INSTITUT FÜR INFORMATIK
Algorithmische Bioinformatik

Universitätsstr. 1 D–40225 Düsseldorf

Heinrich Heine
Universität
Düsseldorf

On a String Partitioning Problem

Michael Wulfert

Bachelorarbeit

Beginn der Arbeit:	01. Oktober 2019
Abgabe der Arbeit:	30. Dezember 2019
Gutachter:	Univ.-Prof. Dr. Gunnar Klau
	Jun.-Prof. Dr. Dorothea Baumeister

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 30. Dezember 2019

Michael Wulfert

Contents

1 Introduction

1.1 Biological question

Due to advances in sequencing techniques, especially the development of next generation sequencing methods, it is possible to get several gigabases of sequence information in short time. Most of these techniques create a great number (up to millions) of short DNA sequence reads (about 100 bp, Levy and Myers, 2016). Using bioinformatic methods, overlapping reads are combined to longer sequences. Such a combined continuous sequence, which can reach millions of letters of the genetic code, is called a contig (Staden, 1980).

Ideally, each contig of a species sequenced represents a whole chromosome, but in reality contigs are often much shorter, due to algorithmic restrictions, e.g. repetitive sequences which are abundant but difficult to analyze.

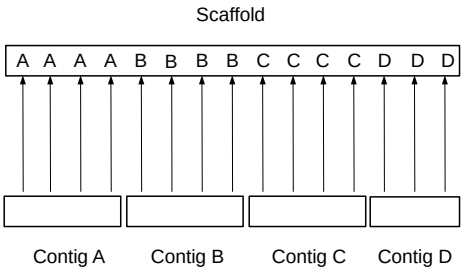

Figure 1.1: Ideally, contigs can be aligned perfectly to a known scaffold.

Meanwhile, the whole genomic sequence of many species is known. In this case, the sequencing results of an other member of such a species can be easily aligned to a known reference sequence as template or scaffold (fig. 1.1).

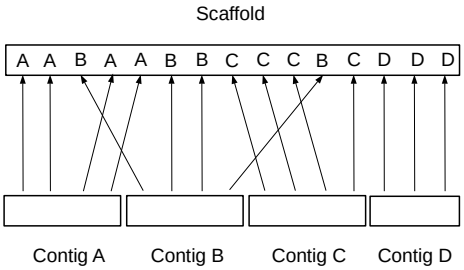

Figure 1.2: In reality, some parts of contigs didn't fit well into the known scaffold.

If any reference sequence is known for a species, their contigs might be compared to the

known sequence of an other, related species. Most of the genetic information will stay in the same order (synteny), but some parts may be copied or deleted or they or their repeats have changed their position in the same chromosome or even were translocated to another chromosome (fig. 1.2).

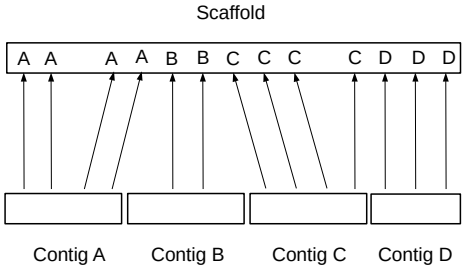

Figure 1.3: By removing some parts from the scaffold, the match between both sequences can be optimized.

In this case, the best overlap between both sequencing results is achieved by removing some not-fitting parts of the contigs (fig. 1.3) or even changing the order of the contigs. Ideally, there are as many parts left as possible.

Therefore, an algorithm is necessary which removes as few as possible letters from an input sequence to get a sequence where every letter occurs in only one block to find the best alignment between the two species compared [Manish Goel, personal communication].

1.2 Algorithmic problem

According to the GROUPING BY SWAPPING problem mentioned in "Computers and Intractability" (Garey and Johnson, 1979), this string partitioning problem can be formulated as the following GROUPING BY REMOVING problem:

INSTANCE: Finite alphabet Σ, string $S \in \Sigma^*$.

QUESTION: How to find the longest string S' converted from S by removing some letters from S so that in S' all occurrences of each letter $\lambda \in \Sigma$ are in a single block?

What is the complexity of this problem?

Example 1.1 GROUPING BY REMOVING

Input: S: = AABAABBBCCCBCDD over Σ = {A, B, C, D}

Result: S' = AA_AABBBCCC_CDD is gained from S by removing only two letters.

In the course of this thesis, sequences of identical letters will be summed up to blocks with their lengths as weights, but even rational nonnegative numbers could be used as weights:

EXPANSION: Every letter $\lambda \in \Sigma$ may have a weight w. Which result sequence S' has the largest sum of remaining letter weights?

2 Combinatorial Complexity

The number of result strings S′ which can be derived from an input string S by a GROUPING BY REMOVING algorithm is a combinatorial question with two different aspects:

2.1 Aspect one: Use every letter

To get all possible Strings S' from S, where every letter of the alphabet Σ occurs at least once, can be done by the following algorithm:

Select one of each type of letters in the string S and tag them. From the begin of the string S to the first tagged letter, keep all letters of this type and remove all other letters. From the first to the second tagged letter in the string, keep all of the second type letters and remove the others. Continue in this way until the last letter.

Let n_σ be the number of letters of type $\sigma \in \Sigma$ in S. Then there are n_A possibilities to select the letter A, n_B possibilities to select the letter B and so on.

In total, there are

$$P_{el} = \prod_{\sigma \in \Sigma} n_\sigma \tag{2.1}$$

possibilities to select one of every letter of the alphabet Σ in S, thus there are $\prod n_\sigma$ possibilities to get a String S′ from S where every letter occurs in exactly one continuous block (el for 'every letter').

Example 2.1 *selecting a string S′ from S*

S = A<u>BA</u>CB<u>C</u>A (selected letters are underlined) gives

S′ = –<u>BA</u>C–<u>C</u>–

It is possible to construct a string S where every possible String S′ has the same length: Take blocks of letters where every letter occurs only once, each followed by a unique letter (Example 2.2).

Example 2.2 *All solutions have the same length*

S	=	*ABC*	*X*	*BCA*	
S′	=	*ABC*	*X*	*– – –*	*(1.)*
or		*AB–*	*X*	*–C–*	*(2.)*
or		*A–C*	*X*	*B– –*	*(3.)*
or		*A– –*	*X*	*BC–*	*(4.)*
or		*–BC*	*X*	*– –A*	*(5.)*
or		*–B–*	*X*	*–CA*	*(6.)*
or		*– –C*	*X*	*B–A*	*(7.)*
or		*– – –*	*X*	*BCA*	*(8.)*

$$P_{el} = n_A \cdot n_B \cdot n_C \cdot n_X = 2 \cdot 2 \cdot 2 \cdot 1 = \quad 8$$

In example 2.2, every possible result string S′ from an input string S of this type has to be tested for maximal length.

2.2 Aspect two: Letters may be omitted

It may be necessary to omit a letter of the alphabet Σ to get a String S′ of maximal length: In S = DDEDD, the longest substring is S′ = DD–DD and not DDE– – or – –EDD, where every letter is used. Therefore, every letter in Σ has one possibility more: to be omitted.

This leads to

$$P_{mc} = \prod_{\sigma \in \Sigma} (n_\sigma + 1) \tag{2.2}$$

possibilities to get a string S′ from S where not every letter of the alphabet Σ has to occur (mc for 'main constraint').

In this case, even the empty string ε which has length zero is a valid solution for S′, so not all substrings S′ can have the same length.

2.3 Suboptimal solutions

Obvious, it is possible that choosing one letter of every type in a string may lead to suboptimal solutions. Depending on the String S, the last tagged letter may occur also after the tag, and also between two tagged letters, the first letter may occur before the second. In all this cases, the resulting string S can't have maximal length.

Example 2.3 *incomplete solutions*

S	=	*AB*AB*C* *(tagged letters underlined)*
S'	=	*AB*– –*C*,
but		*AB*–B*C* *is longer.*

This ignored letters can't be found generally, because their existence depends on the string. In strings like example 2.2 there are any of such ignored letters.

Therefore, basically all $P_{mc} = \prod_{\sigma}(n_\sigma + 1)$ possibilities have to be tested to get the longest substrings. But some possibilities can be ignored.

This leads to the chapter of algorithmic reductions (see chapter 4) after proving the NP completeness of the GROUPING BY REMOVING problem (see chapter 3).

3 NP Completeness

To prove that the GROUPING BY REMOVING problem of this thesis is NP complete, it is to show that it is in NP and is NP hard. The first is shown by an algorithm which tests a solution in deterministic polynomial time P, the second is shown by a reduction of a known NP hard problem to the GROUPING BY REMOVING problem.

3.1 In NP: Test of a solution in P

A given result string S' of length n' from an input String S of length n over an alphabet Σ can be validated in two steps.

First, it is to prove that the original input string S contains the result string S'. This can be done in linear time ($n + n'$ steps): Starting with the first letters in S and S'respectively, compare the actual letter in S with the actual letter in S'. If they are not equal, compare the next letter of S with the actual letter of S', if they are equal, compare the next letter of S with the next letter of S', until the end of the strings. If the end of S' is reached in this way, S contains S'.

Second, it is to prove that the result string S' fits to the main condition that every letter should occur in only one adjacent block. This can be done in $n' \frac{1}{2}|\Sigma|(|\Sigma|-1)$ steps: Add the first letter to a list of used letters. Then read S' letter by letter (n' steps), and if the letters change, compare the new letter with the list of already used letters (up to $\frac{1}{2}|\Sigma|(|\Sigma| - 1)$ comparisons with the list). If it already exists in this list, S' didn't fit to the main constraint. Otherwise, add the new letter to the list of used letters and continue until the end of the result string S'.

In total, because of $n' \leq n$ and $|\Sigma| \leq n$, there are up to

$$(n + n') + n' \frac{1}{2}|\Sigma|(|\Sigma| - 1) \in \mathcal{O}(2n + n|\Sigma|^2) = \mathcal{O}(n|\Sigma|^2) \subset \mathcal{O}(n^3) \subset P \tag{3.1}$$

steps necessary to test a solution, which is in P.

3.2 NP hardness: Reduction of the linear ordering problem

The NP hardness of this problem can be shown by a polynomial-time many-one reduction of a known NP hard problem to this string problem.

3.2.1 Linear Ordering Problem

The LINEAR ORDERING PROBLEM (LOP) is a known NP hard problem (Grötschel et al., 1984):

Given a complete digraph $D_n = (V, R_n)$ with arc weights w for all arcs in R_n, find an acyclic sub digraph (V, T) in D_n such that the sum of all remaining arc weights is as large as possible, where $T \subset R_n$ contains exactly one of the two arcs of every edge in R_n:

$$maximise \quad v(T) := \sum_{AB \in T} w(AB) \tag{3.2}$$

3.2.2 Method: Choose exactly one of several equal letters

If it is necessary to choose exactly one of two blocks of letters λ of length a and b, respectively, by the longest substring algorithm, this can be done by separating the letters λ by at least one unique separator letter \$ which has a higher value L'': $L'' > \max(a, b)$.

$$\ldots \lambda^a \ldots \$^{L''} \ldots \lambda^b \ldots, \quad \text{choose} \quad L'' > \max(a, b) \tag{3.3}$$

Then, the separator letter \$ has to be taken to get the longest possible result string because omitting the separator letter \$ to combine the two letters λ will always lead to a worse solution and therefore only one of the equal letters λ can be taken.

3.2.3 Construction of a string with weighted letters

Given a digraph $D_n = (V, R_n)$ of n nodes V and a complete set R of arcs with weights w, where the weights of the arcs of the same edge may be different ($w(AB) \neq w(BA)$ for $A, B \in R_n$ possible). The edge Λ between the nodes A and B is represented by the letter Λ_{AB}, and Λ_{BA} means the same letter in a string but in a different notation which keeps the following formulas clearer:

$$\Lambda_{AB} = \Lambda_{BA} \tag{3.4}$$

The values of the two arcs representing this edge may have different weights.

Step 1: Choose one arc of every edge

To choose just one arc of every edge of the digraph D with the GROUPING BY REMOVING algorithm, the letters Λ representing outgoing arcs from one vertex can be combined to blocks and the blocks of letters representing outgoing arcs from different vertices can be separated by unique separator letters $\$_X$ with a sufficiently high value L''.

$$\Lambda_{AB}^{w(AB)} \; \Lambda_{AC}^{w(AC)} \ldots \$_A^{L''} \Lambda_{BA}^{w(BA)} \; \Lambda_{BC}^{w(BC)} \ldots \$_B^{L''} \ldots \tag{3.5}$$

(*Here, Λ_{AB} and Λ_{BA} are the same letter, each representing different arcs of the same edge.*)

Step 2: Double letters

Next step is to double every letter Λ. This enables to add additional constraint letters Δ between them. Therefore, the values of the letters are added to a constant L which has to be greater than the greatest weight if any arc weight is negative, or even greater than three times the maximum of the absolute weight of every arc:

$$L > 3 \max_{V,W \in R}(|w(VW)|) \tag{3.6}$$

This has two effects: First, every letter value is positive and can be used in a GROUPING BY REMOVING algorithm and second, a pair of letters $\Lambda^{L-m}\Lambda^{L-m}$ with maximal negative weight $-m$ have always a greater value than one single letter Λ^{L+m} with maximal positive weight m: $L > 3m \Rightarrow 2(L-m) > L+m$.

In this way, a single letter Λ is always smaller than a doubled letter, even with a different value. This assures, that only arcs are taken where all intermediate letters Δ are rejected. Likewise, to get the possible maximum of arcs, any edge can be omitted.

$$\begin{array}{ll} \Lambda_{AB}^{L+w(AB)}\Delta\Delta\Lambda_{AB}^{L+w(AB)} & \Lambda_{AC}^{L+w(AC)}\Delta\Delta\Lambda_{AC}^{L+w(AC)}\ldots\$_A^{L''} \quad \textit{continued in next line} \\ \Lambda_{BA}^{L+w(BA)}\Delta\Delta\Lambda_{BA}^{L+w(BA)} & \Lambda_{CA}^{L+w(CA)}\Delta\Delta\Lambda_{CA}^{L+w(CA)}\ldots\$_B^{L''}\ldots \end{array} \tag{3.7}$$

Step 3: Avoid circular triangles

In order to avoid circles and to get an acyclic digraph, from every oriented triangle Δ_{ABC} formed by the arcs between the vertices A, B and C at least one side has to be avoided. This can be done by using the additional letters Δ between the equal letters Λ of each edge, one triangle letter Δ for every of the n-2 possible triangles with this edge Λ. If the value L' of these letters $\Delta^{L'}$ is larger than the largest weight of any Λ, $L' \geq \frac{4}{3}L$, one of each letter $\Delta^{L'}$ has to be taken and interrupts it's edge letters.

To clarify the formulas, every oriented triangle letter Δ may have three notations for the same letter:

$$\Delta_{ABC} = \Delta_{BCA} = \Delta_{CAB} \;\neq\; \Delta_{CBA} = \Delta_{BAC} = \Delta_{ACB} \tag{3.8}$$

This results in a string with $n-2$ letters Δ between each pair of arc letters Λ:

$$\begin{array}{ll} \Lambda_{AB}^{L+w(AB)}\Delta_{ABC}^{L'}\Delta_{ABD}^{L'}\ldots\Lambda_{AB}^{L+w(AB)} & \Lambda_{AC}^{L+w(AC)}\Delta_{ACB}^{L'}\Delta_{ACD}^{L'}\ldots\Lambda_{AC}^{L+w(AC)}\ldots\$_A^{L''} \\ \Lambda_{BA}^{L+w(BA)}\Delta_{BAC}^{L'}\Delta_{BAD}^{L'}\ldots\Lambda_{BA}^{L+w(BA)} & \Lambda_{CA}^{L+w(CA)}\Delta_{CAB}^{L'}\Delta_{CAD}^{L'}\ldots\Lambda_{CA}^{L+w(CA)}\ldots\$_A^{L''}\ldots \end{array} \tag{3.9}$$

(Here $\Lambda_{AB} = \Lambda_{BA}$ and $\Delta_{ABC} = \Delta_{CAB} \neq \Delta_{ACB} = \Delta_{BAC}$ etc.)

Step 4: Any circles

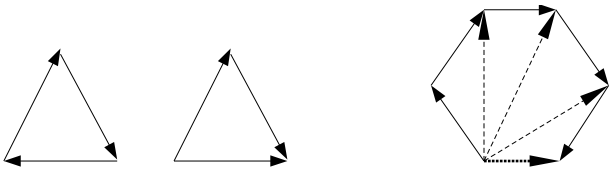

Figure 3.1: Left: cyclic and acyclic triangle.
Right: In an complete acyclic digraph, every path (continuous arrows) can be reduced step by step to one arc (dotted arrow).

If every possible triangle has three arcs but any possible triangle has a circle of arcs, then in every triangle two arcs are consecutive and the third arc has the same start and end vertices as the consecutive arcs and therefore could replace the other two arcs in a path (fig. 3.1).

If there is a circle with $k > 3$ vertices in the digraph, each two consecutive arcs can be replaces by the third arc of this triangle and form a circle with $k-1$ vertices. Recursively, such a circle can be reduced to a triangle with circular arcs ($k = 3$). Therefore, if there is any triangle with circular arcs, there couldn't be any circle in the digraph and the digraph is acyclic.

Step 5: Count remaining letters in the result string

In the result string, the remaining arcs form a complete acyclic digraph.

In an acyclic digraph with n vertices, there is one vertex with $n-1$ outgoing and no ingoing arcs, one vertex with $n-2$ outgoing arcs and one ingoing arc, and so on until the last vertex, which has no outgoing but $n-1$ ingoing arcs.

Therefore, there will remain different numbers of edge-letter pairs Λ between the separating letters \$ in the result string. The number of letter pairs remaining in an block represents the order of the corresponding vertex of the outgoing arcs.

3.2.4 Summary of the construction

- Take exactly one arc of every edge by introducing separator letters $\$_X$ between both arcs of every edge. If the values L'' of the separator letters $\$_X$ are sufficient high, every separator letter has to be taken and didn't alter the solution (step 1).
- Ensure to take the maximum of arcs possible by adding a sufficient high constant L to every arc weight. Because the possible maximum of $\frac{1}{2}n(n-1)$ arcs between n vertices is taken, this didn't alter the solution, too (step 2).
- Avoid circular triangles by introducing separator letters Δ with sufficient high value L'. The value L' of these separator letters Δ also has to be large enough so that at least one of every different Δ_{XYZ} has to be taken. Because the same Δ

are separated by larger \$, only one of each Δ is taken and didn't alter the solution weight (step 3).

- This avoids circles (step 4) and results in a complete acyclic digraph (step 5), where the solution depends only on the arc weights $w(XV)$.

3.2.5 Transforming in polynomial time

If the digraph has n vertices, it has $\frac{1}{2}n(n-1)$ edges and this string has two pairs of letters Λ for every edge, each pair separated by $n-2$ Δ, and $n-1$ separators \$ for the n blocks, in total:

$$\frac{1}{2}n(n-1)\cdot 2\cdot(2+(n-2))+(n-1)=(n^2+1)(n-1) \quad \text{letters} \tag{3.10}$$

Therefore, this string can be constructed in polynomial time $\mathcal{O}(n^3)$

The value L'' of each separation letter \$ has to be larger than the value L' of every triangle letter Δ and even larger as the value $2(L+m)$ of the pair of the largest edge letters Λ. If the additional weights are large enough, $L'' \gg L' \gg L$, one of each separator letter has to be taken to get the longest result string:

$$\begin{aligned} L &> \max_{V,W\in R}|w(VW)| \\ L' &\geq \tfrac{4}{3}L \\ L'' &> 2L' \end{aligned} \tag{3.11}$$

If an algorithm is used where at least one of every letter of the alphabet must occur in the result string, the separator letters \$ and Δ don't need any value.

3.3 Many-one reduction

In this way, the linear ordering problem can be solved by transforming the underlying digraph in a string in polynomial time and finding the result string by removing some of the letters. This shows that the GROUPING BY REMOVING problem described in this thesis is also NP hard and because it is in NP (see chapter 3.1) it is NP complete.

[This many-one reduction was proposed by Sven Schrinner, personal communication.]

4 Constraints and Reductions

4.1 Constraints

There are 2^n possibilities to remove letters from a string of n letters. A result string S' which fits to the main constraint that each of the remaining letters should occur only in one coherent block has still $P_{mc} = \prod_{\sigma \in \Sigma}(n_\sigma + 1)$ possibilities (see equation 2.2).

A minor constraint may be, that every letter should exist in the result string (see equation 2.1).

There might be more than one longest string which fits to the constraint(s). Some algorithms are able to report all possible result strings.

4.2 Reductions

Reductions in this and the following chapters means reduction of complexity in the sense of reduction of numbers of possible solutions to be tested for their lengths, not the transformation of one problem into an other like in the last chapter. Depending on the string, there are several possibilities to reduce the complexity of the GROUPING BY REMOVING problem.

[To test the effects of the following reductions, among other parameters, their usage in the programmed algorithms can be specified in a parameter file: `Parameter.prm`]

4.2.1 Reductions of the input string

It might be possible to reduce the complexity of the input string by combining adjacent identical letters to blocks, by separating the input string in independent substrings and by handling adjacent unique letters or blocks of unique letters together.

Combine adjacent equal letters to blocks

To get the longest possible result string, a stretch of identical letters should be handled as one unit because a part of it will always be shorter than the whole stretch. The equations 2.1 and 2.2 can be taken for m blocks instead of n letters, with $m \leq n$. [Reduction 1: `blockIdenticalLetters`]

Separate independent substrings

If a string can be divided into substrings which didn't share any letters, these substrings can be handled independently. Each substring has less letters and a smaller subalphabet than the whole string, which reduces the number of possible result strings to test for their lengths. Especially the reduction of the alphabet may have a great impact on the possibilities left because the magnitude of the alphabets gives the number of multiplications in the equations 2.1 and 2.2. Occurrences of letters which are interlaced with each other can't be divided into substrings. [Reduction 2: `independentSublists`]

Combine adjacent unique (blocks of) letters

If a letter occurs only once in a string or only in one block, it is called a unique letter or unique block. The conjunction of every pair of adjacent unique blocks cut in halve the remaining possibilities, because they have to be taken together or rejected together to get the longest possible result string. Taking only one of them is always shorter than taking both but divides the remaining parts of the string in the same way. This reduction is specified in each algorithm to ensure that every block represents only one type of letters. [Reduction 6: `uniquesTogether`]

4.2.2 Any skipped letters

To get result strings as long as possible, it should be avoided to skip letters which fit into the result string (see example 4.1). [Reduction 0 and Reduction 5: `takeOnlyNewLetters`]

Example 4.1 *solutions too short*

S	=	*AABABB*	
S'	=	*AA– –BB*	*fulfill the main constraint, but*
		AAB–BB	*or*
		AA–ABB	*is longer.*

If only one letter of the alphabet is left to keep: Take all of them. [Reduction 8: `takeLastLetter`]

If all of the remaining letters have to be removed, the obtained result string can be tested for length immediately. [Reduction 7: `noMoreLetters`]

4.2.3 Discard short solutions

The longest result string has the lowest number of letters removed. By searching for the longest string which fits to the main constraint, all intermediate solutions can be discarded which can't reach the length of an already known solution anymore.

Some algorithms can consider the lengths of the already found strings:

Depending on the algorithm, there are letters or blocks of letters already removed [Reduction 3: `breakIfTooSmall`] and those which have to be removed to fulfill the main constraint because its letters where already used. [Reduction 4: `tooManyToReject`] Furthermore, if there are more blocks of letters left than different letters exist in the remaining string, $m' > |\Sigma'|$, at least $|\Sigma'|$ blocks can be kept, but at least half of the remaining $m' - |\Sigma'|$ blocks have to be removed:

$$r' = \lceil \tfrac{1}{2}(m' - |\Sigma'|) \rceil \tag{4.1}$$

The lengths of the shortest r' remaining blocks may also be subtracted in the calculation of the already reachable result string length. [Reduction 9: `minToReject`]

A lower limit for the length of the result string is the sum of the lengths of the longest block of each type of letter of the alphabet. Only one block of each letter always fulfills the main constraint.

If not all letters have to be used, the number of the most frequent letter in the string may be a higher lower limit, especially if the number n of letters is much higher than the magnitude of the alphabet $|\Sigma|$.

If the GROUPING BY REMOVING problem is formulated as a decision problem, i.e. the question, whether a result string of at least length $K \in \mathbb{N}$ exists or not, this constant K could be used as lower limit in the algorithms searching for a result string. If one result string with at least K letters is found, the answer is yes, if any string of this length can be found, the answer is no.

5 Algorithms

There are several possibilities to solve the GROUPING BY REMOVING problem and to get a result string S' of single-block-letters from an input string S by removing some letters. The common main constraint in all algorithms is, that all different letters occur only in one coherent block in the result string S' each, a minor constraint may be that all letters of the alphabet have to exist in the result string.

A further point is, whether all possible solutions should be reported or only one of it.

5.1 Reduction of the complexity of the input string

5.1.1 Block equal letters

Assigning contigs to scaffolds may lead to series of equal letters. In the longest result sequence, each of this series is taken as a whole or not taken at all. Only a part of such a series is always shorter than the whole. Therefore every series of equal letters can be united to one block which is taken or not and which has a value equal to the number of letters it represents. A single letter is represented by a block of length one. Therefore, the input string of length n will be represented by a list of m block objects ($m \leq n$) where every block object carries the information on the letter it represents and its numbers, as well as other information like the previous and the next block of this letter. This can be done in linear time $\mathcal{O}(n)$. In an extended question even nonnegative rational numbers as values are possible.

Reading the input string and translating it into block objects, also letter objects are created which store the letter name (which may be a complicated contig name), the total number of letters of it's type in the string, the number of it's blocks and more.

5.1.2 Independent substrings

If the occurrences of different letters are entangled, some of this letters have to be removed, but if a string can be divided into independent substrings with disjunct subalphabets, this will effectively reduce the complexity by reducing the magnitude of the actual alphabet. If a string with m blocks of $s = |\Sigma|$ different letters can be divided into two substrings with $m' + m" = m$ letters and subalphabets of $s' + s" = s$ different letters each, the remaining problems are essential smaller than the original problem (see equations 2.2 and 2.1). Especially the reduction of the magnitude of the used alphabets has a great impact.

During translation of a string to a list of blocks, for every block of identical letters there

can be registered whether it is the first and/or the last block of this letters in the string. In a second pass, for each position of the string the difference between first and last occurrences of letters can be determined. If this difference is zero, at this position the string can be parted into independent substrings.

5.1.3 Consecutive unique blocks

Consecutive blocks of unique letters are handled in each algorithm separately.

5.2 tree: Binary decision tree

The solution of the GROUPING BY REMOVING problem can be regarded as a binary decision tree.

In this first algorithm, every block of letters is kept or removed recursively. After the last block, the length of the remaining string is tested for maximal length and added to the results if appropriate (see fig. 5.1 as example). In every recursion step, two boolean variables, `take` to keep a letter and `reject` to remove it, are set to `False` if a constraint or a reduction rules out the respective subtree.

5.2.1 Constraints

The main constraint is tested first: If the actual letter was already taken and is not the last letter in the growing result string, this block has to be rejected (`take = False`). Therefore, a `usedCharacterList` is used.

If, as auxiliary constraint, every letter has to be taken, at least the last occurrence of each letter has to be taken: `reject = False` for the last block of a letter if it wasn't already taken. Even if not all letters have to be taken, a letter is listed in the `usedCharacterList` after all of it's blocks were rejected (see 5.2.5).

5.2.2 Reduce skipped letters

If the actual letter is the same as the last letter in the growing result string, this block has to be taken (`reject = False`). This is tested together with the main constraint.

If a block is rejected, its letter is listed in an `actualRejectedList` and it's never taken until a next block is taken because this would skip the preceding blocks with the same letter in this list.

5.2.3 Discard short solutions

If too many blocks are rejected, the growing string can't reach an already achieved length and the remaining decision tree has not to be regarded further.

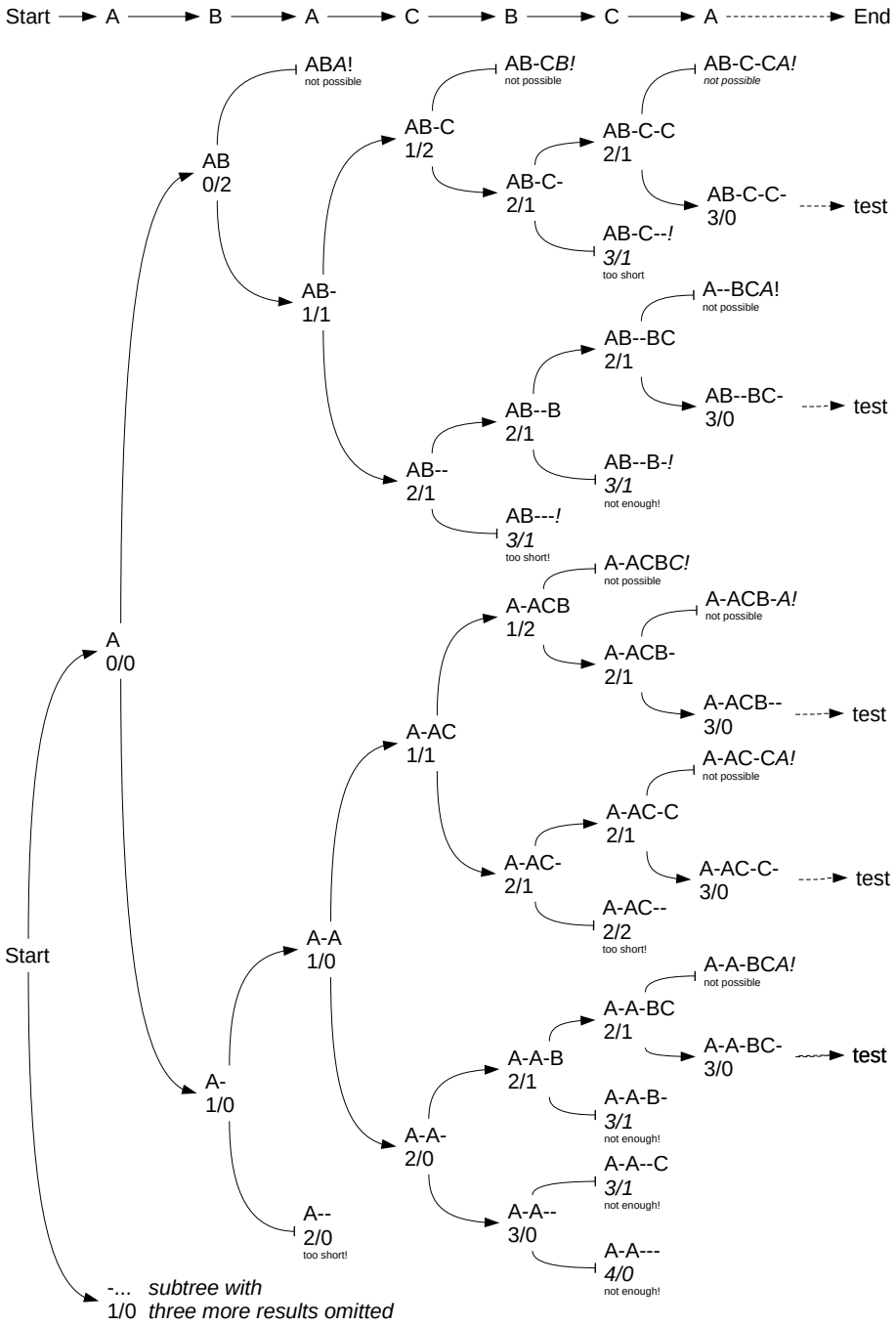

Figure 5.1: Example for the binary decision tree algorithm.
Starting from the root (Start) on the left side, for every new letter both possibilities, to `take` (upwards) and to `reject` (downwards), are considered. Some branches are not possible (tacks): Didn't correspond to the main condition (not possible), skips a possible letter (to short) or rejects to many letters (not enough), therefore already rejected letters / letters to reject are denoted. Not more than three letters can be rejected to reach the maximum length of four letters in this example. The subtree which starts with `reject` was omitted.

First, the lengths of the rejected blocks are summed up in a variable `rejected` in every recursion step.

Second, if a letter taken is followed by another letter, the remaining blocks of this letter can't be taken any more. Therefore, each block object has a `blockSum` which adds the lengths of all blocks of this letter until the actual block and each letter object has a `letterCount` for the total number of this letter in the string. The difference of both is the number of letters which can't be taken any more. A doubled counting of such a block to reject with the previous sum of already rejected blocks must be avoided.

Third, not all of the remaining blocks with so far unused letters can be taken: From m' blocks with s' still usable letters, at least s' blocks can be taken, but at least half of the remaining $m' - s'$ blocks have to be rejected. Therefore, the still usable letters from the remaining string have to be blocked and the sum of lengths of the $\lceil (m' - s')/2 \rceil$ shortest blocks have to be added to the letters to be rejected. But this needs more than $\mathcal{O}(n)$ additional calculations in each recursion step.

5.2.4 adjacent unique blocks

Adjacent unique blocks have to be taken together or rejected together. Therefore, if the actual block is a unique block, also the last taken block and the last rejected block are tested for uniqueness and the booleans `take` and `reject` are set appropriate.

5.2.5 recursion stop

If the last block was regarded, the recursion stops with testing for the length of the taken blocks which is the same as the total length minus the sum of the rejected letters.

This can be already done if all letters where taken (or rejected) after the last block of the last letter in the growing result string was taken. Therefore, even a letter was never taken in a result string, after rejecting it's last block this letter is added to the `usedCharacterList`. If this list contains all letters of the alphabet and the last taken block was the last block of its letter, all remaining blocks have to be rejected and the recursion can stop immediately to test the length of the remaining string.

5.3 walk: Random walk through binary decision tree

In recursion steps where both possibilities remain, to `take` or to `reject` a block of letters, this decision can be made randomly [probability to take: parameter `randomTree-Probability`]. In this way, many possible result strings S' which fulfill the constraints can be found quickly [number of repetitions: `randomTreeRepeats`]. But the number of longest solutions may be tiny: In one random string with 50 random letters out of an alphabet of 5 letters, only two of 75.600 possible solutions have 22 letters, all other result strings are shorter.

Fig. 5.2 shows the number of strings which correspond to the main constraint and their respective lengths. In this example, there are 75.600 possible result strings without further

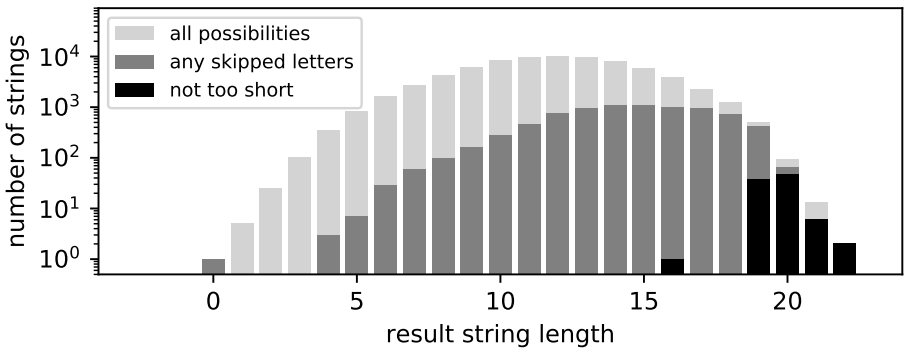

Figure 5.2: Distribution of string lengths, depending on reductions used.
String 50_5.seq (see table 6.5): 50 random letters (5 different) in 45 blocks, there are only two result strings with a maximum length of 22 characters.
gray: only equal letters blocked, 75.600 strings.
dark: any fitting letter skipped: 8.065 strings.
black: also short strings rejected: 94 remaining strings tested for length.
Number of strings on a logarithmic scale.

reductions (light boxes in fig. 5.2). There is one empty string with 0 letters, the peak are 10177 different strings of length 12, but only two strings have the maximum length of 22 letters. (Take notice of the log sale of the y-axis.)

If any compatible letter is skipped, there remain 8065 possible result strings (gray boxes in fig. 5.2) with up to 1091 strings of 15 letters.

This means, depending on the input string, only a very tiny fraction of randomly tested strings may have the maximum length, sometimes any.

Not every leaf in the decision tree has the same probability. Even if the probabilities to keep or to remove a block of letters are equal, the final probability to reach a leaf depends on the number of bifurcations b on the path from the root to the leaf: $p = (\frac{1}{2})^b$.

With the reductions which count the rejected letters, many random ways may lead to a dead end, where neither `take` not `reject` the next block will lead to a longer solution.

5.4 wide: Wide decision tree

An alternative to the binary decision tree, where the recursions are done over the (letters or) blocks of the input string and each block of equal letters is taken or rejected, in a perpendicular manner the recursions can be done over the alphabet.

In one step, a block of equal letters is chosen and all same letters left of it are kept and all on the right side are removed. The remaining substring on the right side now has at least this letter less.

If these steps were done for every letter in a string, and recursively in the same manner for every remaining substring, this results in a wide tree where every vertex (block of

letters) has many children (substrings) instead of a binary tree, where every vertex has not more than two children (`take` and `reject` this block).

Here it makes sense to do also the string reductions (block equal letters, independent substrings, adjacent unique blocks) in every recursion step because the removal of the letters of the preceding steps may alter the string structure. The same reductions as for the binary decision tree are possible, but in a different manner. These reductions were not yet implemented, therefore this algorithm was omitted in the examples in chapter 6.

5.5 dag: Directed acyclic graph

5.5.1 Principle

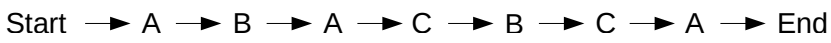

Figure 5.3: A string as a directed acyclic graph

A string can be regarded as a simple directed acyclic graph (see fig. 5.3). The removal of letters from the string can be represented by an arc form the letter before to the letter behind, skipping the removed letters.

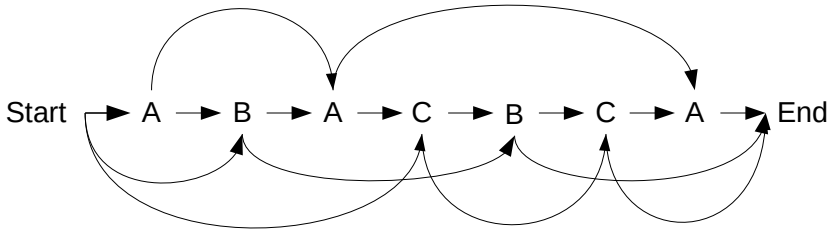

Figure 5.4: Every letter is connected to its equal predecessor and successor with an additional arc. Start and End count for all letters.
For A above the graph, for B below the graph and for C at the bottom.

To get a result string of maximal length, it is not reasonable to skip equal letters. So, the widest arcs connect equal letters (see fig. 5.4). Start and End are regarded as predecessor of every first letter and successor of every last letter in the string.

In between these arcs, the connection should only be done to the next new letter, respectively. Otherwise, one of the intermediate letters will be skipped (see fig. 5.5). In total, every letter is connected only with each nearest different letter between the preceding and the succeeding same letter.

The resulting graph (see fig. 5.6) shows all possible ways to get strings without unnecessary skips. In this graph, the longest possible way has to be found.

If adjacent equal letters are combined to blocks, every vertex represents such a block and has as value the number of letters of its block; for the total length of a way in this graph all weights of its vertices have to be added.

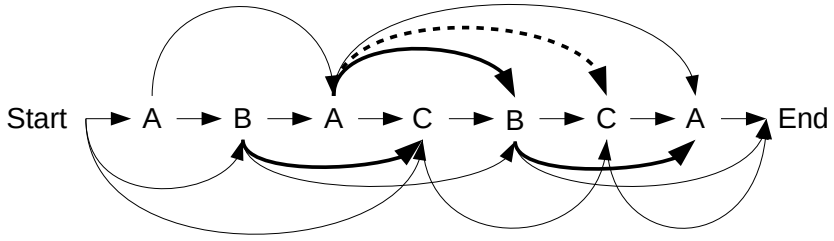

Figure 5.5: Each letter is connected with the ever first new letter until the next same letter. The bold arcs are added, the spotted arc skips the first C and connects to the second C, so it is omitted.

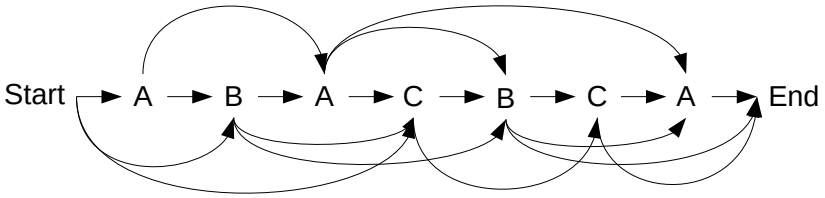

Figure 5.6: All possibilities to reject letters of this string without unnecessary skipping of same letters.

In this way all possible strings which fit to the constraints can be found in a directed acyclic graph with a limited set of arcs.

5.5.2 Algorithm

Letter by letter, for each letter a list with all possible solutions will be calculated, on the basis of the previous results. For an alphabet Σ with $|\Sigma|$ characters, each letter will have a list with at most $2^{|\Sigma|-1}$ different letter combinations: Half of the totally possible $2^{|\Sigma|}$ combinations because every solution of a letter ends with this letter, which also exist in all this strings. For every letter, there are at most $|\Sigma|$ different nearest predecessors, so in total for every of the m letter blocks in the string at most $|\Sigma|$ predecessors with at most $2^{|\Sigma|-1}$ different letter combinations have to be considered. So, this algorithm is in $\mathcal{O}(m \cdot s \cdot 2^{|\Sigma|-1})$ which will be rather fast for small alphabets and even pseudo linear for larger ones.

In detail, for each letter initially the solution list of the preceding block of the same letter is copied and every solution is elongated with the letters of the actual block. These are at most $2^{|\Sigma|-1}$ letter combinations. If a block is the first of its letter, its predecessor is the empty start string ε and the actual list has as single element the letters of this block alone. Then, from each last block with a new different letter between the actual block and this same letter block (at most $|\Sigma| - 1$ blocks of different letters), that solutions of the lists which not contain the actual letter are copied and elongated with the letters of the actual block. This are up to $s^{|\Sigma|-2}$ solutions each because half of them contain already the actual letter.

This elongated solutions are compared with the list of the already existing solutions. From the solutions using the same letters, only the longest are retained. If the result strings are sorted from the end, this can be done in $\mathcal{O}(2^{|\Sigma|-1})$ for each different letter list.

If all possible longest solutions are wanted, each element of this lists may contain several different strings of the same length, using the same letters and all ending with the same letter. If they were stored as a tree instead as a sublist, the elongations might be quite faster and require less memory (but this wasn't yet realized.)

For the predecessors, a list of the last block of every letter is used (`lastLetterBlock-List`). For every new block, in this list the last block of the same letter is searched. The blocks in the list from this same letter block to the end are used as `shortBlockList` to add the elongated lists of this letters and finally the actual block is shifted to the end of the `lastLetterBlockList`.

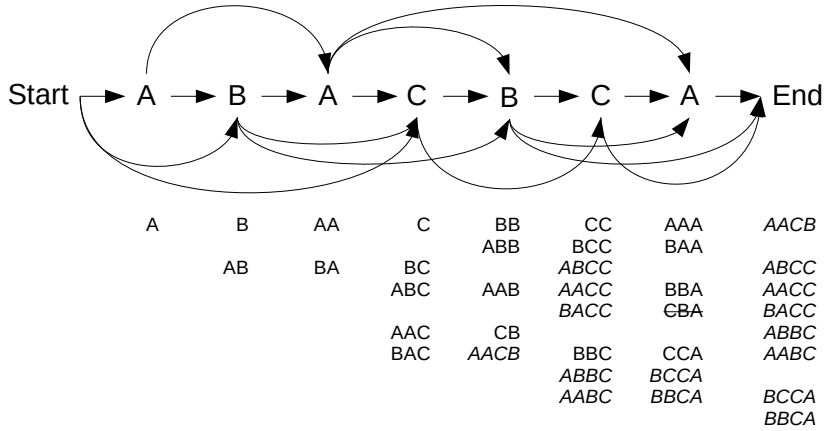

Figure 5.7: Each letter has a list of possible strings, determined by the connected predeceding strings: The strings of the last same letter are all elongated by the actual block; from the other letters only the strings without this letter are taken and elongated. The striked CBA is shorther than other strings with the same letters used. Finally, the longest strings are taken to the result list.

Predecessors of the end of the string are the last blocks of each letter of the alphabet, respectively. The longest strings of all of this last letter blocks are the final solution (see fig. 5.7 as example).

In total, this algorithm needs $\mathcal{O}(m \cdot |\Sigma| \cdot 2^{|\Sigma|-1})$ steps. For small alphabets Σ this will be quite fast.

5.5.3 Reductions

The construction of this algorithm avoids skipped letters.

The length of the solutions grows with the number of processed blocks. So the removal of strings, which will be too short, will happen only near the end of the run.

If there are two adjacent unique blocks, the first block will not be listed in the `lastLetterBlockList` (no outgoing arcs but one to the next block) and the second

block will only elongate the strings of the precedent unique block (no ingoing arcs but one from the preceding block).

5.6 ilp: Integer linear programming

A solution of the GROUPING BY REMOVING problem can be found by integer linear programming (ILP). Therefore, every block of length v_i is represented by an $x_i \in \{0, 1\}$, where $x_i = 1$ means to keep the i-th block and $x_i = 0$ to remove it, while $I = \{1 \dots m\}$ is the set of indices. This means to

$$\text{maximize} \quad \sum_{i \in I} x_i v_i \tag{5.1}$$

The constraints of the GROUPING BY REMOVING problem require some auxiliary conditions:

If x_i and x_j represent blocks of the same letter, they are kept both only if any different letter is kept between them. If there are k blocks of different letters between the i-th and the j-th block, this can be formulated as

$$kx_i + x_{i+1} + \ldots + x_{j-1} + kx_j \leq 2k \tag{5.2}$$

If only one of the blocks between is kept, at least the i-th or the j-th block has to be removed to fulfill this inequation.

There might be further blocks of the same letter between the i-th and the j-th block, but this blocks are not counted nor added, only the blocks of different letters: $k \leq j - i - 1$. For each pair of blocks of equal letters one constraint inequation is needed, so there are $\frac{1}{2} n_\sigma (n_\sigma - 1)$ constraints for each letter σ, in total

$$\sum_{\sigma \in \Sigma} \tfrac{1}{2} n_\sigma (n_\sigma - 1) \tag{5.3}$$

different inequations added as constraints.

If each letters has to be taken, at least one of the x_i which represent a block of this letter has to be taken. This gives $|\Sigma|$ additional constraints, one for each letter σ in the alphabet:

$$\sum_{i \in I_\sigma} x_i \geq 1 \quad \text{where } I_\sigma \text{ is the index set of blocks of letter } \sigma \tag{5.4}$$

From the solution of the ILP, the corresponding longest string can be obtained by taking only the blocks with $x_i = 1$. The python 3 ILP solver `pulp` returns only one possible solution.

5.6.1 Reductions

The reduction, that any appropriate letter should be skipped, can be introduced by the following inequations, one constraint for every pair of blocks of equal letters σ:

$$hx_i - x_a - \ldots - x_h + hx_j \leq h \tag{5.5}$$

where the x_a to x_h represent the h blocks of the same letter between the i-th and j-th block ($a, \ldots h, i, j \in I_\sigma$).

The combination of adjacent unique blocks at positions u and $u+1$ can be introduced by a constraint like the following for each pair of adjacent unique blocks:

$$x_u - x_{u+1} = 0 \tag{5.6}$$

There is no good possibility to eliminate short solutions early, because during the run of the ILP solver, any intermediate result is shown and the lower limits calculated before may be quite small.

But if the GROUPING BY REMOVING problem is formulated as a decision problem which asks whether a result string of at least K letters exists, the following constraint could be introduced

$$\sum_{i \in I} x_i v_i \geq K \tag{5.7}$$

and the solvability of the ILP would answers this decision problem.

6 Examples and Tests

6.1 Short examples and unit tests

For testing purposes, a string of 87 characters was used which can be divided into 8 independent substrings: `AAAABAACCCCCCAAABAA DDEDD FFFFFF GGHHGGHH XYXYXYXYXYXY LJKJKLJK MNMNOOOPPPPQQQNMMM STSTUSUVUSU` (TestSeq.seq).

Most substrings have only one solution, some have more than one solution. In total, there are $1 \cdot 1 \cdot 1 \cdot 2 \cdot 6 \cdot 3 \cdot 1 \cdot 2 = 72$ different solutions for this test string, each with 63 characters (see table 6.1).

For the comparisons without reductions, this string is far too long, therefore a part of this string without the `AAAABAACCCCCCAAABAA` and `XYXYXYXYXYXY` parts was used: `DDEDD FFFFFF GGHHGGHH LJKJKLJK MNMNOOOPPPPQQQNMMM STSTUSUVUSU` (TestSeqK.seq). This string has 56 different letters and 12 longest solutions with 43 letters each.

6.1.1 Unit-tests

Each of the eight substrings of `TestSeq.seq` were used for unit-testing of the algorithms.

A further test string with 49 random characters out of an alphabet of 7 different letters was also used in unit-tests. This string has 61 longest solutions with 21 characters each, 56 of this solutions use all seven letters (49_7.seq): `BEEBCEDFEAEAGDCEBBFDEGEDDFGBBFBGEDFAFGABGEACGAGGC`

6.1.2 Calculation time measurements

The only useful parameter to compare different algorithms is the calculation time. The absolute time depends strongly on the computer system used, but the relations between different examples should be similar on different systems.

The calculation times where measured on a computer with Intel i5-7200U processor with 2.5 GHz and 8 GByte RAM, Ubuntu 18.04.3 LTS and Python 3.6.5. Depending on the multitasking of the operation systems, the calculation times are not always the same. From multiple runs the shortest calculation time was taken and the time is noted with not more than two significant digits.

Because outputs needs a substantial amount of time, nearly all outputs where suppressed during calculation time tests.

Table 6.1: Strings to test the reductions

original string result strings	original length result length	remark number of solutions
AAAABAACCCCCCAAABAA	19 *	first try
AAAA_AACCCCCC____B__	13	1
DDEDD	5	not all letters
DD_DD (*DDE__ and __EDD*)	4	1 (*2 with all letters*)
FFFFFF	6	simple block
FFFFFF	6	1
GGHHGGHH	8	alternating blocks
GGHH__HH	6	2
GG__GGHH		
XYXYXYXYXYXY	12 *	alternating letters
XY_Y_Y_Y_Y_Y	7	6
X_XY_Y_Y_Y_Y		
X_X_XY_Y_Y_Y		
X_X_X_XY_Y_Y		
X_X_X_X_XY_Y		
X_X_X_X_X_XY		
LJKJKLJK	8	count rejected letters
LJK_K__K	5	3
LJ_JK__K		
LJ_J__JK		
MNMNOOOPPPPQQQNMMM	18	blocks of unique letters
_N_NOOOPPPPQQQ_MMM	15	1
STSTUSUVUSU	11	count blocks to reject
ST_TU_U_U_U	7	2
S_STU_U_U_U		

* only in TestSeq.seq, in TestSeqK.seq omitted

6.2 Comparisons of the reductions and algorithms

6.2.1 Pre-algorithmic reductions on the input strings

In the test string with 6 independent substrings (TestSeqK.seq), the separation of this substrings prior to the algorithms has a great impact on the calculation time (last block in table 6.2). With the random walk algorithm, only in a few runs with 10.000 trys each the solution with maximum length was reached. Only in the last part, in the short substrings, the walk algorithm always reaches the maximum length in every of the short substrings. Only the algorithms tree and dag are able to report all possible solutions (second result line each).

Table 6.2: Pre-algorithmic reductions with shortened test string TestSeqK.seq

Input string and one result string:
`DDEDD FFFFFF GGHHGGHH LJKJKLJK MNMNOOOPPPPQQQNMMM STSTUSUVUSU`
`DD_DD FFFFFF GGHH__HH LJ_J__JK _N_NOOOPPPPQQQ_MMM S_STU_U_U_U`
56 letters (17 different) in 36 blocks in 6 indep. substrings, 12 solutions of length 43

tree		**walk ***		**dag**		**ilp**	
recursions (one)		correct (one)		lists (one)		constraints (one)	
(all solutions)				compared (all)			
$P_{mc} = 24.192.000.000$:		*any reduction*					
24.192.000.000	**5-9 d****	*few*	**4,5 s**	645.267	**24 s**	58	**0,14 s**
24.192.000.000	**5-9 d****			645.267	**44 s**		
$P_{mc} = 24.192.000.000$:		**take only new letters**					
26.113.019	**7,5 min**	*few*	**5,8 s**	394.510	**14 s**	116	**0,22 s**
26.113.019	**6,9 min**			394.510	**21 s**		
$P_{mc} = 99.532.800$:		**+ block letters**					
25.551.192	**6,5 min**	*few*	**3 s**	394.510	**12 s**	58	**0,09 s**
25.551.192	**6,9 min**			394.510	**28 s**		
$\Sigma\, P_{mc} = 243$:		**+ independent substring**					
452	**0,003 s**	*all*	**1,1 s**	130	**0,007 s**	58	**0,07 s**
452	**0,003 s**			130	**0,007 s**		

* walk: 10.000 trys; portion of results of correct length, `randomTreeProbability`=0,5
** estimated by extrapolation the calculation times of the subtrees

The reduction `takeOnlyNewLetters` has a great impact on the possibilities left and also on the number of recursions and the calculation times especially in the tree algorithm. In this example, the ilp algorithm is by far the fastest. The construction of the constraints in

this short string needs much more time than the solving of the ILP itself. This masks the effect of the reduction `takeOnlyNewLetters`. Therefore, an other example is given in table 6.6.

6.2.2 Discard short solutions

In this short sequences, the longest subsequence has 11 blocks, counting the letters rejected or to reject has no large effect (see tab. 6.3). The time necessary is minimal, only the number of recursions or lists compared show differences. In longer sequences, these differences will be clearer. In the ilp algorithm there is no possibility to consider the lengths of the intermediate results, so there it's always the same and the fields of the table left blank. Also in the dag algorithm some reductions are not possible.

Table 6.3: Reduction with length with shortened test string TestSeqK.seq

Input string and one result string:
`DDEDD FFFFFF GGHHGGHH LJKJKLJK MNMNOOOPPPPQQQNMMM STSTUSUVUSU`
`DD_DD FFFFFF GGHH__HH LJ_J__JK _N_NOOOPPPPQQQ_MMM S_STU_U_U_U`
56 letters (17 different) in 36 blocks in 6 indep. substrings, 12 solutions of length 43 from $P_{mc} = 243$ possibilities

tree	**walk ***	**dag**	**ilp**
recursions	correct	lists compared	constraints
independent substrings + block letters			
452 **0,003 s**	*many* **1,1 s**	130 **0,006 s**	58 **0,08 s**
+ count rejected			
221 **0,002 s**	*many* **0,7 s**	115 **0,005 s**	
+ count to reject			
132 **0,002 s**	*many* **0,6 s**	115 **0,006 s**	
+ count min to reject			
111 **0,002 s**	*many* **1,0 s**		
+ no more letters (early recursion break)			
108 **0,002 s**	*many* **1,0 s**		

* walk: 10.000 trys; portion of results of correct length, `randomTreeProbability`=0,5

6.2.3 Adjacent unique blocks

The examples above have any adjacent unique blocks. Therefore, a short example which is part of the test sequences is shown (table 6.4). This example was done without the reductions counting the rejections. With considering these counts this reduction has any effect on this short string.

Table 6.4: Reduction for adjacent unique blocks

`MNMNOOOPPPPQQQNMMM` (input string, part of TestSeq.seq and TestSeqK.seq)
`_N_NOOOPPPPQQQ_MMM` (result string)
18 letters (5 different) in 9 blocks, 3 of them unique, 1 solution of length 15
from $P_{mc} = 128$ possibilities

tree		**walk ***		**dag**		**ilp**	
recursions		correct		lists compared		constraints	
independent substrings + block letters							
164	**0,001 s**	*all*	**0,3 s**	*67*	**0,002 s**	12	**0,01 s**
	+ uniques together						
68	**0,001 s**	*all*	**0,3 s**	21	**0,001 s**	14	**0,01 s**

* walk: 10.000 trys; portion of results of correct length, `randomTreeProbability`=0,5

In this short example, the calculation times are minimal but the number of recursions and lists compared are reduced considerably. The two pairs of adjacent unique blocks add two additional constraints to the ilp algorithm.

6.3 Comparison of random strings

Because any nontrivial biological example is available [Manish Goel, personal communication], some further random strings were compared.

In the following examples, all possible reductions are used to reduce the calulation times. The time consuming reduction `minToReject` which calculates the minimum length of the remaining blocks to be removed reduces the number of recursion steps significantly, but slightly elongates the calculation time in longer strings (data not shown).

6.3.1 Same string length with different magnitudes of the alphabet

The calculation time of the algorithms depends strongly on the number of letters in the alphabet Σ. Here, strings of 50 letters but different alphabets were compared. Random

strings were generated and such strings were chosen which can't be divided into substrings and which have the full number of different letters (table 6.5).

In this examples, the calculation times and the number of lists compared in the dag algorithm clearly increase nearly exponentially with the magnitudes of the alphabets, as it has to be expected. Even in a short string with only 50 letters, an alphabet of 20 letters is to large to calculate all possible solutions. The run dies after finding several millions of letter lists, possibly due to memory restrictions.

If every solution is wanted, the tree algorithm is appropriate despite its growing calculation times. If only one solution is sufficient, the ilp algorithm is the fastest.

With large alphabets, the calculation time parameters reduce because there are less blocks remaining for every different letter of the alphabet, also adjacent blocks of unique letters may occur more frequently.

6.3.2 Different string lengths with same alphabet

The calculation time of the algorithms also depends on the number of letters n or blocks m in the string (table 6.6).

With growing lengths, the calculation time parameters grow in all examples. With this limited alphabet of ten letters, the dag algorithm is the fastest. Its calculation times and numbers of compared lists grow nearly linear with the lengths of the strings. Just as the random walk algorithm, which never reaches the full result length in any of at least 10 runs of 10.000 trys each.

In the tree algorithm the increase of recursions done and calculation time shows a steep increase.

In contrary to the examples with the restricted alphabet, in the ilp algorithm the calculation time increases so heavy that in more than one day of calculation time the last example with 200 letters in 176 blocks didn't reached a solution. This may be due to the fact, that with the number n_σ of each letter σ increases the number of independent constraints quadratically. In strings with small alphabets, the numbers n_σ are much higher than in strings of the same length but with larger alphabets.

Here, the effect of the `takeOnlyNewLetters` constraints could be tested: omitting these additional constraints clearly rises the calculation times of the ilp algorithm in larger strings.

For limited alphabets, the dag algorithm seems to be the most suitable algorithm.

Table 6.5: Strings with 50 letters, but different magnitudes of the alphabet

`INPUT STRING` (filename)
`RESULT STRING`
random sequence characteristics

tree		**walk ***		**dag**		**ilp**	
recursions (one)		correct (one)		lists (one)		constraints (one)	
(all solutions)				compared (all)			
`EEADECCECEDCECDCEEACDACDDDBAEBCECACDCDCDABCDEABEAE` (50_5.seq)							
`EE__E__E_E__E___EE__D__DDDB__BC_C_C_C_C_A____A__A_` (ilp solution)							
50 letters (5 different) in 45 blocks, there are 2 solutions of maximal length 22							
from $P_{mc} = 75.600$ possibilities							
1.371	**0,023 s**	*few*	**3,9 s**	475	**0,015 s**	406	**6,9 s**
2.597	**0,40 s**			475	**0,017 s**		
`EJGJHCEJCBGGFABCEGJFBBIBDJIFHHHAHDCEJGEBDAJIJBEDHI` (50_10.seq)							
`_J_J___J__GGF_____FBB_BD___HHH_H_CE__E__A_I_____I` (ilp solution)							
50 letters (10 different) in 46 blocks, there are 155 solutions of maximal length 21							
from $P_{mc} = 22.050.000$ possibilities							
43.839	**1,2 s**	*all*	**7,5 s**	28.090	**0,7 s**	188	**5,1 s**
95.797	**2,4 s**			28.090	**1,2 s**		
`NFJDLDKHJBIBOGGEEDJNHEBAFOAOMLLIHMCLGHJDMCJCKAINEI` (50_15.seq)							
`NFJD_D___B_BOGGEE____E_A__A__LL_HMC_____C_CK_I__I` (ilp solution)							
50 letters (15 different) in 47 blocks, there are 420 solutions of maximal length 25							
from $P_{mc} = 1.327.104.000$ possibilities							
97.497	**3,9 s**	*all*	**11 s**	652.493	**24 s**	110	**1,9 s**
258.673	**10 s**			652.493	**50 s**		
`FITJEJIGLEGMRMCOHTGQKCTPMASTISQODQBDCIQEKGEONIFPEC` (50_20e.seq)							
`FI_J_J_GLE_MR_COHT____T__AS__SQ__QBD____K___N__P__` (ilp solution)							
50 letters (20 different) in 50 blocks, there are 13884 solutions of maximal length 24							
from $P_{mc} = 16.796.160.000$ possibilities							
2.177.371	**2,1 min**	*some*	**20 s**	9.937.973	**20 min**	112	**2,4 s**
7.358.090	**6,3 min**			*not possible*	**		
`JSYCSEXdnFHKYUCLHMXLFALQQRABPEETSZJMKLwiTJVXRZGPLU` (50_25a.seq)							
`JS__S_XdnFHY__CL___L__LQQ_AB_EE____MK_wiT_V_RZGP_U` (ilp solution)							
50 letters (25 different) in 48 blocks, there are 384 solutions of maximal length 30							
from $P_{mc} = 156.728.328.192$ poss., two blocks of adjacent unique letters (lowercase)							
293.827	**22 s**	*any*	**20 s**	22.110.284	**17 min**	66	**0,55 s**
860.665	**64 s**			*not possible*	**		

* walk: 10.000 trys; portion of results of correct length, `randomTreeProbability`=0,5
** dag: not possible, probably due to insufficient memory

Table 6.6: Strings with different string lengths but same alphabet of 10 letters

`INPUT STRING` (filename)
`RESULT STRING`
random sequence characteristics

tree		**walk ***		**dag**		**ilp**	
recursions (one)		correct (one)		lists (one)		constraints (one)	
(all solutions)				compared (all)			
`EJGJHCEJCBGGFABCEGJFBBIBDJIFHHHAHDCEJGEBDAJIJBEDHI` (50_10.seq)							
one dag result string: `JJJGGABBBIIFHHHHDCEEE`							
50 letters (10 different) in 46 blocks, there are 155 solutions of maximal length 21							
from $P_{mc} = 22.050.000$ possibilities							
43.839	**1,2 s**	*all*	**7,5 s**	28.090	**0,7 s**	188	**5,1 s**
95.797	**2,4 s**			28.090	**1,2 s**	[94	**5,0 s** **]
`FJECAHICFCDEICFGDEIIFEJAJHCAHGCFGDGIJFBBCBCGJGDBIG-` `FEHDJBEFDDJCIGECHGEHCBFEBFGBBJIFAFEHBJHIEEHEBEAJEB` (100_10.seq)							
one dag result string: `AIIIIJJHHGGGGGDDDDCCCBBBBFFEEEEEE`							
100 letters (10 different) in 95 blocks, there are 2 solutions of maximal length 33							
from $P_{mc} = 11.772.961.200$ possibilities							
4.263.531	**4,4 min**	*any*	**23 s**	78.340	**2,1 s**	872	**13 min**
7.320.753	**5,5 min**			78.340	**8,5 s**	[436	**36 min** **]
`ACEBJDIJJICJDFGBFJJAIIECIDBAFHJBCHBGHBBAFGFAGCGFGH-` `IGEECHDHDIDIHBDFJGIHHCJEBBFCDGGGCGDFDGIDGDICIGGGBD-` `DJFJDICIBGEBCECFIJAECJFGIEECGIFEJDCBBFEFGCGACBDBEB` (150_10.seq)							
one dag result string: `JJJJJJIIIBBBBBAFFFHHHHHHCCGGGGGGGGGDDDEEEEEEEE`							
150 letters (10 different) in 136 blocks, there are 3 solutions of maximal length 46							
from $P_{mc} = 299.986.771.968$ possibilities							
30.146.148	**30 min**	*any*	**37 s**	117.305	**3,2 s**	1852	**4,6 h**
41.335.975	**34 min**			117.305	**8,6 s**	[926	**30 h** **]
`DDEEEBGFFBGJJFIEHGJDCEBEFHBJFDDCEGFFHGHCHGIAFHCGGF-` `GDCHECEABIDHDAGAIGCCJBHIDDDDJDCJHBFCCBGIICGIFBGGGG-` `DCIEDGFACFDIFGGAFIEAJDDFBAJAIHABCJFHHIAEHEHDGFIGHC-` `FGCGGCHFEADGAEJIIFHIFGGHFHEIAEJCAJBEHEBGEAFHAHJJEI` (200_10.seq)							
one dag result string: `EEEFFFFFFFFFDDDDDDDDCCCBGGGGGGGGGAAAAIHHHHHHHHHHHHHJJ`							
200 letters (10 different) in 176 blocks, there are 36 solutions of maximal length 53							
from $P_{mc} = 4.172.327.107.200$ possibilities							
300.346.290	**4,8 h**	*any*	**55 s**	165.020	**4,5 s**	3038	**>32 h**
417.351.913	**6,2 h**			165.020	**14 s**		

* walk: 10.000 trys; portion of results of correct length, `randomTreeProbability`=0,5
** ilp without `takeOnlyNewLetters` constraints

7 Conclusion and Summary

Analyzing the variety of data achieved by next generating sequencing needs different bioinformatic tools. Sometimes it is helpful to align contigs to a known sequence of a different but closely related species as template. In this case many parts may match well to the scaffold, but some parts, probably from genes or gene copies which had been translocated in the genome, disturb this alignment. An easy way to find and remove such alterations is necessary. This can be done by translating the alignment to a string where each contig is represented by a different letter. Then, alterations may be different letters inserted in blocks of equal letters. By removing this different letters, a possible alignment is shown. In more complex situations, the longest possible alignment is wanted.

This can be translated into the string partitioning problem GROUPING BY REMOVING: Transforming an input string S into a result string S', where every letter occurs only in one coherent block, by removing as few letters as possible. This is an NP-hard problem.

In this thesis, some algorithms with reductions where developed to find such strings S'.

The first algorithm is a binary decision tree, where every letter (or block of equal letters) of the input string recursively is kept and removed. Depending on the string, sometimes one or even both branches are cut if they can't reach the longest possible solution. With this algorithm all possible solutions can be found, but the calculation time and the number of recursion steps increase heavily with the length of the input string.

A heuristic version of this algorithm is the random walk algorithm, where at each bifurcation in the binary decision tree the next branch is chosen by random. But in longer strings only a tiny amount of possible solutions reach the maximal length (see fig. 5.2) and it is quite unlikely to find such a solution even in many random walks.

An other algorithm regards the input string as a directed acyclic graph and constructs letter by letter (or block by block of equal letters) all possible solutions from its predecessors. The calculation time of this algorithm grows linear with the string length n, but with the magnitude of the alphabet Σ used, the demands of time and memory grow more than exponentially with $\mathcal{O}(|\Sigma|\ 2^{|\Sigma|-1})$. For small alphabets this algorithm is the fastest, but for larger alphabets (e.g. $|\Sigma| = 20$ on a laptop) it may overburden the system.

Finally the problem can be translated into an integer linear program. If each (letter or) block of letters σ occurs m_σ times in a string, $\frac{1}{2}m_\sigma(m_\sigma - 1)$ constraints are necessary for every letter. If these numbers remain small, even problems with large alphabets can be solved quickly using available ILP solver, but for larger numbers of any letter the calculation times may grow extreme. With the python ilp module pulp, only one possible solution is found.

So, depending on the characteristics of the input string, different algorithms may be optimal to find a longest solution for the GROUPING BY REMOVING problem.

References

M. S. Garey and D. S. Johnson (1979). Computers and Intractability. Bell Telefone Laboratories, Incorporated.

M. Grötschel, M. Jünger, and G. Reinelt (1984). "A cutting Plane Algorithm for the Linear Ordering Problem". In: Operations Research 32 (6).

S. E. Levy and R. M. Myers (2016). "Advancements in Next-Generation Sequencing". In: Annual review of genomics and human genetics.

R. Staden (1980). "A new computer method for the storage and manipulation of DNA gel reading data". In: Nucleic acids research 8 (16).

Acknowledgements

Thanks to Prof. Dr. Gunnar Klau for the subject and supervising of this work.

Thanks to Manish Goel for imparting the biological background.

Thanks to Sven Schrinner and Philipp Spohr for their supervising, the ideas of the NP reduction and their help by LaTeX problems.

Code

The program code in python 3 and the example strings are available at:
https://gitlab.cs.uni-duesseldorf.de/wulfert/string-assignment/tree/master/BAPython/GBR/
(30.12.2019)