Min-Set-cover problem analysis

CS 301 TERM PROJECT

Fatih Durmaz, Fatih Cemil Demir, Melih Taha Özyurt, Nureddin Kamadan

Sabancı University

Table of Contents

PROBLEM DESCRIPTION

Intuitively, the cover of a set is a set of sets that union of its elements includes all the elements of the first set.

Formally, It is given two sets. First one is a set of sets, **C**. Second one is **S**. Lets say **D** is also a set of sets. **D** forms a cover for **S** if and only if every element in **S** belongs to at least one set in **D**. The problem which is going to be analysed in this document is to find a subset of **C** that has the minimum cardinality among all other subsets of **C** that covers **S**.

This problem can be used in practice. For example let's say a company wants to form a team that satisfies the translation service for following languages:

Needed Languages: { French, German, English, Chinese, Turkish }

In order to create this team, company needs to hire employees from a pool of candidates such that,

Candidates:{ a, b, c, d, e }

a: {French, German, English, Chinese, Japanese}

b: {German, English, Chinese}

c: {Turkish}

d: {English, Chinese}

e: {French}

Company needs to minimize the expenditure and hire employees accordingly. Therefore, the company needs to come up with the smallest subset of candidates that all together satisfies all required languages. Creating a team that satisfies the requirements with the smallest size is actually the minimum cover problem.

To prove a given problem is NP-Complete or not, two things must be proven:

1) Min-Cover problem is in NP.

2) An NP complete problem can be reduced to the Min-Cover problem in polynomial time.

Min-Cover problem is in NP

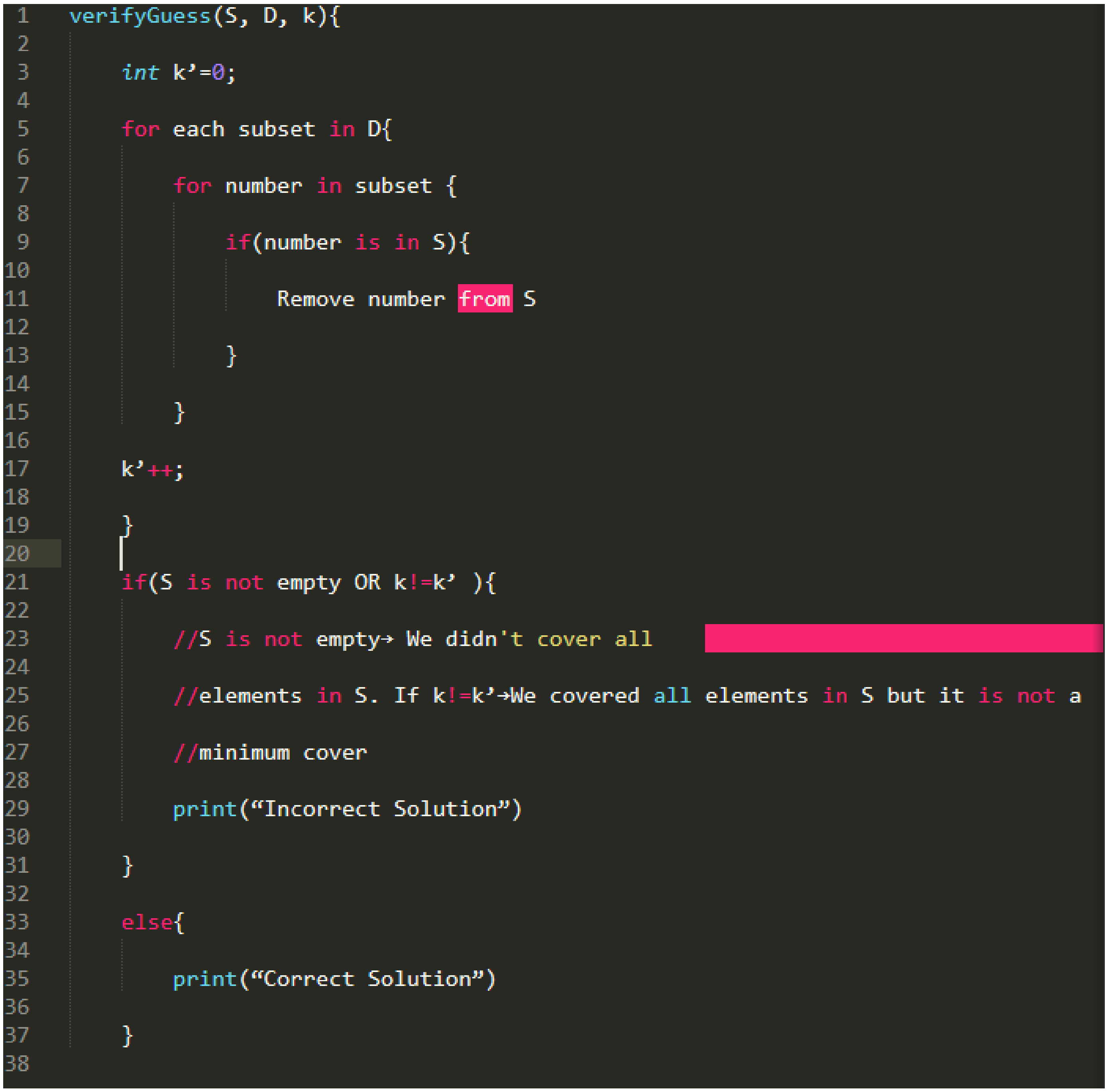
If any problem is in NP, then, given a guess (a solution) to the problem and an instance of the problem (a set S and a positive integer k which indicates the size of minimum cover, in this case), we will be able to verify (check whether the solution given is correct or not) the guess in polynomial time. For example in a given set S and a guess as below, we should be able to verify if this guess is a min-cover for this set or not in polynomial time. In this case, it is obvious that our guess is a min-cover for the set S;

S={1,2,3,4,5,6,7}

C={(1,2,3) ; (1,2) ; (3,5) ; (4) ; (4,6) ; (1,4,5) ; (2,4,7))}

Possible guess: D={(3,5) ; (4,6) ; (1,4,5) ; (2,4,7)}

The guess for the Min-Cover problem is a subset D of C which contains the subsets in the min-cover. The set D is a subset of C which is the set of the subsets of S. We can check whether the set D is a min-cover using the following algorithm:



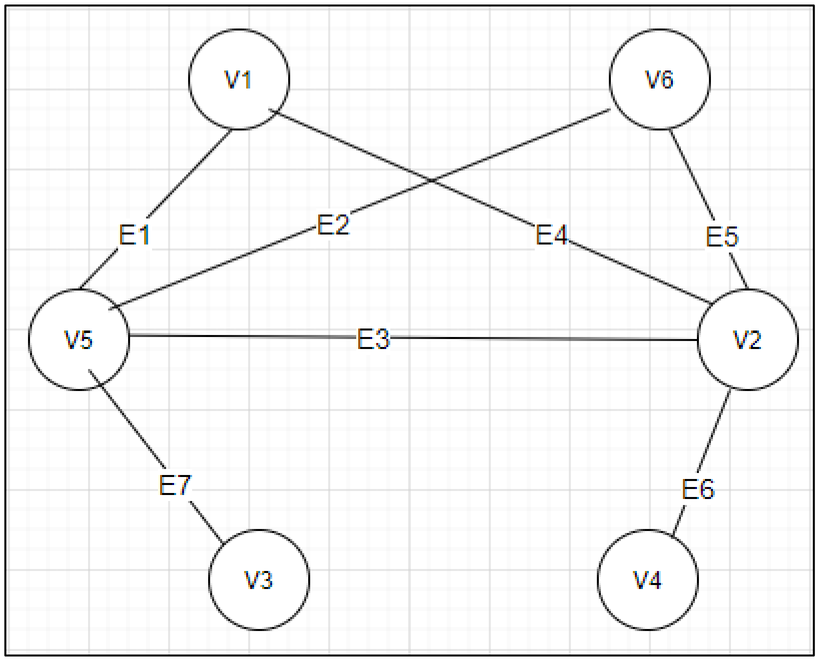
```
verifyGuess(S, D, k){

    int k’=0;

    for each subset in D{

        for number in subset {

            if(number is in S){

                Remove number from S

            }

        }

    k’++;

    }

    if(S is not empty OR k!=k’ ){

        //S is not empty→ We didn't cover all

        //elements in S. If k!=k’→We covered all elements in S but it is not a

        //minimum cover

        print(“Incorrect Solution”)

    }

    else{

        print(“Correct Solution”)

    }

```

The NP Complete Problem can be reduced to Min-Cover in polynomial time

We can prove the Minimum-Cover Problem with a reduction algorithm that can reduce from Vertex Cover Problem to Min-Set Cover in polynomial time.

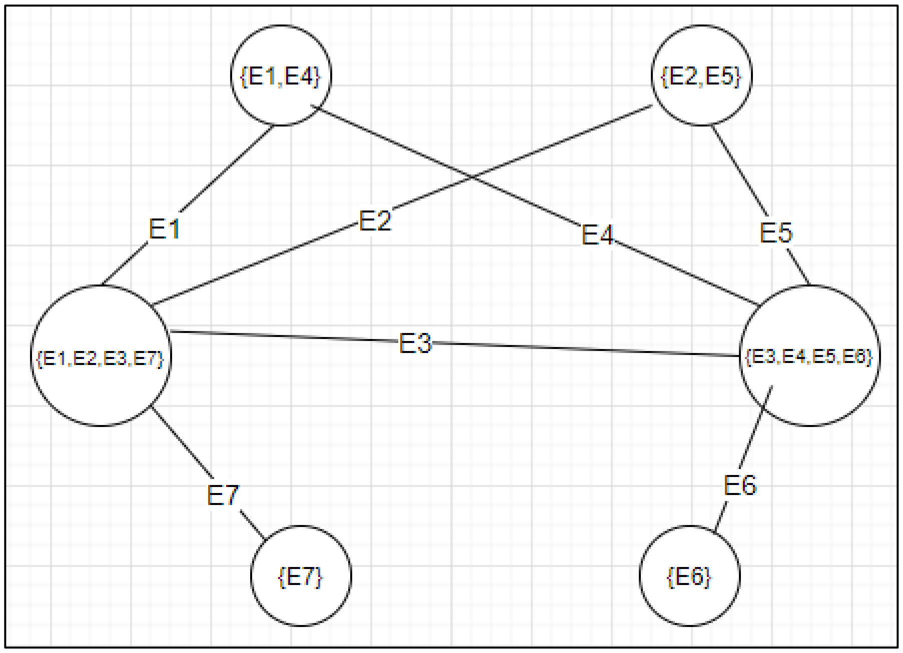
(MIN-VERTEX COVER ≤p MIN-SET-COVER)

Given a graph G(V, E) and we want to find a vertex cover. Normally a vertex set would look like V={V1,V2,V3,V4,V5,V6} and an edge set would look like E={E1,E2,E3,E4,E5,E6,E7}. In this case, in order to show the reduction we can define our initial set S given in the problem as the set of edges. In the minimum cover problem our aim is to choose a minimum number of nodes to cover all edges. Corresponding to each edge in the graph G, assign an element from the set S. Corresponding to each vertex, assign an element from the set C whose elements are an arbitrary subset of S.



Given any instance from Min Vertex Cover like the graph above, we can convert this instance into an instance of Minimum Cover problem. The strategy is to assign all edges to a number. By doing this, our set of edges will be the finite set S which we are given in the problem that we are trying to cover. After that, we will assign each vertex as the union of elements in the edges. For example, for the instance above, the vertices are assigned as such: V1={E1,E4} and V2={E3,E4,E5} and V3 = {E7} and V4 = {E6} and V5={E1,E2,E3,E7} and V6={E2,E5}.

Since we are reducing the min-vertex cover problem into the minimum -cover problem, the solution of the minimum cover to the reduced instances must also give the solution for the min-vertex cover to prove the reduction. Finally, set S that we are given in the problem was reduced by the set of vertices as we decided. As can be seen below, all the



Hence, the instance of the min-vertex cover problem that we have reduced to minimum cover problem is in the form : S={E1,E2,E3,E4,E5,E6,E7} and C = { (E1,E4), (E3,E5,E6), (E7), (E6), (E1,E2,E3,E7), (E2,E5) }. The set D which we will construct as a solution to the min cover problem must also be a solution to the min-vertex cover problem. In this instance the solution set D which is the subset of C with minimum cardinality that covers the set S is : D = { (E1,E2,E3,E7) U (E3,E4,E5,E6) }.

This method can be applied for the all min vertex cover problems. The reason behind this, given a vertex cover instance, the set of the edges of that instance (that also generate a finite set) can be reduced to the set S which is given in the minimum cover problem. In other words, the edges that we are trying to cover with the minimum number of vertices are reduced to the set S in the minimum cover problem that we are trying to cover with the minimum number of subsets from the set D. Hence, any min-vertex cover problem can be transformed as a min-cover problem. Therefore, the solution is also a solution for the minimum vertex cover problem that requires covering all elements with a minimum number of vertices.

For every edge in the instance that we are given from the min-vertex cover problem, we are constructing the set S by assigning each edge. To assign every edge, we have to traverse over all the edges in the graph represented as an adjacency matrix is O(V^2) . For every vertex to find an edge which is connected to that vertex in an undirected graph, is O(V) where V is the number of vertices. Since we do the same thing for all vertices, and we have V vertices, to reduce min vertex problem to minimum cover problem is O(V^2)+ O(V^2) = O(V^2) which is polinomial. Since, min-vertex cover problem is NP hard, min cover problem is also NP hard problem.

ALGORITHM DESCRIPTION

Since the Min-Set Cover problem is an NP-Complete problem a polynomial time solution has not been found yet. However there are heuristic approaches for that problem which solves the problem in polynomial time whereas the solution is not totally correct. In this project a greedy approach for the Min-Set Cover problem will be analysed.

Steps of the Algorithm:

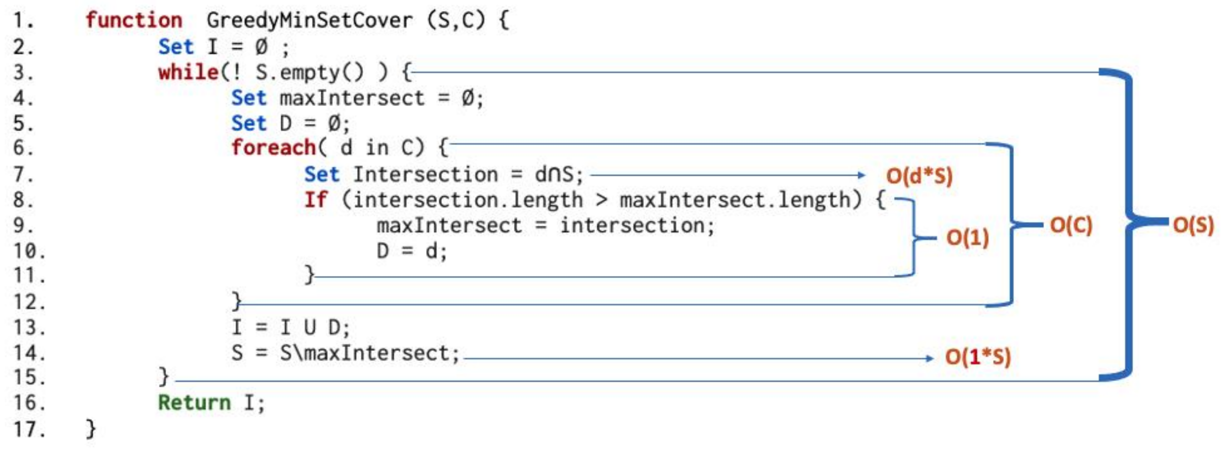
1) Create an empty set for heuristic result
2) Iterate over C (set of sets)
3) For each element of C find the one which has max intersection with S, call it D
4) Remove intersection from S
5) Take union of two sets I and D (element of C that found in step 3)
6) Repeat step 3,4,5 until S becomes empty.

The following pseudocode works in the same manner as described above:

```
function  GreedyMinSetCover (S,C) {
        Set I = Ø ;
        while(! S.empty() ) {
                Set maxIntersect = Ø;
                Set D = Ø;
                foreach( d in C) {
                        Set Intersection = d∩S;
                        If (intersection.length > maxIntersect.length) {
                                maxIntersect = intersection;
                                D = d;
                        }
                }
                I = I U D;
                S = S\maxIntersect;
        }
        Return I;
}
```

This algorithm is considered greedy because at each step, the algorithm looks for an element of C that has the most intersection with S. In other words, while looking for the best option at each step, It can miss the optimal solution.

Worst case asymptotic running time of the algorithm occurs if at each iteration maximum intersection that found covers only 1 element of S, as follows:



```
1.    function  GreedyMinSetCover (S,C) {
2.          Set I = Ø ;
3.          while(! S.empty() ) {
4.                Set maxIntersect = Ø;
5.                Set D = Ø;
6.                foreach( d in C) {
7.                      Set Intersection = d∩S;
8.                      If (intersection.length > maxIntersect.length) {
9.                            maxIntersect = intersection;
10.                           D = d;
11.                     }
12.               }
13.               I = I U D;
14.               S = S\maxIntersect;
15.         }
16.         Return I;
17.   }
```

Total complexity: O(S) * (O(1*S)+O(C)*O(d*S)) = O(S(S + C(d*S))) = O(S*C*d*S)

Calculation of this complexity is straightforward and obvious except 2 steps:

First, at line 7, we have O(d*S). Calculation of intersection of two sets depends on their size independent from each other. For each item in one set, other set should be iterated and found out if there is an intersection or not. We can implement this computation by simply having two nested loops. Therefore, the complexity will be O(d*S).

Second, at line 14 we have O(1*S). For each iteration, maxIntersection will be a set with size 1 for the worst case. Finding and deleting 1 item from set S costs time depending on size of S. Therefore, at line 14, we have O(1*S).

ALGORITHM ANALYSIS

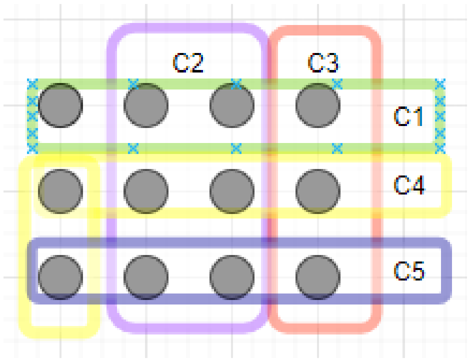
Correctness of the Algorithm

Suppose that the universal set is given as S = {E1,E2,E3,...E_n} and the set C which consists of subsets of S be C = {C1,C2,C3,...,C_m}. At each iteration in the Greedy Algorithm provided above, the C_i which has the maximum potential of covering the elements in S is found and the covered elements are deleted from the set S for further iterations. By doing so, in every iteration, if an element from set C covers at least one element from set S, at least one element from S will be deleted. Program continues to find locally optimal elements from set C which covers the maximum number of elements from set S and delete newly covered elements from S until the set S is empty. At the end, if the program produces a result properly then, our result set will be a min-cover or an approximation of a min-cover. If an element is not found from set C to cover at least one element of S, it means that the set C cannot cover the set S completely and thus, cannot be a cover for set S. In such a case, the algorithm will terminate, and it will be understood that there cannot be any min-cover.

Ratio Bound of the Greedy Approximation Algorithm

As discussed above, there is no polynomial time solution available for the Min-Set Cover problem as we have proven that the problem is NP-Hard problem. However, the Greedy approximate algorithm that we have suggested provides a O(log n) approximate solution.

As we said above the greedy approach provides an approximate solution. Therefore, it may not give a perfect solution. In the figure below, assume black nodes to be the elements of the universal set S and the C1,C2,C3,C4,C5 regions to be the elements of set C which is composed from the subsets of S. Any solution to this problem will be given as the set D. One instance of the solution would be choosing C1,C2,C3,C4 by which the set D becomes D = {C1,C2,C3,C4}. Notice that the optimal set cover in this instance consists of only three elements C1,C4,C5.



The proof of the O(log n) approximation is intuitive as follows:

OPT refers to the optimal solution of the MIN-SET-COVER problem. Let’s assume that we run the greedy algorithm on the set S and subsets of S which are C1,C2, … , C_m. Consider the following example:

C2 = (E1,E2,E3,E4,E5,E6) ,C3 = (E7,E8,E9), C4 = (E10,E11), C1 = (E12) where E1 is the first element covered, E2 second element covered... Also, C2 is the first set picked, C3 is the second, etc. In that example, C2 is the first set selected and it contains 6 new elements. The second selected set which is C3 covers 3 new elements etc. It is clearly seen that since each picked set has the max new element size when they are picked, each successive set covers can have at most the same number of new elements as the previous set. For instance, C2 was the first set that was picked and its size was 6. Therefore, the set that can be picked in the next iteration can have at most 6 new elements.

For each element E_j , let define m_j as the number of elements in the set which contains E_j. For example the set which covers E1 is C2 and its size 6. Therefore $m_1 = 6$. Similarly, E2 covered by C2 so $m_2 = 6$. If we find the all m_j values for our example, it would be as:

$m_1 = 6$, $m_2 = 6$, $m_3 = 6$, $m_4 = 6$, $m_5 = 6$, $m_6 = 6$, $m_7 = 3$, $m_8 = 3$, $m_9 = 3$, $m_{10} = 2$, $m_{11} = 2$, $m_{12} = 1$

Hence, since the sizes of the C_i's non-increasing, we can clearly see

$m_1 >= m_2 >= m_3 >= m_4 >= \ldots.$

Let's set the cost of all C_j as 1. For example, picking C2 has cost 1. Now, we can define cost(E_j) = 1/m_j. If I is the set cover constructed by the greedy algorithm, then;

$$\boxed{| I | = \sum_{j=1}^{n} cost(E_j)}$$

Important step in the proof: Let's consider the situation after elements E1,E2,...E_{j-1} have already been covered and the elements after the index j-1 to the index n are yet to be covered. Notice that n-j+1 elements are still uncovered. We also know that no other set in the set C covers more than m(j) elements because if there were such a set in the C, the greedy algorithm would pick that set firstly since it covers more elements than the set, we picked which covers only j elements. Hence, OPT needs at least n-j+1/m(j) sets to cover the remaining uncovered n-j+1 elements. By this logic, we end up with the inequality:

$$\boxed{\text{OPT} \geq \frac{n-j+1}{m(j)} \geq (n-j+1)cost(E_j)}$$

If we leave cost(E_j) alone, we get the formula:

$$\boxed{cost(E_j) \leq \frac{OPT}{n-j+1}}$$

Finally, we can put all relations together to show that the Greedy Algorithm has a good approximation bound:

$$\boxed{\mid I \mid = \sum_{j=1}^{n} cost(E_{\mathrm{j}})}$$

Since we got cost(E_j) <= OPT / n-j+1 before,

$$\boxed{\leq \sum_{j=1}^{n} \frac{OPT}{n-j+1}}$$

For simplicity, define n-j+1 = i and take OPT out of summation,

$$\boxed{\leq OPT \sum_{j=1}^{n} \frac{1}{i}}$$

This summation is familiar, Harmonic series 1/1 + ½ + ⅓ … is bounded by:

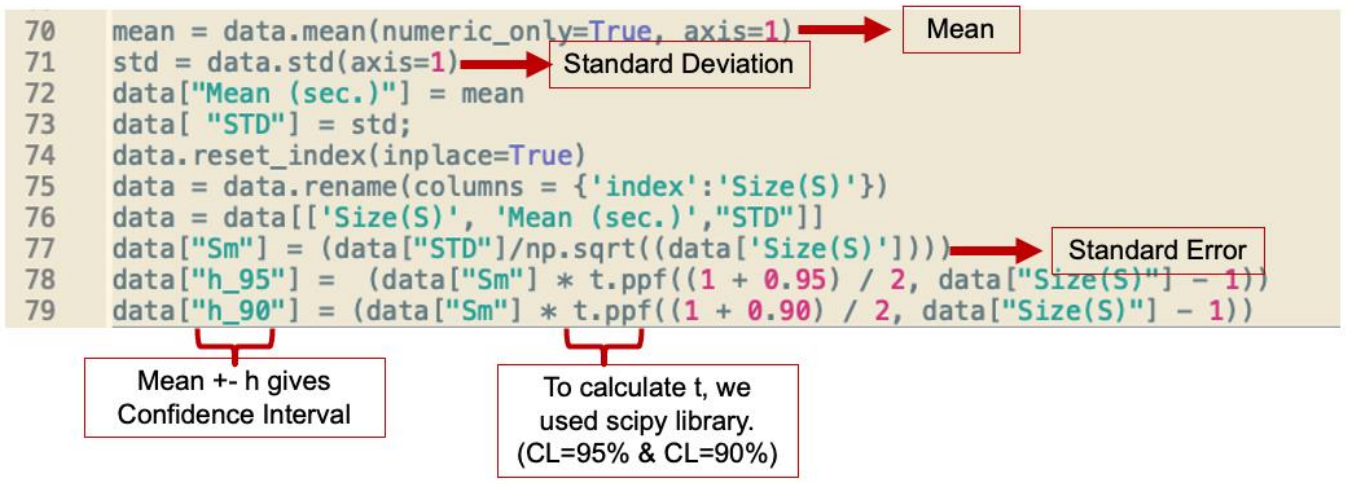
$$\boxed{\leq OPT(\ln n + O(1))}$$

Thus, we have proven that the constructed set cover is O(log n) optimal, in other words the Greedy algorithm that we provided is an O(log n) Approximation algorithm.

EXPERIMENTAL ANALYSIS

Running Time Experimental Analysis

In order to experimentally analyze the algorithm running time complexity, the following functions are used to calculate standard deviation, standard error, sample mean and confidence level intervals. We used Python’s packages for statistical analysis and visualization.



```
mean = data.mean(numeric_only=True, axis=1)
std = data.std(axis=1)
data["Mean (sec.)"] = mean
data[ "STD"] = std;
data.reset_index(inplace=True)
data = data.rename(columns = {'index':'Size(S)'})
data = data[['Size(S)', 'Mean (sec.)',"STD"]]
data["Sm"] = (data["STD"]/np.sqrt((data['Size(S)'])))
data["h_95"] =  (data["Sm"] * t.ppf((1 + 0.95) / 2, data["Size(S)"] - 1))
data["h_90"] = (data["Sm"] * t.ppf((1 + 0.90) / 2, data["Size(S)"] - 1))
```

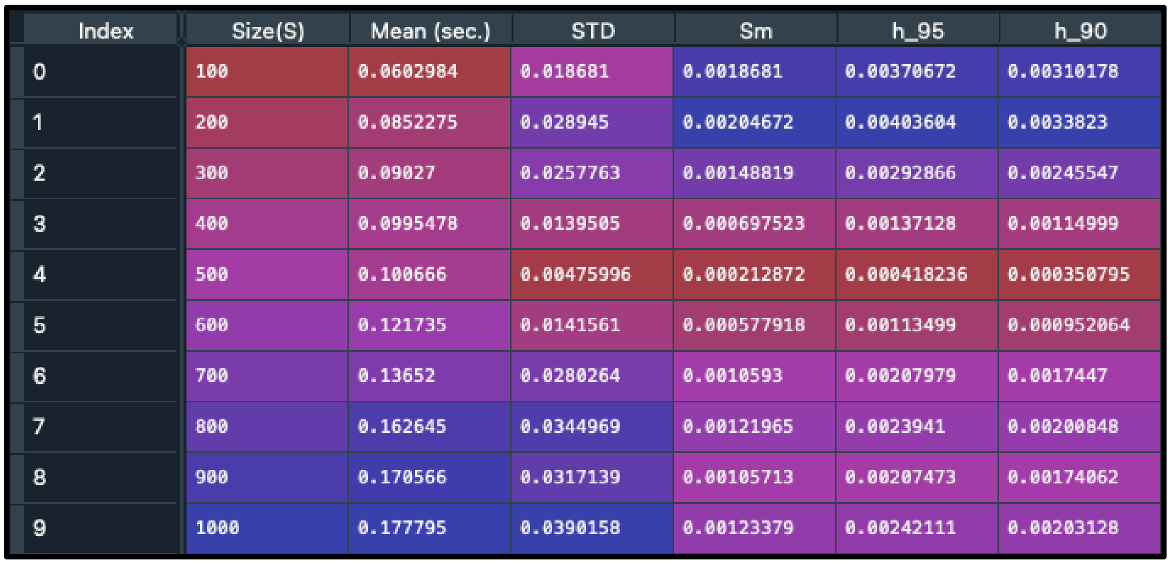
Mean
Standard Deviation
Standard Error
Mean +- h gives Confidence Interval
To calculate t, we used scipy library. (CL=95% & CL=90%)

Note that our approximation algorithm has O(S^2 * d * C) complexity. Therefore, other two inputs will be kept constant while one of them is changing. The data result from testing is given below:

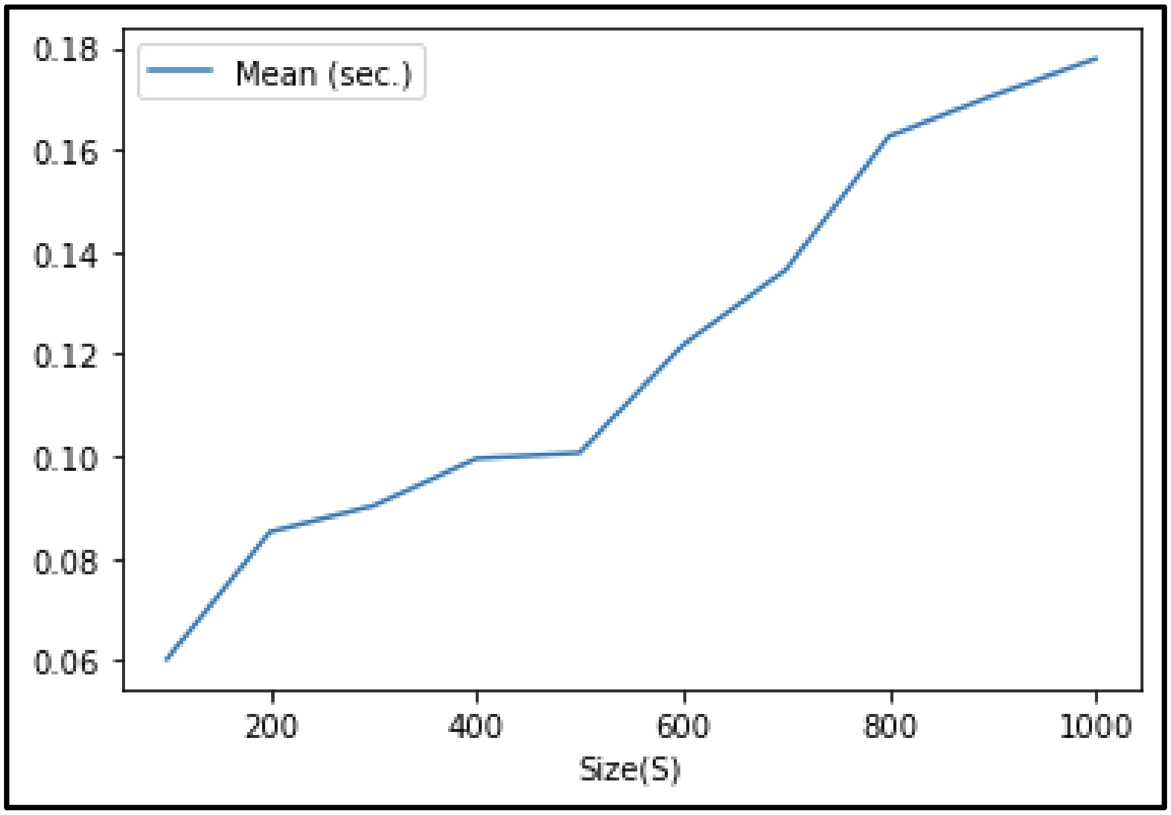
D & C are Constant (50,1000)

S is increasing (100 - 1000)

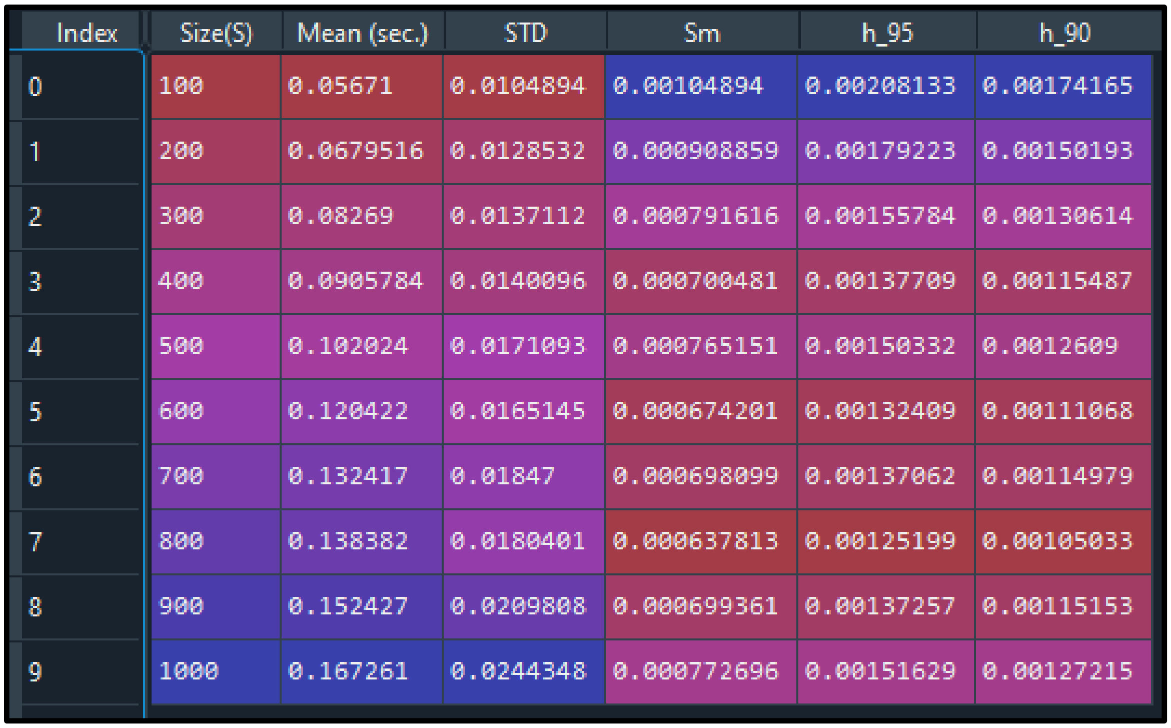
Iteration size 100



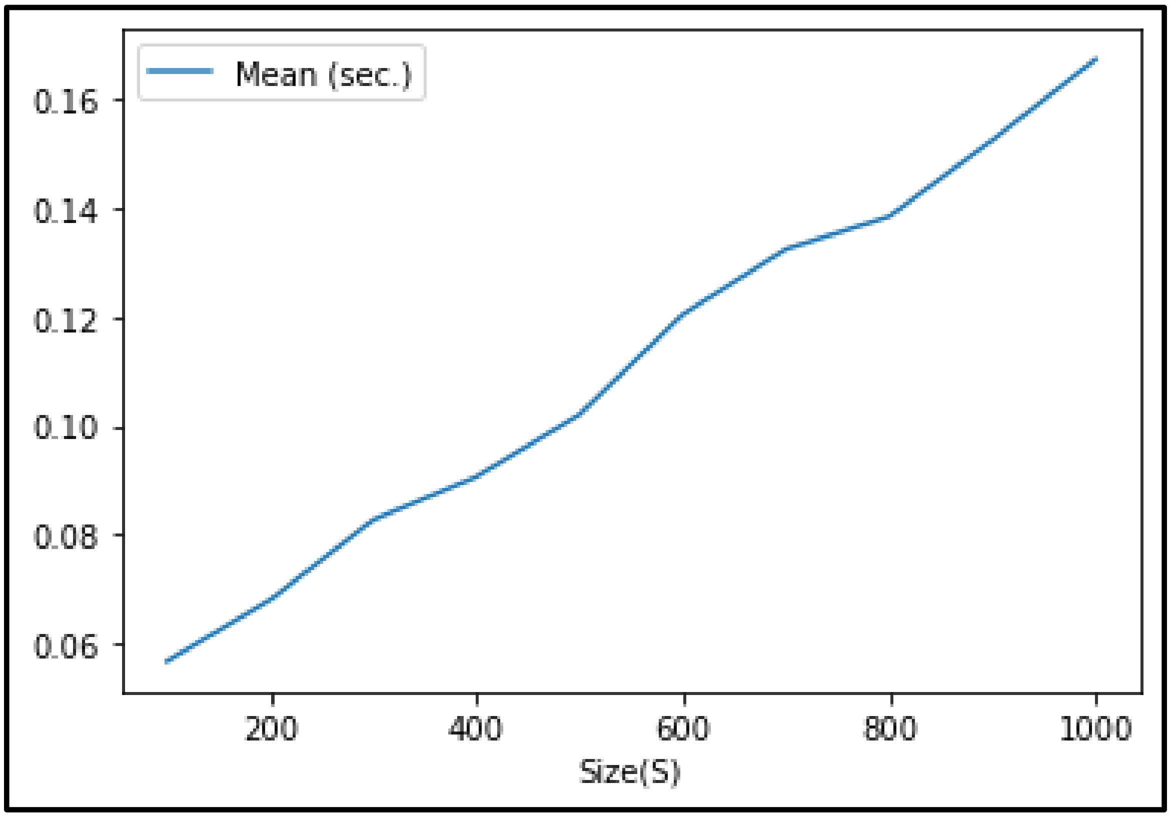
Index	Size(S)	Mean (sec.)	STD	Sm	h_95	h_90
0	100	0.0602984	0.018681	0.0018681	0.00370672	0.00310178
1	200	0.0852275	0.028945	0.00204672	0.00403604	0.0033823
2	300	0.09027	0.0257763	0.00148819	0.00292866	0.00245547
3	400	0.0995478	0.0139505	0.000697523	0.00137128	0.00114999
4	500	0.100666	0.00475996	0.000212872	0.000418236	0.000350795
5	600	0.121735	0.0141561	0.000577918	0.00113499	0.000952064
6	700	0.13652	0.0280264	0.0010593	0.00207979	0.0017447
7	800	0.162645	0.0344969	0.00121965	0.0023941	0.00200848
8	900	0.170566	0.0317139	0.00105713	0.00207473	0.00174062
9	1000	0.177795	0.0390158	0.00123379	0.00242111	0.00203128



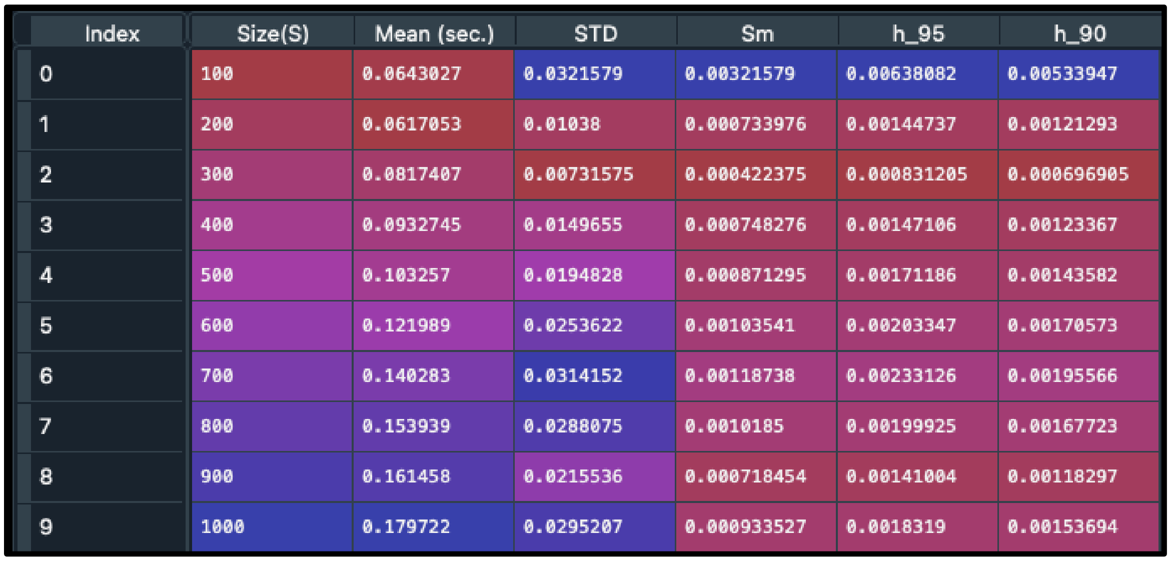
<u>Iteration size 500:</u>



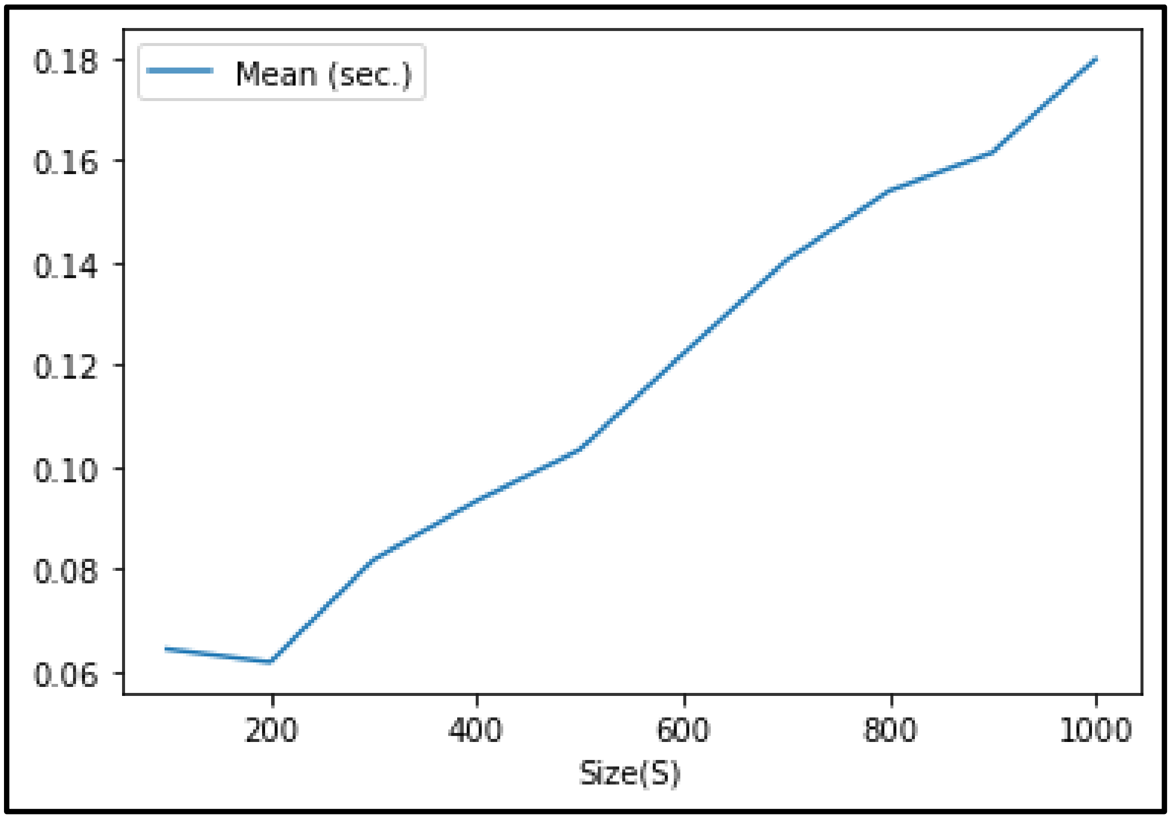
Index	Size(S)	Mean (sec.)	STD	Sm	h_95	h_90
0	100	0.05671	0.0104894	0.00104894	0.00208133	0.00174165
1	200	0.0679516	0.0128532	0.000908859	0.00179223	0.00150193
2	300	0.08269	0.0137112	0.000791616	0.00155784	0.00130614
3	400	0.0905784	0.0140096	0.000700481	0.00137709	0.00115487
4	500	0.102024	0.0171093	0.000765151	0.00150332	0.0012609
5	600	0.120422	0.0165145	0.000674201	0.00132409	0.00111068
6	700	0.132417	0.01847	0.000698099	0.00137062	0.00114979
7	800	0.138382	0.0180401	0.000637813	0.00125199	0.00105033
8	900	0.152427	0.0209808	0.000699361	0.00137257	0.00115153
9	1000	0.167261	0.0244348	0.000772696	0.00151629	0.00127215



<u>Iteration size 1000</u>



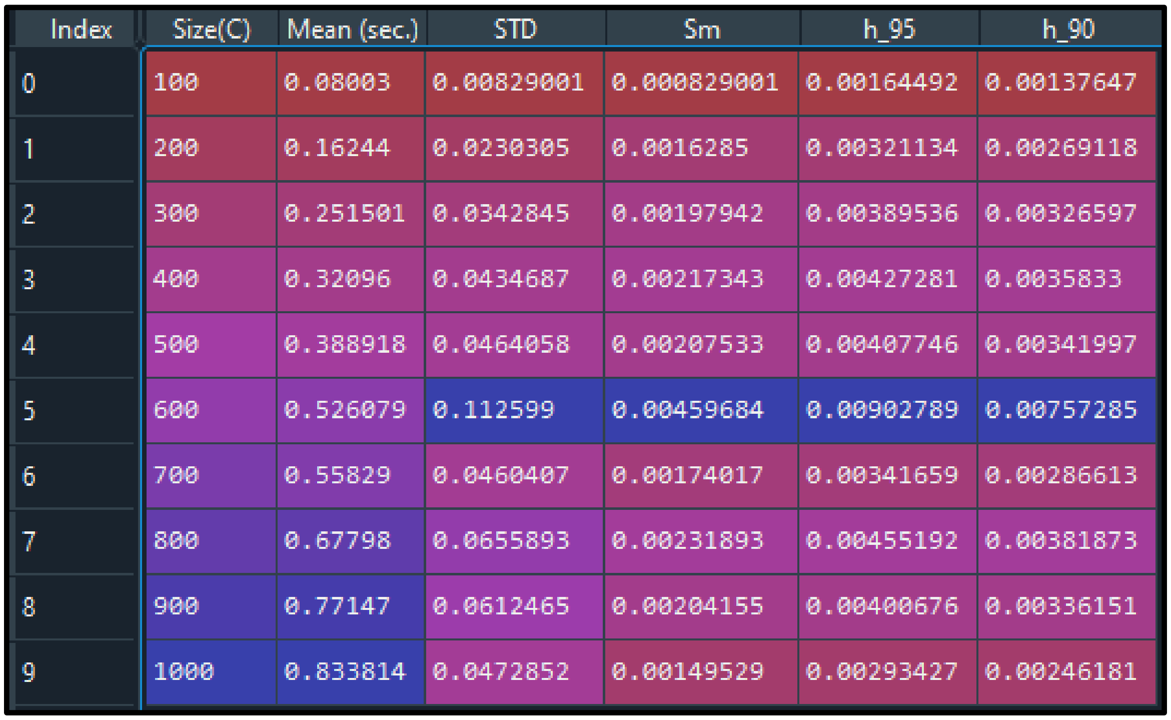
Index	Size(S)	Mean (sec.)	STD	Sm	h_95	h_90
0	100	0.0643027	0.0321579	0.00321579	0.00638082	0.00533947
1	200	0.0617053	0.01038	0.000733976	0.00144737	0.00121293
2	300	0.0817407	0.00731575	0.000422375	0.000831205	0.000696905
3	400	0.0932745	0.0149655	0.000748276	0.00147106	0.00123367
4	500	0.103257	0.0194828	0.000871295	0.00171186	0.00143582
5	600	0.121989	0.0253622	0.00103541	0.00203347	0.00170573
6	700	0.140283	0.0314152	0.00118738	0.00233126	0.00195566
7	800	0.153939	0.0288075	0.0010185	0.00199925	0.00167723
8	900	0.161458	0.0215536	0.000718454	0.00141004	0.00118297
9	1000	0.179722	0.0295207	0.000933527	0.0018319	0.00153694



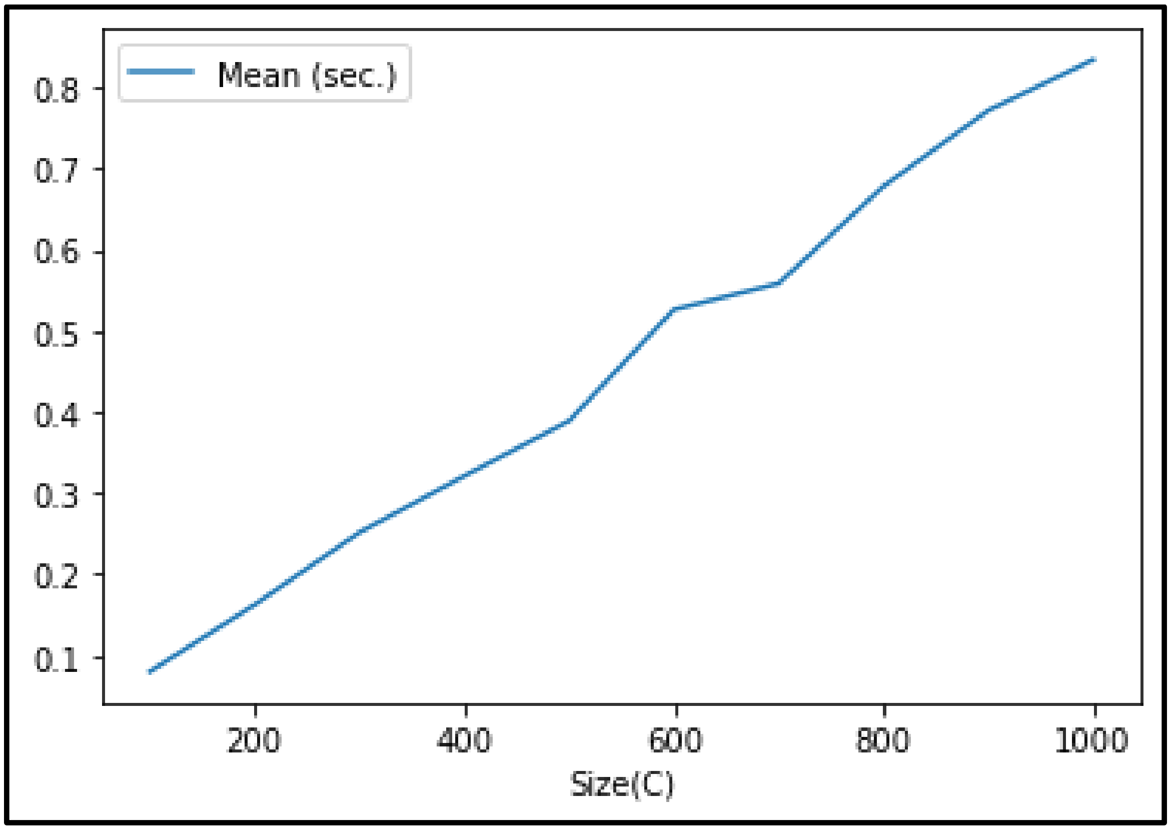
D & S are constant (50,1000)

C is increasing (100 -1000)

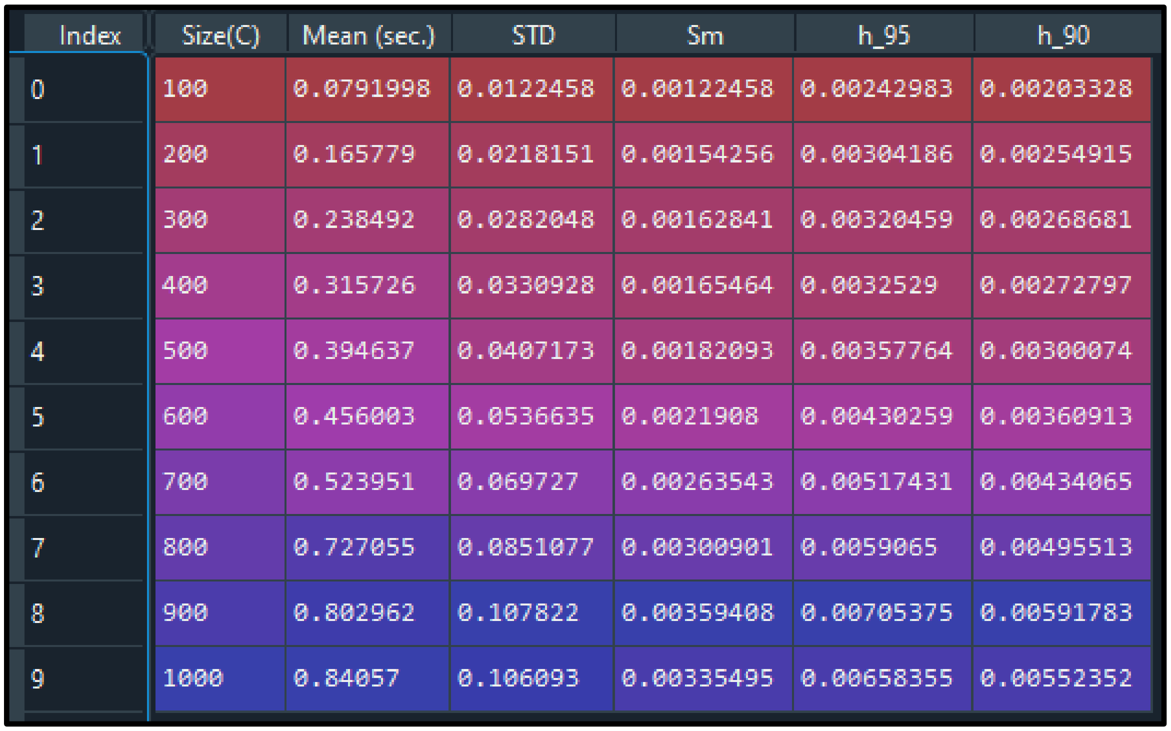
Iteration size 100:



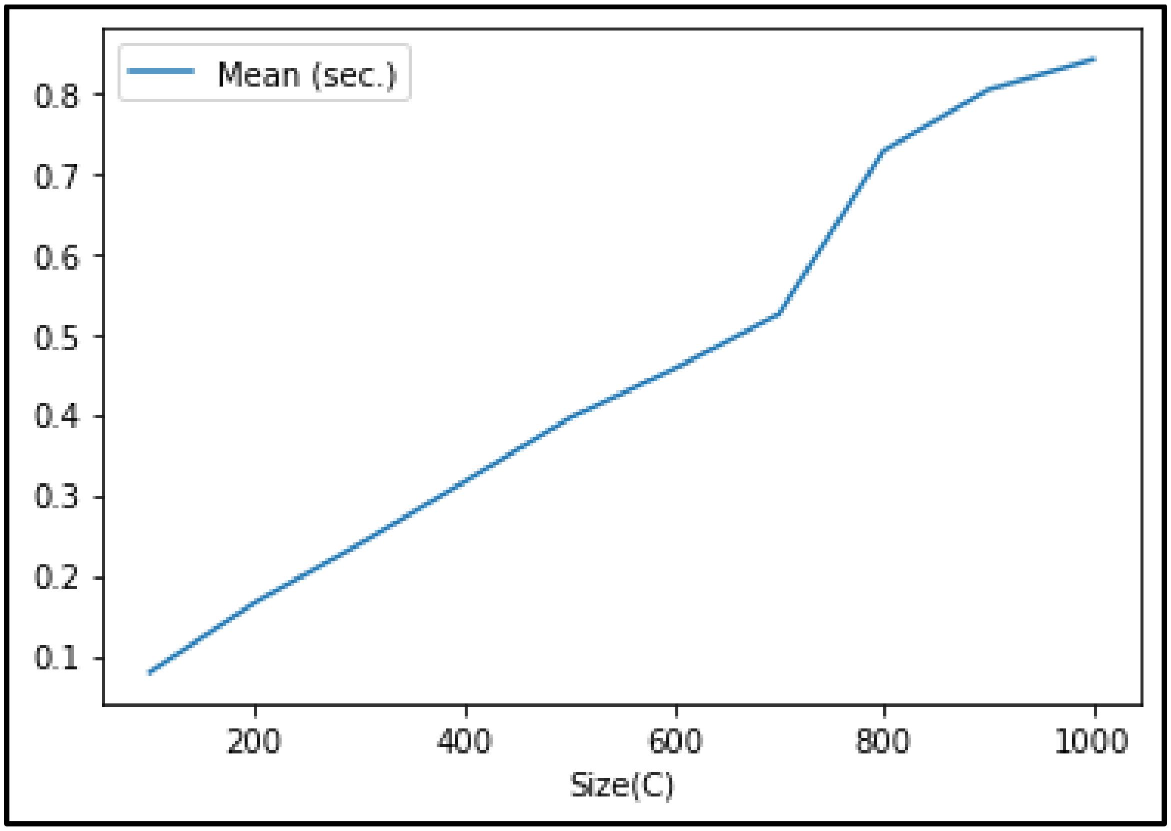
Index	Size(C)	Mean (sec.)	STD	Sm	h_95	h_90
0	100	0.08003	0.00829001	0.000829001	0.00164492	0.00137647
1	200	0.16244	0.0230305	0.0016285	0.00321134	0.00269118
2	300	0.251501	0.0342845	0.00197942	0.00389536	0.00326597
3	400	0.32096	0.0434687	0.00217343	0.00427281	0.0035833
4	500	0.388918	0.0464058	0.00207533	0.00407746	0.00341997
5	600	0.526079	0.112599	0.00459684	0.00902789	0.00757285
6	700	0.55829	0.0460407	0.00174017	0.00341659	0.00286613
7	800	0.67798	0.0655893	0.00231893	0.00455192	0.00381873
8	900	0.77147	0.0612465	0.00204155	0.00400676	0.00336151
9	1000	0.833814	0.0472852	0.00149529	0.00293427	0.00246181



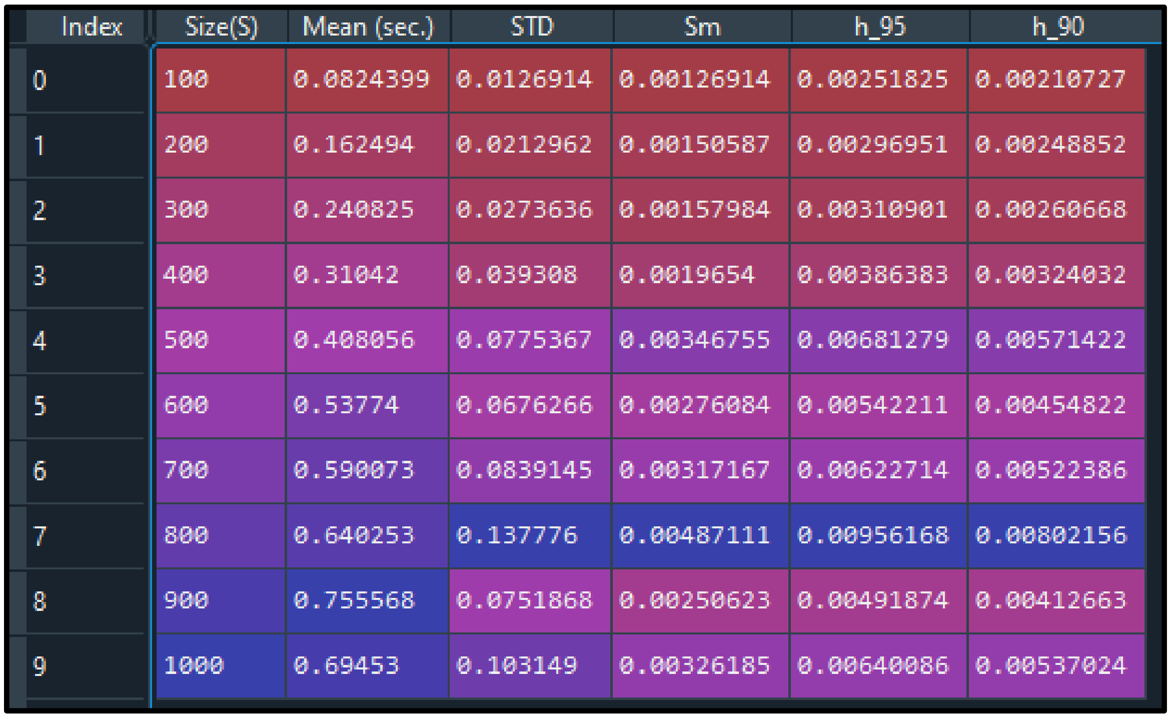
<u>Iteration size 500:</u>



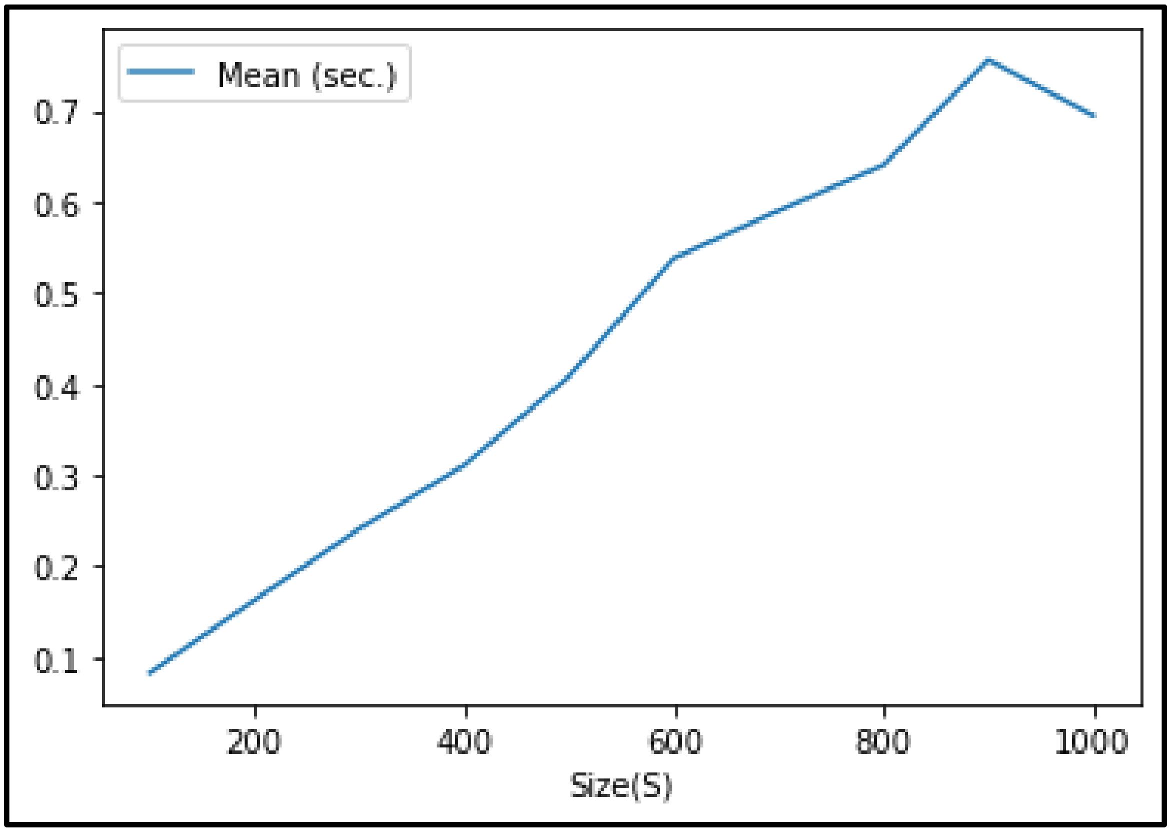
Index	Size(C)	Mean (sec.)	STD	Sm	h_95	h_90
0	100	0.0791998	0.0122458	0.00122458	0.00242983	0.00203328
1	200	0.165779	0.0218151	0.00154256	0.00304186	0.00254915
2	300	0.238492	0.0282048	0.00162841	0.00320459	0.00268681
3	400	0.315726	0.0330928	0.00165464	0.0032529	0.00272797
4	500	0.394637	0.0407173	0.00182093	0.00357764	0.00300074
5	600	0.456003	0.0536635	0.0021908	0.00430259	0.00360913
6	700	0.523951	0.069727	0.00263543	0.00517431	0.00434065
7	800	0.727055	0.0851077	0.00300901	0.0059065	0.00495513
8	900	0.802962	0.107822	0.00359408	0.00705375	0.00591783
9	1000	0.84057	0.106093	0.00335495	0.00658355	0.00552352



<u>Iteration size 1000:</u>



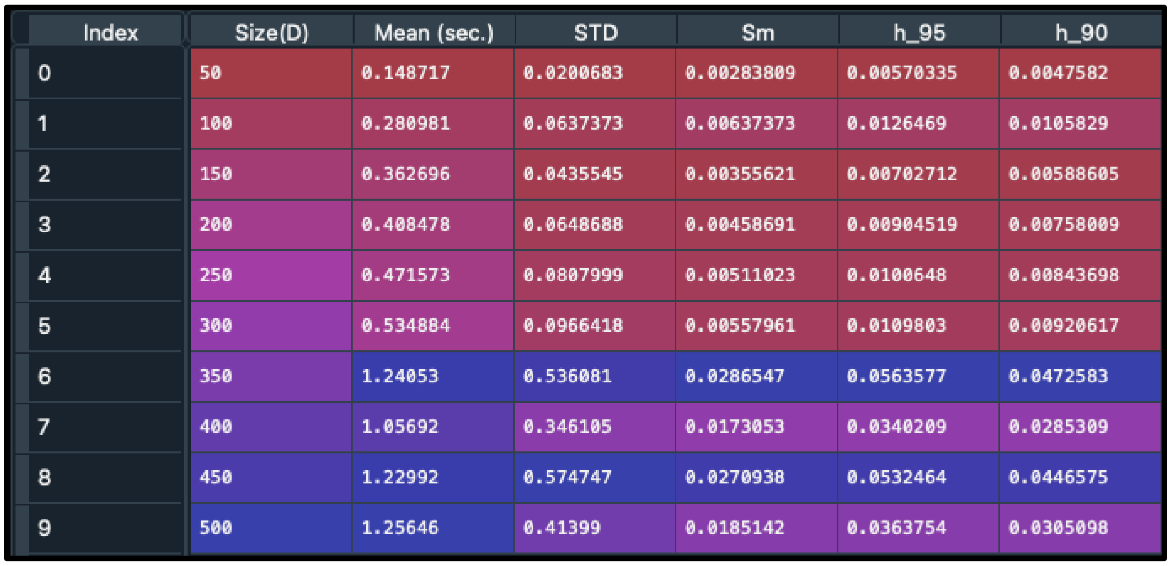
Index	Size(S)	Mean (sec.)	STD	Sm	h_95	h_90
0	100	0.0824399	0.0126914	0.00126914	0.00251825	0.00210727
1	200	0.162494	0.0212962	0.00150587	0.00296951	0.00248852
2	300	0.240825	0.0273636	0.00157984	0.00310901	0.00260668
3	400	0.31042	0.039308	0.0019654	0.00386383	0.00324032
4	500	0.408056	0.0775367	0.00346755	0.00681279	0.00571422
5	600	0.53774	0.0676266	0.00276084	0.00542211	0.00454822
6	700	0.590073	0.0839145	0.00317167	0.00622714	0.00522386
7	800	0.640253	0.137776	0.00487111	0.00956168	0.00802156
8	900	0.755568	0.0751868	0.00250623	0.00491874	0.00412663
9	1000	0.69453	0.103149	0.00326185	0.00640086	0.00537024



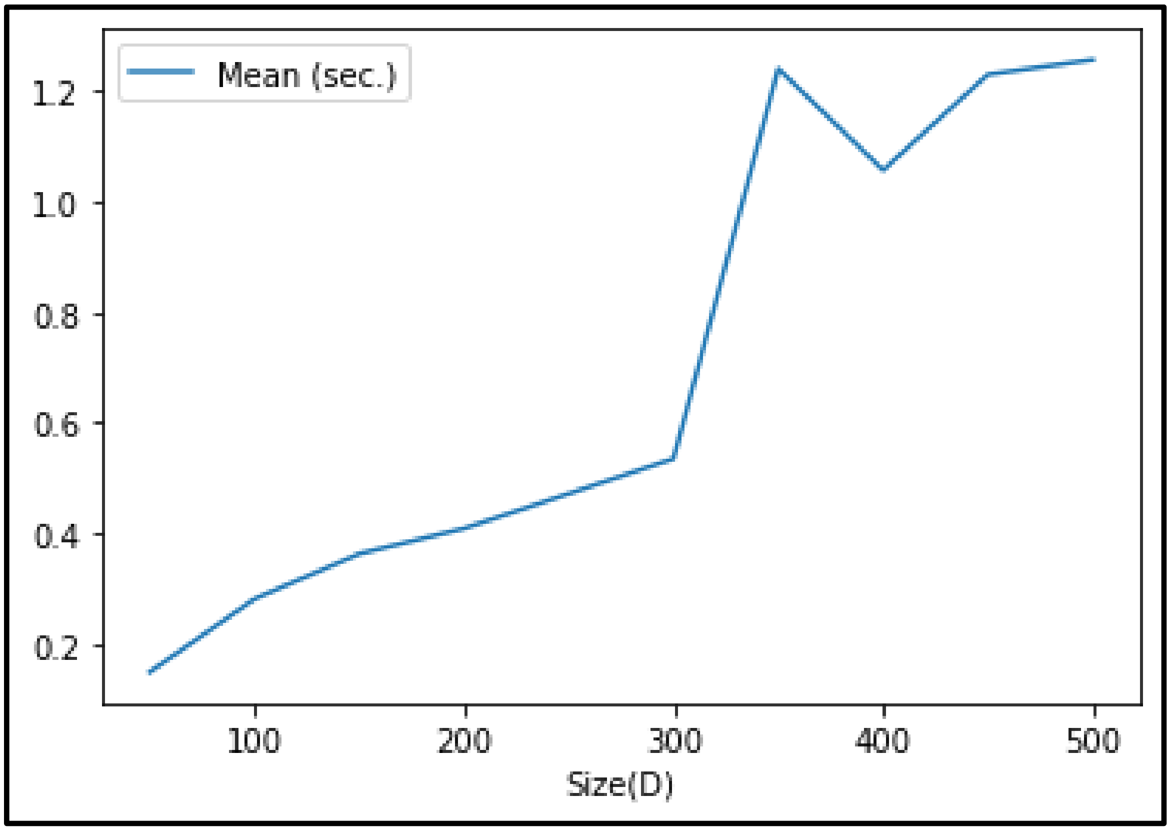
S & C is constant (1000,1000)

D is increasing (10-500)

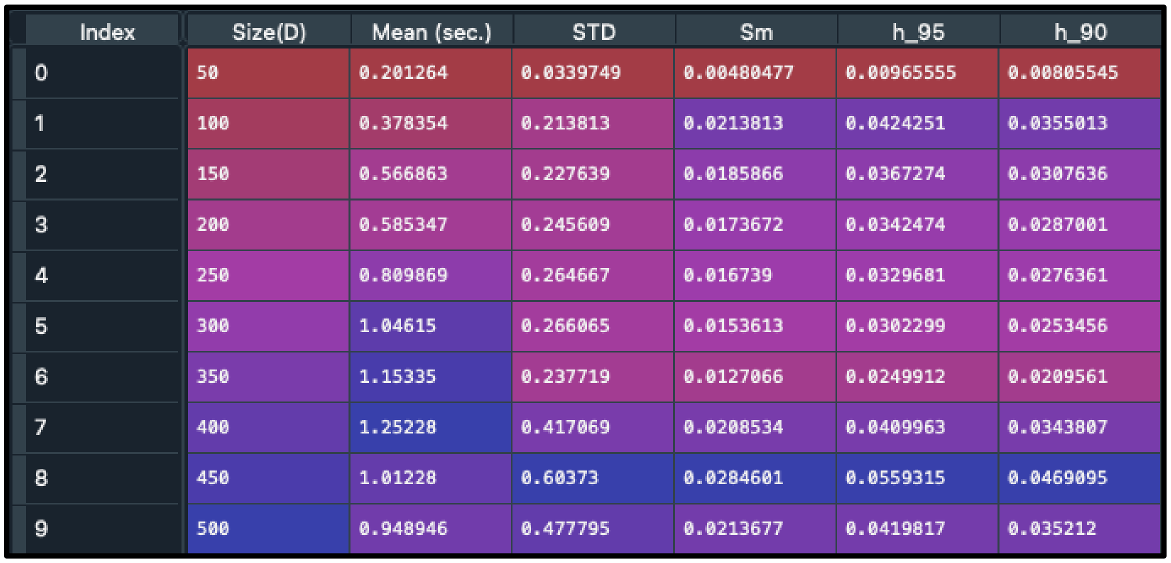
Iteration size 100,



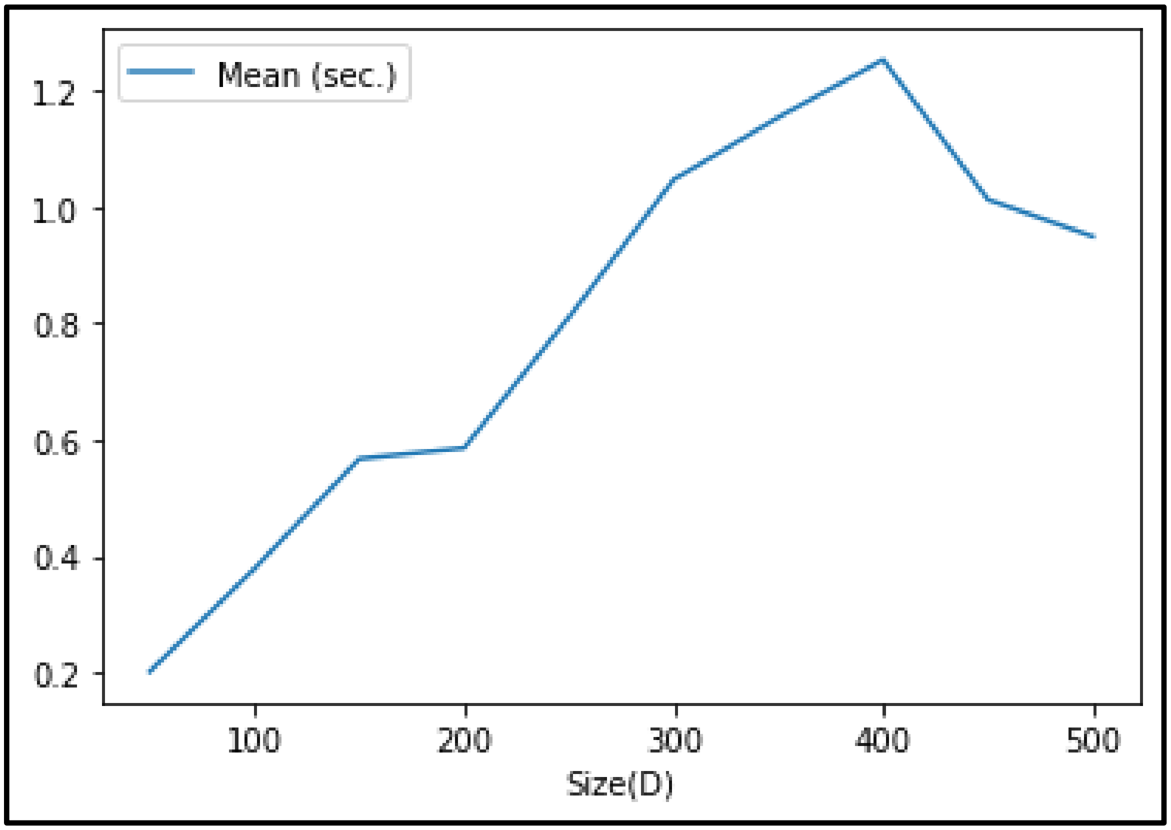
Index	Size(D)	Mean (sec.)	STD	Sm	h_95	h_90
0	50	0.148717	0.0200683	0.00283809	0.00570335	0.0047582
1	100	0.280981	0.0637373	0.00637373	0.0126469	0.0105829
2	150	0.362696	0.0435545	0.00355621	0.00702712	0.00588605
3	200	0.408478	0.0648688	0.00458691	0.00904519	0.00758009
4	250	0.471573	0.0807999	0.00511023	0.0100648	0.00843698
5	300	0.534884	0.0966418	0.00557961	0.0109803	0.00920617
6	350	1.24053	0.536081	0.0286547	0.0563577	0.0472583
7	400	1.05692	0.346105	0.0173053	0.0340209	0.0285309
8	450	1.22992	0.574747	0.0270938	0.0532464	0.0446575
9	500	1.25646	0.41399	0.0185142	0.0363754	0.0305098



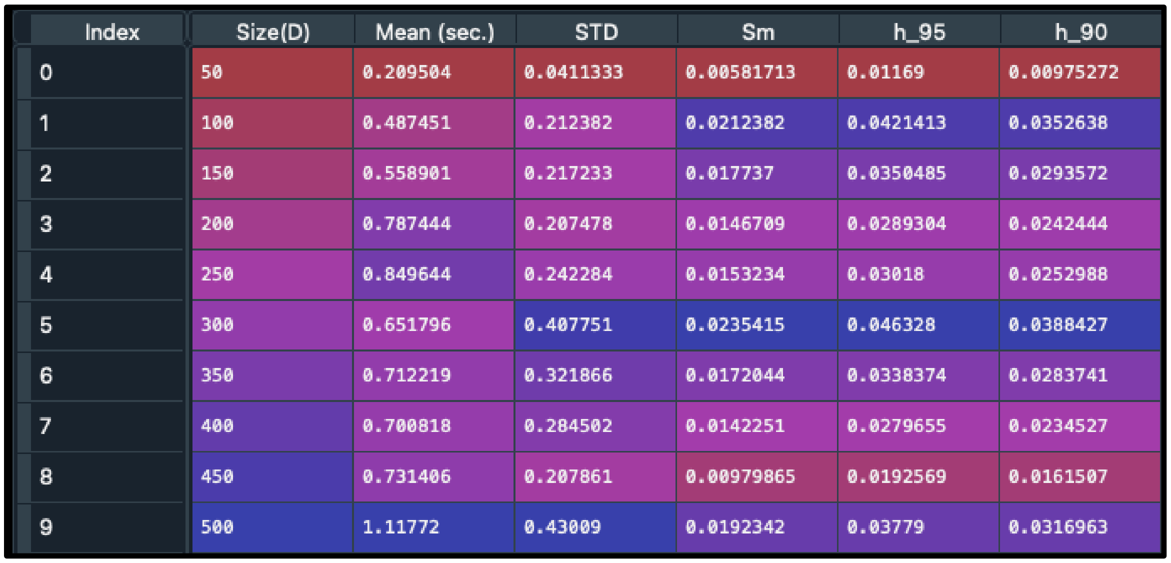
Iteration size 500,



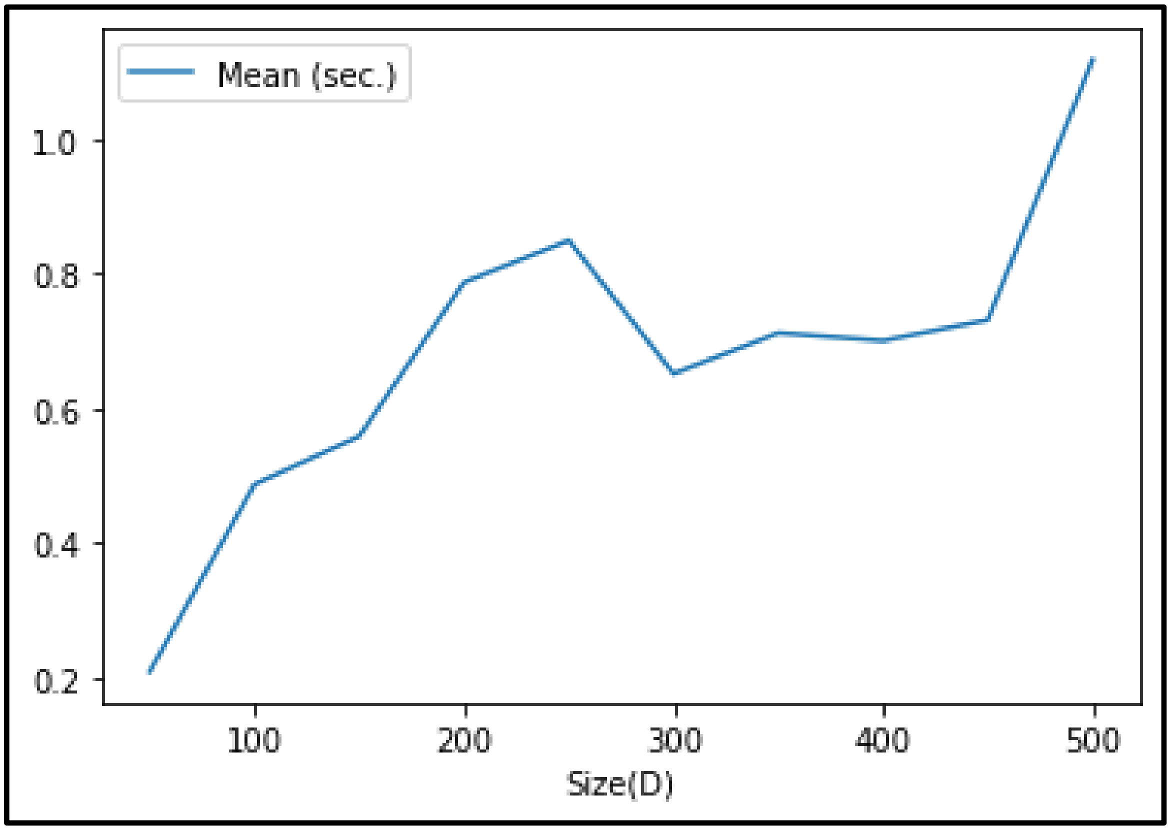
Index	Size(D)	Mean (sec.)	STD	Sm	h_95	h_90
0	50	0.201264	0.0339749	0.00480477	0.00965555	0.00805545
1	100	0.378354	0.213813	0.0213813	0.0424251	0.0355013
2	150	0.566863	0.227639	0.0185866	0.0367274	0.0307636
3	200	0.585347	0.245609	0.0173672	0.0342474	0.0287001
4	250	0.809869	0.264667	0.016739	0.0329681	0.0276361
5	300	1.04615	0.266065	0.0153613	0.0302299	0.0253456
6	350	1.15335	0.237719	0.0127066	0.0249912	0.0209561
7	400	1.25228	0.417069	0.0208534	0.0409963	0.0343807
8	450	1.01228	0.60373	0.0284601	0.0559315	0.0469095
9	500	0.948946	0.477795	0.0213677	0.0419817	0.035212



Iteration size 1000



Index	Size(D)	Mean (sec.)	STD	Sm	h_95	h_90
0	50	0.209504	0.0411333	0.00581713	0.01169	0.00975272
1	100	0.487451	0.212382	0.0212382	0.0421413	0.0352638
2	150	0.558901	0.217233	0.017737	0.0350485	0.0293572
3	200	0.787444	0.207478	0.0146709	0.0289304	0.0242444
4	250	0.849644	0.242284	0.0153234	0.03018	0.0252988
5	300	0.651796	0.407751	0.0235415	0.046328	0.0388427
6	350	0.712219	0.321866	0.0172044	0.0338374	0.0283741
7	400	0.700818	0.284502	0.0142251	0.0279655	0.0234527
8	450	0.731406	0.207861	0.00979865	0.0192569	0.0161507
9	500	1.11772	0.43009	0.0192342	0.03779	0.0316963



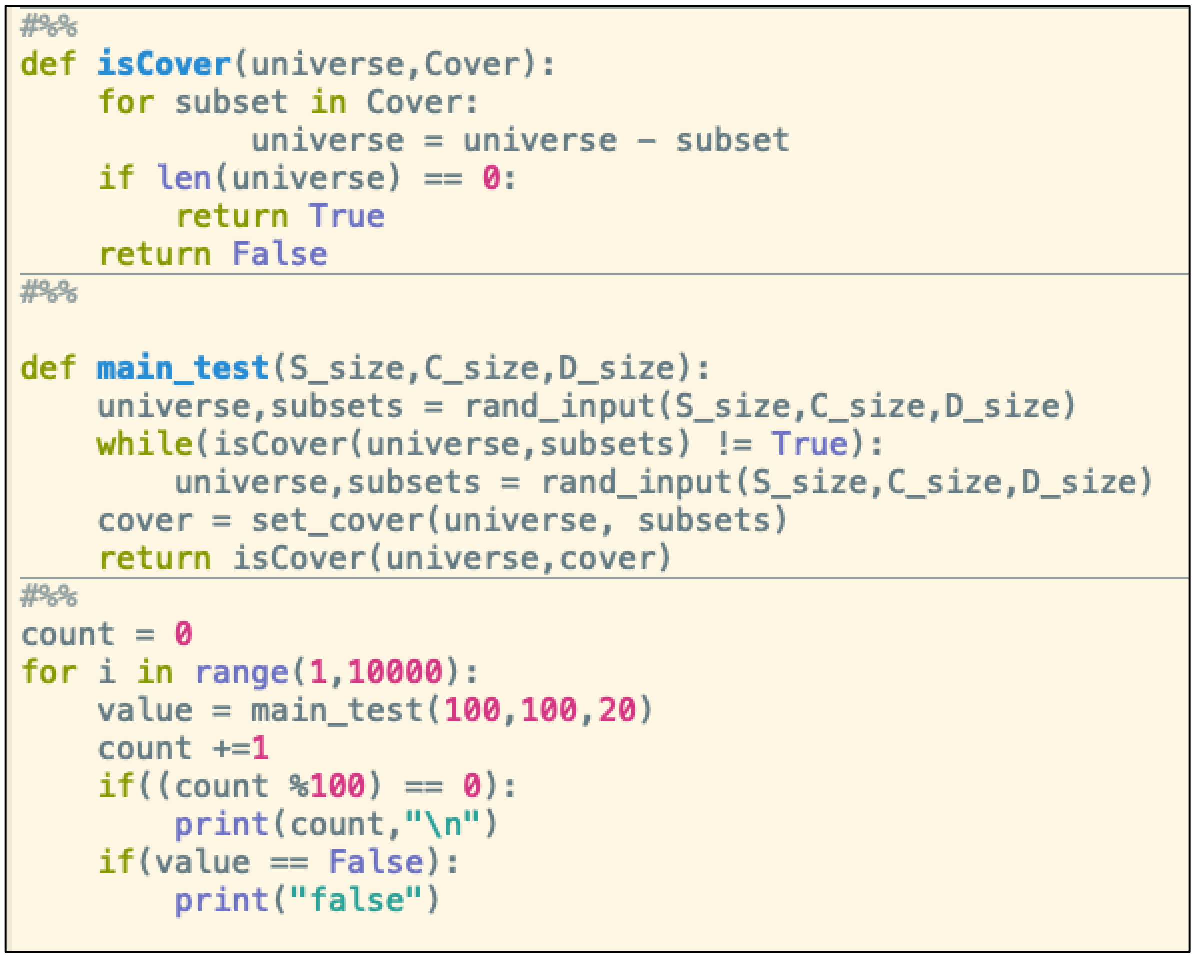
Correctness

Heuristic algorithm the greedy minSetCover does not always give minimum cover for given Set S and Cover set C. However, It always provide a cover for set S if there exists a cover among the subsets of C.

Very simple brute-force testing is applied as given below. isCover function tests whether the given set “Cover” forms a cover for “universe”

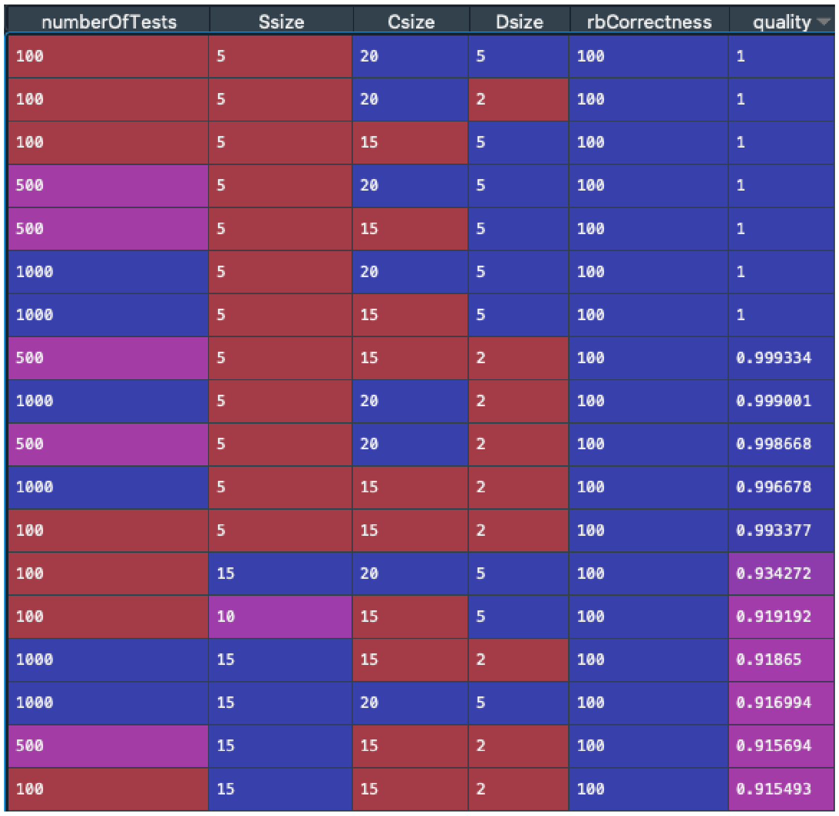
To make sure that testing is done with correct inputs; in main_test, we randomly create inputs until C forms a cover for S. Afterwards the main algorithm, “set_cover” is run. Finally, the returned output of “set_cover” is tested with universe. Main_test returns True if the solution is correct.

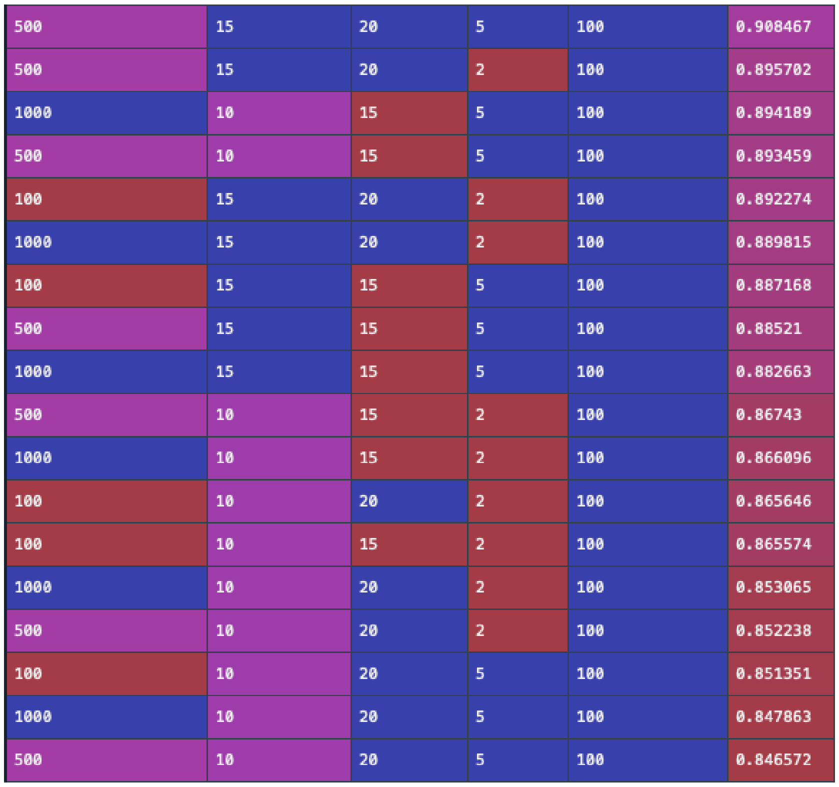
The inputs of the main test are ranges for uniform random variables. For example, giving 100 for S, we set size of S to 100. However, content of S is determined randomly at each iteration. We tested the algorithm 10000 times and got the correct result.



```
#%%
def isCover(universe,Cover):
    for subset in Cover:
            universe = universe - subset
    if len(universe) == 0:
        return True
    return False
#%%

def main_test(S_size,C_size,D_size):
    universe,subsets = rand_input(S_size,C_size,D_size)
    while(isCover(universe,subsets) != True):
        universe,subsets = rand_input(S_size,C_size,D_size)
    cover = set_cover(universe, subsets)
    return isCover(universe,cover)
#%%
count = 0
for i in range(1,10000):
    value = main_test(100,100,20)
    count +=1
    if((count %100) == 0):
        print(count,"\n")
    if(value == False):
        print("false")
```

Ratio Bound





numberOfTests	Ssize	Csize	Dsize	rbCorrectness	quality
100	5	20	5	100	1
100	5	20	2	100	1
100	5	15	5	100	1
500	5	20	5	100	1
500	5	15	5	100	1
1000	5	20	5	100	1
1000	5	15	5	100	1
500	5	15	2	100	0.999334
1000	5	20	2	100	0.999001
500	5	20	2	100	0.998668
1000	5	15	2	100	0.996678
100	5	15	2	100	0.993377
100	15	20	5	100	0.934272
100	10	15	5	100	0.919192
1000	15	15	2	100	0.91865
1000	15	20	5	100	0.916994
500	15	15	2	100	0.915694
100	15	15	2	100	0.915493
500	15	20	5	100	0.908467
500	15	20	2	100	0.895702
1000	10	15	5	100	0.894189
500	10	15	5	100	0.893459
100	15	20	2	100	0.892274
1000	15	20	2	100	0.889815
100	15	15	5	100	0.887168
500	15	15	5	100	0.88521
1000	15	15	5	100	0.882663
500	10	15	2	100	0.86743
1000	10	15	2	100	0.866096
100	10	20	2	100	0.865646
100	10	15	2	100	0.865574
1000	10	20	2	100	0.853065
500	10	20	2	100	0.852238
100	10	20	5	100	0.851351
1000	10	20	5	100	0.847863
500	10	20	5	100	0.846572

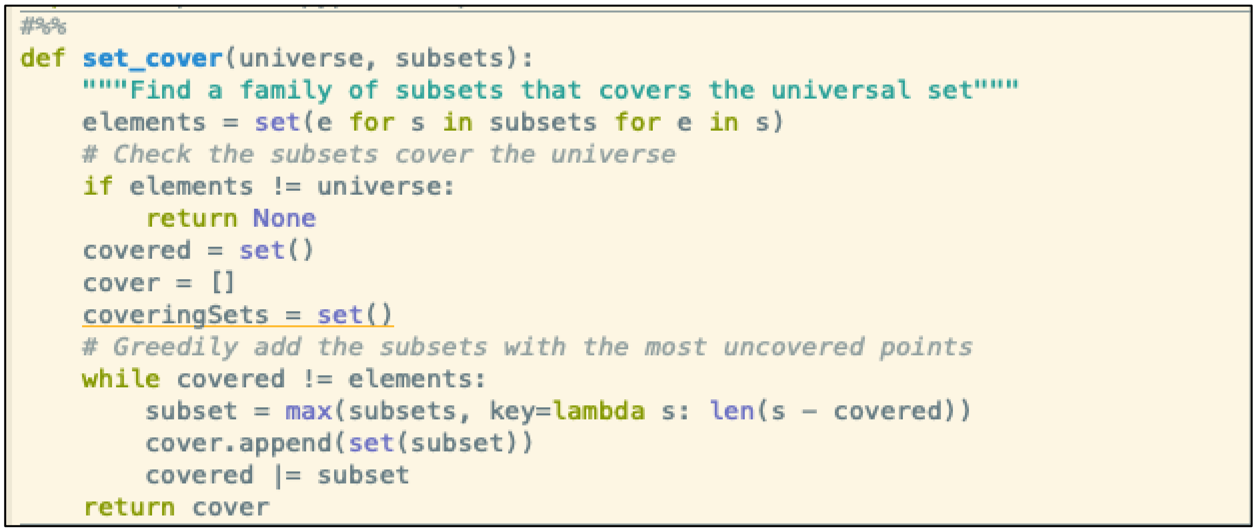
Above, you can see how quality and ratio bound correctness of the greedy algorithm change with different cardinality of S, C, and D with different test sizes. As it can be seen, ratio bound correctness of the algorithm 100% which means that for a set S if there is a min-cover the greedy algorithm finds a cover but, not necessarily the min-cover. According to test results, the effect of the number of tests done for the same instances is so small. Whenever the S sizes are so small or S sizes close to D sizes the quality approaches 1. For medium size S values if size of C is much bigger than the size of S the quality approaches to 0.

TESTING THE CORRECTNESS

White Box Testing

In white box testing method we can reach the instructions of the algorithm. We will try to cover all decisions and paths of the implementation given below. There is only one decision and that is the if statement. We will give test cases such that pass through the if statement and skip the if statement.

These test cases given below are tested in debug mode. Each statement is covered.



```
#%%
def set_cover(universe, subsets):
    """Find a family of subsets that covers the universal set"""
    elements = set(e for s in subsets for e in s)
    # Check the subsets cover the universe
    if elements != universe:
        return None
    covered = set()
    cover = []
    coveringSets = set()
    # Greedily add the subsets with the most uncovered points
    while covered != elements:
        subset = max(subsets, key=lambda s: len(s - covered))
        cover.append(set(subset))
        covered |= subset
    return cover
```

Test Case1

Universe = {1,2,3,4,5,6,7,8,9,10}

Subsets = [{}]

In this case since the implementation didn't detect a cover in the end, It returned None.

Test Case2

Universe = {1,2,3,4,5,6,7,8,9,10}

Subsets = [{1,2,3},{4,5,6},{7,8,9},{10}]

In this case implementation detected a coverage and skipped the if statement. All other statements are covered after the if statement.

It returned: [{1,2,3},{4,5,6},{7,8,9},{10}]

Black Box Testing

As given under Experimental Analysis, we tested the algorithm for 10000 random inputs set. The resulting sets of the algorithm formed a cover for each case. However, It is not guaranteed to find a minimum cover, therefore It does not show the correctness of the algorithm. Some additional test cases that we prepared are given below:

Test Case1

Universe = {1,2,3,4,5,6,7}

subsets= [{1,2,3,4,5,6,7}]

From the algorithm (not the implementation), we expect to have [{1,2,3,4,5,6,7}] as output.

Implementation output: [{1,2,3,4,5,6,7}]

Test Case 2

Universe = {1,2,3,4,5,6,7}

Subsets = [{2,3,4,5,6},{1,2,3,4},{5,6,7}]

From the algorithm (not the implementation), we expect to have [{2,3,4,5,6},{1,2,3,4},{5,6,7}] as output.

Implementation output: [{2, 3, 4, 5, 6}, {1, 2, 3, 4}, {5, 6, 7}]

This is not a correct solution. MinSetCover for this case is: [{1,2,3,4},{5,6,7}]. This happens because our greedy algorithm aims for max coverage at each step. And at the first step It includes {2,3,4,5,6}. However, that subset is not in the Min-Set-Cover.

Test Case 3

Universe = {1,2,3,4,5,6,7}

Subsets = [{1},{2},{3},{4},{5},{6},{7}]

From the algorithm (not the implementation), we expect to have [{1},{2},{3},{4},{5},{6},{7}] as output.

Implementation output: [{1}, {2}, {3}, {4}, {5}, {6}, {7}]

Test Case 4

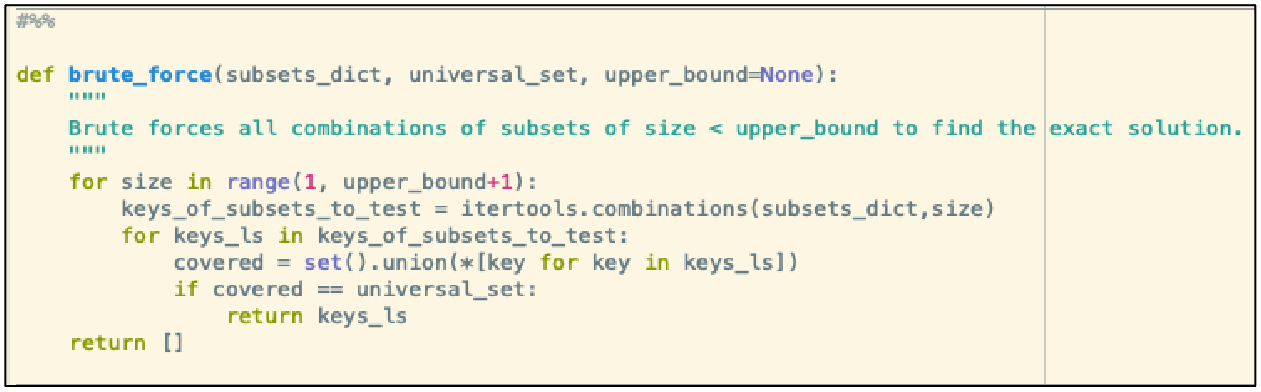
Universe = {1,2,3,4,5,6,7}

Subsets= [{1},{2},{3},{1,2,3},{4},{5},{6},{7},{4,5,6,7}]

From the algorithm (not the implementation), we expect to have [{1,2,3},{4,5,6,7}] as output.

Implementation output: [{4, 5, 6, 7}, {1, 2, 3}]

Brute force algorithm that is given below, upon a selected upper bound, calculates all combinations of subsets. That upper bound is the length of the solution of the greedy algorithm. Brute force just checks the combinations with cardinality <= #upper_bound.



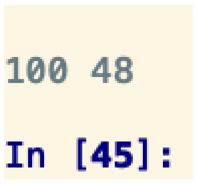
```python
#%%

def brute_force(subsets_dict, universal_set, upper_bound=None):
    """
    Brute forces all combinations of subsets of size < upper_bound to find the exact solution.
    """
    for size in range(1, upper_bound+1):
        keys_of_subsets_to_test = itertools.combinations(subsets_dict,size)
        for keys_ls in keys_of_subsets_to_test:
            covered = set().union(*[key for key in keys_ls])
            if covered == universal_set:
                return keys_ls
    return []
```

Our randomized test cases:

len(S) = 15, len(C)= 15, len(D)= 5

Repeat 100 times. Output:

Correctness is 48%

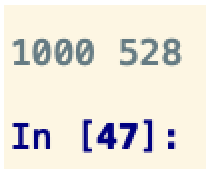


```
100 48

In [45]:
```

len(S) = 15, len(C)= 15, len(D)= 5

Repeat 1000 times. Output:

Correctness is 52.8%

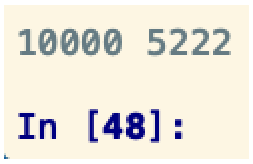


```
1000 528

In [47]:
```

len(S) = 15, len(C)= 15, len(D)= 5

Repeat 10000 times. Output:

Correctness is 52.2%



```
10000 5222

In [48]:
```

We tried some additional cases and observed that, correctness ratio increases when the size of d gets closer to the size of S. Which is straightforward to understand.

DISCUSSION

In conclusion, it is proven that minSetCover is a NP-complete problem which is reduced from vertex cover problem. This means that there is no existing algorithm such that solves the problem in polynomial time. However, there are different heuristic algorithms that can solve the problem in linear time without guaranteeing the optimal solution.

In this project a greedy algorithm is analyzed. Greedy algorithm always provides a set cover, but it is not guaranteed to find minSetCover. Furthermore, It is proven that heuristic algorithm finds the correct answer with ratio bound O(log n) where n is the input size. It is proven in the Algorithm Analysis section. Which indicates that, the cover of heuristic algorithm will be at most O(log n) times bigger than the optimal solution.

Running time of the heuristic algorithm was calculated as $O(S^2C\ d)$ in the algorithm analysis part. In the experimental analysis, running time of the algorithm is investigated by keeping one input varying and others constant. From these experiments, we observed that the implementation grows depending on C in linear time as calculated in algorithm analysis. The implementation grows depending on S in linear time as well. It was polynomial in algorithm analysis but not linear. Data that we obtained by varying D is very unstable. We conclude that the algorithm analysis that we have done should be reviewed in order to find tight bound.

Correctness of the heuristic algorithm is tested in two levels. First one whether It gives a cover for the Set or not. It is tested and gave correct result in this level with 10000 random instances. Second one is whether It gives a minSetCover or not. It is tested for several random inputs. However, It is not tested with large inputs because brute force algorithm takes too much time. The result of these test was around 50% correctness. We also observed that correctness increases when average size of D gets closer to S.

To conclude, a new metric “quality” is defined as 1 / Ratio where Ratio is size(Set-Cover) / size(Min-Set-Cover) to analyze heuristic algorithm in a better way. In order to calculate the the new metric “quality” of the heuristic algorithm, a brute force algorithm for Min-Set-Cover for small number of input size of set S is developed. The experiments show that the newly defined parameter “quality” of the heuristic algorithm is around 0.88 and when the input size (the cardinality of set S) increases, the “quality” tends to decrease.