The longest letter-duplicated subsequence and related problems

Wenfeng Lai, Adiesha Liyanage, Binhai Zhu, Peng Zou

Acta Informatica (2024) 61:315–329
https://doi.org/10.1007/s00236-024-00459-7

ORIGINAL ARTICLE

The longest letter-duplicated subsequence and related problems

Wenfeng Lai[1] · Adiesha Liyanage[2] · Binhai Zhu[2] · Peng Zou[2]

Received: 9 December 2023 / Accepted: 11 July 2024 / Published online: 20 July 2024

Abstract

Motivated by computing duplication patterns in sequences, a new problem called the longest letter-duplicated subsequence (LLDS) is proposed. Given a sequence S of length n, a letter-duplicated subsequence is a subsequence of S in the form of $x_1^{d_1}x_2^{d_2}\ldots x_k^{d_k}$ with $x_i \in \Sigma$, $x_j \neq x_{j+1}$ and $d_i \geq 2$ for all i in $[k]$ and j in $[k-1]$. A linear time algorithm for computing a longest letter-duplicated subsequence (LLDS) of S can be easily obtained. In this paper, we focus on two variants of this problem: (1) 'all-appearance' version, i.e., all letters in Σ must appear in the solution, and (2) the weighted version. For the former, we obtain dichotomous results: We prove that, when each letter appears in S at least 4 times, the problem and a relaxed version on feasibility testing (FT) are both NP-hard. The reduction is from $(3^+, 1, 2^-)$-SAT, where all 3-clauses (i.e., containing 3 lals) are monotone (i.e., containing only positive literals) and all 2-clauses contain only negative literals. We then show that when each letter appears in S at most 3 times, then the problem admits an $O(n)$ time algorithm. Finally, we consider the weighted version, where the weight of a block $x_i^{d_i}$ $(d_i \geq 2)$ could be any positive function which might not grow with d_i. We give a non-trivial $O(n^2)$ time dynamic programming algorithm for this version, i.e., computing an LD-subsequence of S whose weight is maximized.

Adiesha Liyanage, Binhai Zhu and Peng Zou have contributed equally.

✉ Binhai Zhu
bhz@montana.edu

Wenfeng Lai
2290892069@qq.com

Adiesha Liyanage
adiesha@gmail.com

Peng Zou
pengzou@gmail.com

[1] School of Computer Science and Technology, Shandong University, Qingdao 266237, Shandong, China

[2] Gianforte School of Computing, Montana State University, Bozeman, MT 59717, USA

1 Introduction

In biology, duplication is an important part of evolution. There are two kinds of duplications: arbitrary segmental duplications (i.e., select a segment and paste it somewhere else) and tandem duplications (which is in the form of $X \to XX$, where X is any segment of the input sequence). It is known that the former duplications occur frequently in cancer genomes [4, 16, 20]. On the other hand, the latter are common under different scenarios, for example, it is known that the tandem duplication of 3 nucleotides CAG is closely related to the Huntington disease [15]. In addition, tandem duplications can occur at the genome level (acrossing different genes) for certain types of cancer [17]. In fact, as early as in 1980, Szostak and Wu provided evidence that gene duplication is the main driving force behind evolution, and the majority of duplications are tandem [21]. Consequently, it was not a surprise that in the first sequenced human genome around 3% of the genetic contents are in the form of tandem repeats [13].

Independently, tandem duplications were also studied in *copying systems* [7]; as well as in formal languages [2, 5, 22]. In 2004, Leupold et al. posed a fundamental question regarding tandem duplications: what is the complexity to compute the minimum tandem duplication distance between two sequences A and B (i.e., the minimum number of tandem duplications to convert A to B). In 2020, Lafond et al. [9] answered this open question by proving that this problem is NP-hard for an unbounded alphabet. In fact, Lafond et al. proved later that the problem is NP-hard even if $|\Sigma| \geq 4$ by encoding each letter in the unbounded alphabet proof with a square-free string over a new alphabet of size 4 (modified from Leech's construction [14]), which covers the case most relevant with biology, i.e., when $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ (for DNA sequences) or $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{U}\}$ (for RNA sequences) [11]. Independently, Cicalese and Pilati showed that the problem is NP-hard for $|\Sigma| = 5$ using a different encoding method [3].

Motivated by the above applications (especially when some mutations occur after the duplications), some new problems related to duplications are proposed and studied in this paper. Given a sequence S of length n, a letter-duplicated subsequence (LDS) of S is a subsequence of S in the form $x_1^{d_1} x_2^{d_2} \dots x_k^{d_k}$ with $x_i \in \Sigma$, where $x_j \neq x_{j+1}$ and $d_i \geq 2$ for all i in $[k]$ and j in $[k-1]$. (Each $x_i^{d_i}$ is called an LD-block.) Naturally, the problem of computing a longest letter-duplicated subsequence (LLDS) of S can be defined, and a simple linear time algorithm can be obtained. An example can show the idea behind this problem: $B = AACACAGATGAT$, and due to local mutations, insertions and deletions it becomes $S = AACACGTCGAT$, but a longest letter-duplicated subsequence $X_1 = AACCGG$ or $X_2 = AACCTT$ would still give us the skeleton of the initial sequence B. (Recently, Lafond et al. [10] have considered a slightly more complex version but the corresponding running times are significantly higher. In the conclusion section, we will discuss that perspective a little more.)

We remark that recently a similar problem called *longest run subsequence* was studied by Schrinner et al. [18, 19], it differs from our problem in that each letter appears consecutively at most once in the solution as a run (which is a substring containing one or more repetitions of the same letter), and the goal is the same, i.e., the length of such a subsequece is to be maximized. For this problem, additional results on FPT intractability can be found in [6] and additional approximation results can be found in [1].

In this paper, we focus on some important variants around the LLDS problem, focusing on the constrained and weighted cases. The constraint is to demand that all letters in Σ appear in a resulting LDS, which simulates that in a genome with duplicated genes, how to

compute the maximum duplicated pattern while including all the genes. Then we have two problems: feasibility testing (FT for short, which decides whether an LDS of S containing all letters in Σ exists) and the problem of maximizing the length of a resulting LDS where all letters in the alphabet appear, which we call LLDS+. It turns out that the status of these two problems change quite a bit when d, the maximum number a letter can appear in S, varies. We denote the corresponding problems as FT(d) and LLDS+(d) respectively. Let $|S| = n$, we summarize our main results in this paper as follows:

1. We show that when $d \geq 4$, both FT(d) and (the decision version of) LLDS+(d) are NP-complete, which implies that LLDS+(d) does not have a polynomial-time approximation algorithm when $d \geq 4$.
2. We show that when $d = 3$, both FT(d) and LLDS+(d) admit an $O(n)$ time algorithm, by exploiting a new property of the problem.
3. When a weight of an LD-block is any positive function (i.e., it does not even have to grow with its length), we present a non-trivial $O(n^2)$ time dynamic programming solution for this Weighted-LDS problem.

Note that the parameter d, i.e., the maximum duplication number, is of practical interest in bioinformatics, since in many genomes duplication is a rare event and the number of duplicates is usually a small constant. For example, it is known that plants have undergone up to three rounds of whole genome duplications, resulting in a number of duplicates bounded by 8 [23].

An earlier version of this paper appeared in [12], where the NP-completeness results were only shown for $d \geq 6$ and the some partial results were shown for $d = 3$ (a huge gap was left open for $d = 4, 5$). In this paper, we close this gap completely.

This paper is organized as follows. In Sect. 2 we give necessary definitions. In Sect. 3 we focus on showing that the LLDS+ and FT problems are NP-complete when $d \geq 4$. In Sect. 4 we present the linear time algorithms for both FT and LLDS+ when $d = 3$. In Sect. 5 we give polynomial-time algorithms for Weighted-LDS. We conclude the paper in Sect. 6, where we summarize our results and also discuss some related recent research.

2 Preliminaries

Let $\mathbb{N}$ be the set of natural numbers. For $q \in \mathbb{N}$, we use $[q]$ to represent the set $\{1, 2, ..., q\}$. Throughout this paper, a sequence S is over a finite alphabet Σ. We use $S[i]$ to denote the i-th letter in S and $S[i..j]$ to denote the substring of S starting and ending with indices i and j respectively. (Sometimes we also use $(S[i], S[j])$ as an interval representing the substring $S[i..j]$.) With the standard run-length representation, S can be represented as $y_1^{a_1} y_2^{a_2} \dots y_q^{a_q}$, with $y_i \in \Sigma$, $y_j \neq y_{j+1}$ and $a_j \geq 1$, for $i \in [q]$, $j \in [q-1]$. If a letter x appears multiple times in S, we could use $x^{(i)}$ to denote the i-th copy of it (reading from left to right). Finally, a *subsequence* of S is a string obtained by deleting some letters in S.

2.1 The LLDS problem

A subsequence S' of S is a letter-duplicated subsequence (LDS) of S if it is in the form of $x_1^{d_1} x_2^{d_2} \dots x_k^{d_k}$, with $x_i \in \Sigma$, $x_j \neq x_{j+1}$ and $d_i \geq 2$, for $i \in [k]$, $j \in [k-1]$. We call each $x_i^{d_i}$ in S' a *letter-duplicated block* (LD-block, for short). For instance, let $S = abcacabcb$, then $S_1 = aaabb$, $S_2 = ccbb$ and $S_3 = ccc$ are all letter-duplicated subsequences of S, where aaa and bb in S_1, cc and bb in S_2, and ccc in S_3 all form the corresponding LD-blocks.

Certainly, we are interested in the longest ones — which gives us the longest letter-duplicated subsequence (LLDS) problem.

As a warm-up, we solve this problem by dynamic programming. We first have the following observation.

Observation 1 *Suppose that there is an optimal LLDS solution for a given sequence S of length n, in the form of $x_1^{d_1} x_2^{d_2} \ldots x_k^{d_k}$. Then it is possible to decompose it into a generalized LD-subsequence $y_1^{e_1} y_2^{e_2} \ldots y_p^{e_p}$, which has the following properties:*

- $2 \leq e_i \leq 3$, *for* $i \in [p]$,
- $p \geq k$,
- y_j *does not have to be different from* y_{j+1}, *for* $j \in [p-1]$.

The proof is straightforward: For any natural number $\ell \geq 2$, we can decompose it as $\ell = \ell_1 + \ell_2 + \ldots + \ell_z \geq 2$, such that $2 \leq \ell_j \leq 3$ for $1 \leq j \leq z$. Consequently, for every $d_i > 3$, we could decompose it into a sum of 2's and 3's. Then, clearly, given a generalized LD-subsequence, we could easily obtain the corresponding LD-subsequence by combining $y_i^{e_i} y_{i+1}^{e_{i+1}}$ when $y_i = y_{i+1}$.

We now design a dynamic programming algorithm for LLDS. Let $L(i)$ be the length of the optimal LLDS solution for $S[1..i]$. The recurrence for $L(i)$ is as follows.

$$
\begin{aligned}
&L(0) = 0,\\
&L(1) = 0,\\
&L(i) = \max \begin{cases} L(i-x-1)+2 & x = \min\{x | S[i-x] = S[i]\}, x \in (0, i-1] \\ L(i-x)+1 & x = \min\{x | S[i-x] = S[i]\}, x \in (0, i-1] \\ L(i-1) & \text{otherwise.} \end{cases}
\end{aligned}
$$

Note that the step involving $L(i-x)+1$ is essentially a way to handle a generalized LD-subsequence of length 3 (by keeping $S[i-x]$ for the next level computation) and cannot be omitted following the above observation. For instance, if $S = dabcdd$ then without that step we would miss the optimal solution ddd.

The value of the optimal LLDS solution for S can be found in $L(n)$. For the running time, for each $S[x]$ we just need to scan S to find the closest $S[i]$ such that $S[x] = S[i]$. With this information, the table L can be filled in linear time. With a simple augmentation, the actual sequence corresponding to $L(n)$ can also be found in linear time. Hence LLDS can be solved in $O(n)$ time.

2.2 The variants of LLDS

In this paper, we focus on the following variations of the LLDS problem.

Definition 1 ***Constrained Longest Letter-Duplicated Subsequence*** ($LLDS+$ for short)

Input: A sequence S with length n over an alphabet Σ and an integer ℓ.

Question: Does S contain a letter-duplicated subsequence S' with length at least ℓ such that all letters in Σ appear in S'?

Definition 2 ***Feasibility Testing*** (FT for short)

Input: A sequence S with length n over an alphabet Σ.

Question: Does S contain a letter-duplicated subsequence S'' such that all letters in Σ appear in S''?

For LLDS+ we are really interested in the optimization version, i.e., to maximize ℓ. Note that, though looking similar, FT and the decision version of LLDS+ are different: if there is no feasible solution for FT, certainly there is no solution for LLDS+; but even if there is a feasible solution for FT, computing an optimal solution for LLDS+ could still be non-trivial.

Finally, let d be the maximum number of times a letter in Σ appears in S. Then, we can represent the corresponding versions for LLDS+ and FT as LLDS+(d) and FT(d) respectively.

It turns out that (the decision version of) LLDS+(d) and FT(d) are both NP-complete when $d \geq 4$. When $d = 3$, FT(3) can be decided in $O(n)$ time using a simple yet interesting property of the problem, on top of that we can solve LLDS+(3) also in $O(n)$ time using dynamic programming. We present the details in the next two sections. In Sect. 5, we will consider an extra version of LLDS, Weighted-LDS, where the weight of an LD-block is an arbitrary positive function.

3 Hardness for LLDS+(d) and FT(d) when $d \geq 4$

3.1 FT(4) is NP-complete

The idea is to reduce a special version of SAT, which we call $(3^+, 1, 2^-)$-SAT, to FT(4). We first show that is NP-complete; in this version all variables appear positively in 3-CNF clauses (i.e., clauses containing exactly 3 positive literals), and each variable appears exactly once in total; moreover, the negation of the variables appear in 2-CNF clauses (i.e., clauses containing 2 negative literals), possibly many times (in fact, at least 5 times when the graph has a leaf node as can be seen next). A *valid* truth assignment for an $(3^+, 1, 2^-)$-SAT instance ϕ is one which makes ϕ True; moreover, each 3-CNF clause has exactly one true literal.

The reduction was folklore and a sketch of proof was posted in the internet https://cs.stackexchange.com/questions/16634; here we give a formal proof.

Theorem 1 *$(3^+, 1, 2^-)$-SAT is NP-complete.*

Proof As the problem is easily seen to be in NP, let us focus more on the reduction from 3-COLORING. In 3-COLORING, given a graph $G = (V, E)$, one needs to assign one of the 3 colors to each of the vertex $u \in V$ such that for any edge $(u, v) \in E$, u and v are giving different colors.

For each vertex u, we use u_1, u_2 and u_3 to denote the 3 colors, then, obviously, we have the 3-CNF clause $(u_1 \vee u_2 \vee u_3)$. Therefore, the conjunction of positive 3-CNF clauses is

$$C^+ = \bigwedge_{u \in V} (u_1 \vee u_2 \vee u_3).$$

We have two kinds of 2-CNF clauses. First, for each $u \in V$, we have a type-1 2-CNF clause (which means that color-i and color-j cannot be both used to color u:

$$\overline{u_i \wedge u_j} = (\bar{u}_i \vee \bar{u}_j),$$

for $1 \leq i \neq j \leq 3$. Then, for each edge $(u, v) \in E$, we have a type-2 2-CNF clause (which means that one cannot color u and v with the same color-i):

$$\overline{u_i \wedge v_i} = (\bar{u}_i \vee \bar{v}_i),$$

for $i = 1, 2, 3$.

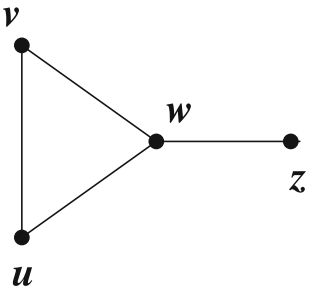

Fig. 1 A simple example for the reduction from 3-COLORING to $(3^+, 1, 2^-)$-SAT. The conjunction of positive 3-CNF clauses is $C^+ = F_1^+ \wedge F_2^+ \wedge F_3^+ \wedge F_4^+ = (u_1 \vee u_2 \vee u_3) \wedge (v_1 \vee v_2 \vee v_3) \wedge (w_1 \vee w_2 \vee w_3) \wedge (z_1 \vee z_2 \vee z_3)$

Let C^- be the conjunction of these 2-CNF clauses. Then $\phi = C^+ \wedge C^-$, and it is clear that G has a 3-coloring if and only if ϕ has a valid truth assignment. We proceed to prove this statement.

If G has a 3-coloring and u is colored with color-i, then we assign $u_i \leftarrow$ TRUE and $u_j, u_k \leftarrow$ FALSE, where $1 \le i \ne j \ne k \le 3$. Then the 3-CNF clause on u in C^+ is certainly satisfied, since there is exactly one true literal in it. For a type-1 2-CNF clause $(\bar{u}_i \vee \bar{u}_j), i \ne j$, since u_i is assigned TRUE, u_j must be assigned FALSE (so does u_k); consequently, for a type-1 2-CNF clause $(\bar{u}_j \vee \bar{u}_k), i \ne j \ne k$, since both u_j and u_k are assigned FALSE it is certainly satisfied. Finally, for an edge $(u, v) \in E$, since u is colored with color-i, v must be colored with a different color-j. Hence $u_i \leftarrow$ TRUE, $v_i \leftarrow$ FALSE and certainly the type-2 2-CNF clause $\bar{u}_i \vee \bar{v}_i$ is satisfied.

For the reverse direction, if ϕ has a truth assignment, for a 3-CNF clause $(u_1 \vee u_2 \vee u_3)$ in C^+, the type-1 2-CNF clauses $(\bar{u}_i \vee \bar{u}_j)$ in C^-, with $1 \le i \ne j \le 3$, would enforce that exactly one of u_1, u_2 and u_3 is assigned TRUE. On the other hand, the type-2 2-CNF clauses $(\bar{u}_i \vee \bar{v}_i)$ in C^-, with $1 \le i \le 3$, would enforce that for an edge $(u, v) \in E$, u_i and v_i cannot both be assigned TRUE. Hence, if for each u_i assigned TRUE we color $u \in V$ with color-i, then we have a valid 3-coloring for G.

The reduction obviously takes linear time. Hence the theorem is proven.

In Fig. 1, we show an example for the above reduction. According to the given graph, the type-1 2-CNF formula involving u is $F_1^- \wedge F_2^- \wedge F_3^- = (\bar{u}_1 \vee \bar{u}_2) \wedge (\bar{u}_1 \vee \bar{u}_3) \wedge (\bar{u}_2 \vee \bar{u}_3)$; the type-2 2-CNF formula corresponding to the edge (u, v) is $(\bar{u}_1 \vee \bar{v}_1) \wedge (\bar{u}_2 \vee \bar{v}_2) \wedge (\bar{u}_3 \vee \bar{v}_3)$, and for type-2 2-CNF formula corresponding to edge (u, w) is $(\bar{u}_1 \vee \bar{w}_1) \wedge (\bar{u}_2 \vee \bar{w}_2) \wedge (\bar{u}_3 \vee \bar{w}_3)$, for convenience we could assume that these clauses are labelled from F_4^- to F_9^- respectively.

We next reduce $(3^+, 1, 2^-)$-SAT to FT(4). Given the input ϕ for $(3^+, 1, 2^-)$-SAT over $3n$ variables $x_1, x_2, \ldots, x_{3n}$, we label its 3-CNF clauses as $F_1^+, F_2^+, \ldots, F_n^+$ and its 2-CNF clauses as $F_1^-, F_2^-, \ldots, F_m^-$ (for 2-CNF clauses we must always list all type-1 clauses before type-2 ones). (In Fig. 1, the vertices are labelled alphabetically which can be interpreted as $u_1 = x_1, u_2 = x_2, u_3 = x_3, v_1 = x_4$, etc.) For each variable x_i, let $L(x_i)$ be the list of type-1 2-CNF clauses containing $\bar{x}_i$, each repeating twice consecutively, followed by type-2 2-CNF clauses, again each repeating twice. Clearly, each 2-CNF clause appears exactly 4 times in all these lists (see the arguments below). As an example, following Fig. 1, $L(u_1) = F_1^- F_1^- \cdot F_2^- F_2^- \cdot F_4^- F_4^- \cdot F_7^- F_7^-$; $L(u_2) = F_1^- F_1^- \cdot F_3^- F_3^- \cdot F_5^- F_5^- \cdot F_8^- F_8^-$; and $L(u_3) = F_2^- F_2^- \cdot F_3^- F_3^- \cdot F_6^- F_6^- \cdot F_9^- F_9^-$.

Let $F_i^+ = (x_{i,1} \vee x_{i,2} \vee x_{i,3})$. We construct $H_i = F_i^+ \cdot L(x_{i,1}) \cdot F_i^+ \cdot L(x_{i,2}) \cdot F_i^+ \cdot L(x_{i,3}) \cdot F_i^+$, where F_i^+ appears 4 times as a letter in H_i. Finally we construct a sequence H as

$$H = H_1 \cdot g_1 g_1 \cdot H_2 \cdot g_2 g_2 \cdot H_3 \ldots g_{n-1} g_{n-1} \cdot H_n,$$

where each separator $g_\ell (1 \le \ell \le n-1)$ appears exactly twice. Note that each F_i^+ only appears 4 times in H_i, each type-1 2-CNF clause F_k^- containing $\bar{x}_i$ appears 4 times (twice consecutively in $L(x_{i,j})$, and twice consecutively in $L(x_{i,j'})$, with $1 \le j \ne j' \le 3$); moreover, each type-2 2-CNF clause F_k^- involving an edge (x_i, x_l) also appears 4 times in H: twice consecutively in H_i and twice consecutively in H_l. Hence, except for g_ℓ, all letters in H appears 4 times.

We claim that ϕ has a valid truth assignment if and only if H induces a feasible LDS which contains all F_j^+ $(j = 1..n)$, all F_k^- $(k = 1..m)$ and all $g_\ell g_\ell$ $(\ell = 1..n-1)$.

The forward direction, i.e., when ϕ has a valid truth assignment, is straightforward and can be proved as follows. In this case, suppose exactly one of $x_{i,1}$, $x_{i,2}$ and $x_{i,3}$ (say $x_{i,j}$, $1 \le j \le 3$) is assigned TRUE, then we delete $L(x_{i,j})$ in H_i and also delete the other 2 copies F_i^+ to form an LDS-block $F_i^+ F_i^+$. Consequently, H_i' is obtained from H_i by deleting $L(x_{i,j})$ and the two other copies of F_i^+ in H_i which are not adjacent to $L(x_{i,j})$. Since all the 2-CNF clauses appear 4 times in H, either two each in $L(x_{i,j})$ and $L(x_{i,j'})$, with $1 \le j \ne j' \le 3$, or two each in H_i and H_l where $(x_i, x_l) \in E$ — $L(x_{i,j})$ and $L(x_{l,j})$ cannot be both deleted as that would imply x_i and x_l having the same color in G. Hence, at least two of them would appear as an LDS-block of size 2 in a feasible LDS solution H', where

$$H' = H_1' \cdot g_1 g_1 \cdot H_2' \cdot g_2 g_2 \cdot H_3' \ldots g_{n-1} g_{n-1} \cdot H_n'.$$

As an example, suppose $x_{1,1}$ is assigned TRUE — corresponding to that u is labelled with color-1 in G, then we have:

$$\begin{aligned} H_1 = {} & F_1^+ \cdot L(u_1) \cdot F_1^+ \cdot L(u_2) \cdot F_1^+ \cdot L(u_3) \cdot F_1^+ \\ = {} & F_1^+ \cdot F_1^- F_1^- \cdot F_2^- F_2^- \cdot F_4^- F_4^- \cdot F_7^- F_7^- \\ & F_1^+ \cdot F_1^- F_1^- \cdot F_3^- F_3^- \cdot F_5^- F_5^- \cdot F_8^- F_8^- \\ & F_1^+ \cdot F_2^- F_2^- \cdot F_3^- F_3^- \cdot F_6^- F_6^- \cdot F_9^- F_9^- \cdot F_1^+. \end{aligned}$$

And corresponding to $x_{1,1}$ (i.e., u_1) being assigned TRUE, $L(u_1)$ and the two copies of F_1^+ not adjacent to it are deleted to have

$$\begin{aligned} H_1' = {} & F_1^+ \cdot F_1^+ \cdot L(u_2) \cdot L(u_3) \\ = {} & F_1^+ F_1^+ \cdot F_1^- F_1^- \cdot F_3^- F_3^- \cdot F_5^- F_5^- \cdot F_8^- F_8^- \\ & \cdot F_2^- F_2^- \cdot F_3^- F_3^- \cdot F_6^- F_6^- \cdot F_9^- F_9^-. \end{aligned}$$

The reverse direction is slightly more tricky. We first show the following lemma.

Lemma 1 *If H admits a feasible solution, then in H_i exactly two non-empty subsequences of $L(x_{i,1})$, $L(x_{i,2})$ and $L(x_{i,3})$ appear in the feasible solution H_i' (or, exactly one of the three is deleted from H_i).*

Proof By construction, the type-1 2-CNF clauses constructed over $x_{i,1}$, $x_{i,2}$ and $x_{i,3}$ in H_i have a special property due to the connection with 3-coloring: such a clause F_k^- appears consecutively in exactly two of the three substrings $L(x_{i,1})$, $L(x_{i,2})$ and $L(x_{i,3})$; moreover, if F_k^- appears in $L(x_{i,j})$ and $L(x_{i,l})$ then it does not appear in $L(x_{i,\ell})$, where $1 \le j \ne l \ne \ell \le 3$. If exactly one of the three non-empty subsequences of $L(x_{i,1})$, $L(x_{i,2})$ and $L(x_{i,3})$, say of $L(x_{i,1})$, appears in a feasible LDS solution H_i', then the type-1 2-CNF clause involving $x_{i,2}$ and $x_{i,3}$, say F_k^-, would be missing in H_i', contradicting the assumption that H (also H_i) has a feasible solution.

On the other hand, due to the construction of H_i, one cannot leave all the three non-empty subsequences of $L(x_{i,1})$, $L(x_{i,2})$ and $L(x_{i,3})$ in a feasible LDS solution H'. The reason is that F_i^+ would not be forming an LDS-block if all the three non-empty subsequences are kept in a feasible LDS solution.

It remains to show, in the feasible solution H_i', how to identify which subsequence comes from $L(x_{i,1})$, $L(x_{i,2})$ and $L(x_{i,3})$. This can be easily done by looking at the type-1 2-CNF clauses (which are always put at the beginning of these 3 lists). Following the example in Fig. 1, we just discuss one case when $L(u_1)$ is completely deleted, the other two are symmetric. When $L(u_1)$ is completely deleted (and two redundant copies of F_1^+ are also deleted), without further deletion, the complete form of H_1' must be $F_1^+F_1^+ \cdot F_1^-F_1^-F_3^-F_3^- \ldots F_2^-F_2^-F_3^-F_3^- \ldots$. Consequently, even if only one copy of $F_3^-F_3^-$ remains in a feasible H_1' we would still know that $F_1^+F_1^+$ is obtained by deleting $L(u_1)$ (or $L(x_{i,1}$ in the general case).

With Lemma 1, the reverse direction can be proved as follows. If the LDS-block $F_i^+F_i^+$ is formed by deleting $L(x_{i,j})$, then we assign $x_{i,j} \leftarrow$ TRUE (and the other two variables in F_i^+ are assigned FALSE). Clearly, this gives a valid truth assignment for ϕ. We thus have the following theorem.

Theorem 2 FT(4) *is NP-complete.*

Since FT(4) is NP-complete, the optimization problem LLDS+(4) is certainly NP-hard.

Corollary 1 *The optimization version of* LLDS+(4) *is NP-hard.*

3.2 Inapproximability results

The results in the previous subsection essentially implies that the optimization version of LLDS+(d), $d \geq 4$, does not admit any polynomial-time approximation algorithm (regardless of the approximation factor), since any such approximation would have to return a feasible solution. A natural direction to approach LLDS+ is to design a bicriteria approximation for LLDS+, where a factor-(α, β) bicriteria approximation algorithm is a polynomial-time algorithm which returns a solution of length at least OPT/α and includes at least N/β letters, where $N = |\Sigma|$ and OPT is the optimal solution value of LLDS+. We show that obtaining a bicriteria approximation algorithm for LLDS+ is no easier than approximating LLDS+ itself.

Theorem 3 *If* LLDS+(d), $d \geq 4$, *admitted a factor-*(α, $N^{1-\epsilon}$) *bicriteria approximation for any* $\epsilon < 1$, *then* LLDS+(d), $d \geq 6$, *would also admit a factor-*α *approximation, where N is the alphabet size.*

Proof Suppose that a factor-(α, $N^{1-\epsilon}$) bicriteria approximation algorithm $\mathcal{A}$ exists. We construct an instance S^* for *LLDS*+(4) as follows. (Recall that S is the sequence we constructed from a $(3^+, 1, 2^-)$-SAT instance ϕ in the proof of Theorem 2.) In addition to $\{F_i | i = 1..m\} \cup \{g_j | j = 1..n+1\}$ in the alphabet, we use a set of integers $\{1, 2, ..., (m+n+1)^x - (m+n+1)\}$, where x is some integer to be determined. Hence,

$$\Sigma = \{F_i | i = 1..m\} \cup \{g_j | j = 1..n+1\} \cup \{1, 2, ..., (m+n+1)^x - (m+n+1)\}.$$

We now construct S^* as

$$S^* = 1 \cdot 2 \ldots ((m+n+1)^x - (m+n+1)) \cdot S \cdot ((m+n+1)^x - (m+n+1)) \cdot ((m+n+1)^x - (m+n+1) - 1) \ldots 2 \cdot 1.$$

Clearly, any bicriteria approximation for S^* would return an approximate solution for S as including any number in $\{1, 2, ..., (m+n+1)^x - (m+n+1)\}$ would result in a solution of size only 2.

Notice that we have $N = m + (n+1) + (m+n+1)^x - (m+n+1) = (m+n+1)^x$. In this case, the fraction of letters in Σ that is used to form such an approximate solution satisfies

$$\frac{m+(n+1)}{(m+n+1)^x} \leq \frac{1}{N^{1-\epsilon}},$$

which means it suffices to choose $x \geq \lceil 2 - \epsilon \rceil = 2$.

4 Linear time algorithms for LLDS+(d) and FT(d) when $d = 3$

4.1 Solving the feasiblility testing version for $d = 3$

For the Feasibility Testing version, as covered earlier, Theorem 2 implies that the problem is NP-complete when $d \geq 4$. We next show that if $d = 3$, then the problem can be decided in linear time. For convenience, we also call an LD-block of length j a j-block. (We only focus on $j = 2, 3$ in the following.) We first prove that the solution has an implicit linear structure.

Lemma 2 *Given a string S over Σ such that each letter in S appears at most 3 times, if a feasible solution for* FT(3) *contains a 3-block then there is a feasible solution for* FT(3) *which only uses 2-blocks; moreover, for each letter a, its second occurrence $a^{(2)}$ must be in the solution.*

Proof Suppose that $S = \ldots a^{(1)} \ldots a^{(2)} \ldots a^{(3)} \ldots$, and $a^{(1)}a^{(2)}a^{(3)}$ is a 3-block in a feasible solution for FT(3). (Recall that the superscript only indicates the appearance order of letter a.) Then we could replace $a^{(1)}a^{(2)}a^{(3)}$ by either $a^{(1)}a^{(2)}$ or $a^{(2)}a^{(3)}$. The resulting solution is still a feasible solution for FT(3). In both cases, $a^{(2)}$ appears in the feasible solution.

Lemma 2 implies that the FT(3) problem can be solved in $O(n)$ time as follows. (Note that in the conference version [12], we did not have this lemma hence we can only solve it in $O(n^2)$ time using 2-SAT.) We first re-number the letters in S by their second (or, middle) occurrence (from left to right) as $c_1, c_2, ..., c_n$. Let $B_1(i) = c_i^{(1)}c_i^{(2)}$ and $B_2(i) = c_i^{(2)}c_i^{(3)}$. Let F be a feasible solution (which is a subsequence of S).

Staring from $i = 1$, we put $F := B_1(i) = c_i^{(1)}c_i^{(2)}$ as the first 2-block. Then we loop through $j = 2$ to n as follows:

- If F overlaps $B_1(j)$ and $B_2(j)$, then report 'no solution' and exit;
- If F overlaps $B_1(j)$, then $F \leftarrow F \cdot B_2(j)$; otherwise, $F \leftarrow F \cdot B_1(j)$.

By construction, the returned solution F is a feasible solution for FT(3) if the above algorithm does not exit with 'no solution'. The reason is that when F is returned by the algorithm, it already contains all the n 2-blocks.

Theorem 4 *Let S be a string of length n.* FT(3) *can be decided in $O(n)$ time.*

Theorem 4 immediately implies that LLDS+(3) has a factor -1.5 approximation as any feasible solution for $FT(3)$ would be a factor-1.5 approximation for LLDS+(3). In the following, we show that LLDS+(3) can in fact be solved in linear time as well, after $FT(3)$ is decided to have a solution.

4.2 The optimization version when $d = 3$

We now extend the solution in the previous subsection to solve LLDS+(3), where the input is a sequence S over Σ of size n; moreover, each letter appears exactly three times in S (i.e., $|S| = 3n$). (Note that this assumption is valid as if a letter only appears once then there is no feasible solution; and if a letter appears twice they must be put in a solution and all the other letters between them must be deleted.) Recall that we assume that there is always a feasible solution (which can be checked by Theorem 4).

Our idea is to use dynamic programming. Recall that the letters in S are re-numbered by their second (or, middle) occurrence (from left to right) as $c_1, c_2, \ldots, c_n$.

Following the previous lemma, LLDS+(3) solution has a natural structure ordered according to $c_i^{(2)}$'s, under the assumption that there is a feasible solution, which enables us to use dynamic programming to solve LLDS+(3).

For each character c_i, $1 \le i \le n$, let $B_1(i) := c_i^{(1)}c_i^{(2)}$, $B_2(i) := c_i^{(2)}c_i^{(3)}$, and $B_3(i) := c_i^{(1)}c_i^{(2)}c_i^{(3)}$. We define $D[i, j]$ as the maximum value of LDS for the sequence of characters $(c_1, \ldots, c_i)$ where the last LD-block containing c_i is $B_j(i)$, for $1 \le j \le 3$; otherwise, if $B_j(i)$ does not lead to a feasible solution, then $D[i, j] \leftarrow -\infty$. Similarly, define $D[i]$ as the maximum value of LDS for sequences of letters $(c_1, \ldots, c_i)$, with

$$D[i] = \max_{j=1..3} D[i, j].$$

The tables $D[-]$ and $D[-, -]$ can be calculated in linear time. In fact, for $D[i, j]$ with $1 \le j \le 3$, we have

$$D[i, j] = \max_{1 \le k \le 3} \begin{cases} D[i-1, k] + |B_j(i)| & \text{if } B_k(i-1) \text{ doesn't overlap } B_j(i), \\ -\infty & \text{if } B_k(i-1) \text{ overlaps } B_j(i). \end{cases}$$

Note that the value of $|B_j(i)|$ is either two or three. The optimal solution value for LLDS +(3) is $D[n]$. The actual solution can be easily retrieved in $O(n)$ time as well.

Theorem 5 *Given a string S of length n, where each letter appears exactly three times, the problem of LLDS+(3) can be solved in $O(n)$ time.*

Proof Assume that FT(3) already has a feasible solution (which can be checked by Theorem 4), an optimal solution for LLDS+(3) is obtained by the above dynamic programming algorithm. The correctness follows from the fact that the algorithm scans all (middle) letters from left to right and the final solution must include all the letters. The $O(n)$ running time is obvious: each $D[i, j]$ can be updated in $O(1)$ time, hence $D[i]$ too can be updated in $O(1)$ time; moreover, we only have an $O(n)$ number of such tables $D[i]$ and $D[i, j]$ to maintain and update.

As a simple example, let $S = 112132323$, then $D[1, 1] = 2$, $D[1, 2] = 2$, $D[1, 3] = 3$. Updating $i = 2$, we have $D[2, 1] = 4$, $D[2, 2] = 5$, $D[2, 3] = 5$. Finally, after updating $i = 3$, we have $D[3, 1] = -\infty$, $D[3, 2] = 6$, $D[3, 3] = -\infty$. The optimal solution 112233 is obtained according to $D[3] = 6$.

In the next section, we show that if the LD-blocks are arbitrarily positively weighted, then the problem can be solved in $O(n^2)$ time. Note that the $O(n)$ time algorithm in Sect. 2.1 assumes that the weight of any LD-block is its length, which has the property that $\ell(s) = \ell(s_1) + \ell(s_2)$, where $s = s_1 s_2$, s_1 and s_2 are LD-blocks on the same letter x, and $\ell(s)$ is the length of s (or the total number of letters of x in s_1 and s_2).

5 A dynamic programming algorithm for weighted-LDS

Given the input string $S = S[1...n]$, let $w_x(\ell)$ be the weight of LD-block $x^\ell, x \in \Sigma, 2 \leq \ell \leq d$, where d is the maximum number of times a letter appears in S. Here, the weight can be thought of as a positive function of x and ℓ and it does not even have to be increasing on ℓ. For example, it could be that $w(aaa) = w_a(3) = 8, w(aaaa) = w_a(4) = 5$. Given $w_x(\ell)$ for all $x \in \Sigma$ and ℓ, we aim to compute the maximum weight letter-duplicated string (Weighted-LDS) using dynamic programming.

Define $T(n)$ as the value of the optimal solution of $S[1...n]$ which contains the character $S[n]$. Define $w[i, j]$ as the maximum weight LD-block $S[j]^\ell$ ($\ell \geq 2$) starting at position i and ending at position j; if such an LD-block does not exist, then $w[i, j] = 0$. Notice that $S[j]^\ell$ does not necessarily have to contain $S[i]$ but it must contain $S[j]$. We have the following recurrence relation.

$$T(0) = 0,$$

$$T(i) = \max_{S[y] \neq S[i]} \begin{cases} T(y) + w[y+1, i] & if\ w[y+1, i] > 0, \\ 0 & otherwise. \end{cases}$$

The final solution value is $\max_n T(n)$. This algorithm clearly takes $O(n^2)$ time, assuming $w[i, j]$ is given. We compute the table $w[-, -]$ next.

1. For each pair of ℓ (bounded by d, the maximum number of times a letter appears in S) and letter x, compute

$$w'_x(\ell) = \max \begin{cases} w'_x(\ell - 1) \\ w_x(\ell) \end{cases},$$

with $w'_x(1) = w_x(1)$. This can be done in $O(d|\Sigma|) = O(n^2)$ time.

2. Compute the number of occurrence of $S[j]$ in the range of $[i, j]$, $N[i, j]$. Notice that $i \leq j$ and for the base case we have $S[0] = \varepsilon$.

$$N(0, 0) = 0,$$

$$N(0, j) = N(0, k) + 1, \quad k = \max \begin{cases} \{y | s[y] = s[j], 1 \leq y < j\} \\ 0 \end{cases}$$

And,

$$N(i, j) = \begin{cases} N(i-1, j), & if\ s[i-1] \neq s[j] \\ N(i-1, j) - 1, & if\ s[i-1] = [j] \end{cases}$$

This step takes $O(n^2)$ time. Note that, in this step, the number of times $S[j]$ appearing $[i, j]$ is computed based its appearance in $[i-1, j]$. Hence $N(i, j)$ is filled in a bottom-up manner.

Table 1 Input table for $w_x(\ell)$, with $S = ababbaca$ and $d = 4$

$x\backslash\ell$	1	2	3	4
a	5	10	20	15
b	4	16	8	3
c	1	3	5	7

Table 2 Table $w'_x(\ell)$, with $S = ababbaca$ and $d = 4$

$x\backslash\ell$	1	2	3	4
a	5	10	20	20
b	4	16	16	16
c	1	3	5	7

Table 3 Part of the table $N[i, j]$, with $S = ababbaca$ and $d = 4$

$i\backslash j$	1	2	3	4	5	6	7	8
8	0	0	0	0	0	0	0	1
...	...	...	...	...	...	...	...	...
3	0	0	1	1	2	2	1	3
2	0	1	1	2	3	2	1	3
1	1	1	2	2	3	3	1	4

3. Finally, we compute

$$w[i, j] = \begin{cases} w'_{s[j]}(N(i, j)), & if\ N(i, j) \geq 2 \\ 0, & else \end{cases}$$

This step also takes $O(n^2)$ time. We thus have the following theorem.

Theorem 6 *Let S be a string of length n over an alphabet Σ and d be the maximum number of times a letter appears in S. Given the weight function $w_x(\ell)$ for $x \in \Sigma$ and $\ell \leq d$, the maximum weight letter-duplicated subsequence (Weighted-LDS) of S can be computed in $O(n^2)$ time.*

We can run a simple example as follows. Let $S = ababbaca$. Suppose the table $w_x(\ell)$ is given as Table 1.

At the first step, $w'_x(\ell)$ is the maximum weight of an LD-block made with x and of length at most ℓ. The corresponding table $w'_x(\ell)$ can be computed as Table 2.

At the end of the second step, we have Table 3 computed.

From Table 3, the table $w[-, -]$ can be easily computed and we omit the details. For instance, $w[1, -] = [0, 0, 10, 16, 16, 20, 0, 20]$. With that, the optimal solution value can be computed as $T(8) = 36$, which corresponds to the optimal solution $aabbaa$.

6 Concluding remarks

Starting with the longest letter-duplicated subsequence problem (LLDS), which is polynomially solvable, we consider the constrained longest letter-duplicated subsequence (LLDS+) and the corresponding feasibility testing (FT) problems in this paper, where all letters in

Table 4 Summary of results on LLDS+ and FT

d	LLDS+(d)	FT(d)	Approximability of LLDS+(d)
$d \geq 4$	NP-hard	NP-complete	No approximation
$d = 3$	P	P	Not applicable

the alphabet must occur in the solutions. We parameterize the problems with d, which is the maximum number of times a letter appears in the input sequence. For convenience, we summarize the results one more time in the following table.

We also consider the weighted version (without the 'full-appearance' constraint), for which we give a non-trivial $O(n^2)$ time dynamic programming solution.

If we stick with the 'full-appearance' constraint, one direction is to consider an additional variant of the problem where the solution must be a subsequence of S, in the form of $x_1^{d_1} x_2^{d_2} \ldots x_k^{d_k}$ with x_i being a subsequence of S with length at least 2, $x_j \neq x_{j+1}$ and $d_i \geq 2$ for all i in $[k]$ and j in $[k-1]$. This was formally studied by Lafond et al. [10] recently. Intuitively, for many cases these variants could better capture the duplicated patterns in S. At this point, the NP-completeness results (similar to Theorem 2 and Corollary 1) would still hold with quite some modifications to the proofs in this paper. However, the running times of the dynamic programming algorithms there are much higher (in the range of $O(n^4)$ to $O(n^6)$), which could be a burden for real applications. Note that, without the 'full-appearance' constraint, when x_i is a subsequence of S, the problem is a generalization of Kosowski's longest square subsequence problem [8] and it is not surprising that it can be solved in polynomial time, even though the running time is much higher.

Acknowledgements This research is partially supported by National Natural Science Foundation of China under grant 61872427, 61732009 and 61628207. We thank anonymous reviewers for several useful comments.

Author Contributions WL, AL, BZ and PZ wrote the main manuscript text and BZ prepared Fig. 1. All author reviewed the manuscript.

Data availability No datasets were generated or analysed during the current study.

Declarations

Conflict of interest The authors declare no competing interests.

References

1. Asahiro, Y., Eto, H., Gong, M., Jansson, J., Lin, G., Miyano, E., Ono, H., Tanaka, S.: Approximation algorithms for the longest run subsequence problem. In: Bulteau, Liptá Z. (eds) 34th annual symposium

on combinatorial pattern matching, CPM 2023, June 26–28, 2023, Marne-la-Vallée, France, volume 259 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. **2**(1–2), 12. (2023)
2. Bovet, D.P., Varricchio, S.: On the regularity of languages on a binary alphabet generated by copying systems. Inf. Process. Lett. **44**(3), 119–123 (1992)
3. Cicalese, F., Pilati, N.: The tandem duplication distance problem is hard over bounded alphabets. In: Paola, F., Lucia M. (eds.) Combinatorial algorithms - 21st international workshop, IWOCA 2021, Ottawa, Canada, July 5–7, 2021, volume 12757 of Lecture notes in computer science, pp. 179–193. Springer (2021)
4. Ciriello, G., Miller, M.L., Aksoy, B.A., Senbabaoglu, Y., Schultz, N., Sander, C.: Emerging landscape of oncogenic signatures across human cancers. Nat. Genet. **45**, 1127–1133 (2013)
5. Dassow, J., Mitrana, V., Paun, G.: On the regularity of the duplication closure. Bull. EATCS **69**, 133–136 (1999)
6. Dondi, R., Sikora, F.: The longest run subsequence problem: further complexity results. In: Gawrychowski, P., Starikovskaya, T. (eds.) 32nd annual symposium on combinatorial pattern matching, CPM 2021, July 5–7, 2021, Wrocław, Poland, volume 191 of LIPIcs, pp. **14**(1–14), 15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
7. Ehrenfeucht, A., Rozenberg, G.: On regularity of languages generated by copying systems. Discret. Appl. Math. **8**(3), 313–317 (1984)
8. Kosowski, A.: An efficient algorithm for the longest tandem scattered subsequence problem. In: Apostolico, A., Melucci, M. (eds.) String processing and information retrieval, 11th international conference, SPIRE 2004, Padova, Italy, October 5–8, 2004, proceedings, volume 3246 of Lecture notes in computer science, pp. 93–100. Springer (2004)
9. Lafond, M., Zhu, B., Zou, P.: The tandem duplication distance is NP-hard. In: Paul, C., Bläser, M. (eds.) 37th international symposium on theoretical aspects of computer science, STACS 2020, March 10–13, 2020, Montpellier, France, volume 154 of LIPIcs, pp. **15**(1–15), 15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
10. Lafond, M., Lai, W., Liyanage, A., Zhu, B.: The longest subsequence-repeated subsequence problem. In: Wu, W., Guo, J. (eds.) Combinatorial optimization and applications, 17th international conference, COCOA 2023, Hawaii, HI, USA, December 15–17, 2023, proceedings, Part I, volume 14461 of Lecture Notes in Computer Science, pp. 446–458. Springer (2023)
11. Lafond, M., Zhu, B., Zou, P.: Computing the tandem duplication distance is NP-hard. SIAM J. Discret. Math. **36**(1), 64–91 (2022)
12. Lai, W., Liyanage, A., Zhu, B., Zou, P.: Beyond the longest letter-duplicated subsequence problem. In: Bannai, H., Holub, J. (eds.) 33rd annual symposium on combinatorial pattern matching, CPM 2022, June 27–29, 2022, Prague, Czech Republic volume 223 of LIPIcs, pp. **7**(1–7), 12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)
13. Lander, E.S., et al.: Initial sequencing and analysis of the human genome. Nature **409**(6822), 860–921 (2001)
14. Leech, J.: A problem on strings of beads. Math. Gaz. **41**(338), 277–278 (1957)
15. Macdonald, M.E., et al.: A novel gene containing a trinucleotide repeat that is expanded and unstable on Huntington's disease. Cell **72**(6), 971–983 (1993)
16. The Cancer Genome Atlas Research Network: Integrated genomic analyses of ovarian carcinoma. Nature **474**, 609–615 (2011)
17. Oesper, L., Ritz, A.M., Aerni, S.J., Drebin, R., Raphael, B.J.: Reconstructing cancer genomes from paired-end sequencing data. BMC Bioinf. **13**(Suppl 6), S10 (2012)
18. Schrinner, S., Goel, M., Wulfert, M., Spohr, P., Schneeberger, K., Klau, G.W.: The longest run subsequence problem. In: Kingsford, C., Pisanti, N. (eds.) 20th international workshop on algorithms in bioinformatics, WABI 2020, September 7–9, 2020, Pisa, Italy (virtual conference), volume 172 of LIPIcs, pp. **6**(1–6), 13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
19. Schrinner, S., Goel, M., Wulfert, M., Spohr, P., Schneeberger, K., Klau, G.W.: Using the longest run subsequence problem within homology-based scaffolding. Algorithms Mol. Biol. **16**(1), 11 (2021)
20. Sharp, A.J., Eichler, E.E., et al.: Segmental duplications and copy-number variation in the human genome. Am. J. Hum. Genet. **77**(1), 78–88 (2005)
21. Szostak, J.W., Ray, W.: Unequal crossing over in the ribosomal DNA of saccharomyces cerevisiae. Nature **284**, 426–430 (1980)
22. Wang, M.-W.: On the irregularity of the duplication closure. Bull. EATCS **70**, 162–163 (2000)
23. Zheng, C., Wall, P.K., Leebens-Mack, J., Pamphilis, C.D.E., Albert, V.A., Sankoff, D.: Gene loss under neighborhood selection following whole genome duplication and the reconstruction of the ancestral populus genome. J. Bioinform. Comput. Biol. **7**(03), 499–520 (2009)