View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

A Faster FPTAS for #Knapsack

Paweł Gawrychowski
Institute of Computer Science, University of Wrocław, Poland
gawry@cs.uni.wroc.pl

Liran Markin[1]
University of Haifa, Israel
liran.markin@gmail.com

Oren Weimann[2]
University of Haifa, Israel
oren@cs.haifa.ac.il

Abstract

Given a set $W = \{w_1, \ldots, w_n\}$ of non-negative integer weights and an integer C, the #Knapsack problem asks to count the number of distinct subsets of W whose total weight is at most C. In the more general integer version of the problem, the subsets are multisets. That is, we are also given a set $\{u_1, \ldots, u_n\}$ and we are allowed to take up to u_i items of weight w_i.

We present a deterministic FPTAS for #Knapsack running in $O(n^{2.5}\varepsilon^{-1.5}\log(n\varepsilon^{-1})\log(n\varepsilon))$ time. The previous best deterministic algorithm [FOCS 2011] runs in $O(n^3\varepsilon^{-1}\log(n\varepsilon^{-1}))$ time (see also [ESA 2014] for a logarithmic factor improvement). The previous best randomized algorithm [STOC 2003] runs in $O(n^{2.5}\sqrt{\log(n\varepsilon^{-1})} + \varepsilon^{-2}n^2)$ time. Therefore, for the case of constant ε, we close the gap between the $\tilde{O}(n^{2.5})$ randomized algorithm and the $\tilde{O}(n^3)$ deterministic algorithm.

For the integer version with $U = \max_i\{u_i\}$, we present a deterministic FPTAS running in $O(n^{2.5}\varepsilon^{-1.5}\log(n\varepsilon^{-1}\log U)\log(n\varepsilon)\log^2 U)$ time. The previous best deterministic algorithm [TCS 2016] runs in $O(n^3\varepsilon^{-1}\log(n\varepsilon^{-1}\log U)\log^2 U)$ time.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases knapsack, approximate counting, K-approximating sets and functions

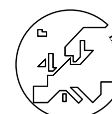
Digital Object Identifier 10.4230/LIPIcs.ICALP.2018.64

1 Introduction

Given a set $W = \{w_1, \ldots, w_n\}$ of non-negative integer weights and an integer C, the #Knapsack problem asks to count the number of distinct subsets of W whose total weight is at most C. This problem is the counting version of the well known Knapsack problem and is #P-hard. While there are many, celebrated, randomized polynomial-time algorithms for approximately counting #P-hard problems, the #Knapsack problem is one of the few examples where there is also a deterministic approximation algorithm (other notable examples are [8, 14, 1]).

From a geometric view, the #Knapsack problem is equivalent to finding the number of vertices of the n-dimensional hypercube that lie on one side of a given n-dimensional hyperplane. The problem is also related to pseudorandom generators for halfspaces (see

[1] Supported in part by Israel Science Foundation grant 592/17
[2] Supported in part by Israel Science Foundation grant 592/17

© Paweł Gawrychowski, Liran Markin, and Oren Weimann;
licensed under Creative Commons License CC-BY
45th International Colloquium on Automata, Languages, and Programming (ICALP 2018).
Editors: Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella;
Article No. 64; pp. 64:1–64:13


Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

e.g. [2, 9, 11]) as these imply deterministic (though not polynomial-time) approximation schemes for #KNAPSACK by enumerating over all input seeds to the generator.

Approximately counting knapsack solutions. The #KNAPSACK problem can be solved with the following simple recursion: $S(i,j) = S(i-1,j) + S(i-1,j-w_i)$ where $S(i,j)$ is the number of subsets of $\{w_1, ..., w_i\}$ whose weight sums to at most j. This recurrence immediately implies a pseudo-polynomial $O(nC)$ time algorithm. More interestingly, this recurrence is the basis of all existing fully polynomial-time approximation schemes (FPTAS). That is, algorithms that for any $\varepsilon > 0$ estimate the number of solutions to within relative error $(1 \pm \varepsilon)$ in time polynomial in n and in $1/\varepsilon$.

Dyer et al. [4] were the first to show how to approximate this recurrence with random sampling. They gave a randomized sub-exponential $2^{O(\sqrt{n}\log^{2.5} n))}\epsilon^{-2}$ time algorithm. Using a more complicated random sampling (with a rapidly mixing Markov chain), Morris and Sinclair [10] obtained the first FPRAS (fully-polynomial *randomized* approximation scheme) running in $O(n^{4.5+\epsilon} + \epsilon^{-2}n^2)$ time. Dyer [3] further improved this to $O(n^{2.5}\sqrt{\log(n\varepsilon^{-1})} + \epsilon^{-2}n^2)$ by using a surprisingly simple sampling procedure (combined with randomized rounding). This to date is the fastest known randomized solution. As for deterministic solutions, the fastest solution to date is by Rizzi and Tomescu [12] and runs in $O(n^3\epsilon^{-1}\log\epsilon^{-1}/\log n)$ time. It is a logarithmic factor improvement (obtained by discretizing the recursion $S(i,j)$ with floating-point arithmetic) over the previous fastest $O(n^3\epsilon^{-1}\log(n\varepsilon^{-1}))$ time solutions of Gopalan et al. [6] (who used read-once branching programs inspired by related work on pseudorandom generators for halfspaces [9]) and of Štefankovič et al. [13] (who approximated a "dual" recursion $S^*(i,j)$ defined as the smallest capacity c such that there exist at least j subsets of $\{w_1, \dots, w_i\}$ with weight c).

Approximately counting integer knapsack solutions. In the more general integer version of #KNAPSACK, the subsets are multisets. That is, in addition to $W = \{w_1, \dots, w_n\}$ we are also given a set $\{u_1, \dots, u_n\}$ and we are allowed to take up to u_i items of weight w_i.

The first (randomized) FPRAS for counting integer knapsack solution was given by Dyer [3] who presented a strongly polynomial $O(n^5 + n^4\varepsilon^{-2})$ time algorithm. A (deterministic) FPTAS for this problem was then given by Gopalan et al. [6] with a running time of $O(n^5\varepsilon^{-2}\log^2 U \log w)$ (see also [5]) where $U = \max_i \{u_i\}$ and $w = \sum_i w_i u_i + C$. The fastest solution to date is by Halman [7] with a running time of $O(n^3\varepsilon^{-1}\log(n\varepsilon^{-1}\log U)\log^2 U)$.

Our results. In this paper we present improved algorithms for both #KNAPSACK and its integer version. Our algorithms improve the previous best algorithms by polynomial factors. For constant ε, we close the gap between the $\tilde{O}(n^{2.5})$ randomized and the $\tilde{O}(n^3)$ deterministic running times. More formally, with the standard assumption of constant time arithmetics on the input numbers, we prove the following two theorems:

▶ **Theorem 1.** *There is a FPTAS running in $O(n^{2.5}\varepsilon^{-1.5}\log(n\varepsilon^{-1})\log(n\varepsilon))$ time and $O(n^{1.5}\varepsilon^{-1.5})$ space for counting knapsack solutions.*

▶ **Theorem 2.** *There is a FPTAS running in $O(n^{2.5}\varepsilon^{-1.5}\log(n\varepsilon^{-1}\log U)\log(n\varepsilon)\log^2 U)$ time and $O(n^{1.5}\varepsilon^{-1.5}\log U)$ space for counting integer knapsack solutions.*

Our algorithm is the first algorithm to deviate from the standard recursion. In particular, on large enough sets, instead of recursing on all but the last item, we recurse in the middle and use convolution to merge the two sub-solutions. This requires extending the recent technique of *K-approximation sets and functions* used by Halman [7] and introduced in [8].

Our extended technique (which we call *sum approximations*) is simple to state and leads to a surprisingly simple solution to #KNAPSACK with an improved running time. In a nutshell, for any function $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ (think of $f(x)$ = the number of subsets with total weight exactly x) let $f^{\leq}$ denote the function $f^{\leq}(x) = \sum_{y \leq x} f(y)$ (hence $f^{\leq}(x)$ = the number of subsets with total weight at most x). Then, in order to approximate the function $f^{\leq}$ it is enough to find any function F such that $F^{\leq}$ approximates $f^{\leq}$.

We examine the properties of such sum approximations F in Section 2, and introduce a number of useful computational primitives on sum approximations. With these primitives in hand, we give a simplified version of Halman's algorithm for #KNAPSACK in Section 3. Then, in Section 4 we present an improved divide and conquer algorithm based on convolutions of sum approximations. Finally, in Section 5 we adapt our algorithm to the integer version, where every item has a corresponding multiplicity. Instead of the *binding constraints* approach used by Halman [7], we show that it is enough to perform a single scan of a sum approximation using nothing more than a standard binary search tree.

2 Approximation of a Function

Consider the following two functions: $f(x)$ = the number of subsets with total weight exactly x, and $f^{\leq}(x)$ = the number of subsets with total weight at most x. More generally:

▶ **Definition 3.** Given a function $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ we define the function $f^{\leq}(x)$ as

$$f^{\leq}(x) = \sum_{y \leq x} f(y).$$

Our goal is to approximate $f^{\leq}(C)$ but we will actually approximate the entire function $f^{\leq}(x)$ for all x. We now describe what it means to approximate a function and present some properties of such approximations.

▶ **Definition 4** ($(1+\varepsilon)$-approximation of a function)**.** Given a function $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ and a parameter $\varepsilon > 0$, a function $F : \mathbb{Z}^+ \to \mathbb{Z}^+$ is a $(1+\varepsilon)$-*approximation of f* if for every x,

$$f(x) \leq F(x) \leq (1+\varepsilon)f(x).$$

The above definition is similar to the definition of K-approximation sets [7] for $K = (1+\varepsilon)$.

▶ **Definition 5** ($(1+\varepsilon)$-sum approximation of a function)**.** Given a function $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ and a parameter $\varepsilon > 0$, a function $F : \mathbb{Z}^+ \to \mathbb{Z}^+$ is a $(1+\varepsilon)$-*sum approximation of f* if $F^{\leq}$ is a $(1+\varepsilon)$-approximation of $f^{\leq}$.

We next examine some useful properties of sum approximations. For a function $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ define its *shift* by w as follows:

$$f|_w(x) = \begin{cases} f(x-w), & x \geq w, \\ 0 & x < w, \end{cases}$$

and for two functions $f, g : \mathbb{Z}^+ \to \mathbb{Z}^+$ define their *convolution* to be:

$$(f * g)(w) = \sum_{x+y=w} f(x)g(y).$$

The following lemma describes four operations on sum approximations. The first three are similar to the ones used in [7, Property 2.1]. The fourth operation (convolution) is novel.

▶ **Lemma 6** (operations on sum approximations). *Let F be a $(1+\varepsilon)$-sum approximation of f and G be a $(1+\varepsilon)$-sum approximation of g, then the following properties hold:*

Approximation: *A $(1+\delta)$-sum approximation of F is a $(1+\delta)(1+\varepsilon)$-sum approximation of f.*

Summation: *$(F+G)$ is a $(1+\varepsilon)$-sum approximation of $(f+g)$.*

Shifting: *$F|_w$ is a $(1+\varepsilon)$-sum approximation of $f|_w$ for any $w>0$.*

Convolution: *$(F*G)$ is a $(1+\varepsilon)^2$-sum approximation of $(f*g)$.*

Proof.

Approximation: Let F' be a $(1+\delta)$-approximation of F. For every x, $f^{\leq}(x) \leq F^{\leq}(x) \leq (1+\varepsilon)f^{\leq}(x)$ and $F^{\leq}(x) \leq F'^{\leq}(x) \leq (1+\delta)F^{\leq}(x)$. We therefore have that $f^{\leq}(x) \leq F'^{\leq}(x) \leq (1+\delta)(1+\varepsilon)f^{\leq}(x)$.

Summation: For every x we have that $f^{\leq}(x) \leq F^{\leq}(x) \leq (1+\varepsilon)f^{\leq}(x)$ and $g^{\leq}(x) \leq G^{\leq}(x) \leq (1+\varepsilon)g^{\leq}(x)$, adding these two equations we get $(f+g)^{\leq}(x) \leq (F+G)^{\leq}(x) \leq (1+\varepsilon)(f+g)^{\leq}(x)$.

Shifting: For $x<w$, $f|_w(x) = 0 = F|_w(x)$. For $x \geq w$ let $y = x-w$. Since $y \geq 0$ we have that $f^{\leq}(y) \leq F^{\leq}(y) \leq (1+\varepsilon)f^{\leq}(y)$ and therefore $f|_w^{\leq}(x) \leq F|_w^{\leq}(x) \leq (1+\varepsilon)f|_w^{\leq}(x)$.

Convolution: We first prove that $(F*G)^{\leq}(w) \geq (f*g)^{\leq}(w)$:

$$\begin{aligned}
(F*G)^{\leq}(w) &= \sum_{x+y\leq w} F(x)G(y) = \sum_{x\leq w}\sum_{y\leq w-x} F(x)G(y) = \sum_{x\leq w}\left[F(x)\sum_{y\leq w-x} G(y)\right] \\
&= \sum_{x\leq w} F(x)G^{\leq}(w-x) \geq \sum_{x\leq w} F(x)g^{\leq}(w-x) = \sum_{x\leq w} F(x)\sum_{y\leq w-x} g(y) \\
&= \sum_{x+y\leq w} F(x)g(y) = \sum_{y\leq w}\sum_{x\leq w-y} F(x)g(y) = \sum_{y\leq w}\left[g(y)\sum_{x\leq w-y} F(x)\right] \\
&= \sum_{y\leq w} g(y)F^{\leq}(w-y) \geq \sum_{y\leq w} g(y)f^{\leq}(w-y) = \sum_{y\leq w} g(y)\sum_{x\leq w-y} f(x) \\
&= \sum_{x+y\leq w} f(x)g(y) = (f*g)^{\leq}(w).
\end{aligned}$$

Next we prove that $(F*G)^{\leq}(w) \leq (1+\varepsilon)^2(f*g)^{\leq}(w)$:

$$\begin{aligned}
(F*G)^{\leq}(w) &= \sum_{x+y\leq w} F(x)G(y) = \sum_{x\leq w}\sum_{y\leq w-x} F(x)G(y) = \sum_{x\leq w}\left[F(x)\sum_{y\leq w-x} G(y)\right] \\
&= \sum_{x\leq w} F(x)G^{\leq}(w-x) \leq \sum_{x\leq w} F(x)(1+\varepsilon)g^{\leq}(w-x) = \\
&= (1+\varepsilon)\sum_{x\leq w} F(x)\sum_{y\leq w-x} g(y) = (1+\varepsilon)\sum_{x+y\leq w} F(x)g(y) \\
&= (1+\varepsilon)\sum_{\leq w}\sum_{x\leq w-y} F(x)g(y) = (1+\varepsilon)\sum_{y\leq w}\left[g(y)\sum_{x\leq w-y} F(x)\right] \\
&= (1+\varepsilon)\sum_{y\leq w} g(y)F^{\leq}(w-y) \leq (1+\varepsilon)\sum_{y\leq w} g(y)(1+\varepsilon)f^{\leq}(w-y) = \\
&= (1+\varepsilon)^2\sum_{y\leq w} g(y)\sum_{x\leq w-y} f(x) = (1+\varepsilon)^2\sum_{x+y\leq w} f(x)g(y) \\
&= (1+\varepsilon)^2(f*g)^{\leq}(w).
\end{aligned}$$

◀

We next describe the way that we represent functions.

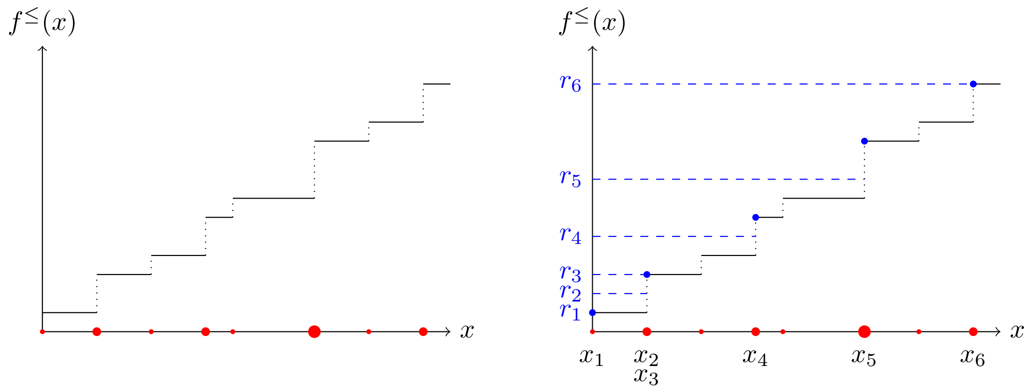


Figure 1 On the left, $f^{\leq}(x)$ compared to $f(x)$. The red point at position x is wider as $f(x)$ is larger. On the right, the blue points are the first entries that have value of at least r_i.

▶ **Definition 7** (a function representation)**.** Given a function $f : \mathbb{Z}^+ \to \mathbb{Z}^+$, the *representation of* f is defined to be a list of all the pairs $(x, f(x))$ where $f(x) > 0$. The list is kept sorted by the x value. The *size* of f (denoted by $|f|$) is the number of pairs in the representation of f. To simplify our presentation, we allow the representation to include multiple pairs with the same value of x. This can be easily fixed with a single scan over the representation.

In the following paragraphs we show how to efficiently implement the following operations on functions: *sparsification*, *summation*, *shifting*, *convolution*, and *query*. The output of each operation is a sum approximation.

Sparsification. Sparsification is the operation of constructing a $(1 + \delta)$-sum approximation of f (see Definition 5). The input is a function $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ and a sparsification parameter $\delta > 0$, the output is a function $F : \mathbb{Z}^+ \to \mathbb{Z}^+$ that is a $(1 + \delta)$-sum approximation of f. The goal is to construct a function F that has a compact representation (i.e. a small number of points with non-zero values). The general idea is based on the one in [7] (function *Compress*) but tailored towards our particular application. We partition the values of $f^{\leq}$ into segments with elements belonging to $[r_i, r_{i+1})$ (see Figure 1), where:

$$r_0 = 0,$$
$$r_{i+1} = \max\{r_i + 1, \lfloor (1 + \delta) r_i \rfloor\}.$$

We call $x_i = \min_x \left\{ f^{\leq}(x) \geq r_i \right\}$ the i-th *breakpoint*. For any x, let $\text{succ}(x)$ be the strict successor of x among $\{x_i\}$, i.e. $\text{succ}(x) = \min_i \{x_i > x\}$ We define the function $\tilde{f^{\leq}}$ (see Figure 2) as:

$$\tilde{f^{\leq}}(x) = f^{\leq}(\text{succ}(x) - 1),$$

where $\tilde{f^{\leq}}(x) = \lim_{x\to\inf} f^{\leq}(x)$ if $\text{succ}(x) = \infty$.

▶ **Lemma 8.** *$\tilde{f^{\leq}}$ is a $(1 + \delta)$-approximation of $f^{\leq}$.*

Proof. First observe that $f^{\leq}(x) \leq \tilde{f^{\leq}}(x)$ (since $\text{succ}(x) > x$ and $f^{\leq}$ is monotone). Consider any x and let i be the unique index such that $r_i \leq f^{\leq}(x) < r_{i+1}$. If $\text{succ}(x) = \infty$ then $\tilde{f^{\leq}}(x) = \lim_{x\to\inf} f^{\leq}(x) < r_{i+1}$. Otherwise, $x_{i+1} > x$ and $\tilde{f^{\leq}}(x) = f^{\leq}(x_{i+1} - 1) < r_{i+1}$. We need to consider two cases: If $r_{i+1} \leq (1 + \delta) r_i$, then $\tilde{f^{\leq}}(x) < r_{i+1} \leq (1 + \delta) f^{\leq}(x)$. If $r_{i+1} = r_i + 1$ and because the values of $\tilde{f^{\leq}}(x)$ are integer, $\tilde{f^{\leq}}(x) \leq r_{i+1} - 1 = r_i$. So in both cases $\tilde{f^{\leq}}(x) \leq (1 + \delta) f^{\leq}(x)$. ◀

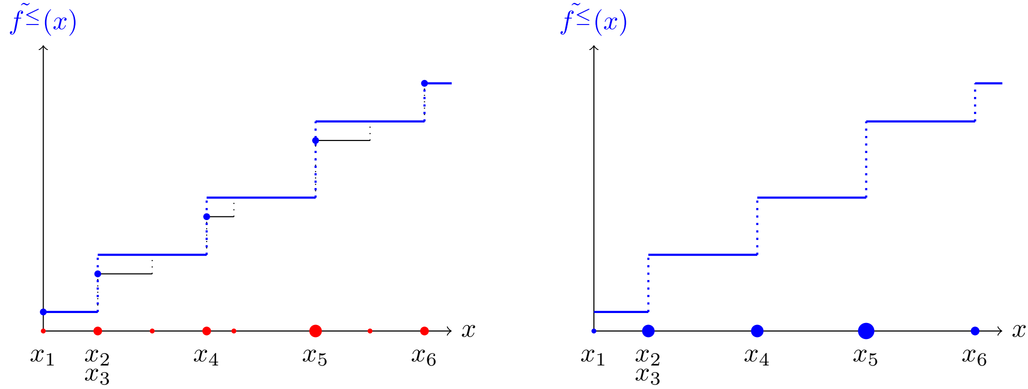


Figure 2 On the left, $\tilde{f}^{\leq}$ (in blue) is defined from $f^{\leq}$ and has the same value in any segment $[x_i, x_{i+1})$. On the right, the construction of $F(x)$. The blue points are only at positions x_i and are wider as $F(x_i)$ is larger.

Algorithm 1 SPARSIFY(f, δ).

Input: a function f represented by a sorted list of all pairs $(x, f(x))$ where $f(x) > 0$ and a sparsification parameter $\delta > 0$.

Output: a function F that is a $(1+\delta)$-sum approximation of f and is represented by a sorted list of at most $\log_{1+\delta} M$ pairs (where M is the maximum value of $f^{\leq}$).

1: initialize $r = accum = prevaccum = prevx = 0$
2: **for** every pair $(x, f(x))$ in sorted order **do**
3: $\quad r \leftarrow \max\{r+1, \lfloor (1+\delta)r \rfloor\}$
4: $\quad$ **while** $accum < r$ **do**
5: $\quad\quad accum \leftarrow accum + f(x)$
6: $\quad\quad$ get the next pair $(x, f(x))$ in the list
7: $\quad$ **end while**
8: $\quad$ add the pair $(prevx, accum - f(x) - prevaccum)$ to F
9: $\quad prevaccum \leftarrow accum - f(x)$
10: $\quad prevx \leftarrow x$
11: **end for**

We can now define the function F (the $(1+\delta)$-sum approximation of f). Observe that by Lemma 8, every F such that $F^{\leq} = \tilde{f}^{\leq}$ is a $(1+\delta)$-sum approximation. We define F as the discrete derivative of $\tilde{f}^{\leq}$. That is,

$$F(x) = \begin{cases} \tilde{f}^{\leq}(x) - \tilde{f}^{\leq}(x-1) & x > 0, \\ \tilde{f}^{\leq}(x) & x = 0. \end{cases}$$

It is easy to see that $F^{\leq} = \tilde{f}^{\leq}$. It is also easy to construct the representation of F in linear time with a single scan over the representation of f (see Algorithm 1). Let M be the maximum value of $f^{\leq}$. Notice that $\tilde{f}^{\leq}$ can have at most $|\{x_i\}| = |\{r_i\}| = \log_{1+\delta} M$ different values. This means that $|F|$ is at most $\log_{1+\delta} M$. The total running time is therefore $O(|f| + \log_{1+\delta} M)$.

Summation. Given two $(1+\varepsilon)$-sum approximations F and G of functions f and g respectively, we wish to construct the function $F + G$ (that is a $(1+\varepsilon)$-sum approximation of $f + g$

by Lemma 6). We construct $F+G$ naively by setting $(F+G)(x) = F(x)+G(x)$. The sorted list of $F+G$ can be obtained in linear time given two sorted lists of F and of G. The total space and time is therefore $O(|F|+|G|)$.

Shifting. Given a $(1+\varepsilon)$-sum approximation F of f, the function shifted by w, $F|_w$ is a $(1+\varepsilon)$-sum approximation of $f|_w$ by Lemma 6. In order to create $F|_w$, we take every pair in the representation of F, namely $(x, y = F(x))$ and change it to $(x+w, y)$. $F|_w(x)$ will be the sum of all the pairs where the first coordinate is x. The total space and time is $O(|F|)$.

Convolution. The convolution $F * G$ contains all the combinations of taking some x value from F and some y value from G. For every pair x, y such that $F(x) \neq 0$ and $G(x) \neq 0$ we add the value $F(x)G(y)$ to the value of $F * G$ at point $x+y$.

We sort these pairs in order to get the representation of $F * G$. For a certain y we have all the points $(x+y, F(x))$ sorted already, those are $|G|$ sorted sequences that we have to merge. The total space of the output $F * G$ is at most the number of such pairs x, y, that is $|F| \cdot |G|$. But it is possible to obtain a stream of the sorted pairs with their value using less space, by using a heap to merge the lists. Assuming without loss of generality that $|G| \leq |F|$, each list is a value y and a pointer to a point in F, and the heap extracts the minimum value of the sum of y and the value in the pointer. The total time to create $F * G$ is therefore $O(|F| \cdot |G| \cdot \log(\min\{|F|, |G|\}))$ and the space $O(|F|+|G|)$.

Query. Given a $(1+\varepsilon)$-sum approximation F of f and a point x, we can query the value $F^{\leq}(x)$ that satisfies $f^{\leq}(x) \leq F^{\leq}(x) \leq (1+\varepsilon)f^{\leq}(x)$ in time $O(|F|)$. This is because computing the function $F^{\leq}(x) = \sum_{y \leq x} F(y)$ takes $O(|F|)$ time by considering every y. Moreover, if we store the representation of F in a balanced binary search tree T then a query can be done in $O(\log|F|)$ time with a prefix sum query on T.

3 The Algorithm of Halman [7] (Simplified)

In this section we present a simplified version of the algorithm of Halman [7] for #Knapsack using sum approximations. The running time of this simple deterministic algorithm is $O(n^3\varepsilon^{-1})$ and the space is $O(n^2\varepsilon^{-1})$.

For a set of weights S, let $k_S(x)$ denote the number of subsets of S with total weight exactly x. The output of the algorithm is the function K_W that is a $(1+\varepsilon)$-sum approximation of k_W. The desired answer, $K_W^{\leq}(C)$, can then be easily obtained using the query operation.

Recall that $k_S|_w(x) = k_S(x-w)$ if $x \geq w$ and 0 otherwise. The algorithm is based on the following observation:

▶ **Lemma 9.** *Let S be a set of integer weights and w be an additional integer weight, then:*

$$k_{S\cup\{w\}} = k_S + k_S|_w$$

Proof. Any subset of $S \cup \{w\}$ with weight x either includes w (the number of such solutions is $k_S(x-w)$) or does not include w (the number of such solutions is $k_S(x)$). Since these options are disjoint, we have that $k_{S\cup\{w\}}(x) = k_S(x) + k_S|_w(x)$. ◀

The algorithm. The algorithm uses the above lemma to construct the set S by inserting one element at a time (until $S = W$), keeping K_S updated. The algorithm starts by setting $K_\emptyset(0) = 1$ and $K_\emptyset(x) = 0$ for any $x \neq 0$. In the i-th step, we compute the function

$K_{\{w_1,\dots,w_i\}}$ from $K_{\{w_1,\dots,w_{i-1}\}}$. Computing $K_{\{w_1,\dots,w_i\}}$ can be done with one shifting and one summation operation: $K_{\{w_1,\dots,w_i\}} = K_{\{w_1,\dots,w_{i-1}\}} + K_{\{w_1,\dots,w_{i-1}\}}|_{w_i}$. Notice that the size $|K_{\{w_1,\dots,w_i\}}| = 2|K_{\{w_1,\dots,w_{i-1}\}}|$ doubles from the summation operation. To overcome this blowup, at the end of each step of the algorithm, we sparsify with parameter $\delta = (1+\varepsilon)^{\frac{1}{n}} - 1$.

Correctness. From Lemmas 6 and 9, it follows that if K_S is a $(1+\alpha)$-sum approximation of k_S, then $K_S + K_S|_w$ is a $(1+\alpha)$-sum approximation of $k_{S\cup\{w\}}$. Furthermore, the approximation factor of $K_{\{w_1,\dots,w_i\}}$ after the sparsification is the approximation factor of $K_{\{w_1,\dots,w_{i-1}\}}$ multiplied by $(1+\delta)$. We get that K_W is a $((1+\varepsilon)^{\frac{1}{n}})^n$-sum approximation of k_W, as required.

Time complexity. The size of $K_{\{w_1,\dots,w_i\}}$ after the sparsification is bounded by $\log_{1+\delta} 2^i$. The time complexity is therefore:

$$\sum_{i=1}^{n} O\big(\log_{1+\delta} 2^i\big) = O\left(\sum_{i=1}^{n} \frac{i}{\log(1+\delta)}\right) = O\left(\frac{1}{\frac{1}{n}\log(1+\varepsilon)} \sum_{i=1}^{n} i\right) = O\big(n^3\varepsilon^{-1}\big)\,.$$

Space complexity. The space is $O(|K_W|) = O(\log_{1+\delta} 2^n) = O\Big(\frac{n}{\log(1+\delta)}\Big) = O\Big(\frac{n^2}{\log(1+\varepsilon)}\Big) = O(n^2\varepsilon^{-1})$, where we have used that $\ln(1+\varepsilon) \geq \varepsilon/2$ for $\varepsilon \in (0,1)$.

4 The Algorithm for Counting Knapsack Solutions

In this section we present a deterministic $O(n^{2.5}\varepsilon^{-1.5}\log(n\varepsilon^{-1})\log(n\varepsilon))$ time $O(n^{1.5}\varepsilon^{-1.5})$ space algorithm for counting knapsack solutions. The algorithm is based upon a similar observation to the one in Lemma 9:

▶ **Lemma 10.** *Let S and T be two sets of integer weights, then:*

$$k_{S\cup T} = k_S * k_T$$

Proof. A subset of $S \cup T$ of weight w must be obtained by taking a subset of weight w_S from S and a subset of weight w_T from T where $w_S + w_T = w$. Thus, $k_S(w_S)$ subsets of S of weight w_S and $k_T(w_T)$ subsets of T weight w_T generate $k_S(w_S)k_T(w_T)$ subsets of $S \cup T$ of weight $w_S + w_T$. Overall, we get that $k_{S\cup T}(w) = \sum_{w_S + w_T = w} k_S(w_S)k_T(w_T) = (k_S * k_T)(w)$. ◀

The algorithm. As in Section 3, our algorithm computes a $(1+\varepsilon)$-sum approximation K_W of k_W recursively. This time however, we do two things differently: (1) The value of the approximation factor is different for each recursion depth. In a recursive call of depth i, we are given a set S and we compute a $(1+\varepsilon_i)$-sum approximation K_S of k_S for some ε_i to be chosen later. (2) Given a set S we recurse differently depending on the size of S:

1. If $|S| > \sqrt{n/\varepsilon}$, then we partition the set S into two sets A and B each of size $|S|/2 = n/2^i$ and make two recursive calls: One computes a $(1+\varepsilon_{i+1})$-sum approximation K_A of k_A and the other computes a $(1+\varepsilon_{i+1})$-sum approximation K_B of k_B. We then find K_S by computing the convolution $K_A * K_B$ and sparsifying with parameter

   $$\delta_i = \frac{\varepsilon^{3/4}}{2c \cdot 2^{i/2} \cdot n^{1/4}},$$

   where ε is the original approximation parameter and $c = \frac{\sqrt{2}}{\sqrt{2}-1}$.
2. If $|S| \leq \sqrt{n/\varepsilon}$, then we apply the algorithm from Section 3 on the set S with parameter $\delta_{\log(\sqrt{n\varepsilon})} = \Omega(\sqrt{\varepsilon/n})$. Observe that in such a case the recursion depth is at least $\log(\sqrt{n\varepsilon})$.

Correctness. From Lemmas 6 and 10, it follows that if K_A is a $(1+\varepsilon_{i+1})$-sum approximation of k_A and K_B is a $(1+\varepsilon_{i+1})$-sum approximation of k_B, then K_S is a $(1+\varepsilon_{i+1})^2(1+\delta_i)$-sum approximation of k_S. This means that ε_i satisfies the following relation:

$$(1+\varepsilon_i) = (1+\varepsilon_{i+1})^2(1+\delta_i)$$

From the above equation and since on the bottom of the recursion with $\delta_i = \delta_{\log(\sqrt{n\varepsilon})}$, the final approximation factor of K_W is:

$$(1+\varepsilon_0) = (1+\delta_{\log(\sqrt{n\varepsilon})})^{\sqrt{n\varepsilon}} \prod_{i=0}^{\log(\sqrt{n\varepsilon})-1} (1+\delta_i)^{2^i} = \prod_{i=0}^{\log(\sqrt{n\varepsilon})} (1+\delta_i)^{2^i}$$

We need to prove that the above product is not larger than $(1+\varepsilon)$. Since $\sum_{i=0}^{x} 2^{i/2} < \frac{\sqrt{2}}{\sqrt{2}-1} \cdot 2^{x/2} = c \cdot 2^{x/2}$ and $(1+\delta_i)^{2^i} = (1+\delta_i)^{1/\delta_i \cdot 2^i \delta_i} \le e^{2^i \delta_i} = e^{\frac{\varepsilon^{3/4}}{2c} n^{-1/4} 2^{i/2}}$ we obtain:

$$\prod_{i=0}^{\log(\sqrt{n\varepsilon})} (1+\delta_i)^{2^i} \le e^{\frac{\varepsilon^{3/4}}{2c} n^{-1/2} \sum_{i=0}^{\log(\sqrt{n\varepsilon})} 2^{i/2}} < e^{\frac{\varepsilon^{3/4}}{2} n^{-1/4} 2^{\log(\sqrt{n\varepsilon})/2}} = e^{\frac{\varepsilon}{2}} \le (1+\varepsilon),$$

where the last inequality follows from $\ln(1+\varepsilon) \ge \varepsilon/2$ for $\varepsilon \in (0,1)$. Moreover, since the recursion changes at depth $\log(\sqrt{n\varepsilon})$ we further need to assume that $\varepsilon \ge 1/n$. These assumptions are without loss of generality since for $\varepsilon > 1$ we could simply use $\varepsilon = 1$ and for $\varepsilon < 1/n$ the previous algorithms are faster.

Time complexity. We analyze the time complexity of every recursion depth i. For depth $i = \log(\sqrt{n\varepsilon})$, we apply the simple algorithm from Section 3 $\lfloor \sqrt{n\varepsilon} \rfloor$ times on sets of size $\Theta(\sqrt{n/\varepsilon})$ with $\delta_{\log(\sqrt{n\varepsilon})} = \Omega(\sqrt{\varepsilon/n})$, the running time is therefore $O(n^{2.5}\varepsilon^{-1.5})$.

For depth $i < \log(\sqrt{n\varepsilon})$, we apply 2^i convolutions and sparsifications. The time of the convolutions is dominant. The total running time is therefore:

$$\begin{aligned}\sum_{i=0}^{(\log\sqrt{n\varepsilon})-1} 2^i \left(\frac{n}{2^{i+1}\cdot\delta_{i+1}}\right)^2 \log\left(\frac{n}{2^{i+1}\cdot\delta_{i+1}}\right) &\le \frac{1}{2}\sum_{i=1}^{\log\sqrt{n\varepsilon}} 2^i \left(\frac{n}{2^i\cdot\delta_i}\right)^2 \log(n\varepsilon^{-1}) \\ &= O\left(\sum_{i=1}^{\log\sqrt{n\varepsilon}} \frac{n^{2.5}}{\varepsilon^{1.5}} \log(n\varepsilon^{-1})\right) \\ &= O(n^{2.5}\varepsilon^{-1.5}\log(n\varepsilon^{-1})\log(n\varepsilon)).\end{aligned}$$

Space complexity. Since each recursive call makes at most two recursive calls, we do not need to keep more than two representation of sum approximations on every level of the recursion. We have seen in Section 2 that it is possible to construct sparsification of convolution in linear space. The space complexity of all recursive calls of depth $i < \log(\sqrt{n\varepsilon})$ is therefore:

$$\sum_{i=0}^{\log(\sqrt{n\varepsilon})} \left(2\cdot\frac{n}{2^i\cdot\delta_i}\right) = O\left(\sum_{i=0}^{\log(\sqrt{n\varepsilon})} \frac{n^{1.25}}{2^{i/2}\cdot\varepsilon^{0.75}}\right) = O\left(n^{1.25}\varepsilon^{-0.75}\right)$$

One call to the algorithm from Section 3 uses $O\left(\frac{\left(\sqrt{n/\varepsilon}\right)^2}{\delta_{\log(\sqrt{n\varepsilon})}}\right) = O\left(\frac{\left(\sqrt{n/\varepsilon}\right)^2}{\sqrt{\varepsilon/n}}\right) = O(n^{1.5}\varepsilon^{-1.5})$ space, and therefore the total space complexity is $O(n^{1.5}\varepsilon^{-1.5})$.

5 The Algorithm for Counting Integer Knapsack Solutions

In this section we show how to generalize the algorithms of Sections 3 and 4 to the integer version of counting knapsack solutions.

5.1 Generalizing the algorithm of Section 3

In Section 3 we showed how to insert into a set S a single item with weight w. We now need to show how to insert a single item with weight w and multiplicity u. The proof of the following lemma is similar to that of Lemma 9.

▶ **Lemma 11.** *Let S be a set of integer pairs representing weights and multiplicity of items, and let (w, u) be the weight and multiplicity of an additional item, then:*

$$k_{S\cup\{(w,u)\}} = k_S + k_S|_w + k_S|_{2w} + \ldots + k_S|_{u\cdot w}$$

We will describe a new operation on sum approximations that creates the sparsification of $G = (K_S + K_S|_w + K_S|_{2w} + \ldots + K_S|_{u\cdot w})$ without actually computing G, i.e. without actually computing all the points with non-zero value.

Events. Observe that a point (x, y) with non-zero value $y = K_S(x)$ implies $u + 1$ points in the above sum: $(x, y), (x + w, y), (x + 2w, y), \ldots, (x + u \cdot w, y)$. We call the first point (x, y) a *start event* (with position x and value y) and the last point $(x + u \cdot w, y)$ an *end event* (with position $x + u \cdot w$ and value y). Overall, for every x with non-zero value $y = K_S(x)$ there are two events, a total of $2|K_S|$ events. It is possible to sort in linear time the sequence of events by their positions $\{x\}_i \cup \{x + u \cdot w\}_i$ because K_S is given sorted. We call the sorted list of events the *event list*.

Similarly to Algorithm 1, we could construct the sparsification of $G = (K_S + K_S|_w + K_S|_{2w} + \ldots + K_S|_{u\cdot w})$ by scanning all points in G. This however would be too costly. Instead, we next present a new operation that constructs the sparsification of G while only scanning the $O(|K_S|)$ events in the event list.

InsertAndSparsify. While scanning the event list, when we see a start event (x, y) then we say that this event is an *active event* and that all the points $(x, y), (x + w, y), (x + 2w, y), \ldots, (x + u \cdot w, y)$ are *active points*. These points will become inactive when we will see the end event $(x + u \cdot w, y)$. As in Algorithm 1, we would like to accumulate the values of all points seen so far. When the accumulator is larger than r, we introduce a new breakpoint (i.e., output a new point to the sparsification of G) and set the new r to be $\max\{r + 1, \lfloor(1 + \delta)r\rfloor\}$. During the scan, apart from the accumulator, we also maintain a value Y = the sum of values of all currently active events.

When we scan a start event (x, y), we add the value y to both the accumulator and to Y. This is not enough. We also need to add to the accumulator the values of all active points whose position is x. These are precisely the points whose start events were at positions $x - w, x - 2w, \ldots, x - u \cdot w$. Notice that all these points have the same start position modulo w. For this reason, we maintain a balanced binary search tree T that stores all active events keyed by their start position modulo w. Each node v in T with key $r \in \{0, \ldots, w - 1\}$ stores a field Y_v = the sum of values of all active points whose start position is x such that $x = r \pmod w$. When we see a start event (x, y), we search in T for the node v with key $x \bmod w$ (or create one if no such node exists), we increase the node's Y_v field by y, and increase the accumulator by y. If (x, y) is an end event we subtract y from Y_v and from Y.

After processing event (x, y) as explained above, we want to process the next event (x', y') in the event list. Before doing so, we need to: (1) increase the accumulator by the total value of all points in the segment $[x, x')$, and (2) if the updated accumulator is larger than r, find and output the (possibly many) new breakpoints whose positions are between x and x'.

(1) We partition the segment $[x, x')$ into three segments: $[x, k_1 w), [k_1 w, k_2 w), [k_2 w, x')$ such that the lengths of the first and last segments are smaller than w, and the length of the middle segment is a multiple of w. Notice that every segment of length w contains an active point from every active event exactly once. Therefore, to obtain the total value of points in the segment $[x, x')$ we query T for the total value in segment $[x, k_1 w)$ (with a *suffix sum* query on the Y_v values) and in segment $[k_2 w, x')$ (with a *prefix sum* query on the Y_v values), and add to it $Y \cdot (k_2 - k_1)$ (the total value in segment $[k_1 w, k_2 w)$).

(2) After increasing the accumulator by the above total value, if the accumulator becomes larger than r, then we will find all the new breakpoints in $[x, x')$ in $O(\log |K_S|)$ time per breakpoint. Suppose the accumulated value at x was *prevaccum*. To find the first breakpoint in the segment $[x, k_1 w)$ we query T for the the first node after $x \bmod w$ such that the sum of Y_v values between $x \bmod w$ and that node is at least $r - prevaccum$ (we call this a *succeeding sum* query on T, and we symmetrically define a *preceding sum* query in which we seek the first node before $x \bmod w$ rather than after). We then output this breakpoint, set *prevaccum* to be the accumulated value in this breakpoint, set $r = \max\{r + 1, \lfloor (1 + \delta) r \rfloor\}$, and continue in the same way to find the next breakpoint in the segment $[x, k_1 w)$. Next, we find the breakpoints in the segment $[k_1 w, k_2 w)$. Since this segment is composed of $(k_2 - k_1)$ subsegments of length w, and since each of these subsegments contributes exactly Y to the accumulator, it is easy (in $O(1)$ time) to find which subsegment contains the next breakpoint. On this subsegment we proceed similarly as on $[x, k_1 w)$. Finally, we need to find the breakpoints in segment $[k_2 w, x')$, again similarly as in $[x, k_1 w)$. See Algorithm 2.

Time and space complexity. Each operation on the binary search tree T takes $O(\log |K_S|)$ time thus the total time for InsertAndSparsify is $O\big((|K_{S \cup \{(w,u)\}}| + |K_S|) \cdot \log |K_S|\big)$. If $S = \{w_1, \dots, w_i\}$ then $|K_S| = \log_{1+\delta} U^i$ and $|K_{S \cup \{(w,u)\}}| = O\big(\log_{1+\delta} U^i\big)$. The total time complexity of the algorithm is therefore:

$$\begin{aligned}\sum_{i=1}^{n} O\big(\log_{1+\delta} U^i \cdot \log(\log_{1+\delta} U^i)\big) &= O\left(\sum_{i=1}^{n} \frac{i \log U}{\log(1+\delta)} \cdot \log\left(\frac{i \log U}{\log(1+\delta)}\right)\right) \\ &= O\left(\frac{\log U}{\frac{1}{n}\log(1+\varepsilon)} \cdot \log\left(\frac{n \log U}{\frac{1}{n}\log(1+\varepsilon)}\right) \sum_{i=1}^{n} i\right) \\ &= O\big(n^3 \varepsilon^{-1} \log U \log\left(n \varepsilon^{-1} \log U\right)\big)\,.\end{aligned}$$

The space complexity is $O(|K_W|) = O\big(\log_{1+\delta} U^n\big) = O\Big(\frac{n \log U}{\log(1+\delta)}\Big) = O(n^2 \varepsilon^{-1} \log U)$.

5.2 Generalizing the algorithm of Section 4

The only change to the algorithm of Section 4 is that when the set size is small (i.e. when we call the algorithm of Section 3) we use InsertAndSparsify as in the previous subsection.

Time and space complexity. We observe that the size of the representation of a $(1+\varepsilon)$-sum approximation has been changed to $|K_S| = \log_{1+\delta} U^{|S|} = \frac{|S| \log U}{\delta}$. As in Section 4, we

Algorithm 2 INSERTANDSPARSIFY(K_S, w, u, δ).

Input: a sum approximation K_S of k_S, a new item with weight of w and multiplicity u, and a sparsification parameter δ.
Output: a function G that is a $(1+\delta)$-sum approximation of $k_{S\cup(w,u)}$.
1: initialize T as an empty binary search tree of pairs (x, y) indexed by $(x \bmod w)$
2: initialize $Y = x = accum = prevaccum = 0$, and $r = 1$
3: **for** every event (x', y') in sorted order **do**
4: $k_1 \leftarrow \lceil \frac{x}{w} \rceil$, $k_2 \leftarrow \lfloor \frac{x'}{w} \rfloor$
5: **while** $accum + T.\mathsf{suffixSum}(x) + Y \cdot (k_2 - k_1) + T.\mathsf{prefixSum}(x' - 1) \geq r$ **do**
6: **if** $accum + T.\mathsf{suffixSum}(x) \geq r$ **then**
7: $bp \leftarrow T.\mathsf{succeedingSum}(x, r - accum)$
8: $accum \leftarrow accum + T.\mathsf{suffixSum}(x) - T.\mathsf{suffixSum}(bp)$
9: **else**
10: $k_3 \leftarrow k_1 + \lfloor (r - accum - T.\mathsf{suffixSum}(x))/Y \rfloor$
11: $bp \leftarrow T.\mathsf{precedingSum}(x', r - (accum + T.\mathsf{suffixSum}(x) + Y \cdot (k_3 - k_1)))$
12: $accum \leftarrow accum + T.\mathsf{suffixSum}(x) + Y \cdot (k_3 - k_1) + T.\mathsf{prefixSum}(bp - 1)$
13: **end if**
14: add the pair $(x', accum - prevaccum - T[bp \bmod w])$ to G
15: $prevaccum \leftarrow accum$, $x \leftarrow bp$, $k_1 \leftarrow \lceil \frac{x}{w} \rceil$, $r \leftarrow \max\{r+1, \lfloor (1+\delta)r \rfloor\}$
16: **end while**
17: **if** (x', y') is a start event **then**
18: $Y \leftarrow Y + y'$, $T[x' \bmod w] \leftarrow T[x' \bmod w] + y'$
19: **else**
20: $Y \leftarrow Y - y'$, $T[x' \bmod w] \leftarrow T[x' \bmod w] - y'$
21: **end if**
22: $accum \leftarrow accum + T.\mathsf{suffixSum}(x) + Y \cdot (k_2 - k_1) + T.\mathsf{prefixSum}(x' - 1)$
23: $x \leftarrow x'$
24: **end for**

calculate the total time complexity for depth $\log(\sqrt{n\varepsilon})$:

$$O\left(\sqrt{n\varepsilon} \cdot \left(\left(\sqrt{n/\varepsilon}\right)^3 \left(\sqrt{\varepsilon/n}\right)^{-1} \log U \log \left(\sqrt{n\varepsilon} \left(\sqrt{\varepsilon/n}\right)^{-1} \log U \right) \right) \right)$$

which is $O\big(n^{2.5}\varepsilon^{-1.5} \log U \log(n\varepsilon^{-1} \log U)\big)$. The total time for depths $i < \log(\sqrt{n\varepsilon})$ is:

$$\begin{aligned}
&\sum_{i=0}^{(\log \sqrt{n\varepsilon})-1} 2^i \left(\frac{n \log U}{2^{i+1} \cdot \delta_{i+1}} \right)^2 \log \left(\frac{n \log U}{2^{i+1} \cdot \delta_{i+1}} \right) \\
&\leq \frac{1}{2} \sum_{i=1}^{\log \sqrt{n\varepsilon}} 2^i \left(\frac{n \log U}{2^i \cdot \delta_i} \right)^2 \log(n\varepsilon^{-1} \log U) \\
&= O\left(\sum_{i=1}^{\log \sqrt{n\varepsilon}} \frac{n^{2.5} \log^2 U}{\varepsilon^{1.5}} \log(n\varepsilon^{-1} \log U) \right) \\
&= O(n^{2.5}\varepsilon^{-1.5} \log(n\varepsilon^{-1} \log U) \log(n\varepsilon) \log^2 U).
\end{aligned}$$

The space complexity is dominated by the space used for INSERTANDSPARSIFY on a set of size $\sqrt{n/\varepsilon}$ and $\delta = \sqrt{\varepsilon/n}$, that is $O(n^{1.5}\varepsilon^{-1.5} \log U)$.

References

1 Mohsen Bayati, David Gamarnik, Dimitriy Katz, Chandra Nair, and Prasad Tetali. Simple deterministic approximation algorithms for counting matchings. In *STOC*, pages 122–127, 2007.

2 Ilias Diakonikolas, Parikshit Gopalan, Ragesh Jaiswal, Rocco A. Servedio, and Emanuele Viola. Bounded independence fools halfspaces. In *FOCS*, pages 171–180, 2009.

3 Martin Dyer. Approximate counting by dynamic programming. In *STOC*, pages 693–699, 2003.

4 Martin Dyer, Alan Frieze, Ravi Kannan, Ajai Kapoor, Ljubomir Perkovic, and Umesh Vazirani. A mildly exponential time algorithm for approximating the number of solutions to a multidimensional knapsack problem. *Combinatorics, Probability and Computing*, 2:271–284, 1993.

5 Parikshit Gopalan, Adam Klivans, and Raghu Meka. Polynomial-time approximation schemes for knapsack and related counting problems using branching programs. *arXiv 1008.3187*, 2010.

6 Parikshit Gopalan, Adam Klivans, Raghu Meka, Daniel Štefankovic, Santosh Vempala, and Eric Vigoda. An FPTAS for #Knapsack and related counting problems. In *FOCS*, pages 817–826, 2011.

7 Nir Halman. A deterministic fully polynomial time approximation scheme for counting integer knapsack solutions made easy. *Theoretical Computer Science*, 645:41–47, 2016.

8 Nir Halman, Diego Klabjan, Mohamed Mostagir, Jim Orlin, and David Simchi-Levi. A fully polynomial-time approximation scheme for single-item stochastic inventory control with discrete demand. *Mathematics of Operations Research*, 34(3):674–685, 2009.

9 Raghu Meka and David Zuckerman. Pseudorandom generators for polynomial threshold functions. In *STOC*, pages 427–436, 2010.

10 Ben Morris and Alistair Sinclair. Random walks on truncated cubes and sampling 0-1 knapsack solutions. *SIAM journal on computing*, 34(1):195–226, 2004. Preliminary version in FOCS 1999.

11 Yuval Rabani and Amir Shpilka. Explicit construction of a small epsilon-net for linear threshold functions. In *STOC*, pages 649–658, 2009.

12 Romeo Rizzi and Alexandru I. Tomescu. Faster FPTASes for counting and random generation of knapsack solutions. In *ESA*, pages 762–773, 2014.

13 Daniel Štefankovič, Santosh Vempala, and Eric Vigoda. A deterministic polynomial-time approximation scheme for counting knapsack solutions. *SIAM Journal on Computing*, 41(2):356–366, 2012.

14 Dror Weitz. Counting independent sets up to the tree threshold. In *STOC*, pages 140–149, 2006.