later steps. What will happen if you extend this idea to get a DC2 algorithm (which sorts odd-position suffixes in the first step, and uses it to sort/merge even-position suffixes in the next steps)? That is, will the DC2 algorithm's recursion and sort/merge steps be "easier or trickier" to implement than the DC3 algorithm steps? Justify briefly.

(Bonus: What subset of suffixes will you sort in the first step of a DC7 algorithm to get an algorithm similar to the DC3 algorithm? Besides ensuring the recursive step would work, your subset should also satisfy the following key "covering" requirement: For any pair of suffixes S_i, S_j that start at any positions i, j of the string $T = t_1 t_2 \ldots t_m$, there exists a small ℓ s.t. $i + \ell, j + \ell$ are sampled suffix positions. Then, the sort/merge step is efficient since: $S_i \leq S_j \iff (t_i, t_{i+1}, \ldots, t_{i+\ell-1}, rank(S_{i+\ell})) \leq (t_j, t_{j+1}, \ldots, t_{j+\ell-1}, rank(S_{j+\ell}))$.)

SOLUTION

The merging step of DC2 algorithm will be more trickier to implement than that of DC3 algorithm. For example, considering the merging step of the DC3 algorithm for sorting the suffixes of the string $S = monsoonnomnoms$



Slide from Stanford's lecture notes - blue indices: Sorted T_0, Yellow indices: Sorted $T_1 \cup T_2$

The image above shows an intermediate step of merging, where a comparison is performed between the suffixes indexed by 6 and 7. We can notice that the first character n is the same in both suuffixes, and since they come from different baskets (T_0 and $T_1 \cup T_2$), we cannot directly say which suffix should is lexicographically more preferred. However, when we inspect the second character, we can directly say that 6 is preferred higher (yellow indices), as the corresponding suffixes that start from second position come from the same basket ($T_1 \cup T_2$), which is already sorted.

On the other hand, the DC2 algorithm does not offer such simplicity especially in the merging step. This is because there would be alternating blue and yellow indices (with reference to the above figure), and in the case when there is the same character encountered in the suffixes being compared, the suffixes starting from the succeeding character will not be already sorted, and we will not be able to directly determine which should be given lexicological preference. Instead, considering the worst case, we have to run through the entire length of the suffixes for the same task.

Bonus Question:

For a generalized DCx algorithm, the subset of suffixes (with indices whose modulo with x is given in the set C_x) should be chosen such that C_x is minimal and it follows the "covering requirement" (minimum difference covers). In such a case, we have $C_x = \{i | X - i - 1 \in C_x'\}$. When $x = 3$, $C_3' = \{0, 1\}$ and $C_3 = \{1, 2\}$. When $x = 7$, $C_7' = \{0, 1, 3\}$ and $C_7 = \{3, 5, 6\}$.

When S_i is a suffix that starts from position i. We have

$$T_3 = \{S_i | i \mod 7 = 3\}$$

$$T_5 = \{S_i | i \mod 7 = 5\}$$

$$T_6 = \{S_i | i \mod 7 = 6\}$$

So, the subset of suffixes that is sorted in the first step of the DC7 algorithm is

$$T_u = T_3 \cup T_5 \cup T_6$$

Similar to DC3, we then sort the remaining suffixes and merge them to with sorted T_u.
The reference for the above answer is the article published by Roman Dementiev [1].

References

[1] Roman Dementiev, Juha Kärkkäinen, Jens Mehnert, and Peter Sanders. Better external memory suffix array construction. *ACM J. Exp. Algorithmics*, 12, August 2008.

[2] W. James Kent. Blat–the blast-like alignment tool. *Genome research*, 12(4):656–664, Apr 2002. 11932250[pmid].

[3] Julianne Meyne, Robert L Ratliff, and ROBERT K MoYzIs. Conservation of the human telomere sequence (ttaggg) n among vertebrates. *Proceedings of the National Academy of Sciences*, 86(18):7049–7053, 1989.

[4] Robert K Moyzis, Judy M Buckingham, L Scott Cram, Maria Dani, Larry L Deaven, Myrna D Jones, Julianne Meyne, Robert L Ratliff, and Jung-Rung Wu. A highly conserved repetitive dna sequence,(ttaggg) n, present at the telomeres of human chromosomes. *Proceedings of the National Academy of Sciences*, 85(18):6622–6626, 1988.

[5] Adam Pavlicek, Vladimir V. Kapitonov, and Jerzy Jurka. *Human Repetitive DNA*, pages 822–831. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[6] Rafal Pokrzywa. Application of the burrows-wheeler transform for searching for tandem repeats in dna sequences. *International journal of bioinformatics research and applications*, 5:432–46, 02 2009.

[7] Rafal Pokrzywa. Application of the burrows-wheeler transform for searching for tandem repeats in dna sequences. *International journal of bioinformatics research and applications*, 5(4):432–446, 2009.

[8] A.K. Sperling and R.W. Li. Repetitive sequences. In Stanley Maloy and Kelly Hughes, editors, *Brenner's Encyclopedia of Genetics (Second Edition)*, pages 150 – 154. Academic Press, San Diego, second edition edition, 2013.

[9] Todd J. Treangen and Steven L. Salzberg. Repetitive dna and next-generation sequencing: computational challenges and solutions. *Nature reviews. Genetics*, 13(1):36–46, Nov 2011. 22124482[pmid].

[10] Ronald J Trent. Chapter 1 - genes to personalized medicine. In Ronald J Trent, editor, *Molecular Medicine (Fourth Edition)*, pages 1 – 37. Academic Press, Boston/Waltham, fourth edition edition, 2012.

[11] Bas van Steensel and Titia de Lange. Control of telomere length by the human telomeric protein trf1. *Nature*, 385(6618):740–743, Feb 1997.