BIOINFORMATICS

Vol. 00 no. 00 2014
Pages 1–5

HyDA-Vista: Towards Optimal Guided Selection of k-mer Size for Sequence Assembly

Seyed Basir Shariat Razavi [1,*], Narjes Sadat Movahedi Tabrizi [2], Hamidreza Chitsaz [1] and Christina Boucher [1*]

[1]Department of Computer Science, Colorado State University, Fort Collins, CO, USA
[2]Department of Computer Science , Wayne State University, Detroit, MI, USA.

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Associate Editor: XXXXXXX

ABSTRACT

Motivation: Intimately tied to assembly quality is the complexity of the de Bruijn graph built by the assembler. Thus, there have been many paradigms developed to decrease the complexity of the de Bruijn graph. One obvious combinatorial paradigm for this is to allow the value of k to vary; having a larger value of k where the graph is more complex and a smaller value of k where the graph would likely contain fewer spurious edges and vertices. One open problem that affects the practicality of this method is how to predict the value of k prior to building the de Bruijn graph. We show that optimal values of k can be predicted prior to assembly by using the information contained in a phylogenetically-close genome and therefore, help make the use of multiple values of k practical for genome assembly.

Results: We present HyDA-Vista, which is a genome assembler that uses homology information to choose a value of k for each read prior to the de Bruijn graph construction. The chosen k is optimal if there are no sequencing errors and the coverage is sufficient. Fundamental to our method is the construction of the *maximal sequence landscape*, which is a data structure that stores for each position in the input string, the largest repeated substring containing that position. In particular, we show the maximal sequence landscape can be constructed in $O(n+n\log n)$-time and $O(n)$-space. HyDA-Vista first constructs the maximal sequence landscape for a homologous genome. The reads are then aligned to this reference genome, and values of k are assigned to each read using the maximal sequence landscape and the alignments. Eventually, all the reads are assembled by an iterative de Bruijn graph construction method. Our results and comparison to other assemblers demonstrate that HyDA-Vista achieves the best assembly of *E. coli* before repeat resolution or scaffolding.

Availability: HyDA-Vista is freely available at `https://sites.google.com/site/hydavista`. The code for constructing the maximal sequence landscape and the choosing the optimal value of k for each read is also on the website and could be incorporated into any genome assembler.

Contact: basir@cs.colostate.edu

*Correspondence: basir@cs.colostate.edu

1 INTRODUCTION

The ability to accurately assemble genomes is a fundamental problem in bioinformatics that is vital to the success of many scientific projects, including the 10,000 vertebrate genomes (Genome 10K) [15], *Arabidopsis* variations (1001 genomes) [29], human variations (1000 genomes) [37], and Human Microbiome Project [38]. The genome assembly process aims to build contiguous sequences, called *contigs*, predominantly from short read sequencing data. Other sources of information have also been used to boost the accuracy, including genetic linkage data [23], optical mapping data [27], and longer sequencing reads (e.g. PacBio data) [16]. A potential source of information that has not been fully explored is the information contained in phylogenetically-close genomes. The genome of an individual of the same species or that of a phylogenetically-close species can potentially be used as an extra source of information, and increase the assembly quality. We argue that genome assemblers can benefit from using a reference genome to help guide the assembly process, particularly in those regions of the genome that are pervaded by repetitive sequences.

In Eulerian sequence assembly [17, 32], a de Bruijn graph is constructed with a vertex v for every $(k-1)$-mer present in an input set of reads, and an edge (v, v') for every observed k-mer in the reads with $(k-1)$-mer prefix v and $(k-1)$-mer suffix v'. A contig corresponds to a non-branching path through this graph. SPAdes [2], IDBA [30], Euler-SR [6], Velvet [41], SOAPdenovo [22], ABySS [36] and ALLPATHS [5] all use this paradigm for assembly. The majority of de Bruijn graph based assemblers follow the same general outline: break the (possibly error corrected) reads into k-mers, construct the de Bruijn graph on the set of resulting k-mers, simplify the de Bruijn graph, resolve the repeated regions by using mate-pair information, and construct the contigs (simple paths in the de Bruijn graph). Therefore, the majority of assemblers require or allow the value of k to be specified by the user.

The problem of determining an appropriate value of k for the de Bruijn graph construction is important since it has a direct impact on assembly quality; stated very briefly, when k is too small the resulting graph is complicated by spurious edges and vertices, and when k is too large the graph becomes too sparse and possibly disconnected. Repetitive regions are especially problematic for genome assembly since they inadvertently result in spurious edges and vertices in the de Bruijn graph [1] and are very sensitive to the choice of k. There has been a significant effort in developing

algorithms that will choose an ideal value for k by preprocessing the sequence reads, and thus, reduce the complexity of the de Bruijn graph [2, 30, 7].

A more obvious combinatorial approach for building a simplified de Bruijn graph would be to allow the value of k to vary; having a larger value of k where the graph is more complex and a smaller value of k where the graph would likely contain fewer spurious edges and vertices. A major difficulty in implementing this approach is determining a practical method that makes this idea feasible assembling large genomes. Peng *et al.* [30] and Bankevich *et al.* [2] both introduced assemblers that use various values of k. IDBA builds the de Bruijn graph in an iterative manner from $k = k_{min}$ to $k = k_{max}$; these values of k are predetermined and (by default) do not change for different datasets or genomes. At iteration i, the de Bruin graph for k_i is built from the current set of reads and the contigs for that graph are constructed, then all the reads that align to at least one of those contigs are removed from the current set of reads. In the next iteration the graph is built by converting every edge from the previous graph to a vertex while treating contigs as edges. SPAdes [2] uses a similar approach but uses all the reads at each iteration.

While this method has been shown to greatly improve assembly quality [2, 30], it is not efficient since all the reads are assembled at each iteration. Thus, one challenge that remains to be addressed is how to efficiently determine which values of k should be used for this iterative assembly process and how to assign a k-mer value for each read. If this could be accomplished prior to assembly of the de Bruijn graph(s) then these iterative assembly methods could be made more efficient without degrading the assemblies quality.

Our Contribution. We introduce an efficient algorithm for determining an optimal value of k for each read prior to constructing the de Bruijn graph, and implement this method into a modified version of HyDA, a *de novo* assembler developed by Movahedi *et al.* [26]. This modified assembler, which we refer to as *HyDA-Vista*, takes as input a phylogenetically-close genome and a set of paired-end reads. Imperative to HyDA-Vista is the construction of the *maximal sequence landscape*, which is a data structure that stores for every position in the input string, the longest repeat containing it. Prior to de Bruijn graph construction, HyDA-Vista constructs the maximal sequence landscape for the phylogenetically-close genome, and aligns the reads to the reference genome. The alignment and landscape allows the optimal value of k for each read be determined in linear time in the length of the read, provided the read is longer than the longest repeat. These values of k are "optimal" in the sense that for unchanged parts of the genome, the de Bruijn graph will have no spurious edges or vertices if there are no sequencing errors, and the length of the repeat is smaller than the read length. Unaligned reads are assigned a default value of k. After the assignment of values of k to each read, HyDA-Vista constructs the de Bruijn graph in an iterative manner.

Our approach for choosing values of k for each read takes into consideration the repeat structure of the genome, which enables us to avoid overly-complex regions of the graph since the assignment of values of k to reads is done prior to the assembly rather than during the assembly. We compare HyDA-Vista versus IDBA [30], SPAdes [2], SOAPdenovo [22], ABySS [36] and HyDA [26]. Our results demonstrate that HyDA-Vista produces the best assembly of *E. coli* before repeat resolution or scaffolding. We aim to achieve the best assembly without repeat resolution and scaffolding and note that such methods could be applied to all these initial assemblies. Lastly, we demonstrate that this method improves the efficiency of iterative assembly.

Roadmap. We review related tools for the problem in the remainder of this section. Section 2 then sets notation and formally lays down the problem and the data structures that will be used for the construction of the maximal sequence. We formally define the maximal sequence landscape in Section 3.1. Section 4.3 gives details of HyDA-Vista. In Section 5 we give results that demonstrate how HyDA-Vista compares against competing assemblers. Finally, Section 6 offers reflections and some future areas of research that warrant investigation.

Related Work. The re-sequencing methods include LOCAS and SUPERLOCAS [35, 19], e-RGA [39], Colombus module of Velvet [1] and IDBA-Hybrid [2]. Since these methods aim to determine structural variations between species, and require extremely high sequence similarity to produce reasonable results, they have only been applied to individuals of the same species. Our focus is to produce high quality *de novo* assemblies using homology information contained in the reference genome of the same species or phylogenetically-close species. Gnerre et al. [12] also consider how to improve assembly quality by using the alignment of reads to a reference genome. Their method simultaneously builds a *de novo* assembly from the reads and aligns these same reads to one or more related genomes. The alignment is then used to improve the assembly quality, e.g., reads that were not used in the assembly are incorporated into the assembly using the alignment.

Complementary to the work of Chikhi and Medvedev [7], Peng *et al.* [30], and Bankevich *et al.* [2], there has been an effort in developing methods that use paired-end data to constrain the construction of the de Bruijn graph [35, 31, 33, 25, 40]. Medvedev et al. [25] introduced the concept of a paired de Brujin graph . Since the insert size is variable among mate pairs, this method requires that all the paths within some threshold be considered in order to ensure an edge is not missed. Thus, Bankevich *et al.* [2] improve upon this idea by developing the rectangle graph, which eliminates the need to consider all paths. Vyahhi *et al.* [40] furthered this study of rectangle graphs for genome assembly. These methods merit mentioning the goal of these methods is the same as the goal of increasing the value of k in certain regions; both aim to minimize spurious edges and branching in the graph but in a different manner.

Determining all maximal exact repeats in a string has been previously studied [14, 9, 4]. It has been shown that all maximal repeats of a string can be found and stored in $O(n)$-time and $O(n)$-space using a suffix tree (although the output maybe of size $\Theta(n^2)$) or directed acrylic graph [9]. Therefore, the maximal sequence landscape, which we define in this paper, can be constructed from either a suffix tree or a directed acrylic graph in $O(n)$-time and $O(n)$-space using these algorithms directly or adapting them. However, the constant in the order notation of the space complexity of these constructions is relatively large. The algorithm we present

[1] Unpublished. `http://bioweb2.pasteur.fr/docs/velvet/Columbus_manual.pdf`

[2] Unpublished. `http://i.cs.hku.hk/~alse/hkubrg/projects/idba_hybrid/index.html`

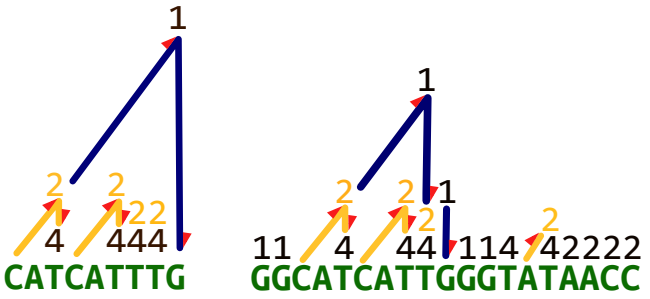


Fig. 1: (left) The self sequence landscape for CATCATTTG, and (right) the sequence landscape of GGCATCATTGGGTATAACC with respect to CATCATTTG. The mountains (yellow or blue) demonstrate occurrences of substrings of source string in the target string. Numbers at the peak of the mountains denote the frequency of occurrence. The maximal sequence landscape is highlighted in yellow, and the red arrows demonstrate the ascent and descent of the landscapes.

uses a suffix array and thus, requires linear space with a smaller constant and $O(n \log n)$-time. Hence, we pay a $\log n$ cost in time to remove the large constant from the linear space time. We also note that the related problem of finding inexact maximal repeats also has been previously studied [20, 11, 3, 34].

2 BACKGROUND

Strings. Consider a fixed alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$ and a total order $\leq_L$ defined over $\Sigma = \Sigma \cup \{\$\}$ where $\$ \notin \Sigma$, and for all $\sigma \in \Sigma$ we have $\$ \leq_L \sigma$. We denote a finite string s as $s_1 s_2 \ldots s_n$, where $s_i \in \Sigma$. We use s_{ij}, where $1 \leq i \leq j \leq n$, to indicate substring $s_i s_{i+1} \cdots s_j$ of string s. We call substrings $s_i^{\text{pre}} = s_{1i}$ and $s_i^{\text{suf}} = s_{in}$ with $i \in \{1, \ldots, n\}$ the i^{th} *prefix* and i^{th} *suffix* of s respectively. Based on the total order $\leq_L$, we define a *lexicographical* total order on the strings in Σ^*.

Arrays. We denote arrays of integers by all capital letter strings like A, SP, LCP, etc. $\mathsf{A}[i]$, with $1 \leq i \leq |\mathsf{A}|$, stands for the integer in the i^{th} cell of array A. Also, $\mathsf{A}[i, j]$ indicates the projection of A onto indices i to j, inclusive of both ends. For an array A, with $|\mathsf{A}| = n$, that holds a permutation of integers $\{1, \ldots, n\}$, *index array* of A is another array $\mathsf{I}(\mathsf{A})$ with $|\mathsf{I}(\mathsf{A})| = n$ such that $\mathsf{I}(\mathsf{A})[i] = j$ if and only if $\mathsf{A}[j] = i$.

Suffix and Longest Common Prefix Arrays. SA_s, for some string s, denotes the *suffix array* associated with s [24]. $\mathsf{SA}_s[i] = l$ for $i \in \{1, \ldots, n\}$ if and only if s_l^{suf} is the i^{th} string in the lexicographically sorted list of all suffixes of s. We also indicate the *longest common prefix array* of some string s with LCP_s, and $\mathsf{LCP}_s[i] = l$ for $i \in \{1, \ldots, n-1\}$ if and only if the length of the longest common prefix between $s^{\text{suf}}_{\mathsf{SA}_s[i]}$ and $s^{\text{suf}}_{\mathsf{SA}_s[i+1]}$ equals to l.

3 APPROACH

3.1 The Maximal Sequence Landscape

We formally define the maximal sequence landscape in this section. Clift *et al.* [9] introduced the concept of a sequence landscape, which is a data structure that stores the occurrences of any substring from a source string s in a target string t. In set representation, the sequence landscape $L_{t|s}$ of a target string t with respect to a source string s is defined as a set of tuples $\{m_1, m_2, \ldots, m_l\}$, where $m_i = (b_i, e_i, f_i)$ corresponds to the occurrence of substring $s_{b_i e_i} = s_{b_i} s_{b_i+1} \cdots s_{e_i}$ from s in t with frequency f_i. If s and t are equal then the sequence landscape categorizes all repeated substrings in the source string s. We define to this special case where $s = t$ as the *self sequence landscape*. Fig. 1 illustrates an example of a self sequence landscape and a sequence landscape. Given a position i of the input string s, all the repeated substrings containing s_i can be recovered from the self sequence landscape in linear-time in the number of different repetitions.

The occurrences of the substrings in the source string are defined as *mountains*. This terminology reflects the visual representation that was first introduced by Clift *et al.* [9] that illustrates each occurrence as a mountain having height equal to the length of the substring, i.e. the height of mountain m_i of $L_{t|s}$ is denoted as $h(m_i)$ and equal to $e_i - b_i + 1$. The *peak* of each mountain is labelled with the frequency of the substring corresponding to it. In Fig. 1 (left), the substring CAT is represented as two mountains each of which has a height equal to three and frequency equal to two.

We say that a *mountain* $m_j = (b_j, e_j, f_j)$ in a landscape $L_{t|s}$ *covers* index i and denote it by $m_j \triangle i$ if and only if $i \in \{b_j, \ldots, e_j\}$. Hence, the *cover set* of a specific index i of the sequence landscape $L_{t|s}$ is the set of all the mountains that covers i. We denote the cover set as $C_{L_{t|s}}(i)$ and define it as follows:

$$C_{L_{t|s}}(i) := \{m_j | m_j \triangle i, h(m_j) > 1, f_j > 1\}. \quad (1)$$

Lastly, we define the *summit of index* i as the highest mountains in its cover set. We denote the summit of i by $S_{L_{t|s}}(i)$ and define it as follows:

$$S_{L_{t|s}}(i) := \{m_j \,|\, h(m_j) \geq h(m_k)\ \forall\, m_k \in C_{L_{t|s}}(i)\}. \quad (2)$$

Please note that the summit of index i can be empty or non-unique so the height of summit of index i is defined to be zero for empty set.

DEFINITION 1. *The* maximal sequence landscape*, which we denote as $L^*_{t|s}$, is the set of the summits of all positions in s that have frequency greater than one. $L^*_{t|s}$ can be formally defined as follows: $L^*_{t|s} = \{S_{L_{t|s}}(i) | i = 1, \ldots, n\}$. The maximal sequence landscape is highlighted in yellow in Fig. 1.*

The maximal sequence landscape is obtained from the sequence landscape by removing all mountains except those that are highest and have frequency greater than one at each position. In the case of the maximal sequence landscape constructed from a self sequence landscape, this results in a data structure containing the longest repeat at each position of the input string. In Subsection 4.1 we give an algorithm that builds the maximal sequence landscape and returns an array containing the length of the longest repeat at each position of the input string (MSL in Algorithm 2). Therefore, given a position i in s, we can determine the length of the longest repeat

in s containing that position in constant time by simply indexing the maximal sequence landscape at position i. By choosing a value for k that is larger than the length of this repeat it can guaranteed that there will be no branching in the corresponding vertices of the de Bruijn graph, if the same substring is not repeated in changed parts of the genome that is being assembled. This is our idea based on which we determine the optimal value of k for each read. We consider the maximal sequence landscape constructed from the self sequence landscape for the remainder of this paper since it is what is used by HyDA-Vista.

4 METHODS

Algorithm 1 gives an overview of HyDA-Vista algorithm. We explain each of these steps in detail in the subsequent subsections.

Algorithm 1 An overview of HyDA-Vista

1: Build the maximal sequence landscape for the reference genome.
2: Align all reads to the reference using BWA.
3: For each aligned read: assign a value of k using the maximal sequence landscape.
4: Unaligned reads are assigned a value of k using a heuristic.
5: The de Bruijn graph is constructed in an iterative manner, as shown in Algorithm 3.

4.1 Construction of the Maximal Sequence Landscape

In this section we demonstrate that the maximal sequence landscape for an input string s can be built in $O(n + n \log n)$-time and $O(n)$-space using a very simple algorithm, where n is the length of s. Algorithm 2 gives the pseudocode for constructing the maximal sequence landscape. Our method relies on the use of suffix array and longest common prefix array and thus, begins by building the suffix array (SA_s), and the longest common prefix array (LCP_s). This construction can be done in $O(n)$-space and $O(n)$-time [18]. Two other auxiliary data structures are constructed at the beginning of the algorithm. However, we delay the definition and construction of those to later in this section. The algorithm then iterates through each position of s and finds the longest repeated substring in s that contains it using SA_s, LCP_s, and the auxiliary data structures. An important aspect of our algorithm that allows us to achieve $O(n + n \log n)$-time is that we only search that interval of SA_s which is between the indices $\mathsf{SA}_{\min}$ and $\mathsf{SA}_{\max}$ at each iteration, not the entire array. In other words this invariant holds at each iteration of our algorithm: $[\mathsf{SA}_{\min}, \mathsf{SA}_{\max}]$ holds the the interval in the suffix array that corresponding suffixes share a same prefix. This prefix is the longest repeat that has been seen so far and covers that position. Thus, each time the largest repeated substring is found for a particular position, the maximal sequence landscape (MSL), $\mathsf{SA}_{\min}$, and $\mathsf{SA}_{\max}$ are updated for search at the next iteration. Two possibilities exist at each iteration i of the algorithm when we are processing s_i;

(a) The longest repeated substring at position $i-1$ can be extended by appending s_i. The maximal sequence landscape, $\mathsf{SA}_{\min}$, and $\mathsf{SA}_{\max}$ are updated.

(b) The longest repeated substring at position $i-1$ *cannot* be extended by appending s_i (either the extended string does not occur or it does occur but its frequency is one). Let $p = s_j \cdots s_{i-1}$ be the longest repeated substring yet found that contains s_{i-1}, and $p' = s_{j+1} \cdots s_i$ be the string obtained by removing the first letter of p and appending s_i. If p''s frequency of occurrence is greater than one, then the maximal sequence landscape, and the search interval is updated as in (a). Otherwise, the search for the longest repeated substring continues by eliminating the first character of p' each time until a repeating match is found or the null string is reached. If the null string is reached then the maximal sequence landscape is empty at that position and the search interval is updated to $[1, n]$.

The search interval contains all indices in SA_s for which the corresponding suffixes have the current longest repeated substring as a prefix. In (a), the interval is updated by performing binary search. In (b) the search interval is no longer valid since we removed a letter from the *beginning* of the current longest repeated substring and we need (a more complicated) scheme to efficiently find the correct search interval. To accomplish this we need two auxiliary data structures that are constructed at the beginning of the algorithm: the SP_s array, and an ordered binary search tree containing all consecutive intervals of LCP_s. $\mathsf{SP}_s[j]$ holds the index in SA_s that is obtained by removing the first letter from the beginning of $s_j s_{j+1} \cdots s_n$ in order to obtain $s_{j+1} \cdots s_n$. This array can be built in linear time by scanning the index array of SA_s (see background for definition of index array). Table 1 illustrates the construction of SP_s in linear time. Thus, to find the correct interval in (b), we locate an index of SA_s (denoted as sp) where the corresponding suffix contains $s_{j+1} \ldots s_{i-1}$ as a prefix, and find the largest interval around sp where all the suffixes in the interval have $s_{j+1} \ldots s_{i-1}$ as prefix. This is the new search interval. The sp index can be found in constant time by correctly indexing SP_s (see the appendix). The second step is equivalent to finding the largest interval $[d, u]$ around sp that for all $j \in [d, u]$ we have $\mathsf{LCP}_s[j] \geq |p'| - 1$. Corollary 2 shows that this can be done in $O(\log n)$-time and $O(n)$-space using an ordered binary search tree. We leave the proof of the corollary, and the construction of the ordered binary search tree to the appendix.

COROLLARY 2. *Given a string p, SP_s, LCP_s, and an ordered binary search tree of all consecutive intervals in LCP_s, the largest interval of SA_s that contains all the suffixes of s that share the same prefix of p can be found in $O(\log n)$-time and $O(n)$-space.*

Theorem 3 gives the space and time complexity of our construction algorithm. See the appendix for the proof of this result. Informally, the time complexity can be seen by first noting that at each iteration of the algorithm the maximal sequence landscape either ascends by one after a number of descents (possible zero) or it is undefined after a number of nonzero descents, and each of these ascents or descents require $O(\log n)$-time. Note that in (a) the maximal sequence landscape is ascending, and in (b) the maximal sequence landscape is descending, and the frequency of a substring in s can be determined in constant time using LCP_s. Second, since each time it ascends one character from s is processed and the number of ascents equals the number of descents, the total number of ascents and descents is $2n$. Therefore, since the data structures are constructed in $O(n)$-time, and since there are at most $2n$ ascents or descents which take $O(\log n)$-time, the running time of the algorithm is $O(n + n \log n)$.

THEOREM 3. *The maximal sequence landscape of string s of size n can be built in $O(n + n \log n)$-time and $O(n)$-space.*

4.2 Assignment of k-mer Sizes to Reads

We assign values of k to the reads using the maximal sequence landscape constructed for the reference genome by first aligning the reads to the reference genome using BWA (version 0.7.4) [21] in paired-end mode. We consider all forward and reverse alignments of every read. Let p be the position in the reference genome where a read of length ℓ aligns, and let k^* be the maximum of $\{\mathsf{MSL}[p] + 1, \mathsf{MSL}[p+1] + 1, \ldots, \mathsf{MSL}[p + \ell] + 1\}$, where MSL is the maximal sequence landscape array that contains the height of the maximal sequence landscape at each position. We compute k^* for each forward alignment and let K^* be the set of all these values. The optimal

Algorithm 2 Maximal sequence landscape construction

Require: String s of length n.
Ensure: Maximum Sequence Landscape Array MSL.

```
MSL ← ∅
Build SA_s, LCP_s, SP_s, and the ordered binary search tree of the LCP_s
b ← 0, e ← 0                                          ▷ b and e denote positions in the string s
[sa_min, sa_max] ← [1, n]
for i := 1 to n do
    e ← e + 1, p ← s_b s_{b+1} ··· s_e
    if p ∈ SA_s and has frequency greater than one then
        Update [sa_min, sa_max]                        ▷ A repeat has been found
    else
        while b ≠ e and no repeat is found do
            b ← b + 1, p = s_b s_{b+1} ··· s_e
            Find the new interval [sa_min, sa_max]     ▷ See Section 4.1
            if p ∈ SA_s and has frequency greater than one then
                Update [sa_min, sa_max]                ▷ A repeat has been found
            end if
        end while
    end if
    Update MSL[i]
end for
```

Table 1. Construction of the suffix pointer array (SP_s) using the suffix array. The dark column in each table denotes the index of the array. We build an inverse index $I(\mathsf{SA}_s)$ from the suffix array, and SP_s can be constructed by scanning this array once, i.e. setting $\mathsf{SP}_s[I(\mathsf{SA}_s)[i-1]] = I(\mathsf{SA}_s)[i]$ for all $i \in [2, n]$ and $\mathsf{SP}_s[1] = 0$.

Suffix Start		
1	1	mybananas$
2	2	ybananas$
3	3	bananas$
4	4	ananas$
5	5	nanas$
6	6	anas$
7	7	nas$
8	8	as$
9	9	s$
10	10	$

⟹

Suffix & LCPArrays			
1	10	0	$
2	4	0	ananas$
3	6	3	anas$
4	8	1	as$
5	3	0	bananas$
6	1	0	mybananas$
7	5	0	nanas$
8	7	2	nas$
9	9	0	s$
10	2	0	ybananas$

⟹

Suffix Array Index	
1	6
2	10
3	5
4	2
5	7
6	3
7	8
8	4
9	9
10	1

⟹

SP	
1	0
2	7
3	8
4	9
5	2
6	10
7	3
8	4
9	1
10	5

value of k for the (forward) read is equal to the maximum value in K^*. We follow the same procedure for the reverse alignments with the exception that we compute the reverse complement of the read. Thus, the optimal values of k can be computed in linear time in the length of the read.

If the computed k-mer size (maximum of all maximal sequence landscape heights of all aligned nucleotides) is larger than the read length, then a default value ($k = 77$ is the default) is used instead. Unaligned reads are also assigned a default k-mer value ($k = 55$ is the default).

4.3 The de Bruijn Graphs

Let $R = \{r_1, \ldots, r_N\}$ denote the set of reads. We also denote the k-mer size assigned to r_i in the previous section by $K(r_i)$. In the first step of constructing the assembly de Bruijn graphs, we partition R into $R_k := \{r \in R \mid K(r) = k\}$, in which k ranges from $k_{\min} = \min_{r \in R} K(r)$ to $k_{\max} = \max_{r \in R} K(r)$. The HyDA-Vista assembly procedure, shown in Algorithm 3, iteratively builds de Bruijn graphs $G_{k_{\min}}, \ldots, G_{k_{\max}}$ with $k = k_{\min}, \ldots, k_{\max}$ respectively and obtains $\mathcal{A}_{k_{\min}}, \ldots, \mathcal{A}_{k_{\max}}$ assembly contig sequences after iterative graph condensation and error removal. Each G_k is constructed from the reads whose assigned k-mer size is not more than k and the contigs resulting from G_{k-1} constructed in the previous iteration,

$$\bigcup_{j=k_{\min}}^{k} R_j \cup \mathcal{A}_{k-1}. \tag{3}$$

The rationale behind this idea is that those reads that have an assigned k-mer size not more than k should ideally not create any repeats when they are assembled with the k-mer size k. The iterative inclusion of contigs from previous rounds, first introduced in IDBA [30] and later adopted by others [2], is an idea that has already shown merit in improving assembly quality. In Algorithm 3, HyDA is a function that accepts a set of input sequences and an integer k, and returns a set of contigs which are obtained from assembling the input sequences with a k de Bruijn graph.

Algorithm 3 Construction of the de Bruijn graphs

function HYDA-VISTA(R, K)
  $k_{\min} \leftarrow \min_{r \in R} K(r)$, $k_{\max} \leftarrow \max_{r \in R} K(r)$
  for all $k_{\min} \leq k \leq k_{\max}$ **do**
    $R_k \leftarrow \emptyset$
  end for
  for all $r \in R$ **do**
    $k \leftarrow K(r)$
    $R_k \leftarrow R_k \cup \{r\}$
  end for
  $R' \leftarrow \emptyset$
  $\mathcal{A}_{k_{\min}-1} \leftarrow \emptyset$ ▷ assembly contigs
  for $k := k_{\min}$ **to** $k_{\max}$ **do**
    $R' \leftarrow R' \cup R_k$
    $\mathcal{A}_k \leftarrow$ HYDA($R' \cup \mathcal{A}_{k-1}, k$) ▷ contigs resulting from assembly with HyDA
  end for
  return $\mathcal{A}_{k_{\max}}$
end function

5 RESULTS

5.1 Improved Efficiency Due to Maximal Sequence Landscape

HyDA-Vista uses the maximal landscape to break the reads into groups by assigning each a value of k. It then uses these groups to build the graph iteratively. This is in contrast to other methods that also iteratively build of the graph; SPAdes [2] uses *all* the reads at *each* iteration, and IDBA [30] uses a more complicated approach to remove some subset of reads at each iteration. Thus, one of the main advantages of using the maximal sequence landscape is that it increases the efficiency of building the assembly graph iteratively without degrading assembly quality (see the next subsection for a comparison of the different assemblers). To demonstrate this efficiency experimentally we ran HyDA-Vista with and without the maximal sequence landscape on multicell *E. coli* (substr. K-12) Illumina data and the *E. coli* (substr. K-12) reference genome. See Subsection 5.2 for a description of this dataset. Without the maximal sequence landscape the assembly took 1,414 minutes, and with the maximal sequence landscape the assembly took 822 minutes with 42 number of minutes for building the maximal sequence landscape and assigning the values of k to the reads.

5.2 Comparison Between Competing Assemblers and HyDA-Vista on *E.Coli*

The first data set consists of approximately 27 million paired-end 100 bp reads from multicell *E. coli* (substr. K-12), generated by the Illumina Genome Analayzer (GA) IIx platform. It was obtained from the NCBI Short Read Archive (accession ERA000206, EMBL-EBI Sequence Read Archive). To assess assembly quality, we aligned the reads to the *E. coli* reference genome (substr. K-12) using BWA (version 0.5.9) [21] with default parameters. We call a read *mapped* if BWA outputs an alignment for it and *unmapped* otherwise. Analysis of the alignments revealed that 98% of the reads mapped to the reference genome, representing an average depth of approximately $600\times$; An analysis using BLAST against known contaminants revealed that the unmapped reads are attributed to minor contamination of the sample [8]. All reads were error corrected using BayesHammer [28] with default parameters.

KmerGenie [7] predicted 41 to be the optimal k-mer value for this dataset. Therefore, for the assemblers that require a single value of k to be specified (SOAPdenovo, ABySS, HyDA) we used $k = 41$. HyDA and HyDA-Vista were ran with a cut off of five. All other parameters of SOAPdenovo and ABySS were kept at their default. SPAdes and IDBA were run with their default parameters in single-end mode. Since the input reads were corrected prior to assembly, the reported data for SPAdes is from the "only assembly" stage. IDBA was run with and without error correction, yielding the same statistics as expected.

Table 2 gives the standard assembly statistics of all the assemblies. All statistics in Table 2 were computed by QUAST in default mode [13]. The results demonstrate that HyDA-Vista achieves the best assembly prior to repeat resolution or scaffolding. Note that upon determination of k-mer sizes, all assemblers were run in single-end mode, i.e., ignoring the pairing information, to study only the effect of our k assignment on contigs. HyDA provides a skeletal de Bruijn graph implementation on which other technologies can be developed. HyDA alone does not compete with some of the state of the art assemblers such as SPAdes and IDBA that employ multiple sophisticated technologies. However, empowered by only the maximum landscape information without any other sophisticated technology particularly pairing information, HyDA-Vista increases the N50 and NG50 more than twice (in comparison to HyDA) and outperforms the competing assemblers in all measures (the NG50 of 36 kbps obtained by HyDA increases to 82 kbps by HyDA-Vista).

6 CONCLUSION

We demonstrated that HyDA-Vista achieves superior performance with respect to standard assembly statistics for ecoli genome before repeat resolution and scaffolding. A crucial aspect of our method is the construction of the maximal sequence landscape for the phylogenetically-close genome, which allows for the optimal k-mer value to be computed for each read. The maximal sequence landscape requires that a pair of substrings be an exact match in order for them to be considered to be the same repetition. An area that warrants future investigation is determining if there is an efficient algorithm for computing the maximal sequence landscape for inexact matches, i.e., the maximal sequence landscape with the condition that two substrings x and y are considered to represent the same repetition (mountain) if the Hamming distance or edit distance between the two is at most d, where d is a parameter in the problem. Another open problem is determining whether the maximal sequence landscape could be constructed with only a suffix array. Since all the data structures (including the suffix array) are constructed in $O(n)$-time and $O(n)$-space the order notation of the running time of such a maximal sequence landscape construction algorithm would likely not improve the running time of the existing algorithm by more than a small constant factor. However, the removal of the auxiliary data structures may simplify the algorithm and would be of theoretical interest.

Table 2. The performance comparison between major assembly tools and HyDA-Vista on the error-corrected standard multicell *E. coli* dataset (6.2 Gbps, 28 million reads, 100bp, treated as single-end) using QUAST in default mode [13]. All statistics are based on contigs no shorter than 500 bp. Since there are not (QUAST-defined) misassemblies in any of the assemblies, the length statistics are based on *correct* contigs. NGA50 (NA50) is a (QUAST-corrected) contig size the contigs larger than which cover half of the *genome* (assembly) size [13, 10]. Total is sum of the length of all contigs. MA is the number of misassemblies. GF is the genome fraction percentage, which is the fraction of genome bases that are covered by the assembly. Unaligned is the total length of all of the contigs that could not be aligned to the reference.

Assembler	NGA50	NA50	Largest (bp)	Total (bp)	MA	GF (%)	Unaligned (bp)
SOAPdenovo	32,032	35,343	101,201	4,304,232	3	95.2	3,421
ABySS	31,237	32,987	110,012	4,530,701	0	97.56	0
SPAdes	60,338	60,768	173,976	4,545,775	0	97.8	3,001
IDBA	57,826	58,549	173,964	4,538,426	0	97.7	2,349
HyDA	36,292	39,069	123,771	4,524,075	0	97.4	0
HyDA-Vista	**82,838**	**94,910**	**204,602**	**4,544,286**	**0**	**97.9**	**0**

ACKNOWLEDGEMENT

The authors would like to thank Ross McConnell from Colorado State University for many insightful discussions and suggestions.

Funding: SBSB and CB were funded by the Colorado Clinical and Translational Sciences Institute which is funded by National Institutes of Health (NIH-NCATS,UL1TR001082, TL1TR001081, KL2TR001080).

REFERENCES

[1]C. Alkan, S. Sajjadian, and E. E. Eichler. Limitations of next-generation genome sequence assembly. *Nature Methods*, 8(1):61–65, 2011.

[2]A. Bankevich et al. SPAdes: A new genome assembly algorithm and its applications to single-cell sequencing. *J. of Comp. Bio.*, 19(5):455–477, 2012.

[3]G. Benson. A space efficient algorithm for finding the best nonoverlapping alignment score. *Theoretical Computer Science*, 145(1):357–369, 1995.

[4]A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, and J. Seiferas. The Smallest Automation Recognizing the Subwords of a Text. *Theor Comput Sci*, 40:31–55, 1985.

[5]J. Butler et al. ALLPATHS: *De Novo* assembly of whole-genome shotgun microreads. *Genome Res.*, 18(5):810–820, 2008.

[6]M. J. Chaisson and P. A. Pevzner. Short read fragment assembly of bacterial genomes. *Genome Res.*, 18(2):324–330, 2008.

[7]R. Chikhi and P. Medvedev. Informed and automated k-mer size selection for genome assembly. *Bioinformatics*, 30(1):31–37, 2014.

[8]H. Chitsaz et al. Efficient *de novo* assembly of single-cell bacterial genomes from short-read data sets. *Nature Biotech*, 29(10):915–921, 2011.

[9]B. Clift, D. Haussler, R. M. McConnell, T. D. Schneider, and G. D. Stormo. Sequence landscapes. *Nucleic Acids Res.*, 14:141–158, 1986.

[10]D. Earl et al. Assemblathon 1: a competitive assessment of de novo short read assembly methods. *Genome Res.*, 21(12):2224–2241, 2011.

[11]W. M. Fitch, T. Smith, and J. L. Breslow. Detecting internally repeated sequences and inferring the history of duplication. *Methods in enzymology*, 128:773–788, 1985.

[12]S. Gnerre, E. S. Lander, K. Lindblad-Toh, and D. B. Jaffe. Assisted assembly: how to improve a de novo genome assembly by using related species. *Genome Biol*, 10(8):R88, 2009.

[13]A. Gurevich, V. Saveliev, N. Vyahhi, and G. Tesler. QUAST: quality assessment tool for genome assemblies. *Bioinformatics*, 29(8):1072–1075, 2013.

[14]D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[15]D. Haussler et al. Genome 10K: a proposal to obtain whole-genome sequence for 10,000 vertebrate species. *J. Hered.*, 100(6):659–674, 2009.

[16]J. Huddleston et al. Reconstructing complex regions of genomes using long-read sequencing technology. *Genome Res.*, 24(4):688–696, 2014.

[17]R.M. Idury and M.S. Waterman. A new algorithm for dna sequence assembly. *J. of Comp. Bio.*, 2(2):291–306, 1995.

[18]J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear word suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[19]J. D. Klein et al. LOCAS–a low coverage assembly tool for resequencing projects. *PLOS ONE*, 6(8):e23455, 2011.

[20]S. Kurtz, J. V Choudhuri, E. Ohlebusch, Chris Schleiermacher, J. Stoye, and R. Giegerich. Reputer: the manifold applications of repeat analysis on a genomic scale. *Nucleic acids research*, 29(22):4633–4642, 2001.

[21]H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[22]R. Li et al. *De novo* assembly of human genomes with massively parallel short read sequencing. *Genome Res.*, 20(2):265–272, 2010.

[23]H.C. Lin et al. AGORA: Assembly Guided by Optical Restriction Alignment. *BMC Bioinformatics*, 13(1):189, 2012.

[24]U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[25]P. Medvedev, S. Pham, M. Chaisson, G. Tesler, and P. A. Pevzner. Paired de bruijn graphs: A novel approach for incorporating mate pair information into genome assemblers. *J of Comp Bio*, 18(11):1625–1634, 2011.

[26]N. S. Movahedi, E. Forouzmand, and H. Chitsaz. *De novo* co-assembly of bacterial genomes from multiple single cells. In *IEEE Conference on Bioinformatics and Biomedicine*, pp. 561–565, 2012.

[27]N. Nagarajan, T. D. Read, and M. Pop. Scaffolding and validation of bacterial genome assemblies using optical restriction maps. *Bioinformatics*, 24(10):1229–1235, 2008.

[28]S.I. Nikolenko, A.I. Korobeynikov, and M.A. Alekseyev. Bayeshammer: Bayesian clustering for error correction in single-cell sequencing. *BMC Genomics*, 14(Suppl 1):S7, 2013.

[29]S. Ossowski et al. Sequencing of natural strains of *Arabidopsis thaliana* with short reads. *Genome Res.*, 18(12):2024–2033, 2008.

[30]Y. Peng, H.C.M. Leung, S.M. Yiu, and F.Y.L. Chin. IDBA – a practical iterative de Bruijn graph *de novo* assembler. In *Research in Computational Molecular Biology*, volume 6044 of *Lecture Notes in Computer Science*, pp. 426–440, 2010.

[31]P. A. Pevzner, H. Tang, and G. Tesler. De novo Repeat Classification and Fragment Assembly. *Genome Res*, 14(9):1786–1796, 2004.

[32]P. A. Pevzner, H. Tang, and M. S. Waterman. An eulerian path approach to dna fragment assembly. *PNAS*, 98(17):9748–9753, 2001.

[33]A. M. Phillippy, M. C. Schatz, and M. Pop. Genome assembly forensics: Finding the elusive mis-assembly. *Genome Biol*, 9(3):R55, 2008.

[34]M. Sagot. Spelling approximate repeated or common motifs using a suffix tree. In *LATIN'98: Theoretical Informatics*, pp. 374–390. Springer, 1998.

[35]K. Schneeberger et al. Reference-guided assembly of four diverse *Arabidopsis thaliana* genomes. *Proc. Natl. Acad. Sci. U.S.A.*, 108(25):10249–10254, 2011.

[36]J. T. Simpson et al. ABySS: A parallel assembler for short read sequence data. *Genome Res.*, 19(6):1117–1123, 2009.

[37]The 1000 Genomes Project Consortium. An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491(7422):56–65, 2012.

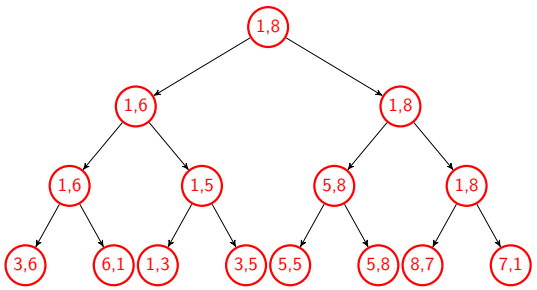
[38]P. J. Turnbaugh et al. The human microbiome project: exploring the microbial part of ourselves in a changing world. *Nature*, 449(7164):804–810, 2007.

[39]Francesco Vezzi, Federica Cattonaro, and Alberto Policriti. e-RGA: enhanced reference guided assembly of complex genomes. *EMBnet. journal*, 17(1):pp–46, 2011.

[40]Nikolay Vyahhi, Alex Pyshkin, Son Pham, and Pavel A. Pevzner. From de bruijn graphs to rectangle graphs for genome assembly. In *Algorithms in Bioinformatics*, volume 7534 of *Lecture Notes in Computer Science*, pp. 249–261, 2012.

[41]R. Zerbino and E. Birney. Velvet: Algorithms for *de novo* short read assembly using de bruijn graphs. *Genome Res.*, 18(5):821–829, 2008.

Fig. 2: An ordered binary tree of all the consecutive intervals the array $\mathsf{L} = [3, 6, 1, 3, 5, 5, 8, 7, 1]$. This data structure is used to efficiently solve the LARGEST INTERVAL PROBLEM. See the proof of Theorem 5.



z

APPENDIX

We now give the definition and construction for the ordered binary tree used in the maximal sequence landscape construction algorithm. Given an integer array L, we build an ordered binary search tree for all consecutive pairs of integers in the list. We create a leaf node for each pair of integers, and create a set of parent nodes at each level of the tree. Each parent node contains the interval $[min, max]$ that corresponds to the largest integral range containing the labels of the child nodes. See Figure 2 for an illustration of this. This is a complete binary tree with $n-1$ leaves so the total number of nodes is $2(n-1)-1$. Hence, the construction of the tree takes $O(n)$-time and $O(n)$-space. Lastly, we note that the height of the tree is $O(\log n)$.

Next, we show that solving the LARGEST INTERVAL PROBLEM for a given list L can be done in $O(\log n)$-time and $O(n)$-space using an ordered binary search tree of all the consecutive intervals in L.

DEFINITION 4. *Given an array of integers L of size n, index $i \in [1, n]$, and threshold α, the* LARGEST INTERVAL PROBLEM *aims to find the largest interval $[a, b] \subseteq [1, n]$ such that $i \in [a, b]$ and $\mathsf{L}[k] \geq \alpha$ for all $k \in [a, b]$.*

We now prove the following theorem.

THEOREM 5. *Given an ordered binary tree of all the consecutive intervals in L, the* LARGEST INTERVAL PROBLEM *can be solved in $O(\log n)$-time and $O(n)$-space.*

PROOF. For a given index i and threshold α, we use the ordered binary search tree to find the largest interval $[a, b]$ around i, where $\mathsf{L}[j] \geq \alpha$ for all $j \in [a, b]$. First, we assume that there exists two leaves containing $\mathsf{L}[i]$. Let ℓ_{i1} and ℓ_{i2} be the leaves that have the minimum and the maximum integer, respectively. In order to find a, we traverse the tree from ℓ_{i1} to the root until we reach the root or a node that has a label with property $\alpha \notin [min, max]$, and then traverse down tree taking lefthand edges to a leaf node is reached. The minimum of this leaf node is a. Next, we can find the value of b using the same approach, with the exception that we start the traversal from ℓ_{i2}, and traverse up the tree while $alpha \notin [min, max]$. Then we traverse down the tree taking the righthand edges until a leaf node is reached. The maximum of at this leaf node is b.

When there exists only a single leaf node containing $\mathsf{L}[i]$ then we are the beginning or the end of the list and only a or b needs to be found. Therefore, at most four traversals from a leaf to the root of the tree are needed, and since the height of tree is $O(\log n)$, the total running time of the algorithm is $O(\log n)$-time. □

Corollary 2 follows directly from Theorem 5.

The results in the main body of the paper were focused on the construction of the maximal sequence landscape from the *self* sequence landscape. Here, we will prove a stronger result that states the maximal sequence landscape for the sequence landscape constructed for s and t can be constructed in $O(n + m \log n)$-time and $O(m)$-space, where n is the length of s and m is the length of t. First, we prove an intermediate result concerning the *silhouette* of a sequence landscape, which is the maximal sequence landscape without any constraint on frequency of each mountain. We remind the reader that each mountain in the maximal sequence landscape has frequency greater than one which is not the case for the silhouette.

LEMMA 6. *The silhouette of the sequence landscape $\mathsf{L}_{t|s}$ of string t with respect to string s can be built in $O(n + m \log n)$-time and $O(n)$-space, where n is the length of s and m is the length of t.*

PROOF. We present a constructive proof. Given s, we construct SA_s and LCP_s. This construction can be done in $O(n)$-time and $O(n)$-space [18]. Next, we define another auxiliary array of s, denoted as SP_s, that is used to efficiently traverse SA_s in a specific order. $\mathsf{SP}_s[i]$ contains the index of the suffix in SA_s that is obtained by removing first letter from $\mathsf{SA}_s[i]$. We note that SP_s can be computed in $O(n)$-time and $O(n)$-space by scanning the SA_s once. Table 1 illustrates this construction of SP_s. SP_s can be interpreted as an array that saves a pointer to the next entry in Burrow's Wheeler transform of s.

Given these arrays, we process t from one letter at a time. We let b_i and e_i be indices of the string t, i.e., $b_i \leq t_i$, and $b_i \in [1, n]$ and $e_i \in [1, n]$. Lastly, we denote $t[b_i, e_i]$ as the current substring of t that we are processing at the i'th iteration of the algorithm. We note that b_0 and e_0 are initialized to zero at the beginning of the algorithm. At iteration i of algorithm, we are trying to match the largest substring of t containing the i'th character with a substring in s. In order to accomplish this, we need to search for the substring $t[b_i, e_i] = t[b_{i-1}, e_{i-1}]t_i$ in SA_s. Since we have already matched first $e_{i-1} - b_{i-1}$ characters of $t[b_i, e_i]$ with some entry in $\mathsf{SA}_s(t[b_{i-1}, e_{i-1}])$ (possibly none), and the starting and ending index for binary search in SA_s is already saved, this can be done in $O(\log n)$. To see this we must consider two cases that arise when processing $t[b_i, e_i]$ at iteration i.

(a) If $t[b_i, e_i]$ is contained in SA_s then the silhouette is updated, e_i is incremented, and the starting and ending indices of the search interval for SA_s are updated for next iteration. The search interval can be updated using binary search on SA_s.

(b) If $t[b_i, e_i]$ is *not* contained in SA_s, then one letter at a time is eliminated from the beginning of $t[b_{i-1}, e_{i-1}]e_i$ until it is found in SA_s, or the null string is reached. We let p' denote the string obtained from eliminating the first character from the current string. The search interval is no longer valid for p' since we removed the first character, and therefore, we need to efficiently find the search interval for SA_s. To find the correct search interval, we locate the index of the suffix in SA_s that has p' as a prefix, we denote this index as sp, and find the interval around sp where each suffix in this interval has p' as a prefix. This is the new search interval. The index sp can be found in constant time using SP_s since $sp = \mathsf{SP}_s[\mathsf{SA}_s[t[b_{i-1}, e_{i-1}]]]$, and it follows from Theorem 5 that the interval can be found in $O(\log n)$-time.

In (a) the silhouette is ascending and in (b) the silhouette is descending. Since each time the silhouette ascends we process a letter from t, and the number of times the silhouette descends is equal to the number of times it ascends, the number of ascents or descents is $O(m)$. Since each ascent or descent requires $O(\log n)$-time the algorithm requires at most $O(m \log n)$-time after the construction of the data structures. All data structures require $O(n)$-time for construction, and $O(n)$-space. Thus, the algorithm requires $O(n + m \log n)$-time and $O(n)$-space. □

THEOREM 7. *The maximal sequence landscape of string t of size m with respect to string s of size n can be built in $O(n + m \log n)$-time and $O(n)$-space.*

PROOF. The construction of the maximal sequence landscape is identical to the construction of the silhouette with the addition that the frequency of each mountain has to be checked in order to determine if it is greater than one. Determining whether the frequency is greater than one can be accomplished in constant time using the LCP_s, i.e., if $\mathsf{LCP}_s[\mathsf{SA}_s[t[b_i, e_i]] + 1]$ or $\mathsf{LCP}_s[\mathsf{SA}_s[t[b_i, e_i]] - 1]$ is greater than or equal to $e_i - b_i$ then we can conclude $t[b_i, e_i]$ has frequency greater than one at iteration i of the algorithm. Hence, this construction takes $O(m+\log n)$-time and $O(n)$-space by Lemma 6. □