Collapsing Superstring Conjecture

Alexander Golovnev
Harvard University, Cambridge, MA, USA

Alexander S. Kulikov
Steklov Institute of Mathematics at St. Petersburg, Russian Academy of Sciences, Russia

Alexander Logunov
St. Petersburg State University, Russia

Ivan Mihajlin
University of California, San Diego, CA, USA

Maksim Nikolaev
St. Petersburg State University, Russia

Abstract

In the Shortest Common Superstring (SCS) problem, one is given a collection of strings, and needs to find a shortest string containing each of them as a substring. SCS admits $2\frac{11}{23}$-approximation in polynomial time (Mucha, SODA'13). While this algorithm and its analysis are technically involved, the 30 years old Greedy Conjecture claims that the trivial and efficient Greedy Algorithm gives a 2-approximation for SCS.

We develop a graph-theoretic framework for studying approximation algorithms for SCS. The framework is reminiscent of the classical 2-approximation for Traveling Salesman: take two copies of an optimal solution, apply a trivial edge-collapsing procedure, and get an approximate solution. In this framework, we observe two surprising properties of SCS solutions, and we conjecture that they hold for all input instances. The first conjecture, that we call Collapsing Superstring conjecture, claims that there is an elementary way to transform any solution repeated twice into the same graph G. This conjecture would give an elementary 2-approximate algorithm for SCS. The second conjecture claims that not only the resulting graph G is the same for all solutions, but that G can be computed by an elementary greedy procedure called Greedy Hierarchical Algorithm.

While the second conjecture clearly implies the first one, perhaps surprisingly we prove their equivalence. We support these equivalent conjectures by giving a proof for the special case where all input strings have length at most 3 (which until recently had been the only case where the Greedy Conjecture was proven). We also tested our conjectures on millions of instances of SCS.

We prove that the standard Greedy Conjecture implies Greedy Hierarchical Conjecture, while the latter is sufficient for an efficient greedy 2-approximate approximation of SCS. Except for its (conjectured) good approximation ratio, the Greedy Hierarchical Algorithm provably finds a 3.5-approximation, and finds *exact* solutions for the special cases where we know polynomial time (not greedy) exact algorithms: (1) when the input strings form a spectrum of a string (2) when all input strings have length at most 2.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Approximation algorithms analysis

Keywords and phrases superstring, shortest common superstring, approximation, greedy algorithms, greedy conjecture

Digital Object Identifier 10.4230/LIPIcs.APPROX-RANDOM.2019.26

Category APPROX

Related Version A full version of the paper is available at `https://arxiv.org/abs/1809.08669`.

Funding *Alexander Golovnev*: Supported by a Rabin Postdoctoral Fellowship.

Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2019).
Editors: Dimitris Achlioptas and László A. Végh; Article No. 26; pp. 26:1–26:23

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The *shortest common superstring problem* (abbreviated as SCS) is: given a set of strings, find a shortest string that contains all of them as substrings. This problem finds applications in genome assembly [33, 24], and data compression [9, 8, 28]. We refer the reader to the excellent surveys [10, 21] for an overview of SCS, its applications and algorithms. SCS is known to be **NP**-hard [9] and even **MAX-SNP**-hard [3], but it admits constant-factor approximation in polynomial time.

The best known approximation ratios are $2\frac{11}{23}$ due to Mucha [22] and $2\frac{11}{30}$ due to Paluch [23] (see [12, Section 2.1] for an overview of the previous approximation algorithms and inapproximability results). While these approximation algorithms use an algorithm for Maximum Weight Perfect Matching as a subroutine, the 30 years old *Greedy Conjecture* [28, 30, 31, 3] claims that the trivial *Greedy Algorithm*, whose pseudocode is given in Algorithm 1, is 2-approximate. Ukkonen [32] shows that for a fixed alphabet, the Greedy Algorithm can be implemented in linear time. It should be noted that GA is not deterministic as we do not specify how to break ties in case when there are many pairs of strings with maximum overlap. For this reason, GA may produce different superstrings for the same input.

Algorithm 1 Greedy Algorithm (GA).

Input: set of strings $\mathcal{S}$.
Output: a superstring for $\mathcal{S}$.

```
while S contains at least two strings do
    extract from S two strings with the maximum overlap
    add to S the shortest superstring of these two strings
return the only string from S
```

▶ **Greedy Conjecture.** *For any set of strings $\mathcal{S}$,* $\mathrm{GA}(\mathcal{S})$ *constructs a superstring that is at most twice longer than an optimal one.*

Blum et al. [3] prove that the Greedy Algorithm returns a 4-approximation of SCS, and Kaplan and Shafrir [15] improve this bound to 3.5. A slight modification of the Greedy Algorithm gives a 3-approximation of SCS [3], and other greedy algorithms are studied from theoretical [3, 25] and practical perspectives [26, 4].

It is known that the Greedy Conjecture holds for the case when all input strings have length at most 3 [30, 7], and it was recently shown to hold in the case of strings of length 4 [18]. Also, the Greedy Conjecture holds if the Greedy Algorithm happens to merge strings in a particular order [35, 19]. The Greedy Algorithm gives a 2-approximation of a different metric called compression [30]. The compression is defined as the sum of the lengths of all input strings minus the length of a superstring (hence, it is the number of symbols saved with respect to a naive superstring resulting from concatenating the input strings).

Most of the approaches for approximating SCS are based on the *overlap graph* or the equivalent *suffix graph*. The suffix graph has input strings as nodes, and a pair of nodes is joined by an arc of weight equal to their suffix (see Section 2.1 for formal definitions of overlap and suffix). SCS is equivalent to (the asymmetric version of) the Traveling Salesman Problem (TSP) in the suffix graph. While TSP cannot be approximated within any polynomial time computable function unless $\mathbf{P} = \mathbf{NP}$ [27], its special case corresponding to SCS can be

approximated within a constant factor.[1] We do not know the full characterization of the graphs in this special case, but we know some of their properties: Monge inequality [20] and Triple inequality [35]. These properties are provably not sufficient for proving Greedy Conjecture [35, 19].

While the overlap and suffix graphs give a convenient graph structure, our current knowledge of their properties is provably not sufficient for showing strong approximation factors. Thus, the known approximation algorithms (including the Greedy Algorithm) estimate the approximation ratio via the overlap graph, and also separately take into account some string properties not represented by the overlap graph. The goal of this work is to develop a simple combinatorial framework which captures all features of the input strings needed for proving approximation ratios of algorithms.

1.1 Our contributions

We continue the study of the so-called *hierarchical graph* introduced by Golovnev et al. [13]. (See also [5] for a related notion of the superstring graph.) This graph is designed specifically for the SCS problem, in some sense it generalizes de Bruijn graph, and it contains more information about the input strings than just all pairwise overlaps. Given an instance of SCS, the vertex set of the corresponding hierarchical graph is just the set of substrings of all the input strings. For a string s and two symbols α, β, the graph contains the arcs: $(s, s\alpha)$ and $(\beta s, s)$. Now, every superstring of the given set of string corresponds to an Eulerian walk in the hierarchical graph (which passes through the vertices corresponding to the input strings), and vice versa. (See Section 2.2 for formal definition and statements.)

1.1.1 Collapsing Conjecture

We define a simple normalization procedure of a walk in the hierarchical graph: replace the pair of arcs $(\alpha s, \alpha s\beta), (\alpha s\beta, s\beta)$ with the pair $(\alpha s, s), (s, s\beta)$ as long as it does not violate connectivity of the walk. It is easy to see that such a normalization never increases the length of the corresponding solution of SCS. First, we observe a surprising property of this normalization procedure: if one takes *any* solution, doubles all of its arcs in the hierarchical graph, and then applies the normalization procedure, then the resulting set of arcs is always the same (i.e., it does not depend on the initial solution). Collapsing Conjecture makes this observation formal (see Section 3). Note that this conjecture implies an extremely simple 2-approximate algorithm for Shortest Common Superstring: take any solution (for example, write down all input strings one after another), then double each arc in the hierarchical graph, and apply the simple normalization procedure. This procedure will result in some superstring S. On the other hand, if one started with an optimal solution, doubled each of its arcs, and normalized the result, then the resulting solution would have length at most twice the length of the optimal solution. By Collapsing Conjecture, this resulting superstring would also be S, which implies that S is a 2-approximation.

[1] We remark that SCS is also a special case of TSP for costs satisfying the triangle inequaliy. This case of TSP can be approximated within a constant factor [29], but this factor is currently much worse than that for SCS.

1.1.2 Greedy Hierarchical Conjecture

We also propose a simple and natural greedy algorithm for SCS in the hierarchical graph: start from the nodes corresponding to the input strings, and greedily build an Eulerian walk passing through all of them. While this Greedy Hierarchical Algorithm (GHA) is as simple as the Greedy Algorithm (GA), it provably performs better in some cases. For example, there are two well-known polynomially solvable special cases of SCS: strings of length 2 and a spectrum of a string. While GA does not always find optimal solutions in these cases, GHA solves them exactly (see Sections B.1 and B.2).

Greedy Hierarchical Conjecture (see Section 4) claims that the set of arcs produced by GHA exactly matches the set of arcs from the Collapsing Conjecture: whichever initial solution one takes, after doubling its arcs and normalization, the resulting set of arcs is exactly the solution found by GHA. Clearly, this conjecture implies Collapsing Conjecture. Perhaps surprisingly, we prove that the two conjectures are equivalent (see Section 5.1): if all doubled solutions after normalization result in the same set of arcs, then this set of arcs is the GHA solution.

The weak form of Greedy Hierarchical Conjecture claims that GHA is a 2-approximate algorithm for SCS. We prove (see Section 5.2) that GHA is an instantiation of GA with some tie-breaking rule. That is, there is an algorithm which always merges some pair of strings with the longest overlap and outputs the same solution as GHA. This result has two consequences. First, by the known results for GA, we immediately have that GHA finds a 3.5-approximation for SCS. Second, this gives us that Greedy Conjecture implies Weak Greedy Hierarchical Conjecture.

1.1.3 Evidence for the Conjectures

We support the Collapsing Conjecture (and the equivalent Greedy Hierarchical Conjecture) by proving its special case and verifying it empirically. We prove the conjecture for the special case where all input strings have length at most 3, which until recently had been the only case where the Greedy Conjecture was proven (see Section A). Despite tesing the conjecture on millions of datasets (both hand-crafted and generated randomly according to various distributions), we have not found a counter-example. Note that even the Weak Greedy Hierarchical Conjecture suffices for getting a 2-approximation for SCS, and this conjecture is not harder to prove than the standard Greedy Conjecture. We implemented the Greedy Hierarchical Algorithm [11], and we invite the reader to its web interface [34] to see step by step executions of the described algorithms and to verify the conjectures on custom datasets.

2 Definitions

2.1 Shortest Common Superstring Problem

For a string s, by $|s|$ we denote the length of s. For strings s and t, by $\mathrm{overlap}(s,t)$ we denote the longest suffix of s that is also a prefix of t. By $\mathrm{pref}(s,t)$ we denote the first $|s| - |\mathrm{overlap}(s,t)|$ symbols of s. Similarly, $\mathrm{suff}(s,t)$ is the last $|t| - |\mathrm{overlap}(s,t)|$ symbols of t. By $\mathrm{pref}(s)$ and $\mathrm{suff}(s)$ we denote, respectively, the first and the last $|s| - 1$ symbols of s. See Figure 1 for a visual explanation. We denote the empty string by ε.

Throughout the paper by $\mathcal{S} = \{s_1, \ldots, s_n\}$ we denote the set of n input strings. We assume that no input string is a substring of another (such a substring can be removed from $\mathcal{S}$ in the preprocessing stage). Note that SCS is a *permutation problem*: to find a shortest

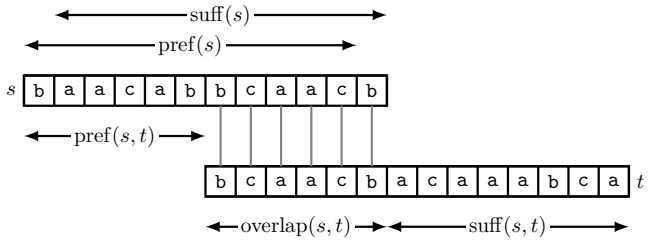


Figure 1 Pictorial explanations of pref, suff, and overlap functions.

string containing all s_i's in a *given order* one just overlaps the strings in this order, see Figure 2. (This simple observation relates SCS to other permutation problems, including various versions of the Traveling Salesman Problem.) It will prove convenient to view the SCS problem as a problem of finding an optimum permutation. It should be noted at the same time that the correspondence between permutations and superstrings is not one-to-one: there are superstrings that do not correspond to any permutation. For example, the concatenation of input strings is clearly a superstring, but it ignores the fact that neighbor strings may have non-trivial overlaps and for this reason may fail to correspond to a permutation. Still, clearly, any *shortest* superstring corresponds to a permutation of the input strtings.

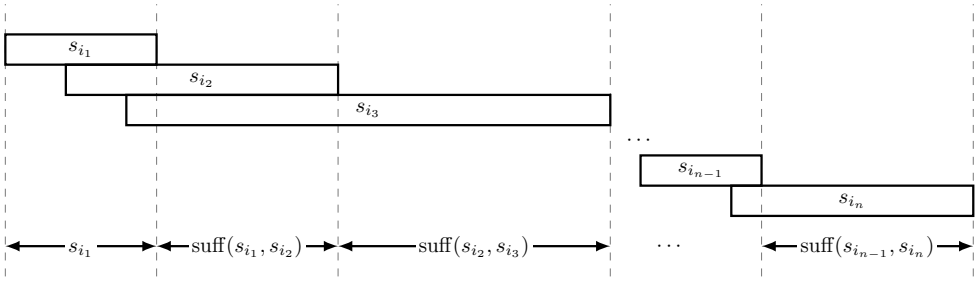


Figure 2 SCS is a permutation problem. The length of a superstring corresponding to a permutation $(s_{i_1}, \ldots, s_{i_n})$ is $|s_{i_1}|$ plus the sum of the lengths of suffixes of consecutive pairs of strings. It is also equal to $\sum_{i=1}^{n} |s_i| - \sum_{j=1}^{n-1} |\operatorname{overlap}(s_{i_j}, s_{i_{j+1}})|$.

2.2 Hierarchical Graph

For a set of strings $\mathcal{S}$, the *hierarchical graph* $HG = (V, E)$ is a weighted directed graph with $V = \{v\colon v \text{ is a substring of some } s \in \mathcal{S}\}$. For every $v \in V$, $v \neq \varepsilon$, the set of arcs E contains an *up-arc* $(\operatorname{pref}(v), v)$ of weight 1 and a *down-arc* $(v, \operatorname{suff}(v))$ of weight 0. The meaning of an up-arc is appending one symbol to the end of the current string (and that is why it has weight 1), whereas the meaning of a down-arc is cutting down one symbol from the beginning of the current string. Figure 3(a) gives an example of the hierarchical graph and shows that the terminology of up- and down-arcs comes from placing all the strings of the same length at the same level, where the i-th level contains strings of length i. In all the figures in this paper, the input strings are shown in rectangles, while all other vertices are ellipses.

What we are looking for in this graph is a shortest walk from ε to ε going through all the nodes from $\mathcal{S}$. It is not difficult to see that the length of a walk from ε to ε equals the length of the string spelled by this walk. This is just because each up-arc has weight 1 and adds one symbol to the current string. See Figure 3(b) for an example.

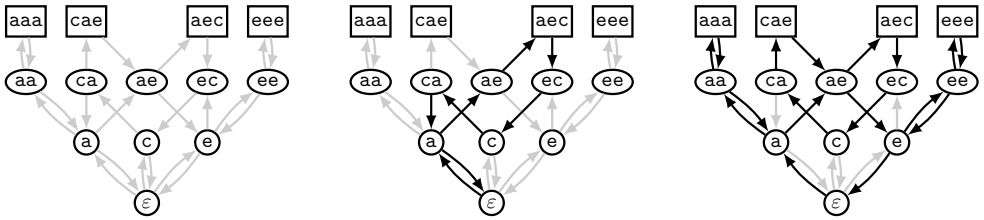


Figure 3 (a) Hierarchical graph for the dataset $\mathcal{S} = \{\texttt{aaa}, \texttt{cae}, \texttt{aec}, \texttt{eee}\}$. (b) The walk $\varepsilon \to \texttt{a} \to \texttt{ae} \to \texttt{aec} \to \texttt{ec} \to \texttt{c} \to \texttt{ca} \to \texttt{a} \to \varepsilon$ has length (or weight) 4 and spells the string aeca of length 4. (c) An optimal superstring for $\mathcal{S}$ is aaaecaeee. It has length 9, corresponds to the permutation (aaa, aec, cae, eee), and defines the walk of length 9 shown in black.

Hence, the SCS problem is equivalent to finding a shortest closed walk from ε to ε that visits all nodes from $\mathcal{S}$. Note that a walk may contain repeated nodes and arcs. The multiset of arcs of such a walk must be Eulerian (each vertex must have the same in- and out-degree, and the set of arcs must be connected). It will prove convenient to define an *Eulerian solution* in a hierarchical graph as an Eulerian multiset of arcs D that goes through ε and all nodes from $\mathcal{S}$. Given such a solution D, one can easily recover an Eulerian cycle (that might not be unique). This cycle spells a superstring of $\mathcal{S}$ of the same length as D. Figure 3(c) shows an optimal Eulerian solution.

A solution to SCS defines a permutation $(s_{i_1}, \ldots, s_{i_n})$ of the input strings, and this permutation naturally gives a "zig-zag" Eulerian solution in the hierarchical graph:

$$\varepsilon \to s_{i_1} \to \text{overlap}(s_{i_1}, s_{i_2}) \to s_{i_2} \to \text{overlap}(s_{i_2}, s_{i_3}) \to \cdots \to s_{i_n} \to \varepsilon\,. \tag{1}$$

This Eulerian solution is shown schematically in Figure 4(a). This schematic illustration is over simplified as the shown path usually has many self-intersections. Still, this point of view is helpful in understanding the algorithms presented later in the text. Figure 4(b) shows an "untangled" optimal Eulerain solution from Figure 3(c): by contracting nodes with equal labels into the same node, one gets exactly the solution from Figure 3(c).

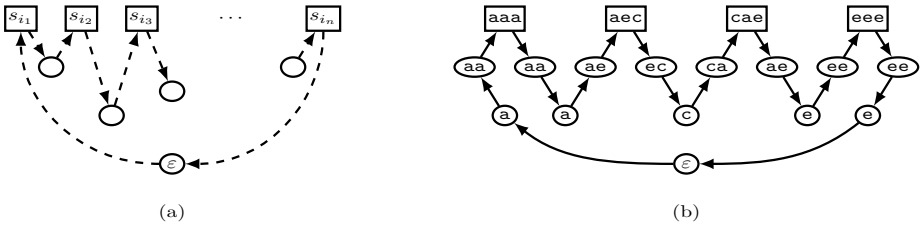


Figure 4 (a) A schematic illustration of a normalized Eulerian solution. (b) Untangled optimal Eulerian solution from Figure 3(c).

Not every Eulerian solution in the hierarchical graph has a nice zig-zag structure described above. In the next section, we introduce a normalization procedure (that we call collapsing) that allows us to focus on nice Eulerian solutions only.

2.3 Normalizing a Solution

In this section, we describe a natural way of normalizing an Eulerian solution D. Informally, it can be viewed as follows. Imagine that all arcs of D form one circular thread, and that there is a nail in every node $s \in \mathcal{S}$ corresponding to an input string. We apply *"gravitation"* to the thread, i.e., we replace every pair of arcs $(\mathrm{pref}(v), v)$, $(v, \mathrm{suff}(v))$ with a pair $(\mathrm{pref}(v), \mathrm{pref}(\mathrm{suff}(v)))$, $(\mathrm{pref}(\mathrm{suff}(v)), \mathrm{suff}(v))$, if there is no nail in v and if this does not disconnect D. We call this *collapsing*, see Figure 5.

Figure 5 Collapsing a pair of arcs is replacing a pair of dashed arcs with a pair of solid arcs: general case (left) and example (right). The "physical meaning" of this transformation is that to get `bac` from `aba` one needs to cut `a` from the beginning and append `c` to the end and these two operations commute.

A formal pseudocode of the collapsing procedure is given in Algorithm 2. The pseudocode, in particular, reveals an important exception (not covered in Figure 5): if $|v| = 1$, then $\mathrm{pref}(\mathrm{suff}(v))$ is undefined and we just remove the pair of arcs $(\mathrm{pref}(v), v)$ and $(v, \mathrm{suff}(v))$.

Algorithm 2 Collapse.

Input: hierarchical graph $HG(V, E)$, Eulerian solution D, node $v \in V$.

if $(\mathrm{pref}(v), v), (v, \mathrm{suff}(v)) \in D$ **then**
  $D \leftarrow D \setminus \{(\mathrm{pref}(v), v), (v, \mathrm{suff}(v))\}$
  if $|v| > 1$ **then**
    $D \leftarrow D \cup \{(\mathrm{pref}(v), \mathrm{pref}(\mathrm{suff}(v))), (\mathrm{pref}(\mathrm{suff}(v)), \mathrm{suff}(v))\}$

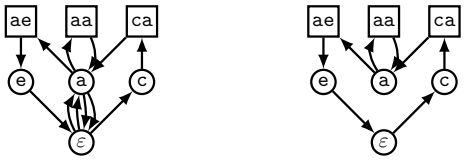
Algorithm 3, that we call Collapsing Algorithm (CA), uses the property described above to normalize any solution. It drops down all pairs of arcs that are not needed for connectivity. (Recall that a set of edges is called an Eulerian solution if it is connected and goes through all initial nodes and ε.)

Algorithm 3 Collapsing Algorithm (CA).

Input: set of strings $\mathcal{S}$, Eulerian solution D in HG.
Output: Eulerian solution D': $|D'| \leqslant |D|$

for level l in HG in descending order **do**
  for all $v \in V$ s.t. $|v| = l$ in lexicographic order: **do**
    while $(\mathrm{pref}(v), v), (v, \mathrm{suff}(v)) \in D$ and collapsing it keeps D an Eulerian solution **do**
      COLLAPSE(HG, D, v)
return D

It is easy to show (we prove this formally in Claim 2 on page 14) that any normalized solution is of the form (1). But it is not true that every zig-zag solution of the form (1) is a normalized solution: see Figure 6 for an example. The normalization procedure does not just turn a solution into some standard form, but it may also decrease its length.



(a) (b)

Figure 6 (a) An Eulerian solution corresponding to the permutation (ae, aa, ca). (b) The solution from (a) after normalization results in a shorter solution corresponding to the permutation (ca, aa, ae). This example also shows that though collapsing a pair of edges is a local change in the graph, it may drastically change the resulting superstring. In this case, it replaces a superstring aeaaca with a shorter superstring caae.

3 Collapsing Conjecture

We are now ready to conjecture an astonishing structural property of the hierarchical graph:

> Take any Eulerian solution, double every arc of it, and normalize the resulting solution; the result is the same for all initial solutions!

For the formal statement of the conjecture we use the following notation: If U and V are two multisets, then $U \sqcup V$ is the multiset W such that each $w \in W$ has multiplicity equal to the sum of multiplicities it has in sets U and V. Formally, the conjecture is stated as follows.

▶ **Collapsing Conjecture.** *For any set of strings $\mathcal{S}$ and any two Eulerian solutions D_1, D_2 of $\mathcal{S}$,*

$$CA(\mathcal{S}, D_1 \sqcup D_1) = CA(\mathcal{S}, D_2 \sqcup D_2)\,.$$

Figures 7 and 8 illustrate the action of the Collapsing Algorithm for optimal and naive solutions, respectively. Note that the resulting solutions are equal. When processing level $l > 1$ nodes, the collapsing procedure does not change the total length of the solution. What one normally sees at the beginning of the $l = 1$ iteration is an Eulerian solution with many redundant pairs of arcs of the form $(\mathtt{a}, \varepsilon)$, $(\varepsilon, \mathtt{a})$. It is exactly this stage of the algorithm where the total length of a solution is decreased by the Collapsing Algorithm.

We have verified the conjecture on millions of datasets (both handcrafted and randomly generated), and we invite the reader to see its visualizations and to check the conjecture on arbitrary datasets at the webpage [34]. Moreover, we support the conjecture by proving that it holds in the (NP-hard) special case where the input strings have length at most 3 in Section A.

If the Collapsing Conjecture is true, then there is a simple and natural 2-approximate algorithm for SCS: take *any* Eulerian solution (e.g., merge the input strings in arbitrary order), double it, and apply the Collapsing Algorithm. Under the conjecture, this results in the same Eulerian solution as for doubled optimal solution and hence the length of the result is at most twice the optimal length.

4 Greedy Hierarchical Conjecture

In this section, we present one more curious property of the Collapsing Algorithm that reveals its intricate connection to greedy algorithms. For this, we introduce the so called Greedy Hierarchical Algorithm (GHA) that constructs an Eulerian solution in a stingy fashion, i.e., tries to add as few arcs as possible:

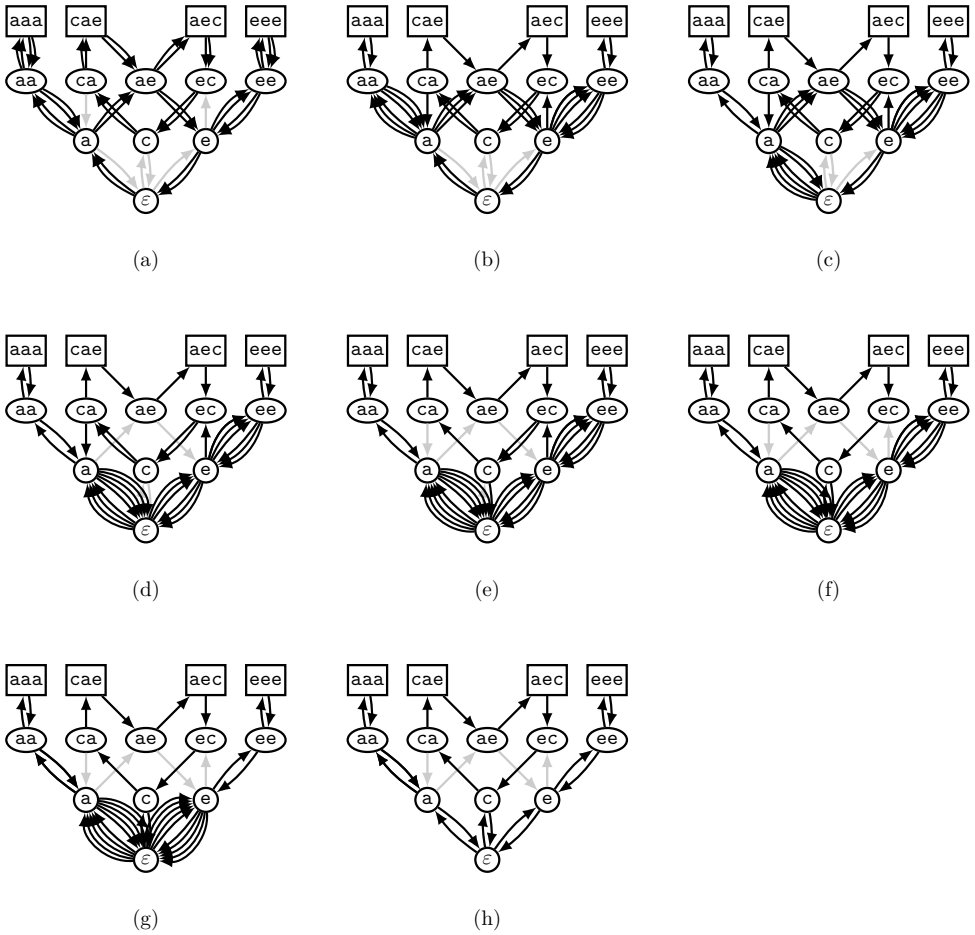


Figure 7 Stages of applying the Collapsing Algorithm to the dataset {aaa, cae, aec, eee} and its **optimal** solution. (a) We start by doubling every arc of the optimal solution from Figure 3(c). (b) After collapsing all nodes at level $l = 3$. (c) After processing the node aa at level $l = 2$. Note that the algorithm leaves a pair of arcs (a, aa), (aa, a) as they are needed to connect the component {aa, aaa} to the rest of the solution. (d) After processing the ae node. The algorithm collapses all pairs of arcs for this node as it lies in the same component as the node c. (e) After processing the ca node. (f) After processing the ec node. (g) After processing the ee node. Note that at this point the solution has exactly the same length as at the very beginning (at stage (a)). (h) Finally, after collapsing all the unnecessary pairs of arcs from the level $l = 1$.

Proceed nodes from top to bottom. For each node, ensure that it is balanced and connected to the rest of the solution.

This is best illustrated with an example, see Figure 9. We start constructing an Eulerian solution D by processing the nodes at level 3. The solution D must visit all these four nodes, so we add all incoming and outgoing arcs to D, see Figure 9(a). We then process the level 2. The node aa is balanced, but if we skip it, it will not be connected to the rest of the solution, so we add to D the arcs (a, aa) and (aa, a). The node ae is balanced, we do nothing for it. The node ca is imbalanced, so we add an arc (c, ca) to D. We balance the node ec similarly. The node ee is processed similarly to the node aa. The result of processing the second level is shown in Figure 9(b). On the last stage we connect the nodes a, b, and c to ε to ensure connectivity, see Figure 9(c). Hence, when processing level l, we only add arcs between levels l and $l - 1$.

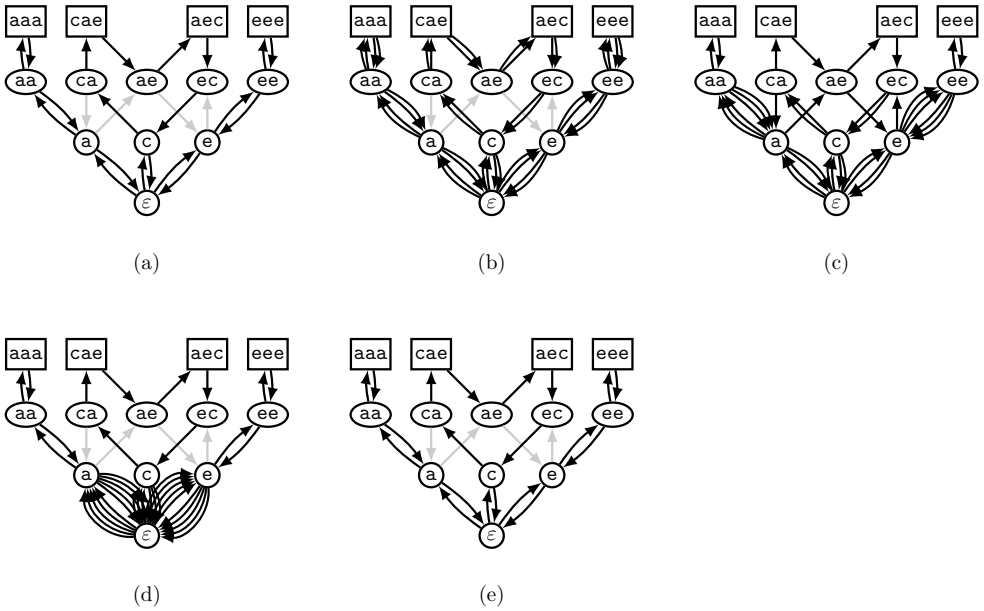


Figure 8 Stages of applying the Collapsing Algorithm to the dataset {aaa, cae, aec, eee} and its **naive** solution resulting from overlapping the input strings in the same order as they are given. (a) The solution of length 10 corresponding to the superstring aaacaeceee. (b) The doubled solution. (c) After collapsing the $l = 3$ level. (d) After collapsing the $l = 2$ level. (e) After collapsing the $l = 1$ level.

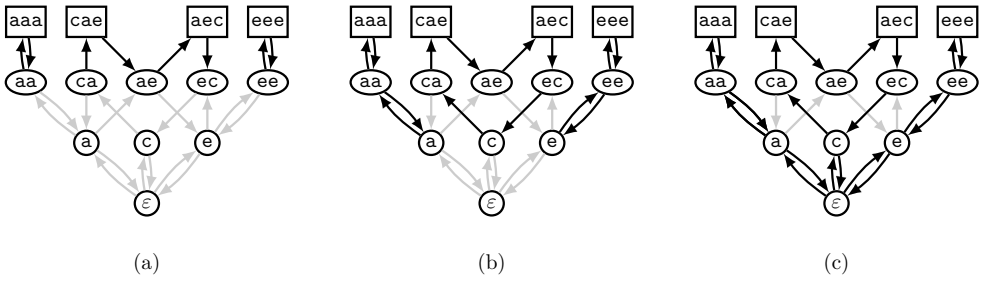


Figure 9 (a) After processing the $l = 3$ level. (b) After processing the $l = 2$ level. Note that for the node aa we add two lower arcs ((a, aa) and (aa, a)) since otherwise the corresponding weakly connected component ({aa, aaa}) will not be connected to the rest of the solution. At the same time, when processing the node ae we observe that it lies in a weakly connected component that contains imbalanced nodes (ca and ec), hence there is no need to add two lower arcs to ae. (c) After processing the $l = 1$ level. The resulting solution has length 10 and is, therefore, suboptimal (compare it with the optimal solution shown in Figure 3(c)).

More formally, GHA first considers the input strings $\mathcal{S}$. Since we assume that no $s \in \mathcal{S}$ is a substring of another $t \in \mathcal{S}$, there is no down-path from t to s in HG. This means that any walk through ε and $\mathcal{S}$ goes through the arcs $\{(\mathrm{pref}(s), s), (s, \mathrm{suff}(s)) \colon s \in \mathcal{S}\}$. The algorithm adds all of them to the constructed Eulerian solution D and starts processing all the nodes level by level, from top to bottom. At each level, we process the nodes in the lexicographic order. If the degree of the current node v is imbalanced, we balance it by adding an appropriate number of incoming (i.e., $(\mathrm{pref}(v), v)$) or outgoing (i.e., $(v, \mathrm{suff}(v))$)

arcs from the previous (i.e., lower) level. In the case when v is balanced, we just skip it. The only exception when we cannot skip it is when *v lies in an Eulerian component and v is the last chance of this component to be connected to the rest of the arcs in D*. (See, for example, the vertex `aa` in Figure 9(a)). The pseudocode is given in Algorithm 4.

Algorithm 4 Greedy Hierarchical Algorithm (GHA).

Input: set of strings $\mathcal{S}$.
Output: Eulerian solution D.

$HG(V, E) \leftarrow$ hierarchical graph of $\mathcal{S}$
$D \leftarrow \{(\text{pref}(s), s), (s, \text{suff}(s)) \colon s \in \mathcal{S}\}$
for level l from $\max\{|s| \colon s \in \mathcal{S}\}$ downto 1 **do**
    for node $v \in V$ with $|v| = l$ in the lexicographic order **do**
        if $|\{(u, v) \in D \colon |u| = |v| + 1\}| \neq |\{(v, w) \in D \colon |w| = |v| + 1\}|$ **then**
            balance the degree of v in D by adding an appropriate number of lower arcs
        else
            $\mathcal{C} \leftarrow$ weakly connected component of v in D
            $u \leftarrow$ the lexicographically largest string among shortest strings in $\mathcal{C}$
            if $\mathcal{C}$ is Eulerian, $\varepsilon \notin \mathcal{C}$, and $v = u$ **then**
                $D \leftarrow D \cup \{(\text{pref}(v), v), (v, \text{suff}(v))\}$
return D

While GHA is almost as simple as the standard Greedy Algorithm (GA), GHA has several provable advantages over GA:

One advantage of GHA over GA is that GHA is more flexible in the following sense. On every step, GA selects two strings and fixes tightly their order. GHA instead works to ensure connectivity. When the resulting set D is connected, an actual order of input strings is given by the corresponding Eulerian cycle through D. This is best illustrated on the following toy example. For the dataset $\mathcal{S} = \{\texttt{ae}, \texttt{ea}, \texttt{ee}\}$, GA might produce a suboptimal solution `aeaee` if it merges the strings `ae` and `ea` at the first step. At the same time, it is not difficult to see that GHA finds an optimal solution for $\mathcal{S}$.

Another advantage of GHA is that, in contrast to GA, it solves *exactly* two well known polynomially solvable special cases of SCS: when the input strings have length at most two and when the input strings form a k-spectrum of an unknown string (that is, the input strings constitute all k-substrings of a string). We prove this formally in Sections B.1 and B.2. Informally, this happens because for such datasets there are no connectivity issues for GHA: for k-spectrum, after processing the highest level GHA gets a weakly connected component; for 2-SCS, after processing the level 2, GHA gets several weekly connected components such that different components do not share common letters and therefore are completely independent. Figure 6(b) illustrates this: while GA may produce a permutation $(\texttt{ca}, \texttt{ae}, \texttt{aa})$, GHA constructs an optimal permutation $(\texttt{ca}, \texttt{aa}, \texttt{ae})$.

In Section B.3, we also show a dataset where GHA produces a solution that is almost two times longer than the optimal one.

In Section 5.2, we show that the approximation guarantee of GHA is no worse than that of GA. Combining with the result of Kaplan and Shafrir [15], this implies immediately that GHA is 3.5-approximate. Moreover, we prove that the standard Greedy Conjecture implies 2-approximation of GHA, which makes it natural to study the approximation ratio of GHA.

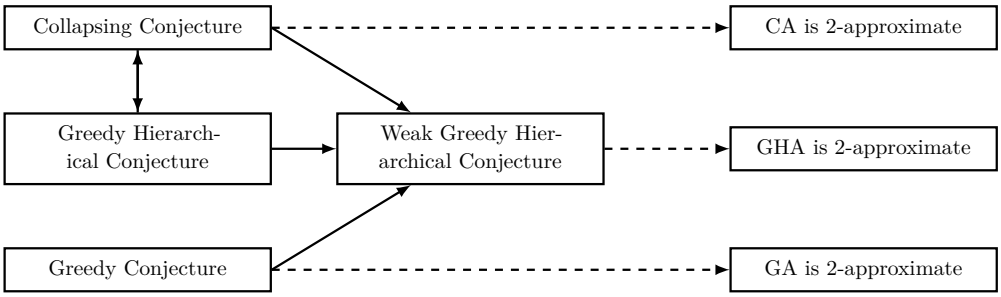


Figure 10 Relations between the conjectures (left), and the 2-approximate algorithms they imply (right). Collapsing and Greedy Hierarchical Conjectures are equivalent. They imply the weak version of the Greedy Hierarchical Conjecture, which also follows from the standard Greedy Conjecture. Each conjecture implies that the corresponding algorithm finds a 2-approximate solution for SCS.

We are now ready to state our second conjecture: the results of the Collapsing Algorithm and Greedy Hierarchical Algorithm coincide!

▶ **Greedy Hierarchical Conjecture.** *For any set of strings $\mathcal{S}$ and any Eulerian solution D,*

$$CA(\mathcal{S}, D \sqcup D) = GHA(\mathcal{S})\,.$$

While the Greedy Hierarchical Conjecture implies that GHA finds a 2-approximate solution, we separately state this weak version of the conjecture.

▶ **Weak Greedy Hierarchical Conjecture.** *GHA is a factor 2 approximation algorithm for the Shortest Common Superstring problem.*

5 Relations between the Conjectures

In this section we prove some relations between the Collapsing and Greedy conjectures. Namely, in Section 5.1 we prove the equivalence of Collapsing and Greedy Hierarchical conjectures. In Section 5.2 we prove that the standard Greedy Conjecture implies Weak Hierarchical Greedy Conjecutre (which is sufficient for a simple 2-approximate greedy algorithm for SCS). Finally, it is easy to see that Greedy Hierarchical Conjecture implies its weak version: indeed, if every doubled solution results in the solution obtained by GHA, then GHA does not exceed twice the optimal superstring length. In Figure 10, we show the proven relations between the conjectures, together with 2-approximate algorithm which follow from each of the conjecture.

5.1 Equivalence of Collapsing and Greedy Hierarchical Conjectures

In this section we prove the equivalence of Collapsing Conjecture and Greedy Hierarchical Conjecture. Recall that Collapsing Conjecture claims that for any pair of Eulerian solutions D_1 and D_2 for the input strings $\mathcal{S}$, we have

$$CA(\mathcal{S}, D_1 \sqcup D_1) = CA(\mathcal{S}, D_2 \sqcup D_2)\,.$$

The Greedy Hierarchical Solution extends this statement to

$$CA(\mathcal{S}, D_1 \sqcup D_1) = GHA(\mathcal{S})\,.$$

Greedy Hierarchical Conjecture trivially implies Collapsing conjecture, and in order to prove their equivalence, it suffices to show that the collapsing procedure applied to the doubled GHA solution results in the GHA solution:

$$CA(\mathcal{S}, GHA(\mathcal{S}) \sqcup GHA(\mathcal{S})) = GHA(\mathcal{S}) .$$

▶ **Theorem 1.** *For any set of strings $\mathcal{S}$,*

$$CA(\mathcal{S}, GHA(\mathcal{S}) \sqcup GHA(\mathcal{S})) = GHA(\mathcal{S}) .$$

Proof. Let us denote two copies of the $GHA(\mathcal{S})$ solution by B and R, which stand for a blue-copy and a red-copy. We will prove the theorem statement by showing that $CA(R \sqcup B)$ collapses all arcs of B and keeps untouched the arcs of R, as this implies that $CA(R \sqcup B) = R = GHA(\mathcal{S})$. For this, without loss of generality assume that the Collapsing Algorithm collapses blue arcs first, that is, if for a vertex v, CA can collapse a blue pair of arcs $(\text{pref}(v), v), (v, \text{suff}(v))$, it does so. Recall that CA processes vertices in the descending order of levels (Algorithm 3, line 1). We will prove that before processing level l, all the arcs above it (i.e., the arcs with at least one vertex at a level $> l$) do satisfy the desired property: all blue arcs are collapsed, and all red arcs are untouched.

The base case trivially holds for $l := \max\{|s| : s \in \mathcal{S}\}$, since the set of arcs above the level l is empty. Assume the claim is true for the level $k > 0$, and let us prove the claim for the level $k - 1$. Note that regardless of the number of collapse operations applied to B, B remains a set of walks: indeed, the collapse procedure keeps the balance of incoming and outgoing arcs for each vertex. By the induction hypothesis all the blue arcs above the level k are collapsed, so we have that if for a vertex v at level k there is an arc $(\text{pref}(v), v)$ in B, then there is also an arc $(v, \text{suff}(v))$ in B, and vice versa. Recall that CA collapses blue arcs when possible, and since every vertex has the same number of blue incoming and outgoing arcs, all pairs collapsed at the level k are monotone.

Now let us show that no red pair can be collapsed. Indeed, if for some vertex v at level k there is a red pair $(\text{pref}(v), v)$, $(v, \text{suff}(v))$, then by construction of GHA v is either in $\mathcal{S}$ or is the last chance of the corresponding component $\mathcal{C} \ni v$ to be connected to the remaining arcs in R (note that the first case is a subcase of the second one, as then $\mathcal{C}$ contains only one vertex). It follows that if CA collapses such a pair of arcs, then v has no blue arcs (as they have been collapsed before the red arcs), and all other vertices in the component $\mathcal{C}$ at the level k collapsed all arcs (since v is the last vertex in $\mathcal{C}$ in lexicographic order). Therefore, this pair is also the last chance of $\mathcal{C}$ to be connected to the rest of the arcs in R, thus, CA cannot collapse it.

It remains to show that all blue pairs at level k are collapsed. This trivially holds because no red pair is collapsed, and, thus, the connectivity of $R \sqcup B$ is maintained by R. This finishes the proof. ◀

5.2 Greedy Implies Greedy Hierarchical

Consider a permutation of the input strings. We say that it is a *valid greedy permutation* if it can be constructed by the Greedy Algorithm: there exist $n - 1$ merges of the n input strings that lead to this permutation such that at every step the two merged strings have the largest overlap. We will prove that GHA always returns a solution which corresponds to a greedy permutation of the input string. That is, while the standard Greedy Algorithm does not determine how to break ties, the Greedy Hierarchical Algorithm is a specific instantiation of the Greedy Algorithm with some tie-breaking rule.

We will use the following simple property of solutions constructed by the GHA algorithm.

▷ Claim 2. Let D be an Eulerian solution constrtucted by GHA. Then D has a "zig-zag" form as in (1).

Proof. First we prove that D is normalized, that is, any application of the collapsing procedure of Algorithm 2 to D will violate the property of Eulerian solution. Indeed, Algorithm 2 can only collapse pairs of arcs of the form $(\text{pref}(s), s), (s, \text{suff}(s))$. The Greedy Hierarchical Algorithm adds such pairs to its solution in two cases: (i) s is an input string (line 2 of Algorithm 4); (ii) s is the the lexicographically largest among the shortest strings in its Eulerian component (line 11 of Algorithm 4). Now note that in the former case, the collapsing procedure applied to s would violate the property that D must contain all input strings, and in the latter case, the collapsing procedure would violate the connectivity property of D.

We finish the proof by showing that every normalized solution is of the form (1). Let $\pi = (s_1, \ldots, s_n)$ be the permutation of the input strings corresponding to a normalized Eulerian solution D. Let us follow the arcs of D in the order of the permutation π, and let P be the set of arcs between the input strings s_i and s_{i+1}. We will prove that P is the union of the sets of arcs of the paths $s_i \to \text{overlap}(s_i, s_{i+1})$ and $\text{overlap}(s_i, s_{i+1}) \to s_{i+1}$. If P contains a pair of consecutive up- and down-arcs, that is, there exists a pair of arcs $(\text{pref}(s), s), (s, \text{suff}(s))$ in P, then this pair would have been collapsed by Algorithm 2, line 2. Therefore, the path P consists of a number of down-arcs followed by a number of up-arcs. It remains to show that the number of down-arcs in P is $d = |s_i| - |\operatorname{overlap}(s_i, s_{i+1})|$. Note that by the definition of $\text{overlap}(\cdot, \cdot)$, the number of down-arcs in P is at least d. On the other hand, if the number of down-arcs in P is strictly greater than d, then both the down-path and up-path in P contain the vertex $\text{overlap}(s_i, s_{i+1})$. This implies that the pair of arcs $(\text{pref}(s), s), (s, \text{suff}(s))$ for $s = \text{overlap}(s_i, s_{i+1})$ would have been collapsed by Algorithm 2, line 2, as it does not violate the connectivity of the solution. Therefore, the number of down-arcs in P is exactly d, which implies that P is the path $s_i \to \text{overlap}(s_i, s_{i+1})$ followed by the path $\text{overlap}(s_i, s_{i+1}) \to s_{i+1}$. ◁

▶ **Theorem 3.** *Every permutation $\pi = (s_1, \ldots, s_n)$ of the input strings constructed by GHA is a valid greedy permutation.*

Proof. Consider the following algorithm A: it starts with the sequence $(s_1, \ldots, s_n)$ obtained by GHA, and at every step it merges two neighboring strings in this sequence that have the largest overlap. It is a greedy algorithm, but instead of considering all pairwise overlaps, it only considers overlaps of neighboring strings in the sequence. Of course, in the end, this algorithm constructs exactly the permutation π. To show that π is a valid greedy permutation, we show that at every iteration of A no two strings have longer overlap than the two strings merged by A.

Consider, for the sake of contradiction, the first iteration when the algorithm A merges some pair of neighboring strings with overlap of length k whereas there are non-neighboring strings p and q with $v = \text{overlap}(p, q)$, $|v| > k$. At this point, p is a merger of input strings $s_a, s_{a+1}, \ldots, s_b$ and q is a merger of input strings $s_c, s_{c+1}, \ldots, s_d$. Then, from the assumption that no input string contains another input string, we have that $v = \text{overlap}(p, q) = \text{overlap}(s_b, s_c)$. Since the algorithm A merges neighboring strings in the decreasing order of overlap lengths, we have that $|\operatorname{overlap}(s_b, s_{b+1})| \le k < |v|$ and $|\operatorname{overlap}(s_{c-1}, c_c)| \le k < |v|$.[2]

Now we consider the Eulerian solution D constructed by GHA in the hierarchical graph. By Claim 2, D has a "zig-zag" form, thus, it contains all arcs from the path $s_b \to \text{overlap}(s_b, s_{b+1}) \to s_{b+1}$, and all arcs from the path $s_{c-1} \to \text{overlap}(s_{c-1}, s_c) \to s_c$.

[2] In the case when s_b is the last string in the solution (or s_c is the first string in the solution) we think of it being followed by ε, and $|\operatorname{overlap}(s_b, \varepsilon)| = 0 < |v|$ still holds.

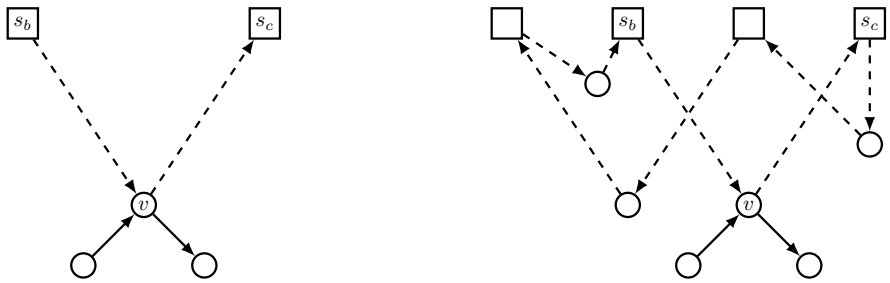


Figure 11 (a) In the Eulerian solution the node $v = \text{overlap}(s_b, s_c)$ has a pair of lower arcs. (b) For this reason, above v, there is an Eulerian component.

Recall that $v = \text{overlap}(s_b, s_c)$, and that $|\text{overlap}(s_b, s_{b+1})| < |v|$ and $|\text{overlap}(s_{c-1}, s_c)| < |v|$. In particular, the paths $s_b \to \text{overlap}(s_b, s_{b+1})$ and $\text{overlap}(s_{c-1}, s_c) \to s_c$ pass through the vertex v, which implies that the vertex v in the solution D has at least one incoming arc from the previous level and at least one outgoing arc to the previous level (see Figure 11(a)). Such a pair of arcs in the Eulerian solution D constructed by GHA may only occur when v is the last chance of its connected component to be connected to the rest of the solution (see line 11 of Algorithm 4). This, in turn, implies that right before the pair of arcs $(\text{pref}(v), v)$ and $(v, \text{suff}(v))$ was added to the Eulerian solution, there was an Eulerian component where v was the lexicographically largest among all shortest nodes. This component is shown schematically in Figure 11(b). All overlap-nodes (the nodes which are equal to $\text{overlap}(s_i, s_{i+1})$) of this component lie on levels $\geqslant k$. Note that the pair $(\text{pref}(v), v)$ and $(v, \text{suff}(v))$ is added to the solution by GHA exactly once (line 11 of Algorithm 4). Therefore, any path following the arcs of D, after going through the arc $(\text{pref}(v), v)$ must traverse the overlying component containing s_b and s_c (as otherwise the path could not reach the overlying component). In turn, this implies that after considering all overlaps of length $|v| > k$, s_b and s_c are already merged into one string, so they cannot be merged at this stage. ◀

Theorem 3 has two immediate corollaries.

▶ **Corollary 4.** *The Greedy Conjecture implies the Weak Greedy Hierarchical Conjecture: if the Greedy Algorithm is 2-approximate, then so is the Greedy Hierarchical Algorithm.*

Since every valid greedy permutation is a 3.5-approximation to the Shortest Common Superstring problem [15], we have the following corollary.

▶ **Corollary 5.** *GHA is a factor* 3.5 *approximation algorithm for the Shortest Common Superstring problem.*

6 Further Directions and Open Problems

The most immediate open problems are to prove the Collapsing Conjecture or the Weak Greedy Hierarchical Conjecture.

6.1 Applications of Hierarchical Graphs

It would also be interesting to find other applications of the hierarchical graphs. We list two such potential applications below.

Exact algorithms. Can one use hierarchical graphs to solve SCS exactly in time $(2-\varepsilon)^n$? It was shown in Section 1 that the SCS problem is a special case of the Traveling Salesman Problem. The best known exact algorithms for Traveling Salesman run in time $2^n \operatorname{poly}(|\,\text{input}\,|)$ [2, 14, 17, 16, 1]. These algorithms stay the best known for the SCS problem as well. The hierarchical graphs were introduced [13] for an algorithm solving SCS on strings of length at most r in time $(2-\varepsilon)^n$ (where ε depends only on r). Can one use the hierarchical graph to solve exactly the general case of SCS in time $(2-\varepsilon)^n$ for a constant ε?

Genome assembly. The hierarchical graph in a sense generalizes de Bruijn graph. The latter one is heavily used in genome assembly [24]. Can one adopt the hierarchical graph for this task? For this, one would need to come up with a compact representation of the graph (as datasets in genome assembly are massive) as well as with a way of handling errors in the input data. Cazaux and Rivals [6] propose a linear-space counterpart of the hierarchical graph.

6.2 Optimal Cycle Covers

A superstring corresponds to a Hamiltonian path in the overlap graph, thus, a minimum-weight cycle cover gives a natural lower bound on its length. The Greedy Conjecture claims that a greedy solution never exceeds twice the length of an optimal solution. It is also believed (see, e.g., [35, 19]) that the greedy solution does not exceed the length of an optimal solution plus the length of an optimal cycle cover. This has interesting counterparts in the hierarchical graphs.

- Note that an optimal cycle cover in the overlap graph can be constructed by a straightforward greedy algorithm: keep taking heavy edges till the cycle cover is constructed. The proof of correctness of this algorithm relies on the Monge inequality. Interestingly, to construct an optimal cycle cover in the hierarchical graph, it suffices to invoke the Greedy Hierarchical Algorithm with lines 7–11 commented out! In a sense, the Monge inequality is satisfied in the hierarchical graph automatically as it contains more information about input strings than just its pairwise overlaps.
- As discussed in Section A, for strings of length 3 even a more general fact than Collapsing Conjecture holds: it suffices to have double edges adjacent to input strings. One simple way to force a particular solution to satisfy this property is to double every edge of it. At the same time, adding a shortest cycle cover to it is guaranteed to be as good.
- Hence, the more general version of the Collapsing Conjecture is the following: take any solution, add any cycle cover to it, and collapse; the result is always the same. We tested this stronger conjecture and did not find any counter-examples.

References

1 Eric Bax and Joel Franklin. A Finite-Difference Sieve to Count Paths and Cycles by Length. *Inf. Process. Lett.*, 60:171–176, 1996.

2 Richard Bellman. Dynamic Programming Treatment of the Travelling Salesman Problem. *J. ACM*, 9:61–63, 1962.

3 Avrim Blum, Tao Jiang, Ming Li, John Tromp, and Mihalis Yannakakis. Linear approximation of shortest superstrings. In *STOC 1991*, pages 328–336. ACM, 1991.

4 Bastien Cazaux, Samuel Juhel, and Eric Rivals. Practical lower and upper bounds for the Shortest Linear Superstring. In *SEA 2018*, volume 103, pages 18:1–18:14. LIPIcs, 2018.

5 Bastien Cazaux and Eric Rivals. A linear time algorithm for Shortest Cyclic Cover of Strings. *J. Discrete Algorithms*, 37:56–67, 2016.

6 Bastien Cazaux and Eric Rivals. Hierarchical Overlap Graph. *arXiv preprint*, 2018. `arXiv:1802.04632`.

7 Bastien Cazaux and Eric Rivals. Relationship between superstring and compression measures: New insights on the greedy conjecture. *Discrete Appl. Math.*, 245:59–64, 2018.

8 John Gallant. *String compression algorithms*. PhD thesis, Princeton, 1982.

9 John Gallant, David Maier, and James A. Storer. On finding minimal length superstrings. *J. Comput. Syst. Sci.*, 20(1):50–58, 1980.

10 Theodoros P. Gevezes and Leonidas S. Pitsoulis. *The shortest superstring problem*, pages 189–227. Springer, 2014.

11 Collapsing Superstring Conjecture. GitHub repository. `https://github.com/alexanderskulikov/greedy-superstring-conjecture`, 2018.

12 Alexander Golovnev, Alexander S. Kulikov, and Ivan Mihajlin. Approximating shortest superstring problem using de Bruijn graphs. In *CPM 2013*, pages 120–129. Springer, 2013.

13 Alexander Golovnev, Alexander S. Kulikov, and Ivan Mihajlin. Solving SCS for bounded length strings in fewer than 2^n steps. *Inf. Process. Lett.*, 114(8):421–425, 2014.

14 Michael Held and Richard M. Karp. The Traveling-Salesman Problem and Minimum Spanning Trees. *Math. Program.*, 1:6–25, 1971.

15 Haim Kaplan and Nira Shafrir. The greedy algorithm for shortest superstrings. *Inf. Process. Lett.*, 93(1):13–17, 2005.

16 Richard M. Karp. Dynamic Programming Meets the Principle of Inclusion and Exclusion. *Oper. Res. Lett.*, 1(2):49–51, 1982.

17 Samuel Kohn, Allan Gottlieb, and Meryle Kohn. A Generating Function Approach to the Traveling Salesman Problem. In *ACN 1977*, pages 294–300, 1977.

18 Alexander S. Kulikov, Sergey Savinov, and Evgeniy Sluzhaev. Greedy conjecture for strings of length 4. In *CPM 2015*, pages 307–315. Springer, 2015.

19 Uli Laube and Maik Weinard. Conditional inequalities and the shortest common superstring problem. *Int. J. Found. Comput. Sci.*, 16(06):1219–1230, 2005.

20 Gaspard Monge. Mémoire sur la théorie des déblais et des remblais. *Histoire de l'Académie Royale des Sciences de Paris*, 1781.

21 Marcin Mucha. A tutorial on shortest superstring approximation, 2007.

22 Marcin Mucha. Lyndon Words and Short Superstrings. In *SODA 2013*, pages 958–972. SIAM, 2013.

23 Katarzyna Paluch. Better approximation algorithms for maximum asymmetric traveling salesman and shortest superstring. *arXiv preprint*, 2014. `arXiv:1401.3670`.

24 Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci. U.S.A.*, 98(17):9748–9753, 2001.

25 Eric Rivals and Bastien Cazaux. Superstrings with multiplicities. In *CPM 2018*, volume 105, pages 21:1–21:16, 2018.

26 Heidi J. Romero, Carlos A. Brizuela, and Andrei Tchernykh. An experimental comparison of two approximation algorithms for the common superstring problem. In *ENC 2004*, pages 27–34. IEEE, 2004.

27 Sartaj Sahni and Teofilo Gonzalez. P-Complete Approximation Problems. *J. ACM*, 23:555–565, 1976.

28 James A. Storer. *Data compression: methods and theory*. Computer Science Press, Inc., 1987.

29 Ola Svensson, Jakub Tarnawski, and László A. Végh. A constant-factor approximation algorithm for the asymmetric traveling salesman problem. In *STOC 2018*, pages 204–213. ACM, 2018.

30 Jorma Tarhio and Esko Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. *Theor. Comput. Sci.*, 57(1):131–145, 1988.

31 Jonathan S. Turner. Approximation algorithms for the shortest common superstring problem. *Inf. Comput.*, 83(1):1–20, 1989.

32 Esko Ukkonen. A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 5(1-4):313–323, 1990.

33 Michael S. Waterman. *Introduction to computational biology: maps, sequences and genomes*. CRC Press, 1995.

34 Collapsing Superstring Conjecture. Webpage. `http://compsciclub.ru/scs/`, 2018.

35 Maik Weinard and Georg Schnitger. On the greedy superstring conjecture. *SIAM J. Discrete Math.*, 20(2):502–522, 2006.

A Proof of Collapsing Conjecture for Strings of Length 3

In this section, we show that the Collapsing Conjecture holds for the special case when input strings have length at most three. Remarkably, this follows from a more general theorem stated below.

▶ **Theorem 6.** *Let $\mathcal{S}$ contain strings of length at most 3 and let L be an Eulerian solution that for each $s \in \mathcal{S}$ contains at least two copies of arcs* $(\mathrm{pref}(s), s)$ *and* $(s, \mathrm{suff}(s))$*. Then* $CA(\mathcal{S}, L) = GHA(\mathcal{S})$.

It is not difficult to see that the theorem indeed implies the Collapsing Conjecture: clearly, $L = D \sqcup D$, where D is an Eulaerian solution, satisfies the condition. Moreover, this also works for $L = D_1 \sqcup D_2$, where D_1, D_2 are arbitrary Eulerian solutions, and for $L = D \sqcup CC$, where CC is a cycle cover, i.e., a set of cycles that go through all input strings. The main difference between an Eulerian solution and a cycle cover is that the later is not required to be connected. For this reason, any Eulerian solution is also a cycle cover (but not vice versa) and hence an *optimal* cycle cover is definitely not longer than an optimal Eulerian solution: $OPT \leq OPTCC$. Hence, Theorem 6 says that the result of GHA is not just no longer than $2 \cdot OPT$, but even no longer than $OPT + OPTCC$.

Before proving Theorem 6, we introduce some notation and prove two auxiliary results. Recall that the Collapsing Algorithm processes the nodes level by level. Denote by L_i an intermediate Eulerian solution right before is starts collapsing the nodes at level i (that is, in L_i all the nodes at levels $> i$ are already collapsed). For an arbitrary Eulerian solution U, by $\mathrm{above}(U, i)$ denote the part of U that lies above the level i: $\mathrm{above}(U, i) = \{(u, v) \in U \colon |u|, |v| \geq i\}$. We show that $\mathrm{above}(D, i) = \mathrm{above}(L_i, i)$ for every i. This is enough since then

$$CA(\mathcal{S}, L) = \mathrm{above}(L_0, 0) = \mathrm{above}(D, 0) = GHA(\mathcal{S})\,.$$

▶ **Lemma 7.** *Let w be a walk from u to v in an Eulerian solution with all its nodes at levels $\leq k$. Consider a single collapsing step for a node t that is either an* intermediate *node of p at level k or is a node at level $< k$ that do not belong to p. Then w is still a walk from u to v in the resulting solution.*

Proof. Indeed, if t does not belong to w, then collapsing it does not change w at all. Otherwise t is an intermediate node of w at level k. Since w does not have any node above level k, w goes through $(\mathrm{pref}(t), t), (t, \mathrm{suff}(t))$. Clearly, collapsing t keeps w a walk. ◀

▶ **Lemma 8.** *Let v be a node in L_2 at level $1 \leq l \leq 2$ (i.e., $l = |v|$). Then there is a walk from ε to v and a walk from ε to v in L_l that does not contain nodes at level 3.*

Proof. We start by proving that there is a walk from v to ε for $|v| = 2$ (the existence of a walk from ε to v is proved in a similar fashion).

Consider a walk w from v to ε in L (there is such a walk as L is an Eulerian solution). All repeated nodes in w may be removed, so one may assume that w passes through its nodes at level 3 exactly once. Then, it is sufficient to show that each such node is collapsed.

Consider a node s of w at level 3 and a pair of arcs $(\mathrm{pref}(s), s), (s, \mathrm{suff}(s)) \in w$ going through it. If s is not at input string (i.e., $s \notin \mathcal{S}$), then CA collapses this pair of arcs and this does not disconnect w. On the other hand, if s is an input string ($s \in \mathcal{S}$), then there are two copies of $(\mathrm{pref}(s), s), (s, \mathrm{suff}(s))$ in L. At least one copy of this pair is collapsed in L and therefore belongs to L_2.

The statement for v with $|v| = 1$ follows from Lemma 7. ◀

Proof of Theorem 6. As discussed above, it suffices to prove that $\mathrm{above}(D, i) = \mathrm{above}(L_i, i)$ for every $i = 2, 1, 0$.

Level $i = 2$. The base case $i = 2$ is straightforward: clearly, the Collapsing Algorithm leaves exactly one copy of arcs $(\mathrm{pref}(s), s)$ and $(s, \mathrm{suff}(s))$ for every $s \in \mathcal{S}$ and fully collapses all other nodes at level 3. Then, $\mathrm{above}(L_2, 2) = \mathrm{above}(D, 2)$ as $(\mathrm{pref}(s), s), (s, \mathrm{suff}(s))$ for $s \in \mathcal{S}$ are the only edges between levels 2 and 3 in D.

Level $i = 1$. Note that $\mathrm{above}(L_2, 2) \subseteq L_2$ and L_2 is an Eulerian cycle. Hence, $\mathrm{above}(L_2, 2)$ is a collection of walks. Consider such a walk w and consider two cases.

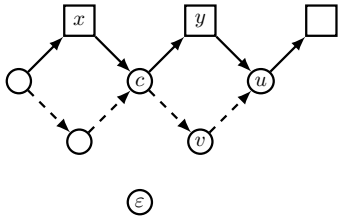
- w is a closed walk. Let v be the lexicographically largest node of w at level 2. What we want to show is that in L_1 this closed walk w is connected to the rest of L_1 through a pair of arcs $(\mathrm{pref}(v), v)$ $(v, \mathrm{suff}(v))$ only.
  By Lemma 8, there is a path from v to ε in L_2 and hence $(v, \mathrm{suff}(v)) \in L_2$; similarly, $(\mathrm{pref}(v), v) \in L_2$. Since v is lexicographically largest at level 2 in w, when CA starts processing the node v, all other nodes at level 2 in w are fully collapsed, i.e., for any such node u, $(\mathrm{pref}(u), u) \notin L_1$ and $(u, \mathrm{suff}(u)) \notin L_1$. Moreover, CA does not collapse the pair of arcs $(\mathrm{pref}(v), v), (v, \mathrm{suff}(v))$ as this would disconnect w from the rest of the solution.
- w is not closed. Denote by v_1 and v_k its first and last nodes. All other nodes of w in $\mathrm{above}(L_2, 2)$ are balanced. What we want to show is that in L_1 the only edges between levels 1 and 2 that connect w to the rest of the solution are $(\mathrm{pref}(v_1), v_1)$ and $(v_k, \mathrm{suff}(v_k))$.
  We prove this for v_k (for v_1 is it shown similarly). By Lemma 8, there is a path from v_k to ε in L_2 and hence $(v_k, \mathrm{suff}(v_k)) \in L_2$. The algorithm CA always works with an Eulerian solution and hence every node is balanced at every stage (i.e., its in-degree is equal to its out-degree). This means that $(v_k, \mathrm{suff}(v_k)) \in L_1$ and that all intermediate nodes of w are not connected to level 1 nodes in L_1.

Level $i = 0$. Note that $\mathrm{above}(L_1, 1)$ is a collection of walks. The case of a non-closed walk in this case is easy as it must be connected to ε directly. For this reason, we focus on a closed walk w in $\mathrm{above}(L_1, 1)$.

We show that for every node v of w with $|v| = 1$, L_1 contains arcs (ε, v) and (v, ε) (recall that for $|v| = 1$, $\mathrm{pref}(v) = \mathrm{suff}(v) = \varepsilon$). This suffices as then CA (when processing level one nodes) collapses all nodes of w at level 1 except for the lexicographically largest one, and this is exactly how w is connected to ε in D_0. Below, we show that $(\varepsilon, v) \in L_1$. It then follows that $(v, \varepsilon) \in L_1$ (as L_1 must be Eulerian).

Lemma 8 guarantees that L_1 contains a path from v to ε that does not contain nodes at level 3. If the first arc of this path goes down to ε, then there is nothing to prove. Hence, consider a case when the first arc goes up to a node u (and hence $v = \mathrm{pref}(u)$). The next arc then must go down to $\mathrm{suff}(u)$. Hence, $(\mathrm{pref}(u), u), (u, \mathrm{suff}(u)) \in D_1$. This

may happen in two cases only: either u is an input string (i.e., $u \in \mathcal{S}$) or u is the last chance of its component to be connected to the rest of the solution (i.e., exactly for this reason GHA added these two edges to the solution). The former case is straightforward: then there were at least two copies of the arcs $(\mathrm{pref}(u), u), (u, \mathrm{suff}(u))$ and CA collapsed at least one copy. Let us then focus on the latter case.
Let $x, y \in \mathcal{S}$ be such that $u = \mathrm{suff}(y)$ and $c := \mathrm{suff}(x) = \mathrm{pref}(y)$, see the picture below (solid arcs belong to L, dashed arc belong to L_2).



Note that

$$v = \mathrm{pref}(u) = \mathrm{pref}(\mathrm{suff}(y)) = \mathrm{suff}(\mathrm{pref}(y)) = \mathrm{suff}(c)\,.$$

Hence, $(c, v), (v, u) \in L_2$ (resulting from collapsing at least one pair of arcs $(c, y), (y, u) \in L$). L_2 also contains a pair of arcs $(\mathrm{pref}(x), \mathrm{suff}(\mathrm{pref}(x))), (\mathrm{suff}(\mathrm{pref}(x)), c)$. When processing the node c, CA collapses the pair of arcs $(\mathrm{pref}(c), c), (c, v)$ as there is an arc (v, u). Hence, $(\varepsilon, v) \in L_1$, as required. (It may be the case that $x = y$. Then $x = \mathtt{aaa}$, $v = \{a\}$. Then the first pair of arcs of the considered path is $\mathtt{a} \to \mathtt{aa} \to \mathtt{a}$ and one may just drop them.)

As a final remark, note that if a walk $w \in \mathrm{above}(L_i, i)$ is connected to the rest of a solution through some a of arcs $(\mathrm{pref}(v), v), (u, \mathrm{suff}(u))$ (v and u may coincide), then any other balanced node in w at level i can be fully collapsed, as every such collapse, thanks to Lemma 7, does not disconnect w or any other walk from $\mathrm{above}(L_i, i)$ from the rest of the solution. ◀

B Greedy Hierarchical Algorithm and Special Cases of SCS

B.1 Strings of Length 2

Gallant et al. [9] show that SCS on strings of length 3 is **NP**-hard, but SCS on strings of length at most 2 is solvable in polynomial time. In this section we show that GHA finds an optimal solution in this case as well. We note that the standard Greedy Algorithm does not necessarily find an optimal solution in this case. For example, if $\mathcal{S} = \{\mathtt{ab}, \mathtt{ba}, \mathtt{bb}\}$, the Greedy Algorithm may first merge `ab` and `ba`, which would lead to a suboptimal solution `ababb` (recall also Figure 6).

First, we can assume that all input strings from $\mathcal{S}$ have length exactly 2. Indeed, since we assume that no input string is a substring of another input string, all strings of length 1 are unique symbols which do not appear in other strings. Take any such s_i of length 1. The optimal superstring length for $\mathcal{S}$ is k if and only if the optimal superstring length for $\mathcal{S} \setminus \{s_i\}$ is $k-1$. The Greedy Hierarchical Algorithm has the same behavior: In Step 2, GHA will include the arcs $(\varepsilon, s_i), (s_i, \varepsilon)$ in the solution, and it will never touch the vertex s_i again (because it is balanced and connected to ε). Thus, s_i adds 1 to the length of the Greedy Hierarchical Superstring as well. By the same reasoning, we can assume that each string of length two is primitive, i.e., contains two distinct symbols.

When considering primitive strings $\mathcal{S} = \{s_1, \ldots, s_n\}$ of length exactly 2, it is convenient to introduce the following directed graph $G = (V, E)$, where V contains a vertex for every symbol which appears in strings from $\mathcal{S}$. The graph has $|E| = n$ arcs corresponding to n input strings: for every string $s_i = ab$, there is an arc from a to b. It is known [9] that the length of an optimal superstring in this case is $n + k$ where k is the minimum number such that E can be decomposed into k directed paths, or, equivalently:

▶ **Proposition 9** ([9]). *Let G be the graph defined above, and let $G_1 = (V_1, E_1), \ldots, G_c = (V_c, E_c)$ be its weakly connected components. Then the length of an optimal superstring is*

$$n + \sum_{i=1}^{c} \max\left(1, \sum_{v \in V_i} \frac{|\operatorname{indegree}(v) - \operatorname{outdegree}(v)|}{2}\right). \tag{2}$$

We will now show that in this case, GHA finds an optimal solution.

▶ **Lemma 10.** *Let $\mathcal{S} = \{s_1, \ldots, s_n\}$ be a set of strings of length at most 2, and let s be an optimal superstring for $\mathcal{S}$. Then $GHA(\mathcal{S})$ returns a superstring of length $|s|$.*

Proof. We showed above that it suffices to consider the case of n primitive strings $\{s_1, \ldots, s_n\}$ of length exactly 2. For $1 \leqslant i \leqslant n$, let $s_i = a_i b_i$, where $a_i \neq b_i$. Consider the partial greedy hierarchical solution D after the Step 2 of the GHA algorithm: $D = \{(a_i, a_i b_i), (a_i b_i, b_i) : 1 \leqslant i \leqslant n\}$. (We abuse notation by identifying the set of arcs D with the graph induced by D.) This partial solution has n up-arcs, so its current weight is n.

Note that by the definition of the graph G above, G contains an arc (a, b) if and only if D has the arcs (a, ab) and (ab, b) of the graph HG. Thus, the indegree (outdegree) of a vertex a in G equals the indegree (outdegree) of the vertex a in the partial solution D. Also, two vertices a and b of G belong to one weakly connected component in G if and only if they belong to one weakly connected component in D. Therefore, the expression (2) in G has the same value in the partial solution graph D. (Indeed, the vertices of D corresponding to strings of length 2 are balanced and do not form weakly connected components.)

Now we proceed to Steps 3–11 of GHA. GHA will go through all strings of length 1, and add $|\operatorname{indegree}(v) - \operatorname{outdegree}(v)|$ arcs for each unbalanced vertex v. The Steps 8–11 ensure that each weakly connected component adds at least a pair of arcs. Since exactly a half of added arcs are up-arcs, we have increased the weight of the partial solution D by

$$\sum_{i=1}^{c} \max\left(1, \sum_{v \in V_i} \frac{|\operatorname{indegree}(v) - \operatorname{outdegree}(v)|}{2}\right). \qquad \blacktriangleleft$$

B.2 Spectrum of a String

By a k-spectrum of a string s (of length at least k) we mean a set of all substrings of s of length k. Pevzner et al. [24] give a polynomial time exact algorithm for the case when the input strings form a spectrum of an unknown string. We show that GHA also finds an optimal solution in this case.

▶ **Lemma 11.** *Let $\mathcal{S} = \{s_1, \ldots, s_n\}$ be a k-spectrum of an unknown string s. Then $GHA(\mathcal{S})$ returns a superstring of length at most $|s|$.*

Proof. Since s has n distinct substrings of length k, $|s| \geqslant n + k - 1$. We will show that GHA finds a superstring of length $n + k - 1$. After Step 2 of GHA, the partial solution $D = \{(\operatorname{pref}(s), s), (s, \operatorname{suff}(s)) : s \in \mathcal{S}\}$. In particular, D is of weight n. For $1 \leqslant i \leqslant k - 1$,

let u_i be the first i symbols of s, and let v_i be the last i symbols of s. Note that u_{k-1} and v_{k-1} are the only unbalanced vertices of the partial solution D after Step 2: all other strings of length $k-1$ appear equal number of times as prefixes and suffixes of strings from $\mathcal{S}$. Therefore, while processing the level $\ell = k-1$, GHA will add one arc to each of the vertices u_{k-1} and v_{k-1}, and will not add arcs to other strings of length $k-1$.

In general, while processing the level $\ell = i$, GHA adds one up-arc to u_i and one down-arc to v_i. In order to show this, we consider two cases. If $u_i \neq v_i$, then u_i has an incoming arc from the previous step and does not have outgoing arcs, therefore GHA adds an up-arc to u_i in Step 6. Similarly, GHA adds a down-arc from v_i. Note that there are no other strings of length $i < k-1$ in the partial solution, so the algorithm moves to the next level. In the case when $u_i = v_i$, we have that all vertices are balanced, but the string u_i is now the shortest string in this only connected component $\mathcal{C}$ of the graph. Therefore, for $i > 0$ we have $\varepsilon \notin \mathcal{C}$, and GHA adds an up- and down-arc to u_i in Step 11.

We just showed that GHA solution for a k-spectrum of a string has the initial set of arcs $D = \{(\mathrm{pref}(s), s), (s, \mathrm{suff}(s))\colon s \in \mathcal{S}\}$, and also the arcs $\{(u_{i-1}, u_i), (v_i, v_{i-1})\colon 1 \leqslant i \leqslant k-1\}$. Thus, the total number of up-arcs (and the weight of the solution) is $n + k - 1$. ◀

B.3 Tough Dataset

There is a well-known dataset consisting of just three strings where the classical greedy algorithm produces a superstring that is almost twice longer than an optimal one: $s_1 = \mathtt{cc}(\mathtt{ae})^n$, $s_2 = (\mathtt{ea})^{n+1}$, $s_3 = (\mathtt{ae})^n\mathtt{cc}$. Since $\mathrm{overlap}(s_1, s_3) = 2n$, while $\mathrm{overlap}(s_1, s_2) = \mathrm{overlap}(s_2, s_3) = 2n-1$, the greedy algorithm produces a permutation (s_1, s_3, s_2) (or (s_2, s_1, s_3)). I.e., by greedily taking the massive overlap of length $2n$ it loses the possibility to insert s_2 between s_1 and s_3 and to get two overlaps of size $2n-1$. The resulting superstring has length $4n+6$. At the same time, the optimal superstring corresponds to the permutation (s_1, s_2, s_3) and has length $2n+8$.

The algorithm GHA makes a similar mistake on this dataset, see Figure 12. When processing the node $(\mathtt{ea})^n$, GHA does not add two lower arcs to it and misses a chance to connect two components. It is then forced to connect these two components through ε. This example shows that GHA also does not give a better than 2-approximation for SCS.

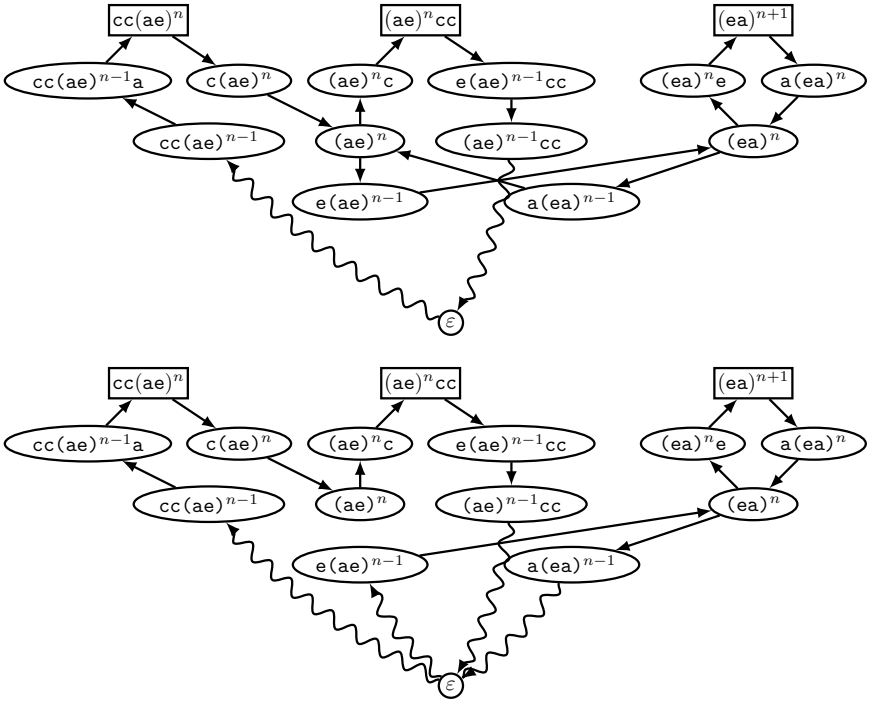


Figure 12 Top: optimal solution for the dataset $\{\mathtt{cc}(\mathtt{ae})^n, (\mathtt{ea})^{n+1}, (\mathtt{ae})^n\mathtt{cc}\}$. Bottom: solution constructed by GHA.