CS301 – Term Project

Project Report

Group Members:

Abdullah Said Uyar

Ali Erencan Gerçek

Ahmet Can Yıldız

Berk Ozer

Semester: 2020 Fall

Instructor: Hüsnü Yenigün

Faculty of Engineering and Natural Sciences

Sabancı Üniversitesi

1) PROBLEM DESCRIPTION

Shortest Common Superstring Problem is a problem where let’s say we have a set of strings S.

S = {string1, string2, string3}. The Shortest Common Superstring Tries to create the shortest string that contains every string in S.

To Give an example, let S contain strings {0001, 01010, 00101}. In Shortest Common Superstring to create the shortest string we need to check the overlapping parts of the strings.

0001, 01010, 00101.

0**001** and **001**01 overlap on characters 001 therefore we can write the string as 000101.

Then we check 00**0101** with **0101**0

Characters 0101 overlaps each other therefore we can write the string as 0001010

From the 0001010 string we can create all substrings **0001**-010, 00-**01010**, 0-**00101**-0.

Not every string created by Shortest Common Superstring is the shortest as changing orders of the strings when adding strings together differentiates the lengths of strings. There are a possible n! number of strings created by Shortest Common Superstring n being the size of S.

There is also Greedy Shortest Common Superstring that can be used to create the Superstring.

The difference between normal and greedy is that greedy only selects the highest count of overlaps when creating the superstring whereas normal can go with any order of strings.

One of the most prominent uses of The Shortest Common Superstring is genetics.In genetics Shortest Common Superstring is used to Sequence DNA in a method called Shotgun sequencing. In Shotgun sequencing The DNA Fragments are compared with each other to find similar nucleobases in the same pattern if the part of a pattern matches that of another Fragment then those fragments are combined. This process repeats till the whole DNA sequence is found.

See figure 1.

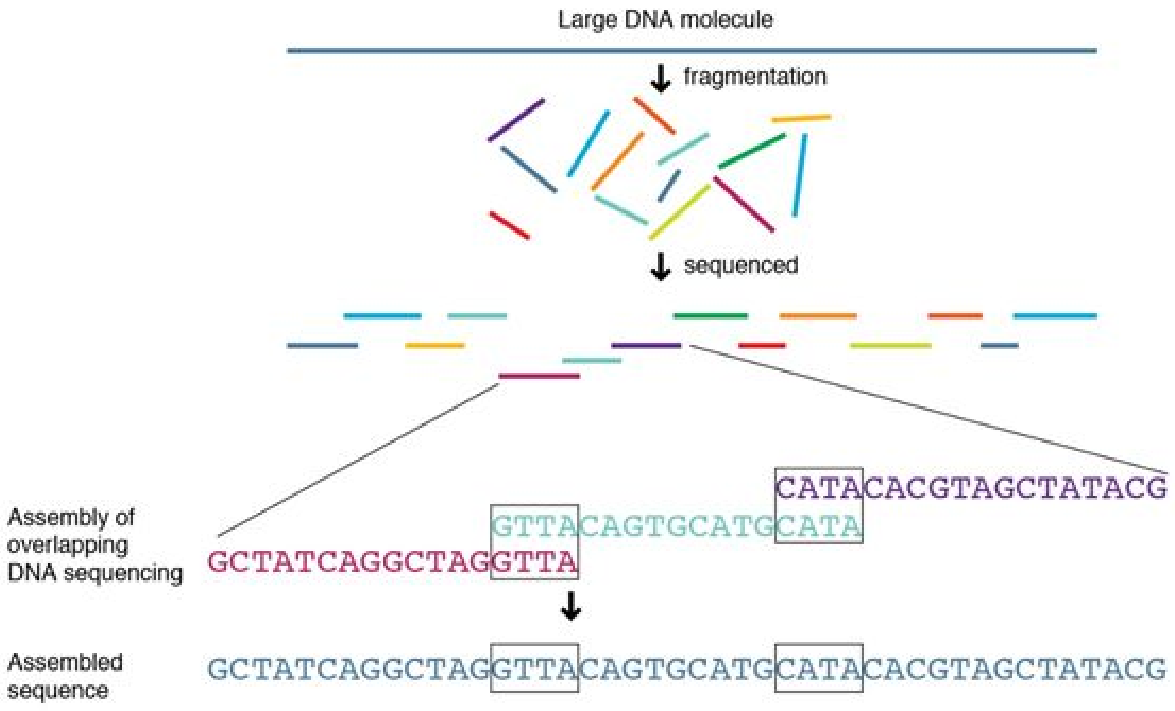


Figure 1.

Theorem. *The SCS problem is* NP-*complete for an alphabet* Σ *of size* $\geqslant 2$.

Proof. The SCS problem is clearly in class NP. To prove the completeness we must therefore give a polynomial time transformation from some known NP-complete problem to the SCS problem over binary alphabet. The transformation we give will be, as in [5], from the *node cover problem* [2, 4]. Given a graph $G = (N, E)$ and an integer k, the node cover problem is to determine if there is a subset $N' \subseteq N$ such that N' has at most k elements and, for each edge $(x, y) \in E$, at least one of x and y belongs to N'.

Let $G = (N, E)$ and k constitute an instance of the node cover problem where $N = \{v_1, v_2, \ldots, v_t\}$, $E = \{(x_1, y_1), (x_2, y_2), \ldots, (x_r, y_r)\}$. We construct a set R of $r+1$ strings over the binary alphabet $\Sigma = \{0, 1\}$. Basically, our construction is a simplified version of the transformation used in [5] to prove the result of the theorem for alphabet size $\geqslant 5$.

The first string in R is the *template* T. In addition, R contains a string S_i for each edge $e_i = (x_i, y_i)$ in E. In these strings, the nodes and edges of G are encoded using the alphabet $\{0, 1\}$. We first describe the encoding, shown in Fig. 1. The *node codeplate* $\bar{N}$ is defined as $t+1$ blocks of $7c$ ones, where $c = \max(r, t)$. *Any* v_i in N we encode with *node code* $\bar{N}[i]$ which is obtained by inserting a zero between the ith and $(i+1)$st blocks of $\bar{N}$. The *multiple node code* $\bar{N}[i_1, i_2, \ldots, i_s]$ has a zero in the i_1st, i_2nd, $\ldots$, and i_sth spots. The special case of $\bar{N}[1, 2, \ldots, t]$ will be denoted by $\bar{N}_s$ and referred to as the *node sink*, since it is a supersequence of all the node codes. The *edge codeplate* $\bar{E}$, the *edge code* $\bar{E}[j]$, and the *multiple edge code* $\bar{E}[j_1, j_2, \ldots, j_s]$ are defined similarly with blocks of $7c$ zeros and *pairs* of ones. (Only the code $\bar{E}[j]$ is shown in Fig. 1.) The code $\bar{E}[1, 2, \ldots, r]$ is called the *edge sink* and denoted by $\bar{E}_s$.

Now we can define the $r+1$ strings of R, shown in Fig. 2. The template T consists of the following codes in the given order: $\bar{E}; \bar{N}_s; \bar{E}_s; \bar{N}; \bar{E}_s; \bar{N}_s; \bar{E}$. We denote the length of T by $q = 7c(4r+3t+7)+4r+2t$. For each $e_i = (x_i, y_i)$ we define S_i as: $\bar{E}[i]; \bar{N}[j]; \bar{N}[m]; \bar{E}[i]$, where j and m are such that $x_i = v_j$, $y_i = v_m$. To distinguish

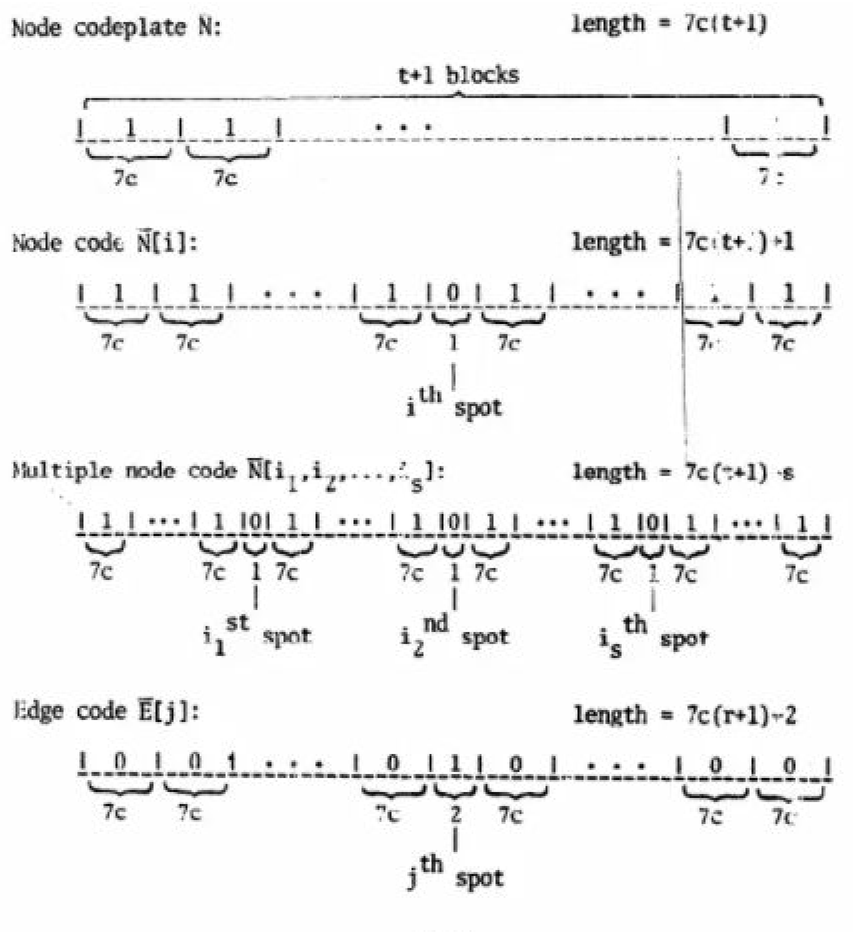


Fig. 1.

between the left and right occurrences of the same code in a string we use superscripts L and R.

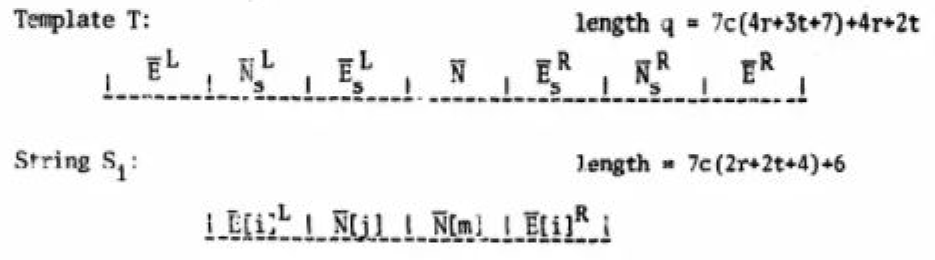


Fig 2.

By proving the following two claims we show that the above transformation from the node cover problem to the SCS problem has the desired properties.

Proof that shortest common superstring problem is NP-complete taken from references[6].

2) Algorithm Description

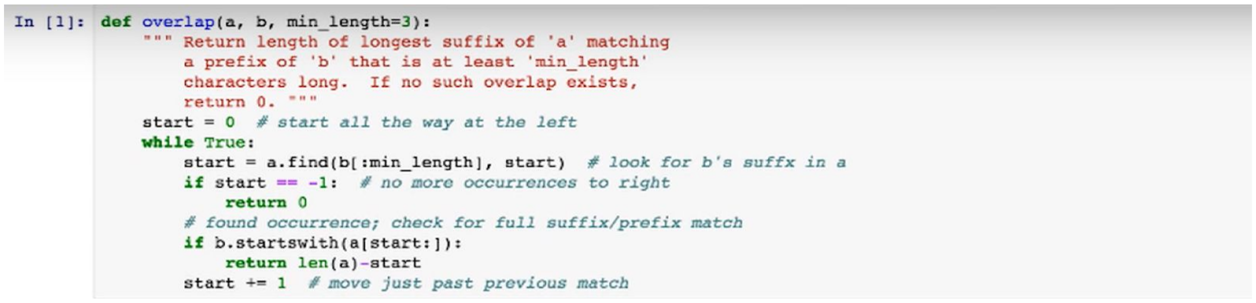
There is no such algorithm that solves the problem accurately in polynomial time. That' s why this problem is NP-complete. In addition, generally no NP-complete problems have a proper solution. This does not mean that NP-complete problems are insoluble.

There are such algorithms to solve the shortest common superstring problem that give intended output. But their running time is changeable according to input sizes. Greedy algorithm is the best way to solve the problem. In that sense, a heuristic algorithm to solve the shortest common superstring is a greedy algorithm.

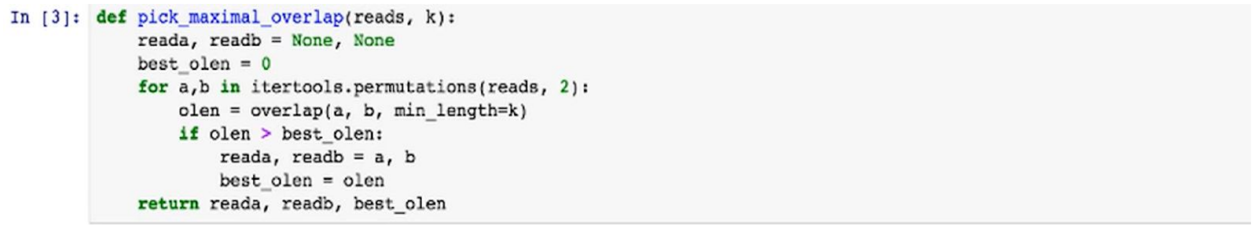
Below are the steps of application of greedy algorithm:

1. Find two inputs with the maximum overlap.
2. Take the first string and merge with the non-overlapping part of the second string.
3. Consider the re-created string as a reference string and compare it with the remaining strings of the input set.
4. Repeat it until the unmerged input remains.

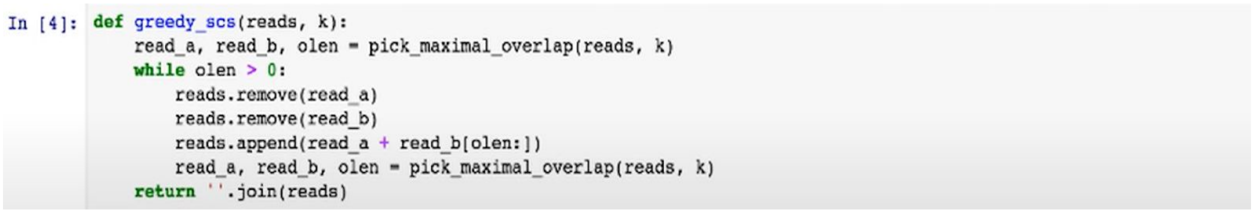
Here is the code of the algorithm:



```
In [1]: def overlap(a, b, min_length=3):
            """ Return length of longest suffix of 'a' matching
                a prefix of 'b' that is at least 'min_length'
                characters long.  If no such overlap exists,
                return 0. """
            start = 0  # start all the way at the left
            while True:
                start = a.find(b[:min_length], start)  # look for b's suffx in a
                if start == -1:  # no more occurrences to right
                    return 0
                # found occurrence; check for full suffix/prefix match
                if b.startswith(a[start:]):
                    return len(a)-start
                start += 1  # move just past previous match
```



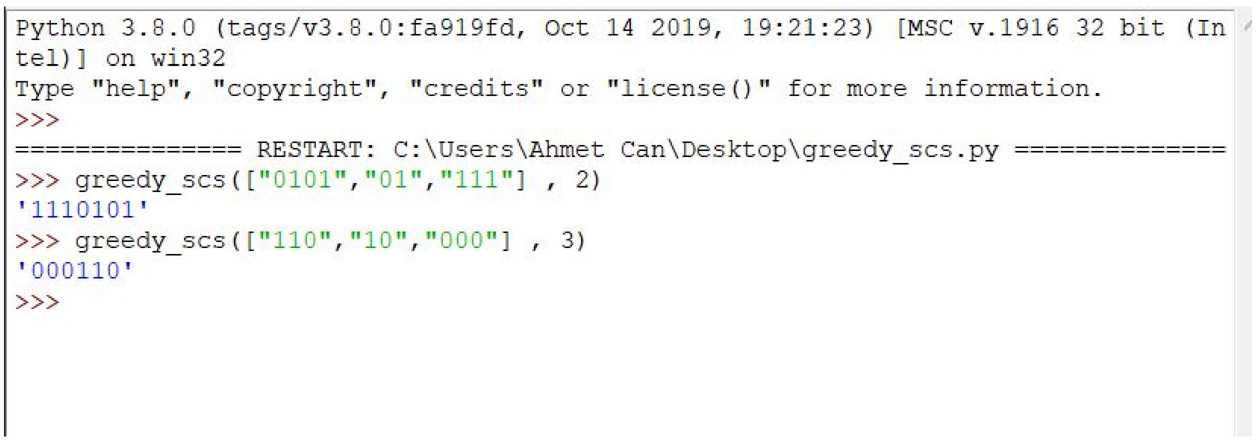
```
In [3]: def pick_maximal_overlap(reads, k):
            reada, readb = None, None
            best_olen = 0
            for a,b in itertools.permutations(reads, 2):
                olen = overlap(a, b, min_length=k)
                if olen > best_olen:
                    reada, readb = a, b
                    best_olen = olen
            return reada, readb, best_olen
```



```
In [4]: def greedy_scs(reads, k):
            read_a, read_b, olen = pick_maximal_overlap(reads, k)
            while olen > 0:
                reads.remove(read_a)
                reads.remove(read_b)
                reads.append(read_a + read_b[olen:])
                read_a, read_b, olen = pick_maximal_overlap(reads, k)
            return ''.join(reads)
```

Codes taken from references[4].

Since the problem is NP-complete, we cannot achieve the exact correct algorithm. However, according to the research we made, greedy algorithms provide optimal results in terms of running time and correctness. Below some test cases we made and the results:



```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=============== RESTART: C:\Users\Ahmet Can\Desktop\greedy_scs.py ==============
>>> greedy_scs(["0101","01","111"] , 2)
'1110101'
>>> greedy_scs(["110","10","000"] , 3)
'000110'
>>>
```

As seen above, our algorithm gives correct results. We will examine the running time of the algorithm in the next steps.

Ratio Bound Approximation:

Greedy Algorithm for Shortest Common Superstring:

Pseudocode: [1,2]

Let arr[] be given a set of binary strings.

1) Create an auxiliary array of strings, temp[]. Copy contents

   of arr[] to temp[]

2) While temp[] contains more than one binary strings

a) Find the most overlapping binary string pair in temp[]. Let this pair be “a” and “b”.

b) Replace “a” and “b” with the binary string obtained after combining them.

3) The only binary string left in temp[] is the result, return it.

Let we think some set such that:

arr[1(0^k), (0^k)1, 0^(k+1)]

For example, if k = 3, arr = [1000, 0001, 0000]

With respect to greedy approach above, we can obtain by combining first and two binary strings,

At first step temp = [10001, 0000]

At Second step, we obtain temp = [100010000] with length of 9 bits.

But actual common superstring must be [100001] with length 6 bits.

So, with this approach, we can derive the ratio bound with k:

arr = [1(0^k), (0^k)1, 0^(k+1)]

At first step we can combine 1(0^k) with (0^k)1, then temp = [1(0^k)1, 0^(k+1)]

At second step, combine 1(0^k)1 with 0^(k+1), then temp = [1(0^k)1(0^(k+1)]

So our shortest common superstring using Greedy Approach, will be 1(0^k)1(0^(k+1) with length (2k+3)

But actual shortest common superstring must be 1(0^k)01 with length (k+3)

So ratio bound will be (2k+3)/(k+3),

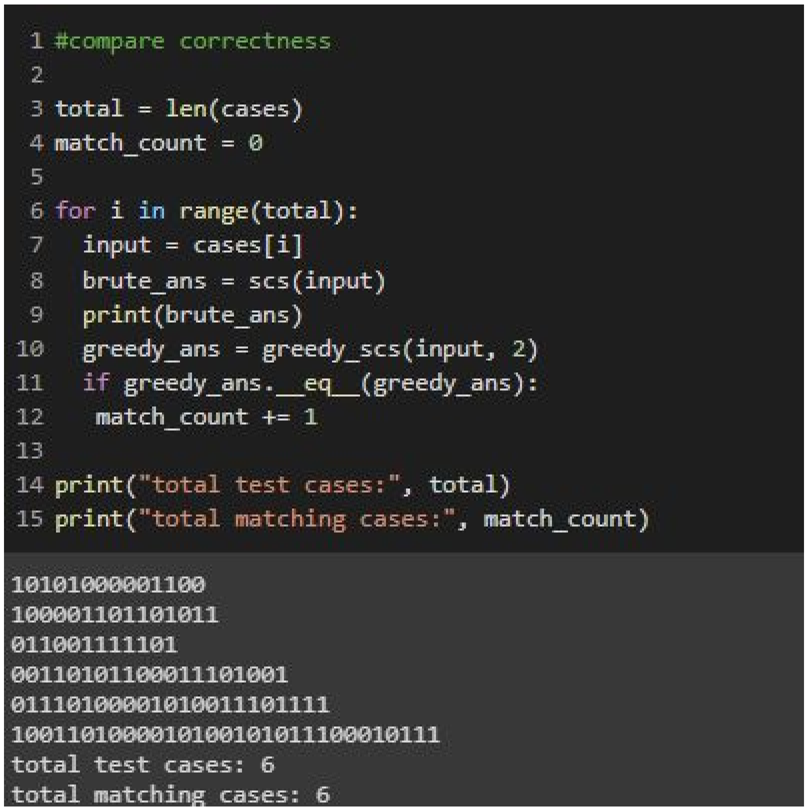
We can find the limit of: $\lim_{k\to\infty}(2k+3)/(k+3)$ so with this greedy approach we guaranteed that,

(SCS Length with Greedy Approach) ≤ 2×(Optimal SCS Length)

Space Complexity:

Let arr[…] contain N number of binary strings. We have to copy each element to constitute a temp[…] array. Other operations after copy does not increase the size of temp. Basically this algorithm works in a temp array with n number of binary strings and temporarily stores the N extra elements. Thus, we need N extra space to find an approximation of shortest common superstring using the greedy algorithm. Overall, *space complexity is* θ *(n)* .

Experiment Correctness



```python
#compare correctness

total = len(cases)
match_count = 0

for i in range(total):
  input = cases[i]
  brute_ans = scs(input)
  print(brute_ans)
  greedy_ans = greedy_scs(input, 2)
  if greedy_ans.__eq__(greedy_ans):
   match_count += 1

print("total test cases:", total)
print("total matching cases:", match_count)
```

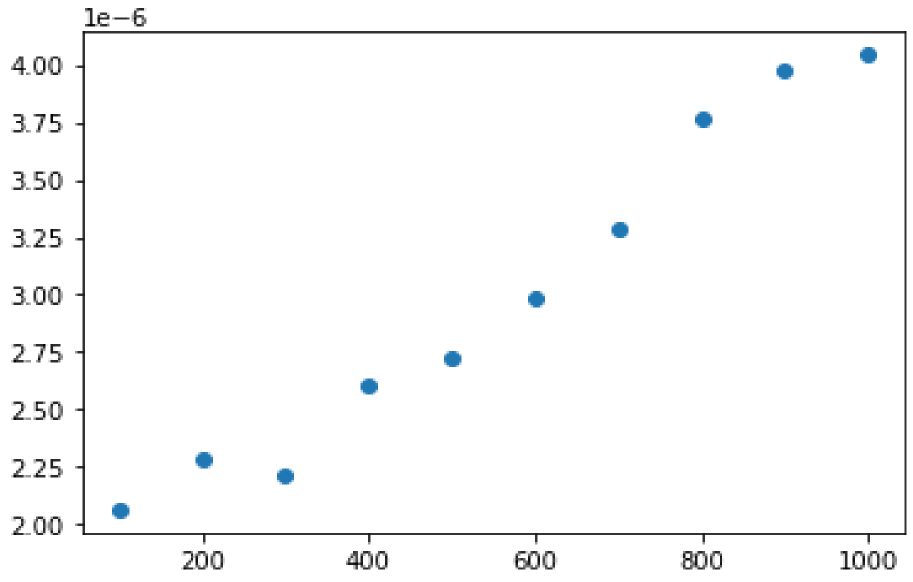
```
10101000001100
100001101101011
011001111101
00110101100011101001
01110100001010011101111
1001101000010100101011100010111
total test cases: 6
total matching cases: 6
```

Running time experimental analysis for Greedy and Brute force:

Running time for Greedy Algorithm:

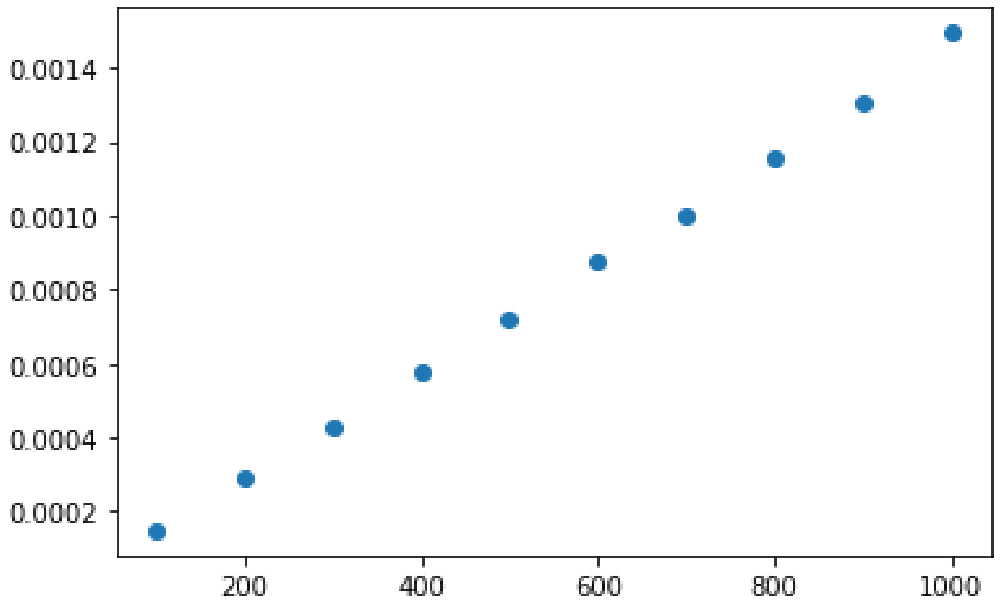
Size	**Mean Time(ms)**	**Standard Deviation**
100	2.0647048950195313e-06	2.442183780537199e-06
200	2.281665802001953e-06	2.568788423128457e-06

300	2.2149085998535154e-06	3.150284510787633e-07
400	2.601146697998047e-06	1.1680539607206632e-06
500	2.727508544921875e-06	3.2396865370679917e-07
600	2.9897689819335935e-06	3.806044779359317e-07
700	3.2901763916015627e-06	8.436110502698012e-07
800	3.7693977355957032e-06	1.088763965504647e-06
900	3.9768218994140624e-06	1.4300663573438144e-06
1000	4.0435791015625e-06	7.747099803797259e-07



Running time for Brute force algorithm:

Size	**Mean Time(ms)**	**Standard Deviation**
100	0.0001482844352722168	2.1469679095716947e-05
200	0.00028818368911743164	3.9679038715588885e-05
300	0.0004290485382080078	4.2791948405918826e-05
400	0.0005766320228576661	2.849328967755924e-05
500	0.0007223010063171387	9.328429305564978e-05
600	0.0008752059936523438	0.00010165816103257168
700	0.0009967517852783204	1.9903193264982803e-05
800	0.0011546683311462402	7.321000676661853e-05
900	0.0013098621368408203	9.408296763022627e-05
1000	0.001495969295501709	0.0002524992071181248



As is seen, when size is 100, running time for greedy and brute force algorithms are approximately 2x10^-6 and 1x10^-4 respectively. Running time of the brute force algorithm is 100 times greater than the greedy algorithm. When we increase the size to 1000, running time for greedy and brute force algorithms are approximately 4x10^-6 and 1x10^-3 respectively. So, Running time of the brute force algorithm is 4000 times greater than the greedy algorithm.

Discussion

As a result, it is proven that the Shortest Common Superstring problem is NP-complete. There is no algorithm that solves this problem in polynomial time. However, as we noticed, greedy algorithm gives the best results. Approximation algorithm which we tried at the first step to solve the problem, gave dramatically high run times. Because it runs in factorial time to compare all inputs. As a result of it, in large input sizes, the approximation algorithm does not work properly. When we tried to examine greedy algorithm, we noticed that it does not guarantee the correct result but at least, it runs much faster than other algorithms that are designed to solve the Shortest Common Superstring problem.

References:

[1]https://www.genome.gov/sites/default/files/tg/en/illustration/shotgun_sequencing.jpg

[2] http://fileadmin.cs.lth.se/cs/Personal/Andrzej_Lingas/superstring.pdf

[3] http://math.mit.edu/~goemans/18434S06/superstring-lele.pdf

[4]https://www.youtube.com/watch?v=uS6ca7yeVb0&list=PL2mpR0RYFQsBiCWVJSvVAO3OJ2t7DzoHA&index=48

[5]https://www.academia.edu/6123020/The_Shortest_Common_Supersequence_Problem_over_Binary_Alphabet_is_NP_Complete

[6] Räihä, K., & Ukkonen, E. (1981). The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science, 16*(2), 189-190. doi:10.1016/0304-3975(81)90075-x

[6.1]D. IGaier, The complexity of some problems on subserquences and supersequences, J. zyxwvutsrqponmlkjih CM :tS (2)(1978) 322-336.