APPORXIMATION HARDNESS OF SHORTEST COMMON SUPERSTRING VARIANTS

Y. WILLIAM YU

Abstract. The shortest common superstring (SCS) problem has been studied at great length because of its connections to the de novo assembly problem in computational genomics. The base problem is APX-complete, but several generalizations of the problem have also been studied. In particular, previous results include that SCS with Negative strings (SCSN) is in Log-APX (though there is no known hardness result) and SCS with Wildcards (SCSW) is Poly-APX-hard. Here, we prove two new hardness results: (1) SCSN is Log-APX-hard (and therefore Log-APX-complete) by a reduction from Minimum Set Cover and (2) SCS with Negative strings and Wildcards (SCSNW) is NP0P-hard by a reduction from Minimum Ones 3SAT.

1. Introduction

Given a set of strings $s_1, \ldots, s_n$, the Shortest Common Superstring optimization problem (SCS) is to minimize $N$ so that there exists a string $S$ of length $N$ such that all $s_i$ are substrings of $S$. SCS and its variants are closely related to the assembly problem in computational genomics [MGMB07]; i.e. piecing together a full genome from small fragments, though redundancy in the genome implies that the correspondence is not perfect. Note that this not to be confused with Shortest Common Supersequence, which deals with subsequences instead of substrings, and can be related to the alignment problem in genomics [RUSI].

SCS was proven in 1994 to be APX-hard from the traveling salesman problem (TSP) using a $2n + 1$ length alphabet, and a 3-approximation algorithm was given [BJL+94]. Since then, a string of algorithmic advances have brought the approximation ratio down to $\frac{57}{23}$ [Muc13, GKM13], and inapproximability results have shown that the minimum is $\frac{333}{332}$ [KS13]. Additionally, Ott showed in 1999 that SCS is APX-hard even when the alphabet is restricted to size 2 by a reduction from TSP with all distances either 1 or 2 [Ott99].

Although base SCS is thus fairly well-characterized as APX-complete, several generalizations have also been studied in the literature (Table 1). In particular, allowing for negative strings (which are not allowed in the
superstrings) and wildcards seem to increase the difficulty of the problem. Shortest Common Superstring with Negative strings (SCSN) can be approximated to within a logarithmic factor $\text{[JL94]}$ using the Group-Merge algorithm $\text{[Li90]}$, but no comparable hardness result has been shown in the literature. Shortest Common Superstring with Wildcards (SCSW) on the other hand is known to be Poly-APX-hard by reduction from minimum chromatic number $\text{[Ma09]}$. Nothing is known about the combination of the two, Shortest Common Superstring with Negative strings and Wildcards (SCSNW).

In this paper, we first briefly review existing reductions for proving APX-hardness and Poly-APX-hardness of SCS and SCSW respectively. Building on insights and strategies from those reductions, we then prove two new hardness results: (1) SCSN is Log-APX-hard and (2) SCSNW is NPOPB-hard.

# 2. Reductions Review

In this section we review reductions for SCS and SCSW. We omit many details, as we are interested only in highlighting some of the gadgets and reduction strategies that we will be using later.

## 2.1. SCS reduction from $O(1)$-degree vertex cover $\text{[Vas05]}$.

Given a set of strings $s_1, \ldots, s_n$, SCS wants to minimize $N$ so that there exists a string $S$ of length $N$ such that all $s_i$ are substrings of $S$. Although the original APX-hardness reduction for SCS was from a variant of the Traveling Salesperson Problem $\text{[BJL+94]}$, we review here (in brief, skipping many details) a more recent reduction from $O(1)$-degree vertex cover $\text{[Vas05]}$, as our new reductions build on several of the ideas.

We start with instance of vertex cover $G = (V, E)$ with $|V| = n$ and $|E| = m$. Let the alphabet $\Sigma = V$ so each vertex $a$ is associated with a single letter $a$. Let an edge $(a, b)$ be represented by strings $ababa$ and $baba$. Suppose $G$ has a vertex set $S$ of size $k$. Assign every edge $(a, b)$ to its
If \( S = \{2, 4\} \), then edges collapse to
- \( 21212 \)
- \( 23232 \)
- \( 24242 \)
- \( 41414 \)
- \( 43434 \)

Then the two vertices 2 and 4 are associated with strings
- \( 2121232324242 \)
- \( 414143434 \)

Which results in a final string
- \( 2121242423232414143434 \)

Figure 1. SCS reduction from \( O(1) \)-degree vertex cover example.

covering vertex (or arbitrarily if both vertices are in \( S \)). Then if \( a \) is the assigned vertex for the edge \((a, b)\), overlap the two strings to get \( ababa \), else overlap the other way to get \( babab \). Then for every \( c \in S \), we can overlap all assigned edge strings by 1 to get \( ca_1ca_1ca_2ca_2c \ldots ca_kca_kc \) of length \( 4k_c + 1 \), where \( k_c \) is the number of edges assigned to \( c \in S \). By concatenating all such strings together, we get a superstring of length \( 4m + k \) (Figure 1).

Conversely, it can be shown that all superstrings for the SCS problem are of length \( 4m + t \), and can be shortened in polynomial time to a string corresponding to a vertex cover. Thus, if we can get a superstring of length \( 4m + k \) for SCS, we can get a vertex cover of size \( \leq k \). Making use of exact bounds from the \( O(1) \)-vertex cover problem, it is possible to show SCS is APX-hard.

We will reuse two of the gadgets later in the SCSN Log-APX-hardness proof:

1. Overlapping strings in two different ways for each edge to select which vertex covers that edge.
2. Creating vertex strings by overlapping edge strings, such that each additional vertex used contributes 1 to the final cost.

2.2. SCSW reduction from minimum chromatic number [Ma09].

Given set of strings \( s_1, \ldots, s_n \) with letters from \( \Sigma \cup \{?\} \) find the shortest string \( S \) with letters from \( \Sigma \) that is a superstring of all \( s_i \), where each ? can match any letter of \( \Sigma \). For genomics, this corresponds to uncertainty in sequencer calls for particular bases in a DNA read.

We start with a minimum chromatic number problem on graph \( G = (V, E) \) with \( V = \{v_1, v_2, \ldots, v_n\} \) and \( E = \{e_1, e_2, \ldots, e_m\} \). Let \( \Sigma = \{A, T, G, C\} \).

For each \( v_i \), let \( t_i \) be a string of length \( m \) such that

\[
t_i[k] = \begin{cases} A, & \text{if } e_k = (v_i, v_j) \text{ and } i < j \\ T, & \text{if } e_k = (v_j, v_i) \text{ and } j < i \\ ?, & \text{otherwise.} \end{cases}
\]

We will reuse two of the gadgets later in the SCSN Log-APX-hardness proof:

(1) Overlapping strings in two different ways for each edge to select which vertex covers that edge.
(2) Creating vertex strings by overlapping edge strings, such that each additional vertex used contributes 1 to the final cost.
Strings in SCSW are
- \( s_1 = XAA???X \)
- \( s_2 = XT?AA?X \)
- \( s_3 = X??T?AX \)
- \( s_4 = X?T?TTX \)

Vertices \( s_1 \) and \( s_3 \) can merge due to independence
- \( XAAT?AX \)

Which results in a final string
- \( XAAT?AXT?AA?X?T?TTX \)

Figure 2. SCSW reduction from minimum chromatic number example.

Then let \( s_i = Xt_iX, \ \forall i \in [1, n] \) where \( X = G_{mn}C_{mn} \), be the SCSW instance. By construction, independent sets can completely overlap with one another. As each color in a coloring corresponds to an independent set, superstrings have length proportional to the minimum chromatic number (exactly \( 2mn + m(2n + 1)k \), where \( k \) is the chromatic number). Any superstring of the SCSW problem can be polynomially shortened to be of the form \( XY_1XY_2X \ldots XY_k \). Reconstructing the independent sets is then just matter of reading off the set edges in each string between the \( X \) border markers.

Unlike the SCS reduction in the last section, this is an L-reduction \cite{Cre97} (or more precisely, after normalizing by \( m(2n+1) \), it is an L-reduction). This is because each new color needed in min chromatic number corresponds to not just a single character, but instead an entire substring \( XY_i \)'s worth.

We will reuse two of the strategies later:

1. Using wildcards to allow collapsing together many input strings into a single section.
2. Forcing each additional color to correspond to a long string so that we have an L-reduction.

While the first strategy is only applicable to our SCSNW proof later, the second is used in both the SCSN and SCSNW reductions in the next section.

3. New hardness results

3.1. SCSN reduction from minimum set cover.

**Theorem 1** (SCSN is Log-APX-hard). Given a set of strings \( s_1, \ldots, s_\eta \) and a set of negative strings \( t_1, \ldots, t_{\text{poly}(\eta)} \), both built from an alphabet \( \Sigma \), optimizing for the shortest string \( T \) that is a superstring of all \( s_i \) but contains no \( t_j \) as a substring is Log-APX-hard.
We will prove this theorem by reduction from min set cover [LY94], but will first need some setup. Our strategy for this reduction will be to use the negative strings to force certain structural conditions. Let the set cover problem be to cover the set of items $S = \{1, \ldots, m\}$ by sets $T_1, \ldots, T_n \in C$, $i \subseteq S$. Let the alphabet $\Sigma = S \cup C \cup \{l_0, l, b, e\}$. We introduce the additional letters $b, e$ (begin and end) to frame the string to remove border effects, and the additional letters $l_0, l$ to force long gaps after certain patterns. For the reduction, let the input positive strings be $\{i \in S\} \cup \{b, e\}$, so we require that each item letter appear at least once and have a particular beginning and end.

**Lemma 2.** For any string $X \in T$, we can disallow arbitrary prefixes and suffixes of bounded length $AXB$ in polynomial time.

**Proof.** The total number of possible strings of the form $AXB$ is $|\Sigma|^{|A|+|B|}$, and listing them all out as negative strings takes $O((|A|+|X|+|B|)\cdot |\Sigma|^{|A|+|B|})$ time, which is polynomial. For ease of notation, we will use “?” as a “wild-card” symbol where applicable.

**Gadget 3** (Frame Gadget). Forces $T = b\ldots e$.

**Design.** Disallow $b$ and $e$?. Then there can be no letters left of $b$ or right of $e$ because then there would be a disallowed substring. For the remainder of this section, unless explicitly noted otherwise, we will not consider $b$ and $e$ valid characters for substrings, since they must be unique and at predetermined locations.

**Lemma 4.** For any string $X \in T$, prefix length $u$, and suffix length $v$, we can force $X$ to extend to a string $AXB$ with $|A| = u, |B| = v$ such that $AXB$ is drawn from a specified set $T(X)$ in polynomial time.

**Proof.** By Lemma 2 we can list all strings of the form $?X?X$ as negative strings in polynomial time. First, we list all strings of the form $?X?$ except those that match some center in $T(X)$ as negative strings. Then we iterate, building single characters onto prefix and suffix until we reach strings of form $AXB$. This takes polynomial time provided $u$ and $v$ as $u$ and $v$ are bounded. Additionally, because of the Frame Gadget, the iteration cannot stop until we reach $AXB$ because otherwise some other character would be the left-most or right-most in $T$.

**Gadget 5** (Item Gadget). Extends the item string “i” to

$$ji\overline{j}(j+1)i(j+1)\ldots \overline{m}\overline{i}T_1\overline{T_2}\overline{\ldots}ji\overline{j},$$

for any choice of rotation $j \in \{1, \ldots, n\}$ for which $i \in j$

**Design.** Extend $?i?$ to $\overline{\tau}\overline{\varepsilon}$ where $\overline{\tau} \in C$. This forces every item to be surrounded on both sides by one of the sets in a triple.
For every $ji$, disallow $??\overline{ji}??$ except
\[(j-1)i(j-1)\overline{ji}?? \quad \text{or} \quad \overline{ji}(j+1)i(j+1) \quad \text{if } i \in \overline{j}\]
\[\overline{ji}(j-1)\overline{ji}(j+1)i(j+1) \quad \text{if } i \notin \overline{j}\]
This forces every triple to be connected on at least one side to its consecutive triple, and buries in the middle triples corresponding to sets that do not cover $i$.

Then, for every string of $p$ triples
\[X = \overline{ji} \ldots (j+p-1 \mod n)i(j+p-1 \mod n),\]
for $p \in \{1, \ldots n\}$, disallow $??X??$ except for
\[??X(j+1)X(j+1)\quad \text{or} \quad \overline{(j-1)i(j-1)}X??\]
This forces these item gadgets to have at least $n+1$ triples. To make sure they do not have more than $n+1$ triples, we just disallow strings with $n+2$ triples. Thus, we have constructed our length $3n + 3$ Item Gadget.

**Gadget 6 (Set Gadget).** Allows a $2 + m(3n + 2)$ penalty to be placed on the string length for each additional set needed in the cover, resulting in an L-reduction.

**Design.** As with the earlier SCS reduction, note that every item can be assigned to a particular set $\overline{j}$ for the cover by rotating the Item Gadget so that it starts and ends with $\overline{j}$. Then, adjacent items assigned to the same set can overlap by 1 character, so for a set $c$ with $k(c) > 0$ assigned items, the set gadget will use up $k(c) \cdot (3n + 2) + 1$ characters. Alone, using the same arguments as in the SCS reduction, this would imply that the superstring uses up $2 + m(3n + 2) + k$ characters, for a set cover of size $k$.

Unfortunately, the above is not an L-reduction as like in the SCSW reduction we need a multiplicative penalty. However, we can achieve that by forcing additional space between adjacent set gadgets. To do this, for every orientation of an item gadget $X$, disallow $X?$ except for $XY$, where $y \in S \cup \{l_0, e\}$. Within a set gadget, the items overlap, so after an individual set gadget will be an item character, so this does not affect the internals of the set gadgets. However, after a set gadget, it must either be the end of the string $e$, or the character $l_0$. Now disallow $l_0^{q}\overline{l}$ except $l_0^{q}\overline{l}$ for $q \in [0, m(3n + 2) - 1]$, forcing any substring starting with $l_0$ to have shape $l_0^{p(3n+2)}$ and thus length $1 + m(3n + 2)$. Thus, the space between adjacent set gadgets is thus $2 + m(3n + 2)$. For a set cover of size $k$, there are $k - 1$ such spaces, so the final superstring will have length $k(2 + m(3n + 2)) + 1$. By normalizing to $2 + m(3n + 2)$, this implies that we have an L-reduction.

**Proof of Theorem 1.** For any instance of min set cover, convert it to an instance of SCSN with alphabet size $n + m + 2$ by the gadgets described in this section. As this is an L-reduction, and min set cover is Log-APX-complete, SCSN is Log-APX-hard for an alphabet of size $n + m + 2$. However, by Theorem 1 in reference [Vas05], which proves that larger alphabet sizes
can be encoded in polynomial time in a binary alphabet, SCSN is Log-APX-hard even for binary alphabets, showing that SCSN is Log-APX-hard for any alphabet, provided that the number of negative strings is allowed to be polynomial in the number of positive strings, completing the proof. □

**Corollary 7.** SCSN is Log-APX-complete.

**Proof.** Recall the existence of a log-approximation algorithm [JL94]. Combined with Log-APX-hardness, this implies that SCSN is Log-APX-complete. □

### 3.2. SCSNW reduction from minimum ones 3SAT.

**Theorem 8** (SCSNW is NPOPB-hard). Given set of strings $s_1, \ldots, s_\eta$ and set of negative strings $t_1, \ldots, t_{\text{poly}(\eta)}$ with letters from $\Sigma \cup \{?\}$, optimizing for the shortest string $S$ with letters from $\Sigma$ that is a superstring of all $s_i$, but contains no $t_j$ as a substring, where each $?$ can match any letter of $\Sigma$, is NPOPB-hard.

We will prove this theorem by reduction from min ones 3SAT (or Distinguished Ones 3SAT), which is NPOPB-complete [Kan94]. Our strategy will be to use a frame gadget to force all clause gadgets to overlap a particular section of the string consisting of variables that can be set true or false. Then, using the variable gadget, we force each variable set to true to cause a large penalty by pushing a substring onto the end of the superstring. In the following, we assign positive strings by “$s_i =$” and negative strings by “$t_i =$”. Additionally, we choose here the alphabet $\Sigma = \{A, T, G, C\}$ to match the bases in the human genome. In the following gadgets, we also use the notation $B = GC, X = C^nG^n, R = (GAC)^n$ for the sake clarity and brevity.

**Gadget 9.** Frame gadget Forces clause gadgets to overlap and variable gadgets not in the variable region to not overlap.

**Design.**
The $s_{f_1}$ string specifies the locations for the $n$ variable set variables, and the $t_{v_i,C}$ and $t_{v_i,G}$ negative strings ensure that those variable locations are either $T$ or $A$. The $t_{f_2}$ negative string forces the superstring to start with $B$, constraining where strings can go in the superstring. The $t_{f_{ijk}}$ negative strings ensure that no two variable gadgets can overlap except through their respective $X$ strings if they are to the right of $R$.

**Gadget 10.** Variable gadget Forces any set variable corresponding to push this gadget out to the end of the superstring.

**Design.**

\[
\begin{cases}
\forall i \in [1,n] & s_{v_i} = X ? \cdots ? A ? \cdots ? X
\end{cases}
\]

If this variable gadget is in the variable region between $B$ and $R$ in the frame, then the corresponding variable must be set to false. However, if the corresponding variable is set to true, then the entire gadget must be pushed over to the right of $R$, and cannot overlap except maximally by overlapping their $X$ regions. Thus, each additional variable set to true costs an additional $3n$ characters to the length of the string.

**Gadget 11.** Clause gadget Requires the variable section of the superstring to be set matching the clauses in the min ones 3SAT problem.

**Design.**

\[
\begin{cases}
t_{c} = BX ? \cdots ? A ? \cdots ? A ? \cdots ? T ? \cdots ?
\end{cases}
\]

(positions are $T$ if negated in clause and $A$ otherwise)

For each clause, we create a negative string with all positions set to the opposite of what we want in the variable region. Thus, we disallow having all variables being the opposite of what would be needed to satisfy the clause. Thus, at least one of the variables must satisfy the clause, so all clauses with these gadgets must be satisfied. This is what forces some of the variables to be set to true.

**Proof of Theorem 8.** For any instance of min ones 3SAT, convert it to an instance of SCSNW with alphabet size 4 by the gadgets described in this section. Construction is polynomial and takes $O(n^5)$ operations, most of which are used up constructing the negative strings of the frame gadget. If there is a Min Ones solution of weight $W$, then the corresponding SCSNW problem has a solution string of length $2 + 2n + n + 2n + 3n + 3nW + n = 2 + 9n + 3nW$. For any solution to the SCSNW problem, one gets a solution to min ones 3SAT by simply reading off the variable locations, and the weight of that solution is no more than $W$ given a superstring of length $2 + 9n + 3nW$. Note that by omitting some of the variable strings, this reduction also
works for minimum distinguished ones 3SAT. After normalizing the SCSNW objective by \( n \) (or equivalently the length of the longest input string), this reduction is an L-reduction. As min ones 3SAT (or min distinguished ones 3SAT) is NPOPB-hard, so thus must be SCSNW, completing the proof. □

As an aside, one might attempt to apply this reduction to SCSN, given that Lemma 2 can be generalized to allow wildcards in arbitrary positions in SCSN. That would of course lead to contradictory results given that SCSN is known to be in Log-APX, and would imply an error in this proof. However, note that this proof of SCSNW hardness requires access to \( \text{poly}(n) \) wildcards per string and the proof of Lemma 2 can only be generalized to allow a constant number of wildcards per string (otherwise, we would need an exponential number of negative strings in SCSN). Thus, this reduction cannot be used for SCSN, and SCSNW is provably harder than SCSN.

4. Discussion

We reviewed the complexity of SCS and variants depending on whether negative strings and wildcards were allowed, and built on those proofs to get new hardness results: SCSN is Log-APX-hard and therefore Log-APX-complete and SCSNW is NPOPB-hard (Table 2 in intro). We conjecture that SCSNW is in NPOPB if there exists a feasible solution, which would imply NPOPB-completeness, but this is nontrivial to show. Future work could include proving completeness results for SCSW and SCSNW.

5. Acknowledgments

Y.W.Y. is supported by a Hertz Foundation fellowship. The author thanks Erik Demaine, Jayson Lynch, Sarah Eisenstat, and the entire Fall 2014 MIT 6.890 class for insightful conversations. Sarah Eisenstat is especially acknowledged for finding prior results in the literature that the author missed. This manuscript was originally conceived as a final project for the Fall 2014 MIT 6.890 class Algorithmic Lower Bounds: Fun with Hard Proofs, taught by Erik Demaine.

References

[BJL+94] Avrim Blum, Tao Jiang, Ming Li, John Tromp, and Mihalis Yannakakis, Linear approximation of shortest superstrings, Journal of the ACM (JACM) 41 (1994), no. 4, 630–647.

[Cre97] Pierluigi Crescenzi, A short guide to approximation preserving reductions, Computational Complexity, 1997. Proceedings., Twelfth Annual IEEE Conference on (Formerly: Structure in Complexity Theory Conference), IEEE, 1997, pp. 262–273.

[GKM13] Alexander Golovnev, Alexander S Kulikov, and Ivan Mihajlin, Approximating shortest superstring problem using de bruijn graphs, Combinatorial Pattern Matching, Springer, 2013, pp. 120–129.

[JL94] Tao Jiang and Ming Li, Approximating shortest superstrings with constraints, Theoretical Computer Science 134 (1994), no. 2, 473–491.
Viggo Kann, *Polynomially bounded minimization problems that are hard to approximate*, Nordic Journal of Computing 1 (1994), 317–331.

Marek Karpinski and Richard Schmied, *Improved inapproximability results for the shortest superstring and related problems*, Proceedings of the Nineteenth Computing: The Australasian Theory Symposium-Volume 141, Australian Computer Society, Inc., 2013, pp. 27–36.

Ming Li, *Towards a dna sequencing theory (learning a string)*, Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on, IEEE, 1990, pp. 125–134.

Carsten Lund and Mihalis Yannakakis, *On the hardness of approximating minimization problems*, Journal of the ACM (JACM) 41 (1994), no. 5, 960–981.

Bin Ma, *Why greed works for shortest common superstring problem*, Theoretical Computer Science 410 (2009), no. 51, 5374–5381.

Paul Medvedev, Konstantinos Georgiou, Gene Myers, and Michael Brudno, *Computability of models for sequence assembly*, Algorithms in Bioinformatics, Springer, 2007, pp. 289–301.

Marcin Mucha, *Lyndon words and short superstrings*, Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, 2013, pp. 958–972.

Sascha Ott, *Lower bounds for approximating shortest superstrings over an alphabet of size 2*, Graph-theoretic concepts in computer science, Springer, 1999, pp. 55–64.

Kari-Jouko Räihä and Esko Ukkonen, *The shortest common supersequence problem over binary alphabet is np-complete*, Theoretical Computer Science 16 (1981), no. 2, 187–198.

Virginia Vassilevska, *Explicit inapproximability bounds for the shortest superstring problem*, Mathematical Foundations of Computer Science 2005, Springer, 2005, pp. 793–800.