Counting and Sampling Subsets Using Block Covers

Elias Lehtinen **Mikko Koivisto**

Department of Computer Science, University of Helsinki, Finland

Abstract

We present a new data structure, *block cover*, for approximate weighted counting and sampling of subsets of a given query set. Such queries are the main computational bottleneck, e.g., in advanced Markov chain Monte Carlo algorithms for Bayesian learning of Bayesian networks. Given a collection of weighted subsets of a ground set N, our key idea is to select a few moderate-size subsets B of N, called blocks, so as to cover all or most of the input collection by the power sets 2^B. An approximate sum over the subsets of a query set Q is obtained by adding up the contributions within each block, which contributions we precompute. We also give a similar, efficient algorithm for generating a subset of Q with probability proportional to its weight. Our empirical results suggest that block covers are superior to previous approaches, which consider the subsets of interest one by one.

1 INTRODUCTION

The protagonist of this paper is the following problem. Suppose we are given a collection $\mathcal{C}$ of subsets of a finite ground set N, each member $S \in \mathcal{C}$ associated with a nonnegative weight $w(S)$; we define $w(S) := 0$ if $S \notin \mathcal{C}$. Our task is to construct a data structure so as to support efficient answering of *subset counting* and *subset sampling* queries for any given query set $Q \subseteq N$. In the former, we ask the sum

$$W(Q) = \sum_{S \subseteq Q} w(S)\,. \tag{1}$$

In the latter, we are required to draw a set $S \subseteq Q$ according to this partition function, i.e., with probability $w(S)/W(Q)$. We will allow approximations that, for a given error tolerance $\delta > 0$, guarantee that the returned sum has a relative error at most δ and that the sampling distribution is within a variation distance at most δ from the exact distribution.

Previous works on the problem have been motivated by applications in Bayesian learning of Bayesian networks. Friedman and Koller (2003) presented a Markov chain Monte Carlo (MCMC) method that simulates a random walk over the orderings of the network nodes. Its main computational bottleneck is the computation of the (posterior) probability of a node ordering, which amounts to answering a subset counting query for each node i with its own collection of subsets $\mathcal{C}_i$ and weight function w_i, and with a query set Q_i that consists of the predecessors of i in the node ordering. Importantly, in this application each $\mathcal{C}_i$ consists of the possible parent sets of node i, which can be assumed be relatively small, say of size at most 6, even if the total number of nodes was in several dozens or hundreds. Several later MCMC methods encounter the same challenge of weighted counting of subsets or close variants of the problem (Grzegorczyk and Husmeier, 2008; Niinimäki et al., 2016; Su and Borsuk, 2016; Viinikka et al., 2020; Nikzad et al., 2025).

Friedman and Koller (2003) gave a simple and effective heuristic for approximate weighted subset counting: As preprocessing, the members of $\mathcal{C}$ are sorted in decreasing order by weight. Given a query set Q, the sorted list is scanned through set by set and the weights of those contained in Q are accumulated, until the weights of the remaining sets are negligible in relation to the already accumulated total weight. Niinimäki and Koivisto (2013) study an optimized variant of this scheme, they dub SORTED, along with its straightforward sampling version. They also present an alternative scheme, TREEDY, which only visits sets contained in Q, however, not strictly in decreasing order by weight. While TREEDY is shown to outperform SORTED in some settings, it was also discovered that neither of the two schemes comes even close to the performance of an idealized scheme, dubbed IDEAL, that would only visit the minimum number of heaviest subsets of Q sufficient for the desired approximation guarantee. IDEAL presents a lower bound for *any* "collector" algorithm that is based on accumulating the weights of individual sets. Intriguingly, (Niinimäki and Koivisto,

Proceedings of the 42nd Conference on Uncertainty in Artificial Intelligence (UAI 2026), PMLR 337:3477–3486.

2013, Sect. 5) mention it as an open problem, *"whether there exist faster algorithms of some different type"*.

Here, we answer the question in the affirmative by giving an algorithm that can even beat IDEAL. This is possible because our algorithm is based on accumulating *precomputed sums of weights*, instead of individual weights.

To introduce our approach, suppose we could precompute and store $W(Q)$ for all $Q \subseteq N$. Then, answering the subset counting query for Q would amount to a single table lookup. Implementing this is infeasible in general, as it would require time and space exponential in $|N|$—far beyond the size of $\mathcal{C}$ (assuming $\mathcal{C}$ only contains small sets). Our key idea is to use multiple small lookup tables, each table storing $W(T)$ only for the subsets T of an appropriately selected small *block* $B \subseteq N$. If every $S \in \mathcal{C}$ is a subset of one of the selected blocks B, then we get $W(Q)$ by summing up $W(Q \cap B)$ over the blocks B, modulo some double counting (which we will avoid by simple arrangements).

In Section 3, we implement this idea by a data structure we call a *block cover*. We also show how subset sampling queries can be efficiently answered using a block cover. We have empirically studied our approach, we call BLOCKS for short, following the previous study of Niinimäki and Koivisto (2013). Comparing BLOCKS to SORTED and IDEAL, we report, in Section 4, favorable results both in the required number of basic operations and in the running times of our C++ implementations.

2 PRELIMINARIES

We begin by reviewing previous approaches to counting and sampling subsets, following mainly (Niinimäki and Koivisto, 2013). To be well prepared for Section 3, we also recall the folklore algorithms for zeta transforms on subset lattices, that is, for efficient computation of the weighted sums over subsets, or equivalently supersets, of Q for all subsets Q of a fixed ground set.

2.1 COLLECTOR ALGORITHMS

Niinimäki and Koivisto (2013) restrict their attention to "collector" algorithms, which answer a counting query by visiting subsets of Q one by one until the accumulated weight sum W' is guaranteed to be within a relative error of δ of the exact sum. Such an algorithm readily gives a way to also answer the corresponding sampling query: draw a uniform random variable U from the range $[0, W']$ and return the first visited set for which the accumulatd weight exceeds U. The sampling distribution can be shown to be within a variation distance δ of the exact distribution (Niinimäki and Koivisto, 2013, Theorem 1).

Ideally, a collector algorithm would only visit sets S that are *relevant*, that is, $S \subseteq Q$ and $S \in \mathcal{C}$. The exact total weight $W(Q)$ only depends on relevant sets. Under relatively mild conditions, one can compute $W(Q)$ by an exact algorithm that traverses through the relevant sets in time that is proportional to their number (multiplied by $|N|$). For example, if $\mathcal{C}$ is closed under inclusion, one can traverse the relevant sets in lexicographical order using membership tests for $\mathcal{C}$. Alternatively, one can represent the set collection $\mathcal{C}$ as a trie for the strings obtained by listing the elements of each set in some fixed total ordering (Rymon, 1992; Knuth, 1998, Sect. 6.3); Savnik et al. (2021) call this data structure a *set-trie*.

Algorithm EXACT

E1 Given a query set Q, visit the relevant sets in lexicographic order and return the sum of their weights.

However, when an approximation to $W(Q)$ is tolerated, an ideal collector algorithm would only visit the heaviest relevant sets to minimize the work:

Algorithm IDEAL

I1 Given a query set Q and tolerance δ, visit the minimum number of the heaviest relevant sets whose total weight W' is at least $(1 - \delta)W$. Return W'.

This algorithm is only a theoretical baseline, as there seems to be no efficient way to implement it in practice. The number of sets visited by IDEAL gives a lower bound for any collector algorithm.

A practical approach, due to (Friedman and Koller, 2003), is to visit the members of $\mathcal{C}$ in decreasing order by weight and stop when sufficient total weight has been accumulated. The following formulation is adapted from Niinimäki and Koivisto (2013):

Algorithm SORTED

S0 Before any queries, sort the sets in $\mathcal{C}$ into decreasing order by weight, $w(S_1) \geq w(S_2) \geq \cdots \geq w(S_m)$. Store the tail sums $T_i := \sum_{j=i+1}^{m} w(S_j)$

S1 Given a query set Q and tolerance δ, initialize $W' := 0$ and iterate through the sets S_i one by one: if $S_i \subseteq Q$, then add $w(S_i)$ to W'; if $W' \geq (1-\delta)(W' + T_i)$, then stop and return W'.

In this formulation, we have not included two optimizations of Niinimäki and Koivisto (2013): (i) If the number of sets visited exceeds that of required by the exact algorithm, then switch to the exact algorithm. (ii) In the stopping rule, the term T_i can lowered to $T_i - T_{i+t}$, where t is the number of relevant sets not yet visited. Both optimizations assume that the number of relevant sets, or a good upper bound for it, can be efficiently computed for any given Q. While this assumption holds when $\mathcal{C}$ consist of all subsets of N of size at most k, it may not hold more generally.

A drawback of SORTED is that, in addition to relevant sets, it can also visit numerous sets that are not contained in the query set Q. In an extreme case, Q is empty and the only relevant set $\emptyset$ is the last set S_m in the sorted list.

Another practical approach, due to Niinimäki and Koivisto (2013), is to visit the subsets of Q in "approximately decreasing" order by weight and, again, stop when sufficient total weight has been accumulated. The TREEDY algorithm implements this idea, using a natural search tree representation of the collection $\mathcal{C}$, by ordering the children of any inner node in the tree into decreasing order by the total weight of the subtree rooted by the node. This ordering, constructed as preprocessing, speficies the order in which the tree is traversed to answer any subset counting query. We refer to Niinimäki and Koivisto (2013, Sect. 3) for details.

Being a collector algorithm, TREEDY necessarily requires at least as much work as IDEAL to answer counting and sampling queries. In the experiments of Niinimäki and Koivisto (2013), TREEDY was often 2 to 5 times faster than SORTED, yet one to two orders of magnitude slower than IDEAL.

Because our study aims at demonstrating superiority over IDEAL, we will not include TREEDY in our experiments. However, the simpler SORTED algorithm will serve as a baseline when the interest is not only in the required number of "basic operations," but also in the actual running times of our algorithm implementations.

2.2 THE FAST ZETA TRANSFORM

Let V be a set of t elements and f a function from the powerset 2^V to real numbers, or more generally to any additive group. Define the functions $\underline{F}$ and $\bar{F}$ by letting

$$\underline{F}(T) := \sum_{\emptyset \subseteq S \subseteq T} f(S) \quad \text{and} \quad \bar{F}(T) := \sum_{T \subseteq S \subseteq V} f(S)$$

for all $T \subseteq V$. Both operators $\underline{\zeta} : f \mapsto \underline{F}$ and $\bar{\zeta} : f \mapsto \bar{F}$ are examples of *zeta transforms* on partial orders, the former on $(2^V, \subseteq)$ and the latter on $(2^V, \supseteq)$, respectively. The inverse operators are called Möbius inversions.

It is a well known fact that, given f, the functions $\underline{F}$ and $\bar{F}$ can each be computed with only $t2^{t-1}$ additions. The algorithm, often called the *fast zeta transform*, is obtained as a special instantiation of Yates's algorithm (Yates, 1937), and has been later rediscovered and applied in various forms and contexts (e.g., Kennes and Smets, 1990; Koivisto, 2006).

3 THE BLOCKS ALGORITHM

This section present our novel block cover data structure. We begin with an outline the data structure in Section 3.1. Then we give our algorithms or answering subset counting and sampling queries in Sections 3.2 and 3.3, respectively. Finally, we give a greedy algorithm for constructing a good block cover in Section 3.4.

3.1 BLOCK COVERS

A *block cover* for the weighted collection $(\mathcal{C}, w)$ is specified by a sequence $B_1, \ldots, B_\ell$ of subsets of N, called *blocks*, such that every member $S \in \mathcal{C}$ is *covered* by some B_i, that is, $S \subseteq B_i$. With each block B_i we associate a corresponding subcollection $\mathcal{C}_i \subseteq \mathcal{C}$, defined recursively by

$$\mathcal{C}_i := \mathcal{C}'_{i-1} \cap 2^{B_i}\,, \text{ with } \mathcal{C}'_i := \mathcal{C} \setminus \left(\mathcal{C}_1 \cup \cdots \cup \mathcal{C}_i\right).$$

Thus the collections $\mathcal{C}_i$ partition $\mathcal{C}$ into disjoint parts.

The block cover equips each block B_i with the lower and upper zeta transforms of the weights:

$$\underline{W}_i := \underline{\zeta}(w_i) \quad \text{and} \quad \bar{W}_i := \bar{\zeta}(w_i)\,,$$

where w_i is defined for all $S \subseteq B_i$ by

$$w_i(S) := \begin{cases} w(S) & \text{if } S \in \mathcal{C}_i\,, \\ 0 & \text{otherwise.} \end{cases}$$

Supposing that the maximum block size is t and that the weight functions w_i are given, the transforms $\underline{W}_i$ and $\bar{W}_i$ for all i can be computed in time $O(2^t t\ell)$.

These constructions enable the following representation, we will employ to answer counting and sampling queries:

Theorem 1. *Let $Q \subseteq N$. For all $j = 0, \ldots, \ell$, we have*

$$W(Q) = \sum_{i=1}^{j} \underline{W}_i(Q \cap B_i) + \sum_{S \subseteq Q : S \in \mathcal{C}'_j} w(S)\,.$$

Proof. Fix j and let $S \subseteq N$. Since the collections $\mathcal{C}_i$ partition $\mathcal{C}$, there is a unique s such that $S \in \mathcal{C}_s$.

If $s > j$, then $S \in \mathcal{C}'_j$. The set S contributes to the second sum exactly when $S \subseteq Q$. It cannot contribute to the first sum, as each $\underline{W}_i(Q \cap B_i)$ is a sum of weights of sets in $\mathcal{C}_i$.

Otherwise, $s \leq j$ and the set does not contribute to the second sum. It contributes $w_s(S) = w(S)$ to $\underline{W}_s(Q \cap B_s)$ exactly when $S \subseteq Q \cap B_s$, equivalently, $S \subseteq Q$. □

In particular, by using all blocks in the cover ($j = \ell$), we can answer any counting query exactly. If using only the $j < \ell$ first blocks, the error is determined by the missing weights that contribute to the latter sum. The upper zeta transforms $\bar{W}_i$ will appear to be useful for answering sampling queries. We give the detailed algorithms and their analyses in the next sections.

See Table 1 for an illustration of a simple block cover with three blocks on a four-element ground set. The collection of sets $\mathcal{C}$ and the weights are the same as in an example of Niinimäki and Koivisto (2013, Table 1). For each set in $\mathcal{C}$, shown is the part $\mathcal{C}_i$ to which it belongs.

Table 1: A block cover on ground set $\{A, B, C, D\}$. For the subset counting query Q, the first two blocks contribute enough weight to guarantee a relative error at most 10%.

Set	Weight	Blocks			Steps
$S \in \mathcal{C}$	$w(S)$	ABD	ACD	BC	$Q = BCD$
AB	99	$\mathcal{C}_1$			
AD	90	$\mathcal{C}_1$			
A	85	$\mathcal{C}_1$			
$\emptyset$	80	$\mathcal{C}_1$			1
B	70	$\mathcal{C}_1$			1
AC	60		$\mathcal{C}_2$		
D	50	$\mathcal{C}_1$			1
BD	14	$\mathcal{C}_1$			1
C	13		$\mathcal{C}_2$		2
CD	12		$\mathcal{C}_2$		2
BC	11			$\mathcal{C}_3$	

3.2 COUNTING

Given a query set Q, our algorithm to answer the counting query applies Theorem 1 in a straighforward manner. For the stopping condition, we adopt the rule of the SORTED algorithm based on precomputed tail sums.

Algorithm BLOCKS

B0 Before any queries, construct a block cover for the weighted collection $(\mathcal{C}, w)$. Store the tail sums $T_j := \sum_{i=j+1}^{\ell} \underline{W}_i(B_i) = \sum_{S \in \mathcal{C}'_j} w(S)$.

B1 Given a query set Q and tolerance δ, initialize $W' := 0$ and iterate through the blocks B_j one by one: add $\underline{W}_j(Q \cap B_j)$ to W'; if $W' \geq (1-\delta)(W' + T_j)$, then stop and return W'.

The approximation guarantee follows rather directly from the construction; we include a proof for completeness.

Theorem 2. BLOCKS *returns a value* $W' \geq (1-\delta)W(Q)$.

Proof. To see that that the algorithm terminates, observe that when $j = \ell$, we have $W' = W(Q)$ (by Theorem 1) and $T_j = 0$. Since $\delta \geq 0$, we have $W' \geq (1-\delta)(W' + T_j)$, and the algorithm stops.

Let W' be the returned value. Let j the smallest index such that $W' \geq (1-\delta)(W' + T_j)$. As $W' = \sum_{i=1}^{j} \underline{W}_i(Q \cap B_i)$ and $T_j = \sum_{S \in \mathcal{C}'_j} w(S)$, we have $W' + T_j \geq W(Q)$ by Theorem 1. Thus $W' \geq (1-\delta)W(Q)$. □

In the example shown in Table 1, we see that BLOCKS only takes two steps to answer a subset counting query to within a relative error of 10%. In comparison, IDEAL would have to collect the weights of the contributing five subsets of Q separately, and SORTED does even more work.

3.3 SAMPLING

To answer a sampling query, our algorithm has three stages. First, the corresponding counting query is answered to get an approximation of W' of $W(Q)$. Next, one of the blocks contributing to the weight sum W' is sampled. Finally, a set contained by the sampled block is generated with probability proportional to its weight. We present the steps in more in algorithms DRAW and GENERATE below.

Algorithm DRAW

D1 Given a query set Q and tolerance δ, compute an approximate sum W' using BLOCKS.

D2 Draw a uniform random variable U from $[0, W']$.

D3 Find the smallest u such that $\sum_{i=1}^{u} \underline{W}_i(Q \cap B_i) \geq U$.

D4 Draw and return an $S \subseteq Q \cap B_u$ with probability proportional to $w_u(S)$.

The total variation distance between probability distributions p and q is given by $d_{\mathrm{TV}}(p, q) = \frac{1}{2} \sum_x |p(x) - q(x)|$, where x runs through the elements of the sample space.

Theorem 3. DRAW *samples from a distribution whose total variation distance from the exact distribution is at most* δ.

Proof. Let p denote the exact distribution and q the approximate probability distribution the algorithm samples from. Let $\sum_{i=1}^{j} \underline{W}_i(Q \cap B_i) = W'$. As $q(S) = p(S)W(Q)/W'$ for all $S \in \mathcal{C}^*_j := \mathcal{C} \setminus \mathcal{C}'_j$ and $q(S) = 0$ for $S \in \mathcal{C}'_j$, we have

$$\begin{aligned} 2\, d_{\mathrm{TV}}(p, q) &= \sum_{S \subseteq Q} |p(S) - q(S)| \\ &= \sum_{S \subseteq Q : S \in \mathcal{C}^*_j} [q(S) - p(S)] + \sum_{S \subseteq Q : S \in \mathcal{C}'_j} p(S) \\ &= \sum_{S \subseteq Q : S \in \mathcal{C}^*_j} q(S) + 1 - 2 \sum_{S \subseteq Q : S \in \mathcal{C}^*_j} p(S) \\ &= 2 - 2W'/W(Q)\,. \end{aligned}$$

Using $W' \geq (1-\delta)W(Q)$ completes the proof. □

The above description of DRAW does not specify how step D4 is implemented. A naive implementation would draw a uniform random variable U from $[0, \underline{W}_u(Q \cap B_u)]$, scan over the subsets S of $Q \cap B_u$ in some prespecified order until the accumulated sum of the weights $w_u(S)$ is greater or equal to U, and finally return the last visited S. In the worst case, this requires visiting all the covered sets.

We next give a significantly faster algorithm for drawing a random subset S of $Q \cap B_u$ according to its weight $w_u(S)$. In fact, our algorithm solves the following more general problem. Given nonnegative weights $f(S)$ for all subsets S of a ground set V and subsets $A, B \subseteq V$ with $A \subseteq B$, draw

a set S from the interval $[A, B] := \{S : A \subseteq S \subseteq B\}$ with probability proportional to $f(S)$.

The idea of the algorithm is to partition the interval sum

$$f[A, B] := \sum_{A \subseteq S \subseteq B} f(S)$$

into two parts based on any element $x \in B \setminus A$:

$$f[A, B] = f[A \cup \{x\}, B] + f[A, B \setminus \{x\}] .$$

With the corresponding probabilities, we include x in the set S or leave it out. This is repeated for the selected smaller interval until every element in $B \setminus A$ is either included in S or left out. More precisely, the procedure is as follows:

Algorithm GENERATE

G1 Given sets A, B, let $R := B \setminus A$ and $Z := f[A, B]$.

G2 If R is empty, return A; else remove an x from R.

G3 Let $Z' := f[A \cup \{x\}, B]$.

G4 With probability Z'/Z, add x to A and let $Z := Z'$; otherwise, remove x from B and let $Z := Z - Z'$.

G5 Go to step G2.

The correctness of this algorithm is shown by a routine proof (omitted). The time requirement is only moderately exponential in $|V|$:

Theorem 4. GENERATE *can be implemented to run in time* $O(2^{|V|/3})$*, assuming access to the lower and upper zeta transforms* $\underline{F}$ *and* $\bar{F}$ *of* f.

Proof. The algorithm iterates the steps G2–G5 at most $|V|$ times. In each iteration, the time requirement is dominated by the computation of $f[A \cup \{x\}, B]$.

We can compute $f[A, B]$ in three ways:

$$\sum_{I \subseteq A} (-1)^{|I|} \underline{F}(B \setminus I) = f[A, B] = \sum_{I \subseteq V \setminus B} (-1)^{|I|} \bar{F}(A \cup I)$$

To see that the latter identity holds (similarly the former), write the right-hand-side sum as

$$\begin{aligned} \sum_{I \subseteq V \setminus B} (-1)^{|I|} \sum_{S \supseteq A \cup I} f(S) &= \sum_{S \supseteq A} f(S) \sum_{I \subseteq V \setminus B} (-1)^{|I|} \mathbf{1}_{S \supseteq I} \\ &= \sum_{A \subseteq S} f(S) \sum_{I \subseteq S \cap (V \setminus B)} (-1)^{|I|} , \end{aligned}$$

and observe that the inner sum equals 1 if $S \cap (V \setminus B)$ is empty, i.e., $S \subseteq B$, and vanishes otherwise.

Thus, $f[A, B]$ can be computed in time $O(2^d)$, where d is the smallest of $|A|$, $|B \setminus A|$, and $|V \setminus B|$. Let these set sizes in the i-th iteration be a_i, r_i, and e_i, respectively, and let $d_i := \min\{a_i, r_i, e_i\}$. Clearly, the worst-case initial values are $a_0 = e_0 = 0$ and $r_0 = |V|$. The invariant $a_i + e_i = i$ implies $d_i \leq i/2$. Since $r_i = n - i$, we have $d_i \leq \min\{i/2, n - i\}$. It follows that $\sum_i 2^{d_i} = O(2^{|V|/3})$. □

Corollary 5. *Step D4 of* DRAW *can be implemented to run in time* $O(2^{t/3})$*, where* t *is the maximum block size.*

Even if the time requirement is exponential in the block size, it is not expected to dominate the complexity in practice. This is because the maximum block size t is limited to at most, say, 15, rendering $2^{t/3}$ at most 32.

3.4 GREEDY CONSTRUCTION

So far, we have worked with a given block cover, along with the associated zeta transforms, for the weighted collection $(\mathcal{C}, w)$. Now, we turn to the task of constructing a good block cover. Considering the presented algorithms, we wish to find a block cover in which (i) the blocks are relatively small, (ii) their number is small, and (iii) the total weight covered by the first blocks is large. We will heuristically pursue a good tradeoff between these goals, rather than making any attempt to formulate a well-defined objective function.

For the sake of exposition, consider the regular setting where $\mathcal{C}$ consists of all subsets of N that have at most k elements, with a fixed k. If all blocks in our cover were of size k, we would need $\binom{n}{k}$ blocks. On the other hand, a single block would suffice if it is the ground set N itself. But how many blocks are needed if they all are of size t?

This question has been studied in combinatorics, where an object corresponding to our block cover is called an (n, t, k)-*covering design*. Erdős and Spencer (1974) showed that

$$\left\{1 + \ln \binom{t}{k}\right\} \binom{n}{k} \Big/ \binom{t}{k}$$

blocks suffice. This is within a factor of t of the obvious lower bound $\binom{n}{k}/\binom{t}{k}$. Furthermore, Godbole and Janson (1996) showed that ℓ blocks of size t, selected at random, cover the k-sets with high probability, supposing $t \leq 2k - 1$ and ℓ is about $\ln \binom{n}{k}$ times the lower bound, or larger. Moreover, a simple greedy construction has been observed to find covering designs, for small values of k and t, of sizes within a small constant factor from the optimal (Gordon et al., 1995). Taken together, these results suggest that good block covers can be constructed by relatively simple means.

For us, also the parameter t, the maximum block size, is a free parameter we wish to optimize. To this end, we minimize the memory requirement of the block cover, which scales as $2^t \ell$, supposing all ℓ blocks are of size t. Substituting the lower bound $\binom{n}{k}/\binom{t}{k}$ for ℓ, the function to be minimized becomes $2^t/\binom{t}{k}$. This is minimized by $t = 2k$.

To take into account the weights of the sets in $\mathcal{C}$, we construct the blocks in a greedy fashion: the first block B_1 is formed by the heaviest sets, block B_2 is formed by the heaviest of the remaining sets not covered by B_1, and so forth. As this greedy algorithm is not guaranteed to find a small block cover, we allow the algorithm to stop when the memory

requirement of the blocks exceeds a prespecified budget M. The returned blocks $B_1, \ldots, B_\ell$ may thus leave some of the sets in $\mathcal{C}$ uncovered. The algorithms given in the previous sections are extended to this setting in a straightforward manner by continuing counting or sampling after the last block using SORTED, if needed.

More precisely, our greedy algorithm for constructing a block cover for $(\mathcal{C}, w)$, subject to the user parameters $t \geq 1$ and $M \geq 2^t$, is as follows.

Algorithm CONSTRUCT

C1 Sort $\mathcal{C}$ into a list $\mathcal{L}$ in decreasing order by weight, $w(S_1) \geq \cdots \geq w(S_m)$. Let $\ell := 0$.

C2 Let $\ell := \ell + 1$. Let block B_ℓ be the union of the largest number of first sets in $\mathcal{L}$ such that $|B_\ell| \leq t$.

C3 Move all sets contained in B_ℓ from $\mathcal{L}$ to $\mathcal{C}_\ell$.

C4 If $\sum_i 2^{|B_i|} > M$, return $(B_i, \mathcal{C}_i)_i$. Else go to step C2.

See Table 1 for an illustration of a block cover produced by CONSTRUCT, with the maximum block size $t = 3$.

Theorem 6. CONSTRUCT *can be implemented to run in time* $O(m \log m + Mt)$.

Proof. Step C1 can be implemented in time $O(m \log m)$ using, e.g., merge sort.

Per block B_i, step C2 takes time $O(2^{|B_i|}t)$, as each considered set union has at most $2^{|B_i|}$ subsets and computing the union of two sets of t or fewer elements takes time $O(t)$, assuming the sets are represented as ordered lists.

Per block B_i, step C3 can also be implemented in time $O(2^{|B_i|}t)$ by using the set-trie representation of $\mathcal{C}$ to traverse the sets in $\mathcal{C} \cap 2^{B_i}$; in the trie, each set is assigned the index of the block to which the set belongs, initialized to 0. Each visited subset of B_i whose assigned index is 0 is added to $\mathcal{C}_i$ and the assigned index is updated to i. For efficiency, the sets are not removed from the list $\mathcal{L}$; instead, in step C2, sets that are assigned a nonzero index are simply skipped.[1]

Per block B_i, step C4 takes $O(m)$ time when the algorithm terminates, and $O(1)$ otherwise, as the value of $\sum_i 2^{|B_i|}$ can be trivially updated after the selection of the block B_i.

In total, steps C2–C4 take time $O(\sum_i 2^{|B_i|}t)$. This is $O(Mt)$, since $\sum_{i=1}^{\ell-1} 2^{|B_i|} \leq M$ and $2^{|B_\ell|} \leq 2^t \leq M$. □

Note that CONSTRUCT only selects the blocks. Computing the needed zeta transforms takes an additional $O(Mt)$ time.

[1]This implementation allows a simple array representation of $\mathcal{L}$. Alternatively, one could represent $\mathcal{L}$ as a doubly linked list, from which elements (here sets) can be removed in constant time.

4 EXPERIMENTS

We next report on experimental results comparing our block cover based approach to the mentioned other approaches to solve subset counting queries. We do not include subset sampling queries in our study, as they are expected to be dominated by the corresponding subset counting queries.

We have implemented the algorithms and data structures in C++, but we did not aim to fully optimize the implementation. Therefore, we here focus on gauging the number of basic operations executed to answer the query in question, rather than running times, which are sensitive to implementation choices. In more precise terms, we take as the *number of operations* the number of sets visited by the algorithm. We will return to discussing this matter in Section 5.

Throughout our study, we vary the error tolerance parameter from 10^{-6} to 10^{-1}. Per possible query set size, we generate 1 000 query sets uniformly at random.

The source code is available at `https://github.com/sums-of-products/blocks`. The experiments were run as single-thread processes on a machine with an Intel i5-11400F CPU (4.40 GHz Turbo) and 16 GB of RAM.

4.1 SYNTHETIC INSTANCES

Following Niinimäki and Koivisto (2013), we generated synthetic weighted set collections $(\mathcal{C}, w)$ as follows. We varied the size of the ground set n in $\{20, 60\}$ and the maximum sets size k in $\{3, 5\}$. We let the collection $\mathcal{C}$ consist of all subsets of $\{1, \ldots, n\}$ of size at most k, and associate each $S \in \mathcal{C}$ with a random weight

$$w(S) = \exp\left\{\lambda \sum_{x \in S} U_x\right\}, \quad U_x \sim \text{Uniform}(\kappa - 1, \kappa)\,,$$

where the elementwise random variables U_x are independent. By varying the parameters λ and κ we formed four classes of synthetic instances:

flat: $\lambda = 10$, $\kappa = k/n$.

steep: $\lambda = 200$, $\kappa = k/n$.

mixture: The weight is obtained as a sum of 5 flat and 5 steep ones, for both the κ ranging over the 5 values $(k-1)/n, k/n, \ldots, (k+3)/n$. This introduces 10 local maxima to the weight function.

shuffled: The same as in mixture, but permuting the weights among sets of same size. This removes the pairwise dependence of the weights of sets that contain many same elements.

Figure 1 shows how the number of operations required by the tested algorithm depends on the error tolerance and varies across the 16 instances. We observe that BLOCKS

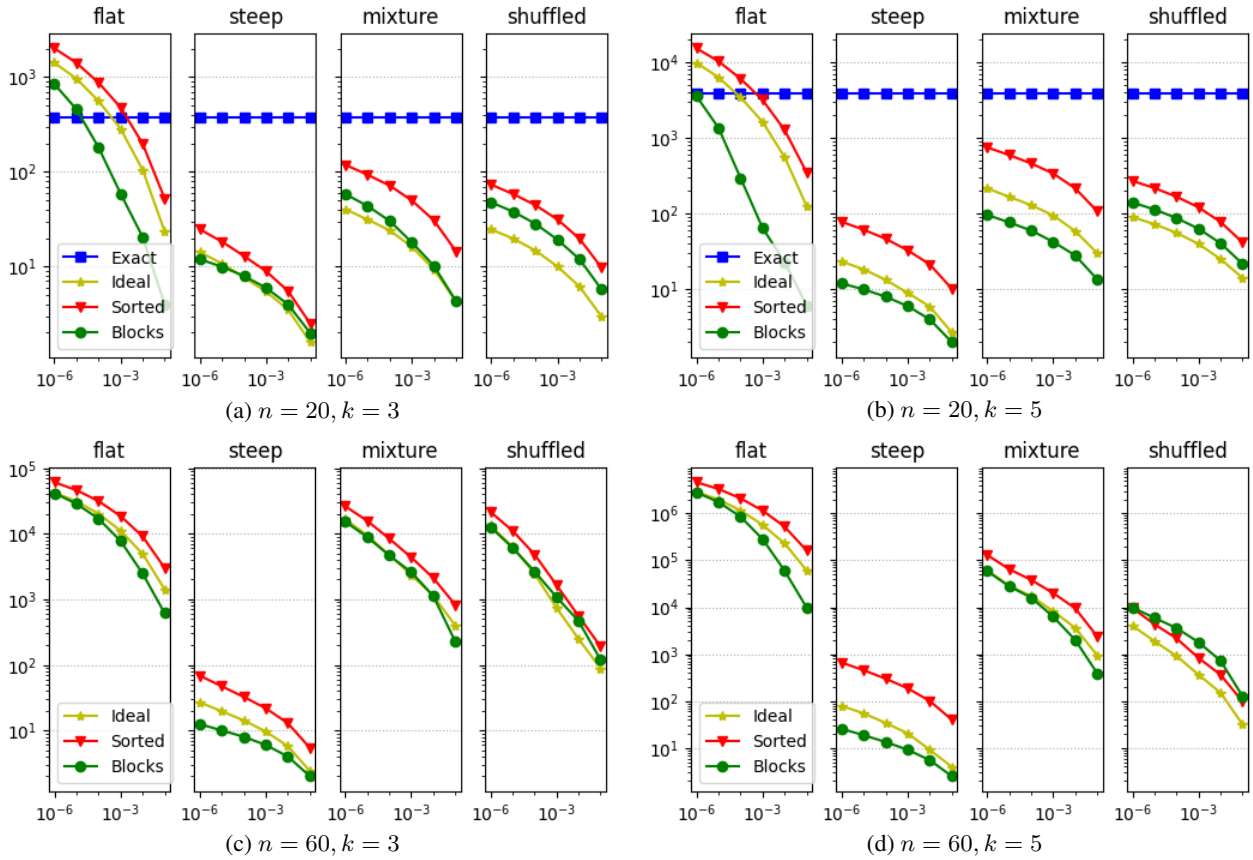

Figure 1: **Number of operations** as a function of error tolerance for synthetic instances. Shown values are averages over 1 000 random subset counting queries.

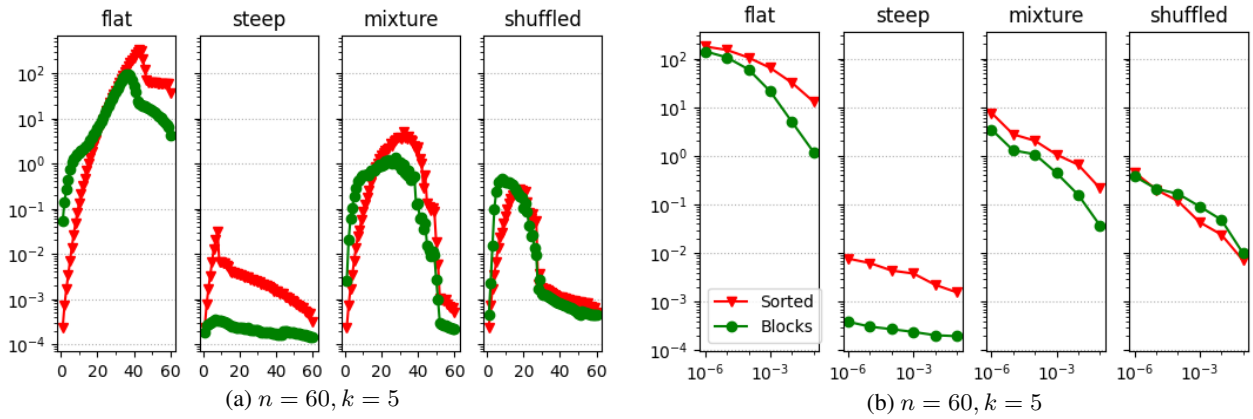

Figure 2: **Running time in milliseconds** as a function of (a) query set size and (b) error tolerance for synthetic instances. Shown values are averages over 1 000 random subset counting queries.

beats IDEAL and SORTED in all instances, except on the four instances of shuffled; the shuffled instances destroy, by design, a natural structure of a weight function that would concentrate high weights onto similar subsets, which is favorable for BLOCKS. The advantage of BLOCKS over SORTED varies up to a factor of around 30, achieved for flat ($n = 20, k = 5$) and steep ($n = 60, k = 5$).

The difference in the required number of operations is also reflected in the running times (Fig. 2(b)). Here we do not include IDEAL due to the nature of the algorithm.[2] We also

[2]For example, simply adding up the weights of a precomputing a list of relevant sets could misleadingly favor IDEAL, by enabling straightforward vectorization of the arithmetic operations.

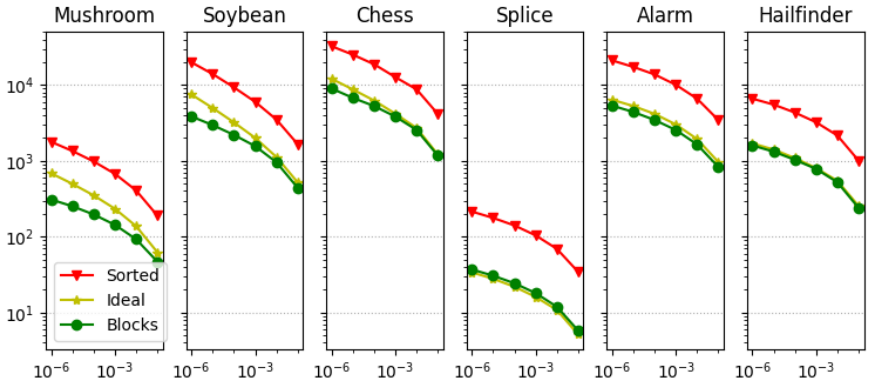

Figure 3: **Number of operations** as a function of error tolerance for benchmark datasets. Shown values are averages over 1 000 random subset counting queries.

observe that the time requirement of BLOCKS and SORTED are heavily dominated by a relative narrow range of query sizes (Fig. 2(a)). The hardest query size and the location of the peak, however, varies across the instance types.

4.2 BENCHMARK DATASETS

We also experiments with benchmark datasets (Table 2). From the UCI Machine Learning Repository (Dua and Graff, 2017), we included all the four datasets with 100 to 10 000 data points (samples) over 20 to 64 discrete variables. In addition, we generated 1 000 data points from the benchmark Bayesian networks Alarm and Hailfinder obtained from the Bayesian network repository (Scutari, 2022).

From each benchmark dataset, we computed the *BDeu local score* (Buntine, 1991; Heckerman et al., 1995) for each of the n' variables, for all possible subsets S of the remaining $n'-1$ variables, of size $|S|$ at most k; see Table 2 for the values of k. We set the weight $w(S)$ to the score of S (represented in the probability scale, not as its lograrithm). Thus, per benchmark dataset, we have n weight functions (instances), each with a ground set of size $n := n'-1$. Weight functions stemming from BDeu scores is one of the main application domains subset counting and sampling queries in the context of Bayesian network learning.

Table 2: Key statistics of the benchmark datasets.

Name	n'	Samples	k
Mushroom	21	5644	5
Soybean	35	266	5
Chess	36	3196	5
Splice	61	3190	4
Alarm	37	1000	5
Hailfinder	56	1000	4

We observe that, consistently across all benchmark datasets, BLOCKS is almost an order-of-magnitude more efficient than SORTED (Fig. 3). Compared to the results on synthetic instances, the required number of operations is less sensitive to the error tolerance, yet the gain is about one order of magnitude if relaxing the error tolerance from 10^{-6} to 10^{-1}.

5 CONCLUDING REMARKS

We have presented a new method for computing large weighted sums over subsets of a given query set and for sampling random subsets from the corresponding distribution. Unlike prior approaches that examine subsets individually, our method leverages precomputed "block sums." The amount of required additional memory can be controlled by the design parameters of the block cover data structure. Our experiments, with a memory budget set to about twice the input size, i.e., the size of the given weighted set family, showed significant gains in the speed of answering subset sum queries.

As a side result of likely independent intrerest, we presented a nontrivial algorithm for sampling from a given set interval. Our algorithm runs in time $O(2^{t/3})$ when the underlying set has t elements, only assuming that the lower and upper zeta transforms have been precomputed, thus requiring not more than linear $O(2^t)$ storage size.

There are interesting directions for future research. First, in our experiments we sampled query sets uniformly at random conditionally on the set size. In applications, such as in MCMC methods for learning Bayesian networks, the sampling distribution can be far from uniform. Even if the previous study of Niinimäki and Koivisto (2013) did not observe any significance qualitative differences in the empirical results between the two settings, nonuniform distributions warrant furher investigations. Specifically, it could be advantageous to select and order the blocks based on what is known (or expected) about the query distribution. A

related question is whether a more sophisticated construction algorithm could find block covers that are significantly better than those found by the simple greedy procedure.

Another direction is to examine efficient implementation of block covers using methods of algorithm engineering (Leiserson et al., 2020; Meyer et al., 2003). Namely, we believe our implementation of BLOCKS, as well as SORTED, can be made significantly faster by paying attention to the cost of arithmetic operations and the cost of checking the stopping condition. Supposing the cost of operations per visited set (or block) can be subtantially reduced, the overall computational cost may become dominated by memomy accessing. Such a shift could favor SORTED over BLOCKS, since BLOCKS accesses scattered memory locations, whereas SORTED benefits from linear accessing. That would motivate seeking implementions in which frequently accessed items are usually stored in the first levels of cache hierarchy.

While our main motivation for the studied counting and sampling problems stems for applications in Bayesian learning of Bayesian networks, the problems may arise also in other domains. For example, Iyer and Bilmes (2015) consider probabilistic models based on (log)-submodular (or supermodular) set functions and deal with sums over subsets similar to our work. Certain *decision problems* related to subset counting have been studied from a theoretical viewpoint (Charikar et al., 2002); if counting or sampling variants of those problems are of interest, the ideas of present work could be useful.

Acknowledgements

We thank Juha Harviainen for preliminary discussions of the some of the ideas studied in this work. The research was partially supported by the Research Council of Finland, grant 351156.

References

Wray L. Buntine. Theory refinement on Bayesian networks. In *The Seventh Annual Conference on Uncertainty in Artificial Intelligence, UAI 1991*, pages 52–60. Morgan Kaufmann, 1991.

Moses Charikar, Piotr Indyk, and Rina Panigrahy. New algorithms for subset query, partial match, orthogonal range searching, and related problems. In *International Colloquium on Automata, Languages, and Programming, ICALP 2002*, volume 2380 of *Lecture Notes in Computer Science*, pages 451–462. Springer, 2002.

Dheeru Dua and Casey Graff. UCI Machine Learning Repository. `http://archive.ics.uci.edu/ml`, 2017. University of California, Irvine, School of Information and Computer Sciences.

Paul Erdős and Joel Spencer. *Probabilistic Methods in Combinatorics*. Academic Press, 1974.

Nir Friedman and Daphne Koller. Being Bayesian about network structure. A Bayesian approach to structure discovery in Bayesian networks. *Mach. Learn.*, 50:95–125, 2003.

Anant P. Godbole and Svante Janson. Random covering designs. *Journal of Combinatorial Theory, Series A*, 75 (1):85–98, 1996.

Daniel M. Gordon, Oren Patashnik, and Greg Kuperberg. New constructions for covering designs. *Journal of Combinatorial Designs*, 3(4):269–284, 1995.

Marco Grzegorczyk and Dirk Husmeier. Improving the structure MCMC sampler for Bayesian networks by introducing a new edge reversal move. *Mach. Learn.*, 71: 265–305, 2008.

David Heckerman, Dan Geiger, and David M. Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. *Mach. Learn.*, 20:197–243, 1995.

Rishabh K. Iyer and Jeff A. Bilmes. Submodular point processes with applications to machine learning. In *The Eighteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2015*, volume 38 of *Proceedings of Machine Learning Research*, pages 388–397. PMLR, 2015.

Robert Kennes and Philippe Smets. Computational aspects of the Mobius transformation. In *The Sixth Annual Conference on Uncertainty in Artificial Intelligence, UAI 1990*, pages 401–416. Elsevier, 1990.

Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Professional, Reading, Massachusetts, 2nd edition, 1998.

Mikko Koivisto. Advances in exact Bayesian structure discovery in Bayesian networks. In *The Twenty-Second Conference on Uncertainty in Artificial Intelligence, UAI 2006*, pages 241–248. AUAI Press, 2006.

Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. There's plenty of room at the top: What will drive computer performance after Moore's law? *Science*, 368(6495):eaam9744, 2020.

Ulrich Meyer, Peter Sanders, and Jop Sibeyn, editors. *Algorithms for Memory Hierarchies: Advanced Lectures*, volume 2625 of *Lecture Notes in Computer Science*. Springer, 2003.

Teppo Niinimäki and Mikko Koivisto. Treedy: A heuristic for counting and sampling subsets. In *The Twenty-Ninth Conference on Uncertainty in Artificial Intelligence, UAI 2013*, pages 469–477. AUAI Press, 2013.

Teppo Niinimäki, Pekka Parviainen, and Mikko Koivisto. Structure discovery in Bayesian networks by sampling partial orders. *J. Mach. Learn. Res.*, 17:1–47, 2016.

Daniele Nikzad, Alexander Zhilkin, Juha Harviainen, Jack Kuipers, Giusi Moffa, and Mikko Koivisto. Scaling up Bayesian DAG sampling, 2025. arXiv:2510.25254 [cs.LG], to appear in UAI 2026.

Ron Rymon. Search through systematic set enumeration. In *The Third International Conference on Principles of Knowledge Representation and Reasoning, KR 1992*, page 539–550. Morgan Kaufmann Publishers Inc., 1992.

Iztok Savnik, Mikita Akulich, Matjaž Krnc, and Riste Škrekovski. Data structure set-trie for storing and querying sets: Theoretical and empirical analysis. *PLOS ONE*, 16(2):e0245122, 2021.

Marco Scutari. Bayesian Network Repository, 2022. URL `www.bnlearn.com/bnrepository`. Accessed: 2025-02-24.

Chengwei Su and Mark E. Borsuk. Improving structure MCMC for Bayesian networks through Markov blanket resampling. *J. Mach. Learn. Res.*, 17(118):1–20, 2016.

Jussi Viinikka, Antti Hyttinen, Johan Pensar, and Mikko Koivisto. Towards scalable Bayesian learning of causal dags. In *Advances in Neural Information Processing Systems 33, NeurIPS 2020*, pages 6584–6594. Curran Associates, Inc., 2020.

Frank Yates. *The Design and Analysis of Factorial Experiments*. Imperial Bureau of Soil Science, Harpenden, England, 1937.