Project Document

Sabancı University - Fall 2021

CS301 Algorithms - Group 9 Project Report

Set Packing

Members:

- Taha Can Karaman - 26365
- Gönül Ayöz - 28488
- Emre Can Eşki - 26973
- Dora Akbulut - 26863

1. Problem Description

Formal Description

Input: C, a set of sets:

$$C = \{C_1, C_2, ..., C_m\}$$

Let D be a subset of C:

$$D = \{D_1, D_2, ..., D_k\}$$

How can a subset D of C can be found such that for all $1 \leq i < j \leq k$,

$$D_i \ \cap \ D_j = \emptyset$$

and the cardinality of D is maximized?

Intuitive Description

In this problem, the input is a finite set of sets, C, and the aim is to find a subset D of C where every element will be mutually disjoint sets where the cardinality of D is maximum. It is an optimization problem since there are 2^m subsets of C, and there will be many subsets of C where the elements can be pairwise disjoint.

Let us demonstrate an instance of the problem with an example:

$$S = \{\{1,2,3,4\},\{3,4\},\{5,7\},\{2,9,14\}\}$$

Now we list the sets R_i that consist of mutually disjoint subsets. Sets with $|R_i| = 1$ are trivial, therefore we will not list them in our example.

$$R_1 = \{\{1,2,3,4\},\{5,7\}\}$$

$$R_2 = \{\{3,4\},\{5,7\},\{2,9,14\}\}$$

Both sets consist of pairwise disjoint sets. $|R_2| > |R_1|$, therefore the correct solution to this instance of the problem would be R_2.

Here is a visualisation of the input in the example instance:

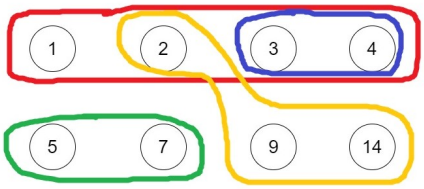

Fig. 1: The input of the problem.

The solution to this instance of the problem is in Fig.2:

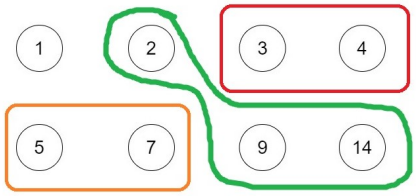

Fig. 2: The output of the problem.

Possible Applications

One real-life application of the set packing problem is the planning of airport staff and flight scheduling. To take off, an airplane needs a crew with a certain amount of pilots, copilots, flight attendants, and navigators. And there are some constraints in this possible crew based on their schedule, experiences, and personal conflicts. For example, a crew that consists of a new pilot and a new copilot, and a new navigator is not preferred.

Let's say an airline has to schedule planes for a specific time and it wants as many planes as possible to take off. For this, it needs to have all possible sets of crew members in a set and then look for the maximum amount of disjoint sets. In other words, it must use set packing.

Set packing has other applications in various different fields such as computer vision, communication networks, facility location, and incremental design process.

Hardness of the Problem

The Set Packing Problem that we are dealing with is known as the Maximum Set Packing Problem, and it is an NP-hard problem. To show that an algorithm is NP-hard, we first need to find a polynomial-time reduction from any known NP-hard problem to our original problem.

We chose the Maximum Independent Set/Clique problem as a known NP-hard problem. And we will show that it can be reduced to Maximum Set Packing in polynomial time.

Karp showed that Clique can be reduced to Set Packing (1972). The Clique problem is about finding the maximum number of pairwise adjacent nodes in a graph, and Independent Set is the reverse, finding independent sets of nodes in a graph. If G' is the complementary graph of G, the solution to Clique for graph G is the same as the solution of Independent Set for G'. We think that the reduction from Independent Set to Set Packing is easier to understand, therefore we will show this reduction. We will demonstrate Karp's proof.

We will explain the proof we found with an example. In Fig. 3, there is a graph G with a set of vertices V and edges E. We can map each vertex v_i as a subset of s_i in our family of sets S. The elements of s_i are the edges that are incident with v_i. Therefore the mapped subset would be as follows:

s_1 = { A, D }
s_2 = { D, B, F }
s_3 = { C, F }
s_4 = { A, B }
s_5 = { C, E }
s_6 = { E }

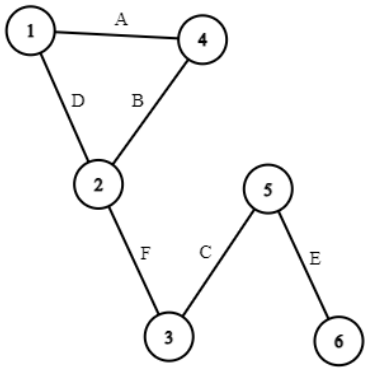

Fig. 3:
Graph G with set of vertices V and edges E

One correct solution to the Set Packing for $S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$ is $\{s_1, s_3, s_6\}$. And $\{v_1, v_3, v_6\}$ is also a correct solution for Independent Set. Therefore we have shown that this instance of Independent Set can be reduced to Set Packing.

In order to reduce Independent Set to Set Packing in polynomial time, we just need to go through each list once and map the graph to sets.

Pseudo-code for reducing Independent Set to Set Packing

```
// initialize the set of subsets S
S = {}
iterate through vertices in V:
  create empty subset s_i = {} that corresponds to v_i
  add s_i to S
iterate through edges E:
  for vertices (v_j, v_k) that edge e_i is connected to:
    add e_i to s_j and s_k
```

2. Algorithm Description

2-a) Brute Force Algorithm

Description

The algorithm was developed by one of our team members, and it is a brute-force algorithm that compares every possible subset of C with each other to find the largest cardinality subset with mutually disjoint elements.

The algorithm works in the following way:

1. It creates every possible subset of C.
2. It keeps track of the subset which has pairwise disjoint elements and has the largest cardinality. These are initially empty and zero, respectively. Let's call it *currentLargest*.
3. Iterates through each subset and checks if all of its elements are mutually disjoint or not.
4. If they are, the cardinality of the current subset is compared with *currentLargest*.
5. The one with larger size becomes the new *currentLargest*.
6. After checking all of the subsets, we have the subset which satisfies the desired conditions.

Pseudo Code

```
bestSize = 0 // initialize the size of the current largest cardinality subset as 0
iterate through all possible subsets S of C
  if all elements of S are pairwise disjoint
    currentSubset = S
  if size(S) > bestSize
    bestSize = size(S)
```

```
    currentLargestCardinalitySubset = S
return S
```

2-b) Greedy Algorithm

Explanation of our algorithm:

$C = \{C_1, C_2, C_3, ..., C_n\}$ is the input to our algorithm, which is a family of subsets. Initially, we sort the sets of C with respect to their cardinalities in increasing order and we initialize the number of disjoint subsets (which will be the output of our algorithm) as 0. If there is an empty subset, we directly increase the number of disjoint sets.
Then, we create an empty list B which will contain the elements of the subsets with disjoint elements in sorted order.
We iterate over the elements of C. For each subset C_i, we check if $C_i \cap B = \emptyset$. If this is the case, we add all elements of C_i to B, and sort B again.
After the iteration is complete, we have the cardinality of the maximum cardinality subset that has mutually disjoint subsets.

The pseudo-code of our algorithm:

```
C = [C1, C2, ...]
Number_of_subsets = 0
//initialize an empty list
B = []
//Create a sorted C
Sort the elements of C with respect to their cardinality
iterate through all elements of C:
//Take the current element of C as Ci.
//For every element of Ci, check if there is an element of B with quick-select.
  if the current element of C is disjoint with B
  or the current element of C is empty:
     Append every  element of the current element of C into B
     Sort B
     Number_of_subsets ++
Return Number_of_subsets
```

Design technique of our algorithm:
Our algorithm is designed in the approximation technique. The reason for this is that our algorithm might not produce a correct solution and the error will be limited to a certain boundary. Therefore a ratio bound for the algorithm can be given.

Ratio bound of our algorithm:
Let's call the optimal solution k^* and the solution our algorithm produces k.
For each set C_i that was mutually disjoint with B, at the worst case, there can be $|C_i|$ sets that were missed because these sets could be disjoint with B and each other as well.

The following is an example situation. Let's say B currently has only one element and it is 0. The element that we currently inspect in this round of iteration is C_1 since it has 4 elements and the other sets have 5. And we will add the elements of C_1 to B. This way, we will have missed $|C_i| - 1$ sets that could be added to the result.

The rows of Table 1 represent the elements of the sets in an instance of the iteration of the algorithm.

B	*0*				
Subset 1	*1*	*2*	*3*	*4*	
Subset 2	*2*	*5*	*6*	*7*	*8*
Subset 3	*3*	*9*	*10*	*11*	*12*
Subset 4	*4*	*13*	*14*	*15*	*16*

Table 1: An instance that the algorithm will produce an incorrect result

For each set that was added to our result, the number of sets we missed increases by $|C_i| - 1$. We will have k many sets and the at the end, and the relationship between $k^*$and k is the following:

$$k^* = |B| - k$$

$$\frac{k^*}{k} = \frac{|B| - k}{k}$$

Let's name the resulting family of subsets we found R. Here, $|B|$ is equal to the number of elements in R. Therefore:

$$|B| = \sum_{i=1}^{|R|} |R_i|$$

And our ratio bound will be:

$$\rho(N) = \frac{\sum_{i=1}^{|R|} |R_i| - |R|}{|R|}$$

Note that $|R|$ is bounded by N, and in the case that $k = k^* = N = |R| = |C|$, the ratio bound will be equal to 1 because no sets will be missed in R.

3. Algorithm Analysis

3-a) Brute Force Algorithm

For this part, the following notation will be used:

m = cardinality of C
n = cardinality of the largest set among the elements of C

D_i = the largest cardinality subset of C with pairwise disjoint elements when the size of C is i.

Let's prove the correction of the algorithm with the induction method.

Base Step (m = 1):
When C has only 1 element, $D_1 = C$ and is the largest cardinality subset of C that is composed of pairwise disjoint elements. Therefore the base case holds.

Inductive Step:
Inductive Hypothesis: We assume that for some subset D_k of C_k with $k < m$, our conditions hold with cardinality $|D_k|$.
Let's check if this is the case for $k + 1$.
We check every subset with the $k + 1^{th}$ element whether there is a subset that satisfies the disjointness and has cardinality larger than $|D_k|$. If it has, then $|D_{k+1}|$ is the largest cardinality subset with pairwise disjoint elements. If it does not, we know that D_k is the largest cardinality set.
In both cases, the inductive hypothesis holds.
Therefore, we have proven that our algorithm is correct through the inductive method.

Show its complexity

```
// finds the biggest subset that every set in it are disjoint
vector<vector<int>> & subsetFinder(vector< vector<int> > & v) {
  bool flag = false;
  // number of possible subsets of C
  int subSetNumber = pow(2, v.size());
  // holds the subset instance
  vector< vector<int> > subsetHolder;
  // holds the biggest disjoint subset so far
  vector< vector<int> > bestSubset (0);
  // we iterate through every possible subset and create subsets
  for(int i = 0; i < subSetNumber; i++) { // LOOP 1
    // for every element in C
    for(int j = 0; j < v.size(); j++) { // LOOP 2
      // this part is for creating different subsets
      if((i & (1 << j)) != 0) {
        subsetHolder.push_back(v[j]);
      }
    }
    // for every element in subsetHolder index[0, length - 2]
    for(int i = 0; i + 1 < subsetHolder.size(); i++) { // LOOP 3
      // for every element in subsetHolder index[1, length - 1]
      for(int j = i + 1; j < subsetHolder.size(); j++) { // LOOP 4
        // compare each pair if they are intersecting
        if(isIntersecting(subsetHolder[i], subsetHolder[j])) // ISINTERSECTING
          flag = true;
      }
    }
    /* if given subset instance is disjoint and is bigger than bestSubset,
      define bestSubset as subsetHolder */
```

```
    if(!flag && subsetHolder.size() > bestSubset.size())
      bestSubset = subsetHolder;
    subsetHolder.clear();
    flag = false;
  }
return bestSubset;
}
```

LOOP1 goes through every subset of C, therefore it runs 2^mtimes.
LOOP2 goes through C, therefore it runs m times.
LOOP3 goes through the elements of the subset instance, and runs m times in the worst case.
LOOP4 goes through the elements of the subset instance, and runs m times in the worst case.
ISINTERSECTING is a function that compares every element of two given subsets. Runs n^2 times in the worst case.

Therefore the total running time is: $2^m * (m + m^2 * n^2)$

And the time complexity of the algorithm: $O(2^m * m^2 * n^2)$

The space complexity of the algorithm is $O(m * n)$ since the extra memory will be required for the subset that satisfies both conditions and that subset can be equal to the size of C at maximum. And in the worst case, all elements of C have cardinality equal to n.

3-b) For Approximation Algorithm

Correctness of the Algorithm

Our algorithm does not always work correctly. For every set C_i we accept as disjoint, there is a chance that there are disjoint sets as many as the length of set C_i that could have been accepted as disjoint sets, but we missed them because we added C_i. This was already demonstrated in section 2-b.

Complexity Analysis

For this part, we will use the following notation:

$C = \{C_1, C_2, C_3, ..., Cn\}$ is the input to our algorithm, a family of sets.

$N = |C|$, number of sets in C.

$m_i =$ the number of elements in C_i.

$k =$ our output, the number of sets that are disjoint.

$R = \{R_1, R_2, R_3, ..., R_k\}$ is the maximum cardinality subset of C that has mutually disjoint elements.

$B = R_1 \cup R_2 \cup R_3 \cup ... \cup R_k$, the flat set of elements of sets that we chose as disjoint.

$B_i =$ the set B at the end of i^{th} iteration of the algorithm.

First, sorting the elements of C takes (assuming computing vector size takes constant time:

$$O(N * log(N))$$

We iterate through the sets in C, this means iterating N times.

At each iteration, either C_i is empty or we make comparisons between the elements of B_{i-1} and C_i and see if the elements of C_i are in B_{i-1}. The number of comparisons in each iteration is:

$$log|B_{i-1}| * m_i$$

If B_{i-1} and C_i are disjoint, we add the elements of C_i to B_{i-1}, sort B and increment the count of disjoint sets by 1 (Note that B_i will be almost sorted, therefore it will take linear time. But we will assume quadratic for worst-case). This takes:

$$|C_i| + B_i^2 + 1$$

In the end, the total running time of the algorithm is:

$$N * log(N) + N * (log|B_{i-1}| * m_i + m_i + B_i^2 + 1)$$

For the worst-case complexity analysis, we will assume that we will always add C_i to our disjoint sets. Therefore we will have compared each element of C_i with each element of B_{i-1} which will be equivalent to $C_1 \cup C_2 \cup ... \cup C_{i-1}$.

Then we will always sort B_i which has $m_1 + m_2 + ... + m_i$ elements. The total number of comparisons are:

$$0 * m_1 + log(m_1) * m_2 + log(m_1 + m_2) * m_3 + log(m_1 + m_2 + m_3) * m_4 + ...log(m_1 + m_2 ... + m_{k-1}) * m_k$$

Note that the first production will be equal to 0, therefore we start our sum from $j = 2$.

Let us formulate the process of comparison at each iteration as the following:

$$m_i * log(\sum_{j=1}^{i-1} m_j)$$

Sorting B_i at each step will take:

$$(\sum_{j=1}^{i} m_j)^2$$

The total running time can be formulated as follows. Note that we have to add the cost of sorting B_1 as well:

$$m_1^2 + \sum_{i=2}^{N} (m_i * log(\sum_{j=1}^{i-1} m_j) + (\sum_{k=1}^{i} m_k)^2)$$

In the worst case, we will assume that each m_i is equal to the maximum among every m_i and we will denote it as m.

$$m^2 + \sum_{i=2}^{N} (m * log(\sum_{j=1}^{i-1} m) + (\sum_{k=1}^{i} m)^2)$$

Let's simplify this a bit:

$$m^2 + \sum_{i=2}^{N} (m * log((i-1) * m) + (i * m)^2)$$

$$= m^2 + \sum_{i=2}^{N} (m * log((i-1) * m)) + \sum_{i=2}^{N} (i * m)^2$$

Here, it can be seen that the sum at the left will always be larger than the right side, therefore we can ignore the right side. Therefore, we have:

$$m^2 + \sum_{i=2}^{N} (m * log(i-1) + m * log(m)) + \sum_{i=2}^{N} (i * m)^2$$

$$= (N-1) * (m * log(m)) + m^2 + \sum_{i=2}^{N} (m * log(i-1)) + m^2 \sum_{i=2}^{N} i^2$$

$$= (N-1) * (m * log(m)) + m^2 + m * \sum_{i=2}^{N} (log(i-1)) + m^2 * (\frac{N * (N+1) * (2N+1)}{6} - 1)$$

$$= (N-1) * (m * log(m)) + m^2 + m * log(\prod_{i=2}^{N} (i-1)) + m^2 * (\frac{2N^3 + 3N^2 + N}{6} - 1)$$

Let's simplify this by turning parts of it to O notation:

$$O(N * m * log(m)) + O(m^2) + m * log((N-1)!) + O(m^2 * N^3)$$

$$= O(m^2 * N^3) + m * log((N-1)!)$$

Note that $(N-1)! = O(N^N)$:

$$O(m^2 * N^3) + m * log(N^N) = O(m^2 * N^3) + m * N * log(N)$$

As a result, our algorithm is:

$$O(m^2 * N^3)$$

Space Complexity

The algorithm uses extra space for set B, which will be equal to the total number of elements in the subsets of R. In the worst case, R will be equivalent to the input C, therefore the worst-case space complexity is equal to $N * m$, where N is the number of sets in C and m is the cardinality of the largest set in C.

4. Sample Generation

Function for generating random sets with a given size:

```
// generates random vectors of given size
vector<int> randomVector_size(int size_v) {
  int size = size_v;
  vector<int> v (size);
  for(int i = 0; i < size; i++) {
    v[i] = rand() %50;
  }
  return v;
}
```

Function for generating random sets of random sizes:

```
// generates random vectors of random sizes
vector<int> randomVectorGenerator() {
  int size = rand() % 6;
  vector<int> v (size);
  for(int i = 0; i < size; i++) {
    v[i] = rand() % 50;
  }
  return v;
}
```

Testing with $|C|$ = 3:

```
vector 1:  23  19  45  6  39  40
vector 2:  16  42  16  15  4  5
vector 3:  45  13  14  31  6  33  23  40  8  21  40

Subset With the Best Cardinality:
23  19  45  6  39  40
16  42  16  15  4  5
Cardinality = 2
Running time : 0.004 seconds
```

Testing with $|C|$ = 5:

```
vector 1:  43  5
vector 2:  10  30  27  49  22  40  14  21  33
vector 3:  26  7  13  7
vector 4:  31  17  42  43  11  36  49  2
vector 5:  22  31  1  19  9  10  3  15  22  37  33  34

Subset With the Best Cardinality:
43  5
10  30  27  49  22  40  14  21  33
26  7  13  7
Cardinality = 3
Running time : 0.01 seconds
```

Testing with $|C|$ = 8:

Testing with $|C|$ =10 :

```
vector 1:  17  34
vector 2:  19  24  28  8  12  14
vector 3:  45  31  27  11  41  45  42  27  36  41  4
vector 4:  3  42
vector 5:  21  16  18  45  47  26
vector 6:  38  19  12
vector 7:  49  35  44  3  11
vector 8:  33  23  14  41  11

Subset With the Best Cardinality:
17  34
3  42
21  16  18  45  47  26
38  19  12
33  23  14  41  11
Cardinality = 5
Running time : 0.021 seconds
```

```
vector 1:  19  38  37  5  47  15  35  0  12  3  0  42
vector 2:  37  21  45  35  47  30  26  41
vector 3:  6  7  23  31  40  25  28  46  40  40
vector 4:  7  37  11  17  6  17  33  28  23  37
vector 5:  34  12  11  28  16  29  4
vector 6:  5  38  49  29  26  31  14
vector 7:  36
vector 8:  2  2  4  37  6
vector 9:  22  47  13  33  3  10  42  47
vector 10:  21  4  23  2  19  4  39  36  4  37

Subset With the Best Cardinality:
5  38  49  29  26  31  14
36
2  2  4  37  6
22  47  13  33  3  10  42  47
Cardinality = 4
Running time : 0.107 seconds
```

Testing with $|C|$ =15 :

```
vector 1:  17  34
vector 2:  19  24  28  8  12  14
vector 3:  45  31  27  11  41  45  42  27  36  41  4
vector 4:  3  42
vector 5:  21  16  18  45  47  26
vector 6:  38  19  12
vector 7:  49  35  44  3  11
vector 8:  33  23  14  41  11
vector 9:  18  47  44  12
vector 10:  37  9  23  41  29  28  16  35  40  42
vector 11:  6  40
vector 12:  14  48  46  5  40  29  20  0  6  1  43
vector 13:  29  23  34  4  6  40  16  26  31  8  44  39
vector 14:  23  37  38  18
vector 15:  29  41

Subset With the Best Cardinality:
17  34
3  42
21  16  18  45  47  26
38  19  12
33  23  14  41  11
6  40
Cardinality = 6
Running time : 3.433 seconds
```

5. Experimental Analysis of The Performance (Performance Testing)

a) Experimental Analysis of Running Time

We will analyze the running time of the algorithm experimentally by calculating the mean, standard deviation, standard error, and then we will use these to create confidence intervals.

1. Standard Deviation

The function above calculates the standard deviation.

$$\sigma = \sqrt{\frac{\Sigma(x_i - \mu)^2}{n-1}}$$

```
double stddeviation(vector<double> & v){
  double m = mean(v);
  double accum = 0.0;
  for(int i = 0; i < v.size(); i++) {
    accum=accum+(m-v[i])*(m-v[i]);
  }
  double stdev = sqrt(accum / (v.size()-1));
  return stdev;
}
```

2. Standard Error

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

```
double standart_error=st/sqrt(sizes[s]);
```

3. Performance Tests for The Algorithm

3.1 Keeping The Cardinality of the Subsets Constant

In this part, we will assume every element of *C* has the cardinality of 30, however the cardinality of *C* is not fixed.

Table 2 shows the statistics when we repeat the algorithm 100 times.

Size	Mean	Standard Deviation	Standard Error	90% Confidence Interval	95% Confidence Interval
100	*0.0000930465*	*0.00029057*	*0.000090557*	0.0000452186 - 0.0001408744	0 - 0.0005787325
500	*0.0000431466*	*0.000626877*	*0.000028034*	0 - 0.0000892919	0 - 0.0003307376
1000	*0.00119493*	*0.00143302*	*0.0000453162*	0.0011203396 - 0.0012695204	0.00092776 - 0.0014621
1500	*0.00225513*	*0.00239902*	*0.0000619424*	0.0022449343 - 0.0022653257	0.001950757 - 0.002559503
2000	*0.00365003*	*0.00375745*	*0.0000840192*	0.0036362004 - 0.0036638596	0.00339274 - 0.00390732
2700	*0.00579523*	*0.00622936*	*0.000119884*	0.005597901 - 0.005992559	0.005582192 - 0.006008268
3500	*0.00893568*	*0.0100445*	*0.000169783*	0.008656217 - 0.009215143	0.008749149 - 0.009122211
4250	*0.0132194*	*0.0152182*	*0.000233436*	0.012835164 - 0.013603636	0.013036861 - 0.013401939
5000	*0.0182866*	*0.0207973*	*0.000294118*	0.017802482 - 0.018770718	0.018096342 - 0.018476858

Table 2: Statistics of running time when input size is changed (running the algorithm 100 times)

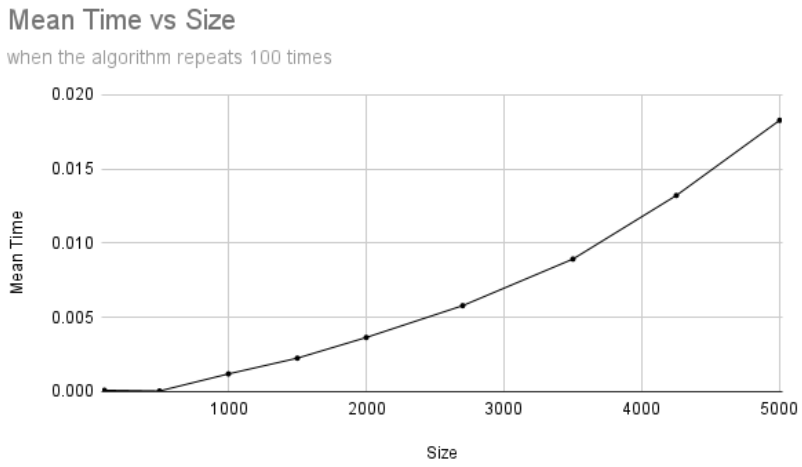

Fig. 4: Shows the relationship between the input size and mean running time (repeating 100 times)

Table 3 shows the statistics when we repeat the algorithm 1000 times.

Size	Mean	Standard Deviation	Standard Error	90% Confidence Interval	95% Confidence Interval
100	*0.000224224*	*0.00122261*	*0.000122261*	0.000022984 - 0.000425464	0.000020481 - 0.000427967
500	*0.000874374*	*0.0232203*	*0.000103844*	0.000703454 - 0.001045294	0.000670631 - 0.001078117
1000	*0.001664*	*0.0036505*	*0.000100088*	0.00149926 - 0.00182874	0.001467628 - 0.001860372
1500	*0.00242442*	*0.00369528*	*0.0000954118*	0.00226738 - 0.00258146	0.002237222 - 0.002611618
2000	*0.00344505*	*0.00465635*	*0.000104119*	0.00327375 - 0.00361635	0.003240768 - 0.003649332
2700	*0.004333*	*0.00562881*	*0.000108327*	0.0041547 - 0.0045113	0.004120463 - 0.004545537
3500	*0.00567525*	*0.00667693*	*0.000112861*	0.005489481 - 0.005861019	0.005453817 - 0.005896683
4250	*0.00765728*	*0.00878073*	*0.00013469*	0.00743558 - 0.00787898	0.007393018 - 0.007921542
5000	*0.00919709*	*0.00982904*	*0.000139004*	0.00896829 - 0.00942589	0.008924365 - 0.009469815

Table 3: Statistics of running time when input size is changed (running the algorithm 1000 times)

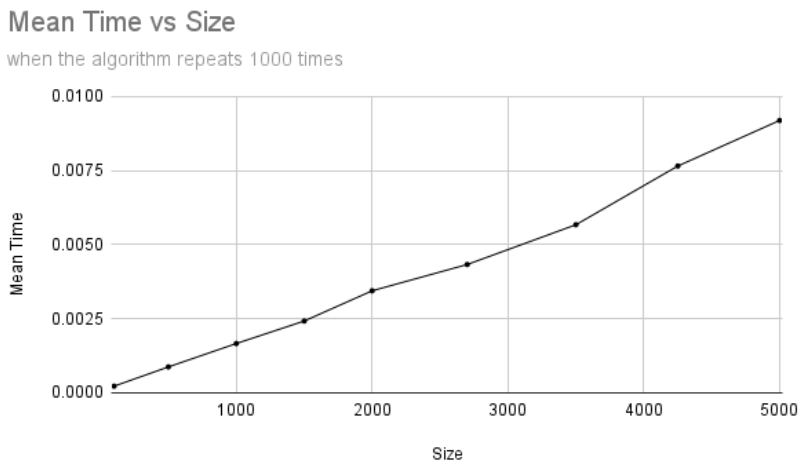

Fig. 5: Shows the relationship between the input size and mean running time (repeating 1000 times)

Table 4 shows the statistics when we repeat the algorithm 2000 times.

Size	Mean	Standard Deviation	Standard Error	90% Confidence Interval	95% Confidence Interval
100	*0.000233117*	*0.000427629*	*0.0000427629*	*0.000022984 - 0.000425464*	*0.000020481 - 0.000427967*
500	*0.00121011*	*0.00125402*	*0.0000560813*	*0.000703454 - 0.001045294*	*0.000670631 - 0.001078117*
1000	*0.00193997*	*0.00151941*	*0.0000480479*	*0.00149926 - 0.00182874*	*0.001467628 - 0.001860372*
1500	*0.0026567*	*0.00186889*	*0.0000482546*	*0.00226738 - 0.00258146*	*0.002237222 - 0.002611618*
2000	*0.00346513*	*0.00237576*	*0.000053123*	*0.00327375 - 0.00361635*	*0.003240768 - 0.003649332*
2700	*0.00448674*	*0.00321041*	*0.0000617843*	*0.0041547 - 0.0045113*	*0.004120463 - 0.004545537*
3500	*0.00567784*	*0.00422348*	*0.0000713898*	*0.005489481 - 0.005861019*	*0.005453817 - 0.005896683*
4250	*0.00698649*	*0.0053242*	*0.0000816695*	*0.00743558 - 0.00787898*	*0.007393018 - 0.007921542*
5000	*0.00838575*	*0.00645989*	*0.000091356*	*0.00896829 - 0.00942589*	*0.008924365 - 0.009469815*

Table 4: Statistics of running time when input size is changed (running the algorithm 2000 times)

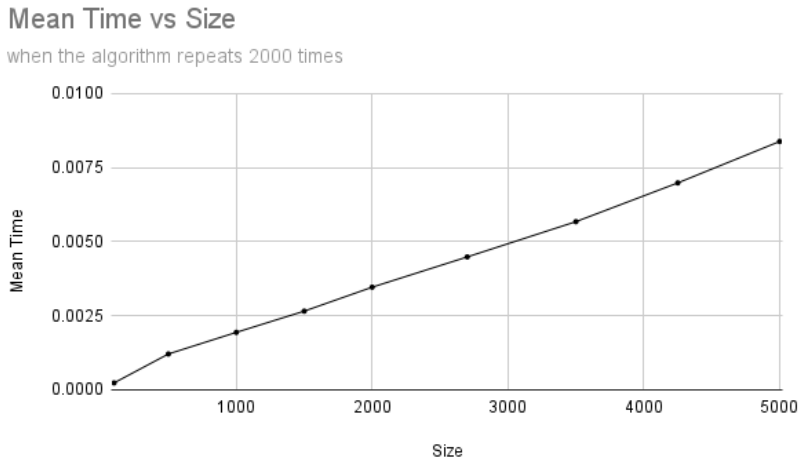

Fig. 6: Shows the relationship between the input size and mean running time (repeating 2000 times)

3.2 Keeping The Cardinality of the Set Constant

In this part, we will assume C has the cardinality of 30, however the cardinality of the elements of C is not fixed.

Table 5 shows the statistics when we repeat the algorithm 100 times.

Size	Mean	Standard Deviation	Standard Error	90% Confidence Interval	95% Confidence Interval
100	*0.000111111*	*0.000855819*	*0.0000855819*	*0 - 0.000253171*	*0 - 0.000280906*
500	*0.00080303*	*0.00218825*	*0.0000978613*	*0.00064063 - 0.00096543*	*0.000608873 - 0.000997187*
1000	*0.00136364*	*0.00297099*	*0.0009395*	*0.001207681 - 0.001519599*	*0.001177241 - 0.001550039*
1500	*0.00204545*	*0.00357459*	*0.000922955*	*0.00189224 - 0.00219866*	*0.001862336 - 0.002228564*
2000	*0.00321818*	*0.00453025*	*0.000101299*	*0.003050023 - 0.003386337*	*0.003017202 - 0.003419158*
2700	*0.00533838*	*0.00696861*	*0.000134111*	*0.005115756 - 0.005561004*	*0.005072304 - 0.005604456*
3500	*0.0078557*	*0.00912197*	*0.000154189*	*0.007599745 - 0.008111655*	*0.00479658 - 0.01091482*
4250	*0.0111225*	*0.0123853*	*0.000189981*	*0.00796881 - 0.01427619*	*0.0107848077 - 0.0114601923*
5000	*0.0149809*	*0.0164137*	*0.000232125*	*0.014595573 - 0.015366227*	*0.014520364 - 0.015441436*

Table 5: Statistics of running time when the size of sets in C is changed (running 100 times)

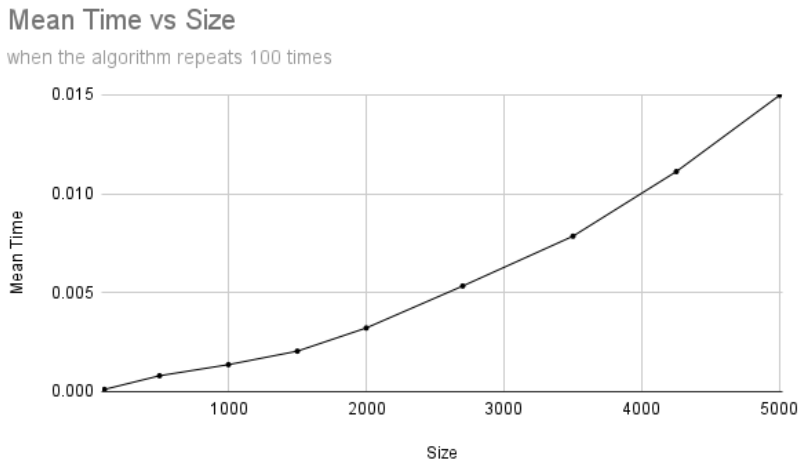

Fig. 7: Shows the relationship between the size of subsets in our input and mean running time (repeating 100 times)

Table 6 shows the statistics when we repeat the algorithm 1000 times.

Size	Mean	Standard Deviation	Standard Error	90% Confidence Interval	95% Confidence Interval
100	*0.0000970971*	*0.000740729*	*0.0000740729*	*0 - 0.0002190211*	*0 - 0.0002424281*
500	*0.000359359*	*0.00155035*	*0.0000693339*	*0.000245235 - 0.000473483*	*0.000223326 - 0.000495392*
1000	*0.000936937*	*0.00354687*	*0.0000796837*	*0.000805778 - 0.001068096*	*0.000780598 - 0.001093276*
1500	*0.0019029*	*0.00354687*	*0.0000915799*	*0.00175216 - 0.00205364*	*0.00172322 - 0.00208258*
2000	*0.00306867*	*0.00434582*	*0.000971755*	*0.00290872 - 0.00322862*	*0.002878012 - 0.003259328*
2700	*0.00492476*	*0.00617816*	*0.000118899*	*0.004729053 - 0.005120467*	*0.004691481 - 0.005158039*
3500	*0.00734835*	*0.00877273*	*0.000148286*	*0.007104271 - 0.007592429*	*0.00443898 - 0.01025772*
4250	*0.010462*	*0.0119719*	*0.000182106*	*0.01016226 - 0.01076174*	*0.010104708 - 0.010819292*
5000	*0.0141426*	*0.0155286*	*0.000219608*	*0.01378113 - 0.01450407*	*0.013711729 - 0.014573471*

Table 6: Statistics of running time when the size of sets in C is changed (running 1000 times)

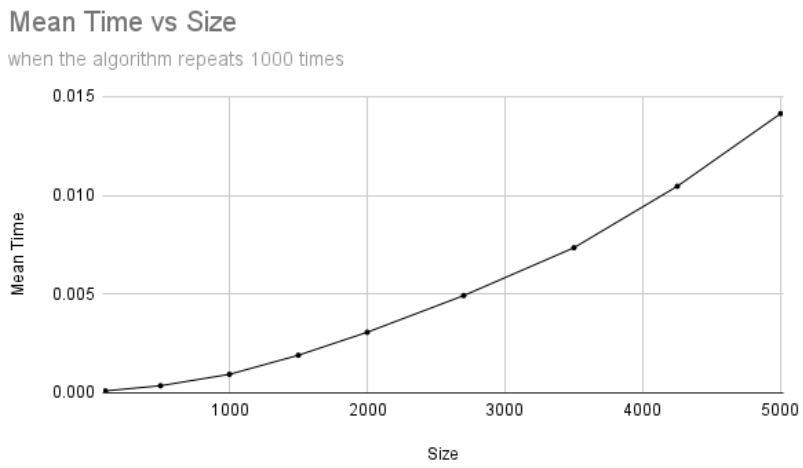

Fig. 8: Shows the relationship between the size of subsets in our input and mean running time (repeating 1000 times)

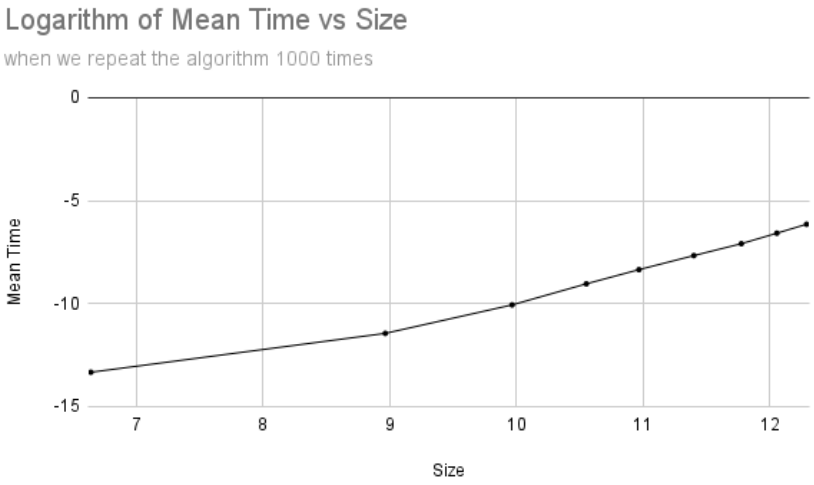

Fig. 9: Shows the relationship between log of the size of subsets in our input and log mean running time (repeating 1000 times)

Table 7 shows the statistics when we repeat the algorithm 2000 times.

Size	Mean	Standard Deviation	Standard Error	90% Confidence Interval	95% Confidence Interval
100	*0.000103052*	*0.000310616*	*0.0000310616*	*0.0000519246 - 0.0001541794*	*0.0000421402 - 0.0001639638*
500	*0.000424962*	*0.000584383*	*0.0000261344*	*0.0003819448 - 0.0004679792*	*0.0003737125 - 0.0004762115*
1000	*0.0010217*	*0.0010978*	*0.0000347153*	*0.0009645585 - 0.0010788415*	*0.0009536232 - 0.0010897768*
1500	*0.00181416*	*0.00172005*	*0.000044116*	*0.0017410586 - 0.0018872614*	*0.0017270689 - 0.0019012511*
2000	*0.00288044*	*0.00266024*	*0.0000594847*	*0.0027825282 - 0.0029783518*	*0.002763791 - 0.002997089*
2700	*0.00461414*	*0.00468446*	*0.0000901525*	*0.00446575 - 0.00476253*	*0.004437351 - 0.004790929*
3500	*0.00709155*	*000749908*	*0.000126758*	*0.006882907 - 0.007300193*	*0.006842978 - 0.007340122*
4250	*0.0102953*	*0.0110545*	*0.000169568*	*0.010016191 - 0.010574409*	*0.009962777 - 0.010627823*
5000	*0.0141727*	*0.0152308*	*0.000215396*	*0.013818158 - 0.014527242*	*0.013750308 - 0.014595092*

Table 7: Statistics of running time when the size of sets in C is changed (running 2000 times)

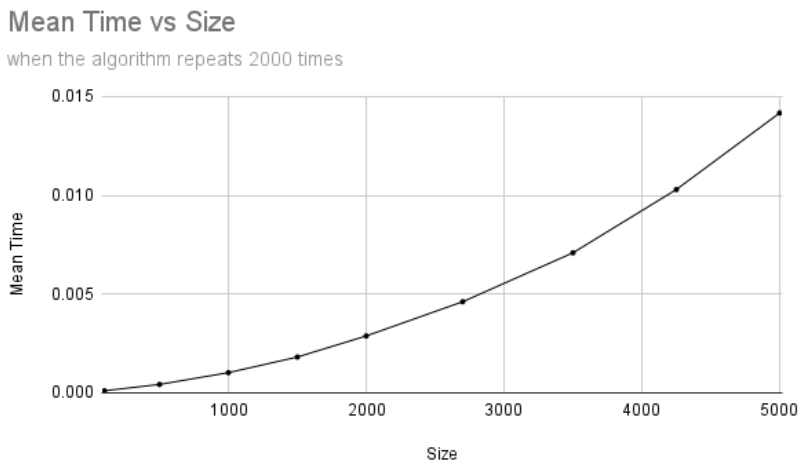

Fig. 10: Shows the relationship between the size of subsets in our input and mean running time (repeating 2000 times)

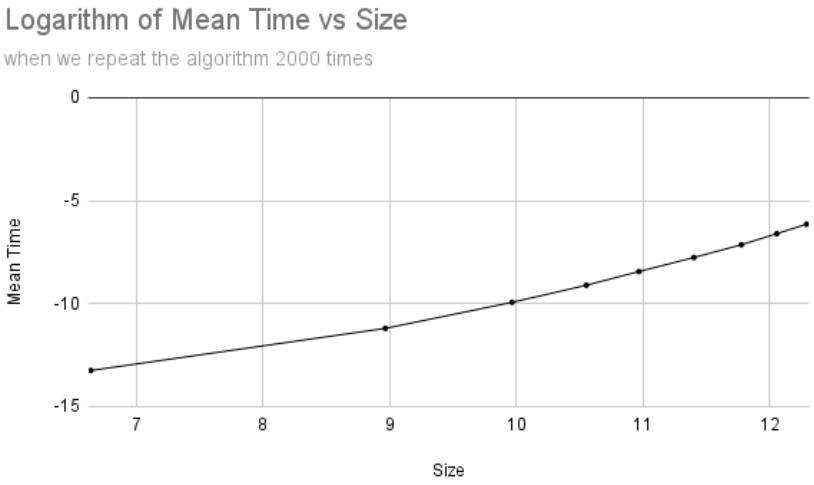

Fig. 11: Shows the relationship between log of the size of subsets in our input and log mean running time (repeating 2000 times)

3.3 Approximation/Brute-Force Running Time Comparison

Quality = (1 / Ratio)

Number of Inputs (rows) \ Cardinality of C (columns)	**10**	**11**	**12**	**13**	**14**	**15**
100	*0.26*	*0.295*	*0.293333*	*0.255*	*0.236*	*0.21*
500	*0.344*	*0.318*	*0.292*	*0.268*	*0.2436*	*0.221*
1000	*0.363*	*0.3135*	*0.288333*	*0.2595*	*0.237*	*0.2176*
2000	*0.3415*	*0.3075*	*0.2865*	*0.25925*	*0.2343*	*0.217333*

Table 8: Number of repeats (rows) vs. Cardinality of C (columns)

We have compared the running time of our approximation algorithm with the brute-force algorithm we have implemented before.

6. Experimental Analysis of the Correctness (Functional Testing)

Black Box Testing.

1. $C = \{\{1\}, \{2\}, \{3\}, ..., \{N\}\}$

2. $C = \{\{1\}, \{1\}, \{1\}, ..., \{1\}\}$
3. $C = \{\{1, 2\}, \{2, 3\}, \{3, 4\}, ..., \{N, N+1\}\}$
4. $C = \{\{1, 2, 3\}, \{2, 3, 4\}, \{3, 4, 5\}, ..., \{N, N+1, N+2\}\}$

	Results from Implementation	Theoretical Results
1	N	N
2	1	1
3	$\lceil \frac{N}{2} \rceil$	$\lceil \frac{N}{2} \rceil$
4	$\lceil \frac{N}{3} \rceil$	$\lceil \frac{N}{3} \rceil$

Table 9: Comparison of the practical and theoretical results of the algorithm (Black Box Testing)

White Box Testing:

1. $C = \emptyset$
2. $C = \{\emptyset, \emptyset, \emptyset, ..., \emptyset\}$
3. $C = \{\{1, 2, 3, 4\}, \{2, 5, 6, 7, 8\}, \{3, 9, 10, 11, 12\}, \{4, 13, 14, 15, 16\}\}$

1^{st} case is for covering the behavior of the if statements on the function “find_greedy”.

2^{nd} case is for covering the behavior of the function “find_greedy” when there is an empty input.

3^{rd} case is for covering the behavior of the function “find_greedy” when the first set blocks the others.

	Results from Implementation	Theoretical Results
1	0	0
2	1	1
3	1	1

Table 10: Comparison of the practical and theoretical results of the algorithm (White Box Testing)

7. Discussion

In our experiments, we have found no defects in the algorithm.

The results of our experiments showed that the algorithm usually works much faster than our worst-case scenarios.

1. When the size of C was increased, the running time increased linearly. The theoretical results we provided in Section 3-b show that the relationship between the running time and input size in the worst-case would be $O(N^3)$. This shows that the worst-case scenario is not very likely to happen.
2. When the sizes of sets in C were increased, the running time increased exponentially. This was the expected result of our theoretical work.

The ratio bound we provided in Section 2-b was related to the size of the result, therefore to the size of the input C. As $|C|$ increases, it gets more likely that the size of the resulting set also increases. In our tests, we used several sizes for the size of sets in C to see the significance of $|C|$ in different situations. The results show that the ratio bound is linked to $|C|$ in practice as well.

8. Reference

Karp, R. M. (1972). Reducibility Among Combinatorial Problems.. In R. E. Miller & J. W. Thatcher (eds.), *Complexity of Computer Computations* (p./pp. 85-103), : Plenum Press, New York. ISBN: 0-306-30707-3