



Fall 2019

CS301 - Algorithms

Project Report

Shortest Common Superstring

Aras Bozkurt, Buse Çarık, Deren Ege Turan, Mücahit Umut Onat

1. Problem Description

The shortest common superstring problem (SCS) is in a given collection of input strings, finding the shortest possible string that contains all strings in the set as a substring.

In a formal way, firstly, the superstring is defined as follows: given a string S over an alphabet Σ , a superstring S' of S to be any string $S' = w_0 x_1 w_1 x_2 w_2 \dots x_k w_k$ over Σ such that $S = x_1 x_2 \dots x_k$ and each $w_i \in \Sigma^*$. And a common superstring of a collection of strings $R = \{r_1, r_2, \dots, r_m\}$ is a string S over the alphabet Σ such that S is the superstring of each string r_i in the set R . SCS can be defined as given an alphabet Σ , positive integer k , and a finite set of strings $R = \{r_1, r_2, \dots, r_m\}$ from the alphabet Σ^* , if present, the shortest common superstring of R of length less than or equal to k .

The applications of the SCS are data compression and genome assembly, which divides DNA sequence into fragments and finds overlaps between fragments to represent it shorter. Then, it can be used to reproduce the sequence in the future.

In this project, instead of a finite alphabet, strings are composed of the binary alphabet. Hence, our problem is stated as below;

For a finite set $R=\{r_1, r_2, \dots, r_m\}$ of *binary strings* (sequence of 0 and 1), is there a binary string w of length at most positive integer k such that every string in R is a substring of w i.e. for each r in R , w can be decomposed as $w=w_0rw_1$ where w_0, w_1 are (possibly empty) binary strings?

$$S = \{011, 100, 001, 010\}$$


$$w = 01011001$$


Fig.1. Example of shortest common superstring

The shortest common superstring problem is NP-complete for an alphabet $\Sigma=\{0,1\}$.

Proof.

A problem is NP-complete if

1. The problem must be in NP.
2. The problem is reduced to an NP-complete problem in polynomial time.

For the first part,

- 1) A polynomial-time algorithm for this problem has not been discovered. And, to prove that this problem is in NP, given a set of strings S , our verifier picks a superstring and checks whether each string in the S is present in this superstring in $O(n^2)$ time.
 - 2) A reduction from the Maximum Asymmetric Travelling Salesman, which is an NP-complete problem, to SCS in polynomial-time proves that SCS is an NP-hard problem.
- Suppose we have a set of string $S = \{s_1, s_2, \dots, s_m\}$ of the binary strings and a positive integer k .
 - Let's construct a directed graph $G=(V, E)$ with a set of vertices V and a set of edges E . The set of V has a vertex corresponding to each string in the set S .
 - Then, we define an $\text{overlap}(s_i, s_j)$ that is the length of the longest prefix of s_j that matches a suffix of s_i . (For example, assume s_i is 101 and s_j is 010, the overlap become 1010 since the first three digits correspond to the s_i and the last three digits correspond to the s_j)
 - The weight of the directed edge from s_i to s_j is the value of $\text{overlap}(s_i, s_j)$. The weight denotes the number of common digits if that edge will be preferred.
 - In Fig 2, a set of four elements is shown in a graphically and the weight of edges is calculated by the above statement.

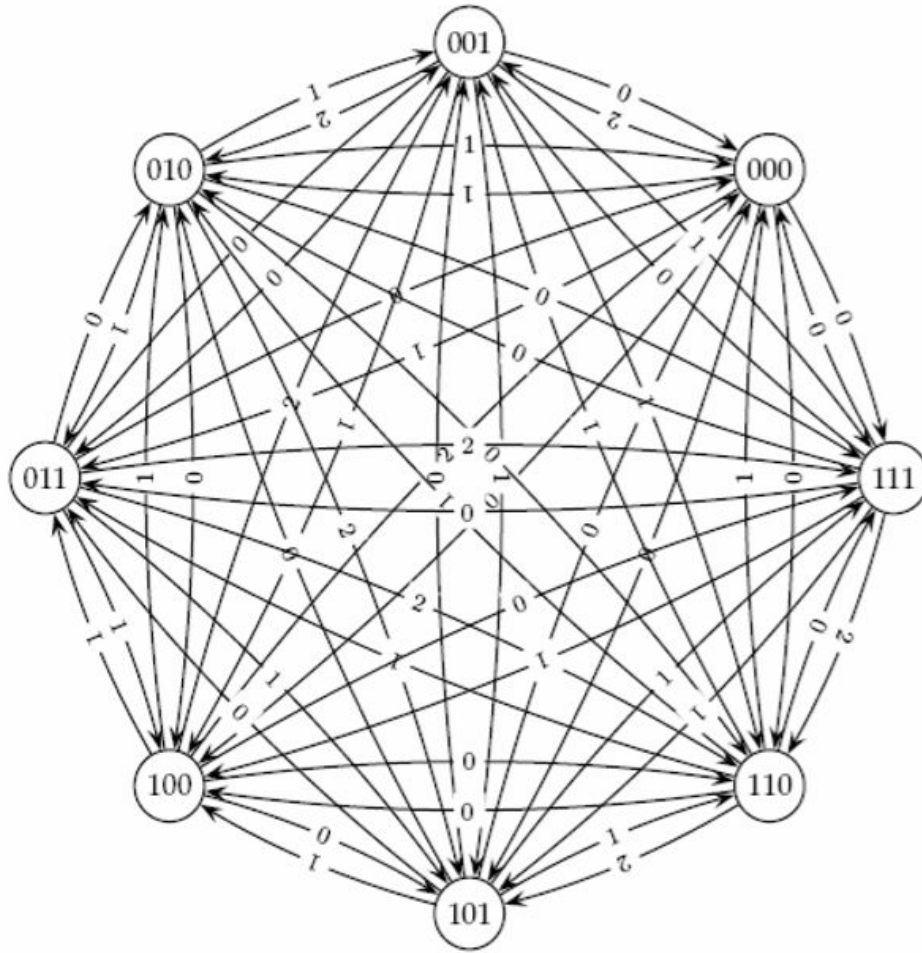


Fig.2. ATSP instance for the Shortest Common Superstring instance

- To find the shortest common superstring, the acyclic maximum weighted path is found which travels each vertex exactly once. The length of the shortest common superstring is subtraction to the total weight of the path from the total length of all strings.
- Maximum Asymmetric Traveling Salesman Problem (MaxATSP) asks the searched the shortest maximum weighted path that every vertex is visited exactly once.

Therefore the above statement might be expressed as an MaxATSP. However, there is a difference between SCS and MaxATSP problems, which is in the MaxATSP, the

path must end in the starting vertex, but in the SCS, there is no need to return to the starting vertex. However, this difference does not cause a differentiation from MaxATSP since MaxATSP considers all Hamiltonian circuits, which requires to check all permutations, and even though the SCS problem does not return to the starting point, it also needs to check all permutations to find the shortest path which correspondence to shortest common superstring.

As a result, SCS can be reduced to the Maximum Asymmetric Traveling Salesman Problem in polynomial time, indicating that the SCS problem is NP-hard. Also, we showed that the SCS is also in NP, so SCS is NP-complete.

2. Algorithm Description

The Shortest Common Superstring problem is an NP-Complete problem for array size larger than 2. Since its running time is $O(n! + mn^2)$, instead of using a brute force method, we used a heuristic solution to solve it in polynomial time, which is a greedy approach. It finds $\frac{1}{2}$ approximation for SCS problem.

What a greedy algorithm does is to select the most optimal choice for that instance as it attempts to find an overall optimal solution for the entire problem. Unfortunately for some cases it does not lead to the optimal solution.

In addition, it should be noted that, to a greedy algorithm work more accurately and efficiently for a given set of strings, it should be provided that none of the strings is a substring of another string on that list.

The simple implementation of the algorithm as follows:

- 1) First, from the given finite string set R , which consist of binary strings, `Eliminate_Substr` function checks and if necessary eliminate the strings in order to regulate the list as the algorithm would work more accurately. The determination process of this function is ; if any pairs exists on the list ($r_1, r_2 \in R$) such that one of the is the substring of another ($r_1 \subseteq r_2$); the algorithm removes that (r_1) substring from the list.
- 2) Then it calculates the sum of all string lengths on the remaining list. This number indicates if there were no overlap between any of the strings, this would be the length of resulted shortest superstring.
- 3) To solve the rest of SCSS problem, it is reduced to a graph for simplicity. First all strings are transformed to nodes. Then the directed edges are drawn for all the possibilities. To load the weights to the edges, all permutations of 2 of the list are extracted. The purpose of this operation is to compute overlap character count pairwise for both sides (r_1r_2 and r_2r_1). Then concatenate the pairs with the least possible total length. To make this happen, the suffix of the first string and the prefix of the second string are matched with the longest length possible. This length is weight of one edge. It is placed on an edge which originated from the beginning string in the concatenation and ends with the ending string in the concatenation. When this process is repeated for all permutations by 2. All edges of the created graph are acquired. Finally just remove the edges with 0 length. At the end with this edges, this problem can be solved same as MaxATSP.

- 4) By sending edges that are found in the previous step to the greedy algorithm (SCSS function) this article based on, a common superstring for R will be found. Ideally it is supposed to be the shortest, but since greedy algorithm does not always guaranteed to produce the optimal solution, it is only hoped to be the shortest superstring. The summary of SCSS function is to take the highest weighted edge in the edges, combine the nodes according to that directed edge and re-form the edges list by the choice of that step. Repeats this method until no edges is remained. It means all possible compositions are made. The formed path/paths total weight/s will be calculated and returned.
- 5) SCSS function will return number of length that the total weight of the graph's path solved by MaxATSM problem with greedy algorithm. This number is the number of characters which are overlapped in our solution for shortest common superstring.
- 6) Finally, when the total overlapped character count(calculated at step 5) is subtracted from Total Strings length (calculated at step 2), the result will give the length of the shortest common superstring of the strings $\in R$.
- 7) Lastly, since it is a decision problem the result found above is compared with the given k value as an input. if k is equal or larger, then the algorithm returns true, otherwise false.

```
def Check(Str_Set,k):
    Eliminate_Substr(Str_Set)
    Total_Len = 0
    for st in Strings:
        Total_Len += len(st)
    E = FindAllOverlaps(Strings)
    SCSS_len = Total_Len - SCSS(E)
    if k >= SCSS_len: return True
    else: return False
```

3. Algorithm Analysis

As it is mentioned above in the “Algorithm Description” section of this article, greedy algorithm on shortest common superstring problem is a heuristic solution and it does not always gives the most optimal solution. On the other hand it solves the problem in polynomial time in the exchange of accuracy. Approximation ratio of greedy algorithm for shortest common superstring is at most 3.5. To check the accuracy of greedy approach, the results should be compared with brute force method which basically just concatenates every possible permutation of the given string set back to back while comparing them by their overlapped suffix-prefix pairs. After checking every scenario, returns the exact shortest common superstring. On “Experimental Analysis” section, we have planned to compute our greedy approach’s success rate by comparing the brute force method. Sadly since its running time is $O(n!)$, it was taking too much time, so we could not manage to measure it.

Blum et al.(1994) proved that the solution of greedy method can not be larger than the exact solution multiplied by 4. Also more recently, Kaplan et al.(2005) proved that the length of string produced by GREEDY is in fact within a factor of 3.5 from the optimal string. Although Blum et al. and many others conjectured that the approximation guarantee of GREEDY is 2, no one could actually manage to narrow the gap regarding the approximation guarantee of GREEDY. But still no one could also give an counter-example which it is larger than 2, so until the against is proved; we may make an assumption that if given k is smaller than the greedy result divided by 2, we can immediately reject it. Still unfortunately this has not been proved yet!

There are also a lot of other approximation algorithms are developed in years. As this is an optimization problem, it is measured by 2 different aspects; The compression indicates the difference between the length of the superstring and the sum of lengths of the given strings. On the other hand superstring indicates the length of the resulting string. So far the best obtained upper bound results for the problem are; $\frac{3}{4}$ for compression measure by Paluch et. al. (2014) and $2+\frac{11}{23}$ for the superstring length measure by Mucha et. al.(2013).

Running time complexity analyses are show on the code below:

```
def Compress2strings(ind,edges): #Complexity of this func is O(n^2) since the len
    a = edges[ind][0]           of edges might be at most n^2 for worst case
    b = edges[ind][1]
    i = len(edges)-1
    while i != -1:
        if edges[i][0] == a:     #Remove edges start with a
            del edges[i]
        elif edges[i][1] == b:   #Remove edges end with b
            del edges[i]
        elif edges[i][0] == b:   #Edges which which b goes are now goes from X
            if edges[i][1] == a:
                del edges[i]
            else:
                edges[i][0] = a
        i = i-1                  #Edges which goes to a, now goes to this new X
    return

def overlap(a,b): #return length of longest suffix of a which matches prefix of b
    start = 0       #Complexity of this func is O(m) where m is max string length
    while True:     of the array
        start = a.find(b[0],start)
        if start == -1:
            return 0
        if b.startswith(a[start:]):
            return len(a)-start
        start += 1
```

```

def FindAllOverlaps(Set):
    #Complexity of this func is O(mn^2)
    Edges= []
    for a,b in permutations(range(len(Set)),2): #O(n^2)
        W = overlap(Set[a],Set[b]) #O(m)
        if W > 0:
            Edges.append([a,b,W])
    return Edges

def SCSS(Edges): #Greedy Algorithm for finding shortest common superstring
    Total_Path_Weight = 0 #Complexity of this func is O(n^3) = O(n)*(O(n^2)+O(n^2))
    while (len(Edges) != 0): #O(n)
        maxWeight = 0
        index = -1
        for E in range (len(Edges)): #Find Longest Weight & its index #O(n^2)
            if Edges[E][2] > maxWeight:
                maxWeight = Edges[E][2]
                index = E
        Total_Path_Weight += maxWeight
        Compress2strings(index,Edges) #O(n^2)
    return Total_Path_Weight

def Eliminate_Substr(SS): #Complexity of this func is O(m*n^2)
    i=0
    while i != len(SS): #O(n)
        t = i+1
        while t != len(SS): #O(n)
            if(SS[t] in SS[i]): #O(m)
                del SS[t]
            elif(SS[i] in SS[t]):
                del SS[i]
                i -= 1
                break
            else: t += 1
        i += 1
    return

```

```

def Check(Str_Set,k):          #O(mn^2) + O(n) + O(mn^2) + O(n^3) = O(n^3 + mn^2)
    Eliminate_Substr(Str_Set) #Eliminate substrings on the Set      #O(mn^2)
    Total_Len = 0
    for st in Strings:         #O(n)
        Total_Len += len(st)
    E = FindAllOverlaps(Strings) #O(mn^2)
    SCSS_len = Total_Len - SCSS(E) #O(n^3)
    if k >= SCSS_len:
        return True
    return False

```

So as it is shown above for the worst case, the overall running time complexity of our algorithm is $O(n^3 + mn^2)$, where n is our array size and m is the max length of string within this array.

4. Experimental Analysis

Experimental analysis consists of

- Running Time Analysis
- Determining Successful Superstring Size for k
- Determining Successful Superstring Size for k with Array Size=20 and

Changing String Length Intervals

For the first two cases the strings are generated randomly for each run and when array size incrementation effect is observed, the string length was fixed for all the inputs. When the effect of string length is observed, the array size was fixed for all the runs. However, in the last case, every array contained 20 random strings (fixed array size) and the string sizes were

randomly chosen in the intervals such as [0,5], [5,10], [10,15] ... [95,100] in order to see the effect of input strings with different string lengths on the output superstring length.

Running Time Analysis

For the running time analysis, there are 2 main cases.

1. In the first part of the experiments, the running time is observed when array size is increased with fixed size of string lengths 10.
2. In the second part of the experiments, the running time is observed when the string lengths are increased while the array size is fixed at 20.

The statistics used in the running time code are as follows:

```

def standardDeviation(results):
    sum = 0
    mean = 0
    standard_deviation = 0
    for i in range(len(results)):
        sum += results[i]

    mean = sum / len(results)

    for j in range(len(results)):
        standard_deviation += pow(results[j] - mean, 2)

    standard_deviation = math.sqrt(standard_deviation / len(results))
    return standard_deviation

def standardError(standard_deviation, n):
    return standard_deviation / math.sqrt(n)

def runningTime(running_times):
    totalTime = 0
    for i in range(len(running_times)):
        totalTime += running_times[i]

    standard_dev = standardDeviation(running_times)
    N = len(running_times)
    m = totalTime / N
    t_value_90 = 1.660
    t_value_95 = 1.984
    standard_error = standardError(standard_dev, N)
    upper_mean_90 = m + t_value_90 * standard_error
    lower_mean_90 = m - t_value_90 * standard_error
    upper_mean_95 = m + t_value_95 * standard_error
    lower_mean_95 = m - t_value_95 * standard_error
    return [m, standard_dev, standard_error, lower_mean_90, upper_mean_90, lower_mean_95, upper_mean_95]

```

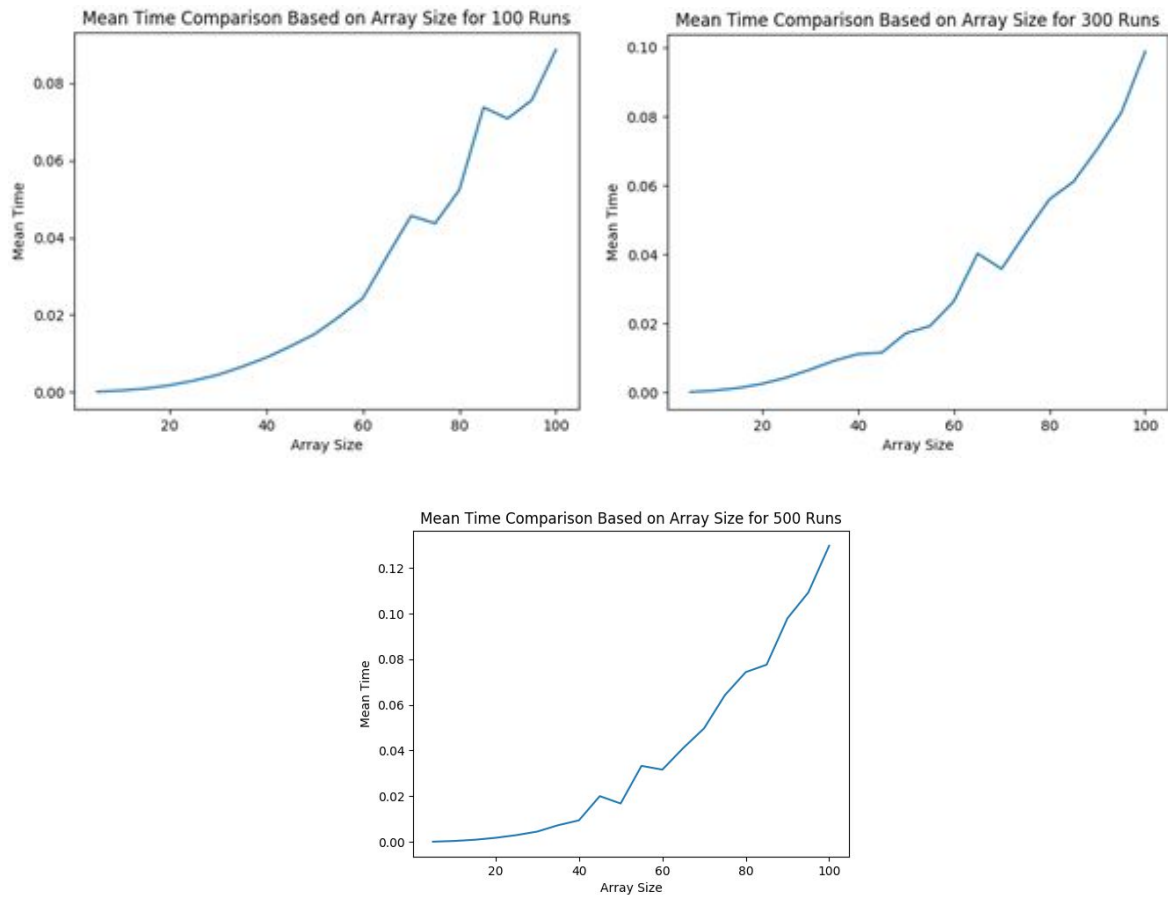
The code is annotated with mathematical formulas:

- The `standardDeviation` function is linked to the formula: $SD = \sqrt{\frac{\sum |x - \bar{x}|^2}{n}}$
- The `standardError` function is linked to the formula: $SE = \frac{\sigma}{\sqrt{n}}$
- The calculations for confidence intervals in the `runningTime` function are linked to the formula: $\bar{x} \pm z \frac{s}{\sqrt{n}}$

The t values are selected as 1.660 for 90% confidence interval and 1.984 for 95% confidence interval since after 100 samples t values are approximately the same. By using this function, the true mean will be in the given intervals with given confidence levels.

1-Running Time vs. Array Size

In these experiments there are 100, 300, and 500 runs for each of the array size incrementation in the range [5,100] with increment size = 5. All strings had 10 as their length.



Mean Time Based on Array Size for 100 Runs

Array Size	Mean Time	Standard Deviation	Standard Error	90% Confidence Level	95% Confidence Level
5	0.000109732151031	7.72883571243e-06	7.72883571243e-07	0.00011-0.00011	0.00011-0.00011
10	0.000430021280011	3.75409645915e-05	3.75409645915e-06	0.00042-0.00044	0.00042-0.00044
15	0.000934708118439	6.58746036189e-05	6.58746036189e-06	0.00092-0.00093	0.00092-0.00093
20	0.00179042577744	0.000212066522147	2.12066522147e-05	0.00176-0.00183	0.00175-0.00183
25	0.00297779321671	0.000334878721555	3.34878721555e-05	0.00292-0.00303	0.00291-0.00304
30	0.00448469877243	0.000511975396613	5.11975396613e-05	0.00440-0.00457	0.00438-0.00459
35	0.0065796661377	0.00139216316547	0.000139216316547	0.00635-0.00681	0.00630-0.00686
40	0.00893982648849	0.00149436851369	0.000149436851369	0.00869-0.00919	0.00864-0.00924
45	0.0118569993973	0.00161576951483	0.000161576951483	0.01159-0.01213	0.01154-0.01218
50	0.0150039553642	0.00190793373503	0.000190793373503	0.01469-0.01532	0.01463-0.01538
55	0.0194254183769	0.00241951165856	0.000241951165856	0.01902-0.01983	0.01895-0.01991
60	0.0242810153961	0.00345015917145	0.000345015917145	0.02371-0.02485	0.02360-0.02497
65	0.0350979852676	0.0074359964295	0.00074359964295	0.03386-0.03633	0.03362-0.03657
70	0.0456251835823	0.0174944742438	0.00174944742438	0.04272-0.04853	0.04215-0.04910
75	0.0437107133865	0.00615142382657	0.000615142382657	0.04269-0.04473	0.04249-0.04493
80	0.0523628973961	0.00583536792638	0.000583536792638	0.05139-0.05333	0.05121-0.05352
85	0.0737262320518	0.0132204007579	0.00132204007579	0.07153-0.07592	0.07110-0.07635
90	0.070827229023	0.0104611850187	0.00104611850187	0.06909-0.07256	0.06875-0.07296
95	0.0755790877342	0.00689486527005	0.000689486527005	0.07443-0.07672	0.07421-0.07695
100	0.0886501836777	0.0109882186574	0.00109882186574	0.08683-0.09047	0.08647-0.09083

Mean Time Based on Array Size for 300 Runs

Array Size	Mean Time	Standard Deviation	Standard Error	90% Confidence Level	95% Confidence Level
5	0.000122786362966	1.9054607131e-05	1.10011825564e-06	0.00012-0.00012	0.00012-0.00012
10	0.00050662746429	8.34049218554e-05	4.8138540849e-06	0.00055-0.00057	0.00055-0.00057
15	0.00129203081131	8.14604685482e-05	4.70312234446e-06	0.00128-0.00130	0.00128-0.00130
20	0.00252215862274	0.000296593059617	1.7123808281e-05	0.00249-0.00255	0.00249-0.00256
25	0.00428865432739	0.000961683860392	5.55228435673e-05	0.00420-0.00438	0.00418-0.00440
30	0.00660872538884	0.00126405500401	7.29802496833e-05	0.00649-0.00673	0.00646-0.00675
35	0.00917361577352	0.00169862156371	9.80699617058e-05	0.00901-0.00934	0.00898-0.00937
40	0.0111118419965	0.00238556150737	0.000137730457845	0.01088-0.01134	0.01084-0.01139
45	0.0115436291695	0.00132973142354	7.67720795332e-05	0.01142-0.01167	0.01139-0.01170
50	0.017150990963	0.00725899379344	0.000419098202069	0.01646-0.01785	0.01632-0.01798
55	0.0191923538844	0.00274950634175	0.000158742822635	0.01893-0.01946	0.01888-0.01951
60	0.0263167373339	0.00772549118691	0.000446031441638	0.02558-0.02706	0.02543-0.02720
65	0.0402355957031	0.0211603604327	0.0012216939792	0.03821-0.04226	0.03781-0.04266
70	0.0357898481687	0.00658035250001	0.000495386882562	0.03497-0.03661	0.03481-0.03677
75	0.0461252514521	0.00942668704849	0.000544250030501	0.04522-0.04703	0.04505-0.04721
80	0.055941936175	0.0123224446363	0.000711436672782	0.05476-0.05712	0.05453-0.05735
85	0.0610211396217	0.00705646474985	0.000407405182285	0.06034-0.06176	0.06021-0.06183
90	0.0705091460546	0.00563813209884	0.000325517708499	0.06997-0.07105	0.06986-0.07115
95	0.0810158554713	0.00637785886943	0.00048369590735	0.08021-0.08182	0.08006-0.08198
100	0.0987580863635	0.0174237139064	0.00100595859141	0.09709-0.10043	0.09676-0.10075

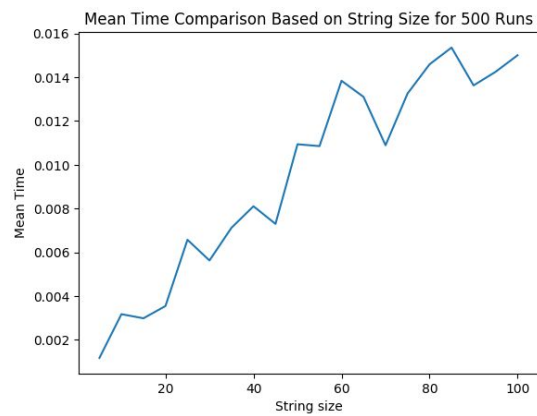
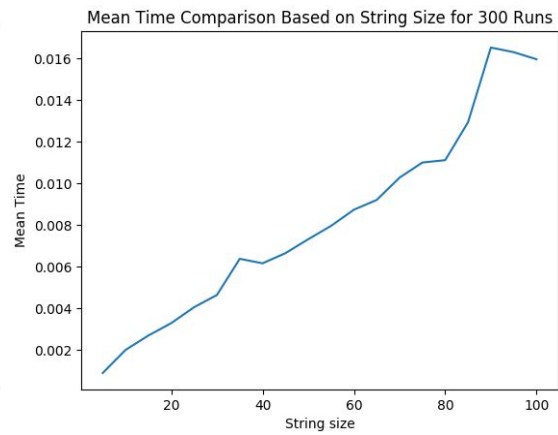
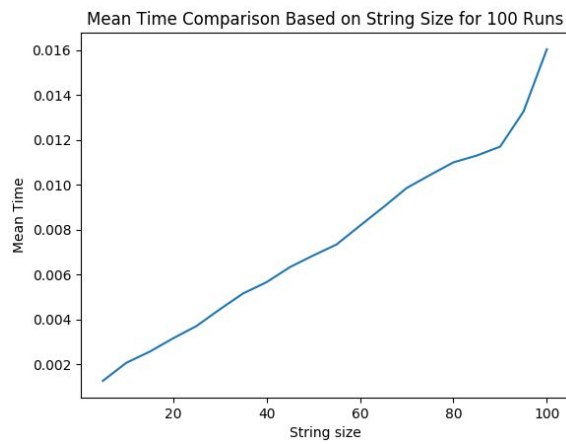
Mean Time Based on Array Size for 500 Runs

Array Size	Mean Time	Standard Deviation	Standard Error	90% Confidence Level	95% Confidence Level
5	0.000102513790131	1.57192486802e-05	7.02986172083e-07	0.00010-0.00010	0.00010-0.00010
10	0.000398312568665	5.68143066075e-05	2.54081303338e-06	0.00039-0.00040	0.00039-0.00040
15	0.000938754558563	8.3134934819e-05	4.16512090671e-06	0.00093-0.00095	0.00093-0.00095
20	0.001786444664	0.000188416367003	8.42623609386e-06	0.00177-0.00180	0.00177-0.00180
25	0.00296707725525	0.000344243388247	1.53950323385e-05	0.00294-0.00299	0.00294-0.00300
30	0.00452248287201	0.000514874546093	2.3025889699e-05	0.00448-0.00456	0.00448-0.00457
35	0.00730132055283	0.00168289383063	7.52613000841e-05	0.00718-0.00743	0.00715-0.00745
40	0.00943919706345	0.00249574197911	0.00011612974392	0.00925-0.00962	0.00922-0.00966
45	0.0200366268158	0.0198111494338	0.000885981536928	0.01857-0.02151	0.01828-0.02179
50	0.0168170418739	0.00217335028402	9.71951794798e-05	0.01666-0.01698	0.01662-0.01701
55	0.0332875614166	0.0183675029376	0.000821419702907	0.03192-0.03465	0.03166-0.03492
60	0.0316333971024	0.0113475932414	0.000507479797375	0.03079-0.03248	0.03063-0.03264
65	0.0410716538429	0.0182887111618	0.000817896027573	0.03971-0.04243	0.03945-0.04269
70	0.049704536438	0.0183648126942	0.000821299391563	0.04834-0.05107	0.04808-0.05133
75	0.0642451286316	0.0251662779781	0.00112547016599	0.06238-0.06611	0.06201-0.06648
80	0.0743268275261	0.033855197193	0.0015140504463	0.07181-0.07684	0.07132-0.07733
85	0.0775759720802	0.0262338515213	0.00117321350626	0.07563-0.07952	0.07525-0.07990
90	0.0978863229752	0.0320042206486	0.00143127225874	0.09551-0.10026	0.09505-0.10073
95	0.109190422058	0.0379415810836	0.00169679908954	0.10637-0.11201	0.10582-0.11256
100	0.12960582489	0.0359408920558	0.00160732555618	0.12700-0.13233	0.12648-0.13285

In general, incrementing the input array size with fixed input string lengths (10) increased the running time polynomially.

2 - Running Time vs. String Length

In these experiments there are 100, 300, and 500 runs for each of the string size incrementation in the range [5,100] with increment size = 5. All arrays contained 20 strings.



Mean Time Based on String Size for 100 Runs

String Size	Mean Time	Standard Deviation	Standard Error	90% Confidence Level	95% Confidence Level
5	0.00126043558121	0.00101352116011	0.000101352116011	0.00109-0.00143	0.00106-0.00146
10	0.00206144332886	0.000430266815446	4.30266815446e-05	0.00199-0.00213	0.00198-0.00215
15	0.00255816936493	0.000355277596877	3.55277596877e-05	0.00250-0.00262	0.00249-0.00263
20	0.00313272331238	0.000339467366464	3.39467366464e-05	0.00310-0.00321	0.00309-0.00322
25	0.00369908094406	0.000512359356704	5.12359356704e-05	0.00361-0.00378	0.00360-0.00380
30	0.00444402450284	0.000444652099831	4.44652099831e-05	0.00437-0.00452	0.00436-0.00453
35	0.00515583276749	0.000554969879687	5.54969879687e-05	0.00506-0.00525	0.00505-0.00527
40	0.00565361022949	0.000592258285882	5.92258285882e-05	0.00556-0.00575	0.00554-0.00577
45	0.00632569313049	0.000688478876073	6.88478876073e-05	0.00621-0.00644	0.00619-0.00646
50	0.00684572696686	0.000631894918983	6.31894918983e-05	0.00674-0.00695	0.00672-0.00697
55	0.00733647346497	0.000536464869745	5.36464869745e-05	0.00725-0.00743	0.00723-0.00744
60	0.0081743144989	0.00085417554922	8.5417554922e-05	0.00803-0.00832	0.00800-0.00834
65	0.00899924039841	0.00115407730291	0.000115407730291	0.00881-0.00919	0.00877-0.00923
70	0.00985316991806	0.00112631118836	0.000112631118836	0.00967-0.01004	0.00963-0.01008
75	0.0104351115227	0.00100398377122	0.000100398377122	0.01027-0.01060	0.01024-0.01063
80	0.0109973740578	0.00155765798573	0.000155765798573	0.01074-0.01126	0.01069-0.01131
85	0.0113011169434	0.00176671444354	0.000176671444354	0.01101-0.01159	0.01095-0.01165
90	0.0116983032227	0.000528340592601	5.28340592601e-05	0.01161-0.01179	0.01159-0.01180
95	0.013259139061	0.00247726775327	0.000247726775327	0.01285-0.01367	0.01277-0.01375
100	0.0160368728638	0.00359761771358	0.000359761771358	0.01544-0.01663	0.01532-0.01675

Mean Time Based on String Size for 300 Runs

String Size	Mean Time	Standard Deviation	Standard Error	90% Confidence Level	95% Confidence Level
5	0.000899478594462	0.000332289297753	1.91847315507e-05	0.00087-0.00093	0.00086-0.00094
10	0.00200689633687	0.000282770406831	1.63257570503e-05	0.00198-0.00203	0.00197-0.00204
15	0.00269887129466	0.000314607464272	1.81638704186e-05	0.00267-0.00273	0.00266-0.00273
20	0.00329374790192	0.000251149662672	1.4500132535e-05	0.00327-0.00332	0.00326-0.00332
25	0.00405735254288	0.000374662718438	2.16311621345e-05	0.00402-0.00409	0.00401-0.00410
30	0.00464276472727	0.000333703578598	1.92663850933e-05	0.00461-0.00467	0.00460-0.00468
35	0.00638066053391	0.00189323667508	0.0001093060704	0.00620-0.00656	0.00616-0.00660
40	0.0061642964681	0.000822307798595	4.74759628875e-05	0.00609-0.00624	0.00607-0.00626
45	0.00665121555328	0.000401046094633	2.31544070694e-05	0.00661-0.00669	0.00661-0.00670
50	0.00732111612956	0.000467360342642	2.69830619633e-05	0.00728-0.00737	0.00727-0.00737
55	0.00796187957128	0.000450589415037	2.60147920066e-05	0.00792-0.00801	0.00791-0.00801
60	0.00873821973801	0.000830944913429	4.7974626945e-05	0.00866-0.00882	0.00864-0.00883
65	0.00920763731003	0.00048419494614	2.79550082494e-05	0.00916-0.00925	0.00915-0.00926
70	0.0102770678202	0.00088145927725	5.08910751e-05	0.01019-0.01036	0.01018-0.01038
75	0.0110042683283	0.00199312733727	0.00011507326047	0.01081-0.01120	0.01078-0.01123
80	0.011168686549	0.000607638892211	3.50820477988e-05	0.01106-0.01118	0.01105-0.01119
85	0.0129362138112	0.00215280187255	0.000124292074063	0.01273-0.01314	0.01269-0.01318
90	0.0165242880543	0.00385573468179	0.000222610945645	0.01615-0.01689	0.01608-0.01697
95	0.0163016327222	0.00472478899431	0.00027785819773	0.01585-0.01675	0.01576-0.01684
100	0.0159625212351	0.0019910813445	0.000114955135023	0.01577-0.01615	0.01573-0.01619

Mean Time Based on String Size for 500 Runs

String Size	Mean Time	Standard Deviation	Standard Error	90% Confidence Level	95% Confidence Level
5	0.00117073249817	0.000594526738935	2.6588044054e-05	0.00113-0.00121	0.00112-0.00122
10	0.0031098846817	0.00322184575429	0.000144085322392	0.00293-0.00341	0.00288-0.00346
15	0.00298115348816	0.000279258273486	1.24888096559e-05	0.00296-0.00300	0.00296-0.00301
20	0.00354185676575	0.000123618136161	5.52837111414e-06	0.00353-0.00355	0.00353-0.00355
25	0.00657058906555	0.00346265231962	0.000244297238482	0.00617-0.00698	0.00609-0.00706
30	0.00562436723709	0.000832703461846	3.72396309158e-05	0.00556-0.00569	0.00555-0.00570
35	0.00712019062042	0.00379323677762	0.00010963870579	0.00684-0.00740	0.00678-0.00746
40	0.00810165786743	0.00515914881779	0.000230724149252	0.00772-0.00848	0.00764-0.00856
45	0.00729690217972	0.000600566750762	2.68581615946e-05	0.00725-0.00734	0.00724-0.00735
50	0.0109372410774	0.0109604520165	0.000490166315461	0.01012-0.01175	0.00996-0.01191
55	0.0108519906998	0.00642508396412	0.000287338490098	0.01038-0.01133	0.01028-0.01142
60	0.013839425087	0.00948376271036	0.000424126762057	0.01314-0.01454	0.01300-0.01468
65	0.0131031541824	0.00762987127819	0.000341218216752	0.01254-0.01367	0.01243-0.01378
70	0.0108900475502	0.00217870206589	9.7434518441e-05	0.01073-0.01105	0.01070-0.01108
75	0.0132550182343	0.00798333926573	0.000357025785712	0.01266-0.01385	0.01255-0.01396
80	0.0145908679962	0.0051869808679	0.000231969104701	0.01421-0.01498	0.01413-0.01505
85	0.0153570628166	0.00895317376688	0.000400398103142	0.01469-0.01602	0.01456-0.01615
90	0.0136272025108	0.00094918186562	4.24487034907e-05	0.01356-0.01370	0.01354-0.01371
95	0.0142479319572	0.000718456069297	3.21303321959e-05	0.01419-0.01430	0.01418-0.01431
100	0.0150060214996	0.000716824142584	3.20573502146e-05	0.01495-0.01506	0.01494-0.01507

In general, incrementing the input string length with fixed input array size (20) increased the running time.

Determining Successful Superstring Size for k

For determining successful superstring size for k, there are 2 main cases.

1. In the first part of the experiments, the mean length of output superstring size is observed when array size is increased with strings of length 10.

2. In the second part of the experiments, the mean length of output superstring size is observed when the string lengths are increased while the array size is fixed at 20.

By doing the following experiments, the aim is to show that our algorithm will compute the corresponding output superstring mean lengths for given array sizes and string lengths. And by choosing the mean length value as k for a given array size (or string size) the algorithm will be correctly computing the shortest common superstring. By choosing k bigger than the mean value, almost every time, the algorithm will be able to find a common superstring of size less than k . By choosing k less than the mean value, almost every time, the algorithm will not be able to find a common superstring of size less than k . The statistics for the mean values, standard derivations, 90% and 95% confidence intervals are given in the tables.

The statistics code used in the superstring length comparisons are as follows:

```

def standardDeviation(results):
    sum = 0
    mean = 0
    standard_deviation = 0
    for i in range(len(results)):
        sum += results[i]

    mean = sum / len(results)

    for j in range(len(results)):
        standard_deviation += pow(results[j] - mean, 2)

    standard_deviation = math.sqrt(standard_deviation / len(results))
    return standard_deviation

```

$$SD = \sqrt{\frac{\sum |x - \bar{x}|^2}{n}}$$

```

def standardError(standard_deviation, n):
    return standard_deviation / math.sqrt(n)

```

$$SE = \frac{\sigma}{\sqrt{n}}$$

```

def calculateStatistics(len_size):
    total = 0
    for i in range(len(len_size)):
        total += len_size[i]

    standard_dev = standardDeviation(len_size)
    N = len(len_size)
    m = total / N
    t_value_90 = 1.660
    t_value_95 = 1.984
    standard_error = standardError(standard_dev, N)
    upper_mean_90 = m + t_value_90 * standard_error
    lower_mean_90 = m - t_value_90 * standard_error
    upper_mean_95 = m + t_value_95 * standard_error
    lower_mean_95 = m - t_value_95 * standard_error
    return [m, standard_dev, standard_error, lower_mean_90, upper_mean_90, lower_mean_95, upper_mean_95]

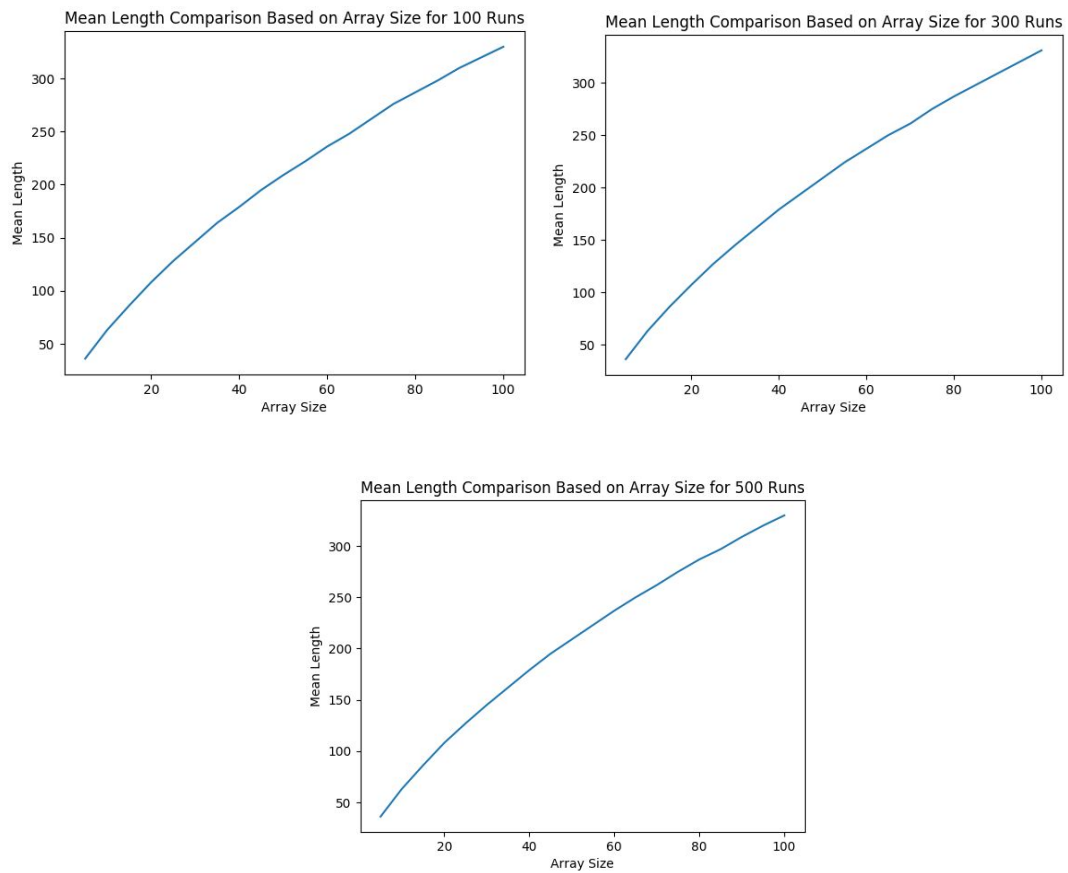
```

$$\bar{x} \pm z \frac{s}{\sqrt{n}}$$

The t values are selected as 1.660 for 90% confidence interval and 1.984 for 95% confidence interval since after 100 samples t values are approximately the same. By using this function, the true mean will be in the given intervals with given confidence levels.

1-The Mean Length vs Array Size

Every experiment ran 100, 300, and 500 times for each of the array size incrementation in the range [5,100] with increment size = 5. All arrays contained 10 as input string length.



Mean Length Based on Array Size for 100 Runs

Array Size	Mean Length	Standard Deviation	Standard Error	90% Confidence Level	95% Confidence Level
5	35	3.46410	0.34641	35.4250-36.5750	35.3127-36.6873
10	63	4.79583	0.47958	62.2039-63.7961	62.0485-63.9515
15	86	7.00000	0.70000	84.8380-87.1620	84.6112-87.3888
20	108	6.24500	0.62450	106.9633-109.0367	106.7610-109.2390
25	128	8.83176	0.88318	126.5339-129.4661	126.2478-129.7522
30	146	9.27362	0.92736	144.4606-147.5394	144.1601-147.8399
35	164	9.48683	0.94868	162.4252-165.5748	162.1178-165.8822
40	179	10.14889	1.01489	177.3153-180.6847	176.9865-181.0135
45	195	11.48913	1.14891	193.0928-196.9072	192.7206-197.2794
50	209	10.72381	1.07238	207.2198-210.7802	206.8724-211.1276
55	222	12.00000	1.20000	220.0080-223.9920	219.6192-224.3808
60	236	11.35782	1.13578	234.1146-237.8854	233.7466-238.2534
65	248	9.74679	0.97468	246.3820-249.6180	246.0662-249.9338
70	262	11.70470	1.17047	260.0570-263.9430	259.6778-264.3222
75	276	12.80625	1.28062	273.8742-278.1258	273.4592-278.5408
80	287	13.07670	1.30767	284.8293-289.1707	284.4056-289.5944
85	298	13.37909	1.33791	295.7791-300.2209	295.3456-300.6544
90	310	14.69694	1.46969	307.5603-312.4397	307.0841-312.9159
95	320	13.74773	1.37477	317.7179-322.2821	317.2725-322.7275
100	330	15.29706	1.52971	327.4607-332.5393	326.9651-333.0349

Mean Length Based on Array Size for 300 Runs

Array Size	Mean Length	Standard Deviation	Standard Error	90% Confidence Level	95% Confidence Level
5	36	3.16228	0.18257	35.6969-36.3031	35.6378-36.3622
10	63	5.00000	0.28868	62.5208-63.4792	62.4273-63.5727
15	86	6.92820	0.40000	85.3360-86.6640	85.2064-86.7936
20	107	8.85565	0.39581	106.3430-107.6570	106.2147-107.7853
25	127	8.06226	0.46547	126.2273-127.7727	126.0765-127.9235
30	145	8.88819	0.51316	144.1482-145.8518	143.9819-146.0181
35	162	9.53939	0.55076	161.0857-162.9143	160.9073-163.0927
40	179	10.00800	0.57735	178.0416-179.9584	177.8545-180.1455
45	194	11.22497	0.64807	192.9242-195.0758	192.7142-195.2858
50	209	10.24695	0.59161	208.0179-209.9821	207.8262-210.1738
55	224	11.31371	0.65320	222.9157-225.0843	222.7041-225.2959
60	237	11.95826	0.69041	235.8539-238.1461	235.6302-238.3698
65	250	12.80625	0.73937	248.7726-251.2274	248.5331-251.4669
70	261	12.00000	0.69282	259.8499-262.1501	259.6254-262.3746
75	275	13.11488	0.75719	273.7431-276.2569	273.4977-276.5023
80	287	13.07670	0.75498	285.7467-288.2533	285.5021-288.4979
85	298	13.45362	0.77675	296.7106-299.2894	296.4589-299.5411
90	309	13.60147	0.78528	307.6964-310.3036	307.4420-310.5580
95	320	13.78405	0.79582	318.6789-321.3211	318.4211-321.5789
100	331	13.11488	0.75719	329.7431-332.2569	329.4977-332.5023

Mean Length Based on Array Size for 500 Runs

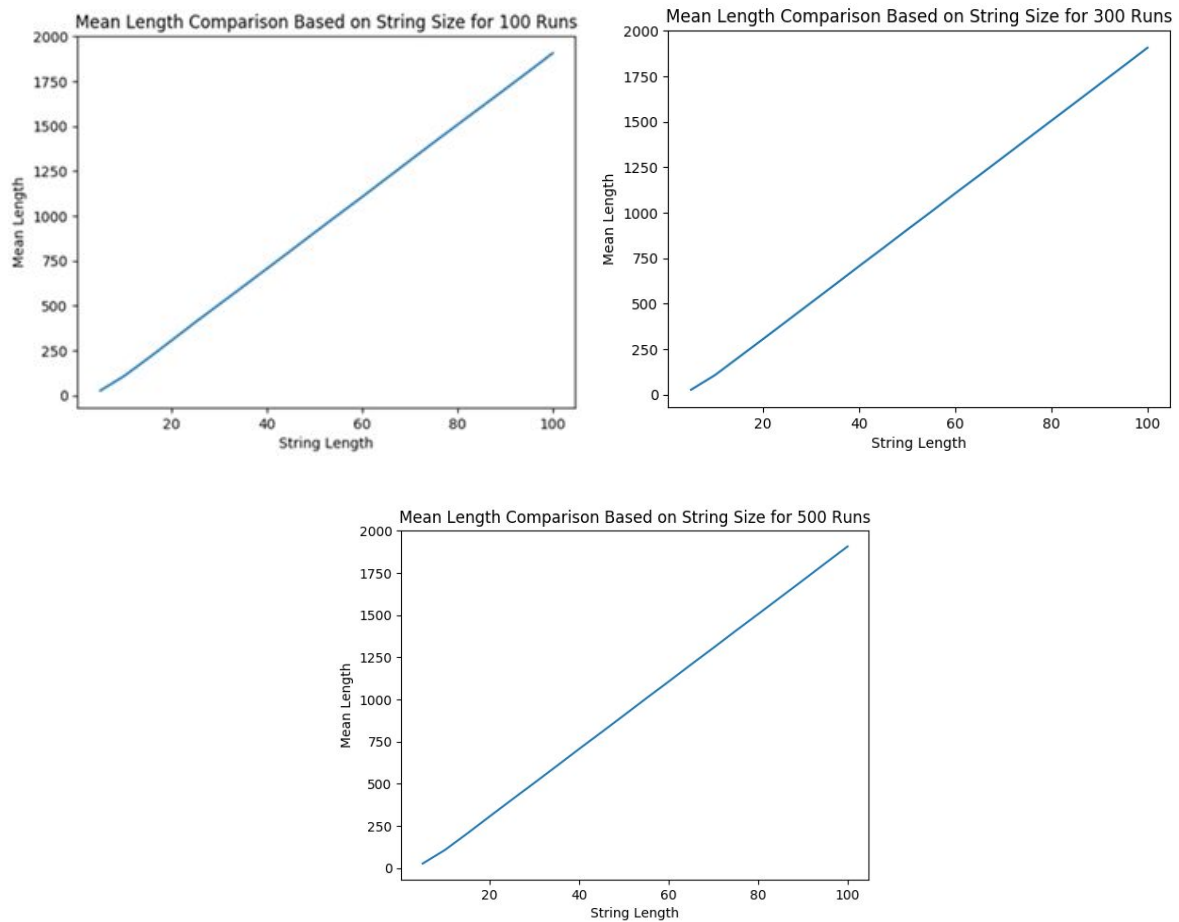
Array Size	Mean Length	Standard Deviation	Standard Error	90% Confidence Level	95% Confidence Level
5	36	3.46410	0.15492	35.7428-36.2572	35.6926-36.3074
10	63	5.00000	0.22361	62.6288-63.3712	62.5564-63.4436
15	86	6.55744	0.29326	85.5132-86.4868	85.4182-86.5818
20	108	7.28011	0.32558	107.4595-108.5405	107.3541-108.6459
25	127	8.12404	0.36332	126.3969-127.6031	126.2792-127.7208
30	145	8.88819	0.39749	144.3402-145.6598	144.2114-145.7886
35	162	9.05539	0.40497	161.3278-162.6722	161.1965-162.8035
40	179	9.64365	0.43128	178.2841-179.7159	178.1443-179.8557
45	195	10.48809	0.46904	194.2214-195.7786	194.0694-195.9306
50	209	11.61895	0.51962	208.1374-209.8626	207.9691-210.0309
55	223	11.18034	0.50000	222.1700-223.8300	222.0080-223.9920
60	237	12.64911	0.56569	236.0610-237.9390	235.8777-238.1223
65	250	12.12436	0.54222	249.0999-250.9001	248.9242-251.0758
70	262	11.91638	0.53292	261.1154-262.8846	260.9427-263.0573
75	275	13.19091	0.58992	274.0207-275.9793	273.8296-276.1704
80	287	13.19091	0.58992	286.0207-287.9793	285.8296-288.1704
85	297	13.52775	0.60498	295.9957-298.0043	295.7997-298.2003
90	309	13.52775	0.60498	307.9957-310.0043	307.7997-310.2003
95	320	13.49074	0.60332	318.9985-321.0015	318.8030-321.1970
100	330	14.14214	0.63246	328.9501-331.0499	328.7452-331.2548

In general, incrementing the input array size with fixed input string lengths (10) increased the mean length of superstring. If k is chosen as the mean superstring length or

higher, the algorithm will successfully compute the shortest common superstring with a size less than k .

2-The Mean Length vs String Length

Every experiment ran 100, 300, and 500 times for each of the input string length incrementation in the range $[5, 100]$ with increment size = 5. All arrays contained 20 strings.



Mean Length Based on String Size for 100 Runs

String Length	Mean Length	Standard Deviation	Standard Error	90% Confidence Level	95% Confidence Level
5	26	2.64573	0.26458	25.5608-26.4392	25.4751-26.5249
10	107	7.34847	0.73485	105.7802-108.2198	105.5421-108.4579
15	205	7.93725	0.79373	203.6824-206.3176	203.4252-206.5748
20	306	9.48683	0.94868	304.4252-307.5748	304.1178-307.8822
25	408	9.00000	0.90000	406.5060-409.4940	406.2144-409.7856
30	507	9.11043	0.91104	505.4877-508.5123	505.1925-508.8075
35	606	9.94987	0.99499	604.3483-607.6517	604.0259-607.9741
40	706	8.83176	0.88318	704.5339-707.4661	704.2478-707.7522
45	806	8.42615	0.84261	804.6013-807.3987	804.3283-807.6717
50	907	7.81025	0.78102	905.7035-908.2965	905.4504-908.5496
55	1007	8.24621	0.82462	1005.6311-1008.3689	1005.3640-1008.6360
60	1106	7.81025	0.78102	1104.7035-1107.2965	1104.4504-1107.5496
65	1207	8.48528	0.84853	1205.5914-1208.4086	1205.3165-1208.6835
70	1308	9.74679	0.97468	1306.3820-1309.6180	1306.0662-1309.9338
75	1409	9.74679	0.97468	1407.3820-1410.6180	1407.0662-1410.9338
80	1508	8.60233	0.86023	1506.5720-1509.4280	1506.2933-1509.7067
85	1607	8.66025	0.86603	1605.5624-1608.4376	1605.2818-1608.7182
90	1706	9.38083	0.93808	1704.4428-1707.5572	1704.1388-1707.8612
95	1806	8.30662	0.83066	1804.6211-1807.3789	1804.3520-1807.6480
100	1907	8.66025	0.86603	1905.5624-1908.4376	1905.2818-1908.7182

Mean Length Based on String Size for 300 Runs

String Length	Mean Length	Standard Deviation	Standard Error	90% Confidence Level	95% Confidence Level
5	27	2.44949	0.14142	26.7652-27.2348	26.7194-27.2806
10	108	7.61577	0.43970	107.2701-108.7299	107.1276-108.8724
15	207	8.83176	0.50990	206.1536-207.8464	205.9884-208.0116
20	306	8.83176	0.50990	305.1536-306.8464	304.9884-307.0116
25	406	8.42615	0.48648	405.1924-406.8076	405.0348-406.9652
30	506	8.60233	0.49666	505.1756-506.8244	505.0146-506.9854
35	606	8.77496	0.50662	605.1590-606.8410	604.9949-607.0051
40	707	9.21954	0.53229	706.1164-707.8836	705.9439-708.0561
45	806	8.83176	0.50990	805.1536-806.8464	804.9884-807.0116
50	907	9.11043	0.52599	906.1269-907.8731	905.9564-908.0436
55	1006	8.88819	0.51316	1005.1482-1006.8518	1004.9819-1007.0181
60	1107	8.12404	0.46904	1106.2214-1107.7786	1106.0694-1107.9306
65	1206	8.06226	0.46547	1205.2273-1206.7727	1205.0765-1206.9235
70	1306	8.88819	0.51316	1305.1482-1306.8518	1304.9819-1307.0181
75	1406	9.11043	0.52599	1405.1269-1406.8731	1404.9564-1407.0436
80	1506	8.60233	0.49666	1505.1756-1506.8244	1505.0146-1506.9854
85	1606	9.05539	0.52281	1605.1321-1606.8679	1604.9627-1607.0373
90	1706	8.60233	0.49666	1705.1756-1706.8244	1705.0146-1706.9854
95	1806	8.18535	0.47258	1805.2155-1806.7845	1805.0624-1806.9376
100	1907	9.27362	0.53541	1906.1112-1907.8888	1905.9377-1908.0623

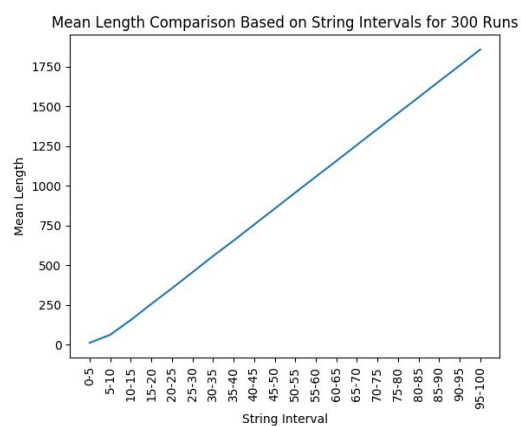
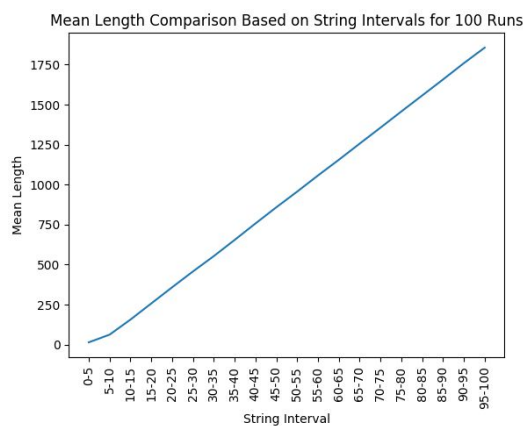
Mean Length Based on String Size for 500 Runs

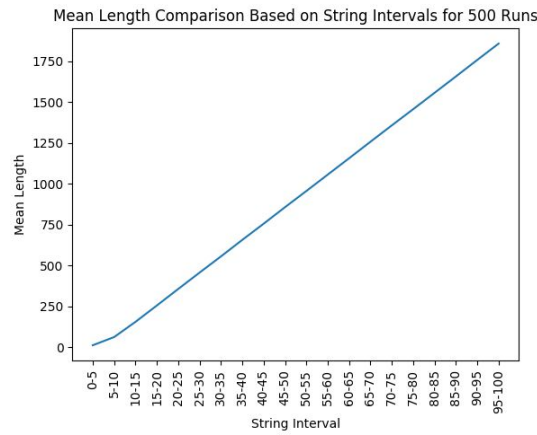
String Length	Mean Length	Standard Deviation	Standard Error	90% Confidence Level	95% Confidence Level
5	27	2.23607	0.10000	26.8340-27.1660	26.8016-27.1984
10	108	7.34847	0.32863	107.4345-108.5455	107.3480-108.6520
15	206	8.60233	0.38471	205.3614-206.6386	205.2367-206.7633
20	307	8.71780	0.38987	306.3528-307.6472	306.2265-307.7735
25	407	8.83176	0.39497	406.3444-407.6556	406.2164-407.7836
30	506	9.11043	0.40743	505.3237-506.6763	505.1917-506.8083
35	606	9.11043	0.40743	605.3237-606.6763	605.1917-606.8083
40	707	9.00000	0.40249	706.3313-707.6681	706.2015-707.7985
45	806	8.83176	0.39497	805.3444-806.6556	805.2164-806.7836
50	906	8.30662	0.37148	905.3833-906.6167	905.2630-906.7370
55	1007	8.88819	0.39749	1006.3402-1007.6598	1006.2114-1007.7886
60	1106	8.24621	0.36878	1105.3878-1106.6122	1105.2683-1106.7317
65	1207	8.30662	0.37148	1206.3833-1207.6167	1206.2630-1207.7370
70	1306	8.88819	0.39749	1305.3402-1306.6598	1305.2114-1306.7886
75	1407	9.21954	0.41231	1406.3156-1407.6844	1406.1820-1407.8180
80	1506	8.60233	0.38471	1505.3614-1506.6386	1505.2367-1506.7633
85	1606	8.77496	0.39243	1605.3486-1606.6514	1605.2214-1606.7786
90	1706	9.59166	0.42895	1705.2879-1706.7121	1705.1490-1706.8510
95	1807	8.48528	0.37947	1806.3701-1807.6299	1806.2471-1807.7529
100	1907	8.83176	0.39497	1906.3444-1907.6556	1906.2164-1907.7836

In general, incrementing the input string length with fixed array size (20) increased the mean length of superstring. If the k is chosen as the mean superstring length or higher, the algorithm will successfully computes the shortest common superstring with size less than k .

Determining Successful Superstring Size for k with Array Size = 20 and Changing String Length Intervals

For determining expected superstring size the random strings with random lengths in a fixed size array are considered for 100, 300, and 500 runs. In these cases the array sizes were fixed at 20.





Mean Length Based on String Intervals for 100 Runs

String Interval	Mean Length	Standard Deviation	Standard Error	90% Confidence Level	95% Confidence Level
0-5	14	2.64575	0.26458	13.5008-14.4392	13.4751-14.5249
5-10	62	7.28011	0.72801	60.7915-63.2085	60.5556-63.4444
10-15	126	11.87434	1.18743	124.0289-127.9711	123.6441-128.3559
15-20	217	10.77033	1.07703	205.2121-228.7879	204.8632-229.1368
20-25	308	10.58301	1.05830	306.2432-329.7568	305.9003-360.0997
25-30	407	11.74734	1.17473	405.0499-458.9501	404.6693-459.3307
30-35	503	12.16553	1.21655	500.9805-555.0195	500.5864-555.4136
35-40	604	11.13553	1.11355	602.1515-655.8485	601.7907-656.2093
40-45	707	10.34408	1.03441	705.2829-758.7171	704.9477-759.0523
45-50	808	11.31371	1.13137	806.1219-859.8781	805.7554-860.2446
50-55	906	11.26943	1.12694	904.1293-957.8707	903.7641-958.2359
55-60	1000	10.00000	1.00000	1000.3400-1058.6600	1000.0160-1058.9840
60-65	1155	11.48913	1.14891	1153.0928-1156.9072	1152.7206-1157.2794
65-70	1256	11.57584	1.15758	1254.0784-1257.9216	1253.7034-1258.2966
70-75	1356	11.09054	1.10905	1354.1596-1357.8410	1353.7996-1358.2004
75-80	1457	11.70470	1.17047	1455.0570-1458.9430	1454.6778-1459.3222
80-85	1557	10.81665	1.08167	1555.2044-1558.7956	1554.8540-1559.1460
85-90	1657	12.40967	1.24097	1654.9400-1659.0600	1654.5379-1659.4621
90-95	1759	11.83216	1.18322	1757.0359-1760.9641	1756.6525-1761.3475
95-100	1856	10.00000	1.00000	1854.3400-1857.6600	1854.0160-1857.9840

Mean Length Based on String Intervals for 300 Runs

String Interval	Mean Length	Standard Deviation	Standard Error	90% Confidence Level	95% Confidence Level
0-5	13	2.64575	0.15275	12.7464-13.2536	12.6969-13.3031
5-10	63	8.12404	0.46904	62.2214-63.7786	62.0694-63.9306
10-15	126	11.40175	0.65828	124.9073-127.0927	124.6940-127.3060
15-20	217	11.18034	0.64350	215.9285-228.0715	215.7193-228.2807
20-25	305	11.35782	0.65574	303.9115-326.0885	303.6990-326.3010
25-30	406	11.66190	0.67330	404.8823-427.1177	404.6642-427.3358
30-35	508	11.22497	0.64807	506.9242-529.0758	506.7142-529.2858
35-40	605	11.57584	0.66833	603.8906-626.1094	603.6740-626.3260
40-45	706	11.18034	0.64350	704.9285-727.0715	704.7193-727.2807
45-50	806	10.53565	0.60828	804.9903-827.0097	804.7932-827.2068
50-55	907	12.00000	0.69282	905.8499-928.1501	905.6254-928.3746
55-60	1007	11.22497	0.64807	1005.9242-1028.0758	1005.7142-1028.2858
60-65	1156	11.35782	0.65574	1154.9115-1157.0885	1154.6990-1157.3010
65-70	1257	11.95826	0.69041	1255.8539-1258.1461	1255.6302-1258.3698
70-75	1357	11.40175	0.65828	1355.9073-1358.0927	1355.6940-1358.3060
75-80	1457	11.22497	0.64807	1455.9242-1458.0758	1455.7142-1458.2858
80-85	1556	11.91638	0.68799	1554.8579-1557.1421	1554.6350-1557.3650
85-90	1657	10.86278	0.62716	1655.9589-1658.0411	1655.7557-1658.2443
90-95	1756	11.66190	0.67330	1754.8823-1757.1177	1754.6642-1757.3358
95-100	1857	12.36932	0.71414	1855.8145-1858.1855	1855.5831-1858.4169

Mean Length Based on String Intervals for 500 Runs

String Interval	Mean Length	Standard Deviation	Standard Error	90% Confidence Level	95% Confidence Level
0-5	13	2.82843	0.12649	12.7900-13.2100	12.7490-13.2510
5-10	63	8.36660	0.37417	62.3789-63.6211	62.2577-63.7423
10-15	156	11.44552	0.51186	155.1503-156.8497	154.9845-157.0155
15-20	256	11.66190	0.52154	255.1342-256.8658	254.9653-257.0347
20-25	357	11.74734	0.52536	356.1279-357.8721	355.9577-358.0423
25-30	457	11.66190	0.52154	456.1342-457.8658	455.9653-458.0347
30-35	556	12.16553	0.54406	555.0969-556.9031	554.9206-557.0794
35-40	657	11.78983	0.52726	656.1248-657.8752	655.9539-658.0461
40-45	756	12.72792	0.56921	755.0551-756.9449	754.8707-757.1293
45-50	857	12.08305	0.54037	856.1030-857.8970	855.9279-858.0721
50-55	956	11.53256	0.51575	955.1439-956.8561	954.9767-957.0233
55-60	1056	11.87434	0.53104	1055.1185-1056.8815	1054.9404-1057.0596
60-65	1156	11.74734	0.52536	1155.1279-1156.8721	1154.9577-1157.0423
65-70	1257	11.53256	0.51575	1256.1439-1257.8561	1255.9767-1258.0233
70-75	1357	11.40175	0.50990	1356.1536-1357.8464	1355.9884-1358.0116
75-80	1456	11.26943	0.50398	1455.1634-1456.8366	1455.0001-1456.9999
80-85	1556	11.53256	0.51575	1555.1439-1556.8561	1554.9767-1557.0233
85-90	1656	11.95826	0.53479	1655.1122-1656.8878	1654.9390-1657.0610
90-95	1757	11.61895	0.51962	1756.1374-1757.8626	1755.9691-1758.0309
95-100	1857	11.66190	0.52154	1856.1342-1857.8658	1855.9653-1858.0347

In general, when random strings lengths observed between fixed interval lengths, it is concluded that the resulting lengths approximately stay between the interval of fixed lengths such as if we look at random string length between 5-10, all length of 5 strings in the same array size gives us a lower bound, at the same time all length of 10 strings leads to an approximate upper limit for the result of this calculation. For example, the case when the interval of lengths is searched the lower mean superstring size will be the mean superstring size of the strings with length 5, and the upper mean superstring size will be the mean superstring size of the strings with length 10.

Testing

For testing Black Box Testing technique is used. Firstly, random number of strings and random string lengths are generated by the function above to observe if it will fail or not.

```

import random
word = ""
Strings=[]
numstr = random.randint(1,10)      #number of the strings
for i in range (0,numstr):
    strlength = random.randint(5,25) #string length
    print ("strlength is : ", strlength)
    for j in range (0,strlength):
        ran = random.randint(0,1)
        str(ran)
        word += str(ran)

    print ("word is : ", word)
    Strings.append(word)
    word = ""
print ("Array is : ",Strings)

```

Also, all elements of the array are the subset of the output string. It is proved by the function below.

```

def verify(Arr, SS):
    count = 0
    for s in Arr:
        for i in range(len(SS)):
            if(s == SS[i:i+len(s)]):
                count = count + 1
                break
    if(count == len(Arr)):
        print('Correct')
    else: print ("nope")

verify(Arr, SS)

```

For random numbers, the results are:

Array	0011001010	010001	11101110	0001010	0000110	10100001
Output	111011101000101000011001010					

Array	00000	10011	100	1011
Output	0000010011011			

Array	000001110	01001100	11101100011	1011111100	01001101000	001011010	01011000	1110101010	00011101
Output	1110101010011010000011101100011011111001011010011001011000								

The results in extreme cases like, number of strings and string lengths are equal to 0 respectively:

Array	
Output	

Array	' '	' '	' '	' '	' '	' '
Output						

In case one, no string was created so output is empty. In case two, six empty strings are created, so again output is empty.

Finally, the case that all strings are the same is tested. Result is:

Array	1111	1111	1111	1111
Output	1111			

5. Discussion

As a result, in this report, the Shortest Common Superstring problem is examined which is a NP-complete problem that is proved by reduction from the Maximum Asymmetric Traveling Salesman problem. Since the problem is NP-complete, an efficient algorithm for as a solution to this problem does not exist. Instead, there are heuristic algorithms that do not give a guarantee to find the correct answer, however, they work reasonably faster than the brute-force solution. Greedy approach which is one of the heuristic algorithms, is used as the solution, but the algorithm does not always find the shortest superstring.

In the experiments, an increase in the array size while the string length is kept constant at 10, caused an increase in the running time. Also, an increase in the string length while the array size is kept constant at 20, caused an increase in the running time. Which concludes that if the array size or the string size is increased, in general, algorithm's running time is increased. For the successful results of the algorithm, the value of k should be chosen

regarding the mean superstring length such as found in the second part of the experiments which is approximately linearly increasing. When the string size is chosen between intervals instead of fixed size, the mean of superstring varies between upper and lower bound of the interval.

References

- [1] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis, “Linear approximation of shortest superstrings,” *Journal of the ACM*, vol. 41, no. 4, pp. 630–647, Jan. 1994.
- [2] Haim Kaplan and Nira Shafir. The greedy algorithm for shortest superstrings. *Inf. Process. Lett.*, 93(1):13–17, 2005.
- [3] K. E. Paluch. Better approximation algorithms for maximum asymmetric traveling salesman and shortest superstring. *CoRR*, abs/1401.3670, 2014.
- [4] K.J. Räihä and E. Ukkonen. The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science*, 16:187–198, 1981.
- [5] L. Bulteau, F. Hüffner, C. Komusiewicz, R. Niedermeier. Multivariate Algorithmics for NP-Hard String Problems. Institut für Softwaretechnik und Theoretische Informatik, TU Berlin, Germany.
- [6] Mucha, M.: Lyndon Words and Short Superstrings. In: Proceedings of the TwentyFourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013. Society for Industrial and Applied Mathematics (2013)