Sabancı University

Fall 2019

CS301 - Algorithms

Project Report

Shortest Common Superstring

UTKU GÖKBERK ŞEN, ATA SARP MİLDAN, FURKAN EKEN, FAİK ŞAHİN, ALP DİNÇER

PROBLEM DESCRIPTION

The Shortest Common Superstring (SCS) is a problem for optimization on sequences. These sequences vary into areas like DNA genome projects, DNA sequence assembly in Computational Biology and also in Computer Science and Engineering problems scheduling, query optimization, data compression and text-editing. As an instance, if we define a Finite Set R={r_1, r_2, … ,r_m} of binary strings or referred as alphabet (sequences of 0 and 1) and a positive integer k.

Is there a binary string w of length at most 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 binary strings?

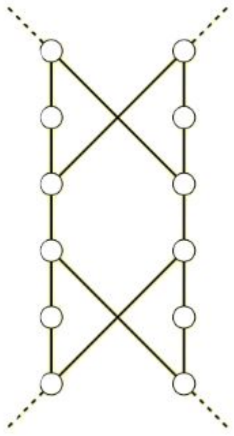
According to [Nikola Kapamadzin, 2015] "NP Completeness of Hamiltonian Circuits and Paths" Hamiltonian circuits can be reduced to the vertex cover problem, and then that Hamiltonian Paths can be reduced to Hamiltonian Circuits. This will complete our logic bringing us to the conclusion that Shortest Common Superstring is NP Complete because we will use Hamiltonian Circuits to solve our problem.

PROOF:

1) First we need to show that Hamiltonian Circuits in NP.
2) Show Hamiltonian Circuits NP-Hard.

A graph G has a Hamiltonian Circuit if there exists a cycle that goes through each vertex in G. We want to show that there is a way to reduce the vertex cover a graph with a vertex cover, to a graph with a Hamiltonian circuit. To do this we will construct a graph G′, so G has a vertex cover of size k if and only if G′ has a Hamiltonian circuit. First to show Hamiltonian circuits are in NP. This is obvious because we simply need to traverse the given edges and make sure every vertex is run through, and that we start and end at the same points, which can be done in polynomial time. Now to show Hamiltonian circuits are NP Hard.

We will run through an example while constructing a Hamiltonian circuit from a vertex cover, and then discuss why this will always work. We begin by creating gadgets that look like the (Figure 1).



.

Figure 1, The Gadget

We can enter and exit each gadget through the dotted lines at the ends. We see that there are exactly three different ways to pass through every vertex in the gadget. The red lines represent the possible paths (Figure 2).

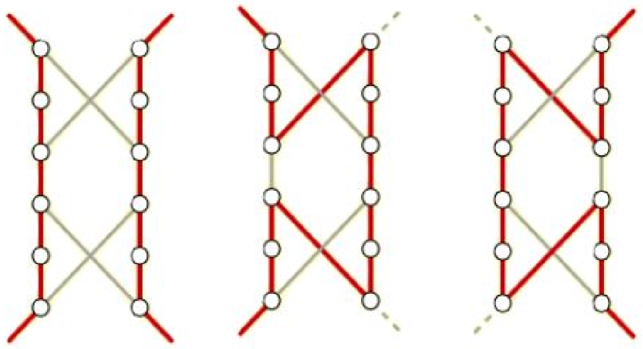


Figure 2

For this first case, we move through one side of the gadget, go somewhere else in the graph, and then come back through the other side. In the other two cases we run through all of the gadgets at once. For every edge in our vertex cover graph, where the vertices in the vertex cover are blue, we place a gadget.We place the gadgets as such, and since every edge contains at least one vertex in the vertex cover, we will color the edge yellow on the side of each gadget where there is a vertex in the vertex cover. (Figure 3)

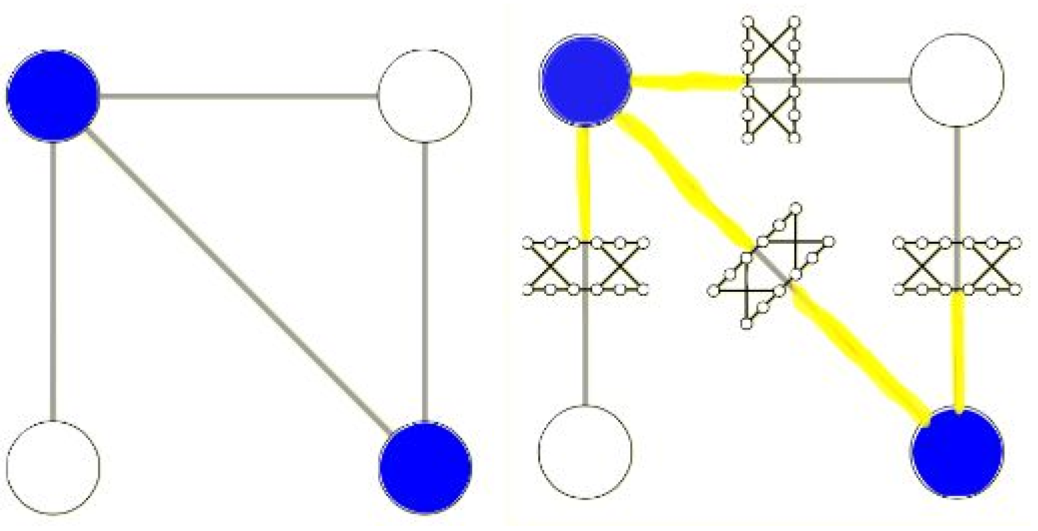


Figure 3, For Graph G

Now create new (green) copy vertices for every vertex in the vertex cover of G. Label these copy vertices $\{v_1, v_2 ... v_k\}$. These copy vertices are part of G′ as are the gadgets.We make connections between gadgets and other gadgets, and between gadgets and the copy vertices. We make a connection between two gadgets if all of the following conditions are met

- Both gadgets share a vertex, v_1, in the vertex cover relative to the edge they correspond to.
- There exist two gadgets that contain the same vertex, v_1, in the vertex cover that only have one connection to other gadgets.
- A maximum of 2 gadgets can be connected to a different gadget, where the connections happen at opposite ends of the gadget.

We generally leave the two gadgets that are furthest away, yet connected to the same v_1 to not be connected, however it does not matter. Here is a way we can connect the gadgets (Figure4).

Within the gadgets, we make the following connections that correspond to which side a vertex in a vertex cover is in relation to the gadget. The side of the gadget facing a vertex cover vertex must be entered and exited in (side where we have 6 vertices in a line).

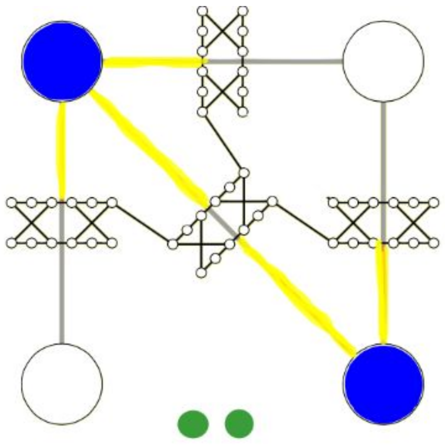


Figure 4

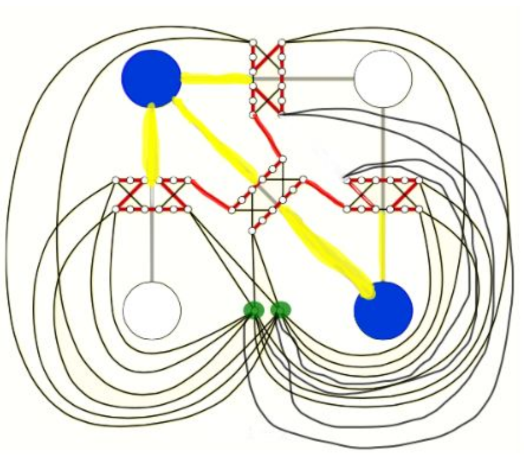


Figure 5

We make a connection for both sides of the gadget (here sides are where the original edge breaks up the gadget, so the two connections represent one for in, and another for out), that either goes to other gadgets or the copy vertices that were copies of the vertex cover. Every gadget must be entered and exited on the side neighboring the red edge, which exists for every edge. If an endpoint of a gadget does not make a connection to another gadget, then it connects to two of the copy vertices. This gives us Figure 6.

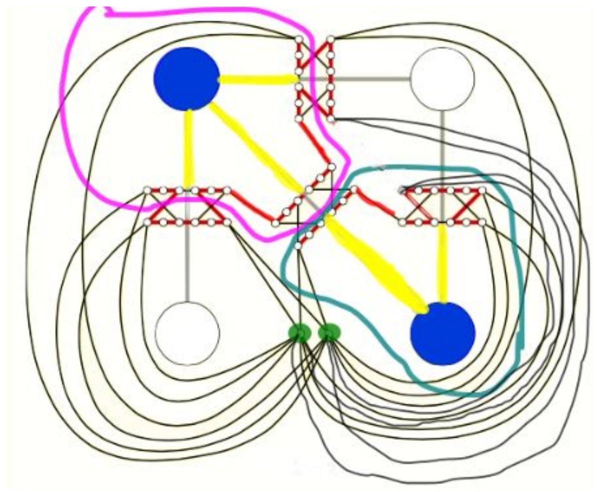


Figure 6

As we can see, a vertex in the original vertex cover always has a certain amount of sides of gadgets that belong directly to it. We can think of each of the vertices in the vertex cover as breaking up the graph into regions, because we surround each vertex in the vertex cover with a red path that comes from all the edges the vertex is contained in. Since each gadget can contain a maximum of 2 vertices in its corresponding edge and a minimum of 1, and there exists paths in the gadget that correspond to both of those conditions, we will always make this kind of surrounding of the vertices in the vertex cover.

Now let's connect gadgets to the copy vertices in red for our actual path. All of the entrances and exits for our gadgets are inside regions corresponding to a vertex in the vertex cover, and there are exactly two endpoints of gadgets that are not connected,in red, to anything. We connect both of these endpoint vertices that aren't connected in red, to distinct copy vertices such that our first connection is to v_1, and our second connection is to v_2. Since we know there is another connection to be made from each vertex, now connect v_2 to an endpoint of the next region. If we had more vertices we would connect the second endpoint of this new region to v_3, but since we have only two copy vertices, we connect back to v_1. This ensures every copy vertex is connected to exactly two gadgets (that come from different regions), and don't make smaller cycles within the graph. This gives us Figure 7.

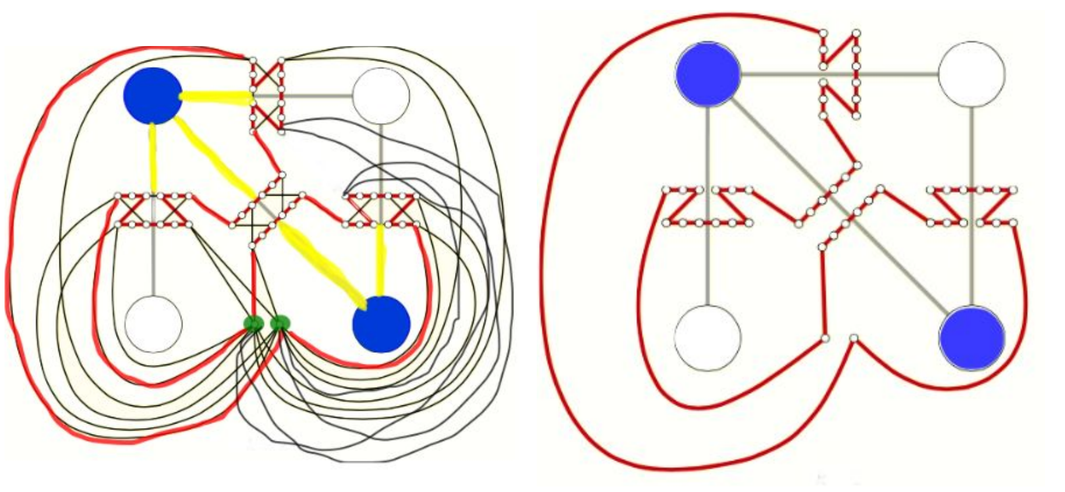


Figure 7. Smaller regions to whole connected

As we can see, this makes a Hamiltonian circuit! When we checked earlier, we enclose each vertex in the vertex cover with our paths from the gadgets with respect to the edges they correspond to. Now we connect each of these unconnected endpoints in the gadgets to distinct copy vertices, which further enclosed the vertices in the vertex cover. Each of these copy vertices connected to exactly one other unconnected endpoint of a gadget from a different region (as these are the only connections that can be made from the copy vertices). However there is one more unconnected endpoint of a gadget in this new region that now connects to the next copy vertex. This continues until we reach the point where the only copy vertex that a regions endpoint can connect to is the very first copy vertex we connected to. We always have this situation because there is one region for each vertex in the vertex cover, and one copy vertex for each vertex in the vertex cover. Once we make this last connection, we finally completely enclose the cycle, which by construction goes through all of our points.

We have shown that any graph with a vertex cover, can be used to make a graph with a Hamiltonian Circuit. Since creating such a graph can be done under polynomial time, simply replace edges with gadgets and make proper connections, we have a reduction from Vertex Coverings to Hamiltonian Circuits.This means that finding whether a graph has a Hamiltonian Circuit or not is NP-Hard. Since this is also in NP, we have an NP Complete Problem.

In conclusion, we showed that finding Hamiltonian Circuits in a graph is NP Complete, how-ever in the proof of Shortest Common Substring, we used Hamiltonian Paths. The reduction here is quite simple to show Hamiltonian Paths are also NP Complete.

ALGORITHM DESCRIPTION

There is not a correct algorithm that solves SCS problem in polynomial time but there are some approximate solutions such as greedy SCS algorithm and a more efficient Reduce-Expand algorithm in time and space $O(n^2+nk)$ by Paolo Barone, Paola Bonizzoni, Gianluca Della Vedova and Giancarlo Mauri.

In this project, we aim to cover and analyze the performance of the greedy approach to SCS problem. This algorithm is very similar to the Hamiltonian Path problem. Different from it, we take the highest weighted edges in our graph instead of the lowest ones. The weights are calculated according to the lengths of the maximum overlap between strings. And instead of deriving a path instantly, we choose the highest valued edges at once and merge the two nodes corresponding to that edge giving us a new node and edge thus recalculating new weights created. This ensures that we only traverse every node once and do not get in a loop.

The algorithm essentially does the following steps:

1. At the beginning, it finds the most overlapping string pair of two.
2. Then it combines these two strings by their overlapping parts.
3. Removes the used strings and replaces them with the new combined string.
4. Repeats these steps above if there are multiple strings left.
5. If there is only one string left, it is our SCS.

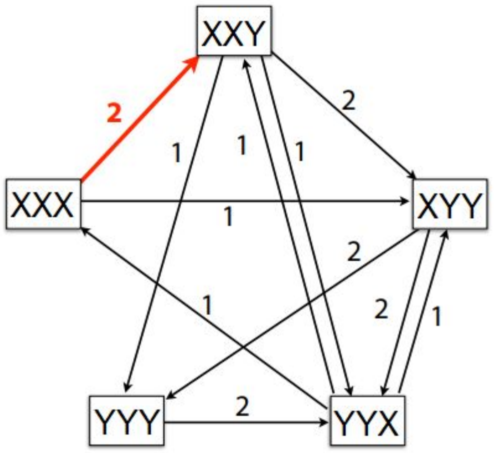


Figure 8, Overlap graph choosing highest weighted edge

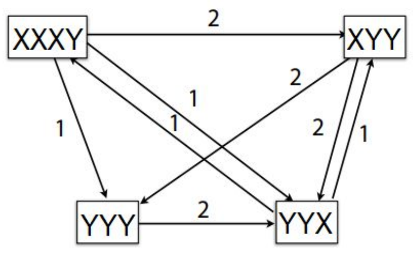


Figure 9, Overlap graph after merging two strings

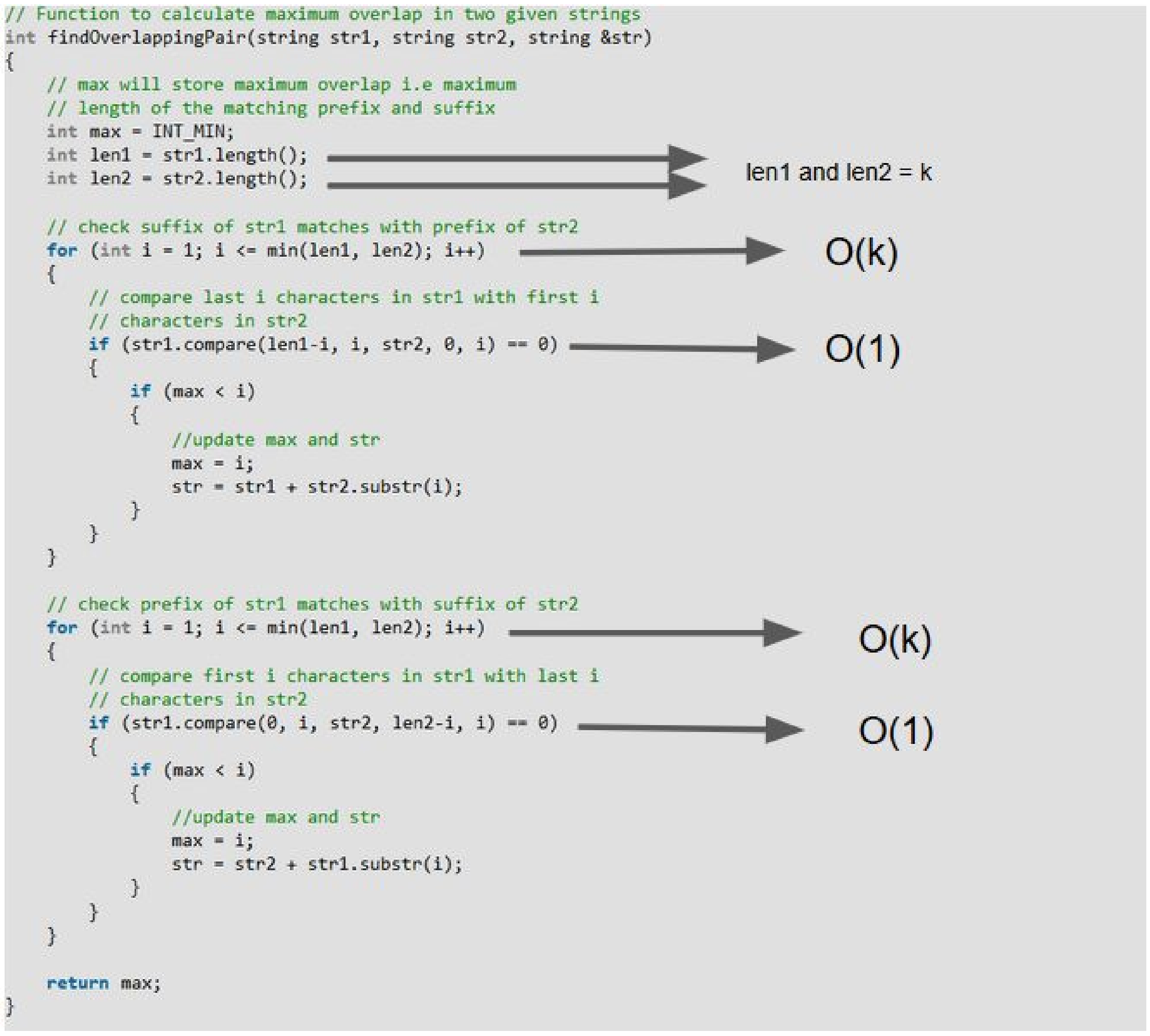
```
// Function to calculate maximum overlap in two given strings
int findOverlappingPair(string str1, string str2, string &str)
{
    // max will store maximum overlap i.e maximum
    // length of the matching prefix and suffix
    int max = INT_MIN;
    int len1 = str1.length();
    int len2 = str2.length();

    // check suffix of str1 matches with prefix of str2
    for (int i = 1; i <= min(len1, len2); i++)
    {
        // compare last i characters in str1 with first i
        // characters in str2
        if (str1.compare(len1-i, i, str2, 0, i) == 0)
        {
            if (max < i)
            {
                //update max and str
                max = i;
                str = str1 + str2.substr(i);
            }
        }
    }

    // check prefix of str1 matches with suffix of str2
    for (int i = 1; i <= min(len1, len2); i++)
    {
        // compare first i characters in str1 with last i
        // characters in str2
        if (str1.compare(0, i, str2, len2-i, i) == 0)
        {
            if (max < i)
            {
                //update max and str
                max = i;
                str = str2 + str1.substr(i);
            }
        }
    }

    return max;
}
```

```
// Function to calculate smallest string that contains
// each string in the given set as substring.
string findShortestSuperstring(string arr[], int len)
{
    // run len-1 times to consider every pair
    while(len != 1)
    {
        int max = INT_MIN;  // to store  maximum overlap
        int l, r;   // to store array index of strings
        // involved in maximum overlap
        string resStr;  // to store resultant string after
        // maximum overlap

        for (int i = 0; i < len; i++)
        {
            for (int j = i + 1; j < len; j++)
            {
                string str;

                // res will store maximum length of the matching
                // prefix and suffix str is passed by reference and
                // will store the resultant string after maximum
                // overlap of arr[i] and arr[j], if any.
                int res = findOverlappingPair(arr[i], arr[j], str);

                // check for maximum overlap
                if (max < res)
                {
                    max = res;
                    resStr.assign(str);
                    l = i, r = j;
                }
            }
        }

        len--;  //ignore last element in next cycle

        // if no overlap, append arr[len] to arr[0]
        if (max == INT_MIN)
            arr[0] += arr[len];
        else
        {
            arr[l] = resStr;   // copy resultant string to index l
            arr[r] = arr[len];  // copy string at last index to index r
        }
    }
    return arr[0];
}
```

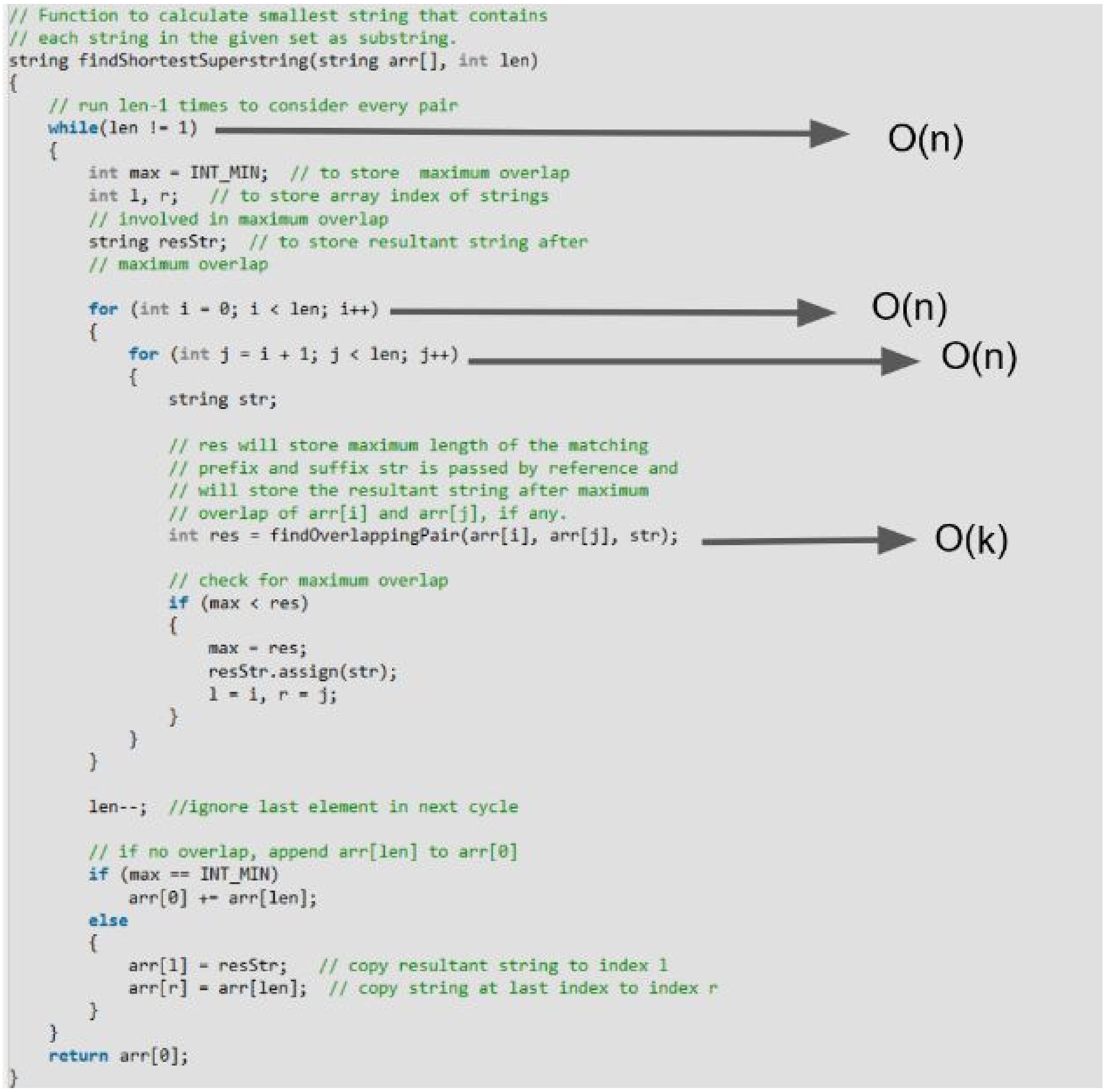
ALGORITHM ANALYSIS

The algorithm we propose does not guarantee that it will find the Shortest Common Superstring because it is a greedy algorithm. We always choose the maximum found overlap at the moment, but that may or may not yield the Shortest Common Superstring. Though, it is guaranteed that it will generate a Superstring since we are using a similar approach to the Hamiltonian Path algorithm. We visit every string and only once.

We calculate the worst case running time of the algorithm in this fashion:



```
// Function to calculate maximum overlap in two given strings
int findOverlappingPair(string str1, string str2, string &str)
{
    // max will store maximum overlap i.e maximum
    // length of the matching prefix and suffix
    int max = INT_MIN;
    int len1 = str1.length();
    int len2 = str2.length();

    // check suffix of str1 matches with prefix of str2
    for (int i = 1; i <= min(len1, len2); i++)
    {
        // compare last i characters in str1 with first i
        // characters in str2
        if (str1.compare(len1-i, i, str2, 0, i) == 0)
        {
            if (max < i)
            {
                //update max and str
                max = i;
                str = str1 + str2.substr(i);
            }
        }
    }

    // check prefix of str1 matches with suffix of str2
    for (int i = 1; i <= min(len1, len2); i++)
    {
        // compare first i characters in str1 with last i
        // characters in str2
        if (str1.compare(0, i, str2, len2-i, i) == 0)
        {
            if (max < i)
            {
                //update max and str
                max = i;
                str = str2 + str1.substr(i);
            }
        }
    }

    return max;
}
```



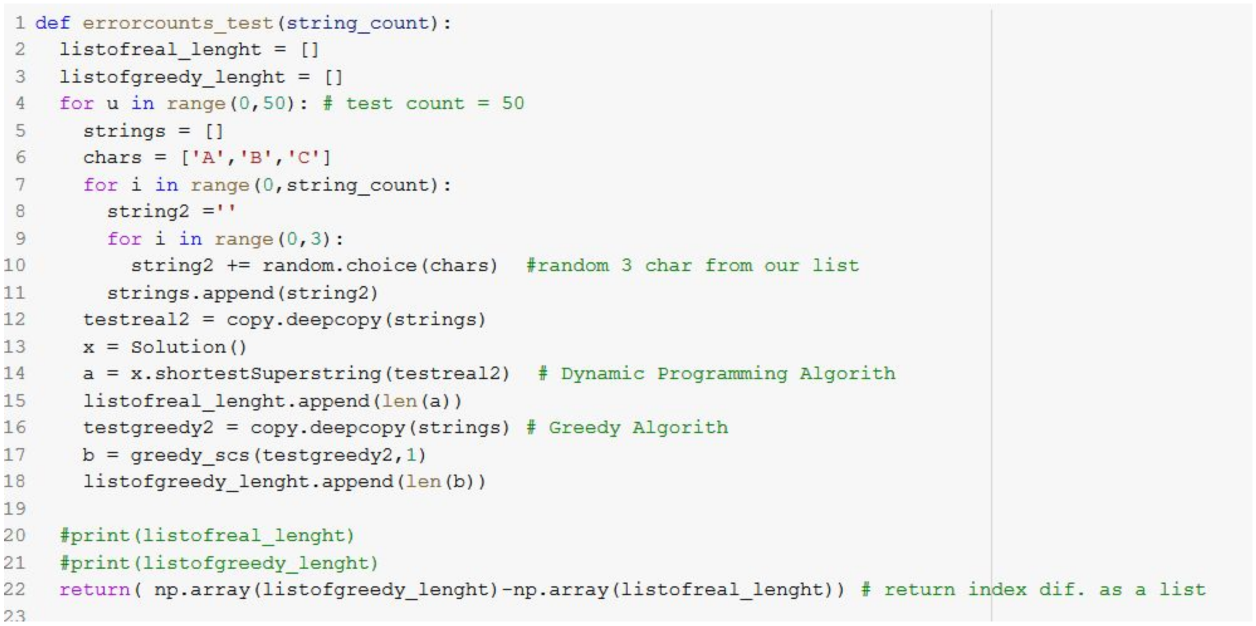
```
// Function to calculate smallest string that contains
// each string in the given set as substring.
string findShortestSuperstring(string arr[], int len)
{
    // run len-1 times to consider every pair
    while(len != 1)
    {
        int max = INT_MIN;  // to store  maximum overlap
        int l, r;   // to store array index of strings
        // involved in maximum overlap
        string resStr;  // to store resultant string after
        // maximum overlap

        for (int i = 0; i < len; i++)
        {
            for (int j = i + 1; j < len; j++)
            {
                string str;

                // res will store maximum length of the matching
                // prefix and suffix str is passed by reference and
                // will store the resultant string after maximum
                // overlap of arr[i] and arr[j], if any.
                int res = findOverlappingPair(arr[i], arr[j], str);

                // check for maximum overlap
                if (max < res)
                {
                    max = res;
                    resStr.assign(str);
                    l = i, r = j;
                }
            }
        }

        len--;  //ignore last element in next cycle

        // if no overlap, append arr[len] to arr[0]
        if (max == INT_MIN)
            arr[0] += arr[len];
        else
        {
            arr[l] = resStr;   // copy resultant string to index l
            arr[r] = arr[len];  // copy string at last index to index r
        }
    }
    return arr[0];
}
```

Running Time: O(k)* O(n*n *n) = O(k*n^3)

EXPERIMENTAL ANALYSIS

SUCCESS RATE

For experimental analysis success rate part, we randomly select 3 characters from char array and append it a string then append the string to string list. We redo this step until reach desired string count. After that we give this string list as an input to our greedy shortest common superstring algorithm and dynamic programming shortest common superstring algorithm. Checking outputs for error count. To reach an accurate error mean we do this test to 50 times then calculate mean of error count. For these steps we create a function as errorcounts_test(string_count) (Figure 10).



```
def errorcounts_test(string_count):
  listofreal_lenght = []
  listofgreedy_lenght = []
  for u in range(0,50): # test count = 50
    strings = []
    chars = ['A','B','C']
    for i in range(0,string_count):
      string2 =''
      for i in range(0,3):
        string2 += random.choice(chars)  #random 3 char from our list
      strings.append(string2)
    testreal2 = copy.deepcopy(strings)
    x = Solution()
    a = x.shortestSuperstring(testreal2)  # Dynamic Programming Algorith
    listofreal_lenght.append(len(a))
    testgreedy2 = copy.deepcopy(strings) # Greedy Algorith
    b = greedy_scs(testgreedy2,1)
    listofgreedy_lenght.append(len(b))

  #print(listofreal_lenght)
  #print(listofgreedy_lenght)
  return( np.array(listofgreedy_lenght)-np.array(listofreal_lenght)) # return index dif. as a list
```

Figure 10, Error Counts Function

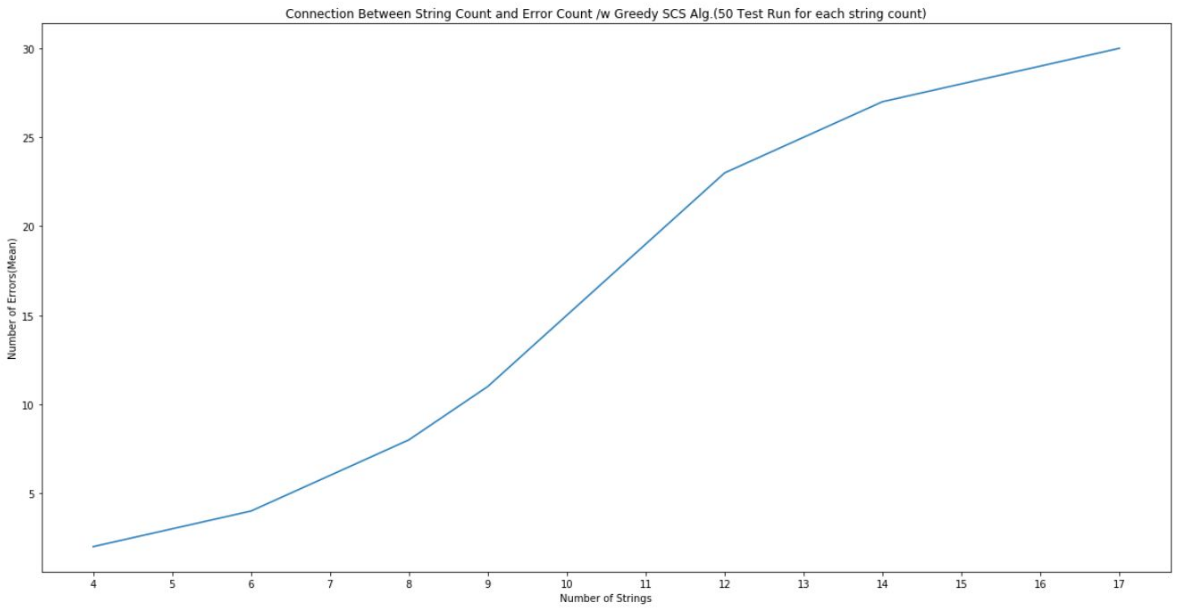
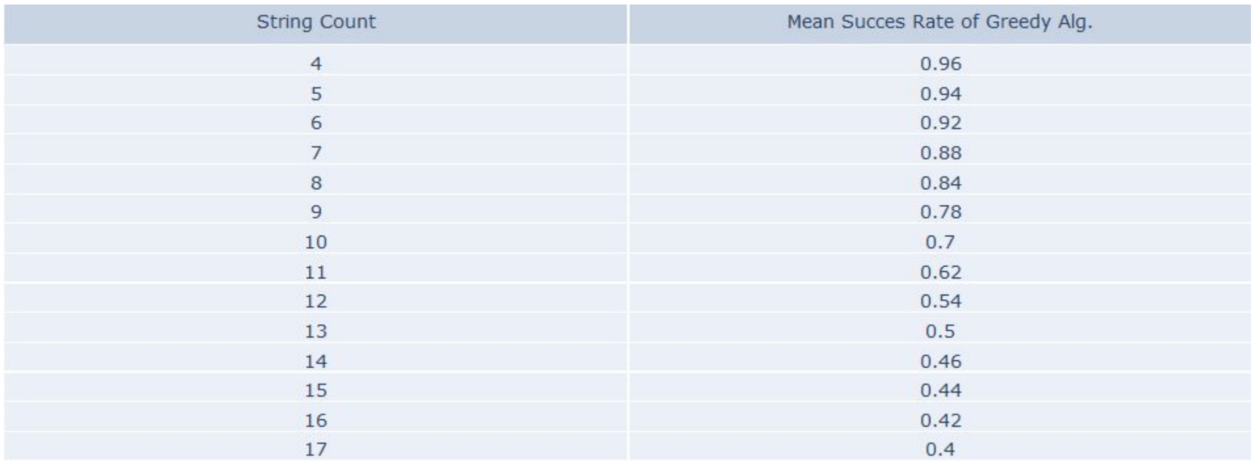


Figure 11, String Count and Error Count Relation



String Count	Mean Succes Rate of Greedy Alg.
4	0.96
5	0.94
6	0.92
7	0.88
8	0.84
9	0.78
10	0.7
11	0.62
12	0.54
13	0.5
14	0.46
15	0.44
16	0.42
17	0.4

Figure 12, Success Graph According to String Count

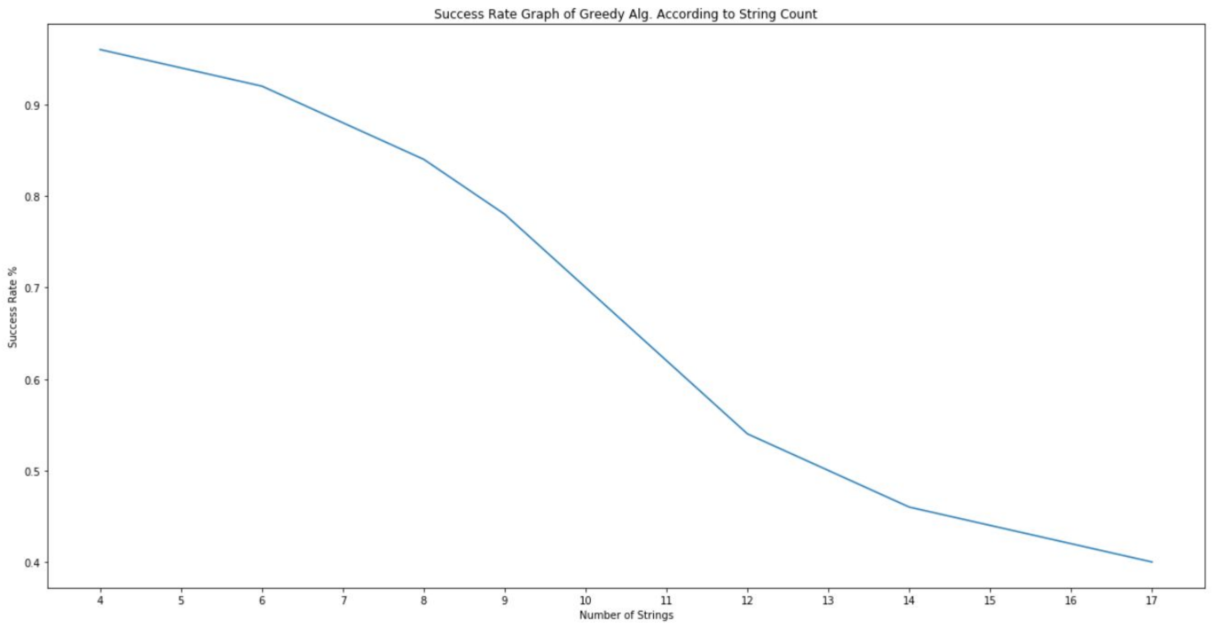


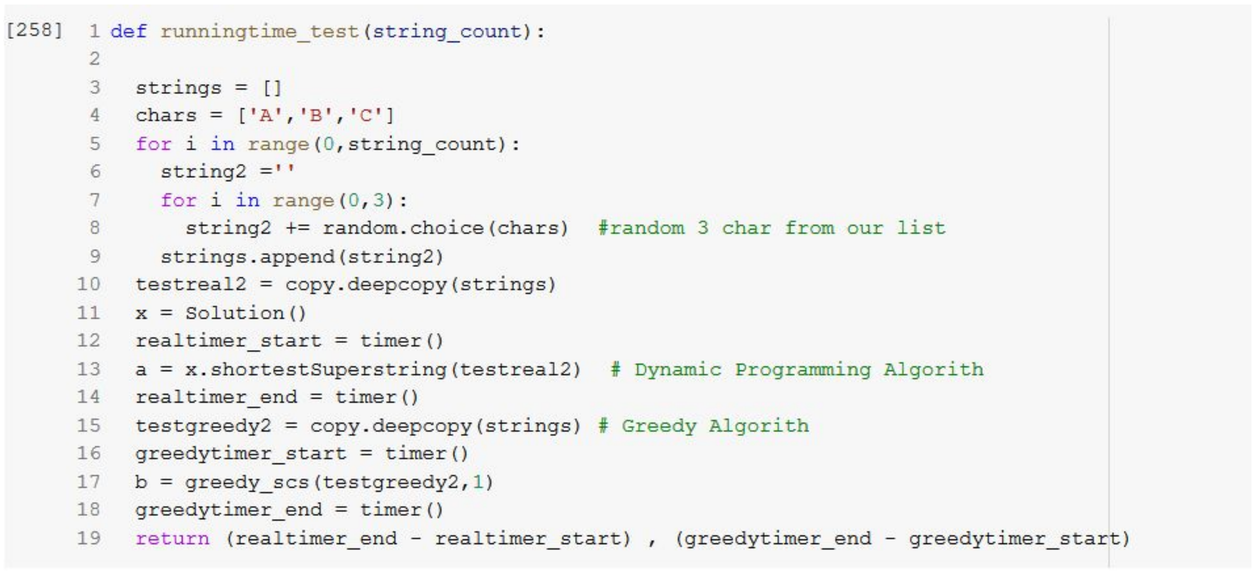
Figure 13, Success Rate Graph of Greedy Alg. According to String Count

It can be understood that while number of strings are increasing, the chance of finding true shortest common superstring of our greedy algorithm is decreasing.

RUNNING TIME EXPERIMENTAL MEASUREMENT

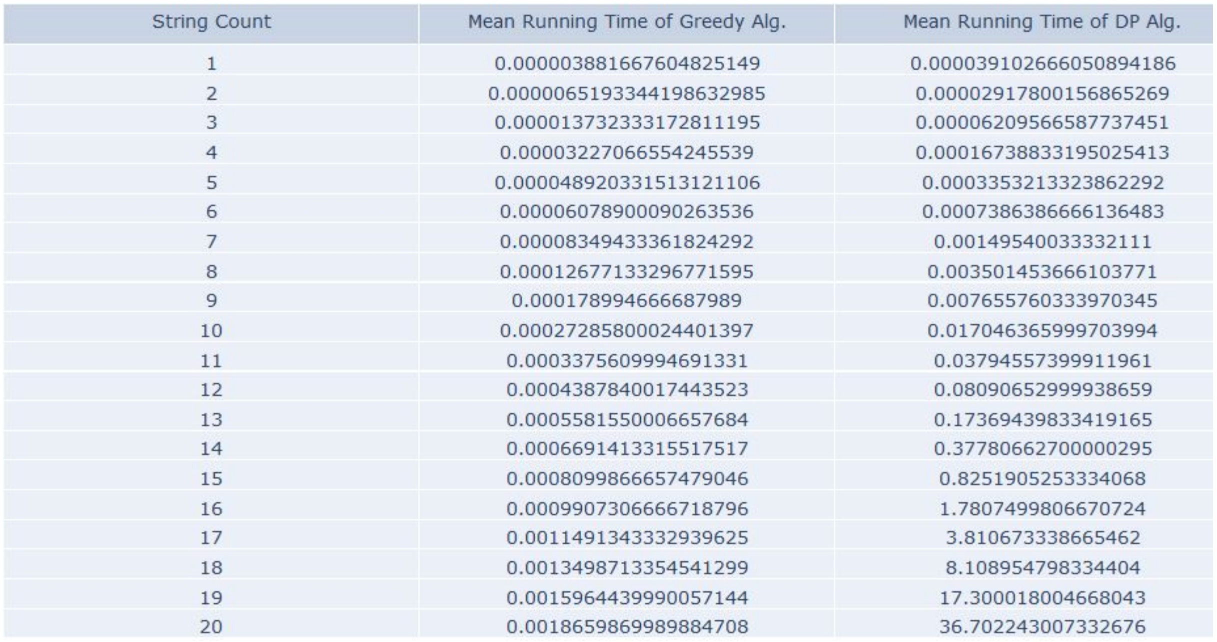
For experimental analysis running time experimental measurement part , we created a function runningtime_test (Figure 14) which gives us running time in seconds. For the sake of accurate running time mean we run this function 20 times with per string count. And store them in a list.

Running Time According to String Count with Fixed Alphabet Size



```
[258]  def runningtime_test(string_count):

         strings = []
         chars = ['A','B','C']
         for i in range(0,string_count):
           string2 =''
           for i in range(0,3):
             string2 += random.choice(chars)  #random 3 char from our list
           strings.append(string2)
         testreal2 = copy.deepcopy(strings)
         x = Solution()
         realtimer_start = timer()
         a = x.shortestSuperstring(testreal2)  # Dynamic Programming Algorith
         realtimer_end = timer()
         testgreedy2 = copy.deepcopy(strings) # Greedy Algorith
         greedytimer_start = timer()
         b = greedy_scs(testgreedy2,1)
         greedytimer_end = timer()
         return (realtimer_end - realtimer_start) , (greedytimer_end - greedytimer_start)
```

Figure 14, Running Time Function



String Count	Mean Running Time of Greedy Alg.	Mean Running Time of DP Alg.
1	0.000003881667604825149	0.000039102666050894186
2	0.0000065193344198632985	0.00002917800156865269
3	0.000013732333172811195	0.00006209566587737451
4	0.00003227066554245539	0.00016738833195025413
5	0.000048920331513121106	0.0003353213323862292
6	0.00006078900090263536	0.0007386386666136483
7	0.00008349433361824292	0.00149540033332111
8	0.00012677133296771595	0.003501453666103771
9	0.00017899466668798	0.007655760333970345
10	0.00027285800024401397	0.017046365999703994
11	0.0003375609994691331	0.03794557399911961
12	0.0004387840017443523	0.08090652999938659
13	0.0005581550006657684	0.17369439833419165
14	0.0006691413315517517	0.37780662700000295
15	0.0008099866657479046	0.8251905253334068
16	0.0009907306666718796	1.7807499806670724
17	0.0011491343332939625	3.810673338665462
18	0.0013498713354541299	8.108954798334404
19	0.0015964439990057144	17.300018004668043
20	0.0018659869989884708	36.702243007332676

Figure 15, Running Time Table of Greedy Alg. and Dp Alg. According Number of Strings(20 Test)

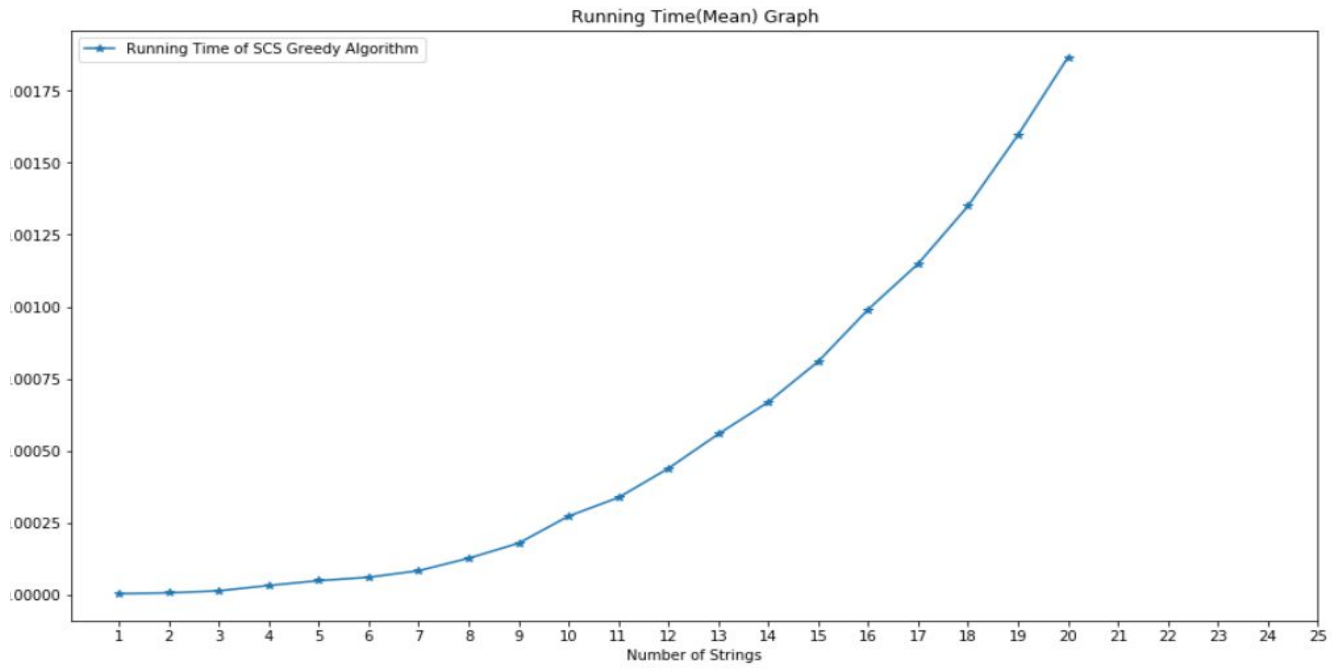


Figure 16, Running Time Graph of Greedy Alg. According to Number of Strings(20 Test)

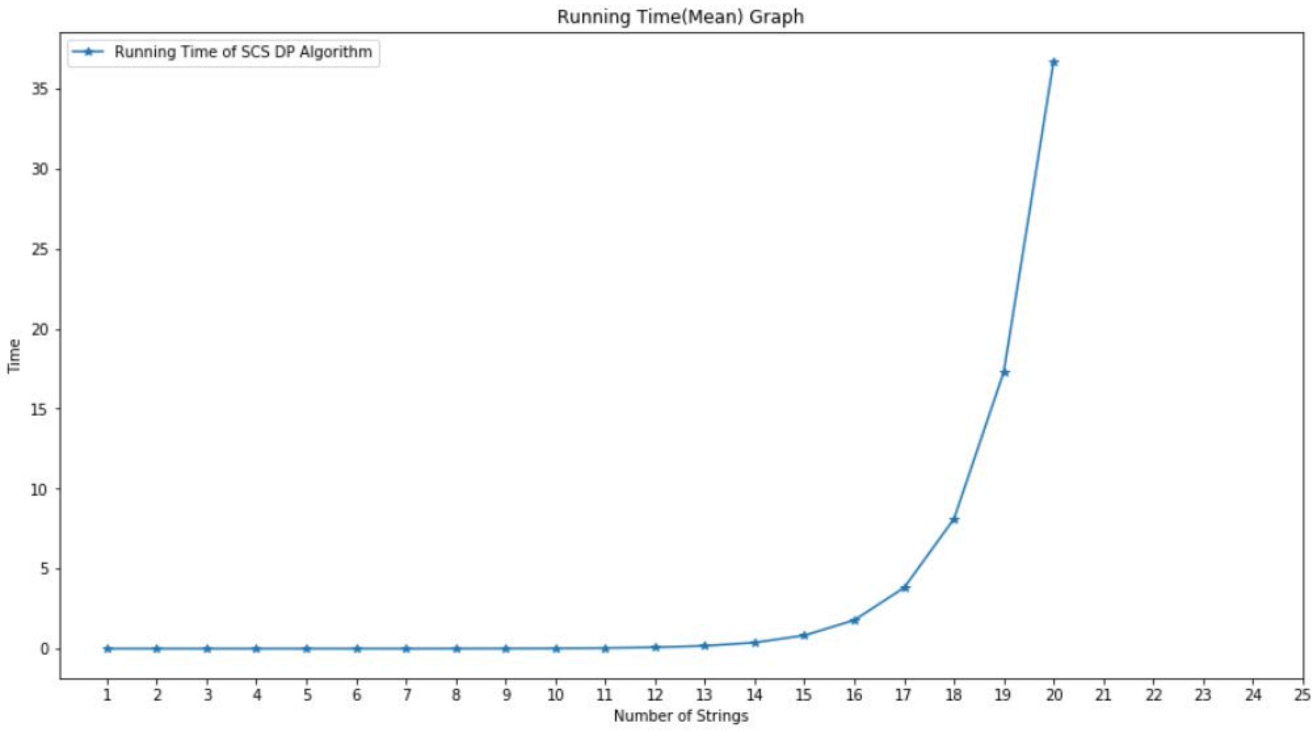


Figure 17, Running Time Graph of DP Alg. According to Number of Strings(Input)

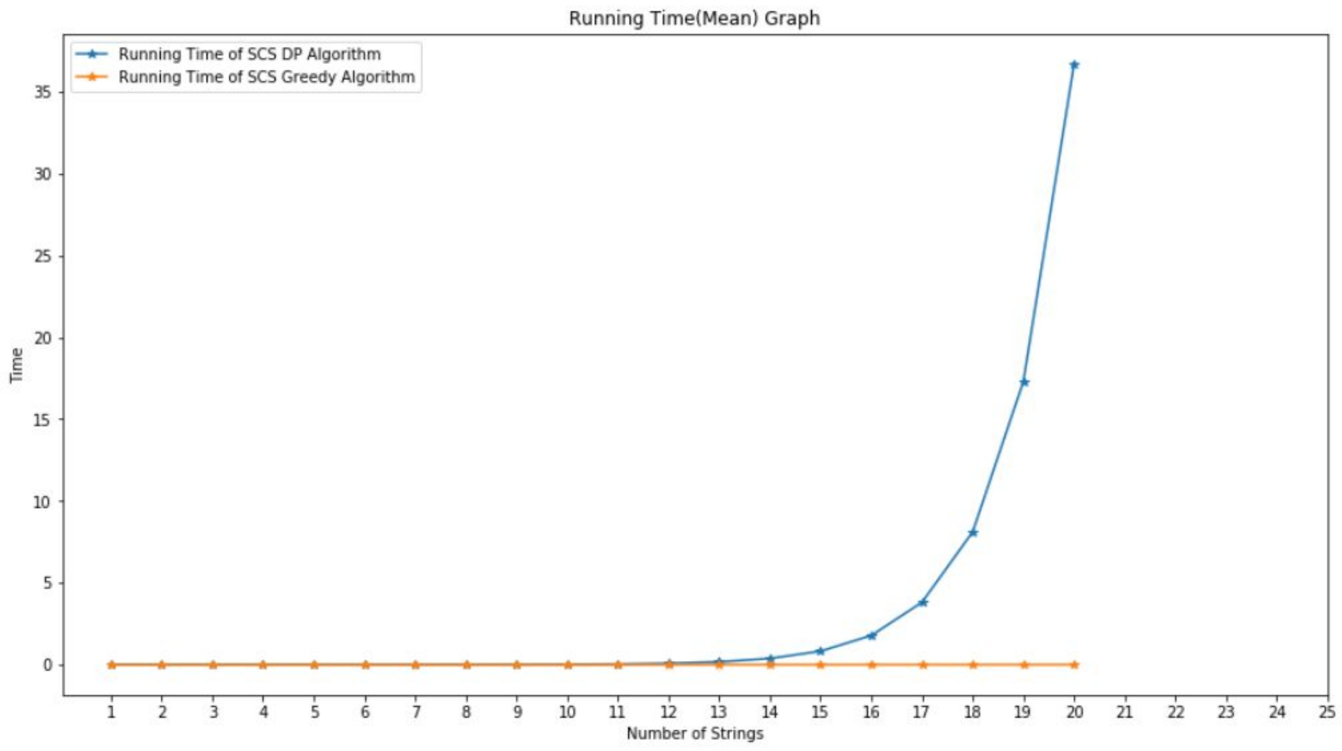


Figure 18, Running Time Graph of DP Alg. and Greedy Alg. According to Number of Strings(Input)

DP Alg. of SCS’ running time and Greedy Alg. of SCS’ running time plotted to graph(figure 18). Even our greedy algorithm have cubic time complexity but DP Alg. of SCS’ time complexity is so much that Greedy Alg. of SCS seems linear in graph. By using Greedy Algorithm we are decreasing of true output chance. But requirement time for algorithm decreases dramatically.

Running Time According to Alphabet Size with Fixed String Count

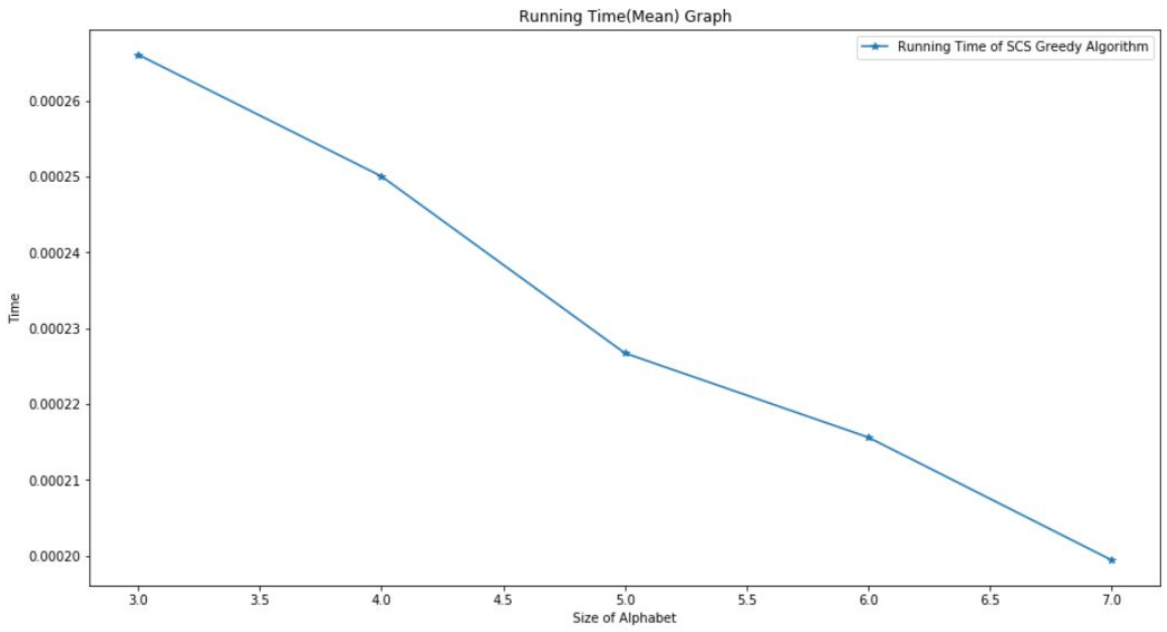
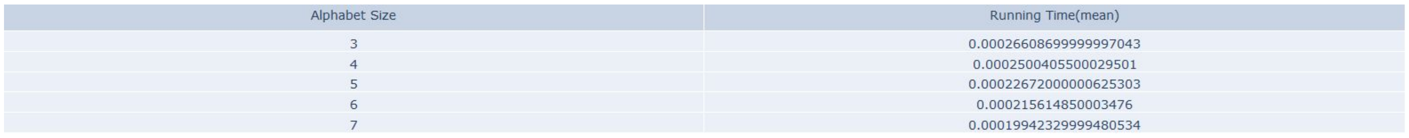


Figure 19 Running Time(mean) Graph According to Alphabet Size with Fixed String Count

As shown in graph when size of Alphabet increases running time of SCS Greedy Algorithm is decreasing. This is result of overlap decreasement between strings with more Alphabet Size. So that loops run less for finding max overlaps.



Alphabet Size	Running Time(mean)
3	0.00026608699999997043
4	0.0002500405500029501
5	0.00022672000000625303
6	0.000215614850003476
7	0.00019942329999480534

Figure 20 Running Time(mean) Table According to Alphabet Size with Fixed String Count

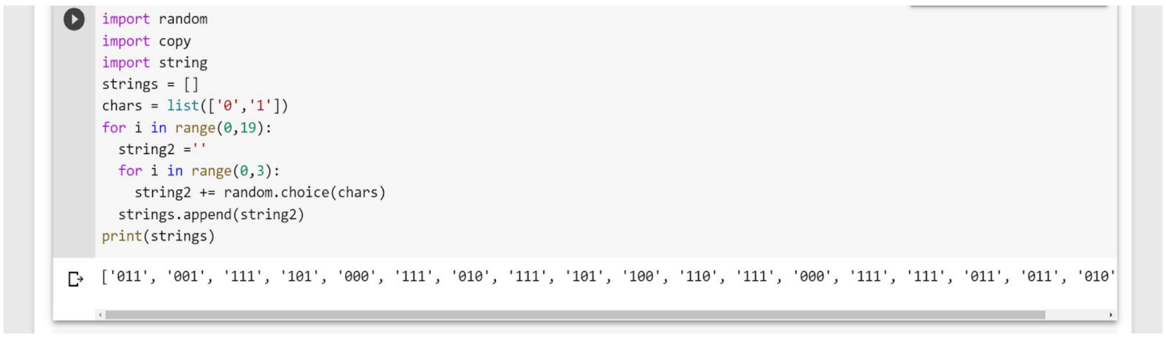
TESTING

Black Box Testing

As our Black Box testing technique we used equivalence class, which divides input into similar classes and checks the output for a single input from each class. If a single element of a group passes the test, we assume that the whole class passes. We divided our inputs into 5 different groups and conducted the test accordingly. The way we conduct tests for each individual group is basically the same. First we generate random strings with lengths set beforehand by range. Later we use greedy algorithm to generate a superstring and check the results by using dynamic algorithm (brute force). If they match that means the class passes the black box test.

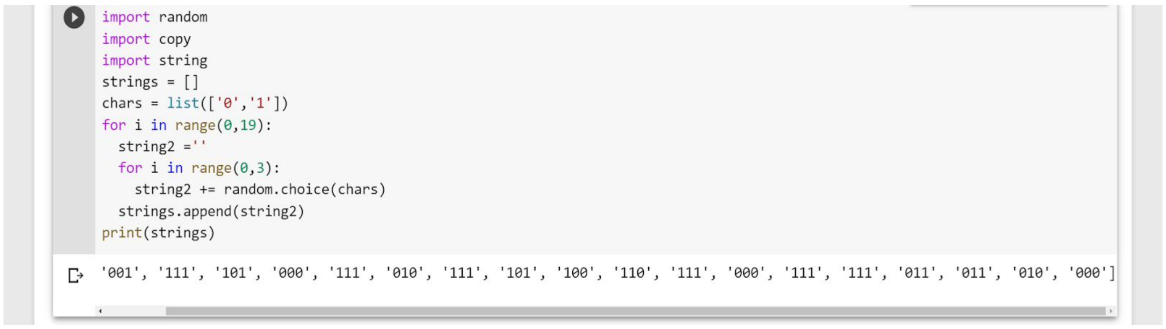
1) Binary Words

Here is a random binary string pool we generated:



```
import random
import copy
import string
strings = []
chars = list(['0','1'])
for i in range(0,19):
  string2 =''
  for i in range(0,3):
    string2 += random.choice(chars)
  strings.append(string2)
print(strings)
```

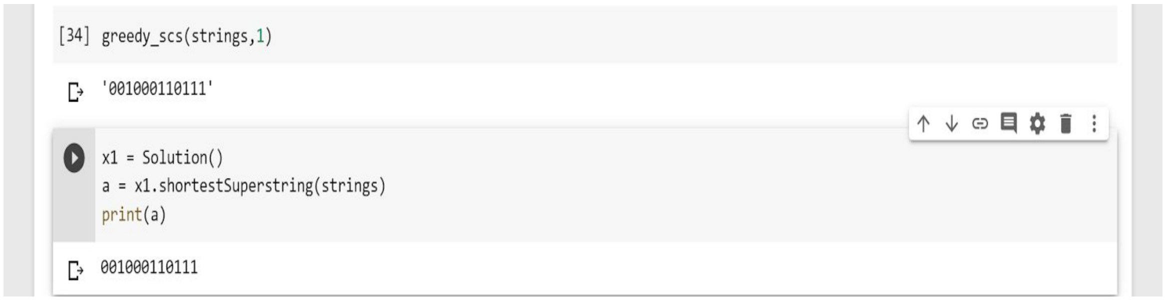
```
['011', '001', '111', '101', '000', '111', '010', '111', '101', '100', '110', '111', '000', '111', '111', '011', '011', '010'
```



```
import random
import copy
import string
strings = []
chars = list(['0','1'])
for i in range(0,19):
  string2 =''
  for i in range(0,3):
    string2 += random.choice(chars)
  strings.append(string2)
print(strings)
```

```
'001', '111', '101', '000', '111', '010', '111', '101', '100', '110', '111', '000', '111', '111', '011', '011', '010', '000']
```

And here you can see the superstring result, which matches both in greedy and dynamic:



```
[34] greedy_scs(strings,1)
```

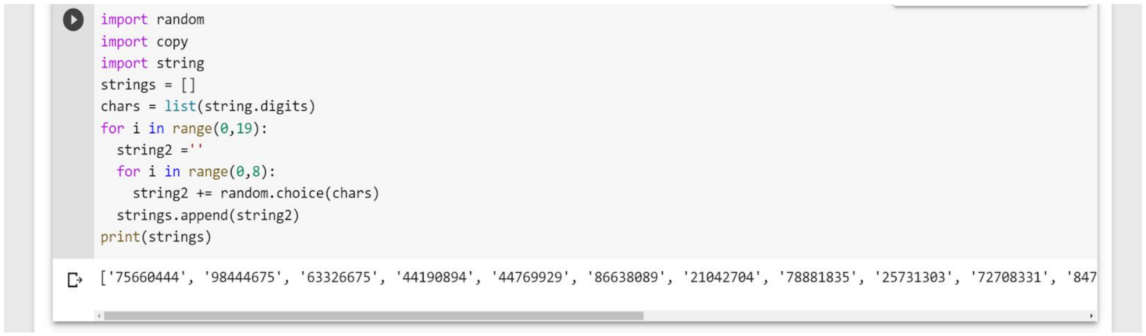
```
'001000110111'
```

```
x1 = Solution()
a = x1.shortestSuperstring(strings)
print(a)
```

```
001000110111
```

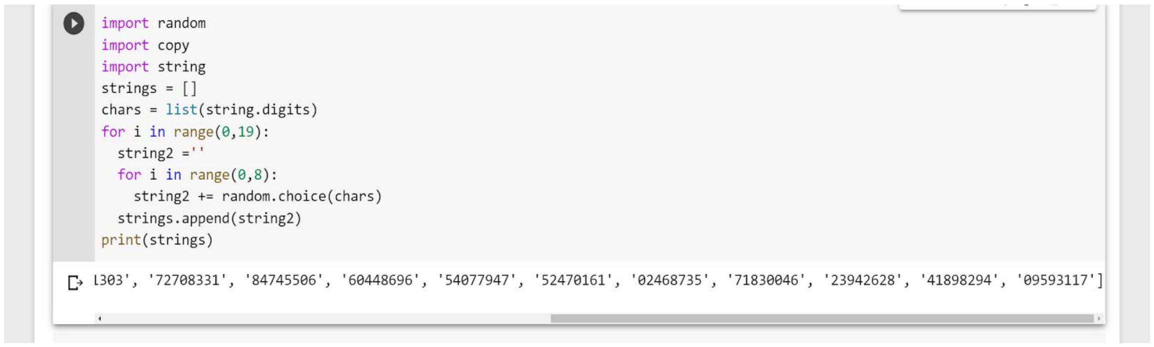
2) Alphabet 0-9

Random generated string pool:



```
import random
import copy
import string
strings = []
chars = list(string.digits)
for i in range(0,19):
  string2 =''
  for i in range(0,8):
    string2 += random.choice(chars)
  strings.append(string2)
print(strings)
```

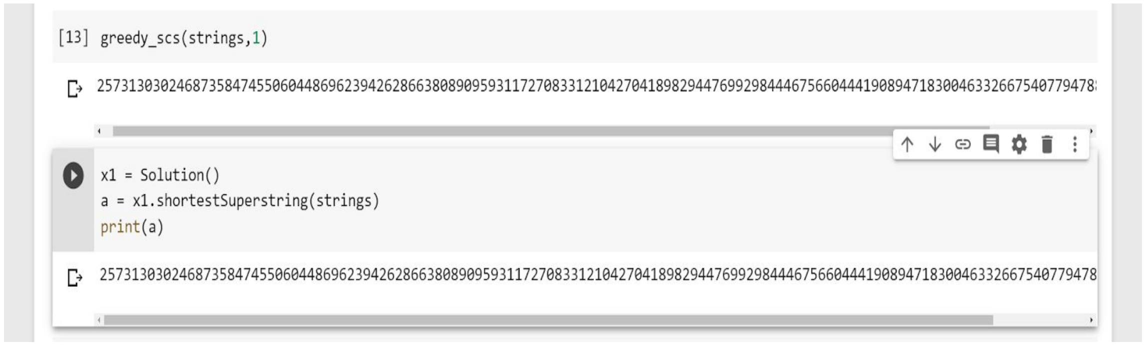
```
['75660444', '98444675', '63326675', '44190894', '44769929', '86638089', '21042704', '78881835', '25731303', '72708331', '847
```



```
import random
import copy
import string
strings = []
chars = list(string.digits)
for i in range(0,19):
  string2 =''
  for i in range(0,8):
    string2 += random.choice(chars)
  strings.append(string2)
print(strings)
```

```
l303', '72708331', '84745506', '60448696', '54077947', '52470161', '02468735', '71830046', '23942628', '41898294', '09593117']
```

Resulting superstring:

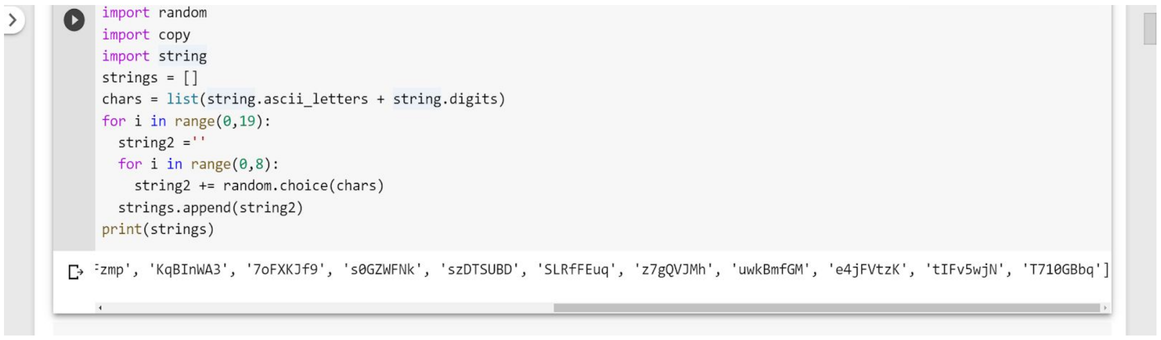


```
[13] greedy_scs(strings,1)

25731303024687358474550604486962394262866380890959311727083312104270418982944769929844467566044419089471830046332667540779478
```

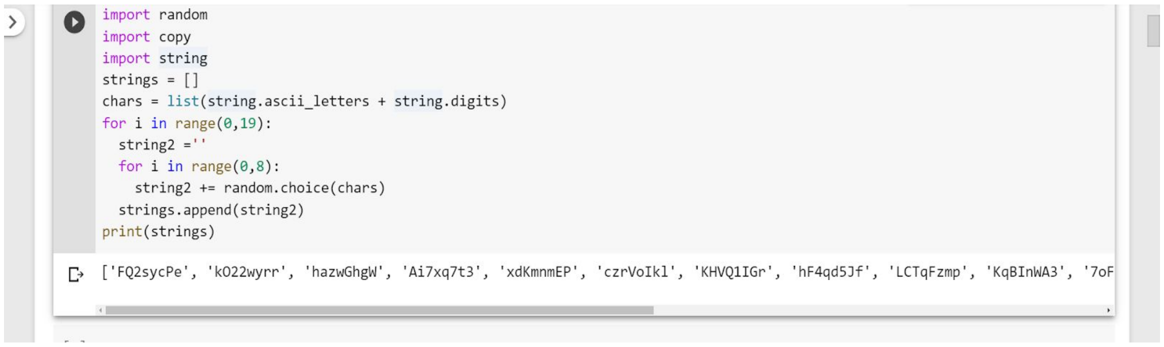
```
x1 = Solution()
a = x1.shortestSuperstring(strings)
print(a)

25731303024687358474550604486962394262866380890959311727083312104270418982944769929844467566044419089471830046332667540779478
```

3) Alphabet a-z, A-Z, 0-9

Random generated string pool:

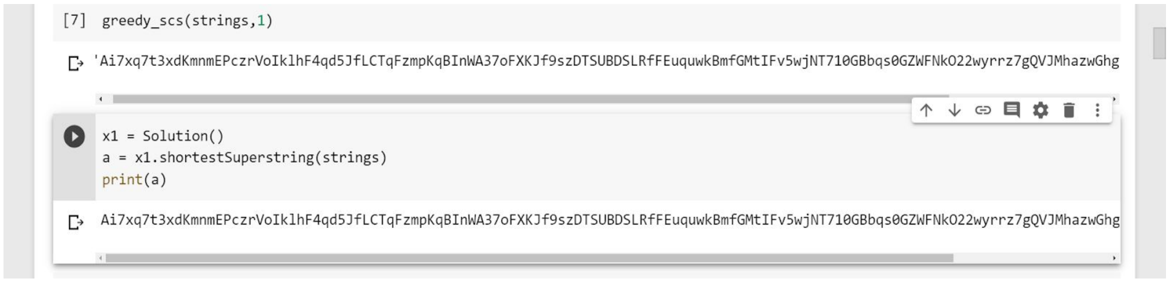


```
import random
import copy
import string
strings = []
chars = list(string.ascii_letters + string.digits)
for i in range(0,19):
  string2 =''
  for i in range(0,8):
    string2 += random.choice(chars)
  strings.append(string2)
print(strings)

zmp', 'KqBInWA3', '7oFXKJf9', 's0GZWFNk', 'szDTSUBD', 'SLRfFEuq', 'z7gQVJMh', 'uwkBmfGM', 'e4jFVtzK', 'tIFv5wjN', 'T710GBbq']
```



```
import random
import copy
import string
strings = []
chars = list(string.ascii_letters + string.digits)
for i in range(0,19):
  string2 =''
  for i in range(0,8):
    string2 += random.choice(chars)
  strings.append(string2)
print(strings)

['FQ2sycPe', 'kO22wyrr', 'hazwGhgW', 'Ai7xq7t3', 'xdKmnmEP', 'czrVoIkl', 'KHVQ1IGr', 'hF4qd5Jf', 'LCTqFzmp', 'KqBInWA3', '7oF
```

Resulting superstring:



```
[7] greedy_scs(strings,1)
```

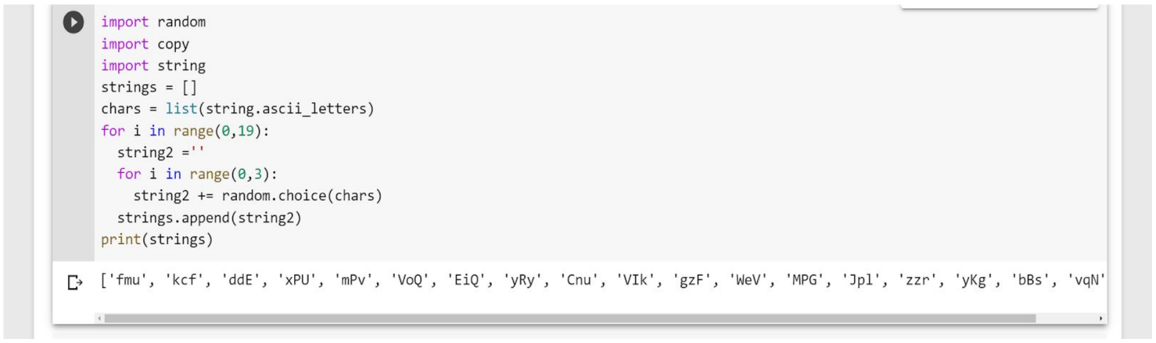
```
'Ai7xq7t3xdKmnmEPczrVoIklhF4qd5JfLCTqFzmpKqBInWA37oFXKJf9szDTSUBDSLRfFEuquwkBmfGMtIFv5wjNT710GBbqs0GZWFNkO22wyrrz7gQVJMhazwGhg
```

```
x1 = Solution()
a = x1.shortestSuperstring(strings)
print(a)
```

```
Ai7xq7t3xdKmnmEPczrVoIklhF4qd5JfLCTqFzmpKqBInWA37oFXKJf9szDTSUBDSLRfFEuquwkBmfGMtIFv5wjNT710GBbqs0GZWFNkO22wyrrz7gQVJMhazwGhg
```

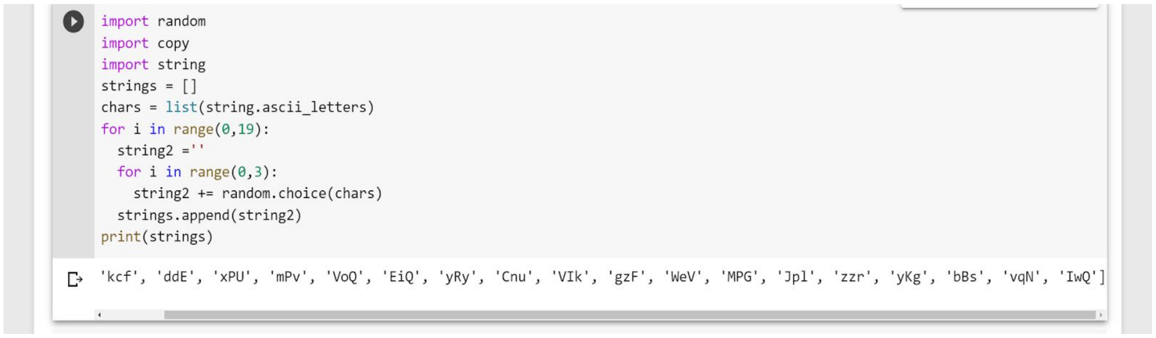
4) Short Words with a length between 2-4

Random generated string pool:



```
import random
import copy
import string
strings = []
chars = list(string.ascii_letters)
for i in range(0,19):
  string2 =''
  for i in range(0,3):
    string2 += random.choice(chars)
  strings.append(string2)
print(strings)
```

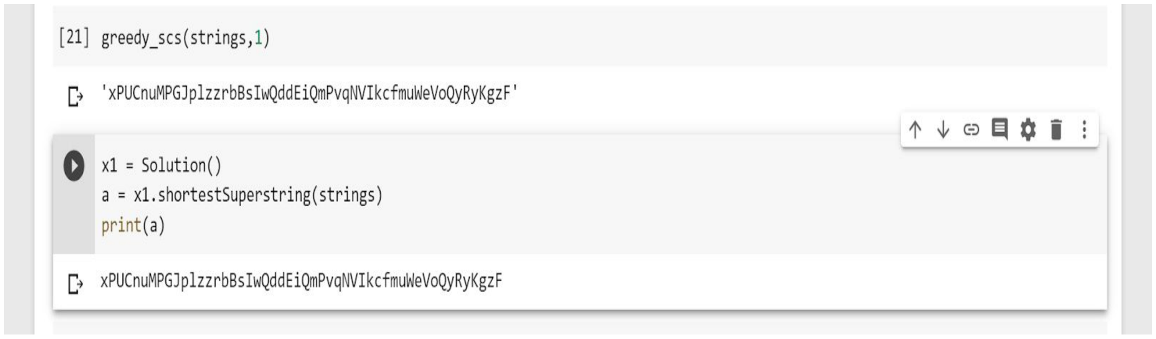
```
['fmu', 'kcf', 'ddE', 'xPU', 'mPv', 'VoQ', 'EiQ', 'yRy', 'Cnu', 'VIk', 'gzF', 'WeV', 'MPG', 'Jpl', 'zzr', 'yKg', 'bBs', 'vqN'
```



```
import random
import copy
import string
strings = []
chars = list(string.ascii_letters)
for i in range(0,19):
  string2 =''
  for i in range(0,3):
    string2 += random.choice(chars)
  strings.append(string2)
print(strings)
```

```
'kcf', 'ddE', 'xPU', 'mPv', 'VoQ', 'EiQ', 'yRy', 'Cnu', 'VIk', 'gzF', 'WeV', 'MPG', 'Jpl', 'zzr', 'yKg', 'bBs', 'vqN', 'IwQ']
```

Resulting superstring:



```
[21] greedy_scs(strings,1)
```

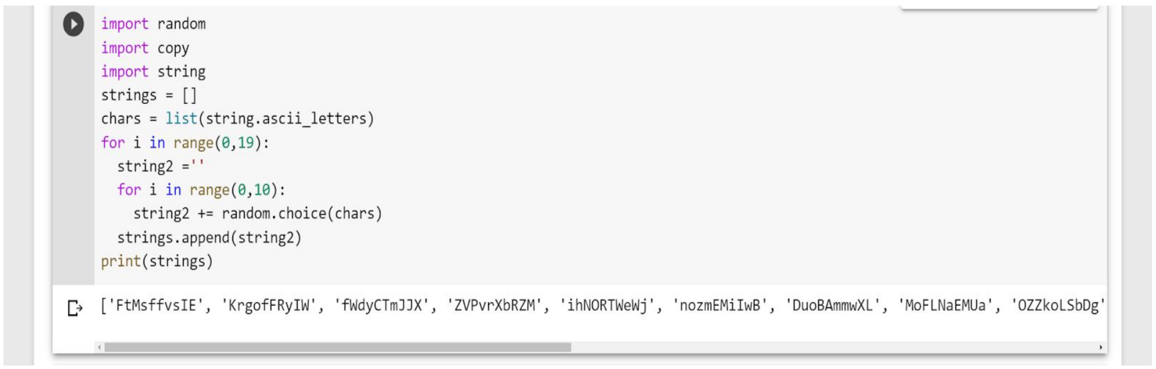
```
'xPUCnuMPGJplzzrbBsIwQddEiQmPvqNVIkcfmuWeVoQyRyKgzF'
```

```
x1 = Solution()
a = x1.shortestSuperstring(strings)
print(a)
```

```
xPUCnuMPGJplzzrbBsIwQddEiQmPvqNVIkcfmuWeVoQyRyKgzF
```

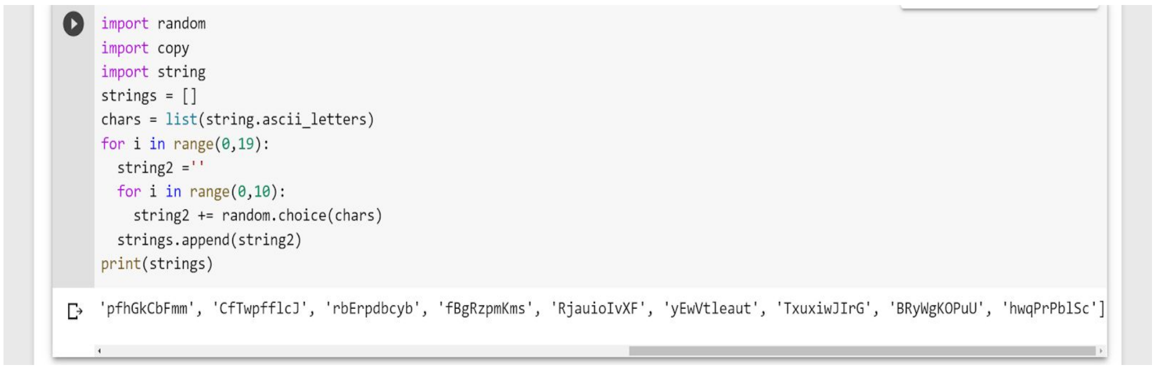
*5) **Long Words with a length between 10-20***

Random generated string pool:



```
import random
import copy
import string
strings = []
chars = list(string.ascii_letters)
for i in range(0,19):
  string2 =''
  for i in range(0,10):
    string2 += random.choice(chars)
  strings.append(string2)
print(strings)
```

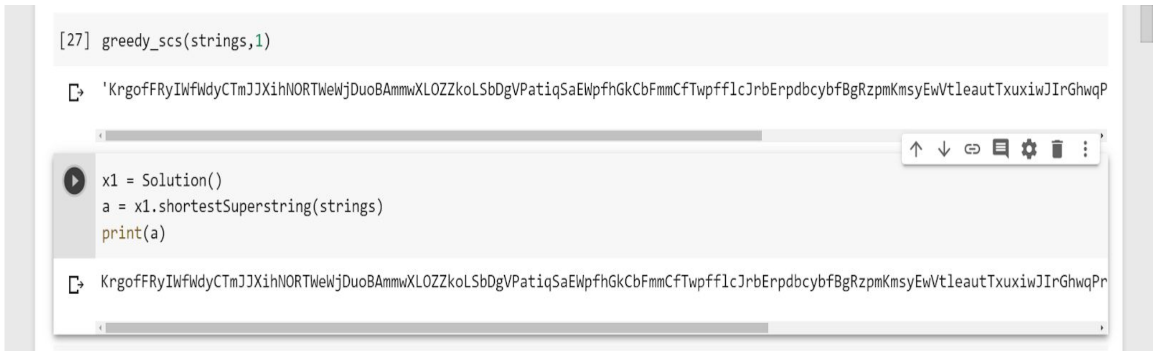
```
['FtMsffvsIE', 'KrgofFRyIW', 'fWdyCTmJJX', 'ZVPvrXbRZM', 'ihNORTWeWj', 'nozmEMiIwB', 'DuoBAmmwXL', 'MoFLNaEMUa', 'OZZkoLSbDg'
```



```
import random
import copy
import string
strings = []
chars = list(string.ascii_letters)
for i in range(0,19):
  string2 =''
  for i in range(0,10):
    string2 += random.choice(chars)
  strings.append(string2)
print(strings)
```

```
'pfhGkCbFmm', 'CfTwpfflcJ', 'rbErpdbcyb', 'fBgRzpmKms', 'RjauioIvXF', 'yEwVtleaut', 'TxuxiwJIrG', 'BRyWgKOPuU', 'hwqPrPblSc']
```

Resulting superstring:



```
[27] greedy_scs(strings,1)

'KrgofFRyIWfWdyCTmJJXihNORTWeWjDuoBAmmwXLOZZkoLSbDgVPatiqSaEWpfhGkCbFmmCfTwpfflcJrbErpdbcybfBgRzpmKmsyEwVtleautTxuxiwJIrGhwqP
```

```
x1 = Solution()
a = x1.shortestSuperstring(strings)
print(a)

KrgofFRyIWfWdyCTmJJXihNORTWeWjDuoBAmmwXLOZZkoLSbDgVPatiqSaEWpfhGkCbFmmCfTwpfflcJrbErpdbcybfBgRzpmKmsyEwVtleautTxuxiwJIrGhwqPr
```

Testing Results:

After testing our algorithm with each string class, we got satisfactory results. We got matching outputs with both greedy algorithm and dynamic algorithm (brute force) while applying them to our various different input classes mentioned above.

CONCLUSION

To sum up all, Shortest Common Substring problem is a NP-Complete problem which we reduced the Vertex Cover problem to Hamiltonian Path problem and then to Hamiltonian Circuit which can be done in polynomial time. We analyzed the algorithm and this algorithm does not necessarily give the shortest common substring.

After experimental analysis of the algorithm, we used 4 strings and the correctness of the algorithm was 96%, when we increased the number of strings the correctness of the algorithm decreased to (94%, 92%, 88%,84%).The reason for the decrease is that greedy algorithm finds maximum overlapping pairs and chooses one arbitrary pair and unions them until there is only one string left but chosen pair of string does not always result in shortest common superstring. The increase in string number leads to decrease in correctness because the number of overlapping pairs increases.

If we take into consideration of experimental analysis running time and algorithm analysis, our algorithm shows consistent characteristics of cubic type functions .

REFERENCES

- Brute Force Algorithm of SCS. (n.d.). Retrieved December 20, 2019, from https://leetcode.com/problems/find-the-shortest-superstring/discuss/233056/A-slow-but-easy-to-understand-Python-solution.

- Shortest Superstring Problem(Greedy Algorithm). (2017, March 22). Retrieved December 20, 2019, from https://www.geeksforgeeks.org/shortest-superstring-problem/.

- Tarhio, J., & Ukkonen, E. (n.d.). Hamiltanion Path. Retrieved December 20, 2019, from http://www.cs.hut.fi/u/tarhio/papers/greedy.pdf.

- Projects, C. to W. (2019, December 18). Shortest common supersequence problem. Retrieved December 20, 2019, from https://www.wikizeroo.org/index.php?q=aHR0cHM6Ly9lbi5tLndpa2lwZWRpYS5vcmcvd2lraS9TaG9ydGVzdF9jb21tb25fc3VwZXJzZXF1ZW5jZV9wcm9ibGVt.

- Using Set Cover. (n.d.). Retrieved December 20, 2019, from https://www.geeksforgeeks.org/shortest-superstring-problem-set-2-using-set-cover/amp/.

- Langmead, B. (n.d.). Assembly SCS. Retrieved December 20, 2019, from http://www.cs.jhu.edu/~langmea/resources/lecture_notes/16_assembly_scs_v2.pdf.