The Shortest Superstring Problem

Theodoros P. Gevezes and Leonidas S. Pitsoulis

An **alphabet** is a finite non-empty set whose elements are called **letters**. A **string** is a sequence of letters. Given two strings s_i and s_j, the second is a **substring** of the first if s_i contains consecutive letters that match s_j exactly. We say that s_i is a **superstring** of s_j. The **Shortest** (**common**) **Superstring Problem** (SSP) is a combinatorial optimization problem that consists in finding a shortest string which contains as substrings all of the strings in a given set. The strings of the set may be overlapping inside the superstring exploiting their common data.

1 Applications

The SSP has several important applications in various scientific domains and this is the reason why it has attracted the interest of many researchers. In computational molecular biology, the DNA sequencing procedure via fragment assembly can be formulated as SSP. In virology, the SSP models the compression of viral genome. In information technology, the SSP can be used to achieve data compression. In scheduling, SSP solutions can be used to schedule operations in machines with coordinated starting times. In the field of data structures, efficient storage can be achieved in specific cases using the solutions of the SSP.

T.P. Gevezes (✉) • L.S. Pitsoulis
Faculty of Engineering, School of Electrical and Computer Engineering, Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece
e-mail: theogev@gen.auth.gr; pitsouli@gen.auth.gr

T.M. Rassias et al. (eds.), *Optimization in Science and Engineering: In Honor of the 60th Birthday of Panos M. Pardalos*, DOI 10.1007/978-1-4939-0808-0_10,
© Springer Science+Business Media New York 2014

1.1 DNA Sequencing

The molecule of the DNA encodes the genetic information used in the developing and functioning of living beings. DNA is a double-stranded sequence of four types of nucleotides: adenine, cytosine, guanine and thymine, and thereby it can be viewed as a string over the alphabet $\{a, c, g, t\}$. In the field of molecular biology, the DNA sequencing procedure determines the sequence of a DNA molecule, that is the precise order of the nucleotides within it. DNA sequencing highly accelerates biological and medical research.

Due to laboratory equipment constraints, only parts of DNA up to few hundred nucleotides can be read reliably, while the length of the DNA molecule in many species is quite longer. To recognize a long DNA sequence, many copies of the DNA molecule are made and cut into smaller overlapping pieces, named **fragments**, that can be read at once. Each fragment is chosen from an unknown location of the molecule. To reconstruct the initial DNA molecule, these fragments must be reassembled in their initial order, a procedure known as the **DNA assembly problem**. Due to the huge amount of data generated by the fragment sequencing methods, an automated procedure supported by a computer software is necessary for the assembly process. Intuitively, shortest superstrings of the sequenced fragments preserve important biological structures [33, 46, 53], and in practice they are proved to be good representations of the original DNA molecule [27, 34]. Therefore, the SSP can be considered as an abstraction of the assembly problem, and consequently many researchers developed assembly methods based on it [18, 45, 51]. The most widely used of them, the **shotgun sequencing**, is essentially the natural greedy algorithm for the SSP. Similar assembly problems arise during reconstruction of RNA molecules or proteins from sequenced fragments.

1.2 Data Compression

In the fields of computer science, information technology and data transmission, a crucial issue is the size of the stored or transferred data. Data compression is the process of encoding data using fewer bits than their original representation. According to whether the compressed data is exactly as the original data or not, we distinguish the **lossless compression** and the **lossy compression**, respectively (see [50]).

Considering data as text over an alphabet, an intuitive method of lossless compression is based on the idea of dividing the text into strings and representing it by a superstring of these strings with pointers to their original positions. Based on this principle, several macro schemes concerning the nature of the pointers are taken under consideration in [55, 56], leading in general applications of the SSP in the field of textual substitution. In programming languages, each alphanumerical string in the code may be represented as a pointer to a common string stored in the memory. Therefore, the target of the compiler is to arrange the alphanumerical strings in such a way that they overlap as much as possible [15, 37]. Other general applications of the SSP on data compression are discussed in [14, 54].

1.3 Modelling the Viral Genome Compression

Viruses are forced to reduce their genome size by environmental factors such as the need for quick replication and the small amount of nucleic acid that can be incorporated in them. One way to compress their genome is by overlapping their genes. Genes are the parts of the DNA that specify all proteins in living beings. Between genes there are generally long sequences of nucleotides that do not be coded into proteins. On the other hand, overlapping genes are common in viruses. Therefore, in most virus species, two or more proteins are coded by the same nucleotide sequence, allowing viruses to increase their repertoire of proteins without increasing their genome length, as indicated in [10].

In [24, 25], the SSP is used to model the viral genome compression. The genes are considered as strings and the purpose is to find a shortest superstring that contains them all. The computational results show that the amount of compression achieved by the viruses in the real world is the same or very close to the one obtained by the algorithms in all the examples considered in [24, 25]. Another conclusion from these computations is that the average compression ratio of viruses is remarkably high considering the fact that the DNA molecules are very difficult to compress in general. Finally, by modelling the viral genome compression as SSP, any exact solution or lower bound of the corresponding SSP instance provides a bound on the real size of a viral genome with a given set of genes.

1.4 Scheduling with Coordinated Starting Times

The **Flow Shop Problem** (FSP) and the **Open Shop Problem** (OSP) concern the scheduling of operations in machines and have particular applications in scheduling and planning of experiments. Given a set of k machines $M_1, M_2, \ldots, M_k$ the problem is to schedule a set of jobs on them, where each job consists of k operations and the i-th operation has to be assigned on the machine M_i. A machine can process at most one operation at a time, and any two operations of a job cannot be processed simultaneously. In the FSP, the operation on M_i has to be finished before the operation on M_{i+1} can start for each job, whereas in the OSP there is not such commitment. In the *no-wait* versions of these problems, it is required the operations of a job to be processed directly one after the other. The optional constraint of *coordinated starting times* necessitates an operation starting on one machine only when each of the other machines is either idle or also starts an operation. In all these cases, the task is to find a schedule such that the overall processing time is minimized.

The FSP and the OSP on two machines, and their corresponding no-wait versions are polynomially solvable in general, but this is not always true when the machines have to coordinate the starting times of operations. In [39], this additional constraint is considered. Each instance of the no-wait version of these problems under the additional constraint of the coordinated starting times can be transformed into an SSP instance, where all strings are of a special form. The NP-completeness of these

problem versions is proved using this transformation. Apart from the computational complexity, this transformation can be applied for solving the constrained FSP and OSP. Each exact and heuristic algorithm for the SSP can be applied to these problems too. Also, the special case of the SSP can be used to derive approximation algorithms for the constrained shop problems.

1.5 Data Structure Storage

In [15], a special case of the SSP is considered, where all the strings are of length at most two. It is proved that this version of the SSP is solvable in polynomial time. This SSP case has applications to the storage of data structures, and specifically to the **Huffman trees** [23] that encode pairs of letters, which are used to an entropy encoding algorithm for lossless data compression, and for efficient representation of directed graphs in memory.

2 Definitions and Notations

Let $\mathbb{N}$ be the set of natural numbers including 0. All the numbers in this chapter are natural, unless otherwise stated. For a real x, $\lceil x \rceil$ denotes the smaller integer greater than or equal to x. For a letter l, the notation $l \in \Sigma$ means that l belongs to alphabet Σ, while for a string s, if all letters of s belong to Σ, we say that s is *over* the alphabet Σ. If s is a string, then $|s|$ denotes its length, that is the number of its letters, while if S is a set, then $|S|$ denotes its cardinality. For a string s and $i, j \in \mathbb{N}$ such that $1 \leq i \leq j \leq |s|$, the substring of s from i-th to j-th letter is denoted by $s_{[i,j]}$. Any substring $s_{[1,j]}$ is a **prefix** of s, and if $j < |s|$, then it is called a **proper prefix**. Similarly, any substring $s_{[i,|s|]}$ is a **suffix** of s, and if $i > 1$, then it is called a **proper suffix**.

The placement of two or more strings one next to the other denotes their **concatenation**, e.g. $s_i s_j$ is the concatenation of s_i and s_j. A **coverage string** between strings s_i and s_j, in this specific order, is a string v such that $s_i = uv$ and $s_j = vw$, for some non-empty strings u, w. In other words, v is a string that is a proper suffix of s_i *and* a proper prefix of s_j. The length of the coverage string is called **coverage** between the corresponding strings and is a non-negative integer. A **join string** of s_i and s_j is the concatenation of these two strings with a coverage string appearing only once, that is uvw. We use $J_{\{s_i,s_j\}}$ to denote the set of all join strings of s_i and s_j *regardless* their order.

The **overlap string** between s_i and s_j is their longest coverage string, and is denoted by $o(s_i,s_j)$. Its length $|o(s_i,s_j)|$ is called **overlap**. The overlap of a string with itself is called **self-overlap**, and notice that it is not limited to half the total string length. The **merge string** of s_i and s_j is the concatenation of these two strings with the overlap string appearing only once, that is the shortest join string between

them. It is denoted by $m(s_i, s_j)$. We have $|m(s_i, s_j)| = |s_i| + |s_j| - |o(s_i, s_j)|$. The length of the prefix of s_i before the overlap string with s_j is called **distance** from s_i to s_j and is denoted by $d(s_i, s_j)$.

Example 1. Suppose that we have the strings $s_1 = bbacb$ and $s_2 = bcabbcabb$ over the alphabet $\{a, b, c\}$, so $|s_1| = 5$ and $|s_2| = 9$. The one-letter string b is a proper suffix of s_1 and a proper prefix of s_2. Moreover, it is the longest such string, and thus the overlap string between them, $o(s_1, s_2) = b$, with overlap 1. The coverage strings between s_2 and s_1 are b and bb, and so $o(s_2, s_1) = bb$ with overlap 2. The self-overlap of the first string is $|o(s_1, s_1)| = 1$, while $|o(s_2, s_2)| = 5$. The corresponding merge strings are $m(s_1, s_2) = bbacbcabbcabb$, $m(s_2, s_1) = bcabbcabbacb$, $m(s_1, s_1) = bbacbbacb$, and $m(s_2, s_2) = bcabbcabbcabb$. The distance from s_1 to s_2 is $d(s_1, s_2) = 4$, while $d(s_2, s_1) = 7$, $d(s_1, s_1) = 4$, and $d(s_2, s_2) = 4$. Finally, the set of all join strings is $J_{\{s_1, s_2\}} = \{bbacbbcabbcabb, bcabbcabbbbacb, bbacbcabbcabb,$ $bcabbcabbbacb, bcabbcabbacb\}$. □

Given a finite set S of strings over an alphabet Σ, the sum of lengths of the strings in S is defined as $||S|| = \sum_{s \in S} |s|$. The **orbit size** of a letter $l \in \Sigma$ is the number of its occurrences in the strings of S.

An instance of the SSP is specified by a finite set $S = \{s_1, s_2, \ldots, s_n\}$ of strings. A string s is a superstring of S, if it is a superstring of all $s_i \in S$. A **multiset** is a generalization of the notion of the set where elements are allowed to appear more than once. Without loss of generality, S is defined to be a set since if S is a multiset, then S has exactly the same superstrings as the set $\{s : s \in S\}$. Also, it is assumed that S is a **substring-free set**, i.e., no string $s_i \in S$ is a substring of any *other* string $s_j \in S$. This assumption can be made without loss of generality, since for any set of strings there exists a unique substring-free set that has the same superstrings, obtained by removing any string is a substring of another.

Given a set $S = \{s_1, s_2, \ldots, s_n\}$ of strings over an alphabet Σ, the SSP is the problem of finding a minimum length superstring of S. Note that such a string may not be unique. The length of a shortest superstring of S is denoted by $\mathrm{opt}_l(S)$, while the corresponding achieved **compression** is defined as $\mathrm{opt}_c(S) = ||S|| - \mathrm{opt}_l(S)$. The decision version of the SSP is described as follows. Given a set S of strings and a $k \in \mathbb{N}$, is there a superstring s of S such that $|s| = k$?

Example 2. Suppose that we have the multiset $S' = \{s_1, s_2, s_3, s_4, s_5, s_6\}$ of strings over the alphabet $\{a, b, c\}$, where $s_1 = bababbc$, $s_2 = bbccaac$, $s_3 = bbcaaabb$, $s_4 = acabb$, $s_5 = bcaaab$, and $s_6 = acabb$. The corresponding substring-free set is $S = \{s_1, s_2, s_3, s_4\}$ with $|S| = 4$ and $||S|| = 27$. The orbit size of the letter a in S is 9, of the letter b is 12, and of the letter c is 6. These are the two shortest superstrings of S: $s = bababbccaacabbcaaabb$ and $s' = bababbcaaabbccaacabb$, with $\mathrm{opt}_l(S) = |s| = |s'| = 19$ and $\mathrm{opt}_c(S) = 7$. □

Let I_n be the finite set $\{1, 2, \ldots, n\}$, and Π_n be the set of all permutations of the set I_n. Any solution for the SSP of n strings can be represented as a permutation $p \in \Pi_n$, indicating the order in which strings must be merged to get the superstring.

It is implied that the shortest superstrings are derived only by string merges. If this is not the case, there would be parts of the superstring that do not correspond to any string, or some consecutive strings would not exploit their longest coverage string and could be joined by a larger coverage. In both cases there would be a shorter superstring. The elements of a permutation $p \in \Pi_n$ are denoted by $p(i)$, $i \in \mathrm{I}_n$, where i indicates the order of each element in p such that $p = (p(1), p(2), \ldots, p(n))$.

Given an order of strings $(s_1, s_2, \ldots, s_n)$ the superstring $s = \langle s_1, \ldots, s_n \rangle$ is defined to be the string $m(s_1, m(s_2, \ldots m(s_{n-1}, s_n) \ldots))$. In such an order, the first string s_1 is denoted by first(s) and the last string s_n is denoted by last(s). Notice that s is the shortest string such that $s_1, s_2, \ldots, s_n$ appear in this order as substrings.

For a set $S = \{s_1, s_2, \ldots, s_n\}$ of strings and a permutation $p \in \Pi_n$, the corresponding superstring is defined as $strSp(S, p) = \langle s_{p(1)}, s_{p(2)}, \ldots, s_{p(n)} \rangle$. For any SSP instance $S = \{s_1, s_2, \ldots, s_n\}$, there exists a permutation $p \in \Pi_n$ such that $strSp(S, p)$ is an optimal solution. For any $p \in \Pi_n$, the length of the superstring $strSp(S, p)$ is given by $|strSp(S, p)| = \sum_{i=1}^{n} |s_i| - \sum_{i=1}^{n-1} |o(s_{p(i)}, s_{p(i+1)})|$. Therefore, the SSP can be formulated as

$$\min_{p \in \Pi_n} \sum_{i=1}^{n} |s_i| - \sum_{i=1}^{n-1} |o(s_{p(i)}, s_{p(i+1)})|. \tag{1}$$

The shortest superstrings that correspond to the permutation p of the optimal solution have length equal to $\mathrm{opt}_l(S)$. A superstring of the minimum length is achieved when the sum of the overlaps between consecutive strings, in the order defined by p, is maximized.

There are two ways to assess the solution quality of a non-exact algorithm for the SSP: the **length measure** and the **overlap** or **compression measure**. According to the first measure, a superstring is better when its length is shorter. In this case, the SSP is described as a minimization problem. According to the second measure, a superstring is better when the achieved compression is greater. In this case, the problem is described as a maximization problem. The two measures are equivalent when applied to exact solutions, but they give different results when they measure the relative preciseness of non-exact solutions obtained by approximation or heuristic algorithms. A good algorithm with respect to one of the above measures is not necessarily a good algorithm with respect to the other measure.

Example 3. For the substring-free set $S = \{s_1, s_2, s_3, s_4\}$ of Example 2 and the two shortest superstrings of it, $s = \langle s_1, s_2, s_4, s_3 \rangle$ and $s' = \langle s_1, s_3, s_2, s_4 \rangle$, we have first$(s) =$ first$(s') = s_1$, last$(s) = s_3$, and last$(s') = s_4$.

Let $s'' = strSp(S, p)$ for the permutation $p = (3, 4, 2, 1)$, which is a superstring of length $|s''| = 24$. According to the length measure the solution s'' is $(|s''| - \mathrm{opt}_l(S))/\mathrm{opt}_l(S) = 26.3\,\%$ far from the optimal length, while according to the compression measure is $(\mathrm{opt}_c(S) - (||S|| - |s''|))/\mathrm{opt}_c(S) = 71.4\,\%$ far from the optimal compression. □

A **directed graph** G is defined by a vertex set $V(G)$ and an arc set $E(G)$ which contains ordered pairs of vertices and is denoted by $G = (V, E)$. For an arc $e = (u, v)$, u is called the **tail** of e, and v the **head** of e. We say that e is **incident** to both vertices,

while for v the arc e is an **incoming arc**, and for u is an **outgoing arc**. An arc with the same tail and head is called a **loop**. For a vertex $v \in V$, the number of incoming arcs of v is denoted by $\deg^-(v)$, and the number of outgoing arcs of v is denoted by $\deg^+(v)$. The overall number of the incident arcs to a vertex v regardless of their direction is the **degree** of v. The degree of a graph is the maximum degree between its vertices. Graph G is **complete** if there is an arc (u,v) for any vertex pair $u,v \in V$, $u \neq v$. For a weight function $w : E \to \mathbb{N}$, we denote by $G = (V,E,w)$ a weighted directed graph. When there is no confusion we denote by w_{ij} the weight of arc $e = (v_i,v_j) \in E$. If the elements of set E have no direction, then they are called **edges** and the corresponding graph is called **undirected**. An undirected graph is called **bipartite** if its vertex set can be partitioned into two subsets, V_1 and V_2, such that every edge is incident to a vertex of V_1 and to a vertex of V_2. If the arc set contains ordered tuples instead of pairs of vertices then we have a **multigraph**.

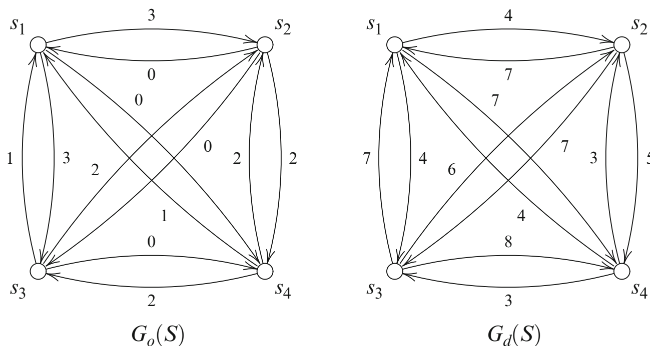
Given a set $S = \{s_1,s_2,\ldots,s_n\}$ of strings, the complete directed weighted graph $G = (V,E,w)$ with

- vertex set $V = \{s_1,s_2,\ldots,s_n\}$,
- arc set $E = \{(s_i,s_j) : s_i,s_j \in V, i \neq j\}$, and
- weight function $w : E \to \mathbb{N}$, with $w_{ij} = |o(s_i,s_j)|$,

is called the **overlap graph** of S and is denoted by $G_o(S)$. If the arc weight function depends on the distance instead of the overlap between the string pairs, that is $w_{ij} = d(s_i,s_j)$, then the corresponding graph is called the **distance graph** of S, and is denoted by $G_d(S)$. Notice that all weights on both graphs are non-negative integers. In the following, it is assumed that the overlap and distance graphs have no loops, unless otherwise stated. For any set $A \subseteq E$ of arcs on both graph, we denote by $o(A)$ the sum of weights of the arcs on $G_o(S)$, that is their total overlap, and by $d(A)$ the sum of weights of the arcs on $G_d(S)$, that is their total distance. For each arc $e = (s,s')$ on both graphs, we have

$$|s| = o(\{e\}) + d(\{e\}). \tag{2}$$

Example 4. For the substring-free set $S = \{s_1,s_2,s_3,s_4\}$ of Example 2 the associated overlap and distance graphs are depicted in the next figure.



For the arc set $A = \{(s_1,s_2),(s_3,s_4)\}$, we have $o(A) = 3$ and $d(A) = 12$. □

A **walk** on a directed graph is a sequence of arcs where the head of each arc except the last one is the tail of the next arc. A walk can be specified either by its vertices or its arcs in the order of appearance in it. A **path** on a directed graph is a walk with no repeating vertices. On an undirected graph, a path is a sequence of consecutive edges that connect no repeating vertices. A walk is called **Eulerian** if it contains all the *arcs* of the graph, while a path is called **Hamiltonian** if it contains all the *vertices* of the graph. A **cycle** is a path where the first and the last vertices are the same. A cycle with k arcs is called a k-cycle. For a string set S and the associated overlap and distance graphs, consider a cycle c on them, a string $s \in S$ corresponds to a vertex of c, and let s' be the unique previous string of s in c. The superstring $\langle s, \ldots, s' \rangle$ where the strings are in the order around c is called the **cycle superstring of** c **with respect to** s and is denoted by $strC(c,s)$. The superstring $\langle s, \ldots, s', s \rangle$ where the strings are in the order around c is called the **extended cycle superstring of** c **with respect to** s and is denoted by $strC^{+}(c,s)$. Notice that $strC^{+}(c,s) = m(strC(c,s), s)$.

Example 5. For the substring-free set $S = \{s_1, s_2, s_3, s_4\}$ of Example 2 and the associated overlap and distance graphs presented in Example 4, consider the cycle $c = (s_1, s_2, s_4, s_1)$. We have $strC(c, s_1) = bababbccaacabb$, and $strC^{+}(c, s_1) = bababbccaacabbababbc$. □

Some combinatorial optimization problems are closely related to the SSP due to their nature and are used in the establishment of many results of the SSP. A **matching** on a directed graph is a set of arcs, no two of which are incident to the same vertex. A **maximum matching** on a weighted graph is a matching with the largest total weight, while the **Matching Problem** (MP) looks for a maximum matching on a weighted directed graph. The MP is defined similarly on undirected weighted graphs. A **directed matching** is a set of arcs, no two of which have the same tail or the same head. In other words, it is a set of disjoint paths and cycles on a graph. The **Directed Matching Problem** (DMP) looks for a maximum directed matching. Both MP and DMP can be solved in polynomial time (see, e.g., [42, 59]). A **cycle cover** on a directed graph is a set of cycles such that each vertex of the graph is in exactly one cycle. The **Cycle Cover Problem** (CCP) on a weighted directed graph consists in finding a cycle cover with maximum total weight. The CCP is solvable in polynomial time by reduction to the MP on bipartite graphs (see, e.g., [42]).

The **Hamiltonian Path Problem** (HPP) on a weighted directed graph consists in finding an optimal Hamiltonian path according to its total weight. If the objective is to minimize the total weight, then the Min-HPP is considered, while if the objective is to maximize the total weight, then the Max-HPP is considered. The decision HPP on a directed graph G asks for the existence of a Hamiltonian path on G. Similarly, we have the maximization and the minimization **Hamiltonian Cycle Problem**, which are also known as **Traveling Salesman Problems** (Min-TSP and Max-TSP). Both HPP and TSP are NP-hard problems [30]. There is a simple relation between these problems. The HPP on a graph G can be transformed to the TSP on a graph G' obtained from G by adding a new vertex u and zero-weighted arcs from u to each vertex of G and from each vertex of G to u.

3 Computational Complexity

The results described in this section concern the computational complexity of the SSP, and justify the fact that there are only few exact algorithms, and on the other hand so many approximation algorithms for it. The SSP cannot be solved efficiently to optimality in polynomial time. It can be approximated within a constant ratio, whereas this ratio has a bound.

3.1 Complexity of Exact Solution

Given a string set S and a string s, there is a polynomial time algorithm for checking if s is a superstring of S, and therefore the decision SSP belongs in class NP.

A string is **primitive** if no letter appears more than once into it. Next theorem establishes the NP-completeness of the decision SSP.

Theorem 1 ([15]). *The decision SSP is NP-complete. Furthermore, this problem is NP-complete even if for any integer $m \geq 3$ the restriction is made that all strings in set S are primitive and of length m.*

The proof is based on a polynomial time transformation from the decision HPP on directed graphs with the following additional restrictions:

- there is a designated start vertex s with $\deg^{-}(s) = 0$ and a designated end vertex t with $\deg^{+}(t) = 0$,
- for each $v \neq t$, we have $\deg^{+}(v) > 1$.

A set S of specific strings of length 3 is constructed, and each string is corresponding to a vertex of a directed graph $G = (V, E)$ that satisfies the above restrictions. Graph G has a Hamiltonian path if and only if set S has a superstring of length $2|E| + 3|V|$. Therefore, there is no efficient algorithm for solving the SSP, unless $\mathrm{P} = \mathrm{NP}$.

Due to the nature of the SSP, several parameters can be considered fixed in order to define restricted cases of the problem. Besides the length of the strings and the primitiveness that were mentioned previously, the cardinality of the alphabet, the orbit size of the letters, and the form of the strings were also examined for the conservation or not of the NP-completeness.

The decision SSP remains NP-complete when it is restricted to an alphabet of cardinality 2 as proved in [15]. A restricted version of the SSP concerning both the alphabet cardinality and the string length is also studied and the result is stated in the next theorem. Let $bits(n)$ denote the number of bits that are necessary to represent n in binary, for any $n \in \mathbb{N}$.

Theorem 2 ([15]). *The decision SSP is NP-complete even if for any real $h > 1$, the strings in set S are written over the alphabet $\{0, 1\}$ and have length $\lceil h\, bits(||S||) \rceil$.*

The proof is based on Theorem 1 and on the encoding of each letter of the initial alphabet with letters of the alphabet $\{0,1\}$ such that no relative changes yielded to the overlaps between the strings after the new encoding.

In [38, 39], NP-completeness results are proved for some special cases of the decision SSP. For a set S of strings over an alphabet Σ, these complexity results can be briefly presented as follows. The decision SSP is NP-complete even if

- all strings in S are of length 3 and the maximum orbit size of each letter in Σ is 8.
- all strings in S are of length 4 and the maximum orbit size of each letter in Σ is 6.
- $\Sigma = \{0,1\}$ and each string in S is of the form $0^p10^q10^r1$ or $10^p10^q10^r$, where $p,q,r \in \mathbb{N}$.
- $\Sigma = \{0,1\}$ and all strings in S are of the form 10^p10^q, where $p,q \in \mathbb{N}$.
- $\Sigma = \{0,1,2\}$ and each string contains a fixed number of each letter.

3.2 Complexity of Approximation

Since the SSP is a hard problem to be solved to optimality, a huge amount of effort is made to develop approximation algorithms. The theoretical framework for the complexity of this aspect establishes that although the SSP is easy to be approximated within *some* constant ratio, it is hard to be approximated within *any* constant ratio. The **linear reduction** (L-reduction) is necessary for what follows.

Definition 1 ([43]). Let A and B be two optimization problems. Problem A L-reduces to B if there are two polynomial time algorithms F and G and real constants $\alpha, \beta > 0$ such that

- given an instance a of A, algorithm F produces an instance b $=$F(a) of B such that opt(b) is at most $\alpha \times$ opt(a), where opt(a) and opt(b) are the costs of the optimal solution of instances a and b respectively, and
- given any solution of b with cost c', algorithm G produces in polynomial time a solution of a with cost c such that $|c - \text{opt}(a)| \leq \beta |c' - \text{opt}(b)|$.

For two optimization problems A and B and the constants α and β of the Definition 1, the following theorem establishes the basic usage of L-reduction.

Theorem 3 ([43]). *If problem A L-reduces to problem B and there is a polynomial time approximation algorithm for B with worst-case error ε, then there is a polynomial time approximation algorithm for A with worst-case error $\alpha\beta\varepsilon$.*

Therefore, if problem B has a polynomial time approximation scheme (PTAS), then so does problem A.

The class Max-SNP is a class of optimization problems defined syntactically in [43]. Every problem in Max-SNP can be approximated in polynomial time within some constant ratio. A problem is Max-SNP-hard if any other problem in Max-SNP L-reduces to it.

Theorem 4 ([8]). *The SSP is Max-SNP-hard.*

The proof is based on an L-reduction from the Min-HPP, where the degree of the associated directed graph is bounded, and all the weights are either 1 or 2, which is Max-SNP-hard [44]. The reduction from this problem to the SSP is similar to the one used to show the NP-completeness of the decision SSP in Theorem 1, with the extra establishment that it is an L-reduction. The strings that are considered for the above L-reduction have bounded lengths, and so the same reduction can be applied to the maximization version of the superstring problem with respect to the compression measure, and concludes to the same hardness result.

Corollary 1 ([8]). *Maximizing the total compression of a string set is Max-SNP-hard.*

In [5], it is proved that if a Max-SNP-hard problem has a PTAS, then P = NP. Therefore, there is no PTAS for the SSP, unless P = NP, which means that there exists an $\varepsilon > 0$ such that it is NP-hard to approximate the SSP within a ratio of $1+\varepsilon$.

The L-reduction described in [8] for the proof of the Max-SNP-hardness of the SSP produces instances with arbitrarily large alphabets. More precisely, each instance of the special Min-HPP with n vertices is transformed to an SSP instance over an alphabet with $2n+1$ letters. However, the SSP is APX-hard even if the alphabet contains just two letters as stated in the next theorem.

Theorem 5 ([41]). *The SSP is APX-hard both with respect to the length measure and the compression measure, even if the alphabet has cardinality* 2 *and every string is of the form* $10^m1^n01^m0^{n+4}10$ *or* $01^m0^n10^p1^q01^m0^n10^r1^s01$*, where* $m,n,p,q,r,s \geq 2$.

4 Polynomially Solvable Cases

Since the SSP is NP-hard, special cases of the problem that can be solved in polynomial time constitute an interesting aspect. Various additional restrictions on the problem's parameters, similar to these described in Sect. 3 lead to polynomial algorithms revealing the boundaries between hard and easily solvable cases of the problem.

Obviously, if the cardinality of the alphabet is equal to 1 or all the strings in the given set are of length 1, then the SSP is trivial. Also, if the number of the strings in the set is fixed, then the SSP is polynomially solvable by enumerating all the different string orders. However, there are more interesting and complicated polynomial cases of the SSP.

Since Theorem 1 establishes the NP-completeness of the SSP for string lengths greater than 2, the question is what happens in the remaining cases. The answer is given by the next theorem.

Theorem 6 ([37]). *For a string set $S = \{s_1, s_2, \ldots, s_n\}$ and an integer k, if $|s_i| \leq 2, i \in \mathrm{I}_n$, then there is a linear time and space algorithm to decide if S has a superstring of length k.*

A **path decomposition** of a directed graph G is a partition of $E(G)$ into edge-disjoint paths. Such a decomposition is minimum if it contains the minimum number of paths. The linear algorithm in Theorem 6 is based on a minimum path decomposition of a graph associated with the string set S. Besides the algorithm for the decision problem mentioned in Theorem 6, there is also a linear algorithm that finds a shortest superstring for strings of length at most 2.

A fixed maximum orbit size for the letters in the alphabet leads to a special case of the SSP that is also solvable in polynomial time. Assume a set S of strings over alphabet Σ and let $m = \max\{|s| : s \in S\}$.

Theorem 7 ([61]). *If the orbit size of each letter in Σ is at most 2 in S, then a shortest superstring for S is found in polynomial time $O(|\Sigma|^2 m)$.*

Another special case of the SSP concerns the fixed difference between the sum of string lengths and the cardinality of the alphabet as cited in [61]. Given a set S of strings over an alphabet Σ, for a fixed difference $||S|| - |\Sigma|$, the SSP is solvable in polynomial time by a special exhaustive enumeration. The difference $||S|| - |\Sigma|$ is mentioned as a measure of dissimilarity of the strings in S.

In [38], restricted cases of the SSP are studied, and a string form that induces polynomial cases is found.

Theorem 8 ([38]). *The SSP over the alphabet $\{0, 1\}$ is polynomial time solvable if each given string contains at most one 1.*

As cited in [61], a particular case of the SSP in which S is the set of *all* three-letter strings over an alphabet Σ is known as the **Code Lock Problem**. In this case, the possible overlaps between the strings are 1 and 2. This problem is reducible to the **Eulerian Walk Problem**, where the existence of a walk that contains all the arcs of a directed graph is sought, and hence, according to [16] it is solvable in polynomial time.

5 Exact Solutions

There are only few exact algorithms in the literature for the SSP. This is due to the computational complexity of the problem, and the lack of necessity for optimal solutions at its main applications in computational molecular biology. In the DNA sequencing practice, the biological properties of a genome molecule can be usually expressed also by a superstring of its fragments that is not the shortest one, but its length is close to the optimum.

5.1 Exhaustive Enumeration

The SSP can be trivially solved by exhaustive enumeration of all possible arrangements of the strings. The merge of the strings in some of these orders would correspond to a shortest superstring. Given a set S of n string, the examination of the superstrings of S that correspond to all permutations in Π_n is enough to find a shortest one. The exhaustive examination of all permutations can be executed in time $O(n!||S||)$, or by a different implementation that also exhaustively enumerates the possible solutions, in time $O(n||S||^{n+1})$ as mentioned in [61]. Optimal solutions for small SSP instances taken by the exhaustive algorithm are used in [24, 25] to compare the compression achieved by the viruses to their genome with the largest possible compression of their genes.

5.2 Integer Programming Formulation

Given an SSP instance specified by a set S of n strings, consider the associated overlap graph $G_o(S)$. An optimal solution to the SSP instance can be obtained by an optimal solution to the Max-HPP on $G_o(S)$, since a maximum Hamiltonian path contains all the vertices (strings) ordered in a single path such that it has the maximum total overlap. Due to the relation between the HPP and the TSP described in Sect. 2, these solutions can be obtained by an optimal solution to the Max-TSP. According to these transformations, optimal solutions for the SSP can be derived by any integer programming formulation for the Max-TSP using branch and bound or cutting plane algorithms. In [17], a benchmark set of instances with known optimal solutions was constructed using the integer program of [40] for the Max-TSP and used to compare the solutions of a heuristic for the SSP with the optimal ones.

6 Approximation Algorithms

The fact that the SSP is Max-SNP motivates many researches to develop approximation algorithms for it. As mentioned in Sect. 2, there are two ways to assess the solution of an approximation algorithm: the length measure considering the SSP as a minimization problem, and the compression measure considering the SSP as a maximization problem.

For a string set S, and any algorithm ALG for the SSP, we use the notation $\text{ALG}_l(S)$ to denote the length of the superstring of S obtained by ALG, and $\text{ALG}_c(S)$ to denote the corresponding achieved compression. An approximation ratio $\varepsilon = \frac{\text{ALG}_l(S)}{\text{opt}_l(S)} \geq 1$ with respect to the length measure means that $\text{ALG}_l(S) \leq \varepsilon \times \text{opt}_l(S)$ for all instances, while an approximation ratio $\varepsilon = \frac{\text{ALG}_c(S)}{\text{opt}_c(S)} \leq 1$ with respect to the compression measure means that $\text{ALG}_c(S) \geq \varepsilon \times \text{opt}_c(S)$ for all instances. Although the two measures

are equivalent regarding the optimal solution, they differ regarding the approximate solutions of the problem. The existence of an algorithm with a constant approximation ratio for the one measure has in general no approximation performance guarantee for the other measure.

In this section, the approximation algorithms for the SSP both with respect to the length and the compression measure are presented, revealing the special features of the superstrings in each case.

6.1 Approximation of Compression

The compression measure counts the number of letters gained in comparison with the simply concatenation of all strings. Algorithms that approximate this gain are presented here.

6.1.1 The Natural Greedy Algorithm

A very well known, simply implemented, and widely used algorithm for the SSP is the natural greedy algorithm. It is routinely used in DNA sequencing practice. It starts with the string set S and repeatedly merges a pair of distinct strings with the maximum possible overlap until only one string remains in S. Next algorithm shows the pseudo-code of the natural greedy for the SSP.

Algorithm: GREEDY

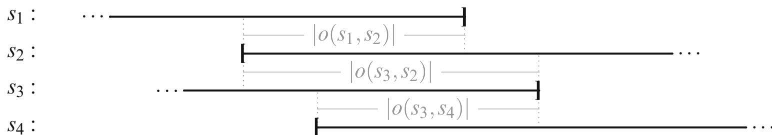
input : string set $S = \{s_1, s_2, \ldots, s_n\}$
output: a superstring of S

1. **for** $i = 1$ **to** $n-1$ **do**
2. $\quad L = \{(s_i, s_j) : s_i, s_j \in S, i \neq j\}$
3. $\quad k = \max\{|o(s_i, s_j)| : (s_i, s_j) \in L\}$
4. $\quad$ let $(s'_i, s'_j) \in L$ be a pairs such that $|o(s'_i, s'_j)| = k$
5. $\quad S = (S - \{s'_i, s'_j\}) \cup \{m(s'_i, s'_j)\}$
6. **end**
7. let s be the only string in S
8. **return** s

The operation of the GREEDY algorithm on the string set S is equivalent to the creation of a Hamiltonian path on the overlap graph $G_o(S)$. In general directed weighted graphs, the total weight of the Hamiltonian path obtained by the greedy approach is at least one third the weight of a maximum path [26]. In the case of the overlap graphs, a stronger result can be obtained by exploiting their properties. A basic lemma that concerns the form of these graphs is restated here in terms of strings.

Lemma 1 ([58]). *Let s_1, s_2, s_3, and s_4 be strings, not necessarily distinct, such that $|o(s_3,s_2)| \geq |o(s_1,s_2)|$ and $|o(s_3,s_2)| \geq |o(s_3,s_4)|$. Then $|o(s_1,s_4)| \geq |o(s_1,s_2)| + |o(s_3,s_4)| - |o(s_3,s_2)|$.*

The proof can be derived directly from the next figure, where the alignment of the four strings according to their overlaps is presented.



Notice that, if s_1 and s_4 are not distinct, then the result of Lemma 1 concerns the self-overlap of the string s_1.

The following theorem establishes the approximation performance of the GREEDY algorithm based on the corresponding analysis of the greedy approach for the Max-HPP and on Lemma 1.

Theorem 9 ([58]). *For a string set, the compression achieved by the* GREEDY *algorithm is at least half the compression achieved by a shortest superstring.*

Next example, presented in [58], shows that the result of the Theorem 9 is the best possible.

Example 6. For the string set $\{ab^k, b^{k+1}, b^k a\}$, $k \geq 1$, over the alphabet $\{a, b\}$, GREEDY may produce the superstring $ab^k ab^{k+1}$ or the superstring $b^{k+1}ab^k a$ that achieves compression k, whereas the shortest superstring is $ab^{k+1}a$ and achieves compression $2k$. Notice that GREEDY can also give the shortest superstring depending on how it breaks ties. □

6.1.2 Approximation Based on Matchings

Apart from GREEDY, two other $\frac{1}{2}$-approximation algorithms for the compression of a superstring based on the MP and the DMP are presented in [62]. For an SSP instance S, consider the associated overlap graph $G_o(S)$. In both algorithms, a matching algorithm is repeatedly applied to $G_o(S)$, to produce a Hamiltonian path.

For the description of the first algorithm the notion of the **arc contraction** is necessary. Given a weighted directed graph G and an arc $e = (u, v) \in E(G)$, the contraction of e is denoted by G/e and gives a new graph obtained from G where the vertices u, v and their incident arcs are replaced by a new vertex w which has as incoming arcs the incoming arcs of u and as outgoing arcs the outgoing arcs of v with the same weights as on G. The MATCH algorithm initially finds a maximum matching on $G_o(S)$, and then contracts the arcs of the matching. This process is repeated on the new graph until a graph with no arcs comes up. $G_o(S) = (V, E, w)$ is the initial overlap graph which remains unchanged, whereas G denotes the graph obtained in each iteration after the arc contractions. Initially, $G = G_o(S)$. Let $\text{maxm}(G)$ be a maximum matching on graph G.

Algorithm: MATCH

input : string set $S = \{s_1, s_2, \ldots, s_n\}$
output: a superstring of S

1. construct the graph $G_o(S)$
2. $P = \emptyset$
3. $G = G_o(S)$
4. **while** $|E(G)| \neq \emptyset$ **do**
5. $\quad M = maxm(G)$
6. $\quad P = P \cup \{$the arcs of $E(G_o(S))$ that correspond to $M\}$
7. $\quad$ **foreach** $(u, v) \in M$ **do**
8. $\quad\quad G = G/(u, v)$
9. $\quad$ **end**
10. **end**
11. let s be the superstring that corresponds to P
12. **return** s

The approximation performance of the MATCH algorithm is based on the observation that any matching on an overlap graph can be extended to a Hamiltonian path on it, since overlap graphs are complete. Moreover, a maximum matching has total weight at least half the weight of a maximum Hamiltonian path. This can easily be shown by considering each Hamiltonian path as two matchings with distinct arcs, constructed by taking alternate arcs from the path. These results imply that the compression achieved by the MATCH algorithm is at least half the optimal compression.

The second algorithm with the same approximation ratio for the SSP is based on the slightly different DMP. Remember that a directed matching on a graph is a set of disjoint paths and cycles. For the description of this algorithm, the notion of the arc contraction is extended naturally to paths. Given a weighted directed graph G and a path $p = (v_1, v_2, \ldots, v_r)$ on it, defined by its vertices, the contraction of p gives a new graph obtained from G where the vertices $v_1, \ldots, v_r$ and their incident arcs are replaced by a new vertex w which has as incoming arcs, the incoming arcs of v_1 and as outgoing arcs the outgoing arcs, of v_r with the same weights as on G. The DIMATCH algorithm described also in [62] operates exactly as MATCH except that it finds a *directed* matching of each step, opens each cycle of it by deleting an arc with the smallest weight, and finally contracts the paths into vertices. The compression achieved by the DIMATCH algorithm is at least half the optimal compression.

6.1.3 Approximation Based on the TSP

Any approximation algorithm for the Max-TSP is also an approximation algorithm for the SSP with respect to the compression measure, or equivalently for the Max-HPP, with the same ratio due to the transformation from the TSP to the HPP. For both problems, it is implied that they are **asymmetric**, which means that they applied on directed graphs, and that the weight of an arc (u, v) is not necessarily equal to the weight of the arc (v, u).

In [7, 31], two approximation algorithms for the Max-TSP are presented. In both cases, procedures with complementary worst cases run on directed graphs with even number of vertices. The best result among them is a Hamiltonian cycle whose weight is at least $\frac{38}{63}$ times the weight of a maximum weight Hamiltonian cycle for the first algorithm, and $\frac{8}{13}$ for the second. Both algorithms achieve their approximation performance without utilizing any special structure of the strings. In both algorithms is required that the complete input graph G has an even number of vertices. In general, for an SSP instance of n strings, the above algorithms achieve approximation ratios $\frac{38}{63}(1-\frac{1}{n})$ and $\frac{8}{13}(1-\frac{1}{n})$, respectively. Finally, an approximation algorithm for the Max-TSP is also designed in [29], achieving the best ratio until now, namely $\frac{2}{3}$. It operates by decomposing a special form of directed multigraphs, where the elements of the arc set are ordered triples of the vertex set.

6.2 Approximation of Length

A plethora of approximation algorithms with respect to the length measure have been developed for the SSP using different variations of the greedy strategy. The best one among them finds a string whose length is at most $2\frac{1}{2}$ times the length of the optimal string.

6.2.1 Naive Approximation Algorithm

A naive algorithm for the SSP is used in [8] for comparison reasons in relative performance of other approximation algorithms. Its approximation performance is not remarkable but the idea is quite simple, showing that it is easy to develop an algorithm for the SSP, but it is not so easy to achieve a good approximation ratio. For a string set S, the algorithm arbitrarily chooses a string from S considering it as the initial current string, and then repeatedly updates the current string by merging it with a remaining string from S that yields the maximum overlap. The performance of this algorithm highly depends on the random choice of the initial point, and it is possible to produce superstrings whose length grows quadratically in the optimal length.

6.2.2 Approximation Algorithm Used in a Learning Process

The first attempt to approximate the shortest superstring of a set was made in [34], where the DNA sequencing procedure is modelled as a string learning process from randomly drawn substrings of it. Under certain restrictions, this may be viewed as a string learning process in Valiant's distribution free learning model [63]. The efficiency of the learning method depends on the solution of an algorithm which approximates the length of a superstring, and seeks in each step for an appropriate join string among the candidate ones.

Given a string set $S = \{s_1, s_2, \ldots, s_n\}$ and a string s, we denote by $subSs(S,s)$ the set of the strings in S that are substrings of s.

Example 7. Suppose that we have the string set $S = \{s_1, s_2, s_3\}$, where $s_1 = caabaa$, $s_2 = abaaca$, and $s_3 = baacaa$ are strings over the alphabet $\{a,b,c\}$. The set of all join strings of s_1 and s_2 regardless their order is $J_{\{s_1,s_2\}} = \{caabaaabaaca, caabaabaaca, caabaaca, abaacacaabaa, abaacaabaa\}$, while the only join string $s \in J_{\{s_1,s_2\}}$ for which $subSs(S,s) = S$ is the string *abaacaabaa*. □

Given a string set S, the GROUP-COMBINE algorithm constructs a superstring of S by an iterative process. The algorithm begins with a string set and combines the strings in groups such that all strings in a group are substrings of a join string of two of them, trying to find as large groups as possible.

Algorithm: GROUP-COMBINE
input : string set $S = \{s_1, s_2, \ldots, s_n\}$
output: a superstring of S

1. $T = \emptyset$
2. **while** $|S| > 0$ **do**
3. find $s_i, s_j \in S$ such that $\min_{s \in J_{\{s_i,s_j\}}} \frac{|s|}{||subSs(S,s)||}$ is minimized
4. let $\bar{s}$ be the join string that achieves the minimum in step 3
5. $S = S - subSs(S, \bar{s})$
6. $T = T \cup \{\bar{s}\}$
7. **if** $|S| = 0$ ***and*** $|T| > 1$ **then**
8. $S = T$
9. **end**
10. **end**
11. let s be the only string in T
12. **return** s

Next theorem establishes the approximation ratio of the algorithm.

Theorem 10 ([34]). *Given a string set, if the length of the optimal superstring is m, then* GROUP-COMBINE *produces a superstring of length* $O(m \log m)$*.*

6.2.3 4-Approximation Algorithms

The first approximation algorithm with a constant ratio for the length of a superstring is described in [8], answering a notorious open problem for the existence of such an algorithm. The algorithm utilizes a minimum cycle cover on the distance graph of a string set to derive a superstring with preferable properties that bound its length. Given an SSP instance S, the CYCLE-CONCATENATION algorithm finds a minimum cycle cover on the graph $G_d(S)$ *with loops* in polynomial time. Then it

opens each cycle of the cover by removing an arc chosen randomly, constructs the superstring that corresponds to the obtained path, and concatenates these strings.

Algorithm: CYCLE-CONCATENATION
input : string set $S = \{s_1, s_2, \dots, s_n\}$
output: a superstring of S

1. construct the graph $G_d(S)$ with loops
2. find a minimum cycle cover $C = \{c_1, c_2, \dots, c_p\}$ on $G_d(S)$
3. **foreach** $c_i \in C$ **do**
4. choose a vertex $s_i \in c_i$ randomly
5. $s'_i = strC(c_i, s_i)$
6. **end**
7. let s be the concatenation of the strings s'_i
8. **return** s

Next theorem demonstrates the approximation performance of the algorithm establishing the first constant approximation ratio for the SSP.

Theorem 11 ([8]). *For a string set S,* CYCLE-CONCATENATION *produces a superstring of length at most* $4 \times opt_l(S)$.

Another algorithm for the SSP with the same constant approximation ratio is MGREEDY which is presented in [8].

Algorithm: MGREEDY
input : string set $S = \{s_1, s_2, \dots, s_n\}$
output: a superstring of S

1. $T = \emptyset$
2. **while** $|S| > 0$ **do**
3. $k = \max\{|o(s'_i, s'_j)| : s'_i, s'_j \in S\}$
4. let (s_i, s_j) be a string pair such that $|o(s_i, s_j)| = k$
5. **if** $i \neq j$ **then**
6. $S = (S - \{s_i, s_j\}) \cup \{m(s_i, s_j)\}$
7. **else**
8. $S = S - \{s_i\}$
9. $T = T \cup \{s_i\}$
10. **end**
11. **end**
12. let s be the concatenation of the strings in T
13. **return** s

Notice that at line 3, the two strings of each pair are not necessarily distinct allowing in this way the self-overlaps. Since the choices at line 4 are made according to the overlaps in S, MGREEDY can be thought as choosing arcs from the graph $G_o(S)$ *with loops*. The choice of the pair (s_i, s_j) corresponds to the choice of the arc $(\text{last}(s_i), \text{first}(s_j))$ on $G_o(S)$ in each step. Therefore, the algorithm constructs paths, and closes them into cycles when distinctness is not satisfied at line 4. Thus,

MGREEDY ends up with a set of disjoint cycles that cover the vertices of $G_o(S)$, which is a cycle cover. The same cycle cover can be thought on graph $G_d(S)$ *with loops*. For a cycle cover C, by Eq. (2), we have $o(C) + d(C) = ||S||$, and so a cycle cover has minimum total weight on $G_d(S)$ if and only if it has maximum total weight on $G_o(S)$, and in both cases it is called optimal.

Theorem 12 ([8]). *The cover created by* MGREEDY *is an optimal cycle cover.*

Notice that the presence of the loops is a necessary assumption for this result. Since MGREEDY finds an optimal cycle cover, the superstring that is produced by it is no longer than the string produced by algorithm CYCLE-CONCATENATION. Therefore, the approximation ratio with respect to the length measure of MGREEDY for the SSP is also equal to 4. Actually, the superstring of MGREEDY could be shorter than the one obtained by CYCLE-CONCATENATION since MGREEDY simulates the breaking of each cycle in the optimal position, that is between the strings with the minimum overlap in the cycle.

6.2.4 GREEDY Is a $3\frac{1}{2}$-Approximation Algorithm

The GREEDY algorithm has already been presented as an approximation for the compression. A notorious open question is how well GREEDY approximates the length of a shortest superstring, while a common conjecture states that GREEDY produces a superstring of length at most two times the length of the optimum [54, 58, 62]. In fact, GREEDY may give a superstring almost twice as long as the optimal one, as shown in the next example from [8].

Example 8. For the string set $\{c(ab)^k, (ba)^k, (ab)^k c\}$, $k \geq 1$, over the alphabet $\{a, b, c\}$, GREEDY may produce the superstring $c(ab)^k c(ba)^k$ or the superstring $(ba)^k c(ab)^k c$ of length $4k+2$, whereas the shortest superstring is $c(ab)^{k+1}c$ of length $2k+4$. □

In [8], it is proved that GREEDY is a 4-approximation algorithm for the SSP. Next theorem improves this approximation ratio based on a more careful analysis on specially formed strings.

Theorem 13 ([28]). *The* GREEDY *algorithm is a* $3\frac{1}{2}$*-approximation algorithm with respect to the length measure.*

6.2.5 A 3-Approximation Algorithm

The algorithm TGREEDY described in [8] operates in the same way as MGREEDY except that in the last step it merges the strings in set T by running GREEDY on them instead of simply concatenates them. Next theorem establishes its approximation performance.

Theorem 14 ([8]). *For a string set S, algorithm* TGREEDY *produces a superstring of length at most* $3 \times opt_l(S)$.

In [8], a relative performance comparison between GREEDY, MGREEDY, and TGREEDY algorithms is presented. TGREEDY always produces better solutions than MGREEDY since in the last step it greedily merges the strings, whereas MGREEDY just concatenates them. The approximation performance of TGREEDY is better than this of GREEDY, but the superiority of one of these algorithms over the other is not guaranteed as shown in the next example.

Example 9. For the string set $\{c(ab)^k, (ab)^{k+1}a, (ba)^k c\}$, $k \geq 1$, over the alphabet $\{a,b,c\}$, GREEDY produces the shortest superstring $c(ab)^{k+1}ac$ of length $2k+5$, whereas TGREEDY produces the superstring $c(ab)^k ac(ab)^{k+1}a$ or the superstring $(ab)^{k+1}ac(ab)^k ac$ of length $4k+6$, since the initial maximum overlap is the self-overlap of the second string.

On the other hand, for the string set $\{cab^k, ab^k ab^k a, b^k dab^{k-1}\}$, $k \geq 1$, over the alphabet $\{a,b,c,d\}$, TGREEDY produces the shortest superstring $cab^k dab^k ab^k a$ of length $3k+6$, since the initial maximum overlap is the self-overlap of the second string, whereas GREEDY produces the superstring $cab^k ab^k ab^k dab^{k-1}$ or the superstring $b^k dab^{k-1} cab^k ab^k a$ of length $4k+5$. □

6.2.6 Generic Approximation Based on Cycle Covers

Algorithms MGREEDY and TGREEDY implicitly construct optimal cycle covers on the associated overlap and distance graphs of a string set, while CYCLE-CONCATENATION explicitly takes advantage of this construction. A generic algorithm that explains this basic idea is presented in [11].

For a string set S, let $C = \{c_1, c_2, \ldots, c_p\}$ be a cycle cover on the graph $G_d(S)$. Suppose that an arbitrary string r_i is picked from each cycle $c_i \in C$, and these strings form the representative set $R = \{r_1, r_2, \ldots, r_p\}$. Let $r = \langle r_1, r_2, \ldots, r_p \rangle$ be a superstring of R. By replacing each r_i, $i \in \mathrm{I}_p$, in r with the string $strC^+(c_i, r_i)$, we get the string

$$\langle strC^+(c_1, r_1), strC^+(c_2, r_2), \ldots, strC^+(c_p, r_p) \rangle,$$

which is called the **extension string of** r **with respect to** C and is denoted by $\mathrm{ext}(r,C)$. Observe that $\mathrm{ext}(r,C)$ is a superstring of S.

For a string set S, the GENERIC-COVER algorithm constructs a minimum cycle cover C on the graph $G_d(S)$, and chooses a random string from each cycle of this cover to form a set R of representatives. Then, it finds a new minimum cycle cover on $G_d(R)$, opens each cycle of this cover in a random position, and concatenates the resulting cycle superstrings to create a superstring of R. Finally, it returns the extension string of this superstring with respect to the cycle cover C to take a superstring of S.

```
    Algorithm: GENERIC-COVER
    input  : string set S = {s_1, s_2, ..., s_n}
    output: a superstring of S
 1. construct the graph G_d(S)
 2. find a minimum cycle cover C on G_d(S)
 3. R = ∅
 4. foreach c_i ∈ C do
 5.     choose a string s_i of c_i randomly
 6.     R = R ∪ {s_i}
 7. end
 8. create the graph G_d(R)
 9. find a minimum cycle cover C_R on G_d(R)
10. foreach cycle c_i ∈ C_R do
11.     let s_i be the head of a randomly chosen arc of c_i
12.     s'_i = strC(c_i, s_i)
13. end
14. let r be the concatenation of the strings s'_i
15. r̄ = ext(r, C)
16. return r̄
```

The GENERIC-COVER algorithm has approximation ratio equal to 3. This algorithm constitutes the base for the design of better approximation algorithms for the SSP as described below.

6.2.7 Handling 2-Cycles and 3-Cycles Separately

For an SSP instance specified by a string set S, $\mathrm{opt}_c(S)$ may grow quadratically in $\mathrm{opt}_l(S)$ in general. Thus, to take advantage of a *compression* approximation to design *length* approximation algorithms with constant ratio based on GENERIC-COVER framework, a key is to construct suitable subproblems for which $\mathrm{opt}_c(S)$ is linear in $\mathrm{opt}_l(S)$. The main difficulty in determining such subproblems and so in improving the length approximation performance of the GENERIC-COVER algorithm appears in handling k-cycles with small k in the cycle cover C_R. In [60], the compression achieved by GREEDY is utilized, to design a length approximation algorithm for the SSP. The algorithm is based on the scheme of GENERIC-COVER handling separately the 2-cycles in the minimum cycle cover C_R. In this way, the algorithm achieves an approximation ratio $2\frac{8}{9}$. In [11], an approximation algorithm that handles separately the 2-cycles and the 3-cycles is developed and gives a superstring of length at most $2\frac{5}{6}$ times the length of a shortest superstring.

6.2.8 Approximation Algorithms Based on the TSP

As cited in [31], a relationship between the SSP and the Max-TSP according to their approximation is given by the following lemma.

Lemma 2 ([8]). *If the Max-TSP has a* $(\frac{1}{2}+\varepsilon)$*-approximation, then the SSP has a* $(3-2\varepsilon)$*-approximation with respect to the length measure.*

Utilizing this relation, the approximation algorithms for the Max-TSP mentioned in Sect. 6.1.3 can be used to derive approximation ratios for the SSP with respect to the length measure. Consider an SSP instance S of n strings and the corresponding Max-TSP instance $G_o(S)$. For even number of strings, the algorithm described in [31] and achieves an approximation ratio of $\frac{38}{63}$ for the Max-TSP, gives a $2\frac{50}{63}$-approximation ratio for the SSP, while the algorithm described in [7] and achieves an approximation ratio of $\frac{8}{13}$ for the Max-TSP, gives a $2\frac{10}{13}$-approximation ratio for the SSP. For odd number of strings the algorithms achieves a ratio of $2(\frac{50}{63}+\frac{1}{n})$ and $2(\frac{10}{13}+\frac{1}{n})$ for the SSP, respectively. The algorithm described in [29] and achieves an approximation ratio of $\frac{2}{3}$ for the Max-TSP, gives a $2\frac{2}{3}$-approximation ratio for the SSP.

6.2.9 Exploiting the Superstring Structures

The approximation algorithms presented above are largely graph-theoretical, meaning that they sufficiently exploit the structure of overlap and distance graphs, but they do not take advantage of the structure inside the strings or in general of the properties not evident in graph representation. In this sense, they solve a more general problem than the one at hand.

An algorithm that captures a great deal of the structure of the SSP instances is presented in [3]. It takes advantage of the structure of strings with large value of overlap, proving several key properties of such strings. It follows the framework of the GENERIC-COVER algorithm using a more sophisticated way to choose the representatives at line 5 and to open each cycle at line 12. After finding a cycle cover on the associated distance graph, the key is to exploit the periodic structure of the cycle superstrings that arise. In this way, the algorithm achieves a bound either to the total overlap of the rejected arcs at line 12 or to the total additional length of extending each cycle at line 15. The result is to construct a superstring whose length is no more than $2\frac{3}{4}$ times the length of an optimal superstring.

This algorithm and the $2\frac{50}{63}$-approximation algorithm for the Max-TSP that is mentioned in Sect. 6.2.8 have complementary worst cases, and so a better ratio can be achieved by their combination. When the worst case of the first algorithm occurs, the Max-TSP algorithm runs as a subroutine on the set of representatives to take a better result. Balancing the two algorithms, an approximation ratio of $2\frac{50}{69}$ for the SSP can be achieved [2].

In [4], the study of the key properties is extended to strings that exhibit a more relaxed form of the periodic structure considered before. Algorithmically,

the new approach is also based on the framework of the GENERIC-COVER and is a generalization of the previous one. On the other hand, the analysis is very different and includes a special structure of 2-cycles. Let c be a 2-cycle in the cycle cover C_R of the GENERIC-COVER algorithm, consisting of the vertices s_i and s_j, which are the representatives of the cycles c_i and c_j in the cycle cover C. Without loss of generality assume that $d(c_i) \geq d(c_j)$. The cycle c is a g**-HO2-cycle** if

$$\min\{|o(s_i,s_j)|,|o(s_j,s_i)|\} \geq g(d(c_i)+d(c_j)).$$

In the new algorithm, during the selection of the representatives a technique is used to anticipate the potential of each string to participate in a $\frac{2}{3}$-HO2-cycle. Such strings have a very specific structure, and if there is a string without such a structure in a cycle, it is chosen as the representative. Otherwise, the knowledge of the structure of the entire cycle can be used to trade the amount of the lost overlap against the additional length of extending the representative to include the rest of the cycle. In this way, a $2\frac{2}{3}$-approximation algorithm for the SSP is designed.

6.2.10 Rotations of Periodic Strings

Two approximation algorithms for the SSP that are based also on the inner structure of the strings and their periodic properties are presented in [9]. They use the same framework of the GENERIC-COVER algorithm, but they make use of new bounds on the overlap between two strings.

Both algorithms pay special attention to the selection of the representatives but without concentrating on k-cycles with small k. Instead of choosing a string obtained by opening each cycle, the new idea is to look for superstrings of the strings in a cycle that are *not* too long and are guaranteed *not* to overlap with each other by too much. Each chosen superstring does not even have to be one of the cycle superstrings obtained by opening the cycle. Given a cycle $c_i = (s_1, s_2, \ldots, s_p, s_1)$ of the cycle cover C of algorithm GENERIC-COVER, a string r_c is a candidate to be a representative of c if for some j

- r_c is a superstring of $strC(s_{j+1})$ and
- r_c is a substring of $strC^+(s_j)$.

A sophisticated procedure is used to choose the representatives such that they satisfy these two conditions and also have an appropriate property to lead to the improved ratio.

After this step, the two approximation algorithms follow different ways. The first algorithm after finding the second cycle cover opens each cycle and concatenates the cycle superstrings, achieving an approximation ratio of $2\frac{2}{3}$. The second algorithm constructs a superstring of the representatives using as subroutine an approximation algorithm with respect to the compression measure for the SSP. As subroutines, we can use the approximation algorithms cited in Sect. 6.1.3. Using

the $\frac{38}{63}$-approximation algorithm described in [31] a length ratio of $2\frac{25}{42}$ is achieved, while using the $\frac{8}{13}$-approximation algorithm described in [7] a length ratio of $2\frac{15}{26}$ is achieved.

6.2.11 $2\frac{1}{2}$-Approximation Algorithms

The best approximation ratio with respect to the length measure for the SSP is the $2\frac{1}{2}$ until now. It can be achieved by two different methods, one from the field of the superstrings and the other from the field of the TSP.

The first algorithm is described in [57]. Given a string set S, the algorithm begins by constructing a minimum cycle cover C on graph $G_d(S)$. Then, instead of choosing representatives, it combines the cycles of C to produce a new cycle cover C', and finally opens each cycle in C' to produce a set of cycle superstrings. The concatenation of these superstrings yields a superstring of S. The algorithm exploits the properties of cycles and cycle covers on a special multigraph to achieve the $2\frac{1}{2}$-approximation ratio.

The second approach that achieves the same length ratio for the SSP is an approximation algorithm for the Max-TSP described in [29]. It finds a Hamiltonian cycle whose weight is at least $\frac{2}{3}$ the weight of a maximum Hamiltonian cycle. Using this procedure as a subroutine in the algorithm cited in Sect. 6.2.10, a length ratio of $2\frac{1}{2}$ for the SSP can be achieved.

7 Parallelizing the Solving Process

In complexity theory, the class NC consists of the decision problems (languages) decidable in polylogarithmic parallel time $O(\log^{O(1)} n)$ on a parallel computer with polynomial number $O(n^{O(1)})$ processors. In this definition, a parallel random access machine (PRAM) is assumed, that is a parallel computer with a central pool of memory, where any processor can access any bit of memory in constant time. The class RNC, which stands for random NC, extends NC with access to randomness. The class RNC consists of the decision problems (languages) that have a randomized algorithm which is solvable in polylogarithmic parallel time on polynomially many processors, and its probability of producing a correct solution is at least $\frac{1}{2}$.

It is conjectured that there are some tractable problems which are inherently sequential and cannot significantly be sped up by using parallelism. For an algorithm, a common method to show that it is hardly parallelizable is to prove that the algorithm is P-complete for the problem it applied to. The GREEDY algorithm belongs to this case since the problem of finding a superstring chosen by the GREEDY algorithm is P-complete [11]. This means that GREEDY is difficult to be parallelized effectively. In the following, parallel approximation algorithms for the SSP are presented.

7.1 NC Algorithm with Logarithmic Length Ratio

Given a ground set X of elements and a family Y of subsets of X, a **set cover of** X **with respect to** Y is a subfamily $Y' \subseteq Y$ of sets whose union equals to X. Assigning a weight $w(x)$ to each element $x \in X$ the total weight of each set and family is naturally defined. The **Set Cover Problem** (SCP) is to find a set cover of the ground set of the minimum weight. The SCP can be approximated within a logarithmic ratio by a parallelizable algorithm [6].

In [11], a similar approach to the one presented in Sect. 6.2.2 for grouping is applied to the SCP. Given a set S of n strings, we define

$$F = \{subSs(S,s) : s \in J_{\{s_i,s_j\}}, s_i, s_j \in S\},$$

that is the family of the sets of substrings of all possible pairwise join strings from S. Considering S as the ground set and F as a family of its subsets, they specify an instance of the SCP. From each set cover $C \subseteq F$ of S, a string s_C can be constructed by merging the join strings that correspond to the sets of C. Observe that s_C is a superstring of S. Let the weight of each set of F be the length of the corresponding join string, and $w(C)$ be the total weight of the set cover C. Because of the merging of the join strings, $|s_C| \leq w(C)$. Also, it is proved that the length of the superstring corresponds to a minimum set cover C^* is at most twice the length of an optimal superstring, that is $|s_{C^*}| \leq 2 \times \mathrm{opt}_l(S)$. These results combined with the parallelization of the SCP imply an NC algorithm with logarithmic approximation for the SSP.

Theorem 15 ([11]). *For a string set S of n strings, there is an NC algorithm that for any $\varepsilon > 0$, finds a superstring whose length is at most $(2+\varepsilon)\log n$ times the length of a shortest superstring.*

Observe that each group of strings selected by the GROUP-COMBINE algorithm is a set of the family F as it was described previously, and so this algorithm constructs implicitly a set cover of S with respect to F. Theorem 15 proves that this result can also be obtained by a parallelizable procedure letting as open problem the design of an NC algorithm with a constant approximation ratio with respect to the length measure for the SSP.

7.2 RNC Algorithm with Constant Length Ratio

An RNC algorithm for the SSP is based on a parallelizable implementation of the sequential $2\frac{5}{6}$-approximation algorithm mentioned in Sect. 6.2.7. The only non-trivially parallelizable steps of this algorithm are the computations of the minimum cycle covers. Remember that, the problem of finding an optimal cycle cover is equivalent to the problem of finding a maximum matching on a bipartite graph. In general, it is not known if it can be done in either NC or in RNC. However, when the weights of the graph are given in unary notation, a condition that can be satisfied in the case

of this algorithm, a maximum matching can be found in RNC (see e.g. [49]), giving the next theorem for the SSP.

Theorem 16 ([11]). *For a string set S, there is an RNC algorithm that finds a superstring of length at most* $2\frac{5}{6} \times opt_l(S)$.

7.3 NC Algorithm with Compression Ratio $\frac{1}{4+\varepsilon}$

Given a weighted directed graph, a natural greedy approach for finding a maximum cycle cover is described as follows. Scan the arcs in non-increasing order of weights, and select an arc that does not have the same head or the same tail with a previously selected arc. Repeat until the selected arcs form a cycle cover. This approach finds a cycle cover of weight at least half the weight of a maximum cycle cover [11]. As mentioned in Sect. 6.2.3, if the graph is an overlap graph *with* loops then this greedy approach always finds a maximum weight cycle cover.

For the development of an NC compression approximation algorithm with a constant ratio for the SSP, a slightly different algorithm from the natural greedy for the CCP is designed. This algorithm achieves a worse approximation ratio, but can be parallelized. It is based on the idea that the natural greedy algorithm could be only a bit worse if in each step it chooses instead of the maximum weight arc, one with a similar weight. The arcs of the graph are partitioned into levels, such that the weights of all arcs in a level are within a constant factor. Given a graph G and a real $c > 1$, an arc $e \in E(G)$ has c-level equal to k if $c^{k-1} < w(e) \leq c^k$, and c-level equal to 0 if $w(e) \leq 1$. The algorithm operates like the natural greedy algorithm assuming that all arcs in each level have the same weight. The usage of this algorithm on overlap graphs for finding superstrings concludes to the next theorem.

Theorem 17 ([11]). *For a set S of n strings, there is an NC algorithm for the SSP that achieves a compression ratio* $\frac{1}{4+\varepsilon}$. *It runs either in time* $O(\log^2 n \log_{1+\varepsilon} ||S||)$ *on a PRAM with* $||S|| + n^4$ *processors or in time* $O(\log^3 n \log_{1+\varepsilon} ||S||)$ *on a PRAM using* $n^2 + ||S||$ *processors.*

8 Inapproximability Bounds

Both minimization and maximization versions of the superstring problem are Max-SNP-hard, which means that there exists an $\varepsilon > 0$ such that it is NP-hard to approximate the SSP within a ratio of $1 + \varepsilon$ with respect to the length measure, or within a ratio of $1 - \varepsilon$ with respect to the compression measure. The practical side of this theoretical result is expressed by explicit bounds to the approximation ratio in both cases.

The first work to this direction appears in [41], where inapproximability bounds are given for a special case of the SSP. Specifically, the result concerns SSP instances

where the alphabet is $\{0,1\}$ and every string is of the form $10^m1^n01^m0^{n+4}10$ or $01^m0^n10^p1^q01^m0^n10^r1^s01$, where $m,n,p,q,r,s \geq 2$. This special case is used also in Theorem 5 that concerns APX-hardness results. Let us refer to this special case as SSP_2 for short. The next two theorems establish the inapproximability results.

Theorem 18 ([41]). *The SSP_2 is not approximable within $1\frac{1}{17245}$ with respect to the length measure, unless P = NP.*

Theorem 19 ([41]). *For every $\varepsilon > 0$, the SSP_2 is not approximable within $1\frac{1}{11216} - \varepsilon$ with respect to the compression measure, unless P = NP.*

In [64], inapproximability bounds for the SSP restricted to instances with equal length strings are given. Moreover, these bounds are extended to instances over alphabets of cardinality 2 improving the previous ones.

Theorem 20 ([64]). *For any $\varepsilon > 0$, unless P = NP, the SSP on instances with equal length strings is not approximable in polynomial time within ratio*

- $1\frac{1}{1216} - \varepsilon$ *with respect to the length measure, and*
- $\frac{1070}{1071} + \varepsilon$ *with respect to the compression measure.*

A very important result about the relation between the inapproximability of the SSP over an alphabet of cardinality 2 and over any alphabet is established in the next theorem. It implies that the alphabet cardinality does not affect the approximability of the SSP.

Theorem 21 ([64]). *Suppose that the SSP can be approximated by a ratio ε on instances over an alphabet of cardinality 2. Then the SSP can be approximated by a ratio ε on instances over any alphabet.*

This result holds for both measures, length and compression. Therefore, the bounds established in Theorem 20 hold also for alphabets of cardinality 2.

The computation of the inapproximability bounds for the SSP reveals the large gap between these and the best known approximation ratios for the problem both for the length measure and the compression measure.

9 Heuristics

The design of the approximation algorithms is oriented to the achievement of the approximation ratio and not to the best possible result. On the other hand, real-world applications usually need practically good results and not theoretically good ratios for the result. A heuristic algorithm can satisfy this requirement by giving solutions to SSP instances that have not approximation performance guarantee but are experimentally close to the optimum. The greedy strategies seem to perform much better than their proved approximation ratios both in average and in real-world cases. In this section, the heuristic algorithms for the SSP are described.

9.1 A Variant of the Natural Greedy

A problem with the GREEDY algorithm is that it makes choices that may forbid good overlaps from future selection. In an attempt to eliminate this behaviour, a heuristic that imitates GREEDY but chooses differently the string pair in each step is described in [58]. Here, the modification is given in terms of strings instead of arcs in the associated overlap graph as made in the original work. The selection criterion in each step is not just the overlap but the overall influence of the choice of each string pair. Given a string set S and two string s_i and s_j in it, let

$$\begin{aligned}\mathrm{oi}(s_i,s_j) = a|o(s_i,s_j)| \\ -\max\{|o(s_{i'},s_j)|, s_{i'} \in S, i' \neq i\} \\ -\max\{|o(s_i,s_{j'})|, s_{j'} \in S, j' \neq j\}.\end{aligned}$$

where a is a parameter that tunes the method. The idea is to take under consideration also the overlaps that would be eliminated if the pair (s_i,s_j) is selected. The pseudo-code of this heuristic algorithm is exactly as the one of GREEDY except that line 3 changes to $k = \max\{\mathrm{oi}(s'_i,s'_j) : (s'_i,s'_j) \in L\}$. In experiments cited in [58] with this heuristic algorithm, the best results were obtained with parameter a values from 2 to 2.5. In this case, the modified algorithm gives superstrings with average additional length from the optimum about $\frac{1}{5}$ the corresponding average additional length of GREEDY.

9.2 A Heuristic Parametrized by a Learning Process

A three-stage heuristic algorithm for the SSP, named ASSEMBLY, is presented in [20]. It is based on the observation that the set of the remaining strings in the GREEDY algorithm after a number of merges is very possible to contain only string pairs with small overlaps. The ASSEMBLY algorithm, in a try to avoid mistakes, terminates the greedy strategy when false merges are expected to occur, a decision based on the number of remaining strings.

The first stage of the algorithm is similar to the GREEDY algorithm except that it is terminated when the remaining string set has a cardinality c. The second stage of the ASSEMBLY algorithm is also based on greedy choices, although not made among all the possible overlaps, but only among these that pass a certification procedure. Given two strings s_i and s_j in the set of the remaining strings with $|o(s_i,s_j)| > 0$, a third string s_k is a **certificate** if its overlap with both s_i and s_j is greater than 0. It is experimentally determined that for two strings s_i and s_j with $|o(s_i,s_j)| > 0$, the existence of a certificate increases the probability their merge string participates to the shortest superstring. The second stage of the ASSEMBLY algorithm has as input the output string set of the first stage, and utilizes the idea of the certification to boost the greedy choices to string pairs that are also certified. It is terminated when the

cardinality of the remaining string set became equal to the parameter b. The third stage of the ASSEMBLY algorithm is a restricted backtracking procedure. Its input is the output string set of the second stage. It excludes some solutions based on a learning process, and then performs an exhaustive search through the rest solution space. In this way, it tries to balance between time efficiency and accuracy.

The ASSEMBLY algorithm is tested both on domains of random and real-world SSP instances. The first is taken by random string generators over specific distribution specifications, and the second is taken by DNA sequence databases. A number of instances of each domain are used as input to a learning procedure to specify the parameters b, c and the excluded solutions of the third stage, and the rest is used to test the ASSEMBLY algorithm. Every version of the ASSEMBLY algorithm was tested on the domain that was used for its training, but also to the other domain. The results show that ASSEMBLY performs significantly better when trained on the same domain it was tested, whereas the randomly trained version has poor performance on the real-world instances. The version of the algorithm that is trained by a successfully sequenced DNA molecule achieves a very high accuracy and effectiveness to instances of the same domain. This indicates that a successfully sequenced *part* of a DNA molecule can be used to significantly speed up the sequencing of the *whole* DNA molecule. The sequenced part can act as input to the learning procedure to determine the suitable parameter values, and the whole molecule can then be obtained by the ASSEMBLY algorithm with high accuracy and significantly sped up. The running time of the algorithm mainly depends on the running time of its third stage, which may be exponential. The tested instances suggest a sub-exponential growth of search space for this stage, but experiments on larger SSP instances are needed to conjecture a polynomial growth.

9.3 Genetic Algorithm

Some heuristic algorithms are inspired by evolutionary processes in nature. **Genetic algorithms** [22] belong to this class of heuristics. They are search methods that simulates the evolution process of natural selection, and used in many scientific fields to solve optimization problems. In a genetic algorithm for an optimization problem, a **population**, that is a collection of candidate solutions, called **individuals**, is evolved to reach better solutions. The evolution happens in **generations** that reflect the alternations to the population. During each generation the **fitness** of each individual in the population is evaluated proportionally to the suitability of its value for the objective function of the optimization problem. The most suitable individuals are selected to perpetuate their kind by recombining their genomes, i.e., their solutions, in specific points and by possibly randomly mutated. In this way, a new population is formed and the procedure is repeated for the next generation. Commonly, the algorithm terminates when either a maximum number of generations is produced, or a satisfactory fitness level is reached.

A genetic algorithm for the SSP is described in [66]. The input of the algorithm is a set S of strings specifying the SSP instance. The genome of each individual in the population is represented as a collection of strings from S in specific order, such that it is a candidate solution to the SSP instance. A crucial point of the algorithm is that an individual may not contain all the strings from S or may contain duplicate copies of the same string. This choice makes the output not a permutation of the strings in S giving in this way new potentials to the algorithm. The algorithm was tested to SSP instances over an alphabet of cardinality 2 using specific values for the parameters of the population size, the number of generations, and the mutation rates. The input instances were generated randomly following the DNA sequencing procedure. The experimental results show that when the number of the strings is 50 the genetic algorithm is better than GREEDY, while its dominance is lost when the number of the strings becomes 80.

9.4 Coevolutionary Algorithm

Coevolutionary algorithms also belong in the class of the biologically inspired evolutionary procedures. They generalize the idea of the genetic algorithms involving individuals from more than one species. **Coevolution** in nature refers to the simultaneous evolution of two or more species with coupled fitness. There are two different kinds of coevolution: the **competitive** one where the purpose is to obtain exclusivity on a limited resource, and the **cooperative** one where the purpose is to gain access to some hard to attain resource. In cooperative coevolutionary algorithms there is a number of independently evolving species representing components of potential solutions which together form complex structures to solve an optimization problem. Complete solutions are obtained by assembling representative members of each species. The fitness of each individual depends on the quality of the complete solutions it participates in. Therefore, the fitness function measures how well an individual cooperates with individuals from other species to solve the optimization problem.

A cooperative coevolutionary algorithm adjusted to the SSP is presented in [66]. It is based on populations of two species that evolve simultaneously. The first population contains prefixes of candidate solutions of the SSP instance, and the second population contains candidate suffixes. Each species population evolves separately and the only interaction between the two populations is through the fitness function. Computation experiments similar to those for the genetic algorithm show that this algorithm performs at least as good as the genetic algorithm and that requires less computation time since the required involved populations are smaller and the convergence is faster. Compared with GREEDY, it reaches better solutions after a number of generations both in experiments with 50 and 80 input strings.

An attempt to combine the cooperative coevolutionary approach with natural greediness concludes to the design of an improved method, which incorporates both parallelism and greed as described in [66]. The method consists of three

stages. In the first stage, three parallel and independent runs of the cooperative coevolutionary algorithm operate, returning as output the populations of the prefixes and suffixes, instead of the merge string of the best representatives. Also the GREEDY algorithm runs and its solution is split into a prefix and a suffix. In the second stage, two new collections of prefixes and suffixes are generated. The first contains the best $\frac{1}{3}$ individuals of the prefix population of each cooperative coevolutionary run, and the prefix of the greedy solution. The second is constructed similarly by the corresponding suffixes. In the third stage the cooperative coevolutionary algorithm runs with the two collections constructed in the second stage as initial populations, instead of random populations. The experimental results show that this algorithm performs better than the simple cooperative coevolutionary algorithm even if the cardinality of its populations and the number of its generations in each stage are quite smaller.

9.5 Preserving Favoured Subsolutions

An extension of the genetic algorithm motivated by the desire to address the failure of this algorithm in specific domains, is the PUZZLE algorithm described in [67]. It is designed to improve the performance of the genetic algorithm on relative ordering problems, i.e., problems where the order between genes is crucial instead of their global locus in the genome. Corresponding genes to strings and genome to superstring the SSP is exactly a problem of this kind. The main idea behind the PUZZLE algorithm is to preserve good subsolutions found by the genetic algorithm by choosing carefully the combination points between two solutions. In this way, it promotes the assembly of increasingly larger good building blocks from different individuals, a result that explains also the name of this algorithm.

Two different populations are evolved in the PUZZLE algorithm. A population of solutions (s-population) and a population of building subsolutions (b-population). Accordingly, we have the p-individuals and the b-individuals. Notice that this situation is completely different from the one described for the cooperative coevolutionary algorithm, since here the two populations are not complementary components of a complete solution. The interaction between these two populations is performed differently in each way. The fitness of a b-individual depends on the fitness of the s-individuals that contain it, while the choice of the combination points in s-individuals is affected by the b-individuals that contain these points.

The PUZZLE algorithm was compared with the genetic algorithm since it is its extension and with GREEDY. Experimental results with SSP instances over alphabet of cardinality 2 show that the PUZZLE algorithm outperforms both GREEDY and genetic algorithm, producing shorter superstrings in the average. The result is obtained by instances with 50 and 80 strings. Comparing with the cooperative coevolutionary algorithm, PUZZLE is better for instances with 50 strings, whereas it is worse for instances with 80 strings.

In [67], two expansions of the PUZZLE algorithm are discussed. The first one is a direct combination of PUZZLE with cooperative coevolution. The two ideas of the complementary components in different populations and of the solutions and subsolutions also in different populations are combined to derive a new algorithm. During this algorithm four populations are evolved:

1. population of prefixes,
2. population of suffixes,
3. population of building sub-prefixes, and
4. population of building sub-suffixes,

where the interaction between 1 and 2 operates according to cooperative coevolutionary algorithm, and the interaction between 1, 3 and between 2, 4 operates according to algorithm PUZZLE. The second expansion of PUZZLE involves ideas from **messy genetic algorithms** [21]. They are iterative optimization algorithms that use local search techniques, adaptive representation of the genomes, and decision sampling strategies.

9.6 Discrete Neural Network

In computer science, **neural networks** are learning programming structures that simulate the function of biological neural networks as the one constitutes the human brain. They are composed of artificial **neurons** and connections between them called **synapses**. Neural networks are used for solving artificial intelligence problems as well as combinatorial optimization problems.

A discrete neural network used for solving the SSP is described in [35]. Discreteness concerns the values that neurons can handle. In general, it is formed by n neurons, where the state of each neuron $i \in \mathrm{I}_n$ is defined by its output v_i. The vector $V = (v_1, v_2, \ldots, v_n)$ whose components are the corresponding neuron outputs is called the **state vector**. The energy of each state vector is given by the **energy function** of the network. The aim of the network is to minimize the energy function via its learning operation which happens in iterations. The energy function usually coincides with the objective function of the optimization problem to solve, such that a local minimum of the former is also a local, and possibly global, optimum to the latter. In the case of the SSP, and given a string set S, any feasible vector of the neural network represents an order of the strings in S, utilizing the permutation expression of the SSP solutions. So, feasible state vectors are those correspond to permutations, and $v_i = k$ means that string s_k is placed in the i-th place in the superstring. Notice that there is an one-to-one correspondence between neurons and strings in S. In each learning iteration, the neural network searches different solutions using neuron updating schemes. Given a vector $V = (v_1, v_2, \ldots, v_n)$ corresponding to the current state, and two neurons i and j, $1 \leq i < j \leq n$, the network considers updates to the following different states:

- $(v_1, \ldots, v_i, v_{i+1}, \ldots, v_j, v_{j+1}, \ldots, v_n)$,
- $(v_1, \ldots, v_i, v_{j+1}, \ldots, v_n, v_{i+1}, \ldots, v_j)$,

- $(v_{i+1},\dots,v_j,v_1,\dots,v_i,v_{j+1},\dots,v_n)$,
- $(v_{i+1},\dots,v_j,v_{j+1},\dots,v_n,v_1,\dots,v_i)$,
- $(v_{j+1},\dots,v_n,v_1,\dots,v_i,v_{i+1},\dots,v_j)$, and
- $(v_{j+1},\dots,v_n,v_{i+1},\dots,v_j,v_1,\dots,v_i)$,

that correspond to the combinations of the three parts that the state vector is separated into according to the specific two neurons. For each of these candidate solutions the one that decrease mostly the energy function value is selected as the next network state. This procedure is repeated until convergence is detected, thus a state vector is found where the updates with all pairs of neurons do not cause any change. Due to the used update scheme, the network remains in a feasible state along all iterations. Once the network converges, the stable state represents a local minimum of the energy function which is equivalent to a local maximum of the total overlap between the strings in S.

Experimental results are performed with SSP instances for strings of fixed and variable lengths. The neural network algorithm runs 100 times for each instance and its results were compared with those of GREEDY. In experiments with fixed string length, neural network outperforms GREEDY in most cases on average, and always on best results. In experiments with variable string lengths, neural network outperforms GREEDY both on average and best results.

9.7 GRASP with Path Relinking

A Greedy Randomized Adaptive Search Procedure (GRASP) is an iterative metaheuristic for combinatorial optimization, which is implemented as a multi-start procedure where each iteration is made up of a construction phase and a local search phase. The first phase constructs a randomized greedy solution, while the second phase starts at this solution and applies repeated improvement until a locally optimal solution is found. The procedure continues until a termination condition is satisfied such as a maximum number of iterations. The best solution over all iterations is kept as the final result. GRASP seems to produce good quality solutions for a wide variety of combinatorial optimization problems. A survey on GRASP can be found in [47] while an annotated bibliography in [12]. Path Relinking (PR) [19] is an approach to integrate intensification and diversification strategies in search for optimal solutions. PR in the context of GRASP is introduced in [32] as a memory mechanism for utilizing information on previously found good solutions.

In [17], an implementation of GRASP with PR for solving the SSP is presented. It solves large scale SSP instances of more than 1,000 strings and outperforms the GREEDY algorithm in the majority of the tested instances. The proposed method is able to provide multiple near-optimum solutions that is of practical importance for the DNA sequencing, and admits a natural parallel implementation. Extended computational experiments on a set of SSP instances with known optimal solutions, produced by using the integer programming formulation presented in Sect. 5.2, indicate that the new method finds the optimum in most of the cases, and its average error relative to the optimum is close to zero.

10 Asymptotic Behaviour

It can be observed a discrepancy between the theoretical results from the worst-case analysis and the experimental observations from the approximation and heuristic algorithms for the SSP. A possible explanation for this fact is given by the average-case analysis for the problem.

The asymptotic behaviour of the compression achieved by an optimal superstring is analysed in [1] under a certain probability model for the lengths of the strings and the letter distribution in them. The average optimal compression of n strings tends to $\frac{n \log n}{H_\mu}$, where $H_\mu = -\sum_{i=1}^{m} p(a_i) \log p(a_i)$ is the Shannon entropy of the choosing law μ for the letters from the alphabet to construct the strings.

The asymptotic behaviour of some algorithms for the SSP is based on the above result and explains the good performance of the greedy strategies. In [13], the algorithms GREEDY, MGREEDY, and NAIVE are analysed in a probabilistic framework and it is proved that they are asymptotically optimal. In [65], the results of the asymptotic behaviour are extended to the TGREEDY and DIMATCH algorithms, after the observation that the performance of TGREEDY is never worse than that of MGREEDY, and that the intermediate result of the maximum directed matching in DIMATCH coincides actually with the result of MGREEDY (see Theorem 12). The steps of DIMATCH up to the construction of the maximum directed matching are analysed in a probabilistic way with the additional assumption that all strings have the same length, and the asymptotic optimality of these algorithms is established.

By the complexity results in Sect. 3.2, we know that there is not PTAS for the SSP for both performance measures unless P = NP. In [48], a *probabilistic* PTAS for the SSP that achieves a $(1+\varepsilon)$-approximation in *expected* polynomial time, for every $\varepsilon > 0$, is presented. This algorithm

1. either returns a possibly non-optimal solution, the solution of GREEDY, in polynomial time,
2. or returns an optimal solution, via a maximum Hamiltonian path on the associated overlap graph, in non-polynomial time.

Under certain conditions in the data of the SSP instance, in the first case GREEDY has asymptotic approximation ratio $1+\varepsilon$ with respect to the length measure, and in the second case the *expected* running time of finding the maximum Hamiltonian path can be polynomial, since it depends on the time spent when it is executed and its execution probability. Analysing these situations, for a random input the algorithm has approximation ratio $1+\varepsilon$ with respect to the length measure and polynomial expected running time.

11 Smoothed Analysis

The classical complexity analysis implies that the SSP is a hard problem in the worst case. The average-case analysis explains the effectiveness of greedy strategies under suitable probability models which are far from reality. In addition to these two

frameworks, the latest developed smoothed analysis explains why greed works so well for the SSP in real-world instances of the DNA sequencing practice. Smoothed analysis is introduced in [52] to demonstrate the fact that some algorithms like the simplex algorithm run in exponential time in the worst case, but in practice they are very efficient.

In [36], the smoothed analysis of the GREEDY algorithm is realized, making the observation that the asymptotic optimal behaviour of the greedy techniques is due to the fact that the random strings do not have large overlaps, and so the concatenation of the strings is not much longer than the shortest common superstring. However, the practical instances arising from DNA assembly are not random and the input strings have significantly large overlaps. By defining small and natural perturbations that represent the mutations of the DNA sequences during evolution, it is proved that for any given instance S of the SSP, the average approximation ratio of the GREEDY algorithm on a small random perturbation of S is $1+o(1)$. This result points out that the approximation inefficiency of SSP instances indicating by the Max-SNP-hardness result can be destroyed by a very small perturbation. As very handily noted, if there had been a hard instance for the DNA assembly problem in history, the hardness would have likely been destroyed by the random mutations of the DNA sequences during the evolution. This result makes the SSP a characteristic case where the complexity is different in the worst-case analysis and in the smoothed analysis.

Acknowledgements This research has been funded by the European Union (European Social Fund—ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF)—Research Funding Program: Thalis. Investing in knowledge society through the European Social Fund.

References

1. Alexander, K.S.: Shortest common superstrings for strings of random letters. In: Crochemore, M., Gusfield, D. (eds.) Combinatorial Pattern Matching. Lecture Notes in Computer Science, vol. 807, pp. 164–172. Springer, Berlin (1994)
2. Armen, C., Stein, C.: Improved length bounds for the shortest superstring problem. In: Akl, S., Dehne, F., Sack, J.R., Santoro, N. (eds.) Algorithms and Data Structures. Lecture Notes in Computer Science, vol. 955, pp. 494–505. Springer, Berlin (1995)
3. Armen, C., Stein, C.: Short superstrings and the structure of overlapping strings. J. Comput. Biol. **2**(2), 307–332 (1995)
4. Armen, C., Stein, C.: A $2\frac{2}{3}$-approximation algorithm for the shortest superstring problem. In: Hirschberg, D., Myers, G. (eds.) Combinatorial Pattern Matching. Lecture Notes in Computer Science, vol. 1075, pp. 87–101. Springer, Berlin (1996)
5. Arora, S., Lund, C., Motwani, R., Sudan, M., Szegedy, M.: Proof verification and the hardness of approximation problems. J. ACM **45**(3), 501–555 (1998)
6. Berger, B., Rompel, J., Shor, P.W.: Efficient NC algorithms for set cover with applications to learning and geometry. J. Comput. Syst. Sci. **49**(3), 454–477 (1994)
7. Bläser, M.: An 8/13-approximation algorithm for the asymmetric maximum TSP. In: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '02), pp. 64–73. Society for Industrial and Applied Mathematics, Philadelphia (2002)

8. Blum, A., Jiang, T., Li, M., Tromp, J., Yannakakis, M.: Linear approximation of shortest superstrings. J. ACM **41**, 630–647 (1994)
9. Breslauer, D., Jiang, T., Jiang, Z.: Rotations of periodic strings and short superstrings. J. Algorithm. **24**, 340–353 (1997)
10. Chirico, N., Vianelli, A., Belshaw, R.: Why genes overlap in viruses. Proc. R. Soc. B. Biol. Sci. **277**(1701), 3809–3817 (2010)
11. Czumaj, A., Gąsieniec, L., Piotrów, M., Rytter, W.: Sequential and parallel approximation of shortest superstrings. J. Algorithm. **23**, 74–100 (1997)
12. Festa, P., Resende, M.: GRASP: An annotated bibliography. In: Ribeiro, C., Hansen, P. (eds.) Essays and Surveys in Metaheuristics. Operations Research/Computer Science, pp. 325–367. Kluwer Academic, Dordecht (2002)
13. Frieze, A., Szpankowski, W.: Greedy algorithms for the shortest common superstring that are asymptotically optimal. Algorithmica **21**, 21–36 (1998)
14. Gallant, J.K.: String compression algorithms. Ph.D. thesis, Princeton (1982)
15. Gallant, J., Maier, D., Storer, J.A.: On finding minimal length superstrings. J. Comput. Syst. Sci. **20**(1), 50–58 (1980)
16. Gerver, M.: Three-valued numbers and digraphs. Kvant **1987**(2), 32–35 (1987)
17. Gevezes, T., Pitsoulis, L.: A greedy randomized adaptive search procedure with path relinking for the shortest superstring problem. J. Comb. Optim. (2013) doi: 10.1007/s10878-013-9622-z
18. Gingeras, T., Milazzo, J., Sciaky, D., Roberts, R.: Computer programs for the assembly of DNA sequences. Nucleic Acids Res. **7**(2), 529–543 (1979)
19. Glover, F., Laguna, M.: Tabu Search. Kluwer Academic, Norwell (1997)
20. Goldberg, M.K., Lim, D.T.: A learning algorithm for the shortest superstring problem. In: Proceedings of the Atlantic Symposium on Computational Biology and Genome Information and Technology, pp. 171–175 (2001)
21. Goldberg, D., Deb, K., Korb, B.: Messy genetic algorithms: Motivation, analysis, and first results. Complex Syst. **3**, 493–530 (1989)
22. Holland, J.H.: Adaptation in Natural and Artificial Systems. The University of Michigan Press, Ann Arbor (1975)
23. Huffman, D.A.: A method for the construction of minimum-redundancy codes. Proc. Inst. Radio Eng. **40**(9), 1098–1101 (1952)
24. Ilie, L., Popescu, C.: The shortest common superstring problem and viral genome compression. Fundam. Inform. **73**, 153–164 (2006)
25. Ilie, L., Tinta, L., Popescu, C., Hill, K.A.: Viral genome compression. In: Mao, C., Yokomori, T. (eds.) DNA Computing. Lecture Notes in Computer Science, vol. 4287, pp. 111–126. Springer, Berlin (2006)
26. Jenkyns, T.A.: The greedy travelling salesman's problem. Networks **9**(4), 363–373 (1979)
27. Jiang, T., Li, M.: Approximating shortest superstrings with constraints. Theor. Comput. Sci. 134(2), 473–491 (1994)
28. Kaplan, H., Shafrir, N.: The greedy algorithm for shortest superstrings. Inf. Process. Lett. **93**, 13–17 (2005)
29. Kaplan, H., Lewenstein, M., Shafrir, N., Sviridenko, M.: Approximation algorithms for asymmetric TSP by decomposing directed regular multigraphs. J. ACM **52**, 602–626 (2005)
30. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (eds.) Complexity of Computer Computations, pp. 85–103. Plenum Press, New York (1972)
31. Kosaraju, S.R., Park, J.K., Stein, C.: Long tours and short superstrings. In: Proceedings of the 35th Annual Symposium on Foundations of Computer Science, pp. 166–177. IEEE Computer Society, Washington, DC (1994)
32. Laguna, M., Martí, R.: GRASP and path relinking for 2-layer straight line crossing minimization. INFORMS J. Comput. **11**, 44–52 (1999)
33. Lesk, A.M.: Computational Molecular Biology. Sources and Methods for Sequence Analysis. Oxford University Press, Oxford (1988)
34. Li, M.: Towards a DNA Sequencing Theory (Learning a String), vol. 1, pp. 125–134. IEEE Computer Society, Los Alamitos (1990)

35. López-Rodríguez, D., Mérida-Casermeiro, E.: Shortest common superstring problem with discrete neural networks. In: Kolehmainen, M., Toivanen, P., Beliczynski, B. (eds.) Adaptive and Natural Computing Algorithms. Lecture Notes in Computer Science, vol. 5495, pp. 62–71. Springer, Berlin (2009)
36. Ma, B.: Why greed works for shortest common superstring problem. In: Ferragina, P., Landau, G. (eds.) Combinatorial Pattern Matching. Lecture Notes in Computer Science, vol. 5029, pp. 244–254. Springer, Berlin (2008)
37. Maier, D., Storer, J.A.: A note on the complexity of the superstring problem. Technical Report 233, Computer Science Laboratory, Princeton University, Princeton (1977)
38. Middendorf, M.: More on the complexity of common superstring and supersequence problems. Theor. Comput. Sci. **125**(2), 205–228 (1994)
39. Middendorf, M.: Shortest common superstrings and scheduling with coordinated starting times. Theor. Comput. Sci. **191**(1–2), 205–214 (1998)
40. Miller, C.E., Tucker, A.W., Zemlin, R.A.: Integer programming formulation of traveling salesman problems. J. ACM **7**, 326–329 (1960)
41. Ott, S.: Lower bounds for approximating shortest superstrings over an alphabet of size 2. In: Widmayer, P., Neyer, G., Eidenbenz, S. (eds.) Graph-Theoretic Concepts in Computer Science. Lecture Notes in Computer Science, vol. 1665, pp. 55–64. Springer, Berlin (1999)
42. Papadimitriou, C.H., Steiglitz, K.: Combinatorial optimization: algorithms and complexity. Prentice-Hall, Englewood Cliffs (1982)
43. Papadimitriou, C.H., Yannakakis, M.: Optimization, approximation, and complexity classes. J. Comput. Syst. Sci. **43**(3), 425–440 (1991)
44. Papadimitriou, C.H., Yannakakis, M.: The traveling salesman problem with distances one and two. Math. Oper. Res. **18**(1), 1–11 (1993)
45. Peltola, H., Söderlund, H., Ukkonen, E.: SEQAID: a DNA sequence assembling program based on a mathematical model. Nucleic Acids Res. **12**(1), 307–321 (1984)
46. Pevzner, P.A., Waterman, M.S.: Open Combinatorial Problems in Computational Molecular Biology, p. 158. IEEE Computer Society, Los Alamitos (1995)
47. Pitsoulis, L., Resende, M.: Greedy randomized adaptive search procedures. In: Pardalos, P., Resende, M. (eds.) Handbook of Applied Optimization, pp. 178–183. Oxford University Press, Oxford (2002)
48. Plociennik, K.: A probabilistic PTAS for shortest common superstring. In: Proceedings of the 34th International Symposium on Mathematical Foundations of Computer Science 2009 (MFCS ’09), pp. 624–635. Springer, Berlin (2009)
49. Reif, J.H.: Synthesis of Parallel Algorithms, 1st edn. Morgan Kaufmann, San Francisco (1993)
50. Shannon, C.E.: A mathematical theory of communication. Bell Syst. Tech. J. **27**, 379–423, 623–656 (1948)
51. Shapiro, M.B.: An algorithm for reconstructing protein and RNA sequences. J. ACM **14**, 720–731 (1967)
52. Spielman, D., Teng, S.H.: Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. In: Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing (STOC ’01), pp. 296–305. ACM, New York (2001)
53. Staden, R.: Automation of the computer handling of gel reading data produced by the shotgun method of DNA sequencing. Nucleic Acids Res. **10**(15), 4731–4751 (1982)
54. Storer, J.A.: Data compression: Methods and theory. Computer Science Press, New York (1988)
55. Storer, J.A., Szymanski, T.G.: The macro model for data compression (extended abstract). In: Proceedings of the Tenth Annual ACM Symposium on Theory of Computing (STOC ’78), pp. 30–39. ACM, New York (1978)
56. Storer, J.A., Szymanski, T.G.: Data compression via textual substitution. J. ACM **29**, 928–951 (1982)
57. Sweedyk, Z.: A $2\frac{1}{2}$-approximation algorithm for shortest superstring. SIAM J. Comput. **29**, 954–986 (1999)

58. Tarhio, J., Ukkonen, E.: A greedy approximation algorithm for constructing shortest common superstrings. Theor. Comput. Sci. **57**(1), 131–145 (1988)
59. Tarjan, R.E.: Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics, Philadelphia (1983)
60. Teng, S.H., Yao, F.: Approximating Shortest Superstrings, pp. 158–165. IEEE Computer Society, Los Alamitos (1993)
61. Timkovskii, V.G.: Complexity of common subsequence and supersequence problems and related problems. Cybern. Syst. Anal. **25**, 565–580 (1989)
62. Turner, J.S.: Approximation algorithms for the shortest common superstring problem. Inf. Comput. **83**, 1–20 (1989)
63. Valiant, L.G.: A theory of the learnable. Commun. ACM **27**(11), 1134–1142 (1984)
64. Vassilevska, V.: Explicit inapproximability bounds for the shortest superstring problem. In: Jedrzejowicz, J., Szepietowski, A. (eds.) Mathematical Foundations of Computer Science 2005. Lecture Notes in Computer Science, vol. 3618, pp. 793–800. Springer, Berlin (2005)
65. Yang, E., Zhang, Z.: The shortest common superstring problem: Average case analysis for both exact and approximate matching. IEEE Trans. Inf. Theory **45**(6), 1867–1886 (1999)
66. Zaritsky, A., Sipper, M.: Coevolving solutions to the shortest common superstring problem. Biosystems **76**(1–3), 209–216 (2004)
67. Zaritsky, A., Sipper, M.: The preservation of favored building blocks in the struggle for fitness: The puzzle algorithm. Evol. Comput. **8**(5), 443–455 (2004)