FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

# BACHELOR THESIS

Ekaterina Milyutina

**Efficient representation of k-mer sets**

Computer Science Institute of Charles University

Supervisor of the bachelor thesis: Mgr. Pavel Veselý, Ph.D.

Study programme: Computer Science

Study branch: General Computer Science

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In…....... date...........                                                                 signature

I want to express sincere gratitude to my supervisor Mgr. Pavel Veselý, Ph.D.  for all his invaluable help and mentorship in writing this thesis. I extend my thanks to the project consultant, Karel Břinda, Ph.D., whose expert insights significantly improved my understanding of this complex topic. I thank my family and my fiancé who supported me through this academic journey.

Title: Efficient representation of k-mer sets

Author: Ekaterina Milyutina

Institute: Computer Science Institute of Charles University

Supervisor of the bachelor thesis: Mgr. Pavel Veselý, Ph.D., Computer Science Institute of Charles University

Consultant: Karel Břinda, Ph.D.

Abstract: In this thesis we explore and compare various methods for efficient k-mer set representation. We evaluate traditional de Bruijn graph representation techniques against greedy approximation algorithms for the Shortest Superstring Problem. We describe the linear-time implementation of the well-known Greedy algorithm by Ukkonen [1990] and extend it to another related algorithm, called TGreedy. In addition, we test selected algorithms on a bacterial genome and pangenome to highlight the differences in the size of their output representation and the computational resources used, providing an insight into their respective efficiencies.

Keywords: k-mers, k-mer set representation, de Bruijn graphs, Simplitigs, Shortest Superstring problem, overlap graphs, mask

# Contents

**Preface** .......... 3
**Preliminaries** .......... 5
**1. K-mers** .......... 6
1.1. Introduction to k-mers .......... 6
1.2. Representations of k-mer Sets .......... 8
1.2.1. de Bruijn Graph .......... 8
1.2.2. Unitigs .......... 9
1.2.3. Simplitigs .......... 10
1.2.3. Matchtigs .......... 12
1.3. Simplitigs Algorithm .......... 14
1.3.1. Time Complexity Analyze .......... 14
**2. Shortest Superstring Problem** .......... 15
2.1. Introduction to Shortest Superstring Problem .......... 15
2.2. Greedy Algorithm for SSP .......... 15
2.3. Overlap Graph .......... 16
2.4. Linear-time implementation of Greedy (Greedy AC) .......... 18
2.4.1. Aho-Corasick Automaton .......... 18
2.4.2. Greedy_AC Algorithm .......... 19
2.5. TGreedy .......... 21
2.5.1. Cycle Cover .......... 22
2.5.2. MGREEDY .......... 22
2.5.3. TGREEDY Algorithm .......... 23
**3. Connection between k-mers and SSP** .......... 25
3.1. Linking k-mers representation problem and SSP .......... 25
3.1.1. Mask .......... 25
3.2. Utilizing SSP Algorithms for K-mers .......... 26
**4. Experiments** .......... 28
4.1. Objective .......... 28
4.2. Setup .......... 28
4.2.1. Programming Environment and the Computational Resources .......... 28
4.2.2. Input Values .......... 29
4.2.3. Data Collection .......... 30
4.3. User Manual .......... 31

4.4. Results ........ 32
4.4.1. Comparison of Superstrings' Lengths ........ 33
4.4.2. Comparison of Computational Efficiency ........ 36
**Conclusion** ........ 40
**Bibliography** ........ 42
**List of Figures** ........ 44
**Attachments** ........ 46
A.1. Tables ........ 46
A.2. Technical Documentation ........ 48
A.2.1. MAIN Modules ........ 48
A.2.2. HELPER Modules ........ 53
A.2.3. ALGORITHM Modules ........ 60

# Preface

The main goal of this thesis lies in exploration, comparison and identification of the most efficient methods for representing k-mer sets, which are the sets of all possible subsequences of length *k* that can be extracted from a genomic sequence and are crucial elements in genomics. By assessing different representational strategies, this work aims to contribute to the broader understanding of k-mer sets data storage and manipulation with it.

In Chapter 1 we start with an analysis of de Bruijn graphs, commonly used structures for representing k-mer sets, and their representation methods. The focus is given to the known state-of-the-art methods, like unitigs and simplitigs, which are vertex-disjoint paths in the de Bruijn graph, and computations of such sequences from de Bruijn graphs and in some cases straight from the k-mer set.

Building on this foundation, the contribution of this thesis is to investigate the connections between k-mers set representations and a well-known NP-hard problem, the Shortest Superstring Problem (SSP). Superstrings are an essential concept in study of k-mer sets representations, as they are results of merging the simplitigs and unitigs, and finding the shortest superstring will lead to optimizing the size of the k-mers representation.

Chapter 2 is dedicated to the investigation of the SSP problem and its well-known greedy approximation algorithms. In addition, we introduce overlap graphs as an essential foundation for greedy algorithms with improved guarantees, and discuss in detail these SSP methods. Particular focus is directed towards a linear-time implementation of the greedy approach (Greedy_AC) that uses a modified Aho-Corasick automaton to find the longest Hamiltonian Path in the overlap graph. In addition, this thesis explores other greedy algorithms based on the cycle cover, such as MGreedy and its improved version, TGreedy, linear-time implementation of which will be introduced for the first time.

By the investigation of de Bruijn graph representations and the Shortest Superstring Problem, in Chapter 3 we derive the conclusion that the SSP greedy algorithms have the potential to optimize the representation of k-mer sets, providing better size of representation in expense of more computational resources, such as runtime and memory usage. Under such observation, Greedy_AC and TGreedy algorithms can be utilized for the k-mer set representation. In this chapter we introduce the concept of mask, applied to the superstring, enabling the preservation and identification of the original k-mers while preventing false positives. Different types of masks, such as binary and case sensitive, is explored and the observations about construction of the mask is obtained.

The practical part, described in Chapter 4 of this thesis, involves the prototype implementation of some of the mentioned algorithms in Python. Despite Python's limitations with large k-mer data sets, it suffices for testing these approaches.

The methods selected for the implementation are: algorithm for simplitigs construction from the hash table of k-mers proposed by Břinda, Baym and Kucherov [2021]; linear time implementation of Greedy of Ukkonen [1990]; and TGreedy, first introduced by Blum et al. [1994], for which we describe the first linear-time implementation.

Through the experiments with these three approaches, the aim is to compare the efficiency of the generated superstrings and the computation times, seeking the most optimal solution for k-mer set representation.

Ultimately, this thesis seeks to lay the groundwork for further research in this field, advancing the understanding and application of efficient k-mer set representation strategies.

# Preliminaries

In this chapter we introduce some definitions that are essential for the whole thesis.

*Alphabet* $\Sigma$ is a finite nonempty set of symbols.

A *string* over $\Sigma$ is a finite sequence of symbols from $\Sigma$. The length of a string is the number of symbols in the sequence, denoted by $|s|$, where $s$ is a string. A contiguous sequence of characters within a string is called *substring*, which shares the same alphabet $\Sigma$.

Two strings $x$ and $y$ have an *overlap* of length $k$ if there exist strings $u$, $v$ and w with $|v| = k$, such that $x = uv$ and $y = vw$. In other words, an overlap is a string that occurs at the end of the first string and at the beginning of the second string, i.e. it is a suffix of one string and a prefix of the other.

The *maximum overlap* between two strings is the longest overlap.

*DNA* (Deoxyribonucleic acid) is the carrier of genetic information. DNA consists of four nucleotides or bases, denoted A, T, G, and C. *Genome* is all the generic information of an organism and consists of DNA sequences. Both definitions are stated by Rith [2019].

# 1. K-mers

## 1.1. Introduction to k-mers

In the field of genomics and bioinformatics, the concept of k-mers plays a fundamental role in understanding and analyzing DNA and protein sequences. The term *k-mer* refers to a fixed-length substring that is created from a longer genomic string, like sequencing reads or transcription. Such a subsequence employs the alphabet ["A", "C", "T", "G"], where the characters stand for the nucleotides Adenine, Cytosine, Thymine, and Guanine, respectively.

A *k-mer set* is a collection of all possible substrings each of a fixed length *k*, produced from a certain genome sequence. Such sequence of size $L$ contains $L - K + 1$ k-mers, which, however, are not necessary to be unique. The *size of a k-mer set representation* refers to the total number of characters in the representation.

As an example let us have a look at an arbitrary DNA sequence and all of its possible unique k-mers. Let the sequence be “GCTACTA” and the table of k-mers is as follows:

| Value of *k* | k-mer Set |
|---|---|
| 1 | G, C, T, A |
| 2 | GC, CT, TA, AC |
| 3 | GCT, CTA, TAC, ACT |
| 4 | GCTA, CTAC, TACT, ACTA |
| 5 | GCTAC, CTACT, TACTA |
| 6 | GCTACT, CTACTA |
| 7 | GCTACTA |

For a particular sequence, the sizes of sets depend on the value of *k* and as it increases to a certain value the size of the sets increases as well. That certain value of k indicates the limit of the number of unique k-mers presented in the DNA sequence.

For instance, the length of the sequence of further used genome of the bacteria s. pneumococcus, is equal to 2061918. As it is seen from the following table the number of the unique subsequences of such a genome is increasing exponentially for the values of *k* in the range from 1 to 11. However, as the size of the k-mer set gets close to the length of the genomic sequence, the growth significantly slows down and reaches its maximum at certain value *k*, which will indicate the maximum number of unique k-mers for this genome and in the observed case is equal to 57. After this value the sizes of the k-mer sets will only decrease, as more extensive repetitions are presented.

| Value of $k$ | Size of k-mer set |
|---|---|
| 1-7 | $4^k$ |
| 8 | 65101 |
| 9 | 241952 |
| 10 | 691301 |
| 11 | 1298893 |
| 12 | 1730646 |
| 13 | 1924270 |
| 14 | 1992657 |
| 15 | 2015184 |
| 20 | 2030690 |
| 30 | 2038948 |
| 40 | 2043714 |
| 50 | 2046162 |
| 56 | 2046814 |
| 57 | 2046816 |
| 58 | 2046807 |
| 100 | 2043877 |
| 1000 | 1982627 |
| 100000 | 145012 |

The examples provided illustrate the *unidirectional model* of k-mers. Such a model treats each k-mer as a distinct entity without considering its reverse complement.

DNA sequence is composed of two long strands twisted together. Each stand consists of a series of nucleotides. Reading of DNA sequence is a process of determining the sequence of nucleotides along one of the DNA strands. When we read the DNA sequences, we start from one end without knowing the actual direction.

In order to account for the possibility of sequencing from either direction, the *bidirectional model* is used. In such a model both the k-mer and its reverse complement are considered equivalent, compared to the unidirectional model.

The *reverse complement* of a DNA sequence is formed by reversal of the sequence and replacement of each nucleotide with its complement: A with T, C with G, G with C and T with A. For instance, for the sequence “GCTACTAGC” its reverse complement is “GCTAGTAGC” and the k-mer “GCT” and its reverse complement “AGC” are treated as equivalent entities.

The bidirectional approach provides a more complete representation of the sequence by accounting both orientations of k-mers. By introducing reverse components, the bidirectional model enables more exploration of the repetitions and overlaps, which appear after reaching the certain limit value of $k$, as demonstrated in the bacteria's DNA sequence example. Bidirectional approach enables the identification of overlaps that may not be immediately apparent when using only the unidirectional model, resulting in a more accurate genomic analysis.

In this thesis, we primarily focus on using the unidirectional model for analyzing DNA sequences. While we primarily utilize the unidirectional model, results obtained within this model are expected to hold true in the bidirectional model as well due to the inherent symmetry of DNA, where complementary pairs (A-T and C-G) exist regardless of the reading direction.

## 1.2. Representations of k-mer Sets

Genomic data is vast and managing such a large amount of data can be computationally and memory intensive. Efficient representation of k-mer sets, obtained from such a genome, allows to store the same information using less space, making such storage cost-efficient. The efficient representation of k-mer sets is crucial for the efficiency of many bioinformatics algorithms as they can be processed faster.

### 1.2.1. de Bruijn Graph

One of the solutions for representation of genomic data is de Bruijn graphs.

The *de Bruijn graph*, a directed graph with the notation *G = (V, E).* The nodes in the graph represent k-mers, i.e., $V$ = set of k-mers. Each edge in the graph represents the *k-1* long overlap between the pair of the k-mers , i.e. $E = \{(u, v) \in K^2 \mid u \rightarrow_{k-1} v\}$, where $K^2$ represents the Cartesian product of the k-mers set with itself (all possible pairs of k-mers), and $u \rightarrow_{k-1} v$ indicates that there is a directed edge from k-mer $u$ to k-mer *v,* where $u$ and *v* overlap by *k-1* characters.

### 1.2.2. Unitigs

One of the important concepts in the representation of such graphs is *unitigs,* which was firstly used by Kececioglu and Mayers [1995]. A unitig is a continuous sequence of nodes (k-mers) in the de Bruijn graph that do not contain any forks, which occur when one k-mer has different possible extensions to other k-mers. In other words, unitig represents a unique vertex-disjoint path through the graph that corresponds to a potential sequence in the original DNA or protein sequence. The example of the unitigs is presented in Figure 1.1.

Starting from the initial de Bruijn graph *G*, the process of identifying unitigs involves traversing the graph and examining each path. The sequence of nodes $n_0, n_1 ... n_p \in G$ is marked as unitig, if node $n_0$ has only one outgoing edge, node $n_p$ has only one in-going edge and any node $n_i$, where $i \in [1, p-1]$, has both in-degree and out-degree equal to one.

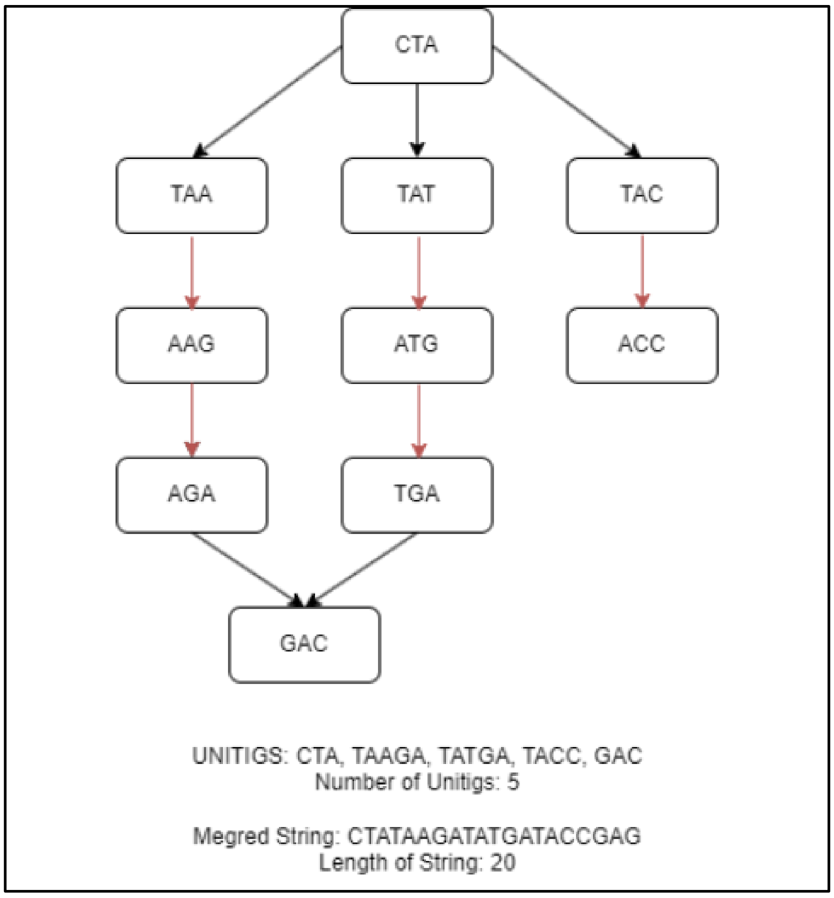
CTA
TAA
TAT
TAC
AAG
ATG
ACC
AGA
TGA
GAC
UNITIGS: CTA, TAAGA, TATGA, TACC, GAC
Number of Unitigs: 5
Megred String: CTATAAGATATGATACCGAG
Length of String: 20

*Figure 1.1: Representation of k-mer sets as unitigs*

The tool for computing unitigs from input data is BCALM 2 developed by Chikhi, Limasset and Medvedev [2016]. This program is designed to take a FASTA file containing DNA or protein sequences as input and efficiently constructs the corresponding de Bruijn graph. Once such a graph is constructed, BCALM 2 employs an algorithm to extract unitigs from the graph. The output of the program is a set of unitigs, where each element of the set corresponds to the path in the de Bruijn graph.

### 1.2.3. Simplitigs

Another method of de Bruijn graph representation is called simplitigs. This method was developed by Břinda et al. [2021] and appeared after the unitigs. It is a generalization of unitigs and represents the spelling of vertex-disjoint paths that span the graph. While unitigs focus on individual paths that meet the criteria of each node having in-degree and out-degree equal to one (Figure 1.1), simplitigs aim to extend and combine these paths to form longer sequences (Figure 1.2). This merging process allows for the representation of longer continuous sequences.

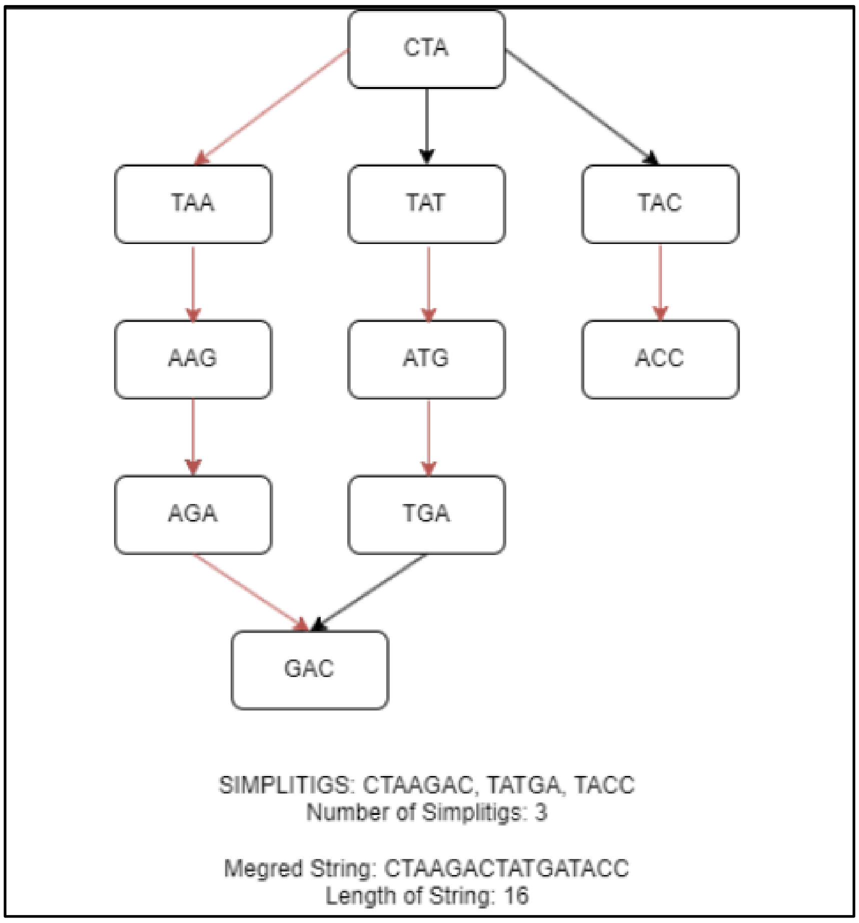

*Figure 1.2: Representation of k-mer sets as simplitigs*

However, compared to unitigs, simplitigs do not carry the information about the de Bruijn graphs topology, since they do not represent each path in the graph and instead represent their concatenations. However, that does not affect the presence of all of the k-mers in the simplitigs and they can be derived from it back.

Nevertheless, the loss of topology is justifiable as simplitigs represent longer sequences, which when merged are shorter than the result of the unitigs concatenation. For example, for the same de Bruijn graph (Figure 1.1 and Figure 1.2) merged unitigs output the string with 20 characters, while simplitigs with 16. Such representation is less memory consuming and allows to save more memory space.

The problem of finding simplitigs in the de Bruijn graph is equivalent to the problem of finding a smallest spectrum-preserved set (SPSS) representation. Introduced by Rahman and Medvedev [2021] algorithm UST (Unitig-STitch) works on the node-centric de Bruijn graph (the definition of de Bruijn graph which we work with), constructed from the input sequences. The algorithm finds the paths in such a graph, starting in each iteration with an arbitrary node, and tracks all the nodes that were visited by the path. Whenever the extension of the path reaches a dead-end, such a path is stored and the neighbors of the last node in the path are checked if such exist. If any of the successor nodes is in another path, such that it is a start of the corresponding path, then the obtained path and the path of the successor are merged together. After that the algorithm chooses another arbitrary node and continues the process, until all of the nodes are visited, and outputs the set of strings.

Another efficient algorithm of simplitigs computation is called ProphAsm and was developed by Břinda et al. [2021]. Compared to the USP algorithm, this method does not require construction of de Bruijn graphs and operates directly on the k-mer set. This algorithm is discussed in 1.3.

Described methods do not provide optimal simplitigs as a solution. Addressing this limitation, a polynomial-time algorithm has been developed by Schmidt and Alanko [2022], designed to find optimal simplitigs. This algorithm is based on the concept of Eulerian paths and cycles in the de Bruijn graph. The directed graph is called *Eulerian* if all nodes have indegree equal to the outdegree. And a bigraph is Eulerian if all nodes have imbalance (difference between indegree and outdegree) 0. The *Eulerian Cycle* is the cycle that visits each edge exactly once.

The idea of such an approach is firstly to construct a bidirected de Bruijn graph in linear time and then add breaking edges into this graph to make it Eulerian. This step is done to ensure that every node in a graph has an equal number of ingoing and outgoing edges. Next the algorithm computes the Eulerian cycle in such a graph and then breaks the obtained cycle at the breaking edges added earlier. The output of such an algorithm is a set of strings, spelled by the resulting walks.

### 1.2.3. Matchtigs

Another method for representation of the de Bruijn graph is called matchtigs, which was introduced by Schmidt at el. [2023] and which concept is closely related to simplitigs. While simplitigs represent vertex-disjoint paths through the graph, matchtigs can include overlapping paths and do not follow the vertex-disjoint criteria. Due to these properties same sequences of k-mers can be included in multiple matchtigs, leading to a potentially more space-efficient way of representation, compared to simplitigs. It is well seen in the example of Figure 1.3, where in Figure 1.3.b k-mers are represented without repetitions, leading to the 7 output sequences with total number of characters being 43, while in Figure 1.3.d, where the repetition of k-mers is allowed, the number of sequences is 5 and the number of characters is 39.

The algorithm for finding matchtigs is similar to the UST algorithm. However, the matchtigs algorithm allows repeated k-mers and uses a different definition of de Bruijn graph, which is edge-centric compacted graphs (Figure 1.3.a). The *edge-centric de Bruijn graph* is a bidirected graph *G = (V, E)*, where each edge represents k-mer, i.e. *E* = set of k-mers and each vertex represents *k-1* long overlap between k-mers, i.e. two edges share the vertex if associated with them k-mers are overlapped by *k-1* characters. The *compacted de Bruijn graph* is a simplification of the standard de Bruijn graph and obtained by merging the paths of nodes with indegree and outdegree of 1 into a single node.

Matchtigs greedy algorithm explores such graphs to find the different paths. In edge-centric de Bruijn graphs, matchtigs are not necessary to be edge-disjoint, meaning that two distinct paths can share a common edge. However, the beginning of the path must be an unvisited edge to ensure that it covers a new k-mer matchtig. As such a path is found, all corresponding edges are marked as visited and the traversal is continued from the next unvisited edge. The process is repeated until there are no more unvisited edges and outputs the set of matchtigs.

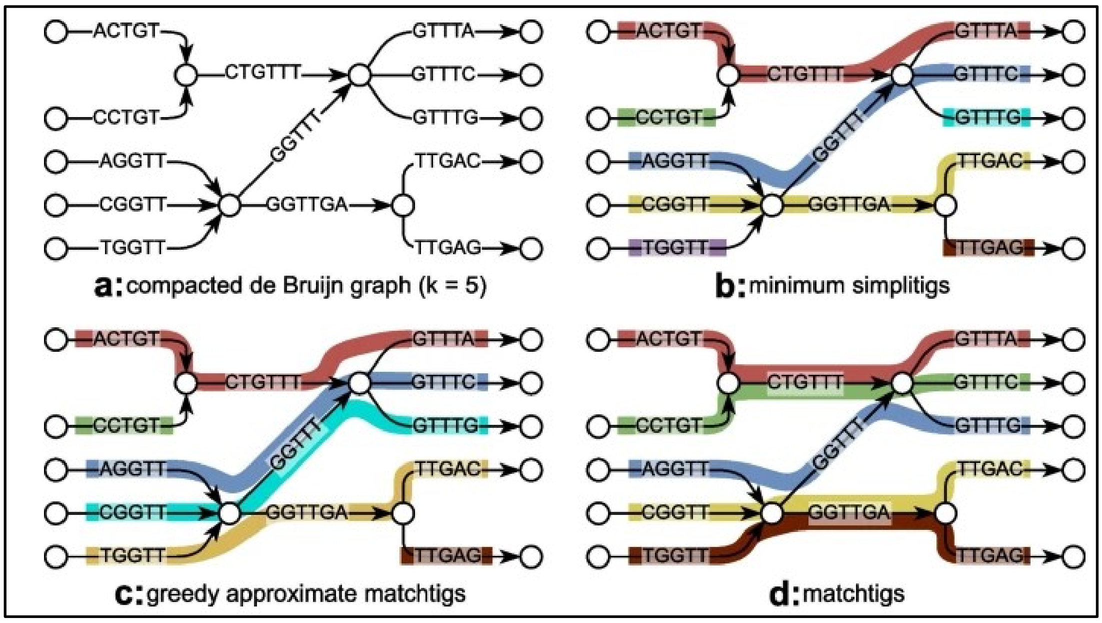

*Figure 1.3: Computations on a edge-centric de Bruijn graph with k = 5. Credits: Schmidt et al. [2023]*

As this greedy approach produces only close to optimal solutions, Schmidt et al. [2023] introduced a more involved and less efficient polynomial time algorithm for finding optimal matchings. If we compare Figure 1.3.c and Figure 1.3.d, which are examples of differently computed matchtigs, the greedy approximate matchtigs has 6 strings and 40 total characters, while optional matchtigs has a reduced number of 5 strings and 39 characters.

This method starts from construction of edge-centric compacted de Bruijn graph and computation of bi-imbalance (difference between outdegree and indegree) for each of the nodes in the graph.

After this the min-cost paths are computed: from each node with negative bi-imbalance the algorithm finds the paths to all reachable nodes with positive bi-imbalance, such that it will require the smallest cost to traverse. The cost of an edge is defined as the number of characters required to join two strings from the negative to the positive node. After obtaining all of such paths the min-cost matching is applied on them in order to decide which bi-imbalances should be fixed by repeating k-mers. Some nodes will stay unmatched, indicating that fixing their bi-imbalance would require breaking edges. That step is done to make the graph balanced (for each node indegree is equal to outdegree).

The algorithm then finds a path that visits every edge in both directions (biEulerian circuit). This path is broken down into separate paths at breaking edges, which were added by the previous step. The output of this method will be the string spelled by the broken paths.

## 1.3. Simplitigs Algorithm

The greedy algorithm, which was chosen for implementation in this specific project, is introduced by Břinda et al. [2021], who also provided its implementation as a tool ProphAsm for computing simplitigs.

This algorithm does not construct the de Bruijn graph and collects all k-mers into the hash table. It chooses an arbitrary k-mer as the seed of a new simplitig and extends it backwards and forwards as much as it can while deleting previously used k-mers from the set. The extension continues by adding one of four nucleotides at a time if the resultant k-mer is present in the set of k-mers. Until all k-mers are covered, this process is repeated.

The ProphAsm algorithm functions by combining overlapping k-mers into a single sequence called simplitigs, which leads to faster computation by reducing the number of strings and total number of characters in the strings required to store k-mer sets. Such an algorithm is very time-efficient; however, it does not find the optimal solution.

Note, that the detailed pseudocode for this algorithm is provided in Attachments, A.2.

### 1.3.1. Time Complexity Analyze

The time complexity of the algorithm depends on the size of the input k-mer set K and the k-mer length k.

Extension backwards and forwards have linear time complexity of *O(k)*, since for each k-mer, there are four possible nucleotides. In the worst-case scenario, this function would be called once for both directions, i.e. once for each additional k-mer, added to the path, implying that overall there will be *k-1* iterations.

The computation of the simplitigs calls the extension for every k-mer in the input set K. Thus the time complexity is *O(Nk)* where *N* is the number of kmers in the set.

# 2. Shortest Superstring Problem

## 2.1. Introduction to Shortest Superstring Problem

Given a collection of strings $S = \{s_1, \ldots, s_n\}$ over an alphabet Σ, a *superstring* $\alpha$ of $S$ is a string containing each $s_i$ as a substring, that is, for each $i$, $1 \leq i \leq n$, α contains a block of $|s_i|$ consecutive characters that match $s_i$ exactly.

For instance, given the set S = {“abcd”, “aebc”, “cdaeb”, “ecba”, “bca”} over the alphabet {‘a’, ‘b’, ‘c’, ‘d’, ‘e’} a particular superstring could be “ecbabcdaebca”.

Recalling the definition of overlap from the Preliminaries, overlaps are an essential concept in the construction of a superstring since they allow merging of the substrings in the correct order. For example, given two strings "AACT" and "CTTA", the overlap between them is "CT", and the merge of them is "AACTTA".

As overlaps represent the common subsequences between two strings in the given set, they indicate the potential connections between the strings. By identifying these overlaps, we can merge the strings together to form a superstring.

Finding the shortest superstring for a given set of strings is the goal of the *shortest superstring problem* (SSP). SSP is a well-known computational problem in computer science and is known to be NP-hard (Garey and Johnson [1979]), implying that finding the exact optimal solution in polynomial time is computationally challenging.

## 2.2. Greedy Algorithm for SSP

Since SSP is an NP-hard problem and it does not have an efficient algorithm that can solve it optimally in polynomial time, approximation algorithms are used instead to find near-optimal solutions. Approximation algorithms provide an approximate solution, whose quality compared to the optimal solution is measured by the approximation ratio. The *approximation ratio R* is calculated as follows: $R = \frac{A}{OPT}$, where $A$ is the value of the approximate solution and *OPT* is the value of the optimal solution. Smaller approximation ratio corresponds to a more accurate approximation. In the sense of the Shortest Superstring problem the quality of the approximation algorithm is evaluated in terms of the length of the resulting superstring.

A greedy approximation algorithm for SSP is one of such approaches and was introduced by Gallant [1982]. This algorithm makes locally optimal choices at each iteration in order to reach an approximate to optimal solution. The idea behind this greedy algorithm is to merge the pair of strings with the maximum overlaps one by one, until all the strings are merged into a single superstring.

For instance for the set of strings *S* = {“AGCT”, “CTAA”, “TAAC”, “CTTG”} the algorithms works as follow:

1. Strings “CTAA” and “TAAC” are merged, as they have the overlap of size 3, which is the largest among any other possible overlaps between any other two strings. After the merge the set is updated as {“AGCT”, “CTAAC”, “CTTG”}.
2. Strings “AGCT” and “CTTAC” are merged, as they have the overlap of size 2, which is the longest overlap among others. The updated set of strings is: {“AGCTAAC”, “CTTG”}.
3. The last two strings are merged together and the output of the algorithm is the superstring “AGCTAACTTG”.

Tarhio and Ukkonen [1988] stated the greedy conjecture, which says that the greedy algorithm has approximation ratio 2. That means that the length of the resulting approximate superstring is at most twice the length of the optimal superstring. In addition, the best known upper bound on the approximation ratio of the Greedy algorithm was introduced by Englert, Matsakis and Veselý [2021] and stated to be approximately 3.425.

## 2.3. Overlap Graph

While the greedy approximation algorithm for SSP is effective, it is important to note that there exist alternative approximation algorithms that offer improved guarantees. These algorithms leverage the concept of overlap graphs to achieve more accurate solutions.

A directed graph *G = (V, E)* is called the *overlap graph* for a set of strings S if there is a one-to-one correspondence between *V* and *S* such that two vertices in *V* are adjacent to each other if and only if the corresponding strings in *S* overlap each other. Each edge in the overlap graph is assigned with the weight, which corresponds to the length of the overlap between two strings associated with adjacent vertices. As the graph is directed the direction of the edges between the vertices matters in order to assign them with the proper weight.

For instance, consider nodes *s*, which corresponds to string “ACCTG”, and *t, corresponding to* “CTGCA”. The edge from *s* to *t* has a weight of 3 as the overlap between *s* and *t* is “CTG”. However, the edge from *t* to *s* has a weight of 1, since the overlap between t and s is “A”.

The overlap graph represents self-overlaps as well. Self-overlap occurs when the string in *S* overlaps with itself, for example, string “ACCAC” has self-overlap of length 2. The overlap graph captures such overlaps as edges connecting a vertex to itself.

To construct the overlap graph, each ordered pair of the strings in the input set are compared. If there is an overlap between two strings, the edge between their corresponding nodes is added and the weight, associated with the edge, corresponds to the length of the overlap.

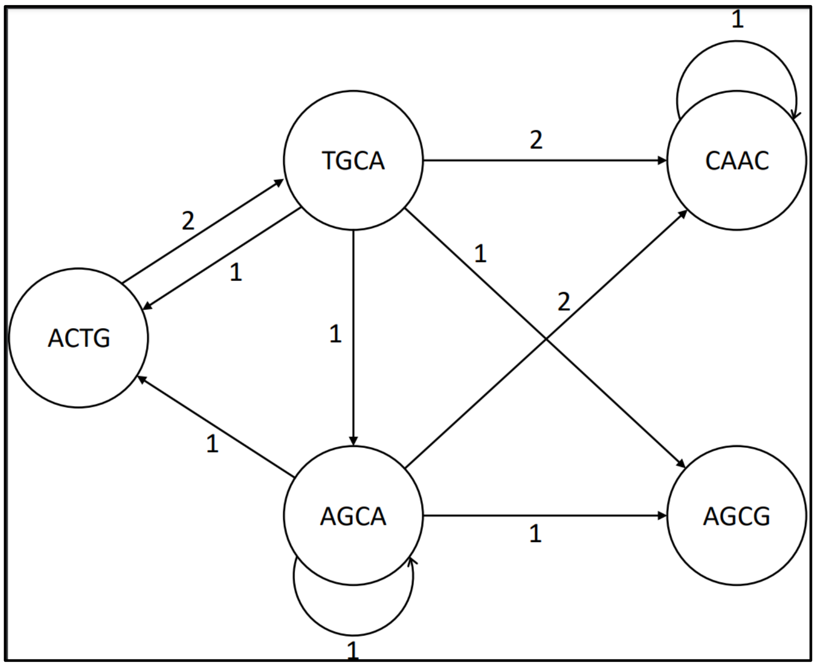

*Figure 2.1: Overlap graph constructed for the set of strings {“ACTG”, “AGCG”, “AGCA”, “CAAC”, “TGCA”}*

The overlap graph is closely connected to the Shortest Superstring Problem. As SSP aims to find the shortest possible superstring, containing all of the input strings, the overlap graph visualizes the relationships between the strings and identifies the potential merges between them.

Shortest superstring could be obtained from the overlap graph by finding the largest overlap Hamiltonian path in this graph. The *Hamiltonian Path* is a simple path in a graph, which visits every vertex exactly once. With the use of algorithms, such as depth-first search

(DFS), such a path can be searched in the overlap graph and be equivalent to the valid solution of the SSP. Nevertheless, it may not be the optimal solution in terms of minimizing the length of the resulting superstring as different Hamiltonian paths can lead to different superstring lengths. Finding the optimal Hamiltonian path, which corresponds to the largest overlap, is known to be an NP-complete problem.

## 2.4. Linear-time implementation of Greedy (Greedy AC)

Ukkonen [1990] developed a linear-time implementation of the greedy approximation algorithm, which is based on the idea of greedy heuristics for finding a longest Hamiltonian Path using the modified Aho-Corasick string-matching machine.

### 2.4.1. Aho-Corasick Automaton

A *trie* is a tree-like data structure that stores a set of strings by encoding the common prefixes among them. In a trie, each node represents a prefix of one or more strings, and each edge represents a character that extends the prefix to a new node

Developed by Aho and Corasick [1975] *Aho-Corasick automaton* is a trie-based data structure that efficiently matches multiple patterns over the input string. The automaton consists of a trie that represents the set of patterns and a set of additional functions that enable efficient traversal of the trie. The goto function transitions (solid lines in Figure 2.2) from a state to another state based on the next input character, while the fail function (dashed red lines in Figure 2.2) enables backtracking to a previously matched prefix of a pattern in case of a mismatch.

The *goto function* is computed using the trie data structure, where each node represents a state and each edge represents a transition. The goto transition generally represents the way an automaton moves and is denoted as $g(s, a) = t$ if there is a transition between from state $s$ to $t$ for character $a$.

The *fail function* of the Aho-Corasick automaton maps each state to the longest proper suffix of a pattern that matches a prefix of an input string. The failure transition is a so-called e-move which does not consume any symbol from the string scanned and is denoted as $f(s) = t$ if there is a failure transition from state $s$ *t*o state $t$.

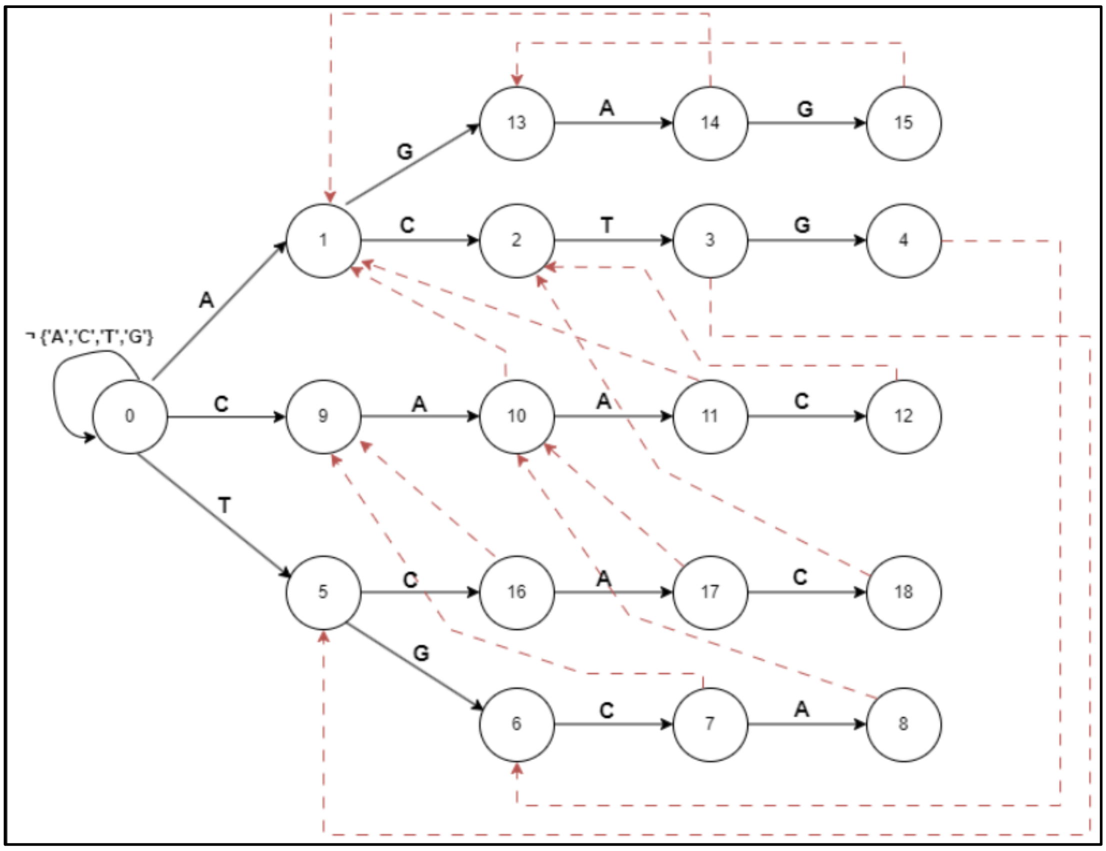

*Figure 2.2: Aho-Corasick automaton constructed from the set of words {"AGTG", "TGCA", "CAAC", "AGAG", "TCAC"} over alphabet {'A', 'C', 'T', 'G'}*

### 2.4.2. Greedy_AC Algorithm

Let *S* be a finite set of strings over the alphabet Σ. The idea of the greedy algorithm is to find and remove two strings in *S* that have the longest overlap among all possible pairs of strings in *S*. Then form the overlapped string from the removed two strings and replace it back in *S*. Repeat until there is only one string in *S* or no two strings have a nonempty overlap.

The implementation of the algorithm consists of two functions: the first one is a preprocessing phase that augments the usual AC machine with the necessary information and the second one implements the greedy heuristics.

The function preprocessing takes a set of k-mers *S* and Aho-Corasick machine with constructed *goto* and *fail* functions for *S*. As the result function outputs several structures:

1. *list_L* – specific dictionary where each key is assigned with a list of values. The key is the state and the list of values are the indices of the words from the input

set *S* that have the state as prefix. *List_L* allows to keep track of the supporters for each state, i.e. it records index for each word passing through each state.

2. *state_F* – dictionary with keys representing the indexes of words in *S,* where the corresponding value is the final state for such a word, i.e. state where this word ends. In other words this dictionary represents the strings of set *S*.
    a. *inverse_E* – dictionary, such that if *state_F(i) = s* then *inverse_E(s) = i.*
3. Reverse breadth-first ordering of the states.
    a. *link_b* – dictionary, where key is the state and its value is successor of the state (backlink of state), i.e. contains the reverse breadth-first search.
    b. *pointer_B* – integer, which represents the first state in such linked list

By the theorem, proposed and proved by Ukonnen [1990], the AC machine of *S* over a small enough alphabet Σ can be constructed and preprocessed with the preprocessing algorithm in time *O(n)*, where *n* denotes the total length of the strings in *S*.

After the preprocessing function the next step is the greedy selection of the overlaps. The selected overlaps will form a Hamiltonian path *H* in the overlap graph.

The algorithm of function Hamiltonian traverses the states of the AC machine in the reversed breadth-first order, following the *link_b* dictionary. During this traversal, the algorithm maintains a dictionary, denoted as *list_P*, where for each state *s* there is a list of all indices *i* such that:

1. For each index *i* in *list_P(s)* the state *s* is encountered while following the failure path starting from *state_F(i)* state. In other words, the i-th string has the state s as a suffix.
2. The Hamiltonian path *H* does not contain an overlap$(x_i, x_j)$ for any *j*.

The algorithm then checks for overlaps $(x_i, x_j)$ that satisfy two conditions:

1. *H* does not contain an overlap starting at $x_i$ and ending at $x_j$
2. Adding $(x_i, x_j)$ to *H,* along with the existing overlaps, does not create a cycle

For each *j* in *list_L(s)* which satisfies the first condition (this is checked by dictionary *forbidden*), the algorithm traverse *list_P(s)* until the first index *i* such that the overlap $(x_i, x_j)$ satisfies the second condition.

To efficiently check the second condition in constant time the algorithm maintains dictionaries *first* and *last*. *First(i)* table keeps track of the first occurrence of *i* in the

Hamiltonian path, i.e. it represents the first node in the path where *i* appears. *Last(i)* stores the last occurrence of the *i*, i.e. indicates the last node in the path.

Initially set as *first(i) = last(i) = i* for all patterns, the dictionaries are updated at any moment when *H* contains a path from $x_i$ to $x_j$ and no arc in *H* ends at $x_i$ or starts at $x_j$. In such cases *fist(j) = i* and *last(i) = j.*

After processing all of the overlaps at state *s*, the algorithm concatenates *list_P(s)* with *list_P(fail(s))*, where *fail(s)* is the failure state of s. This allows the algorithm to continue processing the failure paths that pass through state *s* when the traversal reaches *fail(s).*

Once finished with the state *s* the algorithm moves on to *list_b(s)* and the backlink directs the algorithm to the next state to be processed.

After the Hamiltonian path H is found, a helper function findSingle is called to find all nodes of an AC machine with in-degree 0 or with large self-overlaps. Such strings are not in the Hamiltonian path, but must be taken into the account. After obtaining such a list, all the strings from the initial set *S* are merged together following the Hamiltonian path.

By another theorem of Ukkonen [1990], the Hamiltonian algorithm runs in *O(n)*. By combining both of the theorems the conclusion is derived: the time complexity of the Greedy_AC algorithm is linear and *O(n).*

The detailed pseudocode for this algorithm is provided in Attachments, A.2.

## 2.5. TGreedy

As described in Greedy AC, the Greedy algorithm finds the shortest common superstring for the set of words *S* by repeatedly merging two strings with the maximum overlap until a single string remains. Such algorithm sorts edges by overlap following two conditions:

1. Hamiltonian Path *H* does not contain an overlap starting at $x_i$ and ending at $x_j$
2. Adding $(x_i, x_j)$ to *H,* along with the existing overlaps, does not create a cycle

Such a Greedy algorithm has two variants which ignore the second condition and introduce cycle covers.

### 2.5.1. Cycle Cover

A *cycle cover* in a complete directed weighted graph *G* with self-loops is a set of directed cycles such that the inner degree and the outer degree of each node of *G* are both unit, i.e. each node has exactly one incoming and one outgoing edge. A cycle cover ensures that each node in *G* is included in some cycle. The example of the cycle cover, compared to the Hamiltonian path, is presented in Figure 2.3 as green and red lines respectively.

The concept of cycle cover is quite useful in the context of the shortest common superstring problem. By the construction of cycles, we ensure that each string is included in the superstring while minimizing the overall length of the resulting string.

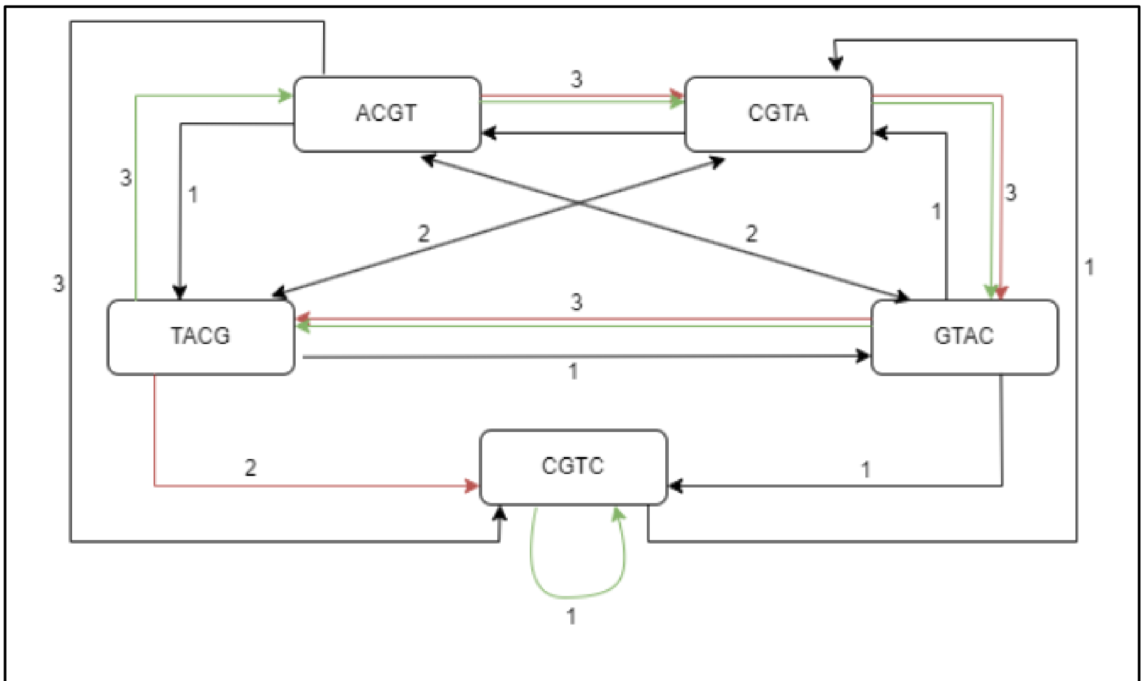
ACGT
CGTA
TACG
GTAC
CGTC
3
3
1
2
2
1
3
3
1
3
1
2
1
1
Maximum Hamiltonian Path: ACGT - CGTA - GTAC - TACG - CGTC
Maximum Cycle Cover: {ACGT - CGTA - GTAC - TACG - ACGT, CGTC}

*Figure 2.3: Maximum Hamiltonian Path (red lines) and maximum Cycle cover (green lines) on the overlap graph constructed for the set of strings {"ACGT", "CGTA", "GTAC", "CGTC", "TACG"}*

### 2.5.2. MGREEDY

One of such Greedy versions is called *MGreedy* introduced by Blum et al [1994]. The goal of this algorithm is to compute an optimal cycle cover by sorting edges from the overlap graph based on their overlap lengths. While iterating over the sorted list of edges, the edge *(s,t)* is added to the cycle cover if no edge *(s, t')* or *(s', t)* has been chosen before *(s,t)*.

The resulting cycle cover obtained from the MGreedy algorithm contains a set of directed cycles that collectively cover all the nodes in the overlap graph. Each cycle represents a sequence of strings with maximum overlaps. In order to get the representative string of the cycle, it must be broken at the smallest overlap edge and the strings merged among the obtained after edge-removal path. The resulting superstring is a concatenation of all representative strings in some arbitrary order.

By Blum's [1994] theorems and observations MGreedy algorithm is of length at most $4 \cdot OPT$. However, a better upper bound of the approximation ration is introduced by Englert et al. [2021] and is equivalent to approximately 3.425.

### 2.5.3. TGREEDY Algorithm

The second version introduced by Blum et al [1994], named *TGreedy* algorithm, is an improvement over the MGreedy algorithm and is used to further optimize the cycle cover. The algorithm operates on the set of representative strings *M* obtained from MGreedy and instead of concatenating them in the arbitrary order, it applies Greedy on the set M in order to merge them by overlaps.

The implementation of TGreedy is an improved version of the Greedy_AC algorithm, specifically the Hamiltonian function. Such an algorithm is the first linear-time implementation of TGreedy and the detailed pseudocode for this approach can be found in Attachments, A.2.

HamiltonianT function takes as input the preprocessed data structures, obtained by preprocessing function, which is the same as for Greedy_AC.

The function's initialization and first loop, which iterates over each k-mer and adds it to the *list_P* dictionary, described in 2.4.2.

Set the initial state to the value of *pointer_B*. Then the function enters the while loop that is executed until the state will be 0, which will signify the completion of the optimal cycle cover, instead of the Hamiltonian path as in Greedy_AC. That is why this part of the original algorithm is extended by introducing a cycle cover construction step.

In order to find such a cycle cover, the code checks a specific case where the word index *first[i]* is the same as the word index *j*.

1. The condition *first[i] == j* checks if the last element in the current merging chain, represented by *first[i]*, is the same as the element *j*.
2. If *first[i] == j*, it means that the current element *j* is already part of the merging chain, creating a cycle. In other words, *j* is a predecessor of itself in the merging chain.
3. To prevent this cycle, the code removes *i* from the dictionary *list_P*, excluding it from further consideration in the merging process.
4. Additionally, it sets *forbidden[j]* to *True* to mark the word *j* as forbidden, indicating that it cannot be merged with any other word anymore.
5. Since the condition has been satisfied the loop breaks and moves to the next word in *list_j* corresponding to *list_L*

By the theorem proposed and proved by Blum et al. [1994], algorithm TGreedy produces a superstring of length at most $3 \cdot OPT$. However, by Englert et al. the better upper bound for the approximation ration was proven to be approximately 2.7125.

The proof of the linear time complexity by Ukkonen still applies to this variant of the Greedy algorithm as the change in the Hamiltonian function does not affect the time complexity.

# 3. Connection between k-mers and SSP

## 3.1. Linking k-mers representation problem and SSP

The goal of approaches, described in Chapter 1, such as unitigs, simplitigs or matchtigs, is to represent the k-mer set as a compact set of sequences, which covers all of the k-mers. Such representation is quite useful for further applications due to its fast calculation, but does not provide the storage of the k-mers with the optimal size of representation. That is a result of the limitation of such approaches by the usage of de Bruijn graphs. The de Bruijn graph prevents the usage of overlaps shorter than *k-1* since in such a graph the edge between two nodes is presented only if there is an overlap of size *k-1* between them.

When considering traditional k-mer set representation algorithms, we derive the observation that when output sequences obtained by these algorithms are merged together, they construct a superstring of k-mers. According to the observation, SSP algorithms offer an alternative for obtaining an efficient representation of k-mer sets. By taking a set of strings as input, these algorithms can effectively handle the set of k-mers, which consists of fixed-size strings over a specific four-character alphabet.

Unlike in traditional approaches, k-mers are represented by the overlap graph, which provides information about all possible overlaps between k-mers, implying that overlaps of size shorter than *k-1* can be analyzed. Instead of starting with an arbitrary k-mer and attempting to extend it as much as possible, SSP algorithms begin with the largest identified overlap and merge the strings accordingly. This leads to the discovery of a shorter and more memory-efficient representation of the strings.

### 3.1.1. Mask

However, it is important to note that representing a k-mer set as a superstring alone is not sufficient. While it serves the purpose of storing the k-mer set efficiently, it does not provide the necessary information for proper extraction and identification of the original k-mers. This happens due to the appearances of false positives in the superstring. For instance, given set of k-mers {"ACG", "ACC", "GAC", "CTA"}, the obtained superstring "ACGACCTA" introduces false k-mers such as "CGA" and "CCT", which are not present in the original set.

In order to identify such false positives, an identification mask must be applied to the resulting superstring, allowing for the accurate retrieval and identification of individual k-mers. Identification mask is a binary string, which may be represented in various ways and can be possibly compressed.

One approach involves using masks in the context of the de Bruijn graph. In this approach, sequences of unitigs or simplitigs are separated by the comma delimiters. As each of such sequences represent k-mers merged with each other with *k-1* overlap, original k-mers are easily derived from such sequences and the comma between them prevents the generation of the false k-mers. The example of such a mask is presented in Figures 1.1 and 1.2.

Alternative type of mask is *case sensitive mask*, which converts specific characters in the superstring to lowercase letters in order to highlight the location of the original k-mers.

For example, consider set *S* = {“ACG”, “CGT”, “TAC”} and superstring “ACGTAC”. Applying the case-sensitive mask would yield the modified superstring “ACgTac”, where the upper letters indicate the positions of the original k-mers.

Another type of mask tracks the position of the original k-mers in the superstring by providing a list of indices. For the same set *S* and superstring, the position mask would be the list {0, 1, 3}, with each integer representing the index of the start of an original k-mer in the superstring.

A *binary mask* is another type of mask that offers a more compact and machine-readable representation of the superstring. In this approach, each character is encoded as a binary digit: *1* indicates the presence of a k-mer, while *0* indicates its absence. The binary mask provides a concise representation of the superstring.

Continuing with the set *S* = {“ACG”, “CGT”, “TAC”} and superstring “ACGTAC” the encoding will be “110100”.

## 3.2. Utilizing SSP Algorithms for K-mers

In order to efficiently represent k-mer sets, we will utilize the algorithms for SSP with the modification. Given a set of strings of fixed size *k*, the greedy algorithms would perform as usual with minor changes. The small size of the alphabet, which consists of only four symbols, makes it easier for the implemented greedy algorithms to navigate through the automaton during the construction of reverse breadth-first ordering of the states.

The mask plays a crucial role in marking k-mers in the output superstring of the greedy algorithms. In this project, we provide two options for the mask: case-sensitive and binary mask. The former is applied on the superstring by default and the latter could be called with the flag –b.

Both implemented mask-applying functions work in a similar manner. They take as input superstring, k-mer set and value of *k*, and iterate over each character (except for the last *k-1*) of the superstring, check whether or not the k-long substring in the superstring is in the k-mer set. This comparison allows for the identification of the existence of original k-mers and false positives.

Several observations can be made from the implementation of these mask functions. Firstly, the length of the mask is always equal to the length of the superstring. This holds since the case-sensitive mask only modifies the case of characters, and the binary mask produces a sequence where each 1 or 0 correspond to a specific character in the superstring.

Secondly, the mask functions do not need to iterate over the last *k-1* characters of the superstring as the length of the k-mers is fixed, and no k-mer can have a length less than k. Therefore, the last k-1 characters in the mask are always lowercase or 0, depending on the chosen option.

Finally, as all k-mers are merged into the superstring, it is trivial that the number of positions of k-mers in the masked superstring cannot be smaller than the size of the k-mer set. However, it is not necessary to be exactly equal to the number of original k-mers. During the construction of the superstring, overlaps between different k-mers may result in the creation of other k-mers that are not false positive, but are repetitions of other original k-mers. In such cases, the mask functions mark each occurrence of any original k-mer in the superstring.

# 4. Experiments

The source code of implemented algorithms and program for testing are in the git repository: https://github.com/Ekatmil/Efficient-representation-of-k-mers-sets

## 4.1. Objective

The primary goal of this research revolves around the comparison of three implemented algorithms - Simplitigs, Greedy_AC and TGreedy - in terms of their efficiency and performance in representing k-mer sets.

The main aim is to conduct comparison of these different approaches across two main dimensions. First, the length of the resulting superstrings, which is a key measurement for the ability of each algorithm to create efficient representation of k-mer sets. Second, the computational efficiency of each approach, such as CPU time and peak memory usage, provides a better understanding of each algorithm's performance under actual computation.

In addition to evaluating each algorithm individually, we will compare them in pairs as well. The performance of the utilized for representing k-mers SSP greedy algorithms (Greedy_AC and TGreedy) will be contrasted against the Simplitigs method. Furthermore, Greedy_AC and TGreedy will be compared in order to highlight the practical difference between implementation of Hamiltonian Path and Cycle cover computations.

The results of this comparison will offer valuable insights into the advantages and limitations of each algorithm, guiding future efforts in optimizing k-mer set representation in bioinformatics research and applications.

## 4.2. Setup

### 4.2.1. Programming Environment and the Computational Resources

All of the algorithms, along with the overall program, are written in Python, utilizing the BioPython package. Python, known for its simplicity and versatility, was the choice for developing the prototype of this program. Python is not adept in dealing with large-scale bioinformatics data due to its memory usage characteristics. When working with large data sets, Python's memory management significantly affects the performance. For the future iterations of the project, it would be advantageous to rewrite and retest it with usage of a more performance-focused language like C++, which would allow more optimal memory usage.

The computational environment for this experiment was a server equipped with 3.3 GHz AMD EPYC 7302 processor and 251 GiB of RAM. Such a setup offers significant computing resources necessary for processing all algorithms.

### 4.2.2. Input Values

Data for the experiment was stored in FASTA files. FASTA is a text-based file format widely utilized in bioinformatics for representing nucleotide sequences. A FASTA formatted file begins with a single-line description, followed by lines of sequence data. The description line is distinguished from the sequence data by a greater-than (">") symbol. For instance:

```
>sequence_0

GTGTCGGAGGCTCCATCGACATGGAACGAGCGGTGGCAAGAAGTTACTAATGAGCTGCTGTCA
CAGTCTCAGGACCCGGAAAGTGGTATTTCCATTACGCGACAGCAAAGCGCCTACCTG

>sequence_1

CAATACGGAGCAACTTCAGCCAATGCTGACTTCCAGAATCAACAAAGCACGATATA
```

For our experiments we use two input files:

1. *spneu.fa* (2.02 MB): This file contains the assembled genome of one S. pneumoniae individual. S. pneumoniae is often considered a standard model species for experimental evaluation of k-mer-based methods.
2. *spneumo_pangenome_k32.fa* (14 MB): This file represents the pangenome of S. pneumoniae, incorporating 616 assemblies found during a vaccination study of children in Massachusetts, USA, described by Croucher et al. [2009]. Pangenomes provide a broader genetic context, capturing the genetic diversity of a species.

The selected range for the value of *k* was chosen to be from 8 to 20 for the *spneu.fa* file and from 8 to 16 for the *spneumo_pangenome_k32.fa* file. The primary reason for this limited range stems from the prototype nature of the approach. Given this current limitations, the implemented greedy algorithms (Greedy_AC and TGreedy) cannot efficiently handle computations involving large datasets, often leading to memory usage failures. These constraints lead to a more conservative choice in the k-value range to ensure viable and meaningful analysis.

Moreover, the selected range of 8 to 16 or 20 is particularly relevant for the pneumococcus. For values of k less than or equal to 8, the de Bruijn graph is almost complete, meaning nearly all possible k-mers are present, amounting to almost $4^k$. On the other hand, as k increases beyond this range, the de Bruijn graph quite simplifies and Simplitigs generate near-optimal results.

### 4.2.3. Data Collection

A Python function *outputStats* is implemented to collect and store the important information about efficiency and performance of the used algorithms. The function is designed to capture the essential data required to evaluate the performance of three algorithms under different *k* values.

The *outputStats* stores several parameters into the CSV file, which allows easy manipulation and analysis of the obtained data.

The list of stored data:

1. name of the input fasta file
2. name of the output file (if was not selected, than stdout)
3. type of the applied mask
4. name of the applied algorithm
5. value of *k*
6. size of k-mers set
7. length of the resulting superstring
8. the CPU time consumed by the algorithm
9. the actual elapsed time (wall-clock time) taken by algorithm
10. the maximum (peak) memory used by the algorithm during its execution

The collection of this statistical data is called with the flag –S or – stats. The detailed pseudocode for this algorithm is provided in Attachments, A.2.

To streamline the process of gathering data for various values of k and different algorithms, an efficient bash script (*run_python.sh*) was implemented. This script automates the execution of the Python program for different k-mer sizes and algorithms, capturing and saving the outputs and statistics in an organized manner.

## 4.3. User Manual

This subchapter provides a user guide for executing the Python script main.py, enabling independent replication of all experiments.

In order to run the code user required to have:

1. Python3 installed in their device
2. Installed BioPython package (https://biopython.org/) , which is required for manipulations with fasta files.

BioPython can be installed on Linux via command lines:

*'conda install -c conda-forge biopython'* or '*pip install biopython'*

Step 1: Download the script main.py and all of its dependencies into one directory.

*List of dependencies*: Load_fasta.py; mask.py; Statistics.py; testStr.py; tgreedy.py; Greedy_AC.py and simplitig.py

Step 2: Open the Command Line Interface on Linux system and run the script as follows:

*python3 main.py –i <input_file>*

or p*ython3 main.py --input <input file>*

Replace <input_file> with FASTA formatted input file.

Step 3: Choose one of the algorithms, which will run on the k-mers set, obtained from <input_file>

*List of algorithm flags*:

-s/ --simplitig – call for simplitigs algorithm

-a/ --aho-corasick – call for Greedy_AC algorithm

-t/ --tgreedy – call for TGreedy algorithm

Step 4: Add additional flags into command line in order to change and/or check the output

*List of additional flags:*

-k K/ --kmer K – the length of each kmer. Not required. Default K is 31

-b/ --bitstring_mask – mask will be saved in the form of 1-0. Not required. Default mask is case_sensetive

-o <output_file>/ --output <output_file> - the output will be saved into <output_file>. Not required. Default output is standard output

-T/--test - runs tests on the output superstring and provides the results as standard output. Not required.

-S <stats_file>/ --stats <stats_file> - stores statistics to the given <stats_file>. Not required

Step 5: Check the Output. The output will be either printed into stdout or written to a file, depending on whether the -o flag was used in the command.

## 4.4. Results

The results are presented as charts for the visual comparison between different approaches. More detailed statistics can be found in Table 1 and Table 2 in Attachments, A.1.

## 4.4.1. Comparison of Superstrings' Lengths

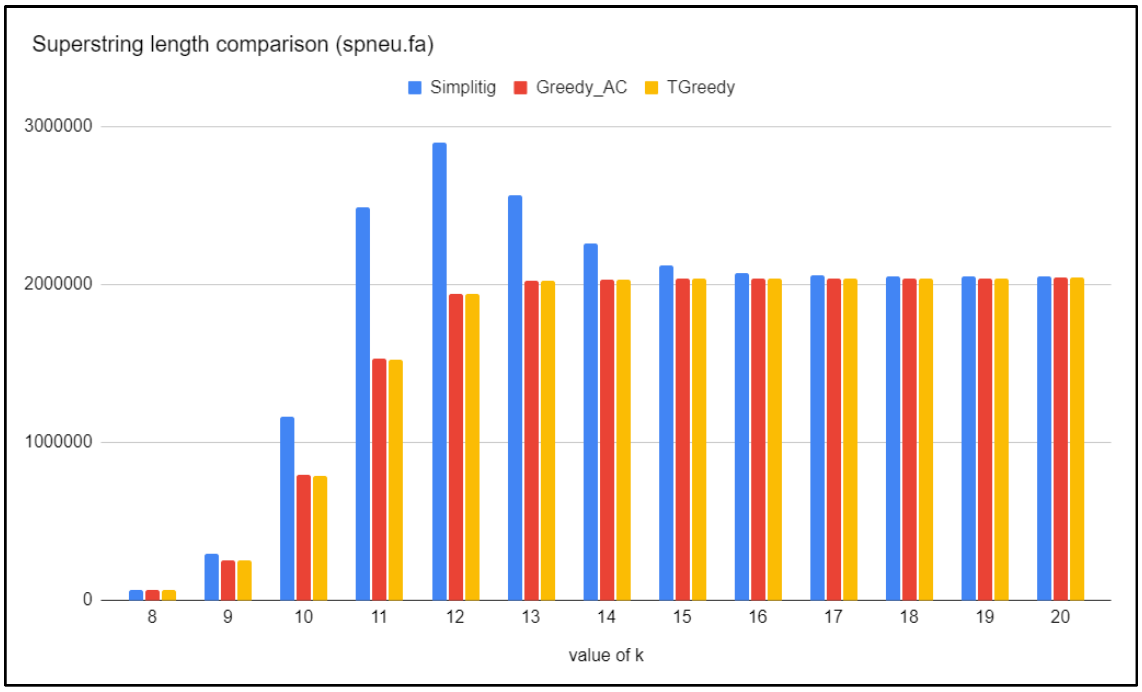

*Figure 4.1: Length of the superstring depending on the value of k (range 8 - 20) for input file: spneu.fa*

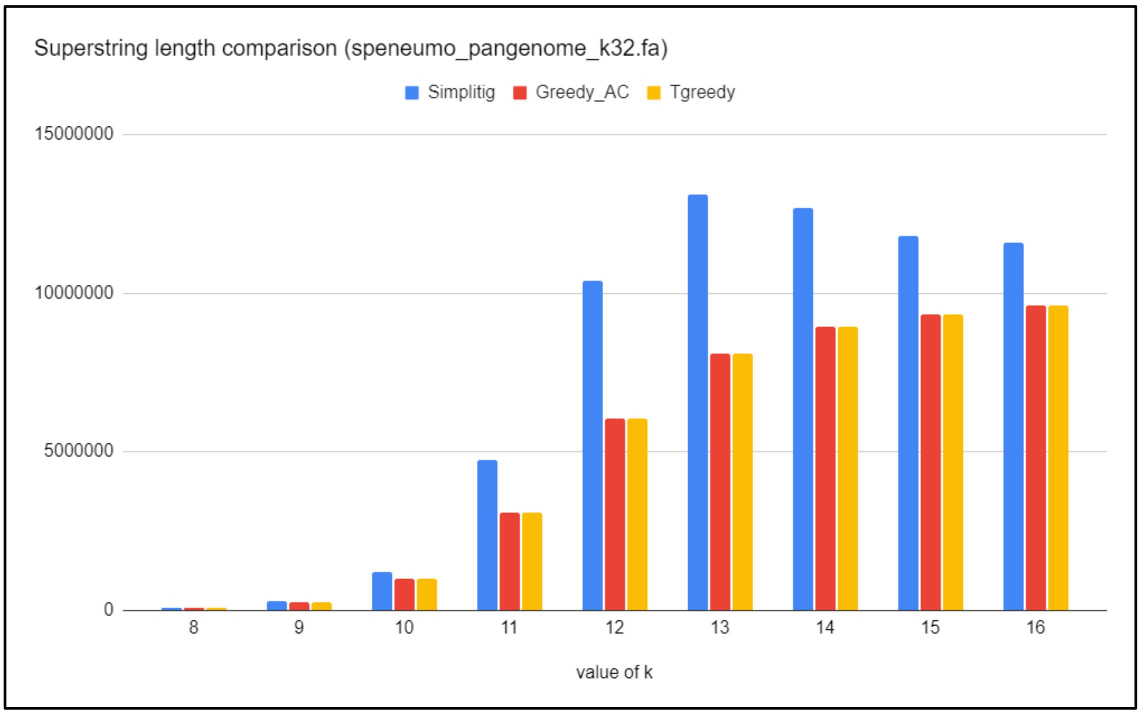

*Figure 4.2: Length of the superstring depending on the value of k (range 8 - 16) for input file: spneumo_pangenome_k32.fa*

The first observation which can be derived from this comparative analysis of superstring lengths is the similarity in length of superstrings' produced by the Greedy_AC and TGreedy. Lengths of superstrings produced by Greedy_AC are usually shorter with ratio 10:3 for input file *spneu.fa*, where TGreedy resulted better superstring for k = {10,11,19}, and 8:1 for input file *spneumo_pangenome_k32.fa*, where TGreedy's superstring was shorter only for k = 11.

For the first experiment (input file: *spneu.fa)* the average difference in superstring lengths for these two algorithms is 313 characters. If we trim the smallest (a difference of 1 character at $k$ = 7) and largest (a difference of 2685 characters at $k$ = 11) differences, the average drops to 107 characters. The reason for such similarity in the lengths is a greedy approach used by both algorithms, which prioritizes merging the pairs with the longest overlaps first and leads to the similar superstrings.

The second observation derived from the comparative analysis concerns the output superstring length of the Simplitigs algorithm in comparison to the Greedy algorithms (Greedy_AC and TGreedy). Across all tested values of $k$ in the experimental range, Simplitigs consistently produced a larger superstring. However, the gap between the length of the Simplitigs superstring and the Greedy superstrings decreases as $k$ increases, since the number of simplitigs decreases. This reduction in their number subsequently leads to a shorter superstring when they are merged together.

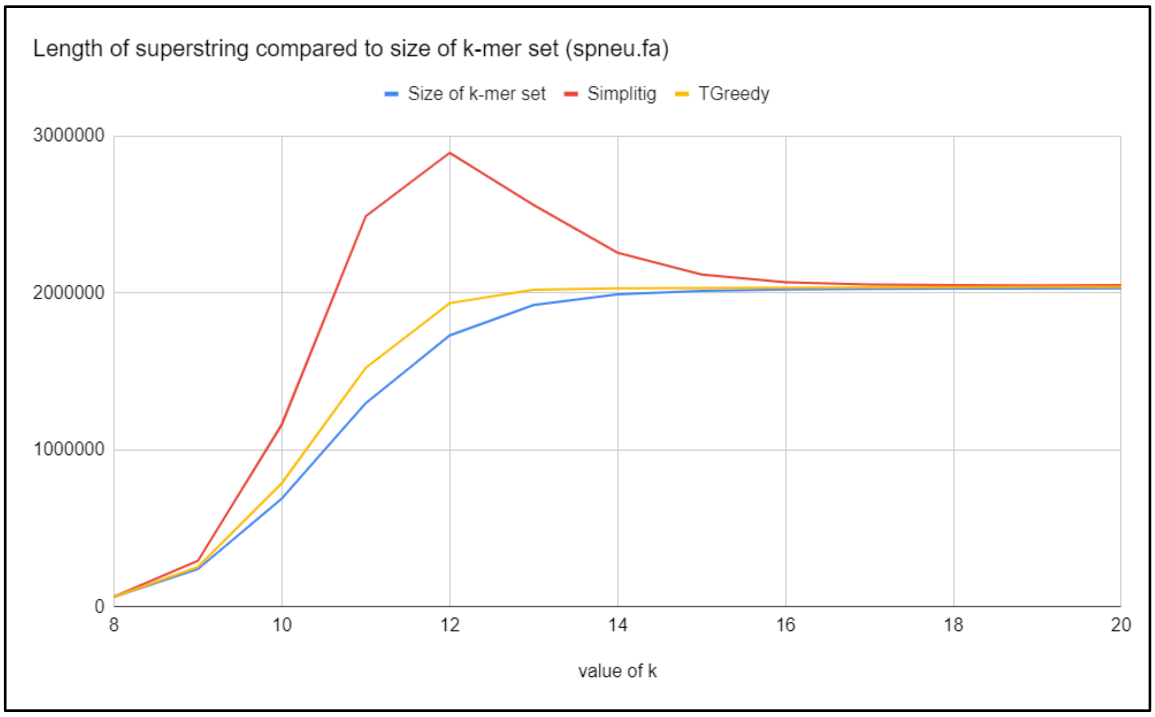

*Figure 4.3: Length of the superstrings produced by Simplitigs and TGreedy compared to the size of k-mer set for the value of k (range 8 - 20) for input file: spneu.fa*

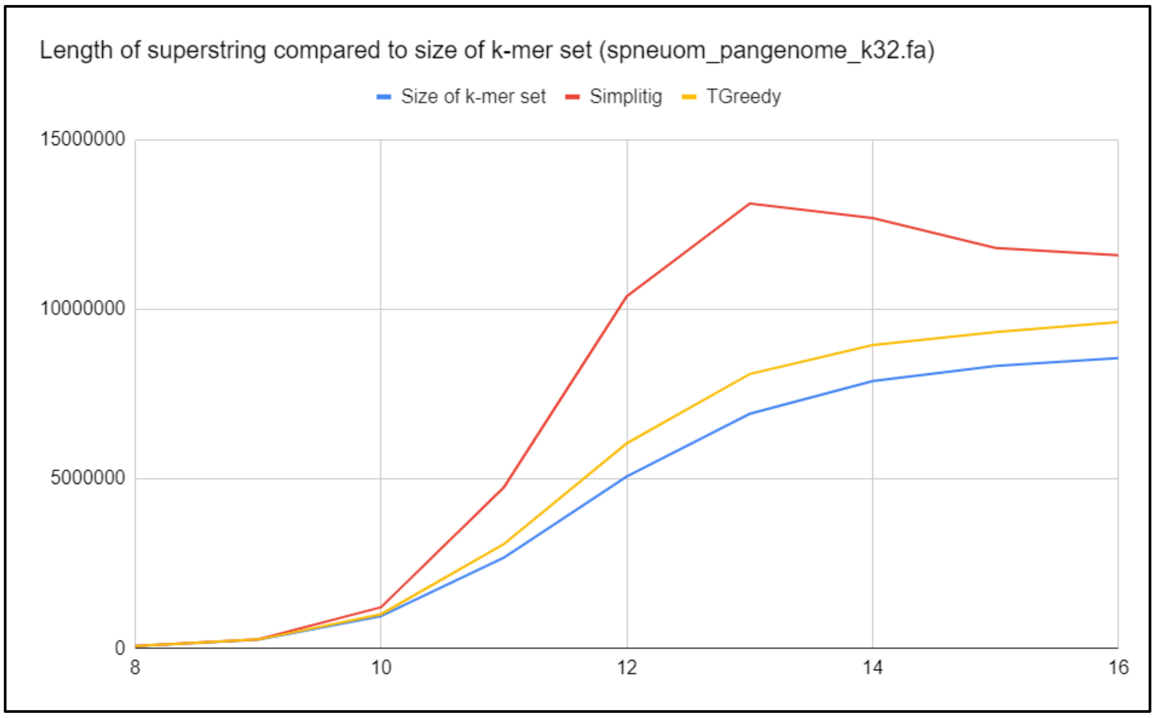

*Figure 4.4: Length of the superstrings produced by Simplitigs and TGreedy compared to the size of k-mer set for the value of k (range 8 - 16) for input file: spneumo_pangenome_k32.fa*

To compare superstring lengths with the size of the k-mer set, we focused only on the outputs from the Simplitigs and TGreedy algorithms. This was based on our earlier observation that the Greedy_AC and TGreedy algorithms produced superstrings of nearly identical lengths.

Unfortunately, the data depicted in Figure 4.4 did not offer enough information for conclusive insights due to the limited range of k, a constraint imposed by intense memory requirements. Nevertheless, Figure 4.3 provides a valuable illustration of the relationship between increasing values of k (within the given range) and superstring length. From this, we can draw our third observation: as $k$ increases the output superstrings' lengths become nearly the same for all of the algorithms, reaching its lower bound, which is the size of the k-mer set. The reason behind this lies in the de Bruijn graph, which for large $k$ does not contain many branches and becomes similar to a path.

## 4.4.2. Comparison of Computational Efficiency

Given the large values and wide range obtained for peak memory usage, calculated in bytes, we utilized a logarithmic scale to effectively represent the data on the charts. Let $Y$ represent the original set of values on the y-axis, we create $Y'$ as the set of new y-values such that $Y' = \{y' | log(y) = y' \cdot 10^6, \; for\ all\ y \in Y\}$.

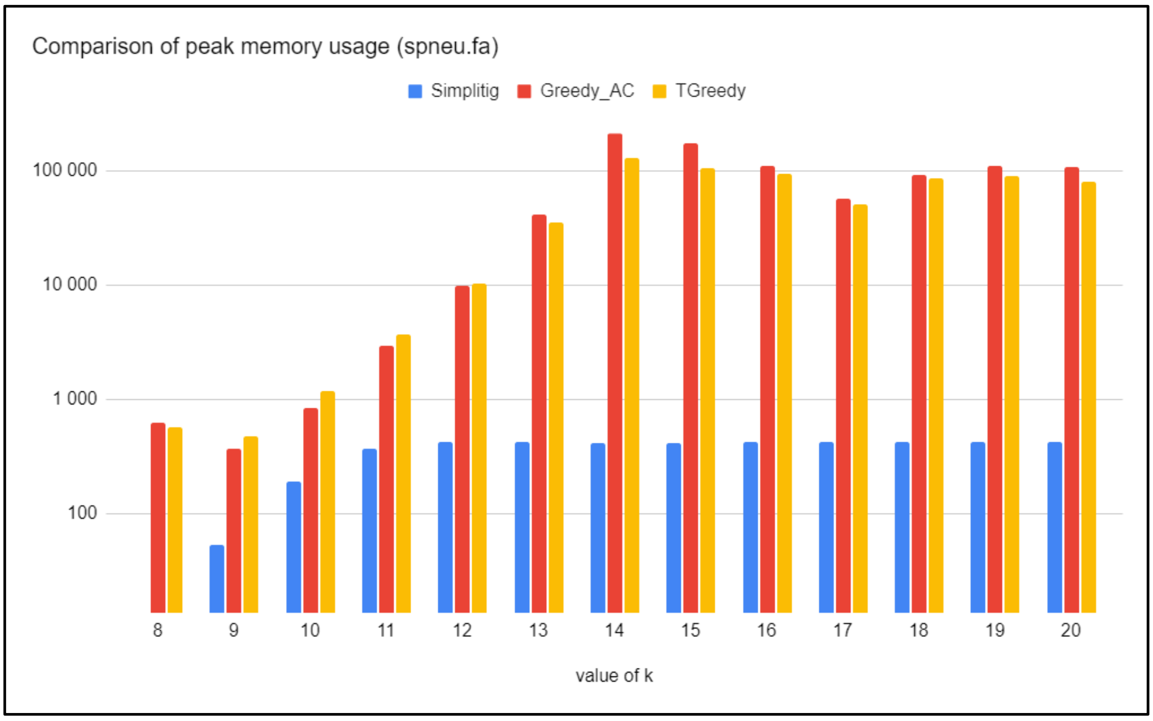

*Figure 4.5: Comparison of the peak memory usage of algorithms Simplitigs, Greedy_AC and TGreedy for the value of k (range 8 - 20) and input file: spneu.fa*

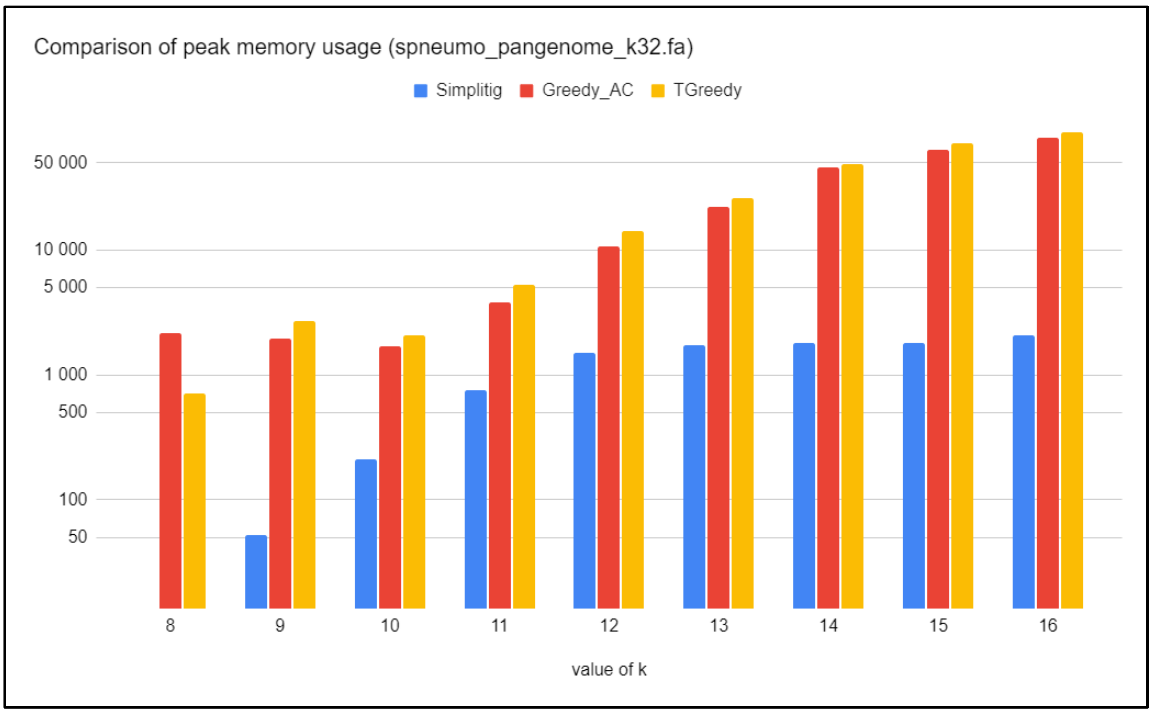

*Figure 4.6: Comparison of the peak memory usage of algorithms Simplitigs, Greedy_AC and TGreedy for the value of k (range 8 - 16) and input file: spneumo_pangenome_k32.fa*

Note, that for $k = 8$, Simplitigs exhibited such low memory usage that the bar is virtually indistinguishable from the x-axis, even after logarithmic scaling.

We make two observations about the results in Figures 5 and 6. Firstly, Greedy_AC and TGreedy exhibit similar levels of high memory consumption, due to their similar data structures used.

Secondly, Simplitigs considerably reduces memory usage compared to the Greedy approaches. The reason for this is that Simplitigs does not construct or store Aho-Corasick automaton with additional data structures, unlike Greedy_AC and TGreedy. Simplitigs only maintains a hash table of k-mers and removes k-mers from the set as they are used in building the simplitigs, further reducing the memory consumption. Conversely, Greedy algorithms must maintain overlap information, causing an increase in memory usage as the value of k increases.

For the representation of CPU times, counted in seconds, we use the logarithmic scale as well with the set $Y' = \{y' | log(y) = y', \ for\ all\ y \ \in Y\ \}$.

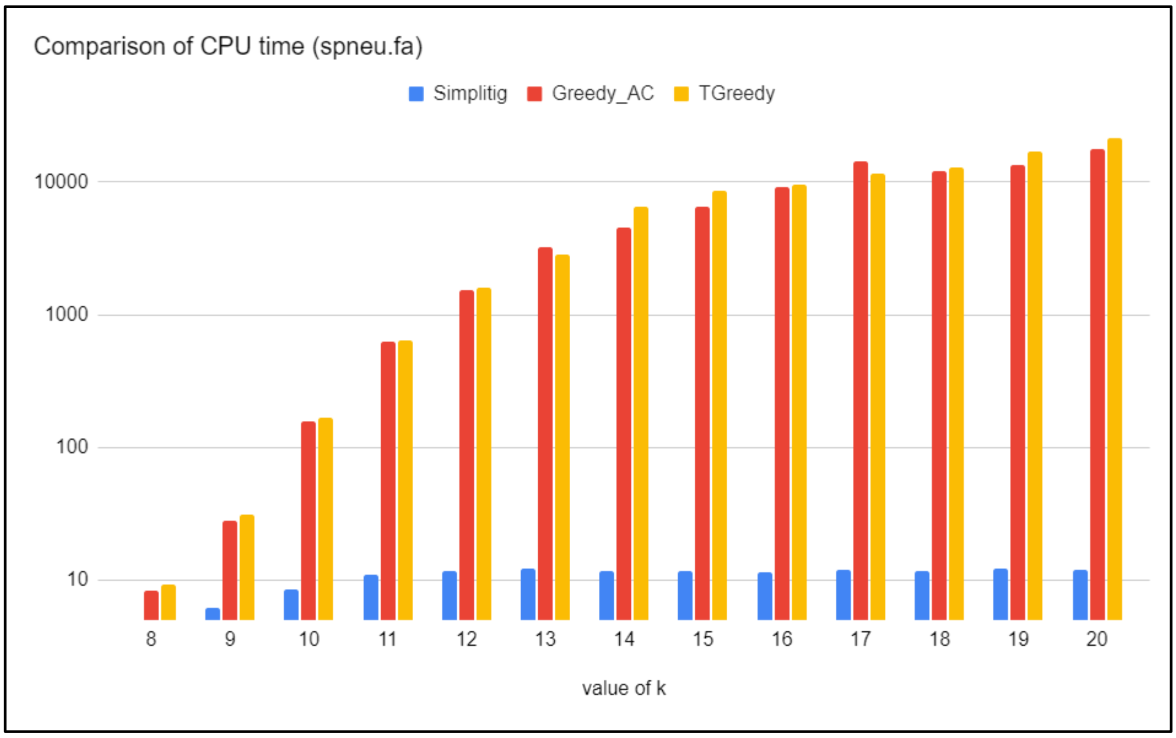

*Figure 4.7: Comparison of the CPU times of the algorithms Simplitigs, Greedy_AC and TGreedy for the value of k (range 8 - 20) and input file: spneu.fa*

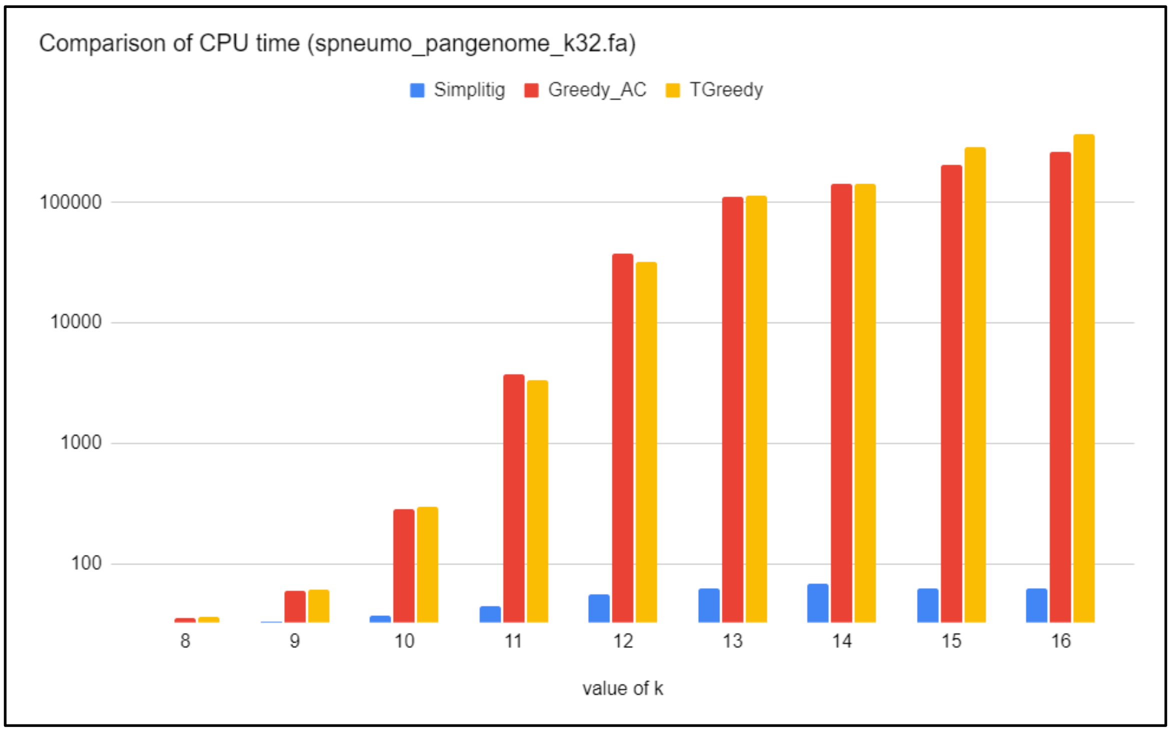

*Figure 4.8: Comparison of the CPU times of the algorithms Simplitigs, Greedy_AC and TGreedy for the value of k (range 8 - 16) and input file: spneumo_pangenome_k32.fa*

Greedy_AC and TGreedy display almost identical CPU times, though TGreedy, on average, takes slightly longer. This is due to the Greedy approach being applied twice in TGreedy: initially to acquire the MGreedy set, which is the optimal cycle cover, and subsequently on a set of representative strings for the cycle.

Simplitigs proves to be significantly faster, which can be attributed to its primary operation - extending simplitigs at both ends and checking for the presence of extended k-mers in the hash table. This operation executes in constant time and scales well, not requiring more resources with an increase in $k$.

In contrast, Greedy_AC and TGreedy consume substantial amounts of memory. For example, with the input file spneumo_pangenome_k32.fa and $k = 13$, Greedy_AC requires more than 22 GB of memory, while Simplitigs uses only 1.7 GB. Greedy algorithms operate based on overlaps between k-mers. As $k$ increases the possibility of overlaps between k-mers increase, leading to an increased number of operations and thus longer CPU time.

The results of CPU time and peak memory usage comparisons could be significantly improved in the future through the use of a more computationally capable language than Python. Despite these potential enhancements, the presented prototype effectively illustrates the crucial comparison information between the three implemented algorithms.

# Conclusion

In this thesis, a variety of methods for representing k-mer sets have been investigated, with special attention given to three specific algorithms: Simplitigs, Greedy_AC, and TGreedy. These algorithms have been explored and tested against a fixed range of values of *k* and two different input files to provide a comprehensive understanding of their advantages and limitations.

Among the tested methods, Simplitigs emerged as the fastest and most memory-efficient approach, being particularly efficient for larger values of *k*. Its relatively low memory consumption and fast computation make it an efficient method for managing large genomic data.

On the other hand, Greedy_AC and TGreedy, which produces similar-length superstrings, demonstrated significantly greater demands in terms of memory consumption and computational time. While their greedy strategy can yield effective results in terms of shorter superstrings, it also requires more extensive computational resources.

The advantage of Greedy_AC and TGreedy, which lies in producing shorter superstrings, compared to Simplitigs, becomes less visible within an increase in the value of *k*. As *k* grows, the length of superstrings outputted by Simplitigs moves closer to the size of the k-mer sets, reducing the gap between Simplitig's and Greedys' superstring lengths.

Studies in this thesis set the stage for further research into the problem of efficient representation of k-mer sets and improvements of the algorithms, utilized for this task. Presented as a prototype written in Python, implementation can be rewritten to more computationally efficient languages, enabling testing with larger k-values and larger FASTA files.

Potential avenues for future work could include a comparison of the implemented Greedy algorithms with matchtigs and eulertigs (optimal simplitigs), enhancing the understanding of the efficiency of various approaches.

Additionally, the bidirectional model that accounts for both orientations of k-mer could be considered for the further implementations, providing additional explorations on the k-mer set representation problem. Such improvements would allow for more precise and efficient management of k-mer data, paving the way for future advancements in this field.

# Bibliography

Karel Břinda, Michael Baym, Gregory Kucherov. Simplitigs as an efficient and scalable representation of de Bruijn graphs. Genome Biology 22, 96 (2021). URL: https://doi.org/10.1186/s13059-021-02297-z

Esko Ukkonen. Linear-Time Algorithm for Finding Approximate Shortest Common Superstrings. Algorithmica 5, 313–323 (1990). URL: https://doi.org/10.1007/BF01840391

Avrim Blum, Tao Jiang, Ming Li, John Tromp, Mihalis Yannakakis. Linear Approximation of Shortest Superstrings. Journal of the Association for Computing Machinery 41(4), 630-647 (1994). URL: https://doi.org/10.1145/179812.179818

Stephanie C. Roth. What is genomic medicine? Journal of the Medical Library Association 107(3), 442-448 (2019). URL: https://doi.org/10.5195/jmla.2019.604

Philip Compeau and Pavel Pevzner. Bioinformatics Algorithms: An Active Learning Approach. Active Learning Publishers 3rd ed., 5-8 (2018)

John D. Kececioglu and E.Mayers. Combinatorial algorithms for DNA sequence assembly. Algorithmica 13, 7-51 (1995). URL: https://doi.org/10.1007/BF01188580

Rayan Chikhi, Antoine Limasset and Paul Medvedev. Compacting de Bruijn graphs from sequencing data quickly and in low memory. Bioinformatics 32(14), i201-i208 (2016). URL: https://doi.org/10.1093/bioinformatics/btw279

Amatur Rahman and Paul Medvedev. Representation of k-Mer Sets Using Spectrum-Preserving String Sets. Journal of Computational Biology 28(4), 381-394 (2021). URL: https://doi.org/10.1089/cmb.2020.0431

Sebastian Schmidt and Jarno N. Alanko. Eulerigs: Minimum Plain Text Representation of k-mer Sets Without Repetitions in Linear Time. Algorithms for Molecular Biology 18, 5 (2022). URL: https://doi.org/10.1186/s13015-023-00227-1

Sebastian Schmidt, Shahbaz Khan, Jarno N. Alanko, Giulio E. Pibiri and Alexandru I. Tomescu. Matchtigs: minimum plain text representation of k-mer sets. Genome Biology 24, 136 (2023). URL: https://doi.org/10.1186/s13059-023-02968-z

Michael R. Garey and David S. Johnson. Computers and Intractability: A Guide to the Theory of NP-completeness. W. H. Freeman and Company, San Francisco (1979) URL: https://doi.org/10.2307/2273574

John K. Gallant. String Compression Algorithms. Princeton University ProQuest Dissertations Publishing (1982)

Jorma Tarhio and Esko Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. Theoretical Computer Science 57(1), 131-145 (1988). URL: https://doi.org/10.1016/0304-3975(88)90167-3

Matthias Englert, Nicolaos Matsakis and Pavel Veselý. Improved Approximation Guarantees for Shortest Superstrings using Cycle Classification by Overlap to Length Ratios. STOC 2022: Proceeding of the 54th Annual ACM SIGACT Symposium on Theory of Computing, 313-330 (2022) URL: https://dl.acm.org/doi/10.1145/3519935.3520001

Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. Communications of the ACM 18(6), 333-340 (1975). URL: https://doi.org/10.1145/360825.360855

Nicholas J. Croucher, Jonathan A. Finkelstein, Stephen I. Peloton, Julian Parkhill, Stephen D. Bentley, Marc Lipsitch, William P. Hanage. Continued impact of pneumococcal conjugate vaccine on carriage in young children. Scientific Data 2, 150058 (2009). URL: https://doi.org/10.1542/peds.2008-3099

# List of Figures

Figure 1.1: Representation of k-mer sets as unitigs

Figure 1.2: Representation of k-mer sets as simplitigs

Figure 1.3: Computations on a edge-centric de Bruijn graph with $k = 5$. Credits: Schmidt et al. [2023]

Figure 2.1: Overlap graph constructed for the set of strings {"ACTG", "AGCG", "AGCA", "CAAC", "TGCA"}

Figure 2.2: Aho-Corasick automaton constructed from the set of words {"AGTG", "TGCA", "CAAC", "AGAG", "TCAC"} over alphabet {'A', 'C', 'T', 'G'}

Figure 2.3: Maximum Hamiltonian Path and maximum Cycle cover on the overlap graph constructed for the set of strings {"ACGT", "CGTA", "GTAC", "CGTC", "TACG"}

Figure 4.1: Length of the superstring depending on the value of *k* (range 8 - 20) for input file: spneu.fa

Figure 4.2: Length of the superstring depending on the value of *k* (range 8 - 16) for input file: spneumo_pangenome_k32.fa

Figure 4.3: Length of the superstrings produced by Simplitigs and TGreedy compared to the size of k-mer set for the value of *k* (range 8 - 20) for input file: spneu.fa

Figure 4.4: Length of the superstrings produced by Simplitigs and TGreedy compared to the size of k-mer set for the value of *k* (range 8 - 16) for input file: spneumo_pangenome_k32.fa

Figure 4.5: Comparison of the peak memory usage of algorithms Simplitigs, Greedy_AC and TGreedy for the value of *k* (range 8 - 20) and input file: spneu.fa

Figure 4.6: Comparison of the peak memory usage of algorithms Simplitigs, Greedy_AC and TGreedy for the value of *k* (range 8 - 16) and input file: spneumo_pangenome_k32.fa

Figure 4.7: Comparison of the CPU times of the algorithms Simplitigs, Greedy_AC and TGreedy for the value of *k* (range 8 - 20) and input file: spneu.fa

Figure 4.8: Comparison of the CPU times of the algorithms Simplitigs, Greedy_AC and TGreedy for the value of *k* (range 8 - 16) and input file: spneumo_pangenome_k32.fa

# Attachments

## A.1. Tables

*Table 1*: Statistics with input file spneu.fa

| Algorithm | Value of k | Size of k-mer set | Length of the superstring | CPU time (seconds) | Peak memory Usage (bytes) |
|---|---|---|---|---|---|
| Greedy_AC | 8 | 65 101 | 65 492 | 8.2687 | 625 738 179 |
| TGreedy | 8 | 65 101 | 65 540 | 9.1629 | 564 868 966 |
| Simplitigs | 8 | 65 101 | 67 047 | 4.9956 | 13 420 043 |
| Greedy_AC | 9 | 241 952 | 254 697 | 28.1152 | 370 194 312 |
| TGreedy | 9 | 241 952 | 254 735 | 31.2585 | 473 048 502 |
| Simplitigs | 9 | 241 952 | 294 672 | 6.1267 | 52 792 741 |
| Greedy_AC | 10 | 691 301 | 792 253 | 156.9808 | 830 039 225 |
| TGreedy | 10 | 691 301 | 788 604 | 166.3084 | 1 177 349 334 |
| Simplitigs | 10 | 691 301 | 1 163 036 | 8.4621 | 188 531 969 |
| Greedy_AC | 11 | 1 298 893 | 1 531 147 | 633.8519 | 2 953 198 228 |
| TGreedy | 11 | 1 298 893 | 1 523 832 | 637.1674 | 3 710 211 500 |
| Simplitigs | 11 | 1 298 893 | 2 489 283 | 10.9716 | 371 386 044 |
| Greedy_AC | 12 | 1 730 646 | 1 936 476 | 1533.3619 | 9 805 288 809 |
| TGreedy | 12 | 1 730 646 | 1 936 531 | 1604.7838 | 10 268 914 465 |
| Simplitigs | 12 | 1 730 646 | 2 893 027 | 11.7894 | 419 297 494 |
| Greedy_AC | 13 | 1 924 270 | 2 022 377 | 3240.7212 | 41 215 285 520 |
| TGreedy | 13 | 1 924 270 | 2 022 399 | 2851.7912 | 35 034 952 672 |
| Simplitigs | 13 | 1 924 270 | 2 561 158 | 12.1327 | 420 272 070 |
| Greedy_AC | 14 | 1 992 657 | 2 031 124 | 4563.3255 | 211 667 754 256 |
| TGreedy | 14 | 1 992 657 | 2 031 216 | 6524.4649 | 130 836 837 056 |
| Simplitigs | 14 | 1 992 657 | 2 257 428 | 11.6134 | 416 730 651 |
| Greedy_AC | 15 | 2 015 184 | 2 033 199 | 6510.0183 | 172 978 902 720 |
| TGreedy | 15 | 2 015 184 | 2 033 252 | 8643.1956 | 106 415 807 488 |
| Simplitigs | 15 | 2 015 184 | 2 118 546 | 11.6958 | 416 180 612 |
| Greedy_AC | 16 | 2 022 978 | 2 035 007 | 9230.1749 | 110 185 613 671 |
| TGreedy | 16 | 2 022 978 | 2 035 136 | 9501.2766 | 93 932 449 637 |
| Simplitigs | 16 | 2 022 978 | 2 069 898 | 11.43 | 417 414 781 |
| Greedy_AC | 17 | 2 026 270 | 2 036 704 | 14351.0917 | 56 454 825 859 |
| TGreedy | 17 | 2 026 270 | 2 036 705 | 11631.8929 | 50 971 406 905 |
| Simplitigs | 17 | 2 026 270 | 2 054 638 | 11.9504 | 419 305 768 |
| Greedy_AC | 18 | 2 028 150 | 2 038 007 | 12000.3824 | 91 927 318 049 |
| TGreedy | 18 | 2 028 150 | 2 038 064 | 12953.3803 | 84 909 974 737 |
| Simplitigs | 18 | 2 028 150 | 2 050 012 | 11.6892 | 421 393 180 |
| Greedy_AC | 19 | 2 029 518 | 2 039 346 | 13386.5356 | 110 559 276 774 |
| TGreedy | 19 | 2 029 518 | 2 039 300 | 17102.1678 | 89 722 050 795 |
| Simplitigs | 19 | 2 029 518 | 2 049 372 | 12.1888 | 423 547 454 |
| Greedy_AC | 20 | 2 030 690 | 2 040 482 | 17735.5472 | 107 958 105 268 |

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| TGreedy | 20 | 2 030 690 | 2 040 530 | 21469.5592 | 80 972 807 392 |
| Simplitigs | 20 | 2 030 690 | 2 050 051 | 12.0138 | 425 724 904 |

*Table 2*: Statistics with input file spneumo_pangenome_k32.fa

| Algorithm | Value of k | Size of k-mer set | Length of the superstring | CPU time (seconds) | Peak memory Usage (bytes) |
|---|---|---|---|---|---|
| Greedy_AC | 8 | 65 524 | 65 555 | 35.1701 | 2 167 100 065 |
| TGreedy | 8 | 65 524 | 65 616 | 36.3618 | 713 810 177 |
| Simplitigs | 8 | 65 524 | 65 755 | 32.4817 | 13 355 369 |
| Greedy_AC | 9 | 259 766 | 261 818 | 59.7714 | 1 955 334 281 |
| TGreedy | 9 | 259 766 | 261 875 | 61.1041 | 2 696 458 055 |
| Simplitigs | 9 | 259 766 | 269 982 | 33.2014 | 52 346 187 |
| Greedy_AC | 10 | 951 599 | 1 007 618 | 282.6751 | 1 695 916 916 |
| TGreedy | 10 | 951 599 | 1 007 692 | 297.6461 | 2 070 124 457 |
| Simplitigs | 10 | 951 599 | 1 213 103 | 37.0353 | 211 362 029 |
| Greedy_AC | 11 | 2 676 289 | 3 080 939 | 3755.7631 | 3 792 562 395 |
| TGreedy | 11 | 2 676 289 | 3 080 973 | 3312.6641 | 5 274 811 139 |
| Simplitigs | 11 | 2 676 289 | 4 747 579 | 44.7824 | 752 321 723 |
| Greedy_AC | 12 | 5 071 523 | 6 052 626 | 37802.0556 | 10 649 279 459 |
| TGreedy | 12 | 5 071 523 | 6 051 860 | 31889.2313 | 14 122 529 140 |
| Simplitigs | 12 | 5 071 523 | 10 388 417 | 55.8926 | 1 497 772 976 |
| Greedy_AC | 13 | 6 920 055 | 8 094 843 | 110664.8328 | 22 249 560 675 |
| TGreedy | 13 | 6 920 055 | 8 094 959 | 113081.1121 | 26 217 431 620 |
| Simplitigs | 13 | 6 920 055 | 13 118 439 | 62.1614 | 1 744 354 666 |
| Greedy_AC | 14 | 7 887 260 | 8 942 835 | 142573.3565 | 45 797 478 998 |
| TGreedy | 14 | 7 887 260 | 8 943 181 | 141629.6616 | 48 873 120 053 |
| Simplitigs | 14 | 7 887 260 | 12 691 644 | 67.6975 | 1 791 469 801 |
| Greedy_AC | 15 | 8 333 095 | 9 329 011 | 207276.1448 | 63 506 119 825 |
| TGreedy | 15 | 8 333 095 | 9 329 553 | 288553.7533 | 71 812 237 757 |
| Simplitigs | 15 | 8 333 095 | 11 807 727 | 62.7245 | 1 799 140 192 |
| Greedy_AC | 16 | 8 564 842 | 9 627 117 | 264684.5564 | 80 049 312 702 |
| TGreedy | 16 | 8 564 842 | 9 627 848 | 371183.2081 | 87 872 452 062 |
| Simplitigs | 16 | 8 564 842 | 11 595 907 | 61.7706 | 2 088 194 999 |

## A.2. Technical Documentation

### A.2.1. MAIN Modules

#### main.py[1]

This module executes different algorithms based on the command-line arguments provided.

**Input:** input Fasta file and arguments.

**Output:** superstring and its mask.

**Algorithm:**

1. Check if there is any algorithm chosen. If no algorithm then message *Error* and exit code 0.
2. Load the Fasta file into the set of kmers *arr* using *load* function.
3. Run the chosen algorithm on the set *arr.*
    a. If simplitigs (argument -s / --simplitig) then get the *superSet* using *compute_simplitig* function and then merge the strings from the set into the superstring *superStr.*
    b. If Greedy_Aho-Corasick (argument -a / --aho-corasick) then get the *superSet* using *FindSuperSet* function and then merge the strings from the set into the superstring *superStr.*
    c. If Tgreedy (argument -t / --tgreedy) then get the *superSet* using *FindSuperSetTgreedy* function and then merge the strings from the set into the superstring *superStr.*
4. Check if the binary mask is chosen (argument -b / --bitstring_mask). Apply the mask accordingly.
5. Check if the output file is chosen (argument -o OUTPUT / --output OUTPUT). Store the output *superStr* and its mask accordingly.
6. Check if testing is chosen (argument -T / --test). Run *testAll* if test is required.
7. Check if a statistic is chosen (argument -S STATS / --stats STATS). Run *outputStats* if statistics is required.

#### Load_fasta.py[2]

This module is used for loading data from a FASTA file format.

Contains one function *def load (k, fileName)*:

[1] https://github.com/Ekatmil/Efficient-representation-of-k-mers-sets/blob/8f2d5ce215194bf419bfbcf7bb770758e5c4db0d/main.py
[2] https://github.com/Ekatmil/Efficient-representation-of-k-mers-sets/blob/8f2d5ce215194bf419bfbcf7bb770758e5c4db0d/Load_fasta.py

**Input:** parsed integer *k*; *fileName* – input fasta file.

**Output:** a set of kmers.

**Purpose:** read sequences from a FASTA file and extract all unique substrings of a specified length from those sequences.

**Algorithm:**

1. Parse the file specified by the *fileName* parameter using *SeqIO.parse* method provided by the *Biopython* package library.
2. Loop over each *record* of sequence obtained from the file and over each *kmer* in the *record.*
    a. Add *kmer* into the set *arr.*
3. Resulting set is returned, containing all the unique kmers present in the records read from the file.

## mask.py[3]

This module is responsible for generating a mask, which describes each nucleotide's location within the output superstring. The masks generated are useful for determining the location of the original kmers in the superstring.

Contains of two functions with the same **input**:

- *Kset* – set of strings (kmers obtained by Load_fasta.py)
- *str1* – superstring

I. *def findMask(Kset, str1, k)*

**Output:** string *str1*, where all occurrences of kmers are found and uppercase.

**Purpose**: replace all occurrences of substrings in *Kset* that have length *k* with uppercase letters in *str1*.

**Algorithm:**

1. Convert the input string *str1* into a list named *l1* in order to each character of the string to be accessed and transformed later.
2. Loop over characters in the *l1* (iterates through a range starting at 0 and ending at the (*len(l1) – k +1*) to ensure that the loop considers every substring of length *k* in the string.)

[3] https://github.com/Ekatmil/Efficient-representation-of-k-mers-sets/blob/8f2d5ce215194bf419bfbcf7bb770758e5c4db0d/mask.py

    a. Select a substring of length *k* from *l1* starting at index *i*.
    b. If substring is in set *Kset*, then replaces the character at index *i* in *l1* with its lowercase version.
3. Once all the required characters have been transformed in the *l1* list, this list is mapped back into a string *StrMask*, where uppercase characters have been replaced by lowercase characters if they occurred in any of the substrings present in the set *Kset*.
4. Create a new copy of the string where all the cases are swapped. This new string is returned as the output of the function.

II. *def findMaskBinary (Kset, str1, k)*

**Output:** string *str1*, where all occurrences of kmers are found and stored in binary mask.

**Purpose**: construct a binary mask from the input string *str1*, where each digit of the mask represents the presence or absence of a substring of length *k* from *str1* in the set *Kset*.

**Algorithm:**

1. Initialize variables *l1* as an empty list to hold the generated binary mask.
2. Loop over each character in the string *str1*.
    a. Substring of length *k* is created by slicing the original string str1 and storing it in the variable *substr*.
    b. If the substring *substr* is found in the set *Kset*, then the corresponding value in the *l1* list at position i is set as '1', otherwise '0'.

Once all the iterations are completed, the elements of the *l1* list is joined together as a string and stored in the variable *StrMask*, which is the output of the function.

## testStr.py[4]

This module is responsible for testing the correctness of the generated superstring with respect to the given kmer set.

Contains four functions with the similar input.

**Input:**

[4] https://github.com/Ekatmil/Efficient-representation-of-k-mers-sets/blob/8f2d5ce215194bf419bfbcf7bb770758e5c4db0d/testStr.py

- *st* – a string, generated superstring
- *lst* – a list of kmers
- *k* – an integer representing the length of kmers
- *bn* – a string representing the binary mask (an unpacking operator in the function testAll)

I. *def allKmers (st, lst)*

**Purpose:** check if all the kmers in a list are in a given string.

**Algorithm:**

1. Create an empty dictionary, *lst_dict*.
2. Initialize each *kmer* from the *lst* with value *False* in the *lst_dict* dictionary.
3. Loop over each *kmer* in the input string.
    a. Check if it exists as a *key* in the *lst_dict*. If it does, it sets the *value* of the corresponding *key* as *True*.
4. Check if all the values of keys in the dictionary *lst_dict* are *True* using *all()* method.
5. If all the kmers from *lst* were found in the *s*t then the test is passed. Otherwise, print missing kmers from *lst* that were not found in *st*.

II. *def noDifferentStr (st, lst, k)*

**Purpose**: check if there are no false kmers in the superstring.

**Algorithm:**

1. Convert all items in *lst* to a set for efficiency.
2. Initialize a boolean variable *checker* to *True*.
3. Loop over each character in the input string *st*.
    a. If the current index plus *k* doesn't go beyond the end of the string and if the current character is uppercase: creates a substring consisting of the characters starting from the current index of length *k*.
        i. If this *kmer* is not found in the set formed by *lst*: print a message indicating that there's a false k-mer at the current index and change the value of *checker* to *False*.
4. If all kmers are present in *lst*, print a message indicating that there are no false kmers.

III. *def applyMask (st, bn)*

**Purpose:** apply binary mask on the superstring and return the superstring with Case_Sensitive mask.

**Algorithm:**

1. Create an empty string variable named *new_st* to store the modified string.
2. Loop over each character in the original string.
    a. Check if the corresponding binary digit for the character is 0 or 1.
    b. If the binary digit is 0, then the corresponding character in the original string needs to be changed to lowercase. Add to the *new_st.*
    c. If the binary digit is 1, the corresponding character in the original string should be left untouched. Add to the *new_st.*
3. After looping through all characters, return the modified string variable *new_st*.

IV. *def testAll (st, lst, k, *bn)*

**Idea**: combine all other test functions together.

**Algorithm:**

1. Check if parameter *bn* (binary mask) is present. If present then call for *applyMask(st, bn[0]).*
2. Call two other functions *allKMers* and *noDifferentStr.*

## Statistics.py[5]

This module is responsible for creation of the csv file with written statistics.

Contains one function.

*def outputStats (file_name, input_name, output_name, mask, algo, k, kmers_len, superStr_len, tm, memory)*

**Input:**

[5] https://github.com/Ekatmil/Efficient-representation-of-k-mers-sets/blob/8f2d5ce215194bf419bfbcf7bb770758e5c4db0d/Statistics.py

- *file_name* – a string containing the name of the file to be created
- *input_name* – a string containing the name of the input file
- *output_name* – a string containing the name of the output file (or *stdout* if no file name)
- *mask* – a string representing the type of the mask used (Case_Sensitive or Binary)
- *algo* – a string representing the algorithm used
- *k* – an integer representing the length of kmers used
- *kmers_len* – an integer representing the length of the kmer set
- *superStr_len* – an integer representing the length of the superstring
- *tm* – a float representing the CPU time taken for execution
- *tm1* – a float representing the wall-clock time taken for execution
- *memory* – a the maximum (peak) memory used by the algorithm during its execution

**Outpu**t: csv file.

P**urpose:** store statistics into the csv file.

**Algorithm:**

1. Create *headers* for columns in the csv file.
2. Create a list of *data* to write in the corresponding column.
3. Open *file_name* file in write mode with "UTF8" encoding.
4. Create *csv.writer()* object and write header and data into the 1st and 2nd rows respectively.

### A.2.2. HELPER Modules

### Automaton_Class.py[6]

This module contains the implementation of the Aho-Corasick automaton for efficient search for multiple patterns in a string.

The class is used to construct the Aho-Corasick automaton (trie).

*Constructor* takes as input kmers – list of strings (kmers), patterns to be searched. Then it initializes two dictionaries goto and fail, where goto stores the edges of the trie and fail stores the fail links.

[6] https://github.com/Ekatmil/Efficient-representation-of-k-mers-sets/blob/8f2d5ce215194bf419bfbcf7bb770758e5c4db0d/Automaton_Class.py

I. *def isLeaf (self, state)*

**Input:** *state* – an integer

**Output:** Boolean variable

**Purpose:** Helper function to check if a state is a leaf in a trie

**Algorithm:**

1. Check if there are any outgoing edges from the *state.*
2. Return the result.

II. *def goto_function (self)*

**Output:** Constructed automaton.

**Purpose**: Build Aho-Corasick automaton and create transitions between states based on the input kmers.

**Algorithm:**

1. Initialize a variable *new_state* equal to 0.
2. Loop through each *kmer* in *self. kmers.*
    a. Initialize variable *state* to 0.
    b. Loop over each *char*acter in *kmer.*
        i. Check if the current state (*state*) has an outgoing transition for the current character *char*. If no such transition then break the loop.
        ii. If there is such a transition (key *(state, char)* exists in *goto*), then update the *state* to the obtained value.
    c. Loop over characters on the *kmer* (This loop is in case if only some characters (or none) were transitioned through until the exit of the loop b.)
        i. Create *new_state* indexes by *new_state + 1.*
        ii. Make a transition between *state* and *new_state.*
        iii. Set *state* to be *new_state.*

III. *def fail_function (self)*

**Output:** Constructed fail links.

**Purpose**: Compute fail transitions between states based on *goto* transitions.

**Algorithm:**

1. Initialize empty list *queue.*
2. Loop over each character in the list *["A", "C", "T", "G].*
    a. Add to *queue* all states that have ongoing edges with *char* and have root (state 0) as parent.
    b. Add such states to the *fail* dictionary with corresponding fail value set to zero.
3. Enter while loop until the *queue* list is empty.
    a. Remove the first element of the *queue* and store in a variable *queue_state.*
    b. Loop over each character in the list *["A", "C", "T", "G].*
        i. Obtain value corresponding to the key *(queue_state, char)* in the *goto*. Move to the next character if the result is -1 (does not exist).
        ii. Add value to the end of the *queue.*
        iii. Construct the fail transition from *queue_state* to current *state.*
        iv. Enter while loop.
            1. Obtain value corresponding to the key *(state, char)* in the *goto*. If the result *res* is not equal to -1 (value exists) then break the loop.
            2. Add self-fail for the current *state.*
        v. Set the fail value of the destination state to the value, which represents the longest proper suffix of the input that ends at this state.

## string_functions.py[7]

This module contains overlap functions required by the Helper_Function_AC.py.

Contains two functions with the same **input:** two strings *str1* and *str2.*

I. *def overlap (str1, str2)*

**Usage:** Helper_Function_AC.addtoSet(); betterOverlap()

**Output:** overlap of two strings.

**Purpose:** Store overlap as tuple (prefix of *str1*, overlap segment, suffix of *str2*, length of overlap).

[7] https://github.com/Ekatmil/Efficient-representation-of-k-mers-sets/blob/8f2d5ce215194bf419bfbcf7bb770758e5c4db0d/string_functions.py

**Algorithm:**

1. Initialize the *maxOverlap* to a very small negative number.
2. Calculate length *len1* and *len 2* for the *str1* and *str 2* respectively.
3. Initializes *fix_i* to 0, which stores the value of *i* when the maximum overlap occurs.
4. Loop over all possible values in range from minimum of two length +1 down to 1. Starting from the end in order to find the longest common suffix and prefix between *str1* and *str2*.
    a. Extract substring *subStr1*, which represents the last *i* characters of *str1.*
    b. Extract substring *subStr2*, which represents the first *i* characters of *str2.*
    c. If *subStr1* and *subStr2* are the same strings and *i* is greater than *maxOverlap* then:
        i. Update *maxOverlap* with *i* (new overlap is found).
        ii. Assign *overlapStr* with *subStr1.*
        iii. Assign *fix_i* with value *i.*
5. Based on the fixing point *fix_i*, the function returns three parts of the strings *str1* and *str2*: the part of *str1* that is not overlapping with *str2*, the overlapping string, and the part of *str2* that is not overlapping with *str1*. The returned value also includes the value of the maximum overlap. If no overlap is found, the function returns an empty string for the first and third elements of the returned list, *str*1 concatenated with *str2* as the second element of the list and a zero for the fourth element of the list.

II. *def betterOverlap (str1, str2)*

**Output:** overlap of two strings.

**Purpose**: Return the result of comparing between two overlaps.

**Algorithm:**

1. Call *overlap (str1, str2)* and store the result to *resStr1.*
2. Call *overlap (str2, str1*) and store the result to r*esStr2.*
3. Compare the fourth element (size of the overlap) of *resStr1* and *resStr2* and return the larger result.

## Helper_Functions_AC.py[8]

This module contains helper functions required by the Greedy_AC algorithm and Tgreedy algorithm implementations.

Consists of five functions.

I. *def addMultipleValues (dict, key, value)*

**Usage**: Greedy_AC.preprocessing(); Greedy_AC.Hamiltonian(); tgreedy.HamiltonianT()

**Input:** Dictionary *dict*, *key* and *value.*

**Output**: Dictionary *dict* with value added to the list associated with the given key.

**Purpose**: store multiple values with one key in the dictionary.

**Algorithm:**

1. Check if the given *key* exists in *dict*. If it does not exist, then an empty list is associated with the given *key*.
2. Check if the given *value* is a list. If it is then append each value into the list associated with the given *key.*
3. Else append the single *value* into the list with the given *key*.
4. Return the dictionary with the changes.

II. *def findSingle (H, n)*

**Usage:** Greedy_AC.FindSuperSet(); tgreedy.FindSuperSetTgreedy()

**Input:**

- *H* – a dictionary representing Hamiltonian Path (state: state)
- *n* – an integer representing number of states (kmers)

**Output:** list of states.

**Purpose:** Get list of states that have only one (indegree 0) or none (self-overlap) neighboring state.

**Algorithm:**

8 https://github.com/Ekatmil/Efficient-representation-of-k-mers-sets/blob/8f2d5ce215194bf419bfbcf7bb770758e5c4db0d/Helper_Functions_AC.py

1. Create a list of all states (keys).
2. Create a set of values from dictionary *H.*
3. Iterate through keys and add into the output list those states for which we have not found in values.
4. Return the output list.

III. *def initializeForbidden (dict, n)*

**Usage**: Greedy_AC.Hamiltonian(); tgreedy.HamiltonianT()

**Input:**

- *dict* – a dictionary
- *n* – an integer representing number of keys

**Output:** *dict* with *False* as value for each key.

**Purpose:** Initialize dictionary with boolean value *False* for all *n* keys.

**Algorithm:**

1. Loop in range *n* and for each iteration assign a key with value *False* to the dictionary.
2. Return dictionary.

IV. *def removeFromDict (dict, state, i)*

**Usage**: tgreedy.HamiltonianT()

**Input:**

- *dict* – a dictionary
- *state* – an integer representing key
- *i* – an integer representing value

**Output**: *dict* without value *i* in state list.

**Idea:** Remove a value from the list associated with the key.

**Algorithm:**

1. Get the value *helper_list* in *dict* associated with the key.
2. Remove value *i* from the *helper_list.*

3. Store updated *helper_list* as value for state in *dict.*
4. Return dictionary *dict.*

V. *def addtoSet (kmer_lst, single_lst, H, outputSet)*

**Usage**: Greedy_AC.FindSuperSet (); tgreedy.FindSuperSetTgreedy()

**Input:**

- *kmer_lst* – a list of strings (kmers)
- *single_lst* – a list of integers representing indices from the *kmer_lst* (obtained by (II))
- *H* – a dictionary representing Hamiltonian Path (state: state)
- *outputSet* – an empty set that will be used to store output

**Output:** *outputSet* with unique sequences.

**Purpose**: Find all unique strings that can be formed by overlapping input kmers.

**Algorithm:**

```
1. Loop over indices x in single_lst.
    a. Assign x to be key.
    b. Initialize variable sStr with the corresponding kmer in kmer_lst.
    c. Enter while loop.
        i. If H contains a value for the current key, then
            1. Get value for a current key.
            2. Compute overlap between two kmers under indices key
               and value using overlap function.
            3. Update the string sStr to be the concatenation of the
               prefix of the first kmer, the overlapping segment, and the
               suffix of the second kmer.
            4. Replace the kmer at the value index in kmer_lst with sStr.
            5. Reassign key to the value.
        ii. Else (If H does not contain a value for the current key):
            1. Add sStr string to the outputSet.
            2. Exit the loop.
2. Return outputSet.
```

## A.2.3. ALGORITHM Modules

### simplitig.py[9]

This module implements the Simplitigs algorithm and provides an efficient way for optimizing complexity in the analysis of genome sequence data.

Consists of four functions with similar **input**:

- *K* - set of k-mers
- *simpling* - a string representing a simple path in a directed de Bruijn graph
- *k* - an integer representing the length of k-mers

I. *def extend_simplitig_forwards (K, simpling, k)*

**Output**: Updated set *K* and extended string *simpling.*

**Purpose**: extend the given *simpling* forwards until it reaches an end or can no longer be extended.

**Algorithm:**

1. Extend while can:
   a. Take the last *k-1* characters of *simpling* and append each *A*, *C*, *G*, *T* character to it to create a new *kmer*.
   b. If new *kmer* is in set *K*, then extend *simpling* to include character and remove *kmer* from the set *K* (to ensure that all unique k-mers are used before a repeat cycle starts).
   c. Break while loop when can not find any more k-mers that match the criteria.
2. Return two values set *K* and *simpling.*

II. *def extend_simplitig_backwards (K, simpling, k)*

**Output**: Updated set *K* and extended string *simpling.*

**Purpose**: extend the given *simpling* backwards until it reaches an end or can no longer be extended.

**Algorithm:**

1. Extend while can:

[9] https://github.com/Ekatmil/Efficient-representation-of-k-mers-sets/blob/8f2d5ce215194bf419bfbcf7bb770758e5c4db0d/simplitig.py

    a. Take the first *k-1* characters of simpling and append each *A*, *C, G*, *T* character to its beginning to create a new *kmer*.
    b. If new *kmer* is in set *K*, then extend *simpling* to include character and remove kmer from the set K (to ensure that all unique kmers are used before a repeat cycle starts).
    c. Break while loop when can not find any more kmers that match the criteria.
2. Return two values set *K* and *simpling.*

III. *def compute_maximum_simplitig_from_kmer (K, seeding_kmer, k)*

**Input:** *seeding_kmer* - a string, representing initial starting kmers

**Output**: Updated set *K* and extended string *simpling.*

**Purpose**: store multiple values with one key in the dictionary.

**Algorithm:**

1. Set *simpling* to be *seeding_kmer.*
2. Apply *extend_simplitig_forwards* and *extend_simplitig_backwards* methods to generate maximum string *simpling* that covers the entire sequence in set *K.*
3. Return two values set *K* and *simpling.*

IV. *def compute_simplitig (K, k)*

**Output**: List containing all the maximum simplitigs.

**Purpose**: identify all the longest contiguous segments of the de Bruijn graph for a given *k* and *K.*

**Algorithm:**

1. Initialize an empty list *maximal_simplings.*
2. Enter while loop until the length of the set *K* is greater than 0.
    a. Assign element from the top of the set to *seeding_kmer*. Remove elements from the set *K.*
    b. Call *compute_maximum_simplitig_from_kmer* with input parameters: *K*, *seeding_kmer*, *k.*
    c. Append the result to the *maximal_simplings* list.
3. Return the *maximal_simplings* list.

## Greedy_AC.py[10]

This module is used for construction of superstrings  by using the Greedy algorithm with an Aho-Corasick automaton.

Consists of four functions:

I. *def preprocessing (kmers, automaton)*

**Input:**

- *kmers* - set of strings (kmers)
- *automaton* - AC machine

**Output**:

- *list_L* - a dictionary where the keys are the states of the AC machine and the values are lists of indices of the kmers that pass through that state.
- *link_B* - a dictionary where the keys are the states of the AC machine and the values are the states of the suffix link of the state.
- *pointer_B* - the state of the AC machine corresponding to the root of the suffix tree.
- *state_F* - a dictionary where the keys are the indices of the kmers and the values are the states of AC machine where kmer ends.
- *depth* - a dictionary where the keys are the states of the AC machine and the values are the depths of those states in the suffix tree.

**Purpose**:Preprocess the set of kmers in other data structures that will ne used in Hamiltonian path algorithm.

**Algorithm:**

1. Initialize empty dictionaries: *lisl_L* (to store multiple values with one key), *state_F* (to store a finite state for each word), *inverse_E* (to create an inverse list of *state_F*), *depth*( distance from starting state to the current state) and *link_B* (for father generation of Hamiltonian path).
2. Loop over each *kmer* in the set *kmers*.
    a. Set the current *state* to zero.
    b. Transite to the next *state* using the *goto* automaton function.
    c. Add obtained value to the *list_L* corresponding to the new *state* containing the index of the current *kmer.*

[10] https://github.com/Ekatmil/Efficient-representation-of-k-mers-sets/blob/8f2d5ce215194bf419bfbcf7bb770758e5c4db0d/Greedy_AC.py

    d. If the current character is the last character of the current *kmer*, its finite *state* is updated to the current *state*,. and the corresponding inverse value of the finite state is also updated with the current *kmer*.
        i. If the current state is not a leaf, then update its finite state to zero (there will be no suffix string from this state).
3. Apply BFS (Breadth-first search) on the AC machine. Traverse all the *states*, adding the *depth* and linked list *B*.
4. Return the preprocessed data structures such as *list_L*, *link_B*, *pointer_B*, and *state_F*.

II. *def Hamiltonian (list_L, link_B, pointer_B, state_F, automaton, n)*

**Input:**

- Data structures obtained by the preprocessing function
- *automaton* - AC machine
- *n* - an integer, representing number of kmers

**Output**: Dictionary H, representing constructed Hamiltonian path.

**Purpose**: Find Hamiltonian cycles in a directed graph using Aho-Corasick automaton.

**Algorithm:**

1. Initialize a dictionaries *list_P* (to store where each failed state of the nodes AC machine corresponds to a list of indexes of failed words this state represents), dictionary *forbidden* (to store where each index of the word in the input set corresponds to a boolean value representing whether the word is a subword of some other word), dictionary *first* (to store each prefix string corresponds to a state in the AC machine), dictionary *last* (to store each suffix string), dictionary *H* (will be used for merge operations).
2. Set all values of *forbidden* dictionary to be *False* using *initializeForbidden.*
3. Loop over each *kmer* in the set and add it to the *list_P* and change the *forbidden* value for the word according to their fail state (forbidden set to *True* if the kmer is the substring and therefore its fail is not to state 0).
4. Set the current state as *pointer_B.*
5. Enter while loop until current *state* is not 0 (root).
    a. If the current *state* has a non-empty value in *list_P*. If it does then:
        i. Go through all of the values until find the one element from which the cycle could start (kmer is not a subword and has value *False* in *forbidden*).
            1. Add such element to *H.*
            2. Set *forbidden* for such element to be *True*.

                    3. Remove element from *list_P*.
                    4. Update dictionaries *first* and *last* to reflect the new set of kmers.
                    5. If *list_P* does not contain any elements then break the loop.
            ii. Adds multiple values to the *list_P* dictionary, assigning the second argument of that function call to the key located at *automaton.fail[state]*.
        b. Update state to value in *link_B* corresponding to the key state.
6. Return *H*, which contains a Hamiltonian path in the graph.

III. *def initialization (a)*

**Input:** *a* - a list, representing set of kmers

**Output**: dictionary *H*, containing Hamiltonian path.

**Purpose**: Creates automaton *A* and constructs the Hamiltonian path.

**Algorithm:**

1. Create AC machine *A*.
2. Call *preprocessing* function with parameters *a* (list of kmers) and *A* (AC machine) to obtain parameters.
3. Return the result of *Hamiltonian* function with obtained parameters.

IV. *def FindSuperSet (kmer_set)*

**Input:** *kmer_set* - a set of kmers

**Output**: set of superstrings.

**Purpose**: find the set of superstrings with zero overlap between each other.

**Algorithm:**

1. Convert the *kmer_set* into the list *lst.*
2. Initialize empty set *outputSet.*
3. Run *initialization* with input *lst* to obtain dictionary *H* of the Hamiltonian path.
4. Call *findSingle* to obtain *single_list* - list of states with indegree 0.
5. Call *addtoSet* with input *lst*, *single_list*, *H* and *outputSet.*
6. Return the *outputSet.*

## tgreedy.py[11]

This module is used for construction of superstring  by using the Tgreedy algorithm with an Aho-Corasick automaton.

Consists of three functions:

I. *def HamiltonianT (list_L, link_B, pointer_B, state_F, automaton, n)*

**Input:**

- *list_L* - a dictionary where the keys are the states of the AC machine and the values are lists of indices of the kmers that pass through that state
- *link_B* - a dictionary where the keys are the states of the AC machine and the values are the states of the suffix link of the state
- *pointer_B* - the state of the AC machine corresponding to the root of the suffix tree
- *state_F* - a dictionary where the keys are the indices of the kmers and the values are the states of AC machine where kmer ends
- *depth* - a dictionary where the keys are the states of the AC machine and the values are the depths of those states in the suffix tree
- *automaton* - AC machine
- *n* - an integer, representing number of kmers

**Output**: Dictionary H, representing constructed Hamiltonian path.

**Purpose**: Find Hamiltonian cycles in a directed graph using Aho-Corasick automaton.

**Algorithm:**

1. Initialize a dictionaries *list_P* (to store where each failed state of the nodes AC machine corresponds to a list of indexes of failed words this state represents), dictionary *forbidden* (to store where each index of the word in the input set corresponds to a boolean value representing whether the word is a subword of some other word), dictionary *first* (to store each prefix string corresponds to a state in the AC machine), dictionary *last* (to store each suffix string), dictionary *H* (will be used for merge operations).
2. Set all values of the forbidden dictionary to be *False* using *initializeForbidden.*
3. Loop over each *kmer* in the set and add it to the *list_P* and change the *forbidden* value for the word according to their fail state (forbidden set to *True* if the kmer is the substring and therefore its fail is not to state 0).
4. Set the current state as *pointer_B.*
5. Enter while loop until current *state* is not 0 (root).

[11] https://github.com/Ekatmil/Efficient-representation-of-k-mers-sets/blob/8f2d5ce215194bf419bfbcf7bb770758e5c4db0d/tgreedy.py

        a. If the current *state* has a non-empty value in *list_P*. If it does then:
            i. Retrieve the list of words associated with the current *state* from *listy_L* dictionary.
            ii. Iterate over the list of words and perform the following checks:
                1. Verify if the word j is not forbidden (i.e., it is not a subword).
                2. If the word j is not forbidden, continue with the following steps:
                    a. Check if the *first[i]*, where *i* is first element in the *list_P[state]*, is equal to *j*. If they are equal, remove *i* from *list_P* and mark *j* as forbidden. This step avoids adding unnecessary overlaps to the superstring.
                    b. If the *first[i]* is not equal to *j*, add the pair *(i, j)* to the *H* dictionary, indicating an overlap between the words.
                    c. Update the forbidden status of word *j* to *True*, signifying that it is now forbidden.
                    d. Update the *first* and *last* dictionaries to reflect the new positions of the words.
                    e. Remove the word *i* from l*ist_P[state]*.
            iii. Update the *list_P* dictionary by adding the current *list_P[state]* to the fail state of the current state.
        b. Update state to value in *link_B* corresponding to the key state.
    6. Return *H*, which contains a Hamiltonian path in the graph.

II. *def initialization (a)*

**Input:** *a* - a list, representing set of kmers

**Output**: dictionary *H*, containing Hamiltonian path

**Purpose**: Creates automaton *A* and construct the Hamiltonian path

**Algorithm:**

1. Create AC machine *A*.
2. Call *preprocessing* function (import from Greedy_AC) with parameters *a* (list of kmers) and *A* (AC machine) to obtain parameters.
3. Return the result of *HamiltonianT* function with obtained parameters.

III. *def FindSuperSetTgreedy (kmer_set)*

**Input:** *kmer_set* - a set of kmers

**Output**: set of superstrings.

**Purpose**: find the set of superstrings with zero overlap between each other.

**Algorithm:**

1. Convert the *kmer_set* into the list *lst.*
2. Initialize empty set *outputSet.*
3. Run *initialization* with input *lst* to obtain dictionary *H* of the Hamiltonian path.
4. Call *findSingle* to obtain *single_list* - list of states with indegree 0.
5. Call *addtoSet* with input *lst*, *single_list*, *H* and *outputSet.*
6. Run function *FindSuperSet* of Greedy_AC on *outputSet* to produce *outputSet_final*
7. Return *outputSet_final.*