Maximum Set Packing

Ege Demirci: 28287 , Efe Tüzün: 28992

Sabancı University, Spring 2022-2023, Algorithms

1. Problem Description

1.a. Intuitive Description

We are given a set of finite sets $C = \{C_1, C_2, \ldots, C_m\}$ as input. We are then asked to find the largest number of mutually disjoint sets of C. An example input and output for the Set Packing Problem is as follows:

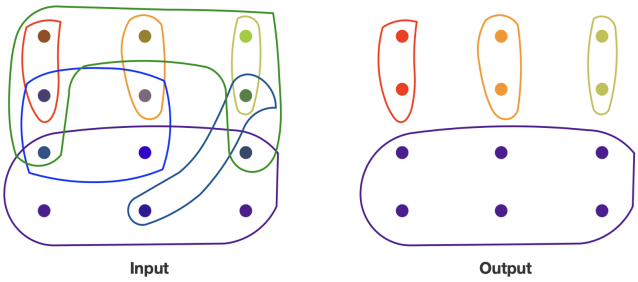


Figure 1: Example Input and Output to Set Packing Problem

1.b. Formal Description

The formal definition of Set Packing is as follows:
Given a collection $C = \{C_1, C_2, \ldots, C_m\}$ of finite sets as input, what is the largest cardinality subset $D = \{D_1, D_2, \ldots, D_k\}$ of C such that $\forall i, j$ that satisfies $1 \leq i < j \leq k$, D_i and D_j have no common elements? In other words, what is the largest value for $|D| = n$, where n is an integer, given the above input and constraints?

1.c. Practical Uses of the Problem

One of the most important practical uses of the Set Packing Problem is the scheduling of the airline flight members. Each plane needs to be assigned different members of the crew such as a pilot, a copilot, and cabin crew. There are numerous constraints on the scheduling that needs to be taken care of, some of which are the relationship between the crew members, work schedules, holiday dates, training to fly different types of aircrafts etc., when composing flight crews. When all possible crew and plane combinations are constructed, we need an assignment such that each person is chosen in exactly one combination, and assigned to exactly one plane, as there is no possibility of a person being in two or more planes simultaneously and vice-versa. Another application domain is resource allocation. In resource allocation, the problem can be used to find the maximum number of tasks that can be performed concurrently with limited resources.

1.d. Hardness of the Problem

As with all other problems, Set Packing has an Optimization Problem and a Decision Problem. The optimization problem, also referred to as the Maximum Set Packing Problem, is the one that we would like to analyze. Also, there exists a corresponding decision problem, which was proven to be NP-Complete by Karp in 1972. Below is the list of problems that Karp has evaluated.

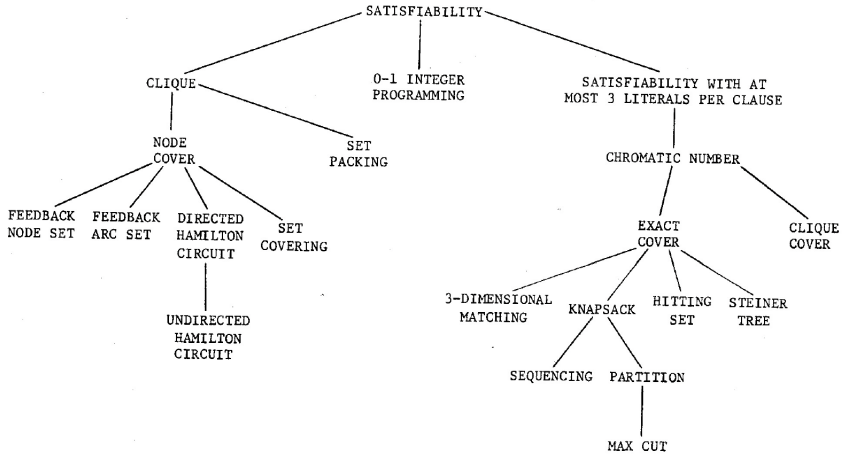


Figure 2: The Original 21 NP-Complete Problems

In that paper (1972), Karp first lists all problems that he will prove are in NP-Complete, all of which are in NP. He later shows, according to the above graph, ways to reduce from known NP-Complete problems. He starts of with proving SAT'S reducibility to Clique. Then, Karp shows an algorithm to get from a clique instance to a set packing instance. This shows that Clique, which is an NP-Complete problem, can be reduced to Set Packing. Thus, he showed that Set Packing was an NP-Complete problem. In his paper, Set Packing is defined as the decision problem variant

On the other hand, our problem, Maximum Set Packing, is NP-Hard. To prove this, we have chosen the Maximum Independent Set problem which is a known NP-hard problem and we aim to demonstrate that it can be reduced to the Maximum Set Packing problem in polynomial time, thus proving it is at least as hard as Maximum Independent Set Problem.

First, we will show an example of how such an algorithm works, and then we will present a pseudocode for the reduction of a given Maximum Independent Set instance to the Maximum Set Packing instance.

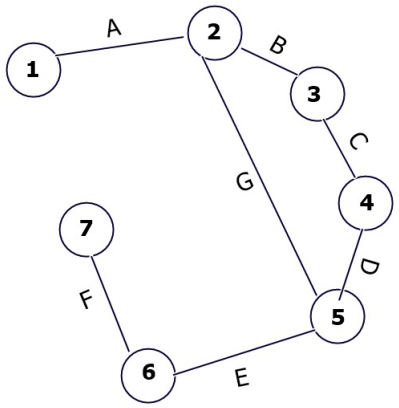


Figure 3: An instance of a Maximum Independent Set Problem

Above is a problem instance for the Maximum Independent Set Problem. Here, we will construct $C = \{C_1, C_2, \ldots, C_7\}$ as follows, where each set C_i corresponds to the set for the node labelled with i. For each node, we will add the labels of the edges to its set, C_i.

$$
\begin{aligned}
C_1 &= \{A\} \\
C_2 &= \{B, G\} \\
C_3 &= \{B, C\} \\
C_4 &= \{C, D\} \\
C_5 &= \{G, D, E\} \\
C_6 &= \{E, F\} \\
C_7 &= \{F\}
\end{aligned}
$$

Thus, we will be constructing an instance of a Maximum Set Packing Problem. Below is the implementation of a polynomial time algorithm that will reduce a Maximum Independent Set Problem instance to the Maximum Set Packing Problem instance.

Algorithm 1: Reducing Maximum Independent Set to Maximum Set Packing

Input: A graph $G = (V, E)$
Output: A set of sets C
$C \leftarrow \{\}$
for v_i *in* V **do**
    Add the set C_i for the vertex v_i as an empty set to C
end
for $e = (u, v)$ *in* E **do**
    Add the edge label to the set C_u and C_v in C
end
return C

We also need to prove that there exists a solution for the Maximum Independent Set Problem iff there exists a solution for the Set Packing Problem. In other words, we need to show that it correctly transforms an instance of the maximum independent set problem into an instance of the maximum set packing problem, such that the solutions of the transformed instance correspond to solutions of the original instance.

Let us define again the two problems formally:

Maximum Independent Set Problem: Given an undirected graph $G = (V, E)$, find a maximum cardinality subset $S \subseteq V$ such that no two vertices in S are adjacent.

Maximum Set Packing Problem: Given a universal set U and a family of subsets $S = \{S_1, S_2, \ldots, S_k\}$ of U, find a maximum number of disjoint subsets from S.

Firstly, let's assume that we have an instance of the maximum independent set problem, which is an undirected graph $G = (V, E)$, and we want to transform it into an instance of the maximum set packing problem.

Our reduction algorithm works by constructing a family of subsets S from the vertices and edges of G. Each vertex v in V corresponds to a subset s_v, which contains no elements at the first state. Then, for each edge $e = (v_i, v_j)$ in E, we add e to both s_i and s_j. Thus, each subset s_v contains all the edges incident to v.

We claim that the resulting family of subsets S and the original graph G have the following properties:

1 - The size of a maximum independent set in G is equal to the size of a maximum set packing in S.

Proof: Suppose S^* is a maximum set packing in S. We can construct an independent set I^* in G as follows: For each subset s_v in S^* that contains an edge $e = (v_i, v_j)$, we include only one of the vertices v_i or v_j in I^*. Since S^* is a set packing, no two subsets in S^* share an element, so intuitively the vertices we choose for I^* are guaranteed to be non-adjacent in G. Thus, I^* is an independent set in G. Similarly, given any independent set I in G, we can construct a set packing S_I in S, where each subset s_v in S_I contains the edges incident to v in I. Therefore, the size of a maximum independent set in G is equal to the size of a maximum set packing in S.

2 - For any subset S^* of V, the set of subsets $s_v : v \in S^*$ in S is a set packing if and only if the vertices in S^* are pairwise non-adjacent in G.

Proof: Suppose $\{s_v : v \in S^*\}$ is a set packing in S. Also, assume that there exist two vertices $v_i, v_j \in S^*$ that are adjacent in G. Then, the subsets s_i and s_j in S share the edge (v_i, v_j), which violates the definition of a set packing. Conversely, suppose the vertices in S^* are pairwise non-adjacent in G. Then, for any two subsets $s_i, s_j \in S$ that correspond to vertices $v_i, v_j \in S^*$, there are no common edges, because v_i and v_j are non-adjacent in G. Therefore, $\{s_v : v \in S^*\}$ is a set packing in S.

Since we proved both statements, we can conclude that the algorithm correctly transforms an instance of the maximum independent set problem into an instance of the maximum set packing problem, and the solutions of the transformed instance correspond to solutions of the original instance. Thus, the reduction algorithm works correct. It will run in $O(n^2)$, where $n = |V|$, assuming that the given graph is a complete graph with n vertices.

Because it runs in polynomial time, and it works correct for all instances of Maximum Independent Set Problem, there exists a polynomial time reduction from Maximum Independent Set to Maximum Set Packing Problem. Thus, we have proved that Maximum Set Packing is an NP-Hard problem.

2. Algorithm Description

2.a. Brute-Force Algorithm

We propose the following brute force algorithm to solve the set packing problem. The algorithm below aims to find a subset of a given set C with mutually disjoint elements with the largest cardinality. The algorithm works as follows:

1. Initialize an empty set called *currentLargest* and a variable called *maxCardinality*, and set it to 0.
2. Generate all possible subsets of the set C using the *powerset* function.
3. For each subset generated, check if its elements are mutually disjoint using the *areElements-Disjoint* function.
4. If the subset's elements are mutually disjoint, count the number of elements in the subset using the *sizeOfSet* function.
5. If the cardinality of the subset (the number of elements in that set) is greater than the current value of *maxCardinality*, set *currentLargest* to this subset and *maxCardinality* to the cardinality of that subset.
6. Return *currentLargest*, the subset with the largest cardinality.

The above algorithm uses three hypothetical helper functions. The *powerSet* function generates all possible subsets of a set. *areElementsDisjoint* and *sizeOfSet* are used to check whether the elements of a subset are mutually disjoint and to find the cardinality of a given set, respectively.

Algorithm 2: Finding the largest subset with mutually disjoint elements

```
Input: A set C
Output: A subset of D with mutually disjoint elements and the largest cardinality
currentLargest ← ∅;
maxCardinality ← 0;
for subset ∈ P(C) do
    if areElementsDisjoint(subset) then
        cardinality ← sizeOfSet(subset);
        if cardinality > maxCardinality then
            currentLargest ← subset;
            maxCardinality ← cardinality;
        end
    end
end
return currentLargest;
```

Our algorithm does not follow any algorithm design techniques.

2.b. Heuristic Algorithm

We propose the following heuristic algorithm to solve the set packing problem. The algorithm works as follows:

1. Initialize an empty list and called *max_set_packing* which will have the most recent maximum set packing solution, and initialize an empty set and called *elements* which will have the elements of the subsets in the maximum set packing.
2. Sort the set C, the input set, using a stable sorting algorithm in increasing order with respect to the cardinalities of the subsets.
3. For each subset in C, check if it is intersecting with *elements*. If it is not, then take the union of its elements with *elements* and add that set to *max_set_packing*. Else, do nothing.
4. Return *max_set_packing*.

The algorithm above follows a greedy approach. It makes locally optimal choices by selecting the smallest cardinality subset that is disjoint with the elements in *max_set_packing*. Note that locally optimal choices do not always yield globally optimal choices, so this algorithm will not always return the correct result, like the most greedy algorithms. This is due to the problem not having the optimal substructure property.

Algorithm 3: Greedy Approach to the Set Packing Problem

Input: A set C
Output: A subset of D with mutually disjoint elements and the largest cardinality
$max\_set\_packing \leftarrow [\,]$;
$elements \leftarrow \emptyset$;
Sort C in ascending order with respect to the cardinality of the subsets;
for $set \in C$ **do**
  if $set \cap elements = \emptyset$ **then**
    $elements = elements \cup set$;
    Add set to $max\_set\_packing$;
  end
end
return $max\_set\_packing$;

Upon further research, we discovered that we could only give a good approximation bound if we restricted the problem to k-Set Packing, which we believe would not be a good choice. Thus, we are not able to provide an approximation bound for the quality of the solution.

3. Algorithm Analysis

3.a. Brute-Force Algorithm

To prove the correctness of the algorithm using induction, we need to show that the algorithm satisfies two conditions: (1) it always terminates, and (2) it always returns a subset of C with mutually disjoint elements with the largest cardinality. We can prove these conditions using mathematical induction.

Theorem: The algorithm described correctly returns the subset of an input set S with the largest cardinality that has mutually disjoint elements. Furthermore, the algorithm terminates for all possible input sets C.

Proof: We will use mathematical induction on the size of the input set C. We will show that the algorithm produces the correct output for all possible input sets.

Base Case: When $|C| = 0$, i.e., the input set is empty. In this case, the algorithm initializes an empty set called *currentLargest* and a variable called *maxCardinality*, and sets it to 0. Then, it returns an empty set, which is the correct output. Therefore, the algorithm is correct for the base case.

Inductive Hypothesis: Assume that the algorithm is correct for all input sets C with $|C| = k$, where k is some positive integer.

Inductive Step: We will show that the algorithm is correct for all input sets C with $|C| = k+1$.

Let C be an input set with $|C| = k + 1$. We can write C as $C' \cup x$, where C' is a set with $|C'| = k$ and $x \notin C'$.

By the inductive hypothesis, the algorithm is correct for C'. Thus, when we apply the algorithm to C', it returns the subset of C' with the largest cardinality that has mutually disjoint elements. Let this subset be called L' and let its cardinality be c'. Note that L' may be the empty set.

Now, consider all possible subsets of C that contain x. There are 2^k such subsets. We can generate all possible subsets of C that include x using the *powerset* function.

For each subset T that we generate, we check if its elements are mutually disjoint using the *areElementsDisjoint* function. If the elements of T are not mutually disjoint, we can ignore it, since it cannot be a candidate for the largest cardinality subset with mutually disjoint elements.

If the elements of T are mutually disjoint, we then count the number of elements in T using the *sizeOfSet* function. If the cardinality of T is greater than c', then we set L'' to T and c'' to its cardinality.

After we have considered all subsets of C that contain x, we return L''. Note that L'' is a subset of C and has the largest cardinality among all subsets of C with mutually disjoint elements that include x. So in both cases we returned the mutually disjoint set with largest cardinality.

The algorithm terminates because it generates a finite number of subsets of C and each subset will be checked exactly once. Therefore, the algorithm eventually terminates.

Thus, the algorithm is correct for all input sets C with $|C| = k + 1$.

By mathematical induction, we have proven that the algorithm is correct for all possible input sets S.

Q.E.D.

As for the time complexity analysis, we will analyze the worst-case running time of our algorithm. The analysis will be based on the algorithm provided in 2.*a*. We will assume that $n = |C|$ and m is equal to the cardinality of the set with the most elements in C. The outer loop that runs through every subset of C will run 2^n times, since there are 2^n subsets of a set with n elements. Using our earlier assumption, we need to make m^2 comparisons at most to find out if there are any intersecting elements of two sets in a given subset of the powerset of C. Since we can have at most n elements in a subset of the powerset, we should make at most $\frac{n*(n-1)}{2}$ comparisons of subsets. Therefore, combining all of this, we will have a running time of $2^n \times \frac{n*(n-1)}{2} \times m^2$, which corresponds to a running time of $O(2^n * n^2 * m^2)$.

3.b. Heuristic Algorithm

For correctness; while it is challenging to prove the algorithm's correctness in all conditions due to its heuristic nature, we can demonstrate that the algorithm will consistently produce a feasible result. In this context, feasibility refers to the algorithm's ability to construct a set packing where no two sets in the subset have any elements in common. We can try to prove feasibility in the form of a theorem as follows:

Theorem: Our algorithm for maximum set packing returns a feasible solution, i.e., a set packing where no two sets in the subset have any elements in common.

Proof:

1. **Initialization:** At the beginning of the algorithm, `max_set_packing` is an empty set, and `elements` is also an empty set. Since the algorithm does not add any sets to `max_set_packing` until the condition `s` $\cap$ `elements` $= \emptyset$ is satisfied, initially, `max_set_packing` is a valid set packing as it contains no sets.

2. **Selection of Sets:** The algorithm then iterates over the sets in `sorted_sets`, which are sorted based on their length in ascending order. This suggests that in each iteration, the algorithm selects the smallest set that has not been intersected with `elements`.

3. **Updating elements and max_set_packing:** When a set `s` is selected, if `s` $\cap$ `elements` $= \emptyset$ (i.e., the intersection of `s` and `elements` is an empty set), it means that `s` does not share any elements with the sets already included in `max_set_packing`. Consequently, `s` can be added to `max_set_packing` without violating the set packing condition.

4. **Preservation of the Set Packing Property:** Since the algorithm only adds sets that do not share any elements with the previously selected sets, the resulting `max_set_packing` maintains the property that no two sets in the packing share any elements.

5. **Termination:** The algorithm terminates when all sets in `sorted_sets` have been considered. At this point, `max_set_packing` contains a set packing where no two sets share any elements.

Therefore, based on the theorem, our algorithm returns a valid set packing. In other words, we can assure that our algorithm would return a feasible solution.

As for the time complexity analysis, we will analyze the worst-case running time of our algorithm. The analysis will be based on the algorithm provided in 2.*b*. We will assume that $n = |C|$ and m is equal to the cardinality of the set with the most elements in C. Sorting the sets in place will take $O(n\ log\ n)$ time, using MergeSort. Then, we will analyze the rest of the function mathematically, for which we will consider a case where each set will have m elements and each set will be pairwise disjoint with all other sets. After making these assumptions, let us analyze the function further.

At each iteration, the union and append operations will take combined $O(m)$ time. If we can calculate the number of comparisons required for the intersection operation, we should be able to find a good upper bound for it. In the first iteration, we will have 0 comparisons, since *elements* is initially empty. In the second iteration, we will make m^2 comparisons to find an intersection in the worst-case, because each set will have m elements. In the next iteration, we will need $2m^2$ comparisons, due to all sets being pairwise disjoint (m elements were added to *elements*). In the last iteration, elements will have $(n-1) * m$ elements and the set will have m elements, requiring $(n-1) * m^2$ comparisons. Using the above information, we can find the number of comparisons using the below formula:

$$\sum_{i=1}^{n-1} m^2 * i$$

$$= m^2 * \sum_{i=1}^{n-1} i$$

$$= m^2 * \frac{n * (n-1)}{2}$$

Thus, we will make $O(n^2 * m^2)$ comparisons. Since the if statement will dominate the other statements, we did not take them into consideration for the above the equation. Combining the sorting step, the worst-case running time of our algorithm will be $O(n^2 * m^2) + O(nlog(n))$. In practice, since this is an upper bound, we might have a better running time.

4. Sample Generation (Random Instance Generator)

Below is the code for generating random samples. The parameters for this sample generation are n (the number of sets), m (maximum number of elements in a set), and $max\_element$ (the range of the universal set).

```
def random_sized_set_packing_instances(n, m, max_element):
  sets = []
  for i in range(n):
    set_size = random.randint(1, m)
    s = set(random.sample(range(1, max_element+1), set_size))
    sets.append(s)
  return sets
```

Listing 1: Code for Random Sample Generator

This function will be generating n sets. Each set's size will be selected randomly between 1 and m (both inclusive), which we can call k. Then, it will randomly sample k elements from the range $\{1, 2, \ldots, max\_element\}$. This set will be appended to our list of sets and in the end, the list of n sets will be returned as a random test instance. The following are some of the outputs with different parameters:

```
sets = random_sized_set_packing_instance(3, 8, 100)
for i in range(len(sets)):
  print(f"Set {i+1}:\t", sets[i])
```

```
Set 1:   {97, 3, 35, 11, 46, 28, 93}
Set 2:   {67, 79, 50, 55, 92, 95}
Set 3:   {88, 36}
```

(a) $n = 3$

```
sets = random_sized_set_packing_instance(5, 7, 25)
for i in range(len(sets)):
  print(f"Set {i+1}:\t", sets[i])
```

```
Set 1:   {7, 9, 21, 22, 24}
Set 2:   {10, 11, 4, 22}
Set 3:   {16, 25, 1}
Set 4:   {17}
Set 5:   {17, 12, 14}
```

(b) $n = 5$

```
sets = random_sized_set_packing_instance(10, 10, 50)
for i in range(len(sets)):
  print(f"Set {i+1}:\t", sets[i])
```

```
Set 1:   {32, 38, 44, 18, 19, 29}
Set 2:   {10, 44, 13, 46, 22, 27}
Set 3:   {1, 4, 39, 44, 48, 24, 26}
Set 4:   {3, 41, 10, 15, 50, 22, 25}
Set 5:   {33, 2, 4, 5, 48, 49, 27, 28}
Set 6:   {33, 2, 39, 9, 13, 50, 24, 27, 29, 30}
Set 7:   {24, 33, 35, 37}
Set 8:   {35, 5, 38, 39, 10, 44, 45, 48, 24, 25}
Set 9:   {3, 35, 6, 41, 11, 15, 22, 29, 31}
Set 10:  {1, 2, 35, 36, 38, 40, 13, 19, 26, 31}
```

(c) $n = 10$

```
sets = random_sized_set_packing_instance(15, 8, 250)
for i in range(len(sets)):
  print(f"Set {i+1}:\t", sets[i])
```

```
Set 1:   {225, 136, 201, 170, 236, 205, 214, 247}
Set 2:   {5, 134, 38, 203, 82, 29}
Set 3:   {99, 132, 165, 206, 175, 78}
Set 4:   {26}
Set 5:   {146, 141, 95}
Set 6:   {226, 35, 4, 198, 45, 145, 18}
Set 7:   {131, 165, 166, 198, 176, 181, 54}
Set 8:   {163, 4, 138, 22, 87, 56, 27}
Set 9:   {46, 79, 80, 241, 145, 174}
Set 10:  {84}
Set 11:  {71, 40, 167, 239, 121, 91}
Set 12:  {3, 196, 72, 239, 117, 182, 90}
Set 13:  {1, 69, 198, 72, 10, 236, 48, 211}
Set 14:  {106, 222}
Set 15:  {162, 66, 40, 201, 172, 76}
```

(d) $n = 15$

Figure 4: Random sample generation for different sizes

5. Algorithm Implementations

5.a. Brute-Force Algorithm

The algorithm was implemented by our team in Python as follows:

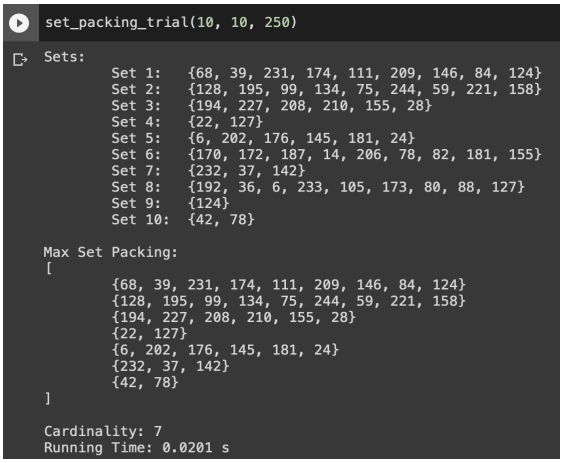
```
def max_set_packing(sets):
  n = len(sets)
  all_subsets = []

  for i in range(2**n):
    subset = []

    for j in range(n):
      if i & (1 << j):
        subset.append(sets[j])

    all_subsets.append(subset)

  maxCard = -1
  maxSetPacking = []

  for subset in all_subsets:
    isPairwiseDisjoint = True

    for i in range(len(subset)-1):
      for j in range(i+1, len(subset)):
        if subset[i] & subset[j] != set():
          isPairwiseDisjoint = False
          break

    if isPairwiseDisjoint and maxCard < len(subset):
      maxCard = len(subset)
      maxSetPacking = subset

  return maxSetPacking
```

Listing 2: Implementation of the Brute-Force Algorithm

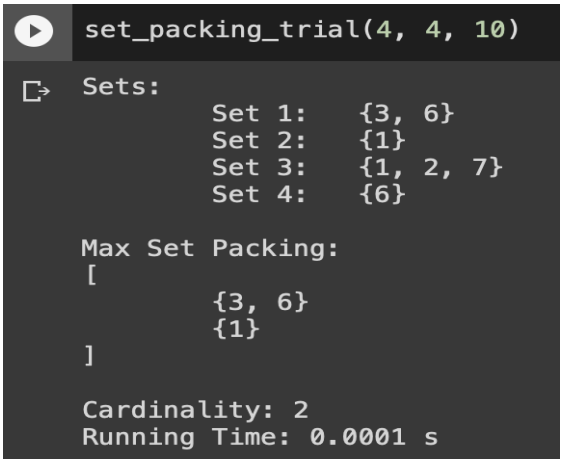
It starts by generating all subsets of the given sets. After that is done, it starts iterating over all of the subsets. If there is a subset with mutually disjoint elements, whose cardinality is bigger than the current maximum set packing, that subset is assigned to $maxSetPacking$ and $maxCard$ is updated. At last, the maximum set packing is returned.

The above algorithm does not work very well when the number of sets in C is increased. The threshold for a low running time is about 15 sets, after which the running time of the program increases exponentially and algorithm takes a long time to complete. However, if the maximum number of elements is increased, the algorithm does not slow down very much. From our initial tests, we concluded that it works correctly, in line with what we have proved in 3.*a*. Here are some results we obtained:

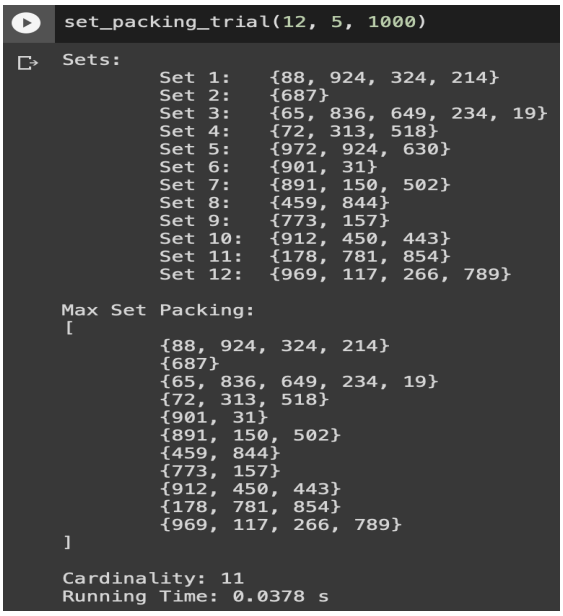
You can see some experiments at the next page:



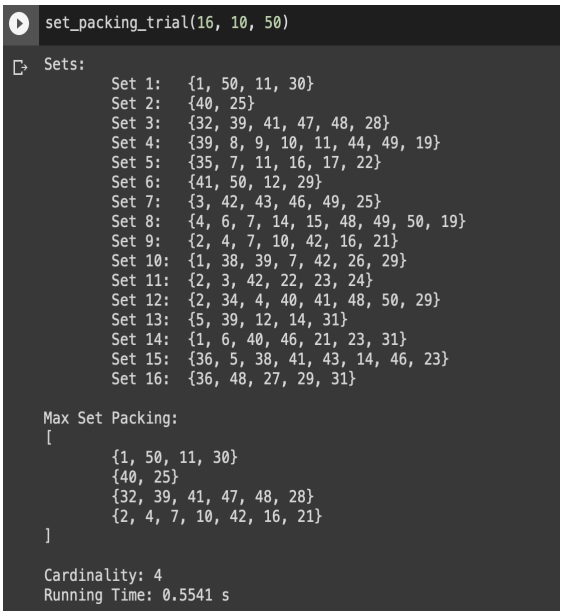
(a) 10 sets with maximum 10 elements, set element range : [0,250]



(b) 4 sets with maximum 4 elements, set element range : [0,10]



(c) 12 sets with maximum 5 elements, set element range : [0,1000]



(d) 16 sets with maximum 10 elements, set element range : [0,50]

Figure 5: Results of the trials - Brute-Force Algorithm

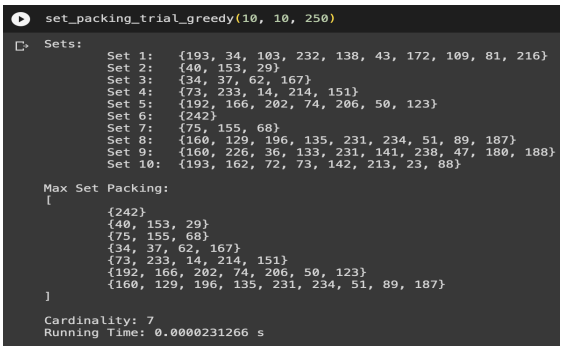
5.b. Heuristic Algorithm

The algorithm was implemented by our team in Python as follows:

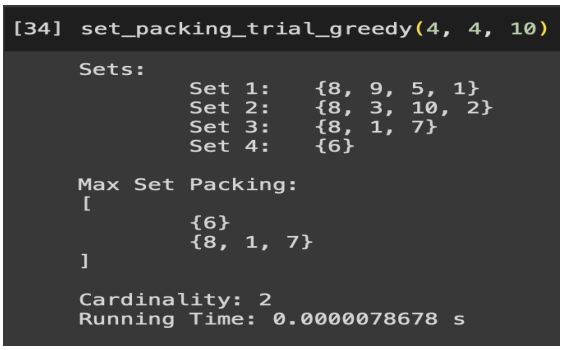
```
def greedy_set_packing(sets):
    max_set_packing = []
    elements = set()
    sorted_sets = sorted(sets, key=len)
    for s in sorted_sets:
        if s & elements == set():
            elements = s | elements
            max_set_packing.append(s)
    return max_set_packing
```

Listing 3: Implementation of the Heuristic Algorithm

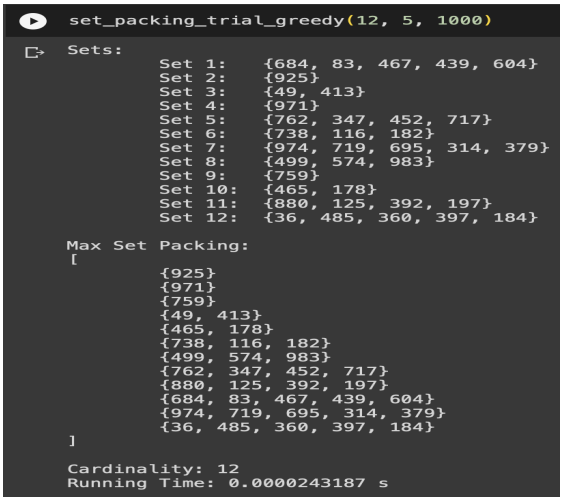
It starts by assigning empty list to *max_set_packing* and empty set to *elements*. Then, it sorts the sets based on their cardinalities (number of elements) in ascending order. Then, it iterates over the sorted sets. In each iteration, it checks whether it has an intersecting elements with *elements*. If not, elements becomes the union of elements and set and the set is appended to the *max_set_packing*. In the end, the *max_set_packing* is returned. We have tested the implementation for various input parameters, and it has proven to be quite efficient and often times correct. In the next page, you can find four experiments with the same values as in the Brute-Force algorithm tests.



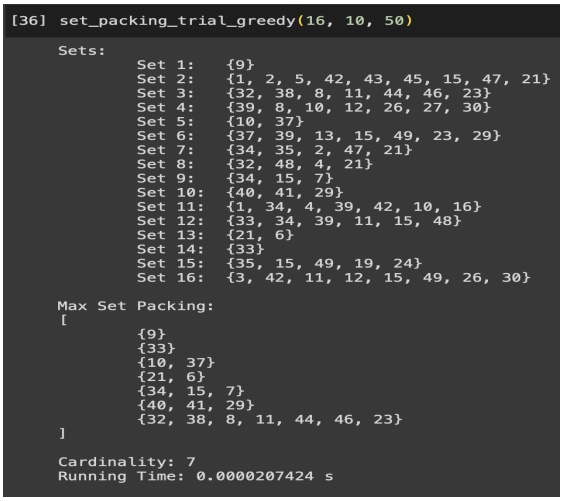
(a) 10 sets with maximum 10 elements, set element range : [0,250]



(b) 4 sets with maximum 4 elements, set element range : [0,10]



(c) 12 sets with maximum 5 elements, set element range : [0,1000]



(d) 16 sets with maximum 10 elements, set element range : [0,50]

Figure 6: Results of the trials - Greedy Algorithm

6. Experimental Analysis of the Performance (Performance Testing)

We conducted an experimental analysis of the algorithm's running time by calculating the mean, standard deviation, standard error and confidence intervals. We did the simulation for input sizes starting from 100 to 100000. For each input size, we collected 250 measurements. During the simulations, we kept the maximum cardinality of a set constant (50) and the universal set was $U = \{1, 2, \ldots, 50000\}$.

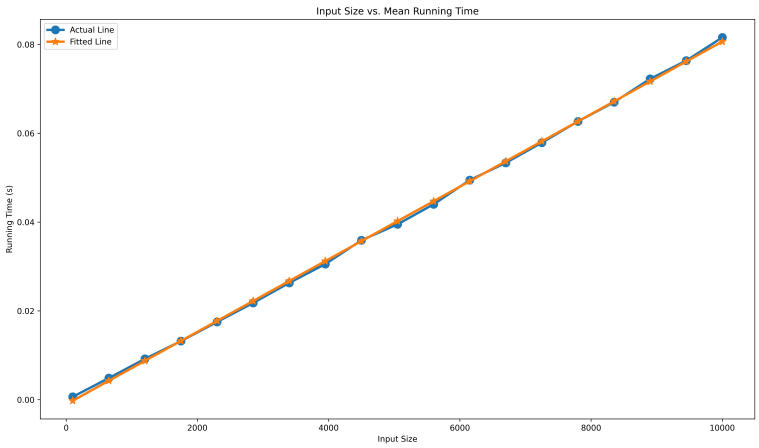


Figure 7: Running Time - Actual vs. Fitted

As can be seen from the chart, there is an almost linear relationship between running time and input size. The result when we express the working time based on the equation of the fitting line:

$$\text{Running Time} = -3.8788 \times 10^{-4} + (7.982302 \times 10^{-6}) \times \text{Input Size}$$

Then, we constructed the table for the confidence intervals for various input sizes.

Input Size	Mean	Standard Deviation	Standard Error	90% CI Lower	90% CI Upper	95% CI Lower	95% CI Upper
100.0	0.0006947613	0.0003463401	0.0000013854	0.0006923590	0.0006971636	0.0006918507	0.0006976718
650.0	0.0049222336	0.0003794343	0.0000015177	0.0049196017	0.0049248654	0.0049190449	0.0049254222
1200.0	0.0091406126	0.0005775650	0.0000023103	0.0091366065	0.0091446187	0.0091357589	0.0091454663
1750.0	0.0140684748	0.0011581704	0.0000046327	0.0140604414	0.0140765081	0.0140587419	0.0140782077
2300.0	0.0184364147	0.0012796951	0.0000051188	0.0184275384	0.0184452910	0.0184256606	0.0184471689
2850.0	0.0231839495	0.0015956501	0.0000063826	0.0231728816	0.0231950173	0.0231705401	0.0231973588
3400.0	0.0271567669	0.0015477646	0.0000061911	0.0271460312	0.0271675026	0.0271437600	0.0271697738
3950.0	0.0311297264	0.0014351136	0.0000057405	0.0311197721	0.0311396807	0.0311176662	0.0311417867
4500.0	0.0365077562	0.0020006269	0.0000080025	0.0364938794	0.0365216331	0.0364909436	0.0365245689
5050.0	0.0403698711	0.0016719524	0.0000066878	0.0403582741	0.0403814682	0.0403558206	0.0403839217
5600.0	0.0457285938	0.0021390107	0.0000085560	0.0457137571	0.0457434305	0.0457106182	0.0457465694
6150.0	0.0505291634	0.0021107191	0.0000084429	0.0505145229	0.0505438038	0.0505114255	0.0505469012
6700.0	0.0550840235	0.0020777725	0.0000083111	0.0550696115	0.0550984354	0.0550665625	0.0551014844
7250.0	0.0592643881	0.0020473590	0.0000081894	0.0592501871	0.0592785891	0.0592471827	0.0592815935
7800.0	0.0633891525	0.0019143355	0.0000076573	0.0633758742	0.0634024308	0.0633730650	0.0634052400
8350.0	0.0679168797	0.0020318773	0.0000081275	0.0679027860	0.0679309733	0.0678998044	0.0679339549
8900.0	0.0730666761	0.0021575592	0.0000086302	0.0730517108	0.0730816415	0.0730485447	0.0730848076
9450.0	0.0777253170	0.0024006762	0.0000096027	0.0777086653	0.0777419687	0.0777051425	0.0777454915
10000.0	0.0824054518	0.0025559344	0.0000102237	0.0823877232	0.0824231804	0.0823839725	0.0824269311

Figure 8: Statistical Performance Results of the Greedy Algorithm

In the table above, you can see the different statistical results of our running time experience. When we look at the results, we see a lot of similarities with the chart, the fact that there is an almost linear relationship between the input size and running time.

7. Experimental Analysis of the Quality

To experimentally analyze the quality, a quality measure can be employed in conjunction with the brute-force algorithm. The quality of the heuristic algorithm can be calculated by comparing its cardinality (number of elements) to that of the brute-force algorithm, using the following formula:

$$Quality = \frac{\text{Cardinality of Heuristic Algorithm}}{\text{Cardinality of Brute Force Algorithm}}$$

The quality measure provides an assessment of how well the heuristic algorithm performs in relation to the brute-force algorithm.

Input Size	Mean Quality
5	1.0000000000
6	0.9960000000
7	1.0000000000
8	0.9957142857
9	0.9960714286
10	0.9976388889
11	0.9948888889
12	0.9959318182
13	0.9936060606
14	0.9917191142
15	0.9921953047
16	0.9925091575
17	0.9951904762
18	0.9881251347
19	0.9910248599
20	0.9877097339

Figure 9: Quality Ratio of Heuristic and Brute-Force Algorithm for Various Input Sizes

As the working time increases exponentially due to certain limits of our brute-force algorithm, we could only test between 5 and 20 as input size. During the simulations, the maximum cardinality of a set was kept constant at 30, and the universal set was $U = \{1, 2, \ldots, 10000\}$. When we examine the results, we can see that the heuristic algorithm gives quite close results to the brute-force algorithm. For instance, even for $size = 20$, a quality value of 0.9877 (which is the lowest quality value in our tests) indicates that the algorithm provides results that are extremely close to the actual values. With increasing input sizes, we may see fluctuations, but those results are yet to be analyzed due to our computational limits.

8. Experimental Analysis of the Correctness (Functional Testing)

8.a. Black Box Testing

Black box testing for our algorithm would involve generating different input scenarios and verifying the function's output against expected results.

Test cases are the following:

(a) Empty Set

(b) Sets with no common elements

(c) Sets with overlapping elements

(d) Sets containing a subset of some other element

(e) Sets with identical elements

```
sets = []
result = greedy_set_packing(sets)
assert len(result[0]) == 0, "Test case 1 failed"
print("Test case 1 passed successfully!")
print("Set Packing:", result[0])
```

```
Test case 1 passed successfully!
Set Packing: []
```

(a)

```
sets = [
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
]
result = greedy_set_packing(sets)
assert len(result[0]) == 3, "Test case 2 failed"
print("Test case 2 passed successfully!")
print("Set Packing:", result[0])
```

```
Test case 2 passed successfully!
Set Packing: [{1, 2, 3}, {4, 5, 6}, {8, 9, 7}]
```

(b)

```
sets = [
    {1, 2, 3},
    {2, 3, 4},
    {3, 4, 5},
    {4, 5, 6}
]
result = greedy_set_packing(sets)
assert len(result[0]) == 2, "Test case 3 failed"
print("Test case 3 passed successfully!")
print("Set Packing:", result[0])
```

```
Test case 3 passed successfully!
Set Packing: [{1, 2, 3}, {4, 5, 6}]
```

(c)

```
sets = [
    {1, 2, 3},
    {2, 3},
    {3, 4},
    {4, 5, 6}
]
result = greedy_set_packing(sets)
assert len(result[0]) == 2, "Test case 4 failed"
print("Test case 4 passed successfully!")
print("Set Packing:", result[0])
```

```
Test case 4 passed successfully!
Set Packing: [{2, 3}, {4, 5, 6}]
```

(d)

```
sets = [
    {1, 2, 3},
    {2, 3, 4},
    {2, 3, 4},
    {3, 4, 5}
]
result = greedy_set_packing(sets)
assert len(result[0]) == 1, "Test case 5 failed"
print("Test case 5 passed successfully!")
print("Set Packing:", result[0])
```

```
Test case 5 passed successfully!
Set Packing: [{1, 2, 3}]
```

(e)

Figure 10: Comparison of different test cases

It can be observed that the algorithm gives correct results in different input scenarios given. Based on the test cases and the results obtained from the algorithm, we can conclude that the algorithm successfully handles various input scenarios and produces the expected outputs. The test cases cover a range of possibilities, including empty sets, sets with no common elements, sets with overlapping elements, sets containing a subset of another element, and sets with identical elements. In all these cases, the algorithm accurately identifies the common elements present in the input sets and provides the correct output. This suggests that the algorithm is robust and effective in identifying common elements in sets, regardless of the specific input configuration.

8.b. White-Box Testing

For White-Box Testing, we used Condition Coverage as the metric. To achieve 100% condition coverage, we devised the four test cases to tackle each of the below issues.

1. Sorted set is empty.
2. Sorted set is not empty.
   (a) The current set s intersects with elements, and the condition s & $elements == set()$ evaluates to false. In this case, the set s is not added to max_set_packing.
   (b) The current set s intersects with elements, and the condition s & $elements == set()$ evaluates to true. In this case, the set s is added to max_set_packing and its elements are added to *elements*.

```
test_set_1 = []
oracle_1 = []
test_set_2 = [{1, 2}, {3, 4}, {5, 6}]
oracle_2 = [{1, 2}, {3, 4}, {5, 6}]
test_set_3 = [{1, 2}, {2, 3}, {3, 4}]
oracle_3 = [{1, 2}, {3, 4}]
test_set_4 = [{1, 2}, {3, 4}, {4, 5}, {5, 6}]
oracle_4 = [{1, 2}, {3, 4}, {5, 6}]

tests = [[test_set_1, oracle_1], [test_set_2, oracle_2], [test_set_3, oracle_3], [test_set_4, oracle_4]]

for t in tests:
    max_pack = greedy_set_packing(t[0])
    print("Excepted Result:", t[1])
    print("Actual Result:  ", max_pack)
```

```
Excepted Result: []
Actual Result:   []
Excepted Result: [{1, 2}, {3, 4}, {5, 6}]
Actual Result:   [{1, 2}, {3, 4}, {5, 6}]
Excepted Result: [{1, 2}, {3, 4}]
Actual Result:   [{1, 2}, {3, 4}]
Excepted Result: [{1, 2}, {3, 4}, {5, 6}]
Actual Result:   [{1, 2}, {3, 4}, {5, 6}]
```

Figure 11: White Box Testing Results

As depicted in Figure 11, each test case is labeled with its corresponding condition coverage result. It can be seen that all the conditions were exercised and evaluated correctly during the testing process. This demonstrates that the algorithm's logic and decision-making paths were thoroughly tested and executed as expected. The successful completion of white-box testing, achieving 100% condition coverage, provides assurance that the algorithm's internal structure and logic have been comprehensively validated. This increases our confidence in the algorithm's correctness and reliability.

9. Discussion

We have seen that while the brute-force algorithm that we implemented gave correct maximum set packings for a limited input size, our greedy algorithm worked much faster, trading off correctness for lower running time. Thus, we were able to test it with larger input sizes. With functional testing approaches (black-box and white-box testing methods), we have seen that there were no defects in the greedy algorithm that we implemented, although additional testing may be required to uncover edge cases that we may have missed. The brute-force algorithm works poorly for input sizes larger than 16. Our heuristic algorithm worked faster than the worst-case running time we have presented in 3.*b*, which is what we have excepted in practice. In the future, with more computational power, we may also test the quality of the solution with larger input sizes.

10. References

1. Karp, R. M. (1972). Reducibility among combinatorial problems. In R. E. Miller & J. W. Thatcher (Eds.), *Complexity of Computer Computations: Proceedings of a Symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York* (pp. 85-103). Plenum Press.