An approximate dynamic programming approach for improving accuracy of lossy data compression by Bloom filters

An Approximate Dynamic Programming Approach for Improving Accuracy of Lossy Data Compression by Bloom Filters

Xinan Yang*, Alexei Vernitski†, Laura Carrea‡

January 18, 2016

Abstract

Bloom filters are a data structure for storing data in a compressed form. They offer excellent space and time efficiency at the cost of some loss of accuracy (so-called lossy compression). This work presents a yes-no Bloom filter, which as a data structure consisting of two parts: the yes-filter which is a standard Bloom filter and the no-filter which is another Bloom filter whose purpose is to represent those objects that were recognised incorrectly by the yes-filter (that is, to recognise the false positives of the yes-filter). By querying the no-filter after an object has been recognised by the yes-filter, we get a chance of rejecting it, which improves the accuracy of data recognition in comparison with the standard Bloom filter of the same total length. A further increase in accuracy is possible if one chooses objects to include in the no-filter so that the no-filter recognises as many as possible false positives but no true positives, thus producing the most accurate yes-no Bloom filter among all yes-no Bloom filters. This paper studies how optimization techniques can be used to maximize the number of false positives recognised by the no-filter, with the constraint being that it should recognise no true positives. To achieve this aim, an Integer Linear Program (ILP) is proposed for the optimal selection of false positives. In practice the problem size is normally large leading to intractable optimal solution. Considering the similarity of the ILP with the Multidimensional Knapsack Problem, an Approximate Dynamic Programming (ADP) model is developed making use of a reduced ILP for the value function approximation. Numerical results show the ADP model works best comparing with a number of heuristics as well as the CPLEX built-in solver (B&B), and this is what can be recommended for use in yes-no Bloom filters. In a wider context of the study of lossy compression algorithms, our research

*Department of Mathematical Sciences, University of Essex, Wivenhoe Park, Colchester CO4 3SQ, UK. Corresponding author: xyangk@essex.ac.uk, 01206 872787

†Department of Mathematical Sciences, University of Essex, Wivenhoe Park, Colchester CO4 3SQ, UK.

‡Department of Meteorology, University of Reading, Reading RG6 6BB, UK.

is an example showing how the arsenal of optimization methods can be applied to improving the accuracy of compressed data.

Keywords: Lossy compression, Bloom filter, Integer linear program, Approximate dynamic programming, Heuristics

1 Introduction

In information technology, lossy compression is a data compression method that reduces the size of the representation at the cost of the loss of some accuracy at the decompression time. In exchange for losing accuracy in representation, lossy data structures not only store all information in constant space but also respond to membership queries in constant time. Examples of lossy data structures include skip lists [24], lossy dictionaries [22] and several hashing techniques.

1.1 Bloom filter

The Bloom filter is one of lossy methods of storing compressed data, introduced in [3]. The kind of data that Bloom filter is especially suitable for are sets. Given a set, a Bloom filter can be produced which represents the set in a compressed form. It can then be queried in the sense that there is an algorithm which, given an object and a Bloom filter representing a set, decides whether the object is or is not an element of the set. The querying algorithm is very efficient and works extremely fast compared to standard algorithms of accessing compressed data (one of the reasons why this algorithm is fast is that it contains many operations which are performed in parallel and that it is easy to implement in hardware) [25]. The size of the Bloom filter can be very small compared to standard ways of compressing data, which is a major advantage of Bloom filters. Nevertheless, there is also an important disadvantage: Bloom filters only represent data approximately, and frequently the querying algorithm gives an incorrect answer to the question about the membership of an object in the set represented by a Bloom filter.

The broad area of applicability of Bloom filters, due to their excellent space and time efficiency, is either in low-performance hardware or for tasks which must be performed extremely fast and speed is slightly more important than accuracy. Bloom filters have a range of uses in information technology [26] [4], from hardware implementations to software applications domain, where it was first conceived to perform space and time efficient dictionary lookups [3]. Broder and Mitzenmacher [4] have coined the Bloom filter principle: 'Whenever a list or set is used, and space is at a premium, consider using a Bloom filter if the effect of false positives can be mitigated'. To give just one example, Bloom filters can be used for routing in computer networks: in this application, a path which a message must follow is represented by a Bloom filter, namely, as a union of those links between computers which together constitute the path. It is appropriate to use Bloom filters in this application because each computer along the path must decide where to forward the message very quickly (literally with the speed of light, assuming that the links between computers are optical cables).

Calculating a Bloom filter of one object is a preliminary stage before building or querying a Bloom filter representing a set of objects. Assume that there is an algorithm which takes any object as its input and produces a binary array of a fixed length G, in which H bits are equal to 1 and other bits are equal to 0. We refer to this array as the Bloom filter of the object. The purpose of the Bloom filters of objects is to serve as uniform labels for each object which may interest us. Informally speaking, the Bloom filters of objects may be likened to bar-codes glued to every object. We denote the Bloom filter of an object s by $\eta(s)$.

Given a set $\mathcal{S}$, the Bloom filter of $\mathcal{S}$ can be computed as the bitwise disjunction of the Bloom filters of the elements in $\mathcal{S}$; in other words, the Bloom filter of the set $\mathcal{S}$ is defined as a binary array of length G, and for each $j = 1, \ldots, G$ the j-th bit is calculated as follows: if in every $\eta(s)$, where $s \in \mathcal{S}$, the j-th bit is 0 then the j-th bit of the Bloom filter is 0; otherwise, if in at least one $\eta(s)$ the j-th bit is 1 then the j-th bit of the Bloom filter is 1. We denote the Bloom filter of a set $\mathcal{S}$ by $\beta(\mathcal{S})$.

Given an object s and a Bloom filter $\beta(\mathcal{S})$, querying it to determine whether s is an element of $\mathcal{S}$ is done as follows. For each $j = 1, \ldots, G$ if the j-th bit of $\eta(s)$ is less than or equal to the j-th bit of $\beta(\mathcal{S})$ then we say that s is recognised[1] by the Bloom filter $\beta(\mathcal{S})$ as belonging to the set $\mathcal{S}$. In an ideal world, one would like to be able to claim that s is recognised as belonging to $\mathcal{S}$ if and only if s is an element of $\mathcal{S}$. However, this is not so. Due to the definition of a Bloom filter, if s is an element of $\mathcal{S}$ then s is recognised by $\beta(\mathcal{S})$; but the converse is not true: not necessarily an object s recognised by $\beta(\mathcal{S})$ is an element of $\mathcal{S}$. This kind of error is called false positives, in the sense that the Bloom filter query recognises the element as belonging to the set, but should not do it.

A number of approaches have been proposed to reduce the number of false positives in Bloom filters. The number H of positions equal to 1 can be varied [16]. Generalisations of the standard Bloom filter have also been considered, such as the yes-no Bloom filter [28] that is further studied in this paper, the retouched Bloom filter [9], the counting Bloom filter [13], the power of two choices [18], the optihash [6] or partitioned hashing [14]. Both the standard Bloom filter and its generalisations listed above can work in the use scenario in which some false positives and some false negatives are allowed. Nevertheless, in this paper we concentrate on the construction of yes-no Bloom filter under the standard use scenario in which we allow some false positives (trying to minimise their number) and do not allow any false negatives (for example, in the application to routing this approach means that the message will definitely be delivered to the right recipient but perhaps also sent to some other computers, thus creating some unnecessary traffic in the network).

1.2 Yes-no Bloom filter

This paper studies a yes-no Bloom filter [28], which is our new generalisation of the standard Bloom filter which actively reduces the number of false positives at the stage

[1] It is also convenient to use the pure-mathematics term 'covered', that is, s is covered by $\beta(\mathcal{S})$, thus stressing that $\beta(\mathcal{S})$ is a lattice join (or, in another interpretation, a set union) of the Bloom filters of the elements of $\mathcal{S}$.

of building the Bloom filter. Let us start with a fictitious and simplified but realistic use scenario. Suppose the management of an airport installs CCTV cameras whose output is automatically compared with photographs of 100 known terrorist suspects. Suppose these photographs are stored in a compressed form as a Bloom filter. Due to Bloom filters' efficiency, even low-performance hardware can effectively compare faces of people in the airport with the faces of the suspects. Then, if the Bloom filter recognises a face, the security staff is called to look at the person and make a decision. As we have discussed, Bloom filters produce false positives; therefore, the security staff will be called more often than needed. In particular, suppose that out of all the employees working in the airport, 300 people have faces that trigger false positives. We can considerably reduce unnecessary checks and nuisance if we actively indicate to the Bloom filter that these specific objects are recognised incorrectly and should not be recognised.

A yes-no Bloom filter consists of two Bloom filters, one called a yes-filter and the other called a no-filter[2].

Now we shall define the algorithms for building the yes-no Bloom filter of a set and for querying a yes-no Bloom filter. We assume that each object has two Bloom filters corresponding to it: the *yes-filter* of length G^+ and the *no-filter* of length G^-. Given an element s, we shall denote its yes-filter by $\eta^+(s)$ and its no-filter by $\eta^-(s)$.

Consider a set $\mathcal{S}$ and another set $\mathcal{T}$ such that that the two sets do not overlap. The set $\mathcal{S}$ is the one that we want to store in a compressed form, and the set $\mathcal{T}$ is a set whose elements are likely to be queried but should not be recognised as elements of $\mathcal{S}$. (In the example above, $\mathcal{S}$ is the set of suspects and $\mathcal{T}$ is the set of airport employees.) Build the yes-filter $\beta^+(\mathcal{S})$ of $\mathcal{S}$ as the bitwise disjunction of all $\eta^+(s)$, where $s \in \mathcal{S}$; in other words, $\beta^+(\mathcal{S})$ is the standard Bloom filter built from all arrays $\eta^+(s)$. The Bloom filter $\beta^+(\mathcal{S})$ will have false positives, including some which are contained in $\mathcal{T}$; let us denote the subset of $\mathcal{T}$ consisting of false positives of $\beta^+(\mathcal{S})$ by $\mathcal{F}$. (In the example above, $\mathcal{F}$ is the set of those employees whose faces are recognised by the Bloom filter.) Then the second, more interesting stage begins: we build the no-filter $\beta^-(\mathcal{S})$ of $\mathcal{S}$ so that $\beta^-(\mathcal{S})$ recognises as many as possible elements of $\mathcal{F}$ but none of the elements of $\mathcal{S}$. (As we shall see later in the paper, unlike building a standard Bloom filter or the yes-filter, this stage involves flexibility as to which elements of $\mathcal{F}$ should be included in $\beta^-(\mathcal{S})$, and, therefore, turns into a meaningful optimization problem.)

Figure 1 shows diagrammatically how the yes-filter and the no-filter are built. The yes-filter recognises all elements of $\mathcal{S}$ (the wave-patterned set) and also many other objects, probably including some elements of $\mathcal{T}$ (the chequer-patterned set). In an ideal scenario (shown on the diagram) we manage to build the no-filter in such a way that it recognises all elements of $\mathcal{T}$ recognised by the yes-filter but does not recognise any elements of $\mathcal{S}$.

Then when we query a yes-no Bloom filter $\beta^+(\mathcal{S}), \beta^-(\mathcal{S})$, given an object s we say that s is recognised by $\beta^+(\mathcal{S}), \beta^-(\mathcal{S})$ if $\eta^+(s)$ is recognised by $\beta^+(\mathcal{S})$ but $\eta^-(s)$

[2]Slightly more general approaches are also possible, which involve more than two Bloom filters, but this particular choice in the design of the structure is considered for the purposes of this paper. See the conclusion for suggestions of further research.

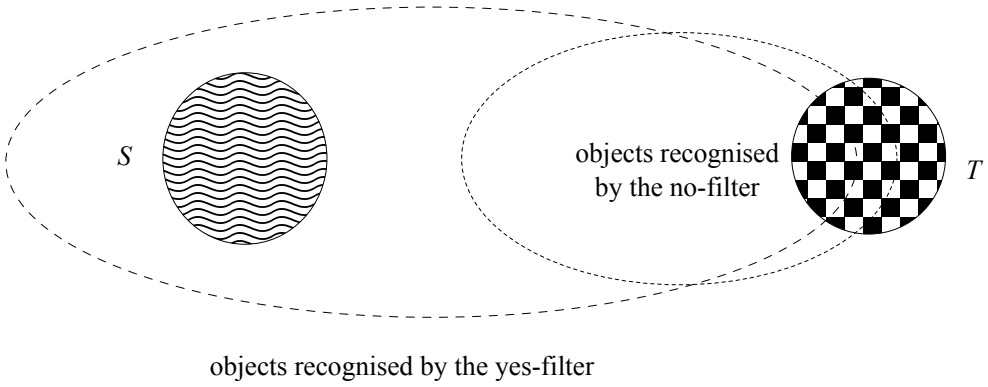

Figure 1: Yes-no Bloom filter.

is not recognised by $\beta^-(\mathcal{S})$. As you can see, the querying algorithm is as fast and easy as the one of the standard Bloom filter.

We have conducted extensive computational experiments [28] showing that the number of false positives of a yes-no Bloom filter is considerably less than the number of false positives of a standard Bloom filter of the same length (by the length we mean the total memory used up, that is, $G^+ + G^-$ for the yes-no Bloom filter and G for the Bloom filter).

As we have said earlier, in this paper we study the use scenario when no false negatives are allowed. However, yes-no Bloom filters can be also used when one allows some false negatives. Also it must be said that in this paper we concentrate on the use scenario in which the set $\mathcal{S}$ is fixed; accordingly, a yes-no Bloom filter representing $\mathcal{S}$ is built once and then only queried. If $\mathcal{S}$ is changed, we simply assume that the yes-no Bloom filter of the updated set is recalculated again. Some other generalisations of the Bloom filter are designed to work with the use scenario in which a limited number of elements may need to be added to $\mathcal{S}$ or excluded from $\mathcal{S}$ very fast, without rebuilding the Bloom filter from the Bloom filters of individual elements of $\mathcal{S}$; such constructions include, for example, counting Bloom filters [13].

1.3 Optimization and paper plan

In our computational experiments with yes-no Bloom filters [28] we were using a very simple heuristic for building the no-filter $\beta^-(\mathcal{S})$, which is very fast but not necessarily picks the best combination of false positives for the no-filter. By using a more advanced optimization algorithm the accuracy of yes-no Bloom filters can be improved further. Indeed, the difference between the standard Bloom filter and the yes-no Bloom filter is not only that the yes-no Bloom filter immediately provides some improvement of accuracy. What is also important is that the standard Bloom filter is rigidly determined by the set $\mathcal{S}$ (as bitwise disjunction of Bloom filters of all its elements), whereas when one builds a yes-no Bloom filter for a pair of sets $\mathcal{S}$ and $\mathcal{T}$, one can choose the no-filter in many different ways. Therefore, building a yes-no Bloom filter can and should be considered as an optimization problem. In particular,

if we do not allow any false negatives and allow some false positives, we can consider an optimization problem of maximizing the number of false positives recognized by the no-filter $\beta^-(\mathcal{S})$.

At this point we also need to comment whether using a more complicated algorithm is feasible. Indeed, running a more complicated and more time-consuming algorithm at the stage of *building* a compressed set is usually possible in applications. For instance, if we refer back to our airport example, it is querying that must be done very fast and perhaps with low-performance hardware, but building a compressed representation of the photographs of suspects is done infrequently and probably on a big computer.

The aim of this work is to develop an optimization model for building a no-filter, and discuss a number of approaches to solve the resulting integer program. The paper is organized as follows. In Section 2 we have a literature review on the optimization problems related to Bloom filters. In Section 3 we construct an optimization model for the no-filter, which maximizes the number of false positives included in the no-filter. Comments on the difficulty of the resulting pure integer program will be made in the end of Section 3. Section 4 is about the structural properties of the ILP model and possible simplifications. Specifically, a reduced ILP is introduced in this section by replacing the explicit cover of every single real positive by some statistical information and consequently ignoring the use of some integer variables in the initial model. Based on this simplification, an Approximate Dynamic Programming (ADP) model is then constructed in section 5, which makes use of the simplified ILP in its value function approximation. This ADP model is compared with a number of heuristics and shown effective in producing good solutions on most of the testing examples in Section 6. In Section 7 we make conclusions and comment on future work.

2 Literature review on optimization works

Various studies have been done on Bloom filters and data compression in general, while most of the optimization research that has been done in this context focuses on the discussion of how to select hash functions to minimize the rate of false positives. For example, [20] introduced the idea of *compressed Bloom filters* which limits the size of transmission by proxies, optimizing the number of hash functions to minimize both the rate of false positives and the size for transmission. On the other hand, [7] proved through statistical studies that the optimal setting of Bloom filter should be determined by the number of items involved in a peer-to-peer keyword searching rather than by minimizing the false positive rate. From the optimization point of view, all above works are related to nonlinear programming which aims to find the best parameters for data compression and transmission; thus, it falls not into the same category of optimization as the one used in this work.

To the best of the authors knowledge, the works most related to integer programming are done by Bruck et.al [5] and M. Zhong et.al [33]. Both works aim to minimize the Bloom filter false positive probability for given object popularities by customizing the number of hashes. The former achieves integrality by allowing arbitrary

solution and then rounding to the nearest integer value, while the latter proposes a constrained nonlinear integer programming model and develops two polynomial-time approximation algorithms based on dynamic programming.

In contrast, our study focuses on how to separate real positives of a given Bloom filter from its false positives and exclude as many as possible false positives. As it deals with a specific set of elements and specific false positives rather than an estimated *false positive rate*, the optimization model we are addressing here definitely requires the usage of integer variables. As a result, a Mixed-Integer Linear Programming (MILP) [21] model is developed; in this model, the objective is to maximize the number of false positives selected for the no-filter, and constraints are imposed to ensure that no false negatives are created through this selection process. We will show this problem shares some similarities with the Multidimensional Knapsack problem [17] [10] [2] and therefore is $\mathcal{NP}$-complete.

The major methodology we use to solve the large scale combinatorial optimization problem is based on Approximate Dynamic Programming [23] [1]. Approximate Dynamic Programming (ADP) is a powerful tool for handling the curse of dimensionality. There are several aspects of this research, most of which focus on the approximation of value-to-go function in traditional Dynamic Programming based on discrete state and action space. The last decade is the period ADP has grown most significantly. It has shown a number of empirical successes in large-scale real-world applications such as in revenue management [32] [31], job scheduling [8], Internet traffic routing [12] [30] and so on. In this work we will develop a one-step improving scheme based on a reduced ILP to solve the optimization model.

3 Mathematical model

In this section we shall discuss the optimization model for picking up elements for the no-filter $\beta^-(\mathcal{S})$. As described in the introduction, by using a Bloom filter we can compress data through representing a set $\mathcal{S}$ as the bitwise disjunction $\beta^+(\mathcal{S})$ of Bloom filters of its elements $s \in \mathcal{S}$. Building a Bloom filter will likely introduce some false positives, $f \in \mathcal{F}$, which are the objects that are incorrectly recognised by $\beta^+(\mathcal{S})$. Ideally we hope to exclude all false positives while retaining all elements of $\mathcal{S}$ by adding a no-filter, $\beta^-(\mathcal{S})$, based on a subset of $\mathcal{F}$. Therefore, any element that belongs to $\beta^+(\mathcal{S})$ but is excluded by $\beta^-(\mathcal{S})$ should be recognized as an element in set $\mathcal{S}$.

It is not difficult to imagine that in most cases, taking all $f \in \mathcal{F}$ to create $\beta^-(\mathcal{S})$ is not a good decision as such a Bloom filter may cover some $s \in \mathcal{S}$ again. To simplify the process, we start from the assumption that we allow some false positives in $\beta^+(\mathcal{S})\backslash\beta^-(\mathcal{S})$ with the hope of excluding as many of them out as possible, while retaining all elements in $\mathcal{S}$. To this end, we will develop an integer programming model and the notation for it is presented as follows.

3.1 Notation

Sets and indices

- $\mathcal{S}, s \in \mathcal{S}$: The set of elements which the Bloom filter aims to represent;
- $\mathcal{F}, f \in \mathcal{F}$: The set of false positives which are incorrectly recognised by the yes-filter $\beta^+(\mathcal{S})$.

Parameters

- N: the number of elements in set $\mathcal{S}$;
- M: the number of false positives; that is the size of the set $\mathcal{F}$;
- G: the total number of bits in the no-filter of each object $\eta^-(s)$;
- H: the number of nonzero bits in the no-filter of each object $\eta^-(s)$;
- $A = (a_{sj})_{N \times G}$: parameter matrix for set $\mathcal{S}$, where
  $a_{sj} = \begin{cases} 1, & \text{if element } s\text{'s bit } j \text{ is } 1 \\ 0, & \text{otherwise} \end{cases}, s = 1, ..., N; j = 1, ..., G.$
- $B = (b_{fj})_{M \times G}$: parameter matrix for false positive set $\mathcal{F}$, where
  $b_{fj} = \begin{cases} 1, & \text{if false positive } f\text{'s bit } j \text{ is } 1 \\ 0, & \text{otherwise} \end{cases}, f = 1, ..., M; j = 1, ..., G.$

Decision variables

- $x_f = \begin{cases} 1, & \text{if false positive } f \text{ is selected for disjunction } \beta^-(\mathcal{S}) \\ 0, & \text{otherwise} \end{cases}$,
  $f = 1, ..., M.$
- $\delta_j = \begin{cases} 1, & \text{if bit } j \text{ is 1 in the disjunction } \beta^-(\mathcal{S}) \text{ with selected false positives} \\ 0, & \text{otherwise} \end{cases}$,
  $j = 1, ..., G.$

3.2 Integer programming model

Our aim here is to maximize the total number of false positives that are picked up for disjunction $\beta^-(\mathcal{S})$, while guaranteeing that none of elements of $\mathcal{S}$ is recognised by $\beta^-(\mathcal{S})$.

$$
\begin{aligned}
(ILP) \qquad & \max \sum_{f=1}^{M} x_f && && (3.1)\\
& s.t. \sum_{f=1}^{M} b_{fj} x_f \leq M\delta_j, && j = 1, ..., G && (3.2)\\
& \sum_{j=1}^{G} a_{sj}\delta_j \leq H - 1, && s = 1, ..., N && (3.3)\\
& x_f \in \{0, 1\}, && f = 1, ..., M && (3.4)\\
& \delta_j \in \{0, 1\}, && j = 1, ..., G && (3.5)
\end{aligned}
$$

Constraint (3.2) relates δ with decision x, which makes $\delta_j = 1$ if bit j is 1 in any selected false positives. Constraint (3.3) guarantees no element of $\mathcal{S}$ is excluded by saying, for each element in $\mathcal{S}$ there is at least one bit that is not being covered by the selected false positives for $\beta^-(\mathcal{S})$.

This is a pure integer programming problem which contains $M + G$ binary variables and $N + G$ constraints. Although the size of the problem increases linearly with its components, due to its integrality nature the solution difficulty grows exponentially as to prove optimality, we need to examine objective value at a maximum of 2^{M+G} nodes.

3.3 Difficulty

It is not difficult to see that the above problem shares some similarities with the so-called Multidimensional Knapsack Problem (MKP) [10]:

- The aim is to maximize the total value (number) of items picked for package.
- Decisions are, for each item type, whether to include it in the package.
- Constraint(s) restrict on the selection of items in such a fashion that if some items have been selected then some others must be excluded. The major difference is that Knapsack problems impose this restriction by limiting the total weights/volumes whereas our problem impose it by checking if the disjunction will cover any elements.

Indeed, we can show that the above problem is reducible to a 0-1 MKP. To achieve this, let us reformulate the problem to interpret the fact that, if several false positives' bitwise disjunction covers any element, we have to exclude the possibility of having all their $x_f = 1$ in the final solution. As for every element in $\mathcal{S}$ it is not difficult to find all h-degree (a combination of $h \leq H$ items) combinations from the limited number of false positives which fully cover this element, we can list all such combinations and restrict for each of them, the number of selected items cannot go higher than $h - 1$. For a fixed element $s \in \mathcal{S}$, let $\mathcal{L}_{hs}$ denote the set of h-degree combinations, each of which fully covers s, then mathematically above requirement can be represented as:

$$\sum_{f=1}^{M} e_{fl} x_f \leq h-1, \quad \forall l \in \mathcal{L}_{hs}, h = 1, ..., H,$$

where $e_{fl} = \begin{cases} 1, & \text{if false positive } f \text{ is an element of combination } l \\ 0, & \text{otherwise} \end{cases}, f = 1, ..., M; l \in \mathcal{L}_{hs}$.

Then, the original model (ILP) is converted into a MKP with less variables but (in general) more constraints:

$$\begin{aligned} & \max \sum_{f=1}^{M} x_f && && (3.6) \\ (MKP) \qquad & s.t. \sum_{f=1}^{M} e_{fl} x_f \leq h-1, && \forall l \in \mathcal{L}_{hs}, h = 1, ..., H, s = 1, ..., N && (3.7) \\ & x_f \in \{0,1\}, && f = 1, ..., M && (3.8) \end{aligned}$$

3.4 A small example

Here we consider an example where there are two elements in $\mathcal{S}$ and four false positives with $H = 2$. Following the previous definition, assume

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}.$$

The (ILP) of this easy example is given as:

$$\begin{aligned} & \max \; x_1 + x_2 + x_3 + x_4 && \\ & s.t. \; x_1 + x_2 \leq 4\delta_1, && \\ & \quad x_3 \leq 4\delta_2, && \\ & \quad x_1 + x_4 \leq 4\delta_3, && \\ (ILP\_eg) \qquad & \quad x_2 + x_3 + x_4 \leq 4\delta_4, && \\ & \quad \delta_2 + \delta_3 \leq 1, && \\ & \quad \delta_1 + \delta_2 \leq 1, && \\ & \quad x_f \in \{0,1\}, && f = 1, ..., 4 \\ & \quad \delta_j \in \{0,1\}, && j = 1, ..., 4 \end{aligned}$$

To convert it into a corresponding (MKP), we have to go through the nonzeros of each element to check which false positive it relates to. For instance, the first element has its second and third bits nonzero, which does not allow the combination

of false positives $\{1,3\}$ and $\{3,4\}$ to be considered in the optimal solution. Therefore we create constraints:

$$x_1 + x_3 \leq 1, \text{ and } x_3 + x_4 \leq 1.$$

Similarly, considering the second element we obtain:

$$x_1 + x_3 \leq 1, \text{ and } x_2 + x_3 \leq 1.$$

And thus the complete (MKP) turns out to be (removing repeated constraints):

$$(MKP\_eg) \qquad \begin{aligned} \max \; & x_1 + x_2 + x_3 + x_4 \\ s.t. \; & x_1 + x_3 \leq 1, \\ & x_3 + x_4 \leq 1, \\ & x_2 + x_3 \leq 1, \\ & x_f \in \{0,1\}, \qquad f = 1, ..., 4 \end{aligned}$$

Solving both systems we will achieve the same optimal solution $x_1 = x_2 = x_4 = 1$ which means $\beta^-(\mathcal{S}) = \{1,2,4\}$. Note that this example shows just a special case of the problem for which the (MKP) is a smaller system comparing with (ILP). Actually in practice when the number of items for both sets increases, the (MKP) will become far larger than the (ILP) as it has to consider every single infeasible combination of false positives.

It is well known that 0-1 MKP is $\mathcal{NP}$-complete, therefore a polynomial time solution approach for our problem may not be available. Nevertheless, the Knapsack problem is amongst the most studied problems in combinatorial optimization on which much theoretical and empirical work has been done, including some clever heuristics and/or pseudo-polynomial approaches like the greedy method and Dynamic Programming. In this work we will be talking about a greedy heuristic based on Approximate Dynamic Programming.

4 Structural properties and simplification

We now try to take the special structural properties of it to design a heuristic which is anticipated to provide reasonably good (nearly maximized), and always feasible solutions. Traditional optimization approaches for the pure integer programming problems are based on Branch-and-Bound. This methodology starts from finding the optimal solution of the LP relaxation of the corresponding ILP and branch on the fractional optimal solution observed to search for the optimal integer solutions. In the intermediate steps, a number of integer feasible solutions are observed, whose corresponding objective values serve as the lower bound of the final optimal integer solution we are looking for. Therefore, it is not difficult to find feasible integer solutions throughout this procedure, while the biggest difficulty is to improve on the current best solution and/or to prove optimality without going through the whole Branch-and-Bound tree explicitly.

4.1 Relaxation

To achieve an acceleration for the solution approach let us firstly consider the partial LP relaxation of the (ILP), defined as ($LPR_{partial}$), which allows decision x to take fractional values between 0 and 1.

$$
\begin{aligned}
& \max \sum_{f=1}^{M} x_f && && (4.1)\\
& s.t. \sum_{f=1}^{M} b_{fj} x_f \leq M\delta_j, && j = 1, ..., G && (4.2)\\
(LPR_{partial}) \qquad & \sum_{j=1}^{G} a_{sj}\delta_j \leq H - 1, && s = 1, ..., N && (4.3)\\
& x_f \in [0, 1], && f = 1, ..., M && (4.4)\\
& \delta_j \in \{0, 1\}, && j = 1, ..., G && (4.5)
\end{aligned}
$$

Lemma 4.1. *The optimal solution of $LPR_{partial}$ must have $x^*_{\tilde{f}} = 0$ or $x^*_{\tilde{f}} = 1, \tilde{f} = 1, ..., M$.*

Proof. Let $(\mathbf{x}^*, \delta^*)$ be the optimal solution of $LPR_{partial}$. Assume $x^*_{\tilde{f}} = \epsilon$ and $0 < \epsilon < 1$. As $\mathbf{x}^* = (x^*_1, ..., x^*_{\tilde{f}}, ..., x^*_M)$ is a feasible solution, it must satisfy:

$$\sum_{f=1}^{M} b_{fj} x_f \leq M\delta_j, \quad j = 1, ..., G$$

Therefore for any j:

- If $b_{\tilde{f}j} = 0$, modifying $x_{\tilde{f}}$ value will not have any impact to this constraint. So if we increase $x_{\tilde{f}}$ value from ϵ to 1, the constraint is also satisfied.
- If $b_{\tilde{f}j} = 1$, to satisfy the constraint we must have

  $$0 < \epsilon = x_{\tilde{f}} \leq \sum_{f \neq \tilde{f}} b_{fj} x_f + x_{\tilde{f}} \leq M\delta_j,$$

  which implies $\delta_j = 1$ and the constraint becomes: $\sum_{f \neq \tilde{f}} b_{fj} x_f + x_{\tilde{f}} \leq M$. As $\sum_{f \neq \tilde{f}} b_{fj} x_f \leq M - 1$, we can always increase the value of $x_{\tilde{f}}$ from ϵ to 1 without violating the constraint.

While in both situations, by increasing $x_{\tilde{f}}$ value from ϵ to 1 we can achieve an improvement on the objective value by $1 - \epsilon$. Therefore we proved in the optimal solution $x^*_{\tilde{f}} = 0$ or $x^*_{\tilde{f}} = 1, f = 1, ..., M$. □

Lemma 4.2. *The optimal solution of $LPR_{partial}$ must also be the optimal solution of ILP.*

Proof. This is a direct conclusion of Lemma 4.1 since if the optimal solution of a LP relaxation turns out to be integer, it must also be the optimal solution of the original IP problem. □

Up until now we have reduced the problem into a Mixed-Integer Programming (MIP) problem, where the only integer restrictions are made on δ values. Now we will revise the constraints to see if we can simplify the problem further by reducing the number of variables that we need to consider in the optimization model.

4.2 Simplification

Remember that constraints (4.3), which are only relating to δ values, impose the restriction that for any element in $\mathcal{S}$ there must be at least one bit to be uncovered by the selected false positives disjunction $\beta^-(\mathcal{S})$. As these constraints are somehow the "hard constraints" which cannot be further simplified, let us consider whether we can get rid of constraints (4.2) instead, and therefore remove x variables completely.

Suppose we can split the problem into two steps according to the constraints, then:

- The first step, corresponding to constraints (4.2), is to create a disjunction $\beta^-(\mathcal{S})$ by picking up a collection of false positives.
- The second step, corresponding to constraints (4.3), is to check whether the disjunction $\beta^-(\mathcal{S})$ fully covers any element in $\mathcal{S}$.

If we hope to do every step individually, in the second step we have to define an aim that we want to achieve, which somehow reflect our initial aim of picking up as many false positives as possible for $\beta^-(\mathcal{S})$. Without the value of xs we cannot do this accurately, but it is not difficult to imagine that, potentially, the more ones in disjunction $\beta^-(\mathcal{S})$ (the more bits covered by $\beta^-(\mathcal{S})$), the more false positives can be selected. To make it more accurate, we can also consider for every single bit, how many false positives it overlaps with. Potentially, the more false positives one bit is covered by, the higher possibility to make more xs equal 1 by allowing $\beta^-(\mathcal{S})$ contains this bit. Therefore let us define a *cardinality*, c_j, for every single bit j, which indicates how many false positives having this bit equals 1:

$$c_j = \sum_{f=1}^{M} b_{fj}, \quad j = 1, ..., G,$$

and then use it to define the objective in an approximated problem containing only second step variables:

$$\max \sum_{j=1}^{G} c_j \delta_j \tag{4.6}$$

$$(APP) \qquad s.t. \sum_{j=1}^{G} a_{sj} \delta_j \leq H - 1, \qquad s = 1, ..., N \tag{4.7}$$

$$\delta_j \in \{0, 1\}, \qquad j = 1, ..., G \tag{4.8}$$

The resulting system (APP) is a reduced pure ILP with G variables and N constraints, which becomes a so-called Set Packing problem [29]. Although the Set Packing problem is also $\mathcal{NP}$-complete [11], with significantly reduced size, the (APP) is more tractable than the original (ILP).

By solving this problem we will get efficiently an integer solution δ^* which indicates a disjunction $\beta^-(\mathcal{S})^*$. Then going through the first step is straightforward:

if a false positive is completely covered by $\beta^-(\mathcal{S})^*$ we include it in the collection, otherwise we exclude it.

Through the later tests with examples, we can see this method works very well and efficiently for applications having thousands of elements and false positives.

5 Approximate dynamic programming model

One typical pseudo-polynomial method to solve the Knapsack Problems is Dynamic Programming (DP) [19] [15]. Nevertheless DP suffers from the so-called "curse of dimensionality" for large size problems [23]. In this section we will consider DP to solve the problem, while avoiding the curse of dimensionality by using an approximated value function.

5.1 DP model

- Stage: $k = 1, ..., M$, consider every false positive in its natural order.
- State: $D_k = (d_1^k, d_2^k, ..., d_G^k)$, which denotes the bitwise disjunction of yet-selected items after considering false positives $1, ..., k$ and

$$d_j^k = \begin{cases} 1, & \text{if bit } j \text{ is covered by yet-selected items from false positives } 1, ..., k \\ 0, & \text{otherwise} \end{cases},$$
$$j = 1, ..., G.$$

- Decision:

$$u_k = \begin{cases} 1, & \text{if bring false positive } k \text{ into } \beta^-(\mathcal{S}) \\ 0, & \text{otherwise} \end{cases}, k = 1, ..., M.$$

- Recurrence:

$$V_k(D_k) = \begin{cases} -\infty, & \text{if } D_k \text{ fully covers ANY element in } \mathcal{S} \\ \max\{1 + V_{k+1}(D_k \oplus b_{k\cdot}), V_{k+1}(D_k)\}, & \text{otherwise} \end{cases},$$
$$k = 1, ..., M-1;$$

$$V_M(D_M) = \begin{cases} -\infty, & \text{if } D_M \text{ fully covers ANY element in } \mathcal{S} \\ 0, & \text{otherwise} \end{cases},$$

where $D_k \oplus b_{k\cdot} \equiv \{d_j^k + b_{kj}(1 - d_j^k) | j = 1, ..., G\}$, which denotes the state after bringing false positive k into consideration.

If the DP can be solved exactly it will definitely find the optimal solution as it considers potentially all possible combinations of false positives. To make decision for stage k we have to evaluate function values for stage $k+1$ at all possible states beforehand. So starting from $k = M$, following a backward decision process we can obtain the optimal decision u_k iteratively for all stages and states, which contain the optimal solution as the best path we ought to follow from $k = 1$ to M.

Nevertheless, it is not always realistic to solve the DP accurately in practice, especially when the number of states increases exponentially with the problem size. As an alternative, let us consider the approximation to the DP model which uses an approximated value function for V_{k+1} that is easy to evaluate without going through all the later stages in detail.

5.2 ADP model

This section introduces an ADP heuristic, which takes use of the simplified model (APP) as we defined in Section 4.2 to approximate the value function V_{k+1}. Basically we will follow all the definitions of DP model in the ADP, besides the fact that we will step forward in stage to check in turn whether the current false positive k should be included in the final disjunction $\beta^-(\mathcal{S})$. As this "forward" process just needs to be carried out once, we could accelerate the whole solution process significantly by using ADP instead of DP.

In the DP model, at stage k we need to solve the recurrence:

$$V_k(D_k) = \max\{1 + V_{k+1}(D_k \oplus b_{k\cdot}), V_{k+1}(D_k)\} \tag{5.1}$$

which results in the best action, u_k, that is to take for false positive k. Instead of calculating V_{k+1} explicitly through the backward recurrence, we modify the (APP) problem to give quickly an approximation to the V_{k+1} value at the current state D_k and $D_k \oplus b_{k\cdot}$, denoted as $\bar{V}_{k+1}(D_k)$ and $\bar{V}_{k+1}(D_k \oplus b_{k\cdot})$. This requires to solve 2 subproblems (APP_0^k) and (APP_1^k) defined as follows.

(APP_0^k) is defined as the optimization problem associated with the decision $u_k = 0$.

$$\max \sum_{j \notin D_k} c_{0j}^k \delta_j \tag{5.2}$$

$$(APP_0^k) \qquad s.t. \sum_{j \notin D_k} a_{sj}\delta_j \leq (H-1) - \sum_{j \in D_k} a_{sj}, \qquad s = 1, ..., N \tag{5.3}$$

$$\delta_j \in \{0,1\}, \qquad j \notin D_k \tag{5.4}$$

where $c_{0j}^k = \sum_{\tilde{k}=k+1}^{M} b_{\tilde{k}j}, j \notin D_k$, which define the *cardinality* of bits according to their appearance in the false positives that have not been considered yet. Solving (APP_0^k) we will get the optimal solution δ_0^k, in terms of which bit should be covered by the disjunction generated with false positives $k+1, ..., M$. And $\bar{V}_{k+1}(D_k)$ is calculated as the number of false positives in $k+1, ..., M$ that are fully covered by this disjunction.

On the other hand, (APP_1^k) is defined as the optimization problem associated with the decision $u_k = 1$. So in addition to D_k which is the current disjunction, we also need to exclude all bits that have been covered by the false positive k and consider the best disjunction that is to be generated with false positives $k+1, ..., M$.

$$\max \sum_{j \notin D_k \oplus b_{k\cdot}} c_{1j}^k \delta_j \tag{5.5}$$

$$(APP_1^k) \qquad s.t. \sum_{j \notin D_k \oplus b_{k\cdot}} a_{sj}\delta_j \leq (H-1) - \sum_{j \in D_k \oplus b_{k\cdot}} a_{sj}, \qquad s = 1, ..., N \tag{5.6}$$

$$\delta_j \in \{0,1\}, \qquad j \notin D_k \oplus b_{k\cdot} \tag{5.7}$$

where $c_{1j}^k = \sum_{\tilde{k}=k+1}^{M} b_{\tilde{k}j}, j \notin D_k \oplus b_{k\cdot}$. Solving (APP_1^k) we will get the optimal solution δ_1^k, and $\bar{V}_{k+1}(D_k \oplus b_{k\cdot})$ is calculated as the number of false positives in $k+1, ..., M$ that are fully covered by this disjunction. With the resulting $\bar{V}_{k+1}(D_k)$ and $\bar{V}_{k+1}(D_k \oplus b_{k\cdot})$ values, we can then solve problem (5.1) for an approximated decision $\bar{u}_k$.

Using this approximation scheme, to solve the entire ADP model we need to solve $2M$ binary subproblems. Nevertheless, as the size of problem decreases gradually with the increased number of bits that have already been covered by D_k, the solution process will get increasingly quicker as each next stage. Besides this, we will also take use of the special recursive manner of (APP_0^k) and (APP_1^k) to further simplify the solution process.

5.3 Acceleration

To improve the efficiency of solving the ADP model, in this section we will discuss some additional simplifications around the binary subproblems, motivated by the fact that there are large similarities between the subproblems of stage k and $k+1$.

- Firstly, observe that if the best action in stage k was decided as $u_k = 0$, we have $D_{k+1} = D_k$ which means constraint set for (APP_0^k) is completely the

same as constraint set for (APP_0^{k+1}). The only difference between these two problems is then the cardinality in the objective, for which (APP_0^{k+1}) has H variables possessing one unit less c_{0j}^k than (APP_0^k). As the cardinality is in general comparable to the number of false positives which is normally hundreds or thousands, it is reasonable to believe that in most cases this 1 unit less will not change the optimal solution significantly. Therefore we can use $\bar{V}_{k+1}(D_k)-1$ to approximate $\bar{V}_{k+2}(D_{k+1})$ if no new false positive has been added to D_k in stage k ($u_k = 0$), and the latter could directly be used in recurrence for stage $k+1$.

- Secondly, if the best action in stage k was decided as $u_k = 0$ and if by solving stage k's subproblem (APP_0^k) we already have $u_{k+1} = 1$ (bits of false positive k+1 are fully covered by δ_0^k), then the optimal solution of subproblem (APP_1^{k+1}) plus $u_{k+1} = 1$ is identical to the optimal solution to (APP_0^k). Therefore we don't need to re-solve the former again in stage $k+1$ but instead use $\bar{V}_{k+2}(D_{k+1} \oplus b_{k+1\cdot}) = \bar{V}_{k+1}(D_k) - 1$.
- Thirdly, if the best action in stage k was decided as $u_k = 1$ and if by solving stage k's subproblem (APP_1^k) we already have $u_{k+1} = 1$ (bits of false positive k are fully covered by δ_1^k), then the optimal solution of subproblem (APP_1^{k+1}) plus $u_{k+1} = 1$ is identical to the optimal solution to (APP_1^k). Therefore we don't need to re-solve the former again in stage $k+1$ but instead use $\bar{V}_{k+2}(D_{k+1} \oplus b_{k+1\cdot}) = \bar{V}_{k+1}(D_k \oplus b_{k\cdot}) - 1$.

By considering all above possibilities, lots of binary subproblems can be skipped for the optimization step. For example, let us consider Ins. 5 as given in the numerical test Section 6.2. Originally we need to solve $2M = 7442$ integer programs, while by applying the acceleration strategy we can remove 2712 of them with criterion 1, 745 with criterion 2, 246 with criterion 3 and end up with only 3739 integer programs to solve which is nearly half of the original number. As a result the running time of ADP algorithm reduces significantly (from $19.71s$ to $10.04s$ for Ins. 5), which creates better possibilities for managing larger size problems. Of course we can further the acceleration design by also considering some warmstart approaches according to the closest previous optimal solution as observed. But this is omitted in this work as the solution process takes only 8 minutes for instances with ten-thousand false positives (Ins. 9 and Ins. 10), and performs much quicker than alternative methods like B&B. For more details on the running time information please refer to Section 6.2.

6 Numerical tests

To test our ADP model, we firstly develop several simple heuristics which are mainly designed to reflect the nature of the problem.

6.1 Simple heuristics

Our aim is to select as many false positives as possible to generate the disjunction $\beta^-(\mathcal{S})$ without covering any elements in it. To achieve this aim, the simplest idea

is to consider false positives in turn, following the order in which they are defined, to check if selecting the next item will cover any elements. If not, we select it for $\beta^-(\mathcal{S})$; otherwise we exclude it. This is actually a **naive method** that has been used by [27] for testing the effectiveness of the yes-no Bloom filter when it had been firstly introduced. We will make use of this naive method as the first heuristic to compare our ADP model with.

It is obvious that the naive heuristic as described above will not lead to a generally "good" solution, since following different sequences of considering items will definitely produces different solutions. Only if we can figure out the "optimal" sequence of considering items, we can build the optimal solution this way. Therefore, in this section we will develop some other rules, which are easy to achieve with simple calculations, for how the sequence can be identified. Our aim here is to define for every false positive a *degree*, which indicates how large the impact will be to the procedure of including other items, if we decide to include this item in $\beta^-(\mathcal{S})$. Note that most of the following discussion are around the case where $H = 2$.

6.1.1 Element-only degree

Naturally, we assume that there are no duplicate items in both the element set and the false positive set. This assumption can easily be satisfied with a preliminary step removing all identical items from both set. Let us consider a single bit j. If this bit is contained in n elements in $\mathcal{S}$, in order to make all of these n_j elements not covered by $\beta^-(\mathcal{S})$, we have to make other n_j bits that are covered by this n_j elements not covered by $\beta^-(\mathcal{S})$. This means, in $\beta^-(\mathcal{S})$ we have to restrict n_j bits to 0 if we decide to select bit j. Therefore, the higher the value of n_j associated with a bit, the less preferred to include this bit as potentially including it will lead to the exclusion of many others.

Secondly, selecting any false positive f will make two bits covered by $\beta^-(\mathcal{S})$. Suppose the two bits are j_1 and j_2, then including item f in $\beta^-(\mathcal{S})$ will make at most $n_{j_1} + n_{j_2}$ other bits been restricted to 0. As the more zeros there are in $\beta^-(\mathcal{S})$, potentially the less false positives we can select, therefore we treat this information as a measure of the *degree* of false positive f, which will be used to decide the sequence in which all the false positive s are considered:

$$DGR_f^e = \sum_{j=1}^{G} b_{fj} n_j, \quad f = 1, ..., M.$$

So the smaller the DGR_f^e value is, the earlier the corresponding item will be considered and the larger opportunity for it to be selected. As this method does not consider the detailed coverage of elements by false positives, it is a very efficient heuristic.

6.1.2 Element-and-False-Positive degree

In practice we may see lots of same values in DGR_f^e as defined above. If two items share the same degree, we hope to take more information into account to decide on

their sequences. This information comes from considering the false positives in a similar fashion. If a bit j is covered, then potentially all false positives containing the same bit are easier to include than those which are not, as they will just bring one more bit to 1 in the disjunction $\beta^-(\mathcal{S})$, and the fewer ones we observe in $\beta^-(\mathcal{S})$, the less chance there is of an element being covered by it. Therefore, it is better to cover earlier the bits j which are contained by more false positives, since this creates more opportunity for later selection.

So for bit j, we denote by m_j the number of false positives containing this bit, and define an alternative *degree* value according to m_js:

$$DGR_f^b = \sum_{j=1}^{G} b_{fj} m_j, \quad f = 1, ..., M.$$

Thus, if there are any ties when we just check the DGR_f^e values, we take DGR_f^b into consideration and pick up items in a non-decreasing order of DGR_f^b.

6.1.3 False-Positive covering degree

Remember that our ultimate goal is to pick up as many as possible false positives without covering any elements by the resulting disjunction $\beta^-(\mathcal{S})$. In practice, selecting any false positive may result in the exclusion of some others. Suppose we can determine accurately how many false positives will be excluded afer selecting a false positive f; let us denote this number by DGR_f^c. Intuitively we would like to consider those having smaller DGR_f^cs with priority.

This method finds explicitly how many false positives will be excluded by selecting a specific false positive f. Consider every single false positive f; by including it we will make at most two more bits in $\beta^-(\mathcal{S})$ be 1. Exactly how many bits in $\beta^-(\mathcal{S})$ will be changed to 1 actually depends on what bits have already been covered by the false positives considered and selected before f. As here our aim is to choose a reasonable order of items, let us treat every single false positive f independently, or equivalently, assume item f is the first being considered, to see how many other false positives will be excluded by selecting item f.

Suppose false positive f has two bits, j_1 and j_2, equal one. Let $\mathcal{S}_{j_1,j_2}$ denote a subset of $\mathcal{S}$ whose elements having j_1 or j_2 bit equals 1. If item f is selected, all elements in $\mathcal{S}_{j_1,j_2}$ will create a disjunction $\beta^-(\mathcal{S}_{j_1,j_2})$ and any false positive (except f) which overlaps with $\beta^-(\mathcal{S}_{j_1,j_2})\backslash\{j_1, j_2\}$ are then needed to be excluded. The total number of such false positives are defined as DGR_f^c. As using this heuristic we have to consider in detail the coverage of all elements by item f, the resulting heuristic works less efficiently than the two above.

6.2 Numerical results on individual tests

Table 1 lists the numerical results of testing above ADP and heuristics on some randomly generated examples (in each example, a certain number of no-filters are picked at random with equal probability from the set of all Bloom filters with given parameters), which are compared with the best solution that solving the ILP model with

CPLEX can find within pre-specified amount of time. Note that here we restricted the CPLEX to go through the Branch-and-Bound step for just limited time as for large problems, although it can quickly find a feasible integer solution, it will take very long time to improve it or prove its optimality. The specified time below is the average time that we observed through tests as when a "quick initial improves" finishes. After the cut-off time as specified, most examples will just continue the branching step without improving the objective for hours.

Ins	Dimensions			Simple Heuristics				APP		ADP		CPLEX		
	G	N	M	Naive	DGR_f^e	DGR_f^b	DGR_f^c	APP	time	ADP	time	ILP	time	gap
1	256	100	289	188	190	190	198	198	0.0029s	201	0.29s	201	0.02s	0%
2	256	100	268	172	181	188	188	189	0.0027s	191	0.30s	192	0.03s	0%
3	512	200	479	301	309	322	332	336	0.0057s	339	1.25s	340	0.03s	0%
4	512	200	501	329	327	331	346	347	0.0063s	353	1.35s	355	0.03s	0%
5	256	400	3721	753	868	883	900	994	0.0221s	1009	10.04s	967	50s	85.10%
6	256	400	3735	759	980	966	959	1109	0.0229s	1132	10.20s	1109	50s	65.30%
7	512	400	5852	2157	2516	2490	2527	2656	0.0409s	2659	48.24s	2651	100s	24.51%
8	512	400	6188	2236	2682	2710	2708	2820	0.0378s	2835	48.17s	2834	100s	28.52%
9	1024	1015	13026	4083	4616	4719	4676	5062	0.1733s	5096	406.13s	5117	600s	48.74%
10	1024	1000	12004	3757	4324	4386	4407	4760	0.1785s	4892	403.23s	4809	600s	46.92%

Table 1: Numerical results on 10 randomly generated examples with different sizes. "gap" represents the percentage difference between the best integer solution and the optimal solution of LP relaxation. Entries with underlines indicate the best possible solution out of all proposed algorithms.

Table 1 summarises numerical results on 10 examples tested with four heuristic methods as proposed, compared with the solution given by APP, ADP models and ILP solved by CPLEX. Small examples, like Examples 1-4, can be solved efficiently to optimality (with gap 0) by CPLEX. At the same time, heuristic methods also provide very good feasible integer solutions which are quite close to the optimal for small problems. ADP works especially well for all small examples, picking just one or two less false positives than what can be found by CPLEX. Note also that the execution times for all simple heuristic algorithms are small comparing to ADP and CPLEX. But since they are generally worse and just served as benchmarks for optimality comparison, we exclude the execution time of these heuristic algorithms.

In contrast, when the number of false positives go above $1,000$, achieving optimality by CPLEX takes unreasonable time (e.g. it costs another 1 hour to reduce the gap from 85.10% to 81.28% in Ins. 5) and effort (computer memory to save the Branch-and-Bound tree). As CPLEX can still report the current best solution out of the existing Branch-and-Bound tree after the cut-off time, we put this solution in the corresponding column of CPLEX together with the gap between the current best integer solution and the optimal objective of its LP relaxation. Looking at the table we can see that for larger size problems, like examples $5-10$, simple heuristics perform much worse than APP and ADP. ADP in general is the best performance heuristic which can always give as good solutions as CPLEX, within roughly a half

of the solution time. As CPLEX cannot give an optimal solution in general, in most cases ADP can do even better than CPLEX like for all examples except Example 9. In general, the larger the gap that CPLEX left until the cut-off time, the more ADP is improving on the CPLEX result.

6.3 Numerical results on random group tests

From above test examples it is clear that the simple heuristic methods as described in the beginning of this section have the worst performance in general. Therefore, in later tests we will just focus on the APP and ADP and compare them with CPLEX solutions (for limited running time) to see if we can improve on CPLEX best feasible solutions within shorter time. Following experiments are all run on 100 random examples. Three typical problem sizes are considered.

Tests	Dimensions			APP		CPLEX			ADP			
	G	N	M	APP	time	ILP	time	gap	ADP	time	AvrImpr	NoImpr
T1	256	100	300	204.17 ±7.33	0.0025s ±0.0001s	207.85 ±6.67	0.02s ±0.00s	0%	205.91 ±7.31	0.26s ±0.01s	-0.94%	32
T2	512	400	6000	2708.96 ±61.84	0.0416s ±0.0015s	2711.92 ±60.27	100s	25.03% ±4.38%	2746.24 ±63.10	49.64s ±1.77s	1.27%	80
T3	1024	1000	12000	4779.32 ±114.36	0.1698s ±0.0211s	4798.83 ±123.15	600s	45.96% ±3.47%	4860.69 ±132.59	404.88s ±10.31s	1.29%	96

Table 2: Numerical results on 100 randomly generated examples with different sizes. Test results are given in the mean±s.d. form. "AvrImpr" indicates the percentage improvement on the average objective value (number of false positives selected) that ADP policy has over CPLEX; "NoImpr" indicates the number of instances that ADP improves on or equal to CPLEX out of 100 random examples.

The T1 is on small examples again, which can be solved to optimality directly by CPLEX. We can see in general ADP perform quite close to it, which can pick up just slightly less (by 0.94%) false positives than the optimal solution and performs the same well as CPLEX on about one third examples (32 out of 100). The T2 is on medium sized examples. Here we allow CPLEX to run for 100 seconds which is believed to be long enough for the Branch-and-Bound method to get initial updates to a good enough node in most cases. We can see ADP works much better than CPLEX in average for this 100 random runs, which improves the CPLEX best feasible solution by 1.27%. As in average there is a relatively small gap (25.03%) between the best feasible integer solution and the LP relaxation observed in CPLEX, we have reason to believe the ADP model gives quite good sub-optimal solution (if not optimal) in most cases. Indeed, ADP works better than CPLEX on 80% of random examples as tested, which emphasises on the strength of it from another point of view. On the other hand, the average running time for this sized ADP model is roughly 50 seconds, which proves the efficiency of ADP model. Similarly for large examples, T3, ADP improves the CPLEX best feasible solution by 1.29%, which picks up roughly 60 more false positives to make the final no-filter more accurate. In addition, ADP

picks up more false positives on 96 out of 100 instances, which means ADP improves the CPLEX result almost certainly.

7 Conclusions and Future Work

In this work we have discussed the optimization model for a new Bloom filter construction, a yes-no Bloom filter. The optimization problem is a pure integer program which is reducible to multi-dimensional Knapsack problems with special constraints. This points out the difficulty of solving the whole problem to integral optimality, especially considering the problem size grows exponentially with the number of items/bits. On the other hand, some of the separation variables in the pure optimization model can be relaxed to continuous ones without changing the final solution. Although the partial relaxed model does not make the solution of the problem easier, it gives us some insights on how to simplify the model in order to design efficient heuristics producing sub-optimal solutions. A number of such heuristics have been presented in this work which all outperform the naive method used in the original paper [27].

In addition to this, as in general the Knapsack problem can be solved by Dynamic Programming techniques, we also build the DP model for this optimization problem which is then extended to Approximate Dynamic Programming (ADP) to deal with the dimensionality in its recurrences. The resulting ADP model uses a one-step improvement scheme, which approximates the value-to-go function at any specific state by solving a simplified ILP with largely reduced size in contrast to the original integer model. The resulting ADP model has been justified effective in its performance, which in general picks up more false positives in the Bloom filter to make the separation more accurate than any heuristics. As claimed by the author in [27], the yes-no Bloom filter as introduced outperforms the standard Bloom filter even if the naive method is used when picking up false positives, whereas the separation provided by the ADP model will definitely improve the current technology of Bloom filter designs. The ADP algorithm as proposed can also be extended to other applications, because the considered problem is similar to Multidimensional Knapsack problems, and the latter has many applications in a number of areas.

As future work, we shall consider multiple no filters. Another, more challenging direction of future work is to evolve the yes-no Bloom filter into a more complicated data structure which would contain several yes-no parts consecutively. Either of these modifications will improve the accuracy but will make it necessary to consider more complicated optimization problems. All in all, there are a number of directions of interesting future work that we can do on this topic as we develop the concept of a yes-no Bloom filter.

References

[1] D. P. Bertsekas, *Dynamic Programming and Optimal Control: Approximate Dynamic Programming*, Athena Scientific, 2012.

[2] D. Bertsimas and R. Demir, *An approximate dynamic programming approach to multidimensional knapsack problems*, Management Science, Vol.48, No.4 (2002), pp. 550–565.

[3] B. H. Bloom, *Space/time trade-offs in hash coding with allowable errors*, Communications of the ACM, 13(7) (1970), p. 422426.

[4] A. Broder and M. Mitzenmacher, *Network applications of bloom filters: A survey*, Internet Mathematics, (2002), pp. 636–646.

[5] J. Bruck, J. Gao, and A. Jiang, *Weighted bloom filters*, IEEE int'l Symp. on Information Theory, (2006).

[6] L. Carrea, A. Vernitski, and M. Reed, *Optimized hash for network path encoding with minimized false positives*, Computer Networks, 58 (2014), pp. 180–191.

[7] H. Chen, H. Jin, L. Chen, Y. Liu, and L. M. Ni, *Optimizing bloom filter settings in peer-to-peer multi-keyword searching*, IEEE Transactions on Knowledge & Data Engineering, vol.24 Issue No.04 (2012), pp. 692–706.

[8] L. Dong and K. Glazebrook, *An approximate dynamic programing approach to the development of heuristics for the scheduling of impatient jobs in a clearing system*, Naval Research Logistics, Volume 57, Issue 3 (2010), pp. 225–236.

[9] B. Donnet, B. Baynat, and T. Friedman, *Retouched bloom filters: allowing networked applications to trade off selected false positives against false negatives*, in Proceedings of the 2006 ACM CoNEXT conference, ACM, 2006, p. 13.

[10] A. Fréville, *The multidimensional 0-1 kanpsack problem: An overview*, EJOR, 155 (2004), pp. 1–21.

[11] M. P. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness (Series of Books in the Mathematical Sciences)*, 1979.

[12] A. Grothey and X. Yang, *Approximate dynamic programming with bzier curves/surfaces for top-percentile traffic routing*, European Journal of Operational Research, Volume 218, Issue 3 (2012), pp. 698–707.

[13] D. Guo, Y. Liu, X. Li, and P. Yang, *False negative problem of counting bloom filter*, Knowledge and Data Engineering, IEEE Transactions on, 22 (2010), pp. 651–664.

[14] F. Hao, M. Kodialam, and T. Lakshman, *Building high accuracy bloom filters using partitioned hashing*, ACM SIGMETRICS Performance Evaluation Review, 35 (2007), pp. 277–288.

[15] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*, Springer, 2004.

[16] A. Kirsch and M. Mitzenmacher, *Less hashing, same performance: Building a better bloom filter*, Random Structures & Algorithms, 33 (2008), pp. 187–218.

[17] J. Lorie and L. Savage, *Three problems in capital rationing*, Journal of Business, 28 (1955), pp. 229–239.

[18] S. Lumetta and M. Mitzenmacher, *Using the power of two choices to improve bloom filters*, Internet Mathematics, 4 (2007), pp. 17–33.

[19] S. Martello and P. Toth, *Knapsack problems: Algorithms and computer interpretations*, Wiley-Interscience, 1990.

[20] M. Mitzenmacher, *Compressed bloom filters*, IEEEACM TRANSACTIONS ON NETWORKING, VOL. 10, NO. 5 (2002), pp. 604–612.

[21] G. Nemhauser and L. Wolsey, *Integer and Combinatorial Optimization*, Wiley Interscience Series in Discrete Mathematics and Optimization, Wiley, New York, 1988.

[22] R. Pagh and F. F. Rodler, *Lossy dictionaries*, ESA 01: Proceedings of the 9th Annual European Symposium on Algorithms, (2001), p. 300311.

[23] W. Powell, *Approximate Dynamic Programming - Solving the Curses of Dimensionality*, John Wiley & Suns, New Jersey, 2007.

[24] W. Pugh, *Skip lists: A probabilistic alternative to balanced trees*, Lecture Notes in Computer Science, Volume 382 (1989), pp. 437–449.

[25] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, *Theory and practice of bloom filters for distributed systems*, Communications Surveys & Tutorials, IEEE, 14 (2012), pp. 131–155.

[26] S. Tarkoma, C. E. Rothenbergand, and E. Lagerspetz, *Theory and practice of bloom filters for distributed systems*, IEEE Communications Surveys and Tutorials, 14(1) (2012), pp. 131–155.

[27] A. Vernitski, *The yes-no bloom filter: representing sets with fewer false positives.* http://repository.essex.ac.uk/12359/, 2015.

[28] A. Vernitski, L. Carrea, and M. Reed, *Yes-no bloom filter: A way of representing sets with fewer false positives for in-packet path encoding.* submitted, 2015.

[29] W. L. Winston, *Operations Research: Applications and Algorithms (foruth edition)*, Brooks/Cole, 2003.

[30] X. Yang and A. Grothey, *Solving the top-percentile traffic routing problem by approximate dynamic programming*, IMA Journal of Management Mathematics, 23(4) (2012), pp. 413–434.

[31] X. Yang, A. K. Strauss, C. S. M. Currieb, and R. Eglese, *Choice-based demand management and vehicle routing in e-fulfilment*, Transportaion Science, (2014).

[32] D. Zhang and D. Adelmany, *An approximate dynamic programming approach to network revenue management with customer choice*, Transportation Science, (2009), pp. 381–394.

[33] M. Zhong, P. Lu, K. Shen, and J. Seiferas, *Optimizing data popularity conscious bloom filters*, Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing, (2008), pp. 355–364.