Space-Efficient Approximations for Subset Sum *

Anna Gál[1**], Jing-Tang Jang[2], Nutan Limaye[3], Meena Mahajan[4], and Karteek Sreenivasaiah[5***]

[1] University of Texas at Austin, panni@cs.utexas.edu
[2] Google, Mountain View, keithjtjang@google.com
[3] IIT Bombay, nutan@cse.iitb.ac.in
[4] Institute of Mathematical Sciences, Chennai, meena@imsc.res.in
[5] Max-Planck Institute for Informatics, Saarbrücken, karteek@mpi-inf.mpg.de

Abstract. SubsetSum is a well known NP-complete problem: given $t \in Z^+$ and a set S of m positive integers, output YES if and only if there is a subset $S' \subseteq S$ such that the sum of all numbers in S' equals t. The problem and its search and optimization versions are known to be solvable in pseudo-polynomial time in general.

We develop a 1-pass deterministic streaming algorithm that uses space $O\left(\frac{\log t}{\epsilon}\right)$ and decides if some subset of the input stream adds up to a value in the range $\{(1 \pm \epsilon)t\}$. Using this algorithm, we design space efficient Fully Polynomial-Time Approximation Schemes (FPTAS) solving the search and optimization versions of SubsetSum. Our algorithms run in $O(\frac{1}{\epsilon}m^2)$ time and $O(\frac{1}{\epsilon})$ space on unit cost RAMs, where $1+\epsilon$ is the approximation factor. This implies constant space quadratic time FPTAS on unit cost RAMs when ϵ is a constant. Previous FPTAS used space linear in m.

In addition, we show that on certain inputs, when a solution is located within a short prefix of the input sequence, our algorithms may run in sublinear time. We apply our techniques to the problem of finding balanced separators, and we extend our results to some other variants of the more general knapsack problem.

When the input numbers are encoded in unary, the decision version has been known to be in log space. We give streaming space lower and upper bounds for unary SubsetSum. If the input length is N when the numbers are encoded in unary, we show that randomized s-pass streaming algorithms for exact SubsetSum need space $\Omega(\frac{\sqrt{N}}{s})$, and give a simple deterministic two pass streaming algorithm using $O(\sqrt{N}\log N)$ space.

Finally, we formulate an encoding under which unary SubsetSum is monotone, and show that the exact and approximate versions in this formulation have monotone $O(\log^2 t)$ depth Boolean circuits. We also show that any circuit using ε-approximator gates for Subset Sum under this encoding needs $\Omega(n/\log n)$ gates to compute the Disjointness function.

* A preliminary version of part of this paper appeared in COCOON '12 [LMS12]
** Supported in part by NSF Grant CCF-1018060
*** This work was done while this author was at IMSc, Chennai.

1 Introduction

The SubsetSum problem was one of Karp's first 21 NP-complete problems [Kar72]. Its decision, search and optimization versions are defined as follows.

Definition 1 (SubsetSum).

- **Input:** *A finite set* $S = \{a_1, \ldots, a_m\}$ *of* m *positive integers and* $t \in Z^+$.
- **Decision:** *decide if there exists a subset* $S' \subseteq S$ *such that the sum of all numbers in* S' *equals* t.
- **Search:** *Output such a subset if one exists.*
- **Optimization:** *Output a subset* $S' \subseteq S$ *such that the sum of all numbers in* S' *is as large as possible but not larger than* t. *(If all numbers exceed* t, *output the empty subset.)*

(S and S' can even be multi-sets; the numbers are not required to be distinct. In that case, we ask for a subset $T \subseteq [m]$ such that $\sum_{i \in T} a_i$ equals t. We use the terms set and subset only to keep notation simple.)

Approximation algorithms for the optimization version of the SubsetSum problem have been extensively studied. Approximation versions can also be defined for the decision and search questions. The approximation problems are framed as follows:

Definition 2 (ϵ-approximate SubsetSum).

- **Input:** *A finite set* S *of positive integers,* $t \in Z^+$, *and* $0 < \epsilon < 1$.
- **Decision:** *Decide if there exists a subset* $S' \subseteq S$ *such that*

$$(1 - \epsilon)t \leq \sum_{a_i \in S'} a_i \leq (1 + \epsilon)t.$$

- **Search:** *Output such a subset if it exists.*
- **Optimization:** *Let* t^* *be the largest possible sum of a subset of* S *without exceeding* t. *Output a subset* $S' \subseteq S$ *such that the sum of all the numbers in* S' *is at least* $(1 - \epsilon)t^*$, *but not larger than* t.

Space Complexity. The focus of this work is the space complexity of SubsetSum. As common in complexity theory (see for instance [AB09]), to measure the amount of workspace used by an algorithm we assume that the input is "read only", the output is "write only", and we only count towards the space bound the space used in addition to storing the input and the output. In other words, storing the input and output is for free, but the input/output space cannot be used by the computation for any other purpose. Space may be measured in the Turing machine model (bit complexity), or in the unit-cost RAM model. An even more stringent model is the streaming model, where not only is the input "read only", but further, the input arrives as a stream, and the algorithm can make a limited number of left-to-right passes over the stream.

We denote by n the length of the input represented in binary, that is $n = \sum_{i=1}^{m} \log a_i + \log t$. The Unary Subset Sum problem (USS) is the same problem, but with the input numbers

given in unary (for instance $1^B 0 1^{a_1} 0 \ldots 1^{a_m}$.) In this case the input length N is $B + \sum_{i=1}^{m} a_i$. (To emphasize the difference, we use B instead of t when considering USS.) Note that $m + B \leq N$, and if the numbers form a set (as opposed to multi-set) and are all less than B, then $N \leq (m+1)B$. However, in general there is no tighter relation between N and m, B. Our bounds for binary encodings are stated in terms of n, m, t, and for unary encodings in terms of N alone.

1.1 Previous Space Efficient Algorithms

The classical solution for the decision, search and optimization versions of the SubsetSum problem is to use dynamic programming (Dantzig [Dan57], Bellman [Bel57]), which runs in time $O(mt)$ and space $O(t)$ in the unit-cost RAM model. This is *pseudo*-polynomial in m, that is, if the value of t is bounded by a polynomial in m, then it runs in polynomial time and space. See Garey and Johnson ([GJ79] Section 4.2) for details.

With unary encodings of the input, the SubsetSum problem USS is known to be in P. In [EJT10], Elberfeld, Jakoby and Tantau showed a powerful meta-theorem for obtaining logspace upper bounds, and used it to conclude that USS is even in logspace. In [Kan10] Daniel Kane gave a considerably simplified logspace algorithm. Improving upon this further, in [EJT12] it is shown that under appropriate encodings USS has polynomial-sized formulas (equivalently, polynomial-sized log depth circuits) and hence is in the class NC^1. They also show that USS has polynomial-sized constant-depth formulas using MAJORITY and NOT gates and hence is in the class TC^0. On the other hand, it can be shown easily that MAJORITY reduces to computing USS and hence USS is TC^0-hard. Thus USS is TC^0-complete.

Approximation algorithms for the optimization version of the SubsetSum problem have also been extensively studied. An approximation algorithm is said to achieve *approximation factor* $1+\epsilon$ if for every input $\max(\frac{t^*}{\hat{t}}, \frac{\hat{t}}{t^*}) \leq 1+\epsilon$, where t^* is the optimal solution and $\hat{t}$ is the solution provided by the algorithm. That is, for small ϵ the computed solution is not too far from the optimal. With this definition, in our setting if for every input $(1-\epsilon)t^* \leq \hat{t} \leq t^*$, the approximation algorithm achieves approximation factor $1+\epsilon'$ with $\epsilon' = \frac{\epsilon}{1-\epsilon}$. The solution computed may not be optimal, but it is required to be feasible; hence for the SubsetSum problem, $\hat{t} \leq t^*$. Note that t^* is not known in advance.

If the parameter $\epsilon > 0$ is not fixed but is also provided as input, then the approximation algorithm with approximation factor $1+\epsilon$ is called an *approximation scheme*. An approximation scheme is a *polynomial-time approximation scheme (PTAS)* if for any $\epsilon > 0$, the running time is polynomial in n, where n is the input length. An approximation scheme is a *Fully Polynomial-Time Approximation Scheme (FPTAS)* if the running time is polynomial in both $1/\epsilon$ and n. See Vazirani [Vaz01] and Cormen et.al [CLRS09] for detailed discussions.

Ibarra and Kim [IK75] gave FPTAS for the SubsetSum optimization problem with running time $O(m/\epsilon^2)$ and space $O(m + 1/\epsilon^3)$. Gens and Levner [GL79] gave an alternative FPTAS algorithm running in time $O(m/\epsilon + 1/\epsilon^3)$ and space $O(m + 1/\epsilon)$. Both results by Ibarra and Kim, and Gens and Levner use dynamic programming. Also see the survey by Gens and Levner [GL80]. As far as we know, the current best FPTAS algorithm for the SubsetSum problem is given by Kellerer, Mansini, Pferschy, and Speranza [KMPS03], which runs in time

$O(\min(m/\epsilon, m + 1/\epsilon^2 \log \frac{1}{\epsilon}))$ and space $O(m + 1/\epsilon)$. These bounds are stated with respect to the unit-cost RAM model. See Woeginger and Yu [WY92] and Bazgan, Santha, and Tuza [BST02] for further variants and FPTAS for the SubsetSum problem.

Any algorithm solving the decision version of SubsetSum can be used to also output an appropriate subset as follows: Fix an arbitrary element a, and run the algorithm for the decision version (with $t - a$ instead of t) to check if there is a subset with the required sum that contains a. Repeat this process until an element is found that can participate in the subset, and then repeat again on a smaller collection of elements. This procedure eventually finds a solution, and it runs the algorithm for the decision version no more than n^2 times. However, the input to the decision version needs to be changed for the various runs of the algorithm. If the updated input sets are kept in working space, the procedure described above will use linear space.

With a slightly more careful reduction, any space efficient algorithm solving the decision version of SubsetSum can be used to obtain a space efficient algorithm for solving the search and optimization versions of SubsetSum. Hence, the algorithms of [EJT10,Kan10] yield algorithms that solve the search and optimization versions of SubsetSum exactly, in time polynomial in n and t, and space $O(\log n + \log t)$ (on the Turing machine model). This significantly improves the space bound of the classical exact solutions from space $O(t)$ to $O(\log n + \log t)$. Note that this also gives an improvement over the algorithm devised by Lokshtanov and Nederlof [LN10] that achieves a $\tilde{O}(m^2)$ space bound and has running time pseudo-polynomial in n.

1.2 Our Results and Techniques

We start by considering the space complexity of approximating Subset Sum in Section 2. We first show that the decision version of approximate Subset Sum has a simple deterministic 1-pass streaming algorithm. The algorithm takes input $\epsilon, t, \tilde{a}$ and, using space $O(\frac{\log t}{\epsilon})$, decides whether some subset of numbers from the stream add up to a sum in $[(1-\epsilon)t, (1+\epsilon)t]$ (Theorem 1). The algorithm OnePassApproxDecision uses logarithmic space, as long as all numbers involved are polynomial in m, the number of integers in the input. In the unit-cost RAM model, it uses $O(\frac{1}{\epsilon})$ registers. In the strict turnstile streaming model, we give a simpler deterministic 1-pass streaming algorithm that uses $O(\log t + \log m)$ space for the special case when $\epsilon \geq 1/3$ (Theorem 5).

Next, we design algorithms for the search and optimization versions of approximate SubsetSum (Theorems 6 and 8). We achieve polynomial running time (depending on the approximation factor, but independent of the value of t) and obtain small space bounds at the same time. We give space efficient fully polynomial time approximation schemes (FPTAS) for the optimization version of SubsetSum. FPTAS were known for SubsetSum but our algorithms use significantly less space at the expense of only slightly more time. While the previous best FPTAS for SubsetSum run in time $O(\min(m/\epsilon, m + 1/\epsilon^2 \log \frac{1}{\epsilon}))$ and space $O(m + 1/\epsilon)$ on unit-cost RAM [KMPS03], our algorithms run in $O(\frac{1}{\epsilon}m^2)$ time and $O(\frac{1}{\epsilon})$ space on unit-cost RAM, where $1 + \epsilon$ is the approximation factor. This implies constant space FPTAS on unit cost RAMs when ϵ is a constant; where the running time as well as the

space used on unit cost RAMs does not depend on the sizes of the numbers involved at all. On the Turing machine model, we cannot completely eliminate the dependence of the space bounds on the sizes of the numbers, since we have to charge space for writing down even a single number. But we do not use any extra space beyond this "trivial" dependence: on Turing machines our space bounds are of the form $O(\frac{1}{\epsilon}\log t + \log m)$, that is the space used corresponds to storing just a constant number of integers for any constant $\epsilon > 0$. This should be contrasted with previous FPTAS, where the space used was linear in m (the number of integers in the input) even on unit cost RAM.

An additional interesting feature of our search algorithm is that it outputs the co-lexicographically first acceptable subset with respect to the given input ordering, and if an acceptable subset exists within a short prefix of the input, it may run in sublinear time. It can be viewed as a streaming algorithm, with the number of passes depending on the size of the subset found.

A key idea in these algorithms is the observation that the OnePassApproxDecision (from Theorem 1) can be modified to yield interesting space bounds even if the numbers involved are not limited to be polynomial in m. We achieve our results on SubsetSum by repeatedly running OnePassApproxDecision, with an extra trick for the optimization version to avoid all dependence on t (the value of the target sum) in the running time, to ensure fully polynomial running time.

In Section 3, we highlight a connection between approximation algorithms for Partition and finding balanced graph separators. Based on this connection and the space efficient approximation algorithms for SubsetSum, we give space efficient algorithms to construct balanced graph separators (Theorem 9). For graphs with constant size balanced separators our algorithm finds a minimum size separator in polynomial time and logarithmic space. Moreover, we find a separator of the smallest size, without knowing the size of the smallest separator in advance. We define separators and give more details on how our bounds relate to previous results in Section 3.

In Section 4, we consider two variants of SubsetSum where similar algorithms can be designed. We extend our results on SubsetSum to Knapsack problems with equal weights (Theorems 10). This allows us to find the smallest possible acceptable subset for approximating the SubsetSum problem, in a space efficient way (Theorem 11). We also obtain space efficient approximation algorithms for the unbounded Knapsack problem with equal weights, which includes the change-making problem as a special case (Theorems 12,13). See Section 4 for definitions.

We consider the space complexity of exactly solving unary SubsetSum in the streaming model in Section 5. In light of the logspace algorithms [EJT10,Kan10] for unary subset sum, a natural question to ask is: Is it crucial to have access to the entire input at any time in order to be able to solve unary SubsetSum in logspace? In other words: how hard is unary SubsetSum with regard to space when the inputs are read in a stream? The input numbers arrive in a stream, and we want to design a small space algorithm that makes one, or maybe a few, passes over the input stream and decides the instance. We use lower bounds from communication complexity to show that any randomized s-pass streaming algorithm for

unary SubsetSum that makes error bounded by $1/3$ must use $\Omega(\frac{\sqrt{N}}{s})$ space (Theorem 14). In addition, we give a simple deterministic two pass streaming algorithm for unary SubsetSum that uses space $O(\sqrt{N}\log N)$.

In section 6, we leave the uniform computation model and look at SubsetSum from the perspective of monotone circuit complexity. As mentioned earlier, USS is known to be complete for the extremely small circuit class TC^0. Note that SubsetSum is naturally monotone in the following sense: if the number of occurrences of a number i in the stream is increased, a Yes instance remains a Yes instance. Can we therefore construct small-depth monotone circuits for it? To model this monotonicity, we consider the following encoding of USS: For each positive integer B, the input consists of the frequency of each number in the stream in unary. That is, an instance consists of B blocks of B bits each, where the ith block w_i has as many 1s as the number of occurrences m_i of the number i in the stream. Thus, the input records the multiplicity of each number in $[B]$ (without loss of generality, no multiplicity exceeds B). Call this problem the multiplicity-USS problem, mUSS. We show, by a monotone reduction to reachability, that this problem has monotone circuits of polynomial size and $O(\log^2 B)$ depth (Theorem 17). The circuit we construct can also be used to solve the approximate version of USS.

We know that if we are allowed USS as gate primitives (or, as an oracle in an oracle circuit), then we can capture all of TC^0 within polynomial-size and constant depth, since USS is TC^0 complete. In a related vein, we ask: How powerful are ε-approximators when used as gate primitives in circuits? We observe that ε-approximators for mUSS (we call them ApproxUSS gates) are at least as powerful as threshold gates (Lemma 2). Using a technique introduced by Nisan in [Nis94], we also show that any circuit computing the Disjointness function using ε-mUSS gates requires $\Omega(n/\log n)$ gates (Lemma 3). However we have not been able to compare ApproxUSS gates explicitly with Linear Threshold gates.

2 Streaming and Space Efficient Approximation Algorithms for Subset Sum

2.1 Decision Algorithms

To begin with, we consider the streaming model. We wish to approximate SubsetSum using a small number of passes on the input and using space polylogarithmic in the length of the input. For any ϵ and t and input stream $\tilde{a} = a_1, \ldots, a_n$ where each $a_i \in [t]$, we say that set $S \subseteq [n]$ is an *ϵ-approximator of t in $\tilde{a}$* if $\left(\sum_{i\in S} a_i\right) \in [t(1-\epsilon), t(1+\epsilon)]$. Given $\epsilon, t, \tilde{a}$, we want to decide whether there is an ϵ-approximator of t in $\tilde{a}$. We prove the following theorem:

Theorem 1. *There is a deterministic 1-pass streaming algorithm* OnePassApproxDecision *that on an input stream $\epsilon, t, \tilde{a}$ with $0 < \epsilon < 1$, outputs 1 if and only if there exists an ϵ-approximator for t in the stream $\tilde{a}$. The space used by the algorithm is*

1. *$O(\frac{1}{\epsilon})$ in the unit-cost RAM model, and*
2. *$O(\frac{\log t}{\epsilon})$ in the Turing machine model.*

Algorithm OnePassApproxDecision
Maintain a set of intervals T.
Initialize: $T \leftarrow \{[t(1-\epsilon), t(1+\epsilon)]\}$.
while End of stream not reached **do**
  $a \leftarrow$ Next number in stream.
  if $\exists$ interval $[\alpha, \beta] \in T$ such that $a \in [\alpha, \beta]$ **then**
    Output YES and halt.
  else
    $T' \leftarrow \{[\alpha, \beta], [\alpha - a, \beta - a] \mid [\alpha, \beta] \in T\}$;
    $T \leftarrow T'$.
    Merge overlapping intervals in T to get a set of pairwise disjoint intervals.
    (If $[a, b], [c, d] \in T$ and $a \leq c \leq b \leq d$, remove $[a, b], [c, d]$ and add $[a, d]$.)
  end if
end while

Proof. Consider the following algorithm: Before seeing why the algorithm is correct, we first consider the space analysis. Note that at the beginning of each iteration, T has a set of disjoint intervals and each interval has size at least $2t\epsilon$. There can be at most $t/(2t\epsilon)$ disjoint intervals from 1 to t, so at any given time, $|T| \leq \frac{1}{\epsilon}$. Since T' has two intervals for each interval of T, $|T'|$ is also $O(\frac{1}{\epsilon})$. Thus the algorithm needs to store the endpoints of at most $O(\frac{1}{\epsilon})$ intervals. In the unit-cost RAM model, each endpoint requires one register, so the space used is $O(|T|) \in O(\frac{1}{\epsilon})$. In the Turing machine model, since $\epsilon < 1$, all endpoints are in the range $\{0, 1, \ldots, 2t\}$, and so the space required to store any one endpoint is $O(\log t)$. Thus the total space used is $O(\frac{\log t}{\epsilon})$.

We now show that algorithm OnePassApproxDecision is correct; that is, it outputs YES if and only if there exists a subset of the input numbers that has sum in $[t(1-\epsilon), t(1+\epsilon)]$. The intuition behind the correctness is the following: We maintain the set of intervals T such that if any number in the union of the intervals in T is seen as input, then there indeed exists a subset that generates t. This is true in the beginning by the way we initialize T. When a number a is read, a copy of each interval in T is shifted down by a to create a new interval. So if a number in any of these new intervals is seen, then it can be combined with a to give a number in one of the older intervals. (The original intervals are also retained, so we can also choose to not use a in creating a subset.) And this property is maintained by updating T with every number seen. Note that no interval in T gets deleted. Intervals in T only get merged into other intervals to become larger intervals and this does not affect the invariant property.

We now describe the proof more formally: For a set of intervals T, define

$$R(T) = \{a \mid \exists [\alpha, \beta] \in T : a \in [\alpha, \beta]\}.$$

That is, $R(T)$ is the union of all the intervals in T. Let $\ell = t(1-\epsilon)$ and $r = t(1+\epsilon)$. Initially, $R(T) = \{a \mid \ell \leq a \leq r\}$.

$\Rightarrow$*:* Assume that OnePassApproxDecision outputs YES. Let T_k denote the collection of intervals after reading k numbers from the stream. OnePassApproxDecision accepts at a stage k when it reads a number $a_k \in R(T_{k-1})$. We show below, by induction on k, that if

$a \in R(T_k)$, then there is a subset of $\{a_1, \ldots, a_k\} \cup \{a\}$ with sum in $[\ell, r]$. This establishes that the YES answers are correct.

In the beginning, T_0 is initialized to $\{[\ell, r]\}$. Thus $a \in R(T_0) \Rightarrow a \in [\ell, r]$.

Now assume that the property holds after reading $k-1$ numbers. That is, if $a \in R(T_{k-1})$, then there is a subset of $\{a_1, \ldots, a_{k-1}\} \cup \{a\}$ with sum in $[\ell, r]$.

If $a_k \in R(T_{k-1})$, the algorithm terminates here and there is nothing more to prove. Otherwise, $a_k \notin R(T_{k-1})$, and the algorithm goes on to construct T_k. The update sets $R(T_k)$ to contain all of $R(T_{k-1})$ as well as all numbers b such that $a_k + b \in R(T_{k-1})$. Now consider an $a \in R(T_k)$. If it also holds that $a \in R(T_{k-1})$, then we can pretend that a_k was not read at all, and using induction, pull out a subset of $\{a_1, \ldots, a_{k-1}\} \cup \{a\}$ with sum in $[\ell, r]$. If $a \notin R(T_{k-1})$, then $a_k + a \in R(T_{k-1})$. By induction, we have a subset of $\{a_1, \ldots, a_{k-1}\} \cup \{a_k + a\}$ with sum in $[\ell, r]$. Hence either way we have a subset of $\{a_1, \ldots, a_{k-1}, a_k\} \cup \{a\}$ with sum in $[\ell, r]$, as desired.

$\Leftarrow$*:* Let S, $|S| = k$, be the first subset of numbers in the input stream that has sum in $[\ell, r]$. That is,

- $S = \{a_{i_1}, a_{i_2}, \ldots, a_{i_k}\}$ for some $i_1 < i_2 < \ldots < i_k$,
- $\sum_{j=1}^{k} a_{i_j} = t - t\lambda$ for some $|\lambda| < |\epsilon|$, and
- there is no such subset in $a_1, \ldots a_{i_k - 1}$.

We will show that OnePassApproxDecision outputs YES on reading a_{i_k}.

To simplify notation, let s_j denote a_{i_j}.

Observe that if a number a enters $R(T)$ at any stage, then it remains in $R(T)$ until the end of the algorithm. This is because an interval is deleted only when an interval containing it is added.

Now we observe the way T gets updated. After reading s_1, $R(T)$ will contain the intervals $\{[\ell, r], [\ell - s_1, r - s_1]\}$. (It may contain more numbers too, but that is irrelevant.) After reading s_2, $R(T)$ will contain $\{[\ell, r], [\ell - s_1, r - s_1], [\ell - s_2, r - s_2], [\ell - s_1 - s_2, r - s_1 - s_2]\}$. Proceeding in this way, and using the above observation that $R(T)$ never shrinks, after reading $s_1, s_2, \cdots, s_{k-1}$, $R(T)$ will contain $[\ell - (s_1 + \cdots + s_{k-1}), r - (s_1 + s_2 + \cdots + s_{k-1})]$. But this interval is the following:

$$\begin{aligned}
&[t(1-\epsilon) - (s_1 + \cdots + s_{k-1}),\ t(1+\epsilon) - (s_1 + \cdots + s_{k-1})] \\
&= [t(1-\epsilon) - (t - t\lambda - s_k),\ t(1+\epsilon) - (t - t\lambda - s_k)] \\
&= [s_k + t\lambda - t\epsilon,\ s_k + t\lambda + t\epsilon] \\
&= [s_k - t(\epsilon - \lambda),\ s_k + t(\epsilon + \lambda)]
\end{aligned}$$

Since $\epsilon > 0$ and $|\lambda| < |\epsilon|$, $s_k \in [s_k + t(\lambda - \epsilon),\ s_k + t(\lambda + \epsilon)]$. Hence the algorithm will output YES when s_k is read. □

We state here some observations concerning algorithm OnePassApproxDecision that will be useful in later sections. These observations can be proved by induction on r.

Definition 3. *Let $S = \{a_1, \ldots, a_m\}$, and $0 < \epsilon < 1$. A subset $S' \subseteq S = \{a_1, \ldots, a_m\}$ is said to be a* good *set with respect to t and ϵ if*

$$(1-\epsilon)t \leq \sum_{a_i \in S'} a_i \leq (1+\epsilon)t.$$

That is, for some $T \subseteq [m]$ that is a ϵ-approximator of t in S, as defined in the beginning of this section, S' is the corresponding set of values $\{a_i \mid i \in T\}$.

Let $S = \{a_1, \ldots, a_m\}$ of m positive integers, $t \in Z^+$, and $0 < \epsilon < 1$. Suppose that algorithm OnePassApproxDecision answers "YES" on input ϵ, t, S, and let a_r be the last element read from the stream before the algorithm stops.

Observation 2 *There exists a good set with respect to t and ϵ that contains a_r.*

Observation 3 *If $(1-\epsilon)t - a_r > 0$, then there is a subset H' of $H = \{a_1, \ldots, a_{r-1}\}$ such that the sum of the elements in H' satisfies*

$$(1-\epsilon)t - a_r \leq \sum_{a_i \in H'} a_i \leq (1+\epsilon)t - a_r.$$

Observation 4 *Let $P = \{a_1, \ldots, a_p\} \subseteq S = \{a_1, \ldots, a_m\}$ be the first p elements of S read when reading the set S as a stream. If P contains a good subset (with respect to a given t' and ϵ') then the algorithm stops with YES (when running it on S with t' and ϵ'), and the last element read is a member of P.*

The strategy in proving Theorem 1 is to maintain a collection of intervals such that a single element in any of these intervals guarantees a YES answer. For $\epsilon \geq 1/3$, a simpler strategy is to maintain the sum of small enough elements. This yields a 1-pass deterministic algorithm even in the strict turnstile model, which we describe next.

First we briefly recall the definition of the turnstile model. Here the input is given as a stream $(\alpha_1, \alpha_2, \ldots, \alpha_n)$, where each α_i is a pair $\langle x, c_i \rangle$, for $x \in \mathbb{Z}$ and $c_i \in \{1, -1\}$.. If $c_i = 1$ then α_i is said to *insert* x; otherwise, it is said to *delete* x. The model is called *strict* turnstile if for every prefix of the input stream and for every element in the input the number of copies inserted is at least as many as the number of copies deleted. Formally, let $I_x = \{i \mid \alpha_i = \langle x, c_i \rangle\}$. Then $\forall i \in I_x : \sum_{j \leq i, j \in I_x} c_j \geq 0$. The turnstile model arises naturally in applications such as handling updates to a *large* database and tracking IP traffic on a network link. Moreover, if the entries/counts are guaranteed to stay non-negative, the model naturally arising is in fact the strict turnstile model. For more details see [Mut05,GKMS01].

Theorem 5. *Let $S = \{a_1, \ldots, a_m\}$ be a finite set of m positive integers. There is a deterministic 1-pass streaming algorithm in the strict turnstile model, that given $t \in Z^+$, $\epsilon \geq 1/3$, and a stream $(\alpha_1, \alpha_2, \ldots, \alpha_n)$, where each α_i is a pair $\langle a, c_i \rangle$, for $a \in S$ and $c_i \in \{1, -1\}$, decides if there is a subset $S' \subseteq S$ and integers $0 \leq b_a \leq \sum_{i \in I_a} c_i$ for $a \in S'$ such that*

$$(1-\epsilon)t \leq \sum_{a \in S'} b_a a \leq (1+\epsilon)t$$

The algorithm runs in linear time and $O(\log t + \log m)$ space on Turing machines. It runs in $O(m)$ time and constant space on unit-cost RAMs.

Proof. The algorithm is as follows.

```
Algorithm TurnstileApproxDecision
Count ← 0; Sum ← 0.
while End of stream not reached do
  ⟨a, c_i⟩ ← Next number in stream.
  if a ≤ (1 + ε)t then
    if a < (1 − ε)t then
      Sum ← Sum + a × c_i
    else
      Count ← Count + c_i
    end if
  end if
end while
if Count > 0 or Sum ≥ (1 − ε)t then
  return Yes.
else
  return No.
end if
```

To see that the algorithm is correct, first note that $Count > 0$ means that at least one element a such that $(1-\epsilon)t \le a \le (1+\epsilon)t$ survived; that is, $\sum_{i \in I_a} c_i > 0$ after reading the whole stream. Any such element forms an acceptable sum in itself.

Next suppose that $Sum \ge (1-\epsilon)t$ in the end. If we also have $Sum \le (1+\epsilon)t$, then the sum of all surviving elements (except those that exceed $(1+\epsilon)t$) is an acceptable sum. If $Sum > (1+\epsilon)t$, then we must have a good surviving subset as well, since the elements contributing to Sum are each less than $(1-\epsilon)t \le 2\epsilon t = (1+\epsilon)t - (1-\epsilon)t$. Thus, some subcollection of them must form an acceptable sum.

Finally, we argue that if both $Count \le 0$ and $Sum < (1-\epsilon)t$, then no acceptable sum can be formed by the surviving elements. $Count \le 0$ means that none of the surviving elements is in the correct range by itself. $Sum < (1-\epsilon)t$ means that the sum of all surviving elements (except those that exceed $(1+\epsilon)t$) is smaller than $(1-\epsilon)t$, but then no subset of the surviving elements can be in the correct range. □

2.2 Search Algorithms

Our approximation algorithms for the search and optimization versions of SubsetSum are based on repeatedly running algorithm OnePassApproxDecision. Unlike in the straightforward reduction to the decision version described in Section 1, we do not keep in working memory the elements found so far, and each time we run the algorithm on the same set S.

First, consider the search for an approximate solution.

Theorem 6. *There is a deterministic algorithm* ApproxSearch *that, given a finite set $S = \{a_1, \ldots, a_m\}$ of m positive integers, $t \in Z^+$, and $0 < \epsilon < 1$, outputs a subset $S' \subseteq S$*

such that

$$(1-\epsilon)t \leq \sum_{a_i \in S'} a_i \leq (1+\epsilon)t$$

if such a subset exists, and otherwise it outputs "NO". The algorithm runs in $O(\frac{1}{\epsilon}mn) = O(\frac{1}{\epsilon}n^2)$ *time, and* $O(\frac{1}{\epsilon}\log t + \log m)$ *space on Turing machines, where* $n = \sum_{i=1}^{m} \log a_i + \log t$*, and in* $O(\frac{1}{\epsilon}m^2)$ *time, and* $O(\frac{1}{\epsilon})$ *space on unit-cost RAMs.*

Furthermore, if an appropriate subset exists among the first r *elements of the set* S*, the algorithm reads only the* r*-element prefix of the input as a stream in at most* r *passes, and terminates within* $O(\frac{1}{\epsilon}r^2)$ *steps on the unit-cost RAM model, and* $O(\frac{1}{\epsilon}rn_r)$ *steps on Turing machines, where* $n_r = \sum_{i=1}^{r} \log a_i + \log t$*.*

Before we prove the theorem, we make some remarks concerning the result itself.

Remark 1. We note that we could also obtain the same worst case bounds using algorithm OnePassApproxDecision as a blackbox, with a careful reduction. However, our current proof yields the additional property that on certain inputs the algorithm may run in sublinear time, and may not have to read the whole input. This is because the algorithm outputs the co-lexicographically first subset that gives an appropriate sum with respect to the given input ordering. If there is such a subset among the first few elements of the input, the algorithm will run in sublinear time, and does not have to read the whole input. In fact, one can argue that the algorithm reads only the shortest possible prefix of the input.

Remark 2. Note that the algorithm runs in polynomial time regardless of t, and it uses constant space on unit cost RAMs for any constant $\epsilon > 0$.

Remark 3. We state our space bounds on Turing machines in terms of $\log m$. Of course this could be replaced by $\log n$ as well since $m \leq n$. However, in general $\log m$ may be significantly smaller than $\log n$ if the input includes very large values for some a_i-s. As noted above, we can simply ignore such values if they are larger than $(1+\epsilon)t$, and we can detect if a given value is too large using $O(\log t)$ workspace.

Proof. (of Theorem 6) The idea is to repeatedly run the algorithm OnePassApproxDecision on the same set S, thus keeping the workspace used by the algorithm small. We only change the values of the target value t' and the approximation factor γ for the various runs. In the algorithm below, we treat OnePassApproxDecision as a function call that returns either No, or a number r where r is the last element read from the stream before deciding Yes.

First, we run algorithm OnePassApproxDecision on S, $\gamma = \epsilon$, and $t' = t$. If the algorithm answers "NO", we stop and output "NO".

Suppose that the algorithm answers "YES" on this input, and let a_r be the last element read from the stream before the algorithm stops. By Observation 2, we know that a_r can be part of a correct output (a good set), so we write it on the output tape, even though we may not have finished finding a good set yet.

If $r = 1$, or if $r > 1$ and also $(1-\gamma)t' \leq a_r \leq (1+\gamma)t'$, then the good set construction is complete and we stop.

Algorithm ApproxSearch
$t' \leftarrow t$; $\gamma \leftarrow \epsilon$; $r \leftarrow$ OnePassApproxDecision(S, t', γ).
if r="No" **then**
  Output No and Halt
else
  Write a_r on output tape.
  while $r > 1$ **do**
    if $(1-\gamma)t' \leq a_r \leq (1+\gamma)t'$ **then**
      Halt.
    else
      $\gamma \leftarrow \frac{\gamma t'}{t'-a_r}$; $t' \leftarrow t' - a_r$; $r \leftarrow$ OnePassApproxDecision(S, t', γ).
      Write a_r on output tape.
    end if
  end while
  Halt.
end if

Otherwise, note that $(1-\gamma)t' - a_r > 0$ must hold. We now update the target t' to $t'' = t' - a_r$, the target sum for the rest of the good set. We also update the approximation factor to γ' so that the intervals $[(1-\gamma)t' - a_r, (1+\gamma)t' - a_r]$ and $[(1-\gamma')t'', (1+\gamma')t'']$ are the same. By Observation 3, we know that the algorithm OnePassApproxDecision with these parameters must terminate with "YES", and if $a_{r'}$ is the last element read, then by Observations 3 and 4, we know that $r' < r$.

We iterate the above process, until $r = 1$ or the element a_r completes the good set construction. Since r strictly decreases in each iteration, the process must terminate in at most m iterations. When the procedure terminates, a good set is listed on the output tape.

Next we estimate the space used. In each iteration, we store at most $O(\frac{1}{\epsilon})$ intervals, and it takes constant space on unit cost RAMs and $O(\log t)$ space on Turing machines to store one interval (by storing its two endpoints). We reuse space for subsequent iterations. The time bound follows since each iteration scans the input once, and processing each newly read integer may take up to $O(\frac{1}{\epsilon})$ steps to update the intervals. □

In the special case of $\epsilon = \frac{1}{3}$, the same idea that was used in Theorem 5 can be used to get an algorithm that is even simpler than what would follow by substituting $\epsilon = \frac{1}{3}$ in Theorem 6. Not only is the algorithm simpler, it is a 2-pass streaming algorithm.

Theorem 7. *There is a deterministic 2-pass streaming algorithm, that given a finite set $S = \{a_1, \ldots, a_m\}$ of m positive integers and $t \in Z^+$, outputs a subset $S' \subseteq S$ such that*

$$\frac{2}{3}t \leq \sum_{a_i \in S'} a_i \leq \frac{4}{3}t$$

if such subset exists, otherwise it outputs "NO". The algorithm runs in linear time $O(n)$ and $O(\log t + \log m)$ space, where $n = \sum_{i=1}^{m} \log a_i + \log t$, on Turing machines, and in $O(m)$ time and constant space on unit-cost RAMs.

Proof. During the algorithm we will only keep in working memory the value of a current sum s and some counters.

During the first pass, we start by initializing s to 0. For each $i = 1, \ldots, n$, we look at the value a_i on the input tape. There are 3 cases:

1. If $a_i > \frac{4}{3}t$ we move to the next element.
2. If $\frac{2}{3}t \leq a_i \leq \frac{4}{3}t$ we output a_i and stop.
3. If $a_i < \frac{2}{3}t$, we let $s = s + a_i$ Then, if $s < \frac{2}{3}t$ we move to the next element. If we run out of elements and still have $s < \frac{2}{3}t$, we stop and output NO. If $\frac{2}{3}t \leq s \leq \frac{4}{3}t$ we move to the second pass, see below. Note that $s > \frac{4}{3}t$ is not possible, since the previous sum was less than $\frac{2}{3}t$ and we added a value less than $\frac{2}{3}t$.

The only reason we need a second pass is because during the first pass we did not know yet if we have to answer NO or output a set, and the output tape is write only.

In the second pass we proceed exactly as in the first pass, but in addition, every time we add an element a_i to the sum, we also write it on the output tape. The algorithm stops when we reach for the first time a value $s \geq \frac{2}{3}t$. At this point, we also have the members of a good subset listed on the output tape.

We argue that the algorithm finds a good subset if one exists as follows. Elements larger than $\frac{4}{3}t$ cannot participate in a good subset, and if a single element a_i is in the correct range we find it during the first scan. If no such element exists, than all members of a good subset must be less than $\frac{2}{3}t$. If the first scan terminates with s still less than $\frac{2}{3}t$ that means that the sum of all the elements less than $\frac{2}{3}t$ is a value also less than $\frac{2}{3}t$. But then the sum of any subset of these elements is also less than $\frac{2}{3}t$, and no good subset exists. □

We can extend this argument to other values of ϵ and keep the running time linear in n for small values of $\frac{1}{\epsilon}$. In particular, we still get a 2-pass algorithm for $\epsilon = \frac{1}{5}$ with $O(\log t + \log m)$ space. But we cannot keep the algorithm linear time and logarithmic space for arbitrary constant ϵ. As $\frac{1}{\epsilon}$ gets larger, our quadratic time algorithm of Theorem 6 gives better running time, while still using logarithmic space on Turing machines and constant space on unit cost RAM, for arbitrary constant ϵ.

2.3 FPTAS for the optimization version of SubsetSum

We now describe an FPTAS for the optimization question.

Theorem 8. *There is a deterministic approximation scheme for* SubsetSum *optimization, that for any $\epsilon > 0$ achieves approximation factor $1 + \epsilon$, and runs in time $O(\frac{1}{\epsilon}n^2)$ and space $O(\frac{1}{\epsilon}\log t + \log m)$ on Turing machines, where $n = \sum_{i=1}^{m} \log a_i + \log t$, and in $O(\frac{1}{\epsilon}m(m + \log m + \log \frac{1}{\epsilon}))$ time and $O(\frac{1}{\epsilon})$ space on unit-cost RAMs.*

Proof. At the outset, we compute the sum of all elements a_i that do not exceed t. If all elements exceed t (or if the set S is empty) we stop and output NO. If the above sum is nonzero and it is smaller than t then this is the optimal sum and there is nothing else to do. Thus, in what follows we may assume that the sum of all elements a_i that do not exceed t is nonzero, and t is bounded above by this sum. On the other hand, the optimal sum t^* is at least as large as the largest a_i not exceeding t. This implies $t^* \geq t/m$. We will use this fact later when we analyze the running time.

Given an instance of SUBSETSUM optimization with S, $t \in Z^+$, and $\epsilon > 0$, we run the following algorithm after the preprocessing described above.

Algorithm FPTASOPTIMIZATION
Initialize: $i \leftarrow 0$; $c_0 \leftarrow \frac{t}{2}$; $\xi_0 \leftarrow \frac{1}{2}$; $\gamma_0 \leftarrow 1$.
/* Maintain the invariants $\gamma_i c_i = \xi_i t$; $c_i - \xi_i t \le t^* \le c_i + \xi_i t \le t$. */
while $\gamma_i > \frac{\epsilon}{2+\epsilon}$ **do**
  $i \leftarrow i + 1$;
  $\xi_i \leftarrow \frac{1}{2}\xi_{i-1}$;
  $t_i \leftarrow c_{i-1} + \xi_i t$;
  $r_i \leftarrow \frac{\xi_i t}{t_i}$;
  if ONEPASSAPPROXDECISION(S, t_i, r_i) =YES **then**
    $c_i \leftarrow t_i$; $\gamma_i \leftarrow r_i$;
  else
    $c_i \leftarrow c_{i-1} - \xi_i t$; $\gamma_i \leftarrow \frac{\xi_i t}{c_i}$;
  end if
end while
$\ell \leftarrow i$; $t' \leftarrow c_\ell$; $\gamma \leftarrow \gamma_\ell$.
Run APPROXSEARCH on S, t', γ to obtain subset T.
return T.

The while loop implements a binary search procedure for an approximation to the optimal sum t^*, by repeatedly calling the algorithm ONEPASSAPPROXDECISION with varying parameters. In the while loop at stage i, initially the optimal is known to lie in the interval of width $2\xi_{i-1}t$ centred around c_{i-1}, and we want to check whether it is in the upper half of this interval. Since the width halves at each stage, this is precisely the interval $[c_{i-1} \pm \xi_i t]$, and the approximation parameter r_i is chosen so that it is also the same as the interval $[t_i(1 \pm r_i)]$. With these parameters, we call ONEPASSAPPROXDECISION. If the answer is YES, we can set c_i, γ_i to t_i, r_i respectively. Otherwise, t^* lies in the lower half of the previous interval, namely, $[c_{i-1} - xi_i t \pm \xi_i t]$, so c_i and γ_i are chosen accordingly. Thus at the beginning of each stage we maintain the invariants that $c_i\gamma_i = \xi_i t$ and $c_i - \xi_i t \le t^* \le c_i + \xi_i t \le t$. Thus $c_i = \hat{t} - \gamma_i t_i$, where $\hat{t}$ is the upper end of the current target-search interval. That is, sums larger than $\hat{t}$ have already been eliminated from consideration. Throughout, we have $\hat{t} \le t$.

Let ℓ denote the value of the index i when we exit the while loop, that is $\gamma_\ell \le \frac{\epsilon}{2+\epsilon}$ for the first time. We will show later that $O(\log m + \log \frac{1}{\epsilon})$ iterations will suffice. Thus our running time will not depend on t. (Note that with a more straightforward reasoning we would get up to $\log t$ iterations, which we want to avoid.) Setting $t' = c_\ell$ and $\gamma = \gamma_\ell$, the invariants tell us that

$$(1 - \gamma)t' \le t^* \le (1 + \gamma)t' \le t.$$

where t^* denotes the value of the optimal sum.

At this point, we know that there exists a good subset S' of S with respect to t' and γ, that is

$$(1 - \gamma)t' \le \sum_{a_i \in S'} a_i \le (1 + \gamma)t'.$$

Next we run our ApproxSearch algorithm from Theorem 6 that outputs such a subset. Note that we have chosen the parameters so that $(1-\delta)t^* \leq (1-\gamma)t'$. By definition, we cannot find a sum larger than the optimal sum. Thus we obtain an approximation scheme with approximation factor $\frac{1}{1-\delta} = 1+\epsilon$.

It remains to estimate ℓ. Recall, that by our initial remarks, $t^* \geq t/m$. Thus, $c_\ell = t' \geq \frac{1}{1+\gamma}\frac{t}{m}$, and $\frac{t}{c_\ell} \leq (1+\gamma)m \leq 2m$. Recall that $\gamma_\ell = \xi_\ell \frac{t}{c_\ell}$, and we stop as soon as $\gamma_\ell \leq \frac{\epsilon}{2+\epsilon}$, that is, we stop as soon as $\xi_\ell \leq \frac{\epsilon}{2+\epsilon}\frac{c_\ell}{t}$. Hence we certainly stop by the stage when $\xi_\ell \leq \frac{\epsilon}{2+\epsilon}\frac{1}{2m}$. Thus, we stop after at most $\ell = O(\log m + \log \frac{1}{\epsilon})$ iterations. □

3 An Application: Finding Balanced Separators

Balanced separators are used in VLSI circuit design (Ullman [Ull84]), algorithms on planar graphs (Lipton and Tarjan [LT79,LT80], Miller [Mil86]), graphs with excluded minors (Alon, Seymour, and Thomas [AST90]), and in general in algorithmic applications on graphs that involve a divide and conquer strategy.

Informally, a *balanced* separator of a graph G is a set of nodes whose removal yields two disjoint subgraphs of G that are comparable in size. More formally, balanced separators are defined as follows.

Definition 4. *(Lipton and Tarjan [LT79]) Let $G = (V, E)$ be an undirected graph. A set of nodes $S \subseteq V$ is a* balanced separator *of G if the removal of S disconnects G into two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, that satisfy $|V_i| \geq \frac{1}{3}|V|$ for $i = 1, 2$.*

We say that S is a balanced h-separator *of G if S is a balanced separator of G and S consists of h vertices. We say that G is* h-separable*, if it has a balanced h-separator.*

Note that the definition can be easily extended to directed graphs. Also, instead of requiring that the size of both components is between $\frac{1}{3}|V|$ and $\frac{2}{3}|V|$, we can consider components with size closer to $\frac{1}{2}|V|$.

Definition 5. *Let $G = (V, E)$ be an undirected graph. A set of nodes $S \subseteq V$ is a* ϵ- balanced separator *of G if the removal of S disconnects G into two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, that satisfy $|V_i| \geq (1-\epsilon)\frac{|V|}{2}$ for $i = 1, 2$.*

The two subgraphs G_1 and G_2 could be disconnected within themselves. This implies that any graph $G = (V, E)$ has a ϵ-balanced separator of size $|V|$, for any constant $1 > \epsilon > 0$.

One special case of the SubsetSum problem is the Partition problem, which was also listed among Karp's first 21 NP-complete problems [Kar72]. The problem is defined as follows. Given a finite set S of positive integers, decide if there is a subset $S' \subseteq S$ such that $\sum_{a_i \in S'} a_i = \sum_{b_i \in S \setminus S'} b_i$. For the search version, output such a subset if it exists. In the SubsetSum problem, if we set $t = \frac{A}{2}$, where $A = \sum_{a_i \in S} a_i$, then we get the Partition problem. As for the SubsetSum problem, the classical solution for Partition is to use dynamic programming (Dantzig [Dan57], Bellman [Bel57]), which runs in time $O(mt)$ and space $O(t)$.

We observe that space efficient approximation algorithms for the decision version of the Partition problem yield space efficient algorithms to find balanced separators.

Theorem 9. *There is a deterministic algorithm, that given an undirected graph $G = (V, E)$ and $0 < \epsilon < 1$ as input, outputs a smallest-size ϵ-balanced separator. If the smallest ϵ-balanced separator has size h, the algorithm runs in time $O(\frac{1}{\epsilon}|V|^{(h+O(1))})$, and uses space $O((h + \frac{1}{\epsilon}) \log |V|)$.*

Some comments before we formally prove the theorem: Note that the value h is not provided as input to the algorithm, and the algorithm outputs a separator of the smallest possible size. The algorithm runs in polynomial time and logarithmic space for graphs with constant size balanced separators.

As far as we know, this is the first space efficient algorithm that finds balanced separators in the sense of Definition 4 for general graphs. In the case of planar graphs, [INP+13] gave a polynomial time $O(n^{1/2+\epsilon})$ space algorithm to find balanced separators. In [EJT10], Elberfeld, Jakoby, and Tantau considered a different notion of separators, (also commonly referred to as balanced separators in the literature) that we call *component-wise* balanced separators, to distinguish them from the definition of Lipton and Tarjan [LT79] that we use.

Definition 6. *[EJT10] Let $G = (V, E)$ be an undirected graph. A set of nodes $S \subseteq V$ is a* component-wise *balanced separator of G if the removal of S disconnects G into connected subgraphs $G_i = (V_i, E_i)$ that satisfy $|V_i| \leq \frac{1}{2}|V|$ for each i.*

Elberfeld, Jakoby, and Tantau [EJT10] observed the following simple space efficient algorithm for finding *component-wise* balanced separators. Enumerate all possible subsets S of V first by size and then by lexicographic order for each size, starting from subsets of size 1. For each subset S use Reingold's algorithm [Rei08] that solves undirected connectivity in logspace to compute the size of each component in the graph obtained by removing S from G. Output the first subset such that the size condition is satisfied by each component. It is clear that this algorithm can compute the component-wise balanced separator with minimum size, and uses $O(h \log |V|)$ space if the smallest component-wise separator has size h.

In particular, this algorithms runs in logspace for graphs that have constant size component-wise separators. This was used in [EJT10] to find the tree decomposition of graphs with constant tree-width in logarithmic space.

However, for finding balanced separators in the sense of Definition 4, we also need to check whether the components obtained after removing a given subset of vertices can be grouped into two subsets of vertices, such that each has size at least $(1 - \epsilon)\frac{|V|}{2}$. We observe that this corresponds to solving an instance of the ϵ-approximate Partition problem (a special case of ϵ-approximate SubsetSum), where the set $S = \{a_i\}$ consists of the sizes of the components $a_i = |V_i|$.

Moreover, since the values a_i are sizes of subsets of vertices of the graph, we are guaranteed that the value $A = \sum_i a_i$ is polynomial in $n = |V|$. Thus, our space efficient algorithms for approximating the SubsetSum and Partition problems yield space efficient algorithms for finding balanced separators.

Since for finding balanced separators we only need to check if a subset of components with the appropriate sum of sizes exists, algorithm OnePassApproxDecision, that solves the decision version of approximate SubsetSum, can be used.

Putting all this together we obtain the proof of Theorem 9.

Proof. (of Theorem 9.) We define three logspace bounded Turing machines, to perform the following tasks. Note that each machine uses the original graph G as input together with some additional inputs, including the list of vertices in a fixed set W. Other than the space used for storing their respective inputs, each machine uses at most $O(\log |V|)$ workspace, where V is the set of vertices of V. Moreover, the length of the output of each machine is $O(\log |V|)$.

For an undirected graph $G = (V, E)$, a set $W \subseteq V$ and a node $v \in V \setminus W$ we denote by G_W the graph obtained from G by removing the vertices in W and all edges adjacent to vertices in W. We denote by $G_W(v)$ the connected component of G_W that contains v.

The first Turing machine M_1 on input G, W and v computes the number of nodes connected to v in G_W, that is, M_1 outputs $|G_W(v)|$. This can be done by using Reingold's logspace algorithm for undirected connectivity [Rei08] to check, one-by-one, for each node $u \in V \setminus W \setminus \{v\}$ if u is connected to v in G_W. We let Reingold's algorithm run on the original graph G (we do not store the representation of G_W), we simply ignore edges of G adjacent to vertices in W when running the algorithm. We reuse space for each repetition.

M_2 on input G and W outputs the number of connected components of G_W. Note that there is always at least one component, so we start by initializing a counter to $count = 1$. We assume that the first vertex belongs to the first component. Next, for $i = 2, \ldots, |V \setminus W|$, we check (using Reingold's algorithm) if $v_i \in V \setminus W$ is connected to any of the vertices $v_j \in V \setminus W$ for $1 \leq j < i$. If v_i is not connected to any of the previously considered vertices, we increase the counter, and move on to considering the next vertex. If v_i is connected to some previously considered vertex, then we move on to the next vertex without increasing the value of *count*. When all vertices in $V \setminus W$ have been considered, the value of *count* equals to the number of connected components of G_W.

Let G_W^ℓ denote the ℓ-th connected component of G_W, that is $G_W^1 = G_W(v_1)$, and $G_W^\ell = G_W(v_i)$, where i is the smallest integer such that v_i is not contained in any of the components $G_W^1, \ldots, G_W^{\ell-1}$.

M_3 on input G, W and ℓ computes the size of the ℓ-th connected component $|G_W^\ell|$, as follows. Initialize a counter to $count = 1$. Next, similarly to M_2, for $i = 2, \ldots, |V \setminus W|$, we check (using Reingold's algorithm) if $v_i \in V \setminus W$ is connected to any of the vertices $v_j \in V \setminus W$ for $1 \leq j < i$. If v_i is connected to some previously considered vertex, then we move on to the next vertex without increasing the value of *count*. If v_i is not connected to any of the previously considered vertices, we increase the counter by 1, and then if $count < \ell$, move on to considering the next vertex. If $count = \ell$, we let $v = v_i$, and use M_1 on input G, W and v to compute $|G_W^\ell| = |G_W(v)|$.

We are now ready to describe the algorithm. We consider integers $1 \leq k \leq |V|$ in increasing order, and for each k we consider subsets $W \subseteq V$ of size k in lexicographic order. For fixed k and W, we use $k \log |V|$ workspace to keep the list of vertices in W. We use M_2 on G and W to compute the number of connected components of G_W. We denote this number by m. For $\ell = 1, \ldots, m$, let a_ℓ denote $|G_W^\ell|$, the size of the ℓ-th connected component.

We now need to check whether the connected components can be grouped into two subsets of vertices, such that each has size at least $(1-\epsilon)\frac{|V|}{2}$. That is, we need to solve an instance of ϵ-approximate SubsetSum on input $S = \{a_\ell\}$ and $t = \frac{|V|}{2}$. However, we do not have enough work space to explicitly keep the list of integers in S.

We run algorithm OnePassApproxDecision (the decision algorithm for solving ϵ-approximate SubsetSum) on $S = \{a_\ell\}$ and $t = \frac{|V|}{2}$, by using M_3 to compute the value a_ℓ when the algorithm needs it. Note that since the values a_i are sizes of subsets of vertices of the graph, we are guaranteed that the value $A = \sum_i a_i$ is polynomial in $|V|$. Thus, the space used for solving approximate SubsetSum is always bounded by $O(\log |V|)$. Storing the sets W takes $O(h \log |V|)$ space, where h is the size of the smallest ϵ-balanced separator. □

Note that our running time is mostly dominated by the time we spend enumerating the $\binom{|V|}{k}$ possible subsets of size $k \leq h$. All other computations take polynomial time. For graphs with constant size balanced separators our algorithm finds a minimum size separator in polynomial time and logarithmic space.

Even the simpler problem of finding minimum size *component-wise* balanced separators is NP-hard for general graphs [HR73], and as far as we know, our running time is comparable to the running time of previously known algorithms to find minimum size balanced separators. However, our algorithm is more space efficient than previous solutions.

4 Extensions to Other Variants of Knapsack

We also consider variants of the more general Knapsack problem. Similarly to SubsetSum one can consider decision, search, and optimization versions. The *decision version* of the 0-1 Knapsack problem can be defined as follows. Given a capacity $c \in Z^+$, a target value $t \in Z^+$, and a set of size m, where each element has value a_i and weight w_i, output YES if and only if there is a subset $S \subseteq [m]$ such that $\sum_{i \in S} a_i = t$ and $\sum_{i \in S} w_i \leq c$. In the search version, we have to output the subset S if it exists. The *optimization version* is defined as follows. Given a capacity $c \in Z^+$, and a set of size m, where each element has value a_i and weight w_i, find a subset $S \subseteq [1, m]$ such that $\sum_{i \in S} w_i \leq c$ and $\sum_{i \in S} a_i$ is as large as possible. Note that the optimal value is not known in advance. An equivalent definition of the optimization version is to maximize $\sum_{i=1}^{m} a_i x_i$ with the constraints $\sum_{i=1}^{m} w_i x_i \leq c$ and $x_i \in \{0, 1\}$ for $1 \leq i \leq m$. Note that we get the SubsetSum problem as a special case of 0-1 Knapsack by letting $w_i = a_i$ and $c = t$. A common generalization of the 0-1 Knapsack problem is the *integer* Knapsack *problem*, where every x_i is a non-negative integer. This is also known as the *unbounded* Knapsack *problem*, since the number of times we can pick each item is unbounded.

An interesting variant is the *change-making problem*. The decision version is defined as follows. Given positive integers a_i for $1 \leq i \leq m$, t, and c, output YES if and only if $\sum_{i=1}^{m} a_i x_i = t$ and $\sum_{i=1}^{m} x_i \leq c$, where $x_i \geq 0$ is an integer. This corresponds to paying a fixed amount t using a limited number of coins, where a_i are the coin denominations. Note that this is a special case of the unbounded Knapsack problem by letting $w_i = 1$. The change-making problem was first studied by Lueker [Lue75], who showed that the problem

is NP-hard. It was later studied by Kozen and Zaks [KZ94], and Adamaszek and Adamaszek [AA10]. Also see Martello and Toth [MT90], and Kellerer and Pferschy and Pisinger [KPP04] for a comprehensive coverage of the problem.

As we noted in Remark 1, the algorithm in Theorem 6 finds the co-lexicographically first subset with acceptable weight. This yields the property that the algorithm reads the shortest possible prefix of the input: it has to read at least as far as the last element of the subset with the earliest last element with respect to the given ordering of the input.

However, this does not mean that the algorithm finds a good subset with the smallest possible number of elements. For example there may be a good subset of size k among the first r input elements, but the last $k' < k$ elements might also form a good subset, which will not be found, since the algorithm will not read further than the first r elements of the input sequence. This may be reasonable, if faster running time is more important.

We can modify our algorithm to always find the smallest possible good subset, but for this we always have to read the whole input, so this version will never run in sublinear time.

The problem of finding a good subset with fewest elements is in fact another variant of the more general Knapsack problem: the (search version) of the 0-1 Knapsack problem with weights $w_i = 1$ and capacity $c = k$, corresponds to finding a good subset of size at most k.

4.1 0-1 Knapsack with Equal Weights

To keep the notation simple we present our results for $w_i = 1$, but they directly generalize to any Knapsack instance where all weights are equal.

Theorem 10. *There is a deterministic one pass streaming algorithm, that given an integer $m \geq k > 0$, a finite set $S = \{a_1, \ldots, a_m\}$ of m positive integers, $t \in Z^+$, and $0 < \epsilon < 1$, decides if there is a subset $S' \subseteq S$ of size at most k such that*

$$(1 - \epsilon)t \leq \sum_{a_i \in S'} a_i \leq (1 + \epsilon)t.$$

The algorithm runs in $O(\frac{1}{\epsilon}kn)$ time and $O(\frac{1}{\epsilon}k \log t + \log m)$ space on Turing machines, and in $O(\frac{1}{\epsilon}km)$ time and $O(\frac{1}{\epsilon}k)$ space on unit-cost RAMs.

Moreover, if a good subset of size at most k exists among the first r elements of the set S, the algorithm terminates within $O(\frac{1}{\epsilon}kr)$ steps on the unit-cost RAM model, and $O(\frac{1}{\epsilon}kn_r)$ steps on Turing machines, where $n_r = \sum_{i=1}^{r} \log a_i + \log t$.

Proof. (Sketch.) Similarly to algorithm OnePassApproxDecision, we maintain a collection of intervals. In addition, for each interval, we assign an index, indicating the length of partial sums associated with the interval. At the beginning of the algorithm, there is only one interval and its index is 0. As we proceed, whenever we form a new interval, its index will be the index of the interval from which the new interval is formed plus 1. We also merge overlapping intervals like before, but now we only merge intervals that have the same index. This way, every partial sum associated with a given interval has the same size, and the index

of a given interval corresponds to the number of elements participating in the possible partial sums associated with the interval. More precisely, if the index of an interval is j, and we find that some $a_i \in S$ belongs to the interval, then we know that there is a good sum using a_i and some other j elements a_ℓ with $\ell < i$.

Also, whenever creating a new interval would result in an index greater than or equal to k, we do not create the interval, and continue with the algorithm. This is because creating an interval with index at least k would only be useful for detecting sums of size greater than k.

This way, the space bound is increased by a factor of k, and we maintain up to $\frac{1}{\epsilon}k$ intervals. The algorithm stops with yes if it finds that some a_r belongs to one of the existing intervals. In this case, a_r is the last element read from the input, and the rest of the input is never read. The algorithm stops with "NO" if it read all of the input and none of the elements belonged to an existing interval. The bounds on running time follow since we read each input element at most once, and for each input we perform at most $O(\frac{1}{\epsilon}k)$ comparisons to check if it belongs to one of the intervals. □

We can use this algorithm to find the smallest good subset without knowing its size in advance in a space efficient way.

Theorem 11. *There is a deterministic algorithm, that given a finite set $S = \{a_1, \ldots, a_m\}$ of m positive integers, $t \in Z^+$, and $0 < \epsilon < 1$, outputs the smallest subset $S' \subseteq S$ such that*

$$(1-\epsilon)t \leq \sum_{a_i \in S'} a_i \leq (1+\epsilon)t$$

if such subset exists, otherwise it outputs "NO". If the size of the smallest good subset is k, the algorithm runs in $O(\frac{1}{\epsilon}k^2 n)$ time and $O(\frac{1}{\epsilon}k \log t + \log m)$ space on Turing machines, and in $O(\frac{1}{\epsilon}k^2 m)$ time and $O(\frac{1}{\epsilon}k)$ space on unit-cost RAMs.

Proof. (Sketch.) We run the algorithm from Theorem 10 repeatedly for increasing k' starting from $k' = 1$. If a good set of size k' exists, the algorithm will stop with "YES". If the smallest good set has size k, we will get a "YES" answer during the k-th repetition of running the algorithm.

After this, another $k-1$ repetitions allow us to output the good set, similarly to the proof of Theorem 6, by running the algorithm of Theorem 10 repeatedly, this time for decreasing k' starting from $k' = k - 1$. □

4.2 Change-Making Problem: Unbounded Knapsack with Equal Weights

In the SubsetSum problem, as well as in the 0-1 Knapsack problem, each element a_i can be used at most once to contribute to the sum. The unbounded Knapsack problem allows to use the same element a_i several times towards the sum.

If such repetitions are allowed, then we can save a factor of k in the space bounds, where k is the number of elements (counted with multiplicities) contributing towards the sum. The running time remains the same, but there is a catch: in the case of 0-1 Knapsack the size

k of a good subset must be always at most m, since each element can be used at most once. Hence our algorithms for 0-1 KNAPSACK with equal weights are always polynomial time. But if repetitions are allowed, then in general k may be much larger than m, so our algorithms are not polynomial time in general: they are polynomial in k and n (and in k and m on unit-cost RAMs). On the other hand, since the space bound does not depend on k, they run in constant space on unit-cost RAMs for any constant approximation factor.

This version of unbounded KNAPSACK with weights $w_i = 1$ is sometimes called the "Change-Making problem" (see introduction). In the terminology of the Change-Making problem k (the size of the subset counted with multiplicities) corresponds to the number of coins used. To keep the notation simple we present our results for $w_i = 1$, that is for the Change-Making problem, but they directly generalize to any unbounded KNAPSACK instance where all weights are equal.

Theorem 12. *Given an integer $k > 0$, there is a deterministic k pass streaming algorithm, that given a finite set $S = \{a_1, \ldots, a_m\}$ of m positive integers, $t \in Z^+$, and $0 < \epsilon < 1$, decides if there is a subset $S' \subseteq S$ such that*

$$(1-\epsilon)t \leq \sum_{a_i \in S'} a_i x_i \leq (1+\epsilon)t,$$

where each $x_i \geq 0$ is an integer, and $\sum_{a_i \in S'} x_i \leq k$. The algorithm runs in $O(\frac{1}{\epsilon}kn)$ time and $O(\frac{1}{\epsilon}\log t + \log m)$ space on Turing machines, and in $O(\frac{1}{\epsilon}km)$ time and $O(\frac{1}{\epsilon})$ space on unit-cost RAMs.

Proof. (Sketch.) The proof is similar to the proof of Theorem 10, but now we only need to maintain $O(\frac{1}{\epsilon})$ intervals at a time. First we run the algorithm from Theorem 10 once with $k = 1$. Unless a good subset of size 1 exists, this will read the whole input without finding an element that belongs to the initial interval $[l, r]$. We keep the intervals created by this run in working memory, note that each of these intervals has index 1, except the initial interval $[l, r]$ which has index 0. At this point, we discard the interval with index 0, we will not need it anymore. Next, we proceed similarly to running the algorithm of Theorem 10 with $k = 2$, but now we already have a collection of intervals with index 1 stored in memory. Every time we read a new element a_i, we check if it belongs to one of the intervals with index 1. If yes, we stop with "YES". If not, we create a new interval, but only using the intervals with index 1, thus we only create intervals with index 2 during this pass. If we do not succeed in finding an element that belongs to one of the intervals during this pass, we eliminate all the intervals with index 1 but keep all the intervals with index 2.

During the j-th pass we will check if the elements read belong to one of the intervals with index $j-1$ that we kept from the previous pass. During the j-th pass, we only create new intervals with index j. At the end of the j-th pass, we discard the intervals with index $j-1$, and keep all intervals with index j.

Thus, at any time during the process we keep at most $2\frac{1}{\epsilon}$ intervals. Each pass of the input will take $O(\frac{1}{\epsilon}n)$ time, and we make at most k passes. □

The reason this approach with the smaller space bound could not be used in the proof of Theorem 10 is that this way we cannot prevent including the same element a_i more than once into the sum. The algorithm in Theorem 10 is a one pass algorithm, so we never even consider the same element more than once. But in the algorithm in the proof of Theorem 12 we keep restarting reading the input, and we have no way of knowing if the currently read element a_i has already contributed towards one of the partial sums associated with a given interval.

We can use this algorithm to find the smallest possible k (that is the smallest number of coins in the Change-Making problem) without knowing its value in advance in a space efficient way.

Theorem 13. *There is a deterministic algorithm, that given a finite set $S = \{a_1, \ldots, a_m\}$ of m positive integers, $t \in Z^+$, and $0 < \epsilon < 1$, outputs the smallest integer k, and a sequence $b_1, \ldots, b_k$ such that $b_i \in S$ for $i = 1, \ldots, k$ and*

$$(1-\epsilon)t \leq \sum_{i=1}^{k} b_i \leq (1+\epsilon)t.$$

The algorithm runs in $O(\frac{1}{\epsilon}k^2 n)$ time and $O(\frac{1}{\epsilon}\log t + \log m)$ space on Turing machines, and in $O(\frac{1}{\epsilon}k^2 m)$ time and $O(\frac{1}{\epsilon})$ space on unit-cost RAMs.

Proof. (Sketch.) Similarly to the proof of Theorem 11 we run the algorithm from Theorem 12 repeatedly for increasing k' starting from $k' = 1$. If a good set of size k' exists, the algorithm will stop with "YES". If the smallest good set has size k, we will get a "YES" answer during the k-th repetition of running the algorithm.

After this, another $k-1$ repetitions allow us to output the good set, similarly to the proof of Theorem 6, by running the algorithm of Theorem 12 again repeatedly this time for decreasing k' starting from $k' = k-1$, □

5 Exact Unary Subset Sum in the Streaming Model

We first show lower bounds for 1-pass and many-pass streaming algorithms that decide unary SubsetSum. We do so by lower bounding a communication game corresponding to SubsetSum. A communication game corresponding to SubsetSum is as follows: both Alice and Bob are given an integer B. Further, each of them has a set of numbers and they have to determine if there is a subset of numbers among the union of their sets that adds to B. The goal is to minimize the number of bits exchanged between Alice and Bob. Additionally, there may be constraints on how often the communication exchange changes direction (the number of rounds).

A standard lower bound technique (see [AMS99]) shows that an s-pass space $O(S)$ streaming algorithm yields a protocol with communication complexity $O(sS)$ and $2s-1$ rounds. Thus a communication complexity lower bound yields a streaming space lower bound. We use this technique to show that any 1-pass streaming algorithm for unary SubsetSum needs $\Omega(\sqrt{N})$ space.

Lemma 1. *Any deterministic or randomized 1-pass streaming algorithm for unary* SubsetSum *uses space* $\Omega(\sqrt{N})$.

Proof. We reduce the Index problem to SubsetSum. The Index_n function is defined as follows: Alice has $x \in \{0,1\}^n$ and Bob has an index $k \in [n]$. The goal is to find x_k. Alice can send one message to Bob, after which Bob should announce what he believes is the value of x_k. It is known that the 1-way randomized communication complexity of Index_n is $\Theta(n)$ (see [BYJKS02] or [KNR95]).

The reduction from Index_n to SubsetSum is as follows: The target sum is $B = 2n$. Alice creates a set $S = \{2n - i | x_i = 1\}$. Bob creates the set $T = \{k\}$. Since each number in S is at least n, any subset of S that has two or more numbers would have a sum strictly greater than $2n$. Hence any subset of $S \cup T$ that has a sum of $2n$ can have at most one number from S. Now it is easy to see that if $x_k = 1$, the subset $\{(2n - k), k\}$ has sum $2n$. Whereas if $x_k = 0$, there is no subset of $S \cup T$ that has sum $2n$. Thus a protocol that correctly decides the SubsetSum game instance where $B = 2n$, Alice has S and Bob has T, with communication cost c also correctly decides $\text{Index}_n(x, k)$ with communication cost c.

Assume that there is a space $f(N)$ 1-pass streaming algorithm for unary SubsetSum, where f is some non-decreasing function. Then there is a cost $f(N)$ protocol for the SubsetSum game, and hence by the above reduction, a cost $f(\ell)$ protocol for Index_n, where ℓ is the length of the resulting unary SubsetSum instance. Since $\ell \leq n^2$, $f(\ell) \leq f(n^2)$. By the lower bound for Index, $f(n^2) \in \Omega(n)$, and so $f(N) \in \Omega(\sqrt{N})$. □

A generalization of the above proof gives a space lower bound for streaming unary SubsetSum that depends on the number of passes.

Theorem 14. *Any deterministic or randomized s-pass streaming algorithm for unary* SubsetSum *uses space* $\Omega(\sqrt{N}/s)$.

Proof. We give a reduction from $\overline{\text{Disj}_n}$ to SubsetSum. The Disjointness problem Disj_n is defined as follows: for $x, y, \in \{0,1\}^n$, $\text{DISJ}_n(x, y) = \wedge_{i=1}^n \neg(x_i \wedge y_i)$. (That is, if x and y are characteristic vectors of sets $X, Y \subseteq [n]$, then $\text{Disj}_n(x, y) = 1$ if and only if $X \cap Y = \emptyset$.) Its complement $\overline{\text{Disj}_n}$ is the intersection problem. Alice and Bob are given $x \in \{0,1\}^n$ and $y \in \{0,1\}^n$ respectively. The goal is to determine if there exists an $i \in [n]$ such that $x_i = y_1 = 1$. It is known [KS92,Raz92,BYJKS04] that any randomized protocol for $\overline{\text{Disj}_n}$, with any number of rounds, must exchange $\Omega(n)$ bits to bound error probability by 1/3.

The reduction from $\overline{\text{Disj}_n}$ to SubsetSum is as follows: We set the target to $B = 12n - 1$. Alice constructs the set $S = \{8n - 2i | x_i = 1\}$. Bob constructs the set $T = \{4n + 2i - 1 | y_i = 1\}$. Notice that all numbers in S are greater than $B/2$, and that all numbers in T lie in the interval $(B/3, B/2)$. Further note that each number in S is even and that each number in T is odd. We claim that $\overline{\text{Disj}_n} = 1$ exactly when $S \cup T$ has a subset adding to B. To see why, first observe that

1. Using numbers only from S cannot give a sum of B since B itself does not appear in S, and the sum of any two numbers from S exceeds B.
2. Using numbers only from T cannot give a sum of B since

(a) B does not appear in T;
(b) Any two numbers in T add to an even number greater than $B/2$, but B is odd; and
(c) adding three or more numbers from T gives a sum greater than B.

Thus we see that if any subset of $S \cup T$ adds to B, then it must contain exactly one number from S and one from T. That is, it must be of the form $\{8n-2i, 4n+2j-1\}$. To add up to $12n-1$, it must be the case that $i = j$. Hence such a subset exists if and only if there exists an $i \in [n]$ such that $x_i = y_i = 1$.

Now, as in Lemma 1, assume that there is an s-pass streaming algorithm for unary SubsetSum using space $f(N)$, where $f(.)$ is some non-decreasing function. Then there is a cost $(2s-1)f(N)$ protocol for the SubsetSum game with $2s-1$ rounds, and hence by the above reduction, a cost $(2s-1)f(n^2)$ protocol for Disj_n. (The reduction from Disj_n creates unary SubsetSum instances of length $N \in O(n^2)$.) By the lower bound for Disj_n, $(2s-1)f(n^2) \in \Omega(n)$, and so $f(n) \in \Omega(\sqrt{N}/s)$. □

Next we show that the above lower bounds are close to tight, by giving a deterministic two pass algorithm using $O(\sqrt{N}\log N)$ space.

Theorem 15. *There is a deterministic 2-pass streaming algorithm for unary* SubsetSum *that uses space $O(\sqrt{N}\log N)$ and runs in polynomial time.*

Proof. In the first pass, we simply read the input to learn its length, that is the value of N, as well as the value B. We write these down using $O(\log N)$ space. Next, we initialize an array C of size $\sqrt{N}$ to all zeros. We will use the entries of this array as counters, with values at most N, so we reserve $\sqrt{N}\log N$ space for the entries of the array. In addition we will write a list of numbers, using space not occupied by the array.

In the second pass, whenever we see a number $a_i \leq \sqrt{N}$, we increment the entry $C[a_i]$ of the array by one. If we see a number $a_i > \sqrt{N}$, we append its value (in binary) to the list after the array.

After finishing the second pass, we have all of the input sequence stored locally in an implicit form using space $O(\sqrt{N}\log N)$. Next, we run the algorithm of Kane [Kan10] on this locally stored input. This will take at most $O(\log N)$ additional space, and time polynomial in N. □

We also observe that there are two simple one pass streaming algorithms for SubsetSum that can be used for unary SubsetSum as well as SubsetSum when the input is represented in binary. We assume that B is the first number in the stream. The first algorithm uses space $O(m \log B)$ storing those input numbers that do not exceed B. The other algorithm uses space $O(B \log m)$ storing how many times each integer in $[B]$ appears in the input. Then both algorithms run Kane's algorithm [Kan10] on the locally stored implicit input representation. Note however that these bounds are incomparable with $\sqrt{N}$ in general.

6 Circuit Complexity of Monotone Approximate Unary Subset Sum

6.1 Problem Formulation

In this section, we consider the monotone circuit complexity of USS. Without the monotone restrictions, it is known that USS is complete for the circuit class TC^0 ([EJT12]). However, in a very natural sense, Subset Sum is a monotone problem, and so we can consider monotone circuits for it. The encoding of the input becomes crucial for achieving monotonicity. We choose the following encoding:

For each positive integer B, the input consists of the frequency of each number in the stream in unary. An instance $w \in \{0,1\}^{B^2}$ consists of B blocks of B bits each. For each $k \in [B]$, if k occurs in the stream m_k times, then the kth block w_k has exactly m_k 1s; that is, $\sum_{j=1}^{B} w_{kj} = m_k$. Thus the input records the multiplicity of each number in $[B]$ (we assume that no multiplicity exceeds B).

Define the transitive relation $\preceq$: For $u = (u_1, u_2, \ldots, u_B), v = (v_1, v_2, \ldots, v_B)$ with $u_k, v_k \in \{0,1\}^B$, $u \preceq v$ if and only if $\forall k \in [B]$, $\sum_{j=1}^{B} u_{kj} \leq \sum_{j=1}^{B} v_{kj}$.

We define the multiplicity-USS problem, denoted as mUSS, and its approximation variant ϵ-mUSS, as follows.

$$\begin{aligned}
\mathsf{mUSS}(w,B) = 1 \iff & \exists y = (y_1, y_2, \ldots, y_B) : \\
& y_k \in \{0,1\}^B \quad \forall k \in [B], \quad y \preceq w, \text{ and} \\
& B = \left(\sum_{k=1}^{B} k \left(\sum_{j=1}^{B} y_{kj} \right) \right) \\
\epsilon\text{-}\mathsf{mUSS}(w,B) = 1 \iff & \exists y = (y_1, y_2, \ldots, y_B) : \\
& y_k \in \{0,1\}^B \quad \forall k \in [B], \quad y \preceq w, \text{ and} \\
& B(1-\epsilon) \leq \left(\sum_{k=1}^{B} k \left(\sum_{j=1}^{B} y_{kj} \right) \right) \leq B(1+\epsilon)
\end{aligned}$$

We call such a y a *witness* for (w, B). The vector y represents a subset of the multi-set represented by w such that the elements in y sum to B (or to a number within ϵ of B, respectively).

For example, for $B = 4$, the stream 1 3 2 2 1 4 3 can be encoded by any of the following strings (and by many more): 1100 1100 1100 1000, 1010 0101 0011 0010. Some witnesses for this instance are 1100 1000 0000 0000 (use two 1s and a 2), 0100 0000 0001 0000 (use a 1 and a 3), 0000 0000 000 1000 (use the 4).

Fact 16 mUSS *is a monotone function, i.e. for each positive integer B, and for each $u = (u_1, u_2, \ldots, u_B)$, if* $\mathsf{mUSS}(u, B) = 1$*, and if $v = (v_1, v_2, \ldots, v_B)$ is obtained from u by changing some* 0*s to* 1*s, then* $\mathsf{mUSS}(v, B) = 1$*.*
*Similarly, for each ϵ and B, ϵ-*mUSS*is a monotone function.*

6.2 Monotone Depth Upper Bound

It has been known for over three decades ([MS80]) that USS is in nondeterministic logspace; hence USS reduces to the problem Reach defined below:

> Given: a layered directed acyclic graph G, two designated nodes s, t
> Output: 1 if there is a path from s to t in G, 0 otherwise.

It is well-known that Reach has monotone circuits of depth $O(\log^2 n)$, where n is the number of vertices in the input instance. (This follows from the construction of [Sav70]. See for example [AB09].) We show that with the encoding described above, exact and approximate versions of mUSS reduce to Reach via a very weak form of reduction known as monotone projections.

$f : A \to B$ is called a monotone projection if $\forall x \in A, y \in B$ such that $f(x) = y$, it holds that $\forall i, \exists j$ such that $y_i = x_j$. In other words, a monotone projection produces an output string by either rearranging bits of the input or copying bits of the input or both.

Showing a monotone projection from mUSS to Reach gives us small depth monotone circuits for mUSS.

Theorem 17. *For every positive integer B, $\mathsf{mUSS}(\cdot, B)$ and ϵ-$\mathsf{mUSS}(\cdot, B)$ have monotone circuits of depth $O(\log^2 B)$.*

Proof. We prove this by reducing an instance of mUSS into an instance of Reach via a monotone projection.

The following is the high level idea:
For every integer B, and given $w \in \{0,1\}^{B^2} = (w_1, w_2, \ldots, w_B)$ we create a graph with B^2+1 layers. The zero-th layer consists of the source vertex and the other B^2 layers have $(B+1)$ vertices each. We further partition the set of B^2 layers into B blocks of B consecutive layers each.

Let $v^i_{j,k}$ denote the i-th vertex in the layer j in the block k. Intuitively, each layer corresponds to a bit position in the input string. We add edges in order to ensure that a vertex $v^i_{k,j}$ is reachable from the source vertex if and only if the stream corresponding to the first $k-1$ blocks of w and j bits from the kth block has a subset that adds to i.

If after reading l bits of w there is a subset that adds to i then this subset continues to exist even after reading more bits. To capture this phenomenon, we add *horizontal edges* from every vertex v in layer l to the copy of v in layer $l+1$.

If the bit $w_{kj} = 1$, then using this copy of k, for each existing subset sum s, the subset sum $s+k$ can also be created. To capture this, we include *slanted* edges from each $v^i_{j,k}$ to $v^{i+k}_{j+1,k}$.

Thus, there is a path form the source to vertex $v^i_{B,B}$ exactly when there is a subset that sums to i. By connecting $v^i_{B,B}$ for appropriate i to a new target node t, we reduce mUSS or ϵ-mUSS to Reach.

Formally, the graph for reducing mUSS to Reach is defined as follows. Let $v^i_{j,k}$ denote the i–th vertex in the layer j in the block k, where $0 \le i \le B$, and $j, k \in [B]$. Let $V = \{v^i_{j,k} \mid 0 \le i \le B, 1 \le j, k \le B\} \cup \{v^0_{0,0}\}$. We now describe the edge set for the graph. We first describe

the horizontal edges. Let $E_1 = \{v^i_{j,k} \to v^i_{j+1,k}, \mid 0 \le i \le B, 1 \le j \le B-1, 1 \le k \le B\}$ be the edges that connect the i–th vertex in the layer j, of the block k to the i–th vertex in the layer $j+1$ of the same block. Let $E_2 = \{v^i_{B,k} \to v^i_{1,k+1} \mid 0 \le i \le B, 1 \le k \le B-1, \}$ be the inter-block edges that connect the i–th vertex in the layer B of the block k to the i–th vertex in the first layer of the block $k+1$. Let E^w_1 and E^w_2 denote sets of intra-block and inter-block edges obtained from the input instance w of mUSS, respectively (these are the slant edges):

$$E^w_1 = \left\{v^i_{j,k} \to v^{i+k}_{j+1,k} \mid w_{jk} = 1, 0 \le i \le B-k, 1 \le j \le B-1, 1 \le k \le B\right\}$$
$$E^w_2 = \left\{v^i_{B,k} \to v^{i+k}_{1,k+1} \mid w_{Bk} = 1, 0 \le i \le B-k, 1 \le k \le B-1\right\}$$

The Reach instance we create is:
$\left(G = (V, E = E_1 \cup E_2 \cup E^w_1 \cup E^w_2 \cup \{v^0_{0,0} \to v^0_{1,1}\}), s = v^0_{0,0}, t = v^B_{B,B}\right)$.

Claim. After reading the j–th bit of w_k if $s_1, s_2, \ldots, s_\alpha$ are all possible subset sums which have sum at most B, then the set of vertices reachable from $v^0_{0,0}$ in the j–th layer of the block k is $\{v^{s_1}_{j,k}, v^{s_2}_{j,k}, \ldots, v^{s_\alpha}_{j,k}\}$

Proof. We prove this by induction on j, k. The base case is: $j = 1, k = 1$, i.e. the first bit of w_1 is read. Either $w_{1,1} = 0$ or $w_{1,1} = 1$. In the first case, the possible subset sum is 0. And from our construction, $v^0_{0,0} \to v^0_{1,1}$ is an edge in G. In the latter case, the possible subset sums are $0, 1$. As $w_{1,1} = 1$, there is an edge $v^0_{0,0} \to v^1_{1,1}$ in E^w_1. And $v^0_{0,0} \to v^0_{1,1}$ is also an edge in E. Hence the base case.

After reading j–th bit in w_k, let the possible subset sums be $s_1, s_2, \ldots, s_\alpha$. By induction hypothesis the set of vertices reachable from $v^0_{0,0}$ in the j–th layer of the block k is $\{v^{s_1}_{j,k}, v^{s_2}_{j,k}, \ldots, v^{s_\alpha}_{j,k}\}$. Inductive step –

Case 1 [$j < B$, **i.e. next bit in the same block**]**:** Either $w_{j+1,k} = 0$ or 1. If $w_{j+1,k} = 0$, then the possible subset sums after reading $j+1$–th bit remain unchanged, i.e. $s_1, s_2, \ldots, s_\alpha$. By our construction, $\{v^{s_1}_{j,k} \to v^{s_1}_{j+1,k}, v^{s_2}_{j,k} \to v^{s_2}_{j+1,k}, \ldots, v^{s_\alpha}_{j,k} \to v^{s_\alpha}_{j+1,k}\}$ is a subset of E_1. By induction hypothesis, $v^{s_1}_{j,k}, v^{s_2}_{j,k}, \ldots, v^{s_\alpha}_{j,k}$ are reachable from $v^0_{0,0}$. Therefore, $v^{s_1}_{j+1,k}, v^{s_2}_{j+1,k}, \ldots, v^{s_\alpha}_{j+1,k}$ are reachable. No other vertex in layer $j+1$ is reachable (as E^w_1 has no edges in this layer).

If $w_{j+1,k} = 1$, then the possible subset sums after reading $j+1$–th bit are: $s_1, s_2, \ldots, s_\alpha, s_1 + k, s_2 + k, \ldots, s_\alpha + k$. (If $s_l + k > B$ for some l, then $s_l + k$ need not be considered.) By our construction, $\{v^{s_1}_{j,k} \to v^{s_1+k}_{j+1,k}, v^{s_2}_{j,k} \to v^{s_2+k}_{j+1,k}, \ldots, v^{s_\alpha}_{j,k} \to v^{s_\alpha+k}_{j+1,k}\}$ is a subset of E^w_1. And $\{v^{s_1}_{j,k} \to v^{s_1}_{j+1,k}, v^{s_2}_{j,k} \to v^{s_2}_{j+1,k}, \ldots, v^{s_\alpha}_{j,k} \to v^{s_\alpha}_{j+1,k}\}$ is a subset of E_1. Using the induction hypothesis and these edges, the inductive step follows.

Case 2 [$j = B$, **i.e. next bit in block** $k+1$]**:** The proof of this case is similar to that of Case 1. The arguments goes through by using the induction hypothesis and the inter-block edges from sets E_2, E^w_2. □

For the given instance, $\mathsf{mUSS}(w, B) = 1$ iff after reading the last bit of w_B there is a subset that adds to B iff $v^B_{B,B}$ is one of the reachable nodes from $v^0_{0,0}$ (using Claim 6.2). As

the graph G has $B^2(B+1)+1 = O(B^3)$ vertices, Reach can be computed by a $O(\log^2 B)$ depth monotone circuit. Since the reduction can be performed using projections, the theorem follows. □

Remark 4. A more direct way of seeing this was pointed out to us by an anonymous referee and is as follows: For a set $\{0\} \subseteq U \subseteq \{0, 1, ..., n\}$, let χ_U be its characteristic string of length $n+1$ (so the i-th bit of χ_U is 1 iff $i \in U$). Let $U + V = \{x + y \mid x \in U \wedge y \in V\}$ be the "addition" of two sets. We can compute χ_{U+V} from χ_U and χ_V using a monotone constant depth circuit:

$$\chi_{U+V}(i) = \vee_{j=0}^{i} [\chi_U(j) \wedge \chi_V(i-j)].$$

Now, to test whether S contains a subset summing up to some t, consider the sets $\{\{0, x\} \mid x \in S\}$ and "sum up" their characteristic strings in a tree-like fashion. Then the result has a 1 at position t iff S has a subset summing up to t. Furthermore, we can compute this using a monotone circuit of polynomial size and $O(log^2 B)$ depth. (If unbounded fanin $\vee$ gates are allowed, it has logarithmic depth.)

6.3 Monotone Approximate Subset Sum as a circuit primitive

We now examine the power of ϵ-approximators for mUSS when used as a primitive to compute other functions. In [Nis94], Nisan showed that any circuit for DISJ_n using linear threshold gates requires $\Omega(n/\log n)$ gates. We introduce a new kind of gate, an ApproxUSS gate, that we show is at least as powerful as a Threshold or Majority gate, and show that any circuit that uses ApproxUSS gates to compute Disjointness needs size $\Omega(n/\log n)$. However, we do not know whether linear threshold gates can simulate ApproxUSS gates with at most sub-logarithmic blowup in the number of gates or vice versa.

We define approximate USS gates, denoted ApproxUSS, as gates that solve the ϵ-mUSS problem defined in Section 6. An $\text{ApproxUSS}_{\epsilon,B}$ gate takes a bit string x of length B^2 as input, and outputs 1 exactly when ϵ-mUSS(x, B) =1.

While it is trivial to see that majority can be computed with a single call to an oracle for mUSS, it is not immediately clear that oracle access to ϵ-mUSS when $\epsilon > 0$ is also sufficient. We show that this is indeed the case, by showing that ApproxUSS gates are at least as powerful as standard threshold gates. Specifically, we show that an ApproxUSS gate can simulate majority with only a polynomial blowup in the number of wires.

Lemma 2. *The MAJ_{2n+1} function can be computed by an $\text{ApproxUSS}_{\varepsilon,B}$ gate with $B = O(n^3)$ and a suitable non-zero value for ε.*

Proof. Given $w \in \{0,1\}^{2n+1}$ as an input to MAJ, we come up with a vector $y \in \{0,1\}^{B^2}$ such that $MAJ(w) = 1$ if and only if $\text{ApproxUSS}_{\epsilon,B}(y) = 1$. We will specify B and ϵ, and also another parameter N, shortly. The reduction from w to y is a simple projection given as follows:

The string y we wish to compute represents a stream of numbers. We describe the projection in terms of the stream. If $w_i = 1$, then we add $\lfloor N/i \rfloor$ copies of i to the stream. So $y_{i,j} = 1$ exactly when $w_i = 1$ and $j \leq \lfloor N/i \rfloor$. Thus, if $w_i = 0$, adding all copies of i in y

gives 0, whereas if $w_i = 1$, adding all copies of i in y gives $i \times \lfloor N/i \rfloor$. Note that for every i, $N - 2n \leq N - (i-1) \leq i \times \lfloor N/i \rfloor \leq N$.

Let $\alpha = nN$, $\beta = (n+1)(N-2n)$, and $\gamma = (2n+1)N$. Let M denote the maximum subset sum in the stream corresponding to y. (Add up *all* the numbers in the stream.) If w is a No instance of MAJ, then $M \leq \alpha$. If w is a Yes instance of MAJ, then $\beta \leq M \leq \gamma$.

To turn this into a proper reduction, it suffices to ensure that

- $N \leq B$, so that it is possible to create $\lfloor N/i \rfloor$ copies of i for each i,
- $\alpha < \beta$, so that we can distinguish between yes and no instances,
- The range $[\beta, \gamma]$ coincides with $[B - B\epsilon, B + B\epsilon]$, so that a Yes instance of MAJ gets mapped to a YES instance of ApproxUSS.

We see that the above conditions are met by choosing $N = 2(n+1)^2$, $B = (\beta+\gamma)/2$, and $\epsilon = \frac{\gamma-\beta}{2B} = \frac{\gamma-\beta}{\gamma+\beta}$. This gives $\beta = 2n^3 + o(n^3)$, $\gamma = 4n^3 + o(n^3)$, and hence $B = 3n^3 + o(n^3)$, and $\epsilon = \frac{2n^3+o(n^3)}{6n^3+o(n^3)} \sim 1/3$. □

On the other hand, it is not known whether ApproxUSS, for $\epsilon \neq 0$, can be decided with a single oracle call to majority. It is conceivable that demanding a YES answer for a wider range of inputs (the approximation) makes the problem harder. It is therefore interesting to examine the power of circuits using ApproxUSS gates. We follow this thread below.

The communication problem ccApproxUSS corresponding to ApproxUSS can be described as follows. Let $S \subseteq [B^2]$. Both Alice and Bob know S, B and ϵ. Alice knows the bits x_i for $i \in S$, and Bob knows the remaining bits of x. They must decide whether $\mathsf{ApproxUSS}_{\epsilon,B}(x) = 1$, that is, whether $\epsilon\text{-}\mathsf{mUSS}(x, B) = 1$.

In Theorem 1 we proved that for every ϵ, there is a one-pass streaming algorithm that ϵ-approximates USS using space $O((\log B)/\epsilon)$. The algorithm works for every possible ordering of the numbers in the input stream. This implies that there is a $O((\log B)/\epsilon)$ bit one-round protocol for ccApproxUSS for worst case partitioning of the input (for every $S \subseteq [B^2]$). (The string x determines the multi-set of numbers. For any partition of x, Alice and Bob can construct a stream of numbers forming this multi-set, where Alice has the initial part of the stream and Bob has the latter part. Treat the indices in B^2 as pairs k, j where k is the block number and j is the index within the block. Alice includes in her stream a copy of k for each bit $x_{kj} = 1$ in her part. Bob does the same.) Therefore, using an argument similar to that of [Nis94], we can prove the following lemma:

Lemma 3. *Let C be a circuit that computes $DISJ_n$ using s $\mathsf{ApproxUSS}_{\varepsilon,B}$ gates, where $\epsilon \in \Theta(1)$, and the value of B at each $\mathsf{ApproxUSS}_{\varepsilon,B}$ is bounded above by a polynomial in n. Then $s \in \Omega(n/\log n)$.*

Proof. Let C be such a circuit, with s ApproxUSS gates. Let t denote the maximum value of B in any of the gates. We use C to obtain a protocol for DISJ_n as follows. Alice and Bob evaluate C bottom-up, reaching a gate only after all its children have been evaluated. At each ApproxUSS gate, we know that $\log B \in O(\log t) \subseteq O(\log n)$. When an ApproxUSS gate has to be evaluated, an efficient protocol of $O((\log t)/\epsilon)$ bits for ccApproxUSS is invoked with the appropriate partition of the inputs of the gate. As there are s ApproxUSS

gates, the entire protocol for computing DISJ_n uses $O(((\log t)/\epsilon) \times s)$ bits of communication. However, we know that any protocol for DISJ_n requires $\Omega(n)$ bits of communication ([KS92,Raz92,BYJKS04]). Hence, $s \log t = \Omega(\epsilon n)$. By assumption, $\epsilon \in \Theta(1)$, and $\log t = O(\log n)$. Hence $s = \Omega(n/\log n)$. □

7 Open Problems

We now discuss a few aspects of our results and some open questions.

- Our algorithms work for multi-set versions, where an element may appear in the input stream more than once. If the multiplicities of all numbers are restricted to be between $\{0, 1\}$, then the problem does not become easier. In fact, our lower bound proof for USS in Section 5 (Theorem 14) generates such instances.
- Our algorithms for 0-1 Knapsack with equal weights are fully polynomial time, but the space bounds depend on k, the size of the smallest good subset. Thus, in general, the space bounds for this problem may be linear in m if k is large (e.g. constant fraction of m). As we noted above, our algorithms for unbounded Knapsack with equal weights eliminate the dependence on k in the space bounds, but the running time is not polynomial, since in this version of the problem k does not have to be bounded by m. It remains open if Knapsack can be approximated in polynomial time and $o(m)$ space simultaneously, even in the case of equal weights.
- We know that USS is in TC^0 [EJT12] and we have proved that mUSS is in monotone NC^2 (polynomial-sized $\log^2$ depth circuits). It is known that there exists a monotone formula of polynomial size which cannot be computed by constant depth polynomial sized monotone threshold circuits [Yao89]. However, the question of whether monotone TC^0 is contained in monotone NC^1 is open (see for instance [Ser04]). Majority is known to be in monotone NC^1 [Val84]. Moreover, it is easy to observe that majority reduces to USS. In the light of these results, the question of whether mUSS is contained in monotone NC^1 is interesting.

Acknowledgements

Some of this collaboration was initiated and continued in the excellent setting of Dagstuhl Seminars 12421 and 14391 on Algebraic and Combinatorial Methods in Computational Complexity and Dagstuhl Seminar 14121 on Computational Complexity of Discrete Problems.

The authors thank the anonymous referees whose comments helped improve the presentation in this paper.

References

AA10. Anna Adamaszek and Michal Adamaszek. Combinatorics of the change-making problem. *European Journal of Combinatorics*, 31:47–63, 2010.

AB09. Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

AMS99. Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

AST90. Noga Alon, Paul D. Seymour, and Robin Thomas. A separator theorem for graphs with an excluded minor and its applications. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 293–299, 1990.

Bel57. Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

BST02. Cristina Bazgan, Miklos Santha, and Zsolt Tuza. Efficient approximation algorithms for the SUBSET-SUMS EQUALITY problem. *J. Comput. Syst. Sci.*, 64(2):160–170, 2002.

BYJKS02. Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, and D. Sivakumar. Information theory methods in communication complexity. In *Proceedings of the 17th Annual IEEE Conference on Computational Complexity*, pages 93–102, 2002.

BYJKS04. Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *Journal of Computer and System Sciences*, 68(4):702–732, 2004. (preliminary verion in FOCS 2002).

CLRS09. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

Dan57. George Dantzig. Discrete-variable extremum problems. *Operations Research*, 5:266–277, 1957.

EJT10. Michael Elberfeld, Andreas Jakoby, and Till Tantau. Logspace versions of the theorems of Bodlaender and Courcelle. In *FOCS*, pages 143–152, 2010. ECCC TR 62, 2010.

EJT12. Michael Elberfeld, Andreas Jakoby, and Till Tantau. Algorithmic meta theorems for circuit classes of constant and logarithmic depth. In *29th STACS)*, volume 14 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 66–77, 2012. See also ECCC TR 128, 2011.

GJ79. M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

GKMS01. Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 79–88, 2001.

GL79. George Gens and Eugene Levner. Computational complexity of approximation algorithms for combinatorial problems. In *Mathematical Foundations of Computer Science 1979, Proceedings, 8th Symposium, Olomouc, Czechoslovakia, September 3-7, 1979*, pages 292–300, 1979.

GL80. Gens and Levner. Complexity of approximation algorithms for combinatorial problems: A survey. *SIGACTN: SIGACT News*, 12, 1980.

HR73. Laurent Hyafil and Ronald L. Rivest. Graph partitioning and constructing optimal decision trees are polynomial complete problems. Technical Report Rapport de Recherche no. 33, IRIA – Laboratoire de Recherche en Informatique et Automatique, 1973.

IK75. Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22(4):463–468, 1975.

INP+13. Tatsuya Imai, Kotaro Nakagawa, A. Pavan, N. V. Vinodchandran, and Osamu Watanabe. An $O(n^{1/2+\epsilon})$ space and polynomial-time algorithm for directed planar reachability. In *Proceedings of IEEE Conference on Computational Complexity*, pages 277–286. IEEE, 2013.

Kan10. Daniel M. Kane. Unary subset-sum is in logspace. *CoRR (arxiv)*, abs/1012.1336, 2010.

Kar72. Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, 1972.

KMPS03. Hans Kellerer, Renata Mansini, Ulrich Pferschy, and Maria Grazia Speranza. An efficient fully polynomial approximation scheme for the subset-sum problem. *J. Comput. Syst. Sci.*, 66(2):349–370, 2003.

KNR95. Ilan Kremer, Noam Nisan, and Dana Ron. On randomized one-round communication complexity. *Computational Complexity*, 8:596–605, 1995.

KPP04. Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack problems*. Springer, 2004.

KS92. Bala Kalyanasundaram and Georg Schnitger. The probabilistic communication complexity of set intersection. *SIAM Journal on Discrete Mathematics*, 5(4):545–557, 1992.

KZ94. Dexter Kozen and Shmuel Zaks. Optimal bounds for the change-making problem. *Theor. Comput. Sci.*, 123(2):377–388, 1994.

LMS12. Nutan Limaye, Meena Mahajan, and Karteek Sreenivasaiah. The complexity of unary subset sum. In Joachim Gudmundsson, Julián Mestre, and Taso Viglas, editors, *COCOON*, volume 7434 of *Lecture Notes in Computer Science*, pages 458–469. Springer, 2012.

LN10. Daniel Lokshtanov and Jesper Nederlof. Saving space by algebraization. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 321–330, 2010.

LT79. Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM J. Appl. Math.*, 36:177–189, 1979.

LT80. Richard J. Lipton and Robert Endre Tarjan. Applications of a planar separator theorem. *SIAM J. Comput.*, 9(3):615–627, 1980.

Lue75. George Lueker. Two np-complete problems in nonnegative integer programming. Technical Report 178, Computer Science Laboratory, Princeton University, 1975.

Mil86. Gary L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *J. Comput. Syst. Sci.*, 32(3):265–279, 1986.

MS80. Burkhard Monien and I.H. Sudborough. The interface between language theory and complexity theory. In R.V. Book, editor, *Formal Language Theory*. Academic Press, 1980.

MT90. Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley, 1990.

Mut05. S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

Nis94. Noam Nisan. The communication complexity of threshold gates. In *In Proceedings of Combinatorics, Paul Erdos is Eighty*, pages 301–315, 1994.

Raz92. Alexander A. Razborov. On the distributional complexity of disjointness. *Theoretical Computer Science*, 106(2):385–390, 1992.

Rei08. Omer Reingold. Undirected connectivity in log-space. *Journal of the ACM*, 55(4), 2008.

Sav70. W. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177 – 192, 1970.

Ser04. Rocco A. Servedio. Monotone boolean formulas can approximate monotone linear threshold functions. *Discrete Applied Mathematics*, 142(1-3):181–187, 2004.

Ull84. Jeffrey D. Ullman. *Computational Aspects of VLSI*. Computer Science Press, Maryland, 1984.

Val84. L.G Valiant. Short monotone formulae for the majority function. *Journal of Algorithms*, 5(3):363 – 366, 1984.

Vaz01. Vijay V. Vazirani. *Approximation algorithms*. Springer, 2001.

WY92. Gerhard J. Woeginger and Zhongliang Yu. On the equal-subset-sum problem. *Inf. Process. Lett.*, 42(6):299–302, 1992.

Yao89. A. C. Yao. Circuits and local computation. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, STOC, pages 186–196, NY, USA, 1989. ACM.