An Inquiry of The Set Packing Problem

Kai Li*

E-mail: kaivictorlee@163.com

Abstract

Given a hyper-graph $H = (X, E)$, X is a set of elements, nodes or vertices, and E is the non-empty subsets of X. In Set Packing Problem's context, we want to find all the subsets in E where none of its elements intersect with each other. The problem in its general form is *NP-Complete*. In this paper, we provide a detailed proof to show its intractability. We would also study its variants and derivatives to observe their similarities and differences in their structural features. We would also provide a special-case solution for the Maximal Set Packing Problem when its subsets size is bounded and show that there is a polynomial time algorithm to solve it.

Introduction

Theoretical Computer Scientists have made tremendous efforts in the past decades attempting to solve computationally intractable problems such as Travelling Salesman Problem, independent set, Vertex Cover, and so on. Another human scientific revolution will occur had these problems solutions found in a reasonable amount of computational time. Unfortunately, Karp unveiled the intractability of a list of problems, such as Clique, 0-1 Integer Programming, and Satisfiability with at most 3 literals per clause [1]. Many other problems can be reduced from these three original problems to show their NP-completeness. In this paper, we mainly focus on surveying the Set Packing problems alongside its variants and

derivatives. First, we provide an intuitive definition and a standard definition in its Integer Linear Programming formulation. We then move on talking in-depth about its relationship with its variants and derivatives, whereby, we may gain a better understanding about the motivations of the problem. The Set Packing Problem is a well-known NP-complete problem. We will offer a detailed proof to explain its intractability. Many real-world applications are closely resembling the Set Packing Problem and make use its derivatives features to invent ingenious solutions. Finally, we will look at specific heuristics and special-case algorithm specially for the *Maximum Set Packing Problem* bounded by m.

The Set Packing Problem Description

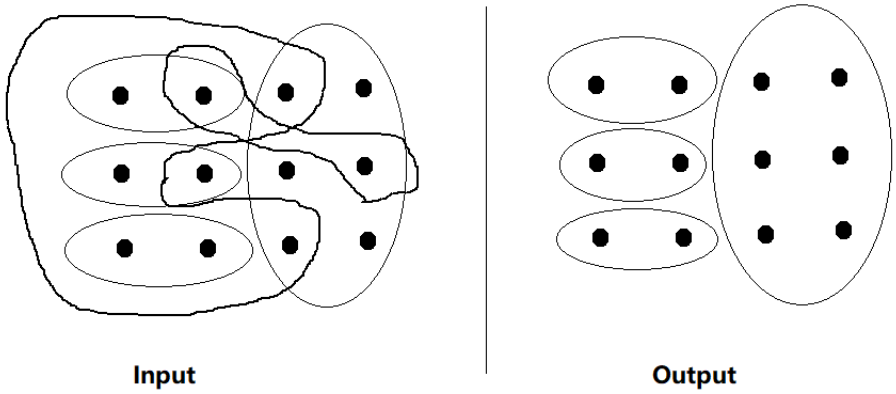
In simple words, given a set of subsets S and a number of elements that are used in those subsets, find k number of subsets in S where all subsets have non-overlapping elements. Another more formulaic definition is: given a set packing graph instance $G(V, E)$, where $V = \{S_1, S_2, ..., S_n\}$ and $E = \{(S_i, S_j) | S_i \cap S_j \neq \emptyset\}$. However, our goal is $S_i \cap S_j = \emptyset$ and tries to find the least number of subsets that cover all the elements with non-overlapping elements, see figure 1 for pictorial explanation. Moreover, let us consider the Set Packing Problem (SPP) in its Integer Programming formulation:

$$SPP(\mathbf{A}, \mathbf{v}) = \begin{cases} max \quad \mathbf{v}x \\ s.t. \quad \mathsf{A}x \leq \mathbf{e}, \\ x \in \{0, 1\}^m \end{cases} \tag{1}$$

where we have $x = \{x_1, x_2, ..., x_m\}$ is a vector of 0-1 decision variables with x_j meaning whether S_j is selected in the packing, $A \in \mathbb{R}^{n \times m}$ where A is a matrix with 0-1 values a_{ij} in row i and column j indicating whether subset S_j contains object i. The vector $v \in \mathbb{R}^m$ is a vector of payoffs, and $e \in \mathbb{R}^n$ is a vector of ones indicating all variables take on at most 1. The payoff is a convenient mathematical tool to reward the algorithm procedure

to make better decision to achieve the maximum values [2]. In other words, we would want more elements covered or included by the chosen subsets S_j to achieve our goal that is all elements covered using the non-overlapping subsets. We use payoffs to set up our SPP as an maximum Integer Linear Programming (MILP) problem.

Figure 1: A Set Packing Instance



Variants and Derivatives of The Set Packing Problem

Before we dive in the discussion of exploring the heuristics for greedy algorithm and approximation approximation for solving SPP, we should look at its two variants, the Set Cover Problem (SCP) and the Set Partition Problem (SPAP), both of which have proven of their NP-completeness by Richard M. Karp. By comparing the similarities and differences in their problem structure, we shall find inspirations as to how other researchers have exploited this feature to prove SPP's NP-completeness.

The first thing to note is the Set Covering problem is a minimization problem. The idea is that given a universe of elements $U = \{e_1, e_2, ..., e_n\}$ and a set collection of subsets $S = \{s_1, s_2, ..., s_m\}$, we need to find the smallest number of subsets which cover all the elements in U. One example application will be: a company tries to allocate time slots to schedule its workers' shifts during the 12 work hours; the manager uses a time slots generator to generate sets of random time slots and some overlap and some don't; so now the manager

needs to find the smallest number of randomly generated sets that cover all the time slots only once to meet the 12 hours work shifts. We can also model the set covering problem in its integer linear programming form:

$$SCP(\mathbf{A}, \mathbf{c}) = \begin{cases} min \quad \mathbf{c}x \\ s.t. \quad \mathsf{A}x \geq \mathbf{e}, \\ x \in \{0,1\}^m \end{cases} \tag{2}$$

A is a $m \times n$ matrix of zeros and ones, $e = (1, ..., 1)$ is a vector of m ones and c is a vector of n rational elements. It is quite obvious to see the similarities between SCP and SPP, the former is a minimization problem and the latter is a maximization problem with the inequality sign reversed for all elements in the coverage. By observing the SCP and its solutions that were already found by other researchers, we could potentially use this fact to perhaps transform our SPP problem into SCP or inspire us to find solution using similar approaches already used other researchers in SCP solutions.

The Set Partition Problem (SPAP) is another variant many other researchers tend to associate SPP and SCP to. The SPAP in itself has features that are similar to SCP. The SPAP is: given a set of S of numbers, we partition it into two sets A and $\overline{A} = S - A$ such that

$$\sum_{x \in A} x = \sum_{x \in \overline{A}} x$$

The SPAP's NP-completeness can be shown by reducing $Subset - Sum$ to it [4]. The only difference between the SCP and SPAP is that we can replace all the inequalities to equation signs to turn SCP or SPP into SPAP. In the case of SPP, we choose all the subsets such that their elements will not overlap with each other. We ended up with changing all the element variables from zeros to ones in SPP. On the other hand, the SPAP separates all elements into two sets, the set of elements we choose and the set of elements we do not choose; at the end, each of the two sets must complements one another. We again can tap into this similarity

in the problem structure and explore heuristics and ingenious algorithmic approaches other researchers have already found to help us solve the SPP. Since we have discussed the two variants of the SPP, the SCP and SPAP, we now discuss the Online Set Packing (OSP), a derivative of the SPP [5]. One element arrive at a time from some network, we need to find a way to assign each of them to a bounded number of sets before the arrival of next element. We receive payoff for filling up each set but none for the unfinished. Our goal is to maximize the number of completed sets.

$$\begin{aligned} \textbf{max} \quad & \sum_{i=1}^{m} w_i x_i \\ \text{subject to} \quad & \sum_{i:s_i \ni u_j} x_i \leq b_j, \qquad j = 1, ..., n \\ & x_i \in \{0, 1\}, \quad i = 1, ..., m \end{aligned} \tag{3}$$

From the integer programming formulation (3), we can see it is very much similar to SPP except the changes in constraints. The variable value x_i takes on binary value determining whether set S_i is completed or not. And w_i is the payoff received by completing set S_i and b_j is the number of sets element u_j can be assigned to. Several differences between OSP and SPP that are worth noting are element needs to arrive one at a time, the variable x_i can go from one to zero indicating a rejected set that can not be taken later (a set that has repeated elements in the previous selected sets), and the target function at a current state is taken only over variables that will not be constrained anymore.

Another derivative of SPP worth noting is the maximum k-Set Packing problem (k-SP). As previously mentioned SPP in Integer Programming formulation, the generic Maximum Set Packing Problem (MSPP) is, given a collection of sets $C = \{S_1, ..., S_m\}$ over a certain domain of elements $D = \{x_1, ..., x_n\}$, the goal is to find a maximum packing, or a maximum number of pairwise non-intersecting sets from the collection of sets C. Just as Figure 1 illustrated, elements or vertices x_i are encircled by contour lines, or hyper-edge in graph theory, a edge that connects k vertices where $k \geq 2$. The k-SP problem simply has its hyper-edge bounded

by certain number k of vertices. What we are interested is to find the maximum number of subsets S_i where all are non-intersecting or independent from each other. Hyper-edges that are disjoint or non-overlapping are called matching. It is also not difficult to perceive the close connection between MSPP and *Independent Set* where each hyper-edge in MSPP is independent or non-intersecting from each other. In this sense, we can view MSPP as the problem of Maximum Independent Set Problem (MISP) in hyper-graphs of degree at most k [6].

Figure 2: An IS Instance

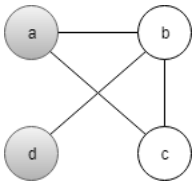
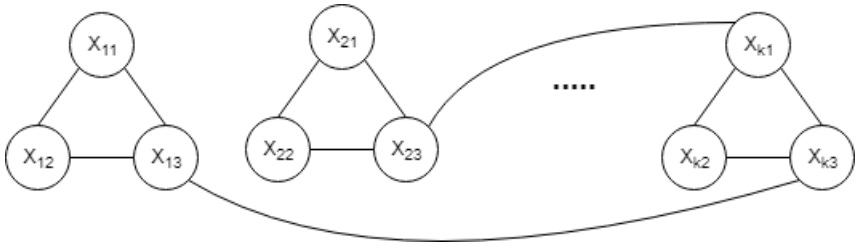


Figure 3: A 3-SAT Construction



The Set Packing Problem's NP-completeness

We want to show the Set Packing Problem (SPP) is in *NP-Complete*. To prove this is true, we need to show SPP is in *NP* then reduce a known NP-Complete problem to it. Here, we will attempt to use Independent-Set, but we first also need to show its NP-Completeness. The Satisfiability problem, particularly the one that has three variables in each of its clauses, is one of Karp's 21 NP-complete problems [1]. We want to show that Independent Set is in NP-Complete by reducing the Satisfiability with 3 variables clauses to it. This proof is shown as follows:

Proof. We first claim Independent Set (IS) is in NP. Given any graph $G(V, E)$, we have to show there is polynomial time verifier that could confirm the set S we selected has size of k and its elements or nodes are independent (or not incident) of each other. For instance, Figure 2 is an IS instance where the IS set $S = \{a, d\}$. Clearly, it takes poly-time to verify each one of element in S to see check if it does not violate any constraint in the IS problem. In the case of IS instance shown in Figure 2, $k = 2$ or the size of the IS instance. Hence, we have shown IS is in NP. Next, we need to show that IS is *NP-Hard* by reducing *3-SAT* to it.

As per Figure 3, we have a construction of clause gadgets, each of which consists of 3 nodes, and k of these clause gadgets. In formulation, we have a boolean-value conjunctive normal formulation $F = C_1 \wedge C_2 \wedge ... \wedge C_m$ and each clause C_i consists of boolean-value variables from $X_1, X_2, ..., X_n$ and their negations $\overline{X_1}, \overline{X_2}, ..., \overline{X_n}$.

Since we have transformed *3-SAT* expression construction into its graph form, we now need a black-box algorithm to find all the nodes to build an Independent Set. First, we assume F has a satisfiable assignment A. That said, we know at least one of the variable in each clause in F is true according to assignment A. Let us then define S to be the set of nodes selected one of the satisfiable nodes in each clause gadget. We have selected m of these nodes since we pick one from each clause gadget. Note that we pick our nodes in the graph G based on our assignment A and those edges $e_i \in E$ connects both X_i and $\overline{X_j}$ between two clauses. So A cannot possibly satisfy both X_i and $\overline{X_j}$. That said, we know the nodes we selected will not violate the IS rule, i.e., no two nodes will be incident to each other by an edge. We then confirm the set S our algorithm finds is an Independent Set.

Next, we need to verify the correctness of our algorithm or that it finds a correct set of nodes that are independent. We know that our algorithm finds a set S of size M in a graph. Assume that we have a set S of independent elements and each of these elements or variables is picked from one clause gadgets. Since we also happen to have m of these clause gadgets and we mentioned that we will only pick one of these variables in each clause that satisfy the assignment, we can assure that we have an independent set because of the

elements we chose. On the contrary, assume we given a satisfiable assignment A with the *3-SAT* formulation expression, we can again pick our variables nodes in each clause in our graph construction. Clearly, we would pick one of the variable nodes in each clause where $A(X_i) = true$. Doing this can also help us avoid violating the independent set rule. Hence, we will obtain an IS from the satisfiable assignment in the provided graph $G(V, E)$.

Since we have proved both IS in NP and it is in NP-Hard by $3 - SAT \leq_p IS$, we have successfully shown *Independent Set* is in *NP-Complete*.

■

It is important to spend the time to verify IS is in *NP-Complete*, so that we can use this fact to show $IS \leq_p SPP$ and SPP is in *NP-Complete*. The proof for showing SPP is in *NP-Complete* is shown as follows:

Proof. Suppose we are given a certificate $C = \{S_1, S_2, ..., S_k\}$ with size of k subsets in it, we need to verify that each pair of sets S_i, S_j, $i, j \leq k$ and that $S_i \cap S_j = \emptyset$. If we check all elements of in a subset of S_i and S_j at a time, it takes $O(k^2 m)$, m is the maximum size of each subset S_i or S_j. That shows the Set Packing Problem (SPP) is in *NP*. Next, we need to show SPP is *NP-Hard* by showing *Independent Set (IS)* $\leq_p SPP$ (we proved IS is in NP-Complete already).

Given an instance graph $G(V, E)$ of IS bounded by target value k, we can construct a collection C of k' subsets where $C = \{S_1, S_2, ..., S'_k\}, k' = k$. As per the definition of Independent Set, we know each vertex associates with a number of edges, and those any pair vertices incident to an edge will not be picked at a same time in Independent Set construction. Any vertex share an edge with another, in the context of SPP, is considered as an intersection or overlapping vertices in hyper-edge in the SPP's hyper-graph. In this case, we will construct a hyper-graph (like Figure 1: input graph) from the given instance graph given by IS such that each vertex $v_i \in S_i$ where vertex v_i only exists in hyper-edge e when we selects it from IS's graph G. And the size of the collection of subsets C is $k', k' = k$. We

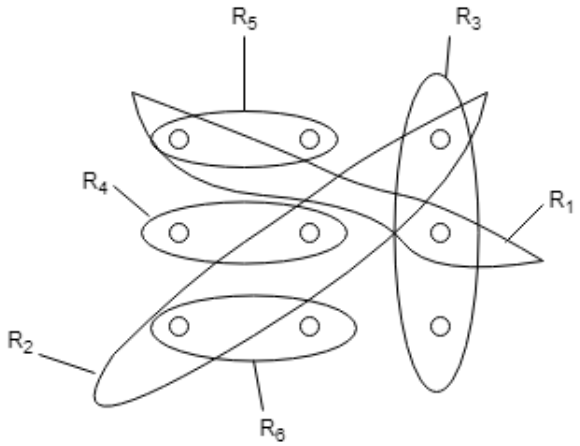
would have to relabel some vertices from G and group them into hyper-edges in SPP. This transformation can be completed in polynomial time.

We then need to prove the correctness of our algorithmic transformation from IS to SPP. Suppose we have graph instance G that has independent set (named, I) of size k. We can build a collection of subsets C with subsets S_i and each variable $v_i \in I$. $Size(I) = size(C)$ and, furthermore, no pair S_i and S_j share the same hyper-edge e' or $S_i \cap S_j = \emptyset$. We then have built a working instance of SPP with size k.

On the contrary, suppose we are given an instance of SPP with size k', we need to build an Independent Set with size k from that instance where $k' = k$. First, we need to build a set I where each vertex $v_i \in C$ in I. And $size(I) = size(C)$. We in turn select v_i from each s_i and make sure no v_i share the same e' with another which can also avoid violating the rule of independent set. We then can confirm that I is an yes instance of independent set with size k. This proves the correctness of our algorithmic transformation for $IS \leq_p SPP$.

Since we have shown SPP is in NP and $IS \leq_p SPP$, we have proved that SPP is in *NP-Complete*. ■

Figure 4: A SPP instance in Train Scheduling Context



The Set Packing Problem's Applicability

According to Tri-Dung, the Complete Set Packing Problem (CSP), a derivative of SPP, is often applicable for "routing and scheduling trains at intersection in railway operations", "for selecting winning bids in combinatorial auctions", and "for surgical operational scheduling" [2]. In the case of Train Scheduling, we have a number of hyper-edges (an edge that connect more than two nodes) in the instance hyper-graph shown in Figure 4. Each hyper-edge $R_i, 1 \leq i \leq 5$, represents a passable train route. It is the responsibility of trains control center to set up a system which coordinates all the train routes that run in different scheduling hours, often this system can be implemented using SPP. For instance, an instance of SPP can be $C = R_3, R_5, R_4, R_6$ where none of these hyper-edge or route has overlapping points—i.e., no trains run on the same time in two different routes. However, if one were to pick R_1, R_2, and R_3, the trains either never move or clash into one another (note that one train can not move in two different directions at a same time). We thus see how useful it is to have SPP incorporated into the Routing Train Scheduling System to help coordinate the different timing in running hours. Similarly, SPP can apply to final exams location scheduling, surgical operational scheduling as mentioned early, and other scheduling events that share similar traits.

A Special Case for the Set Packing Problem Solution

We know SPP is NP-Complete and it is quite difficult to solve when the size of SPP exceed $k = 10$ where k is the number of subsets in the collection of subsets C of SPP. Despite its difficulty of solving in time-efficient manner, we would consider a naive approach regardless, i.e., looking at all subsets-pair of k in C if we are dealing with k-*SPP* or k number of subsets which are disjoint in C of SPP. Note that this naive approach for general SPP would amount to time $O(n^k N)$ where N is the total number of items in the universal set U and k is the number of disjoint subsets [7]. Knowing the naive approach is impractical in any real-world application, we should opt for other alternatives. There are usually three common ways

to go about such a dilemma: first is to just accept the exponential function when input size is tolerably small, second is to find special cases of the problem where it could solve in polynomial time, and third is to find an approximated algorithm to allow some "error" rate that gives us an answer close enough to the optimal.

In this paper, we would like to explore into the special case where the size of subsets in C is bounded by m, or $k < m$, that should suffice to solve our particular situation. We assume that m is the size of each subsets in C. For instance, when we only concern about 50 disjoint subsets, $m = 50$, the problem can become more tractable versus its general case, e.g., $m > 50$. W. Jia et al. proposes an algorithm that will run in time $O(mn)$ for a *k-maximal Set Packing* M_0 in C where m is the size of C and n is the size of input elements in the universe. Evidently, this algorithm of *k-maximal* for *m-Set Packing* is substantially more efficient in time than the naive version which runs in $O(n^k N)$. The *k-maximal* algorithm for *m-Set Packing* is shown as follows [7]:

Algorithm 1 k-maximal for m-SPP

```
Require: An (C, m, k) instance for m-SPP
  M_0 = ∅
  while C ≠ ∅ do
    ρ = C.next
    if no e ∈ ρ is labeled used then
      append ρ to M_0
      label all e ∈ ρ as used
      if |M_0| == k then
        return M_0
        break
      end if
    end if
  end while
  return M_0
```

Algorithm 1 is quite similar to the naive version except that there are some bookkeeping and constraints such as $k \leq m$ that k is bounded by some number. Note that we have already given the collection of subsets C and the bound m so we know when the size of *m-SPP*, $M_0 == k$, we should stop the *while-loop* and return the concatenated *m-SPP* M_0

which is also the *k-maximal Set Packing* in our case. In addition, we have the bound m, the number of disjoint subsets we want, and the number of elements in C is n, we therefore have the time $O(mn)$.

Another greedy-approach W. jia et al. proposes is quite useful by applying the idea of partial sets σ^*. At first, we should define a few terms before delving into the meat of the algorithm. A partial set σ^* is a set σ in collection C with 0+ elements in σ replaced by the symbol $*$. A set in C without $*$ is called a regular set, and a regular set is also a partial set since it has 0+ symbol $*$. The set that has only non-$*$ elements in a partial set σ^* is called $reg(\sigma^*)$. Finally, a partial set σ^* is consistent with a regular set σ if and only if $reg(\sigma^*) \subseteq \sigma$. The algorithm of Greedy-replacing approach that yields both a m-SPP and a partial set packing is shown as follows [7]: Algorithm 2 uses a partial set packing to build a *k-maximal*

Algorithm 2 Greedy-Replacing (partial set packing to regular set packing)

Require: a (C, m, k) instance for partial set packing P_k
  $Q_k = P_k$
  while $C \neq \emptyset$ **do**
    $\sigma = C.next$
    if $\exists$(a unique partial set $\sigma^* \in Q_k$) s.t. $reg(\sigma^*) \subseteq \sigma$ and $\sigma_i \cap \sigma_j = \emptyset$ except $\sigma^* \in Q_k$ **then**
      $Q_k = (Q_k - \{\sigma^*\}) \cup \{\sigma\}$
    end if
  end while
  $M = collection(reg(\sigma^*) \subseteq Q_k)$
  return M and Q_k

set packing by replacing all the sets σ^* with σ whenever there is a unique partial set σ^* in the partial set Q_k such that it is regular, or has no symbol $*$ elements, and it does not intersect with any other set. We do this in procedural until the collection list is exhausted. At the end, we assign the collection of regular sets $reg(\sigma^*)$, the set that consists of non-"*" elements, to the set packing M. Note that the partial set σ^* is just any set that has some elements (zero or more) that have been replaced by the symbol $*$. W. jia et al. concluded two important facts from this greedy-replacing approach: $|M| = k$ and $|M| < k$ and for every set $\sigma \in C - M$, there is at least one set $\sigma' \in M$ that $\sigma \cap \sigma' \neq \emptyset$. The second fact is

stating that there is at least one pair of sets, one comes from the sets $C - M$ and another from M, that the two sets intersect one another when $|M| < k$. More importantly, the time complexity of the Algorithm 2 is bounded by $O(g(k, m)n)$. In the Algorithm 1, $g(k, m) = m$. "The function $g(k, m)$ depends on the parameter k and the $size(\sigma)$ m but is independent of the number n of sets in the given collection c [7]".

Summary

In short, we have provided a detailed problem description for the Set Packing Problem (SPP) in the beginning of this manuscript. We discuss its variants and derivatives about their similarities and nuances in features. We have proven is indeed a NP-complete problem to show its intractability in general form by reducing Independent set to it. In addition, we briefly discuss a few possible applications of SPP out of its many uses in different fields of its real-world applications. At the end, we discuss the heuristics and special cases that will help us solve the derivative of SPP, m-SPP with in polynomial time.

References

(1) Richard M. Karp, *Reducibility Among Combinatorial Problems*. University of California at Berkeley, California, 86-103, 1972.

(2) Tri-Dung Nguyen, *A fast approximation algorithm for solving the complete set packing problem*, European Journal of Operational Research, Volume = 237, issue = 1, pages = 62-70, date = 16 August 2014, url = `https://ac.els-cdn.com/S0377221714000459/1-s2.0-S0377221714000459-main.pdf?_tid=68e85932-b84a-4798-a072-699ceca299b7&acdnat=1521508508_31ba525403e73be580cdff3a18ff98ab`

(3) Karla Hoffman, Manfred Padberg, *Set Covering, Packing, and Partitioning Prob-*

lems, Springer, 3482-3485, url =https://link.springer.com/content/pdf/10.1007%2F978-0-387-74759-0_599.pdf.

(4) Marvin Nakayama, Foundations of Computer Science II, url =https://web.njit.edu/~marvin/cs341/hw/hwsoln13.pdf.

(5) Yuval Emek, Magnus M. Halldorsson, Yishay Mansour, Boaz Patt-Shamir, Jaikumar Radhakrishnan, Dror Rawitz, *Online Set Packing*, 2010 Semantics Scholar, url =https://pdfs.semanticscholar.org/f23c/9783724b575cf1f178644c8f7b6e04ff376e.pdf.

(6) Elad Hazan, Shmuel Safra, Oded Schwartz, *On The Complexity of Approximating k-set Packing*, springer, 2006, 20-29, url =https://link.springer.com/content/pdf/10.1007%2Fs00037-006-0205-6.pdf.

(7) Weijia, Jia, Chuanlin Zhang, and Jianer Chen, *An efficient parameterized algorithm for m-set packing, elsevier*, Journal of Algorithms, 2004, 106-117, url=https://pdfs.semanticscholar.org/b993/c7d6436c95a631da382166ee6dfd8779d197.pdf.