**Set Cover Implementation**

**Pseudocode**

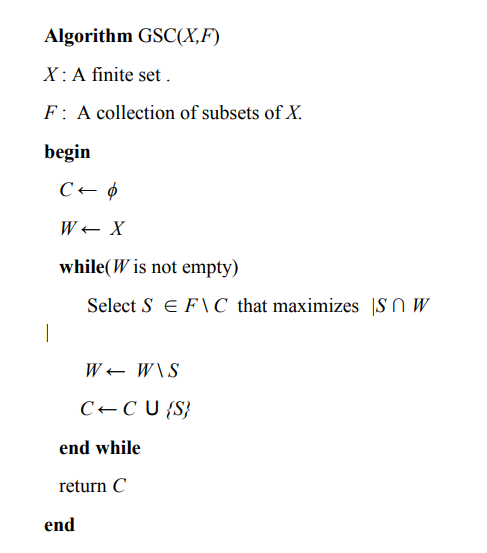
My program uses the following methods:

**set\_cover(universe, subsets):** This method is the greedy implementation of the set cover algorithm. Set cover is a type of optimization problem. An optimization problem is the problem of finding the best solution from all feasible solutions [3]. The main aim of a set cover algorithm is to minimize the number of sets that cover the universal set. For example, given a set of elements (also referred to as the universal set δ) and a collection of n sets S whose union is the universal set δ, the set cover problems attempts to find the smallest subset of S whose union equals the universal set δ. Set cover problem is an example of an NP-complete problem [3]. The logic behind a greedy implementation of a set cover problem is: the set that contains the largest number of uncovered elements is chosen. The greedy set cover algorithm does not find the optimal solution but finds an approximate solution. The approximation algorithm is judged by the approximation ratio ρ. The greedy algorithm G achieves an approximation ratio of ρ if |G(Σ)| ≤ ρ|O(Σ)|, for any input Σ(δ,S), where X = universal set, S = set of subsets. Ideally, we would like *ρ* to be as small as possible, say, a small constant. Unfortunately, the best that we can show for set cover is that ρ is a slowly growing function of m = | δ |, and in particular *ρ* = *log (m)*.[5] | δ | stands for the cardinality of the universal set δ.

The time complexity of the set\_cover() implementation is judged by how small ρ is. Therefore, the time complexity of ρ is log(N), where N = number of elements in the universal set.

The code for set\_cover() was taken from [8] and was modified by me to output the indices of the subsets and not the subsets.

The pseudocode for greedy implementation of set cover approximation algorithm is given below [4]:



**Figure 1: The above figure shows a greedy method of implementing the set cover problem. The algorithm is used for unweighted sets X and F, as given above [4].**

**distance (p0, p1)**: This method implements the distance formula that is used to find the distance between two points p0 and p1. The distance formula is derived from Pythagorean theorem. The distance between two points p0 and p1 is given by the formula:

In this method, points p0(x0, y0) and p1(x1, y1) are tuples that contains two integer values, x and y, that represent the abscissa and ordinate of p0 and p1 respectively. The pseudocode for distance formula implementation is given below:

func **distance\_formula(p0, p1):**

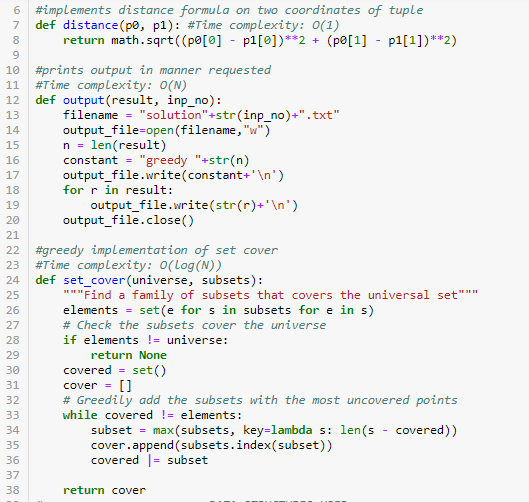
//program takes in two tuples p0 and p1

//p0 = (x0, y0); p1 = (x1,y1)

//p0 = (x0, y0); p1 = (x1,y1)

return math.sqrt((p0[0] - p1[0])\*\*2 + (p0[1] - p1[1])\*\*2)

**output(result):** This method returns the output of the program in the manner that has been requested by the project specifications i.e., in a text file named solutionxx where xx is the number that corresponds to the number of the instance file. For example, if instance file is ‘instance08.csv’ then the solution file will be ‘solution08.txt’



**Figure 1**: Screenshot taken from CS430 Final Project – Sarbani Bhattacharyya

**Code**

I have implemented the greedy method of solving the problem posed by the project. The code finds the least amount of points P that lie at atmost 1-unit distance from each of the centers in set C. The function that implements the solution to the problem is called imp() which takes in the name of the csv file that contains the input provided to us, in the form of files named ‘instance01’, ‘instance02’, ‘instance03’, ‘instance04’, ‘instance05’, ‘instance06’, ‘instance07’, ‘instance08’, and ‘instance09’.

Algorithm for function imp(usr):

1. The function imp(**usr**) takes in a csv file as an input.
2. It reads in the first line of the csv file( using **csvfile.readline()**) and extracts the number of centers(**c\_n**) from it.
3. Then, it iterates over **c\_n** number of lines till it reaches the line which specifies the number of points in set P(which is represented by **N** in the code).
4. The program then reads the entire file, and stores all the coordinate point values in a list called **data**. Since data contains coordinates of centers and points, it has to be divided into two parts. This is done using the values of **c\_n** and **N** as the specifiers for the ranges of the new list of coordinate points, representing centers in C and points in P respectively.

**setC:** dictionary that stores the coordinate points of the centers in C in this format; **setC** ={1: [0.25, 0.25], 2: [0, 0.25],….}

**setP:** dictionary stores the coordinate points of the centers in P in this format; **setP** = {1: [0.707, 0.707], 2: [0, 0.707],….}

Dictionaries were used so that for each set, I could keep track of the indices of each of the coordinate points in the respective sets C and P.

1. A nested for loop is used to find the distance between one point in **setC** and a point in **setP**. The function **distance()** is used to find the distance between these two points. If the distance is found to be less than 1, the value of P is added to a list.

**reference\_dict()**: Dictionary that stores values of points in P that are atmost 1 distance away for each center in C. The format is: {1: [1,3,4,5], 2: [6,7,8], 3: [2,3,4,5,6,7], 4: [1,2,3,4], …. }

1. Now, values of **reference\_dict()** are extracted into a list called **lst**. This is formatted to form the list called **subsets**. The set universe is generated using **set(range(1, N+1))** which generates a set containing elements from {1, 2, 3, . . . , N}
2. Now, the sets **universe** and **subsets** are fed into the function **set\_cover().** Set\_cover() uses a greedy algorithm to find the minimum number of **subsets** whose union forms the **universe** set. The set\_cover algorithm usually returns a set of sets(i.e., a set consisting of the minimum number of subsets required to cover the universal set)[4]. However, the method set\_cover() in the program returns the indices of the elements in **subsets** that satisfy the above condition. This is stored in a list called **res**.
3. The list **res** is inserted into the function **output()** which returns the output in the desired format.

**Code** : The below snippet of code shows the code for the function imp().

**Figure 2:** Screenshot taken from CS430 Final Project – Sarbani Bhattacharyya

**Complexity of methods used:**

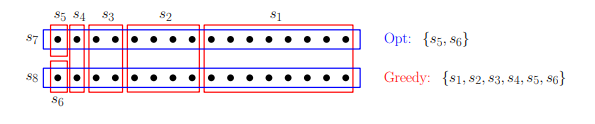
**imp():** The time complexities of the methods that comprise imp() is explained in the comments in the code above. O(log(N)) is the time complexity of the greedy set cover method. O( **N2**) is the time complexity of the doubly nested for loops that iterate over dictionaries **setC** and **setP**. O( log(**N**) ) is the time complexity of the set\_cover() method, which is a greedy approach to the problem. The time complexity of the method is O( **N2**) + O( log(**N**) ) = O(**N2**), since **N2** grows at a much faster rate than log(**N**) as the value of N increases. N = number of elements that the program has to iterate over.

**distance():** This is a O(**1**) complexity, that is, a function of constant time complexity.

**output():** This is a O(**N**) complexity, that is, a function of constant time complexity. This is because the function is printing out values from the list called **res.** So, if there are **N** elements in **res**, then the for loop in **output()** will run N times, thereby making the time complexity of **output()** to be O(**N**).

The above methods are the ones used to execute. Therefore, the overall time complexity of the program is:

Overall time complexity: O( **N2**) + O( log(**N**) ) + O(**N**) + O(**1**) = O( **N2**)

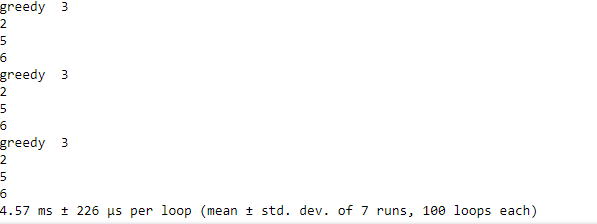
The **set\_cover() algorithm** will fail in the case given below. The figure given below shows a universal set of 32 elements. The optimal set cover consists of sets s7 and s8, each of size 16. Initially all three sets s1, s7, and s8 have 16 elements. the greedy algorithm will first select the set s1. We remove all the covered elements. Now s2, s7 and s8 all cover eight of the remaining elements. Again, if we choose poorly, s2 is chosen. The pattern repeats, choosing s3 (covering four of the remainder), s4 (covering two) and finally s5 and s6 (each covering one)[4]. From the pattern, we can generalize this to any number of elements that is a power of 2. While there is an optimal solution with 2 sets, the greedy algorithm will select roughly log(*m*) sets, where *m* = |δ|.

**Figure 1:** The above figure shows a case where the greedy set\_cover() algorithm fails to give the correct solution [3].

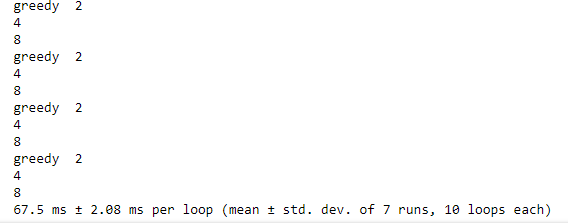
**Time Analysis**

I used %timeit to find the time taken to run for each instance.

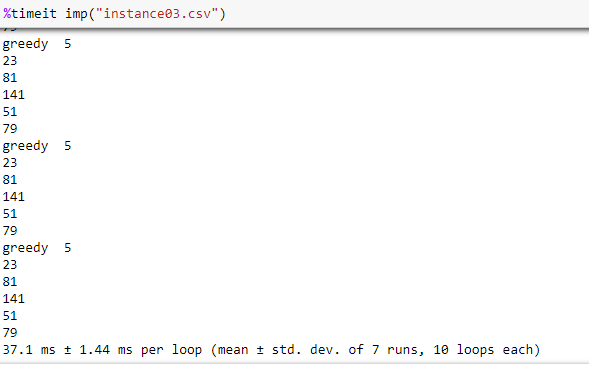
**≫ Instance01.csv**



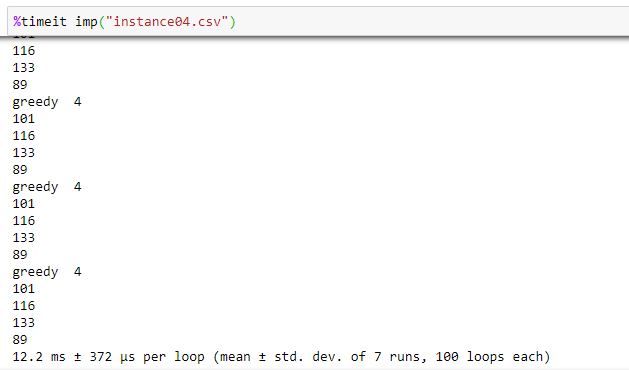
**≫ Instance02.csv**



**≫ Instance03.csv**



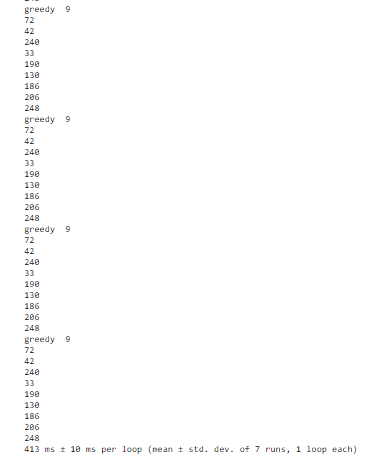
**≫ Instance04.csv**



**≫ Instance05.csv**



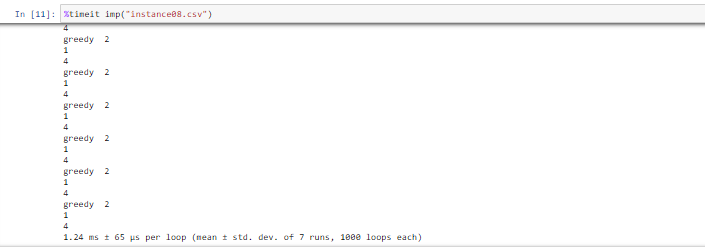
**≫ Instance06.csv**



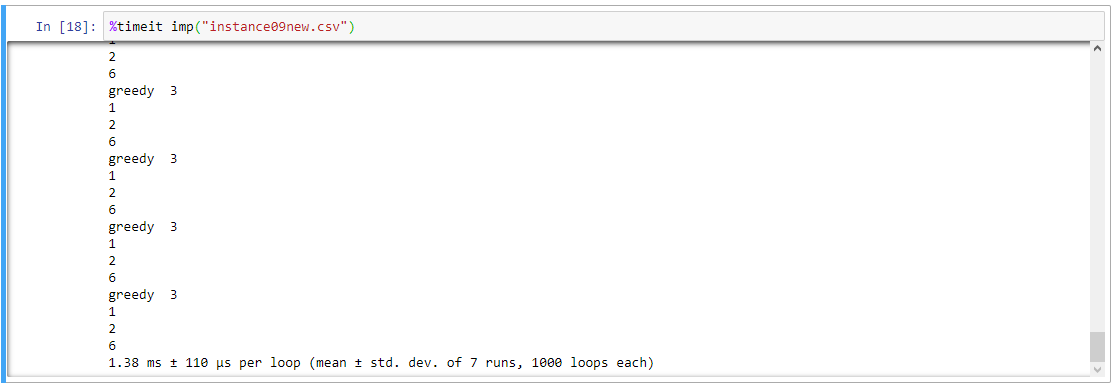
**≫ Instance07.csv**



**≫ Instance08.csv**



**≫ Instance09.csv**



From the use of %timeit on each of the given instances, as the size of set C(set of the centers) or set P(set of points) or both increases, the time increases by a factor of 2. This is consistent with our time complexity analysis of this program which was found to be O(N2) in the worst case. As the number of centers and points increase, the program must iterate over more number of elements and therefore, the time it takes to execute the program should also increase.

**References:**

1. <https://mcdtu.files.wordpress.com/2017/03/introduction-to-algorithms-3rd-edition-sep-2010.pdf> - Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein
2. <http://www.cs.iit.edu/~calinesc/forwarding.pdf> - Research paper provided by Professor Calinescu
3. <https://math.mit.edu/~goemans/18434S06/setcover-tamara.pdf>
4. <https://arxiv.org/ftp/arxiv/papers/1506/1506.04220.pdf>
5. <http://www.cs.umd.edu/class/fall2017/cmsc451-0101/Lects/lect09-set-cover.pdf>
6. <https://www.ics.uci.edu/~pattis/ICS-33/lectures/complexitypython.txt>
7. [http://www.orie.cornell.edu/~dpw/cornell.ps. p7-18](http://www.orie.cornell.edu/~dpw/cornell.ps.%20p7-18)
8. <http://www.martinbroadhurst.com/greedy-set-cover-in-python.html>