GENOME-SCALE ALGORITHM DESIGN
by Veli Mäkinen, Djamal Belazzougui, Fabio Cunial and Alexandru I. Tomescu
Cambridge University Press, 2nd edition, 2023
www.genome-scale.info

Exercises for Chapter 8. Classical indexes

8.1 Consider a k-mer index in which the pointers to the lists of occurrences of k-mers are encoded using a table of size σ^k in which the ith element is a pointer to the list of the occurrence positions of k-mer $P = p_1p_2 \cdots p_k$ in $T = t_1t_2 \cdots t_n$ for $i = p_1 \cdot \sigma^{k-1} + p_2 \cdot \sigma^{k-2} + \cdots + p_k$, where each $p_j \in \{0, 1, \ldots, \sigma - 1\}$. Show that both this table and the gamma-encoded lists can be constructed in $O(\sigma^k + n)$ time by two scans from left to right over T. *Hint.* Use a first scan to determine the size of the lists and a second scan to fill both the lists and the table.

8.2 The table of the previous assignment is sparse if k is large. Show that by using techniques from Section 3.2 one can represent the table in $\sigma^k(1 + o(1)) + \ell \log(n + 2n \log \ell)$ bits, where ℓ is the number of non-empty lists, while still retaining constant-time access to the list. *Hint.* Use a bitvector marking the non-empty elements and supporting rank queries. Concatenate all encodings of the occurrence lists and store a pointer to the starting position of each list inside the concatenation.

8.3 Show how to represent the pointer table of the previous assignment in $\sigma^k(1+o(1))+(\ell+n(1+2\log \ell))(1+o(1))$ bits. *Hint.* Use a bitvector supporting rank queries and another supporting select queries.

8.4 Derive an analog of Theorem 8.2 replacing γ-coding with δ-coding. Do you obtain a better bound?

8.5 The order-k *de Bruijn graph* on text $T = t_1t_2 \cdots t_n$ is a graph whose vertices are $(k-1)$-mers occurring in T and whose arcs are k-mers connecting $(k-1)$-mers consecutive in T. More precisely, let $\texttt{label}(v) = \alpha_1\alpha_2 \cdots \alpha_{k-1}$ denote the $(k-1)$-mer of vertex v and $\texttt{label}(w) = \beta_1\beta_2 \cdots \beta_{k-1}$ the $(k-1)$-mer of vertex w. There is an arc e from vertex v to w with $\texttt{label}(e) = \alpha_1\alpha_2 \cdots \alpha_{k-1}\beta_{k-1}$ iff $\alpha_2\alpha_3 \cdots \alpha_{k-1} = \beta_1\beta_2 \cdots \beta_{k-2}$ and $\texttt{label}(e)$ is a substring of T. See Section 9.7 for more details on de Bruijn graphs and on their efficient representation. Modify the solution to Exercise 8.1 to construct the order-k de Bruijn graph.

8.6 The table of size σ^k on large k makes the above approach infeasible in practice. Show how hashing can be used for lowering the space requirement.

8.7 Another way to avoid the need for a table of size σ^k is to use a suffix array (enhanced with an LCP array) or a suffix tree. In the suffix tree of T, consider all paths from the root that spell strings of length k; these are the distinct k-mers occurring in T. Consider one such path labeled by X ending at an edge leading to node v. The leaves in the subtree of v contain all the suffixes prefixed by k-mer X in lexicographic order. Show that one can sort all these lexicographically sorted lists of occurrences of k-mers in $O(n)$ time, to form the k-mer index.

8.8 Modify the approach in the previous assignment to construct the de Bruijn graph in linear time.

8.9 Visualize the suffix array of the string `ACGACTGACT#` and simulate a binary search on the pattern `ACT`.

8.10 Recall the indicator vector $I[1..n]$ for string $T = t_1 t_2 \cdots t_n$ with $I[i] = 1$ if suffix $T_{i..n}$ is small, that is, $T_{i..n} < T_{i+1..n}$, and $I[i] = 0$ otherwise. Show that it can be filled with one $O(n)$ time scan from right to left.

8.11 Recall R and SA_R from the linear-time suffix array construction. Give pseudocode for mapping SA_R to suffix array SA_T of T so that $\mathsf{SA}_T[i]$ is correctly computed for small suffixes.

8.12 Give the pseudocode for the linear-time algorithm sketched in the main text to fill suffix array entries for large suffixes, assuming that the entries for small suffixes have already been computed.

8.13 We assumed there were fewer small suffixes than large suffixes. Consider the other case, especially, how does the string sorting procedure need to be changed in order to work correctly for substrings induced by large suffixes? Solve also the previous assignments switching the roles of small and large suffixes.

8.14 Show how to compute the $\mathsf{LCP}[2..n]$ array in linear time given the suffix array. *Hint.* With the help of the inverse of the suffix array, you can scan the text $T = t_1 t_2 \cdots t_n$ from left to right, comparing suffix $t_1 t_2 \cdots$ with its predecessor in suffix array order, suffix $t_2 \cdots$ with its predecessor in suffix array order, and so on. Observe that the common prefix length from the previous step can be used in the next step.

8.15 Visualize the suffix tree of a string `ACGACTGACT` and mark the nodes corresponding to maximal repeats. Visualize also Weiner links.

8.16 Consider the suffix tree built from the suffix array. Give a linear-time algorithm to compute suffix links for its leaves.

8.17 Visualize the suffix tree on a concatenation of two strings of your choice and mark the nodes corresponding to maximal unique matches. Choose an example with multiple nodes with two leaf children, of which some correspond to maximal unique matches, and some do not.

8.18 Show that the number of implicit Weiner links can be bounded by $2n - 2$. *Hint.* Show that all implicit Weiner links can be associated with unique edges of the suffix tree of the reverse.

8.19 Show that the number of implicit Weiner links can be bounded by n. *Hint.* Characterize the edges where implicit Weiner links are associated in the reverse. Observe that these edges form a subset of a cut of the tree.

8.20 Recall the indicator vector $I[1..|C|]$ in the computation of MUMs on multiple sequences. Show how it can be filled in linear time.

8.21 The *suffix trie* is a variant of suffix tree, where the suffix tree edges are replaced by unary paths with a single character labeling each edge: the concatenation of unary path labels equals the corresponding suffix tree edge label. The suffix trie can hence be quadratic in the size of the text (which is a reason for the use of suffix trees instead). Now consider the suffix link definition for the nodes of the suffix trie that do *not* correspond to nodes of suffix tree. We call such nodes *implicit nodes* and such suffix links *implicit suffix links* of the suffix tree. Show how to *simulate* an implicit suffix link $sl(v, k) = (w, k')$, where $k > 0$ ($k' \geq 0$) is the position inside the edge from the parent of v to v (the parent of w to w), where $v = (v, 0)$ and $w = (w, 0)$ are *explicit* nodes of the suffix tree and (v, k) (possibly also (w, k')) is an implicit node of suffix tree. In the simulation, you may use parent pointers and explicit suffix links. What is the worst-case computation time for the simulation?

8.22 Consider the following *descending suffix walk* by string $S = s_1 s_2 \cdots s_m$ on the suffix *trie* of $T = t_1 t_2 \cdots t_n$. Walk down the suffix tree of T as deep as the prefix of S still matches the path followed. When you cannot walk any more, say at s_i, follow a suffix link, and continue walking down as deep as the prefix of $s_i s_{i+1} \cdots$ still matches the path followed, and follow another suffix link when needed. Continue this until you reach the end of S. Observe that the node v visited during the walk whose string depth is greatest defines the *longest common substring* of S and T: the string X labeling the path from root to v is such substring. Now, consider the same algorithm on the suffix *tree* of T. Use the simulation of implicit suffix links from previous assignment. Show that the running time is $O(\sigma + |T| + |S| \log \sigma)$, i.e., the time spent in total for simulating implicit suffix links can be *amortized* on scanning S.

8.23 The suffix tree can also be constructed directly without requiring the suffix array. *Online* suffix tree construction works as follows. Assume you have the suffix tree $\mathsf{ST}(i)$ of $t_1 t_2 \cdots t_i$, then update it into the suffix tree $\mathsf{ST}(i+1)$ of $t_1 t_2 \cdots t_{i+1}$ by adding t_{i+1} at the end of each suffix. Let $(v_1, k_1), (v_2, k_2), \ldots, (v_i, k_i)$ be the (implicit) nodes of $\mathsf{ST}(i)$ corresponding to paths labeled $t_1 t_2 \cdots t_i$, $t_2 t_3 \cdots t_i$, …, t_i, respectively (see the assigments above for the notion of implicit and explicit nodes). Notice that $sl(v_j, k_j) = (v_{j+1}, k_{j+1})$.

a) Show that $(v_1, k_1), (v_2, k_2), \ldots, (v_i, k_i)$ can be split into lists $(v_1, 0), \ldots, (v_l, 0)$, $(v_{l+1}, k_{l+1}), \ldots, (v_a, k_a)$, and $(v_{a+1}, k_{a+1}), \ldots, (v_i, k_i)$ such that nodes in the first list are leaves before and after appending t_{i+1}, nodes in the second list do not yet have a branch (or a next character along the edge) starting with character t_{i+1}, and nodes in the third list already have a branch (or a next character along the edge) starting with character t_{i+1}.

b) From the above it follows that the status of nodes $(v_1, 0), \ldots, (v_l, 0)$ remains the same and no updates to their content are required (the start position of the suffix is a sufficient identifier). For $(v_{l+1}, k_{l+1}), \ldots, (v_a, k_a)$ a new leaf for t_{i+1} needs to be created. In the case of explicit node $(v, 0)$, this new leaf is inserted under node v. In the case of implicit node (v, k), the edge from the parent of v to v is split after position k, and a new internal node is created, inheriting v as child and the old parent of v as parent, and t_{i+1} as a new leaf child. The suffix links from the newly created nodes need to be created during the traversal of the list $(v_{l+1}, k_{l+1}), \ldots, (v_{a+1}, k_{a+1})$. Once one has detected (v_{a+1}, k_{a+1}) being already followed by the required character and created the suffix link from the last created node to the correct successor of (v_{a+1}, k_{a+1}), say (w, k'), the list does not need to be scanned further. To insert t_{i+2}, one can start from the new *active point* (w, k') identically as above, following suffix links until one can walk down with t_{i+2}. Notice the analogy to descending suffix walk, and use a similar analysis as there to show that the simulation of implicit suffix links amortizes and that the online construction requires $O(n)$

steps (the actual running time is multiplied by an alphabet factor depending on how the branching is implemented).

8.24 The *directed acyclic word graph* (DAWG) is an automaton obtained by minimizing a suffix trie. Suppose that we build the suffix trie of a string $T\#$. Then two nodes v and u of the suffix trie will be merged if the substring that labels the path to v is a suffix of the substring that labels the path to u and the frequencies of the two substrings in $T\#$ are the same. Show that the number of vertices in the DAWG is linear in the length of T, by showing that the following statements hold.

a) There is a bijection between the nodes of the suffix-link tree of $\underline{T}\#$ and the vertices of the DAWG labelled by strings whose frequency is at least two in $T\#$.

b) There is a bijection between all prefixes of $T\#$ that have exactly one occurrence in $T\#$ and all the vertices of the DAWG labeled by strings whose frequency is exactly one in $T\#$.

Then show that the number of edges is also linear, by showing a partition of the edges into three categories:

a) edges that are in bijection with the edges of the suffix link tree of $\underline{T}\#$.

b) edges that are in bijection with a subset of the edges of the suffix tree of $T\#$, and

c) edges that connect two nodes that are in bijection with two prefixes of $T\#$ of lengths differing by one.

8.25 The *compacted DAWG* (CDAWG) of a string $T\#$ is obtained by merging every vertex of the DAWG of $T\#$ that has just one outgoing arc with the destination of that arc. Equivalently, the CDAWG can be obtained by minimizing the suffix tree of T in the same way as the suffix trie is minimized to obtain the DAWG. Note that the resulting automaton contains exactly one vertex with no outgoing arc, called the *sink*. Show that there is a bijection between the set of maximal repeated substrings of $T\#$ and the set of vertices of the CDAWG, excluding the root and the sink.

8.26 Complete the details of the $O(kn)$ approximate string matching algorithm of Section 8.7 by giving a detailed pseudocode including initialization and reporting of the matches.