INTERNATIONAL JOURNAL OF SOFTWARE ENGINEERING & COMPUTER SYSTEMS (IJSECS)
ISSN: 2289-8522 e-ISSN: 2180-0650
VOL. 6, ISSUE 1, 8 – 17
DOI: https://doi.org/10.15282/ijsecs.6.1.2020.2.0065

ORIGINAL ARTICLE

D-GREEDY: Greedy shortest superstring with delayed random choice

Mohammad F. J. Klaib[1,*], Mutaz Rasmi Abu Sara[1], and Masud Hasan[1]

[1]Department of Computer Science, Taibah University, Al-Medina, Saudi Arabia

ABSTRACT – The shortest superstring problem for a given set of strings S is to find a string of minimum length such that each string in S is a substring of the resulting string. This problem is known to be NP-complete. A simple and popular approximation algorithm for this problem is the *Greedy Algorithm*, which at each step merges a pair of strings that have maximum overlap. If more than one pair have maximum overlap, it takes a pair in random. In this paper, we modify the Greedy Algorithm such that instead of taking a pair in random, it takes the pair for which the overlap in the next step is maximum. This modification brings our algorithm closer to the optimal. We analyze our algorithm and compare it with the Greedy Algorithm for different types of input. For some previously well-studied input strings, our algorithm performs better or same as the Greedy Algorithm. We also implement both the Greedy Algorithm and our algorithm. The experimental results show that our algorithm can outperform the Greedy Algorithm substantially (by an amount as much as 23%) in many cases, and as a whole our algorithm is same or better than the Greedy Algorithm.

ARTICLE HISTORY
Received: 11 Jan 2020
Accepted: 15 April 2020

KEYWORDS
Shortest Superstring
Greedy Strategy
Approximation Algorithm
Approximation Ratio
Random Choice
String Overlap

INTRODUCTION

Given a set of strings *S*, the *shortest (common) superstring problem (SSP)* for *S* is to find a string *s* of minimum length such that each string in *S* is a substring of *s*. As an example, consider $S = \{CGTA, TAC, TAG\}$. The shortest possible superstring *s* is *TACGTAG* of length seven. Every string in *S* is a substring of *s*. The string *TAC* is a substring of *s* at the front, *TAG* is a substring at the end of *s*, and *CGTA* is a substring inside of *s*. Here, the overlaps among the strings of *S* that cause the compression in *s* are *TA* and *C*. There are two chunks of *TA* in *S*, but *s* contains only one. Similarly, there are two *C* in *S*, but *s* contains only one. This is the maximum compression possible in *s*. Any other superstring will have less amount of compression and consequently will have length more than *s*, and thus, will not be a resulting superstring. For example, *CGTACTAG* is another superstring of all strings in *S*, but it is not the shortest, as it has length eight.

In a general setting, there are *n* strings in *S*. All these *n* strings are generated from the same alphabet Σ, which is given along with *S*. The set Σ can vary for different applications. For example, in bioinformatics, Σ could be just the four letters *C*, *G*, *T*, *A*, i.e., $\Sigma = \{C, G, T, A\}$, because these four letters are used to express all DNA sequences. For an application with binary strings only, Σ could be simply $\Sigma = \{0, 1\}$. However, in general, the problem SSP is studied for Σ that consists of all letters, digits, and symbols in a language. In this paper we also follow that suit.

The problem SSP has numerous applications in different areas. The most important application that has been mentioned in the literature is in bioinformatics (Kaplan et al., 2005), (Gallant et al., 1980), (Braquelaire et al., 2017). In bioinformatics, one important problem is to find the sequence of letters (*C*, *G*, *T*, *A*) in a DNA fragment by a widely used technique, called shotgun sequencing. In that technique, the original DNA fragment is cut into smaller fragments in random. Those smaller fragments are copied into multiple copies. Then, they are all sequenced (by biological process) by the letters (*C*, *G*, *T*, *A*). Finally, these small sequences are realigned to get the sequence of the original DNA fragment. In this last step, there can be many alignments possible. However, the challenge is to find the one that represents the original DNA with smallest possible length so that the error is minimum. This is a computational problem and is exactly same as SSP that we study in this paper.

Other important applications of SSP can also be found in the literature. One such application is in data compression (Storer et al., 1982).

The Problem SSP is a very primitive and well-studied problem in computer science. It has been studied both theoretically and experimentally. Theoretically, the most important result, although it dates to about thirty-three years back, is that finding the exact solution of SSP is NP-complete. That means, no polynomial time algorithm exists so far for finding *s* from *S*. This is for the case when Σ is not restricted. However, the problem remains NP-complete even for binary alphabet, i.e., $\Sigma = \{0, 1\}$ (Maier et al., 1977), (Garey et al., 1979), (Gallant et al., 1980).

As a consequence, researchers sought to find *approximation algorithms*. Recall that an approximation algorithm for an NP-complete problem is a polynomial-time algorithm whose approximated solution is guaranteed to be within a fixed ratio of the optimal solution (Cormen et al., 2009). This ratio is called *approximation ratio*. An approximation ratio is computed theoretically for the worst-case scenario of the algorithm. An approximation ratio that is closer to 1 is better, as a ratio 1 means the approximation algorithm and an optimal algorithm are the same.

*CORRESPONDING AUTHOR | Mohammad F. J. Klaib | ✉ mklaib@taibahu.edu.sa

Two types of approximation algorithms are developed by the researchers for SSP. In one type, the approximation algorithms are simple, but the approximation ratio is higher (thus, far from an optimal solution). In the other type, approximation ratios are close to 1, but the algorithms are much involved.

The first ever approximation algorithm for SSP is called the *Greedy Algorithm* (Gallant et al., 1980) and falls in the first category. The algorithm is very simple, and it works in an iterative manner. At each step, it merges a pair of strings from S that have maximum overlap among all pairs. If more than one pair have maximum overlap, it takes a pair in random. It removes the pair from S and insert into S their maximally overlapped string. The algorithm continues in this way until S has only one string. This final string is the output string s of the Greedy Algorithm.

Because of the random choice made by the Greedy Algorithm, in a good case the algorithm may pick a pair that gives better overlaps in subsequent steps, and finally, the algorithm gives a superstring that is very close to the optimal. On the other hand, in a bad case the algorithm may pick a pair that gives worse overlaps in the subsequent steps, and finally, the algorithm gives a superstring that is very far from the optimal. But the good news is that researchers proved that in any case (good or bad) the length of the final superstring generated by the Greedy Algorithm is no more than 3.5 times the length of the optimal superstring of S. It means that the approximation ratio of the Greedy Algorithm is 3.5 (Kaplan et al., 2005). However, researchers also conjectured that the actual approximation ratio of the Greedy Algorithm is far better and could be as less as 2, but no one could prove it yet (Alanko et al., 2017), (Golovnev et al., 2019).

In another consequence of the NP-completeness of SSP, researchers also presented different experimental techniques to find approximate solution to SSP Moreover, researchers also presented experimental evidences to support the above-mentioned conjecture on the approximation ratio of 2 of the Greedy Algorithm (Braquelaire et al., 2017), (Cazaux et al., 2018), (Romero et al., 2004). From their experiments, a solution was never generated whose length is more than twice the length of an optimal solution. This indicates that the actual approximation ratio of the Greedy Algorithm is highly likely to be 2.

Our results. In this paper, we modify the Greedy Algorithm such that instead of taking a pair in random, it takes the pair for which the overlap in the next step is maximum. If in the next step there are more than one pair that give maximum overlap, then we take one of them at random and do not look into the third step anymore. Thus, we delay the random choice by going into one step deeper to see which pair of strings would be a good choice for the current step. We call our algorithm D-GREEDY.

We compare the performance of our algorithm in two ways. First, we analyze our algorithm and compare it with the Greedy Algorithm for different types of inputs. Those inputs include some well-known pattern of strings that have been used before by researchers to analyze the Greedy Algorithm as well as their own algorithms. Our analysis shows that our algorithm can outperform the Greedy Algorithm in many cases, and the Greedy Algorithm never outperforms our algorithm. Second, we implement both the algorithms, for which the experimental results show that our algorithm can outperform the Greedy Algorithm substantially in many cases. The amount could be as much as 23%. Moreover, as a whole our algorithm is same or better than the Greedy Algorithm.

The rest of the paper is organized as follows. In the literature review section, we review the previous results, as there are numerous literature available related to the Greedy Algorithm for SSP. In the methodology and the algorithm section, we describe and give pseudo code of the Greedy Algorithm and our algorithm. In the comparing with existing greedy analysis section, we analyze and compare our algorithm with the Greedy Algorithm with several well-studied examples. Experimental results section gives the experimental results. In the result discussion section, we discuss the results that we achieve in this paper. Finally, conclusion section concludes the paper with some future work.

LITERATURE REVIEW

It started in 1977, when Maier and Storer (Maier et al., 1977) proved that SSP is NP-complete. This has also been mentioned by Garey and Johnson in their seminal book (Garey et al., 1979) as Problem SR9 (Garey et al., 1979). However, a formal publication on the NP-completeness proof appears in (Gallant et al., 1980).

In 1980, the Greedy Algorithm, which was the very first approximation algorithm for this problem, was given by Gallant et al. (Gallant et al., 1980). The algorithm was presented without giving any approximation ratio. In 1994, Blum et al. (Blum et al., 1994) proved that the approximation ratio of the Greedy Algorithm is 4. Later, in 2005, Kaplan and Shafrir (Kaplan et al., 2005) proved that the Greedy Algorithm is better than that by proving that its approximation ratio is 3.5.

Subsequently, several other approximation algorithms with constant approximation ratio were presented by different authors. The approximation ratio of these algorithms ranges from 4 to $2\frac{11}{23}$. See (Mucha, 2013) for a list of all these approximation algorithms.

In spite of these approximation algorithms with better ratios, the Greedy Algorithm remains a popular choice for the SSP for two reasons. One reason is the simplicity of the Greedy Algorithm. It is easy to implement the Greedy Algorithm compared to the above-mentioned constant ratio approximation algorithms. (Although, some of the above-mentioned algorithms are based on the Greedy Algorithm. For example, Blum et al. (Blum et al., 1994) gave two algorithms with ratios 4 and 3 that are based on the Greedy Algorithm.)

The other reason of the Greedy Algorithm being a popular choice is its much-anticipated approximation ratio. It has been long conjectured that the Greedy Algorithm has a far better approximation ratio of 2 (Alanko et al., 2017), (Tarhio et al., 1988), (Turner, 1989), (Blum et al.,1994), (Weinard et al., 2006), (Nabi et al., 2012), (Mucha, 2013), (Kulikov et

al., 2015), (Cazaux et al., 2018 (August)), (Golovnev et al., 2019). Although there exist examples (Blum et al.,1994), (Weinard et al., 2006), (Nabi et al., 2012) that show that the Greedy Algorithm can produce a superstring whose length is twice the optimal, there is no known example where the Greedy Algorithm performs worse than twice the optimal. Moreover, the implementations of the Greedy Algorithm that can be found in the literature also suggest that in most cases the approximation ratio of the Greedy Algorithm is less than 2 (Alanko et al., 2017), (Tarhio et al., 1988), (Turner, 1989). See (Gevezes et al., 2014) for a survey on different results on approximation guarantee and implementation of the Greedy Algorithm.

SSP and the Greedy Algorithm for SSP continue to draw attention by the researchers as they study their variations. The interesting thing here is that in most of these variations, the researchers design their own approximation algorithms by using, in way or other, the classical Greedy Algorithm. Golovnev et al. (Golovnev et al., 2013 (June)) studied SSP where all the strings in S have the same size of r or less. This variation of the problem is called *r-SSP*. They presented an approximation algorithm with ratio $1\frac{1}{3}$ for 6-SSP. This ratio is better than the best-known approximation ratio of the Greedy Algorithm, which is 3.5. In a different paper (Golovnev et al., 2013 (August)), they also made an attempt to solve SSP exactly in less than 2^n number of steps. They presented an exact algorithm for $r = 3$ that runs in $n^{\frac{n}{3}}$ time.

Jiang et al. (Jiang et al., 1992) introduced an interesting variation of SSP, where the resulting superstring contains as a substring every string of S or its reversal. They mentioned that this variation may be applicable in bioinformatics. For this variation, they proposed an approximation algorithm with ratio 4. Also, their algorithm is based on greedy strategy, but not exactly same as the Greedy Algorithm. Later, Fici et al. (Fici et al., 2016) studied this variation and proposed an approximation algorithm. They also adapted the original Greedy Algorithm and achieved an approximation ratio of 2. However, they computed the approximation ratio by comparing the total amount of compression (i.e., overlap) done in the superstring generated by their algorithm with the compression in the shortest superstring, rather than comparing them by their lengths. Cazaux and Rivals (Cazaux et al., 2018 (August)) continued to work on the variations where the ratio is computed by compression. With this measure, they proposed several approximation algorithms with ratio 2 for 3-SSP, 3.5 for 6-SSP, and 1.5 for 2-SSP.

Cazaux and Rivals (Cazaux et al., 2016), (Cazaux et al., 2018 (July)) studied some other variations of SSP. In one variation (Cazaux et al., 2016), which is called cyclic SSP, the resulting superstring is to be a cyclic string, rather than a linear string. This version is solvable in polynomial time and they presented a linear-time algorithm to solve this problem. In another variation (Cazaux et al., 2018 (July)), which is called SSP with multiplicities, the resulting superstring should contain each string of S at least m times, for a given integer m. The authors provided an approximation algorithm for this problem with ratio 2.

From the practical point of view, Braquelaire et al. (Braquelaire et al., 2017) presented an approximation algorithm and ran it on large biological real data and found that their algorithm gives approximation ratio close to 2. Cazaux et al. (Cazaux et al., 2018 (June)) took a different approach. For a given set of strings S, they presented a linear-time algorithm to compute the lower and upper bounds on the length of the shortest superstring of S. More importantly, they ran their algorithm on different sets of input and found that the two bounds are almost very close. From this finding, they argue that most of the time the Greedy Algorithm would generate a solution that is very close to the optimal solution. Not only that, Romero et al. (Romero et al., 2004) showed that approximation algorithms that are carefully designed can be in practice far better than their proven worst-case ratios.

These empirical evidences on the goodness of the approximation algorithms (including the Greedy Algorithm) for SSP are also supported by the theoretical analysis provided by Ma (Ma, 2009). Ma theoretically showed that the actual approximation ratio of the Greedy Algorithm for random input is far better than the proven ratio (which is 3.5) and, in fact, is close to 1 (i.e., the optimal).

METHODOLOGY AND THE ALGORITHM

From the above literature review, it is quite evident that for SSP the Greedy Algorithm remains a prime choice for finding a solution that is close to the optimal. Our approach in this paper follows the same evident way to reach a solution that is closer to the optimal. We modify the Greedy Algorithm so that it goes closer to the optimal solution, and we experimentally verify that it indeed goes closer to the optimal.

We start with the definition of some notations that will be used throughout the rest of the paper. Let x and y be two strings in S. We assume that none of x or y is a substring of the other. This assumption is natural. Because, if any of them is a substring of the other, then the former one can simply be deleted from S.

The *overlap* of two strings x and y (in this order) is the maximum suffix of x which is also a prefix of y. The *merge*(x, y) is the string of minimal length where x is a prefix and y is a suffix. For example, for $x = abccd$ and $y = ccdab$, overlap of x and y is *ccd*, overlap of y and x is *ab*, *merge*(x, y) is the string *abccdab*, and *merge*(y, x) is the string *ccdabccd*.

The Greedy Algorithm continuously merges pairs of different strings having maximum overlap until a superstring is found. At each step, the Greedy Algorithm finds the pair of strings in S that have maximum overlap. It merges that pair, insert the merged pair into S, and then removes the pair from S. The Greedy Algorithm continues in this way until a single string remains in the set S, which is the resulting superstring. At any step, if there are more than one pair having maximum overlap, the Greedy Algorithm selects a pair randomly. Algorithm 1 is the pseudo code of the Greedy Algorithm.

We consider the Greedy Algorithm in a different way. At any step (say, step X), if more than one pair of strings have maximum overlap, we do not choose a random pair in the current step (step X). Rather, for all such pairs, we see which

one would give maximum overlap in the next step (step X+1). We choose that pair for the current step (step X). If more than one pair give maximum overlap in the next step (step X+1), we choose one of them at random for the current step (step X). Thus, our algorithm can be considered as a "double greedy" strategy with the random choice delayed by one step.

Algorithm 1 Greedy Algorithm

```
S = {w1, w2, w3, …, wn}
while ||S|| > 1 do
        Find a pair of strings that have maximum overlap. If more than
        one pair have maximum overlap, then choose one of them in random.
        Let (wi, wj) be that pair.
        S = {S − {wi, wj}} ∪ merge(wi, wj)
end while
return S
```

More formally, at any step, for each pair (w_i, w_j) having maximum overlap, our algorithm does the following. It takes a copy S' of the original set of strings S, merges the pair, removes the pair from S', and add the newly merged string to the set S'. Then it finds the maximum overlap among all pairs of strings in S'. Our algorithm takes the set S' as S for which the maximum overlap in the second step is the maximum over all pairs (w_i, w_j). If there are more than one pair that give maximum overlap in S', then it takes the S' for one of those pairs at random. (Thus, our algorithm does not go further to the third step or so on to select pairs of strings.) Algorithm 2 is the pseudo code of our algorithm. Also see comparing with existing greedy analysis section for illustrations.

Algorithm 2 D-GREEDY

```
S = {w1, w2, w3, …, wn}
while ||S|| > 1 do
    Find a pair of strings that have maximum overlap
   if more than one pair have maximum overlap then
       x = 0
       for each such pair (wi, wj) ∈ S do
        S' = {S − {wi, wj}} ∪ merge(wi, wj)
        Find the maximum overlap y between two strings in S'
        if y ≥ x then
           x = y, a = i, b = j
            // Equality in the above if statement handles the case of
            // no overlap among the strings in S'.
            // It also takes the last value of (wi, wj) if there are more
            // than one pair giving the maximum x.
            // This maintains the randomness in the second step,
            // because any of the pair can be the last value of (wi, wj).
         end if
       end for
     end if
     S = {S − {wa, wb}} ∪ merge(wa, wb)
 end while
 return S
```

Observe that, instead of stopping at the second step (step X+1), our algorithm could continue to look into the third step (step X+2), forth step (X+3), and so on as long as there is a tie between more than one pair. That would give an optimal superstring, because all possible outcomes are being checked in exhaustive way. However, that would take exponential time, because there could be exponential number of possible paths to move forward, and each path would lead to a different superstring. An optimal superstring would be one of those superstrings. However, since our algorithm does not move forward after the second step (step X+1), it is closer but not exactly same as an optimal solution.

Running time. It is natural that the running time of our algorithm will be higher than that of the Greedy Algorithm, because our algorithm looks one step further to decide which strings to take in the current step.

More precisely, the difference between our algorithm and the Greedy Algorithm is the inner for loop in Algorithm 2. There are many polynomial time implementations of the Greedy Algorithm (Alanko et al., 2017), (Tarhio et al., 1988), (Turner, 1989). In fact, the best among them by Alanko and Norri (Alanko et al., 2017) runs in $O(n \log m)$ time, where m is the total length of the strings in S. Our algorithm is also polynomial as long as that inner for loop runs in polynomial time. The inner for loop finds for each pair (w_i, w_j) the maximum overlap between two strings in S'. In worst case, there can be $O(n^2)$ values of (w_i, w_j) in S. Finding the maximum overlap between a pair of strings in S' can also be done in $O(n \log m)$ time by the same implementation of the Greedy Algorithm by Alanko and Norri (Alanko et al., 2017), because the Greedy Algorithm also finds maximum overlap among a set of strings. Therefore, the inner for loop runs in $O(n^3 \log m)$ time.

Similarly, by using the Greedy Algorithm we can find the maximum overlap between two strings in the outer while loop in $O(n \log m)$ time. Therefore, the total running time of our algorithm is $O(n \log m)$ times $O(n^3 \log m)$, which becomes $O(n^4 \log^2 m)$.

It is worth mentioning that both the Greedy Algorithm and our algorithm are approximation algorithms of an NP-complete problem, and an approximation algorithm is feasible as long as it is polynomial (Cormen et al., 2009). Since our algorithm runs in polynomial time, it is a feasible approximation algorithm similar to the Greedy Algorithm.

COMPARING WITH EXISTING GREEDY ANALYSIS

In this section we provide different examples to analyze our algorithm and to compare it with the Greedy Algorithm and an optimal algorithm. We analyze our algorithm by non-trivial generic examples. By "generic" we mean that the input strings can be arbitrary large. Some of these examples have been presented before by other authors to analyze the Greedy Algorithm. We show that those examples have important role to analyze our algorithm too.

We provide three generic examples. Our first example is to illustrate how the Greedy Algorithm and our algorithm work. This example also illustrates how our algorithm can outperform the Greedy Algorithm. Then we show a second example where our algorithm outperforms the Greedy Algorithm substantially. In fact, we show that the Greedy Algorithm can be twice worse than our algorithm. This example also shows that our algorithm performs close to optimal. Finally, we show a third example where our algorithm and the Greedy Algorithm perform similarly. For this example, an optimal algorithm is much better than our algorithm and the Greedy Algorithm. The following three subsections provide these three examples respectively.

An example for illustration

Consider the following input $S = \{cabab, babd, baba\}$. At first step, the pair of strings (*cabab*, *babd*) and the pair (*cabab*, *baba*) have the maximum overlap of *bab*. Since the Greedy Algorithm randomly chooses one of the two pairs for merging, if it chooses the pair (*cabab*, *babd*), then *S* becomes {*cababd*, *baba*}. In the second step, the overlap is zero between *cababd* and *baba*. Therefore, the Greedy Algorithm simply merges the two strings in any order. The resulting string by the Greedy Algorithm is *cababdbaba* or *babacababd*, whose length is ten.

On the other hand, our algorithm at the first step checks which pair gives better overlap in the next step. Among the two pairs (*cabab*, *babd*) and (*cabab*, *baba*) having the maximum overlap *bab*, if our algorithm chooses the pair (*cabab*, *babd*) (as it was also chosen by the Greedy Algorithm), then the maximum overlap in the next step is zero. (This is same as the Greedy Algorithm that we have seen in the previous paragraph). On the other hand, if our algorithm chooses the other pair (*cabab*, *baba*), then in the next step *S* becomes {*cababa*, *babd*}. This would give a maximum overlap of *ba* among the two strings *cababa* and *babd* in the next step. Therefore, our algorithm chooses this later pair in the first step. This gives $S = \{cababa, babd\}$ for the second step of our algorithm. After second step, the resulting string by our algorithm is *cababab*d, which is of length eight. Therefore, our algorithm outperforms the Greedy Algorithm for this example.

The next example will show that our algorithm can substantially outperform the Greedy Algorithm and that our algorithm performs same as an optimal algorithm.

Our algorithm (D-GREEDY) substantially outperforms the Greedy Algorithm

Tahmid and Sen (Nabi et al., 2012) presented the following example to analyze the Greedy Algorithm. By this example, they showed that the Greedy Algorithm can perform close to twice the optimal. However, we shall show that this is a very good example for our algorithm. In fact, our algorithm will be same as optimal for this example.

By this example, $S = \{ab^n, b^{n+1}, b^nc\}$. At the first step, the Greedy Algorithm has three pairs (ab^n, b^{n+1}), (b^{n+1}, b^nc), and (ab^n, b^nc) with maximum overlap b^n of size *n*. If the Greedy Algorithm randomly chooses the third pair, then *S* for the second step becomes $\{ab^nc, b^{n+1}\}$. The Greedy Algorithm has no more overlap in the next step and it simply merges the two strings in any order. Therefore, the resulting string by the Greedy Algorithm is ab^ncb^{n+1} or $b^{n+1}ab^nc$ of length $2n + 3$.

On the other hand, our algorithm at the first step will choose either the first pair (ab^n, b^{n+1}) or the second pair (b^{n+1}, b^nc). Because, if the first pair is chosen, then *S* becomes $\{ab^{n+1}, b^nc\}$. This will give a maximum overlap of b^n in the second step. Similarly, if the second pair is chosen, then *S* becomes $\{ab^n, b^{n+1}c\}$. This will also give a maximum overlap of b^n in the second step. In either case, our algorithm in the second step will have maximum overlap of *bn*. The resulting superstring by our algorithm after the second step will be $ab^{n+1}c$ of length $n + 3$.

This example shows that our algorithm can outperform the Greedy Algorithm substantially. In fact, our algorithm can produce a superstring of size just half of that produced by the Greedy Algorithm. Because, the ratio of the length of two superstrings generated by the Greedy Algorithm and our algorithm is $\frac{2n+3}{n+3}$, which tends to 2 as *n* gets bigger.

Note that the superstring generated by our algorithm is optimal. Therefore, for this example, our algorithm is same as optimal.

The next example will show that our algorithm performs same as the Greedy Algorithm and that an optimal algorithm is far better than these two algorithms.

Greedy Algorithm is same as D-GREEDY

The example that we provide in this section will show that the Greedy Algorithm as well as our algorithm can perform twice the optimal. This example was first provided by Blum et al. (Blum et al.,1994) and subsequently referred by several other authors (Weinard et al., 2006), (Nabi et al., 2012) to show that the much-anticipated approximation ratio of the Greedy Algorithm cannot be better than 2.

By this example, $S = \{c(ab)^n, (ba)^n, (ab)nd\}$, for $n = 1, 2, 3, \ldots$. For this input, the Greedy Algorithm and our algorithm perform exactly the same way. In the first step, the only pair $(c(ab)^n, (ab)^nd)$ gives the maximum overlap of $(ab)^n$. Note that the overlap length is $2n$. After the first step, S becomes $\{c(ab)^nd, (ba)^n\}$. In the next step, there is no overlap. Therefore, the two strings are simply merged in any order. The resulting superstring becomes $c(ab)^nd(ba)^n$ or $(ba)^nc(ab)^nd$ of length $4n + 2$.

However, an optimal algorithm in the first step can choose a different pair $((ba)^n, (ab)^nd)$. Here, the maximum overlap is $(ab)^{n-1}a$ of length $2n-1$. Observe that, the amount of overlap in this case is smaller than the one chosen by the Greedy Algorithm and our algorithm (which was of length $2n$.) This choice by an optimal algorithm makes S for the second step as $S = \{c(ab)^n, b(ab)^nd\}$. In this step, the maximum overlap for an optimal algorithm is $(ba)^{n-1}b$. This gives the resulting superstring as $c(ab)^{n+1}d$ of length $2n+4$. The ratio of the lengths of the superstring resulting from the Greedy Algorithm (as well as our algorithm) and the optimal algorithm is $\frac{4n+2}{2n+4}$, which tends to 2 as n gets bigger.

EXPERIMENTAL RESULTS

The above examples motivated us to implement the Greedy Algorithm and our algorithm and to run them on larger input with several variations. We do the experiments for different types of input strings, such as different alphabet size, different string length, and different number of strings. We show that for different input patterns, our algorithm can substantially outperform the Greedy Algorithm. As the input strings become more random, our algorithm and the Greedy Algorithm get closer in performance—with most of the time our algorithm is better than the Greedy Algorithm and the rest of the time, same. This holds as the string size, alphabet size, or the number of inputs become larger. The following two subsections present these experimental results.

Our algorithm (D-GREEDY) substantially outperforms the Greedy Algorithm

First, we consider the following inputs with some small number of strings of small lengths (input S_1 to S_8 below).

S_1 = {*ABB*, *AAB*, *BBA*, *AAA*, *BBB*}
S_2 = {*CATG*, *CTAAGT*, *GCTA*, *TTCA*, *ATGCATC*}
S_3 = {*babd*, *cabab*, *baba*}
S_4 = {*ABA*, *ABB*, *AAA*, *AAB*, *BBB*, *BBA*, *BAB*, *BAA*}
S_5 = {*ng_lon*, *_long_*, *a_long*, *long_l*, *ong_ti*, *ong_lo*, *long_t*, *g_long*, *g_time*, *ng_tim*}
S_6 = {0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111}
S_7 = {cab^k, ab^kab^ka, b^kdab^{k-1}}, where $k = 10$
S_8 = {*CGTACG*, *TACGTA*, *ACGTAC*, *GTACGT*, *GTACGA*, *TACGAT*}

Table 1 shows that for these inputs, our algorithm outperforms the Greedy Algorithm by an amount as much as 23%. Moreover, for none of the input the Greedy Algorithm outperforms our algorithm.

Table 1. Comparison of our algorithm (D-GREEDY) with the Greedy algorithm

Input	Greedy Algorithm		D-GREEDY		
	Superstring	Length	Superstring	Length	Improvement (%)
S_1	*BBBAAABBA*	9	*AAABBBA*	7	22%
S_2	*GCTAAGTTCATGCATC*	16	*GCTAAGTTCATGCATC*	16	0%
S_3	*babacababd*	10	*cababababd*	8	20%
S_4	*BBBABBAABAAA*	12	*BAAABBBABA*	10	16.7%
S_5	*g_long_lona_long_time*	21	*a_long_long_time*	16	23.8%
S_6	*1011010111100111000010000*	24	*1001010000111101100*	19	20.8%
S_7	*bbbbbbbbbdabbbbbbbbbcabbbbbbbbbbbbbbbbbbbbaba*	41	*bbbbbbbbbdabbbbbbbbbcabbbbbbbbbbbbbbbbbbbaba*	41	0%
S_8	*TACGTACGTACGAT*	14	*GTACGTACGAT*	11	21.4%

Our next experiment is for more generalized input S. The strings of S have some patterns. But S can have arbitrarily large number of strings. S is generated by the following equation.

$S = \{(XYY, XXY, YYX, XXX, YYY)n\}$, where $X, Y \in [A, Z]$

For any value of n, values of (X, Y) are taken n times in random and the set S contains for each (X, Y) the strings XYY, XXY, YYX, XXX, YYY. For example, for n = 3, suppose that (X, Y) is (C, G), (I, E) and (X, B). Therefore,

S = {*CGG*, *CCG*, *GGC*, *CCC*, *GGG*, *IEE*, *IIE*, *EEI*, *III*, *EEE*, *XBB*, *XXB*, *BBX*, *XXX*, *BBB*}

The experimental results for this input setting are shown in Table 2. From the table it can be seen that our algorithm outperforms the Greedy Algorithm by an amount as much as 15%. Moreover, for none of the input the Greedy Algorithm outperforms our algorithm.

Table 2. Comparison of our algorithm (D-GREEDY) with the Greedy algorithm

n	Superstring length		
	Greedy Algorithm	D-GREEDY	Improvement (%)
25	123	104	15.44%
50	226	201	11.06%
75	303	279	7.9%
100	368	350	4.9%
250	692	686	0.86%
500	1005	1002	0.29%

Greedy Algorithm almost similar to D-GREEDY

Remember that when our algorithm looks one step deeper to find a larger overlap, it does not guarantee that a larger overlap will indeed be found. However, if it is not found, then it takes exactly the same overlap that would be taken by the Greedy Algorithm. Therefore, if our algorithm cannot outperform the Greedy Algorithm, then it remains the same as the Greedy Algorithm. Moreover, it does not become worse than the Greedy algorithm. The following experiments will show these facts.

We continue our experiments for larger and more random input. Our inputs still have some patterns. We vary the input in three ways based on the number of strings n, the alphabet size, and the length of the strings in S.

First, we consider the input setting where the length of the strings in S is fixed to ten and the alphabet size is fixed to three. We vary n from five to one hundred. The input strings are generated as follows:

$S = \{(C_1C_2C_3 \ldots C_{10})^1 \ldots (C_1C_2C_3 \ldots C_{10})^n\}$,

where $C_i \in \{A, B, C\}$. For example, for n = 4, S could be as follows.

S = {*BBCABCCCBA*, *ABCBAABABC*, *ABBACABACA*, *BCCBAABCAC*}

The experimental results for this input setting show that our algorithm is either better than the Greedy Algorithm by an amount up to 2.66% or same as the Greedy Algorithm. See Table 3. In the table, the range of values in an entry for the Greedy Algorithm indicates the results of the Greedy Algorithm for different runs. For example, for n = 10, the Greedy Algorithm gives superstrings of length ranging from seventy-five to seventy-seven. This is same for our algorithm for entries having range of values.

Table 3. Comparison of our algorithm (D-GREEDY) with the Greedy algorithm for variable number of input strings

n	Superstring length		
	Greedy Algorithm	D-GREEDY	Improvement (%)
5	44	44	0%
10	75-77	75	0% − 2.66%
15	112-113	112	0% − 0.8%
20	143-144	143	0% − 0.69%
30	212	212	0%
40	278-281	278	0% − 1.07%
50	235-328	325	0% − 0.9%
60	399	399	0%
70	450-452	450-452	0% − 0.44%
80	513-515	513	0% − 0.19%
90	565	565	0%
100	622-625	622-625	0% − 0.48%

Next, we consider the input setting where the length of the strings in S is fixed to ten and n is fixed to fifty. We vary the alphabet size from three to fifteen. The input strings are generated as follows.

$S = \{(C_1C_2C_3 \ldots C_{10})^1 \ldots (C_1C_2C_3 \ldots C_{10})^{50}\}$

where $C_i \in \{A, B, C\}$ or $C_i \in \{A, B, C, D\}$ or … or $C_i \in \{A, B, C, \ldots, O\}$. For example, for alphabet size four, S could be as follows.

S = {*DCABBDCAAB*, *AADDACBDAA*, *ABDADABCCD*, …, *ABCCBADCCA*}

The experimental results for this input setting show that our algorithm is either better than the Greedy Algorithm by an amount up to 0.92% or same as the Greedy Algorithm. See Table 4.

Table 4. Comparison of our algorithm (D-GREEDY) with the Greedy algorithm for variable alphabet size

String length	Superstring length		
	Greedy Algorithm	D-GREEDY	Improvement (%)
3	325-328	325	0% − 0.92%
4	377-378	377	0% − 0.26%
5	404	404	0%
6	420	420	0%
7	414	414	0%
8	432-433	432	0% − 0.23%
9	436	436	0%
10	437	437	0%
11	444-446	444	0% − 0.45%
12	448-449	448	0% − 0.22%
13	458-459	458	0% − 0.21%
14	453	453	0%
15	448-450	448	0% − 0.44%

Next, we consider the input setting where n is fixed to fifty and the alphabet size is fixed to ten. We vary the length of the strings in S from three to fifteen. The input strings are generated as follows.

$S = \{(C_1C_2 \ldots C_y)^1 \ldots (C_1C_2 \ldots C_y)^{50}\}$,

where $C_i \in \{A, B, C, \ldots, J\}$ and $y \in [3, 15]$. For example, for string length four, S could be as follows.

$S = \{JFBD, GAHI, HGFA, \ldots, EJBG\}$

The experimental results for this input setting show that our algorithm is either better than the Greedy Algorithm by an amount up to 2.19% or same as the Greedy Algorithm. See Table 5.

Finally, we consider input strings that are completely random. The strings are English words taken from some randomly chosen websites. The experimental results show that our algorithm is same or better than the Greedy Algorithm. See Table 6.

Table 5. Comparison of our algorithm (D-GREEDY) with the Greedy algorithm for variable length of input strings

String length	Superstring length		
	Greedy Algorithm	D-GREEDY	Improvement (%)
3	92-94	92	0% − 2.17%
4	137-140	137	0% − 2.19%
5	194	194	0%
6	238-239	238	0% − 0.42%
7	297-298	297	0% − 0.33%
8	345-347	345	0% − 0.29%
9	382-384	382	0% − 0.52%
10	437	437	0%
11	496-497	496	0% − 0.2%
12	536-538	536	0% − 0.37%
13	597-598	597	0% − 0.16%
14	645-646	645	0% − 0.15%
15	690-691	690	0% − 0.14%

Table 6. Comparison of our algorithm (D-GREEDY) with the Greedy algorithm for random input strings

Website link	Number of Words	Superstring length		
		Greedy Algorithm	D-GREEDY	Improvement (%)
(Web 4, 2020)	197	118	118	0%
(Web 3, 2020)	345	194	194	0%
(Web 2, 2020)	504	249	249	0%
(Web 5, 2020)	890	353	353	0%
(Web 1, 2020)	3311	723	719	0.55%

RESULT DISCUSSION

It can be observed from our examples and experiments that when the input strings are shorter and have some pattern, our algorithm can outperform the Greedy Algorithm substantially. As the input gets more random and longer strings, our algorithm and the Greedy Algorithm performs almost similar, with the Greedy Algorithm never outperforming our algorithm.

This observation aligns with that in (Tarhio et al., 1988), where the authors' observation was that as the input strings get more random and longer, the Greedy Algorithm and an optimal algorithm perform almost similar. In their experimental results (Tarhio et al., 1988), for longer input strings the average length of the superstrings generated by the Greedy Algorithm was just 1% larger than the optimal, while for shorter strings, this was 15%.

In addition, remember that it was shown experimentally by Cazaux et al. (Cazaux et al., 2018 (June)) and theoretically by Ma (Ma, 2009) that the Greedy Algorithm performs almost optimally. This indicates that any improvement over the Greedy Algorithm will be marginal. This is exactly what our experimental results show in Tables 3-6. The improvement by our algorithm in those tables are very marginal.

Another important observation is that the Greedy never outperforms our algorithm, because from the results shown in the tables the improvement done by our algorithm is never negative. Moreover, the examples presented in the comparing with existing greedy analysis section also show that our algorithm is either better than the Greedy Algorithm or same. That means, the modification in our algorithm for searching a larger overlap (and thus searching for a shorter superstring) by looking one step deeper is not harmful. Rather, it implies that either we find a better solution, or we remain same as the Greedy Algorithm.

Therefore, it can be mentioned that our algorithm performs as an improved version of the Greedy Algorithm and, therefore, closer to an optimal algorithm.

In most applications, input strings have some patterns. For example, in bioinformatics the set of alphabet consists of only four letters $\{C, G, T, A\}$, and therefore, input strings with these four letters will have some patterns. This implies that the type of input that we have considered in our experiments are reasonable from application point of view.

CONCLUSION

In this paper we have given a modified version of the Greedy Algorithm, which is a popular approximation algorithm for the shortest superstring problem. Our modification is to reduce the randomness of the Greedy Algorithm for choosing a pair of strings for merging and to apply exhaustive search to some extent. Thus, our algorithm moves closer to an optimal algorithm. To analyze and compare our algorithm with the Greedy Algorithm and an optimal algorithm, we have provided several non-trivial examples and many experimental results. Our analysis and result show that our algorithm can outperform the Greedy Algorithm substantially in many cases. Moreover, for all other cases, our algorithm is better than or same as the Greedy Algorithm. Therefore, our algorithm can be considered as a better version of the Greedy Algorithm. In future, we would like to modify the Greedy Algorithm further for better performance in terms of both the running time and the size of the superstring.

REFERENCES

Alanko, J. & Norri, T. (2017). Greedy shortest common superstring approximation in compact space. In Proceedings of the 24th International Symposium on String Processing and Information Retrieval, SPIRE 2017, volume 10508 of LNCS, Palermo, Italy, 1–13.

Blum, A., Jiang, T. Li, M., Tromp, J. & Yannakakis, M. (1994). Linear approximation of shortest superstrings. Journal of the ACM, 41(4), 630–647.

Braquelaire, T., Gasparoux, M., Raffinot, M., & Uricaru, R. (2017). On the shortest common superstring of NGS reads. In Proceedings of the 14th Annual Conference on Theory and Applications of Models of Computation, TAMC 2017, volume 10185 of LNCS, , Bern, Switzerland, 97-111.

Cazaux, B. & Rivals, E. (2016). A linear time algorithm for shortest cyclic cover of strings. Journal of DiscreteAlgorithms, 37, 56–67.

Cazaux, B., Juhel, S., & Rivals, E. (June 2018). Practical lower and upper bounds for the shortest linear superstring. In Proceedins of the 17th International Symposium on Experimental Algorithms, SEA 2018, L'Aquila, Italy, 1-14.

Cazaux, B. & Rivals, E. (July 2018). Superstrings with multiplicities. In Proceedins of the Annual Symposium on Combinatorial Pattern Matching, CPM 2018, Qingdao, China, 1-16.

Cazaux, B. & Rivals, E. (August 2018). Relationship between superstring and compression measures: New insights on the greedy conjecture. Discrete Applied Mathematics, 245(2), 59–64.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms. MIT Press, Cambridge, Massachusetts, USA, 3rd edition.

Fici, G., Kociumaka, T., Radoszewski, J., Rytter, W., & Walen, T. (2016). On the greedy algorithm for the shortest common superstring problem with reversals. Information Processing Letters, 116(3), 245-251.

Gallant, J., Maier, D., & Storer, J. A. (1980). On finding minimal length superstrings. Journal of Computer and System Sciences, 20(1), 50–58.

Garey, M. R. & Johnson, D. S. (1979). Computers and intractability: A guide to the theory of NP-completeness. Freeman, New York.

Gevezes, T. & Pitsoulis, L. (2014). The shortest superstring problem. Optimization in Science and Engineering: In Honor of the 60th Birthday of Panos M. Pardalos, Springer, New York, 189–227.

Golovnev, A., Kulikov, A. S., & Mihajlin, I. (June 2013). Approximating shortest superstring problem using de bruijn graphs. In Proceedins of the 24th Annual Symposium on Combinatorial Pattern Matching, CPM 2013, LNCS, Bad Herrenalb, Germany,

7922, 120-129.

Golovnev, A., Kulikov, A. S., & Mihajlin, I. (August 2013). Solving 3-superstring in 3n/3 time. In Proceeding of the 38th International Symposium on Mathematical Foundations of Computer Science, MFCS 2013, LNCS, Klosterneuburg, Austria, 8087, 480-491.

Golovnev, A., Kulikov, A. S., Logunov, A., Mihajlin, I., & Nikolaev, M. (2019). Collapsing superstring conjecture. In Proceedings of the 23rd RANDOM / 22nd APPROX Annual Conference, MA, USA, 1-23.

Jiang, T., Li, M., & Du, D. Z. (1992) . A note on shortest superstrings with flipping. Information Processing Letters, 44(4), 195-199.

Kaplan, H. & Shafrir, N. (2005). The greedy algorithm for shortest superstrings. Information Processing Letters, 93, 13–17.

KidsWorldFun. (Web 2 2020). Love and the time story. Retrieved from http://www.kidsworldfun.com/shortstories_loveandtime.php.

KidsWorldFun. (Web 3 2020). The cunning fox and the clever stork. Retrieved from http://www.kidsworldfun.com/shortstories_foxandstork.php.

KidsWorldFun (Web 4 2020). The goose with the golden eggs. Retrieved from http://www.kidsworldfun.com/shortstories_goosewithgoldeneggs.php.

KidsWorldFun (Web 5 2020). The lion and the mouse. Retrieved from http://www.kidsworldfun.com/shortstories_lionandmouse.php.

Kulikov, A. S., Savinov, S., & Sluzhaev, E. (2015). Greedy conjecture for strings of length 4. In Proceedings of 26th Annual Symposium on Combinatorial Pattern Matching, CPM 2015, Ischia Island, Italy, 9133, 307–315.

Ma, B. (2009). Why greed works for shortest common superstring problem. Theoretical Computer Science, 410, 5374-5381.

Maier, D. & Storer, J. A. (1977). A note on the complexity of the superstring problem. Technical Report 233, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ, USA.

Mucha, M. (2013). Lyndon words and short superstrings. In Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, USA, 958–972.

Nabi. T. un. & Sen, A. (2012). On the approximability of greedy shortest superstring. Bachelor's Thesis, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh.

Romero, H. J., Brizuela, C. A., & Tchernykh, A. (2004). An experimental comparison of approximation algorithms for the shortest common superstring problem. In Proceedins of the 5th Mexican International Conference on Computer Science, Colima, Mexico, 27-34.

Steel, F. A. (Web 1 2020). Jack and the Beanstalk. Retrieved from https://americanliterature.com/childrens-stories/jack-and-the-beanstalk.

Storer, J. A. & Szymanski, T. G. (1982). Data compression via textual substitution. Journal of the ACM, 29(4), 928-951.

Tarhio, J. & Ukkonen, E. (1988). A greedy approximation algorithm for constructing shortest common superstrings. Theoretical Computer Science, 57, 131–145.

Turner, J. S. (1989). Approximation algorithms for the shortest common superstring problem. Information and Computation, 83(1), 1–20.

Weinard, M. & Schnitger, G. (2006). On the greedy superstring conjecture. SIAM Journal on Discrete Mathematics, 20(2), 502–522.