Solving Medium-Density Subset Sum Problems in Expected Polynomial Time*

Abraham D. Flaxman[1] and Bartosz Przydatek[2]

[1] Department of Mathematical Sciences, Carnegie Mellon University
Pittsburgh, PA 15213, USA
abie@cmu.edu

[2] Department of Computer Science, ETH Zurich
8092 Zurich, Switzerland
przydatek@inf.ethz.ch

Abstract. The subset sum problem (SSP) (given n numbers and a target bound B, find a subset of the numbers summing to B), is a classic NP-hard problem. The hardness of SSP varies greatly with the density of the problem. In particular, when m, the logarithm of the largest input number, is at least $c \cdot n$ for some constant c, the problem can be solved by a reduction to finding a short vector in a lattice. On the other hand, when $m = \mathcal{O}(\log n)$ the problem can be solved in polynomial time using dynamic programming or some other algorithms especially designed for dense instances. However, as far as we are aware, all known algorithms for dense SSP take at least $\Omega(2^m)$ time, and no polynomial time algorithm is known which solves SSP when $m = \omega(\log n)$ (and $m = o(n)$).
We present an expected polynomial time algorithm for solving uniformly random instances of the subset sum problem over the domain $\mathbb{Z}_M$, with $m = \mathcal{O}((\log n)^2)$. To the best of our knowledge, this is the first algorithm working efficiently beyond the magnitude bound of $\mathcal{O}(\log n)$, thus narrowing the interval of hard-to-solve SSP instances.

1 Introduction

The subset sum problem (SSP), one of the classical NP-hard problems, is defined as follows: given n numbers and a target bound B, find a subset of the numbers whose sum equals B.

In this paper, we consider a case arising commonly in cryptographic applications where the numbers are represented by m-bit integers, and the sums are computed modulo M, where M is another m-bit integer. In other words, the addition is performed in $\mathbb{Z}_M$. More formally, the subset sum problem of dimensions n and m is:

Given: n numbers $a_1, \ldots, a_n$, with $a_i \in \mathbb{Z}_M$, and a target $B \in \mathbb{Z}_M$, where M is an m-bit integer

* Appeared in Proc. STACS 2005, LNCS 3404, pp. 305–314.
This version corrects some typos in Sect. 2.3.

FIND: a subset $S \subset \{1,\dots,n\}$, such that

$$\sum_{i \in S} a_i \equiv B \mod M \ .$$

We focus on *random* instances of the problem, where both the input numbers and the bound are picked uniformly at random. Similar random instances (with different parameters than we will eventually select) were shown by Chvatal [Chv80] to be hard instances for a class of knapsack algorithms.

The hardness of random SSP instances varies significantly with the choice of parameters, in particular the magnitude of m as a function of n (cf. [IN96]):

$m > n$: such instances are "almost 1-1" (each subset has a different sum), and are efficiently solvable by a reduction to a short vector in a lattice when $m \geq c \cdot n$, for some constant c [LO85,Fri86,CJL+92].

$m < n$: such instances are "almost onto" (with multiple solutions for most targets), and are efficiently solvable by various techniques in high-density case, i.e., for $m = \mathcal{O}(\log n)$ (e.g., by dynamic programming, or using methods of analytical number theory [CFG89,GM91]).

Despite various efficient approaches to dense instances, as far as we are aware, all these algorithms take at least $\Omega(M)$ time, and so none of them works in polynomial time when $m = \omega(\log n)$.

1.1 Contributions

In this work we propose an expected polynomial time algorithm which solves uniformly random SSP instances with m up to $(\log n)^2/16$. Our algorithm starts by dividing the input instance into small, efficiently solvable subinstances. The solutions of subinstances lead to a reduced instance (simpler than the original input), which we solve recursively. Finally, the solution of the reduced instance is combined with the solutions of subinstances to yield a solution of the original instance.

To the best of our knowledge, this is the first algorithm working efficiently beyond the magnitude bound of $\mathcal{O}(\log n)$, thus narrowing the interval with hard-to-solve SSP instances.

1.2 Related Work

Our algorithm bears some similarity to an approach developed by Blum *et al.* [BKW03] in the context of computational learning theory. By employing a recursive approach much like ours, they provide an algorithm for learning an XOR function in the presence of noise.

Beier and Vöcking [BV03] presented an expected polynomial time algorithm for solving random knapsack instances. Knapsack and subset sum have some compelling similarities, but the random instances considered there are quite different from ours, and this leads to the development of quite a different approach.

1.3 Notation and Conventions

A tuple $(a_1, \ldots, a_n; B, M)$ denotes an instance of SSP with input numbers a_i and target B to be solved over $\mathbb{Z}_M$.

For the clarity of presentation we occasionally neglect the discrete nature of some terms in summations to avoid the use of rounding operations (floors and ceilings). However, this simplification does not compromise the validity of our results. All asymptotic notation is with respect to n, we write $f(n) \sim g(n)$ to mean $f(n)/g(n) \to 1$ as $n \to \infty$. All logarithms are base 2.

2 The New Algorithm

We begin with a special case, an algorithm applicable when M is a power of 2. Then we present another special case, an algorithm applicable when M is odd. In general, we apply a combination of the two special cases. Given any modulus M we write $M = \bar{M} \cdot M'$, with $\bar{M} = 2^{\bar{m}}$ and M' odd. We use the first algorithm to reduce the original problem $(a_1, \ldots, a_n; B, M)$ to a problem $(a'_1, \ldots, a'_{n'}; B', M')$, and then use the second algorithm to solve the reduced problem.

In the algorithms below ℓ is a parameter whose value will later be set to $(\log n)/2$. For simplicity, the description presented below focuses on the core part of the algorithms, which can fail on some inputs. Later we show that the failures have sufficiently low probability so that upon failure we can run a dynamic programming algorithm (which takes exponential time) and obtain an *expected* polynomial time algorithm.

2.1 Subset Sum Modulo Power of 2

Given an instance $(a_1, \ldots, a_n; B, M)$, with $M = 2^m$ and $B \neq 0$, we transform it to an equivalent instance with target zero, i.e., $(a_1, \ldots, a_n, a_{n+1}; 0, M)$, where $a_{n+1} = M - B$ and we require that a valid solution contain this additional element a_{n+1}. To solve the target-zero instance we proceed as follows: we find among the input numbers a maximum matching containing a_{n+1}, where two numbers a_i, a_j can be matched if the sum $(a_i + a_j)$ has its ℓ least significant bits equal to zero, (in other words, if $(a_i + a_j) \equiv 0 \mod 2^\ell$.) From the matching we generate a "smaller" instance of SSP, which we solve recursively: given a matching of size s, $((a_{i_1}, a_{j_1}), \ldots, (a_{i_s}, a_{j_s}))$, where wlog. $a_{i_s} = a_{n+1}$, we generate an instance $((a_{i_1}+a_{j_1})/2^\ell, \ldots, (a_{i_s}+a_{j_s})/2^\ell; 0, 2^{m-\ell})$, and we require that a valid solution of this instance must contain the last element. Note that the instance to be solved recursively is indeed smaller. It has at most $(n+1)/2$ input numbers, and both the modulus and the input numbers are shorter by ℓ bits. When the recursion reaches the bottom, we extract a solution of the original problem in a straightforward way. Figure 1 presents the algorithm in pseudocode. Note that the algorithm returns a set $\mathcal{S}$ of disjoint subsets, where the the last subset is a solution to the input problem, and all remaining subsets sum up to zero modulo 2^m. These extra subsets are used in the combined algorithm in Sect. 2.3.

```
procedure SSPmod2(a_1, ..., a_n, B, m, ℓ)
  a_{n+1} := −B
  S := SSPmod2rec(a_1, ..., a_{n+1}, m, ℓ)
  /** wlog. S = (S_1, ..., S_s) and (n+1) ∈ S_s **/
  return (S_1, ..., S_{s−1}, S_s \ {n+1})

procedure SSPmod2rec(a_1, ..., a_{n+1}; m, ℓ)
  S := ()
  V := {1, ..., n, n+1}
  E := {(i, j) : (a_i + a_j) ≡ 0 mod 2^ℓ}
  E' := maximum matching in G = (V, E) containing vertex (n+1)
  /** wlog. E' = (e_1, ..., e_s), with e_s containing (n+1) **/
  if E' is non-empty then
    if ℓ < m then
      ∀e_k ∈ E', e_k = (i_k, j_k), let a'_k := (a_{i_k} + a_{j_k})/2^ℓ
      S' := SSPmod2rec(a'_1, ..., a'_s; m − ℓ, ℓ)
      if S' is not empty then
        /** wlog. S' = (S'_1, ..., S'_t), with each S'_i ⊆ {1...s}, and s ∈ S'_t **/
        ∀S'_i ∈ S' let S_i := ⋃_{e_k : k ∈ S'_i, e_k = (i_k, j_k)} {i_k, j_k}
        S := (S_1, ... S_t)
    else
      ∀e_k ∈ E', e_k = (i_k, j_k), let S_k := {i_k, j_k}
      S := (S_1, ... S_s)
  return S
```

Fig. 1. The proposed algorithm for solving dense SSP instances modulo a power of 2

We remark that the above method can be used to solve instances of SSP with some other moduli, for example when M is a power of small primes, or when M is "smooth" (meaning the product of small primes). However, the method does not generalize easily to arbitrary moduli, and in particular gives no obvious way to handle a large prime modulus. In the next section we describe a different algorithm, which works with high probability for arbitrary *odd* moduli.

2.2 Subset Sum With An Odd Modulus

The algorithm for SSP with an odd modulus has on a high level the same strategy as the algorithm from the previous section, i.e., it successively reduces the size of the numbers by matching them in pairs. However, it differs in one significant detail. Instead of working on least significant bits, it zeros out the most significant bits at each step of the recursion.

Given an instance $(a_1, \ldots, a_n; B, M)$, with M odd and $B \neq 0$, we begin, as in the previous case, by transforming it to an equivalent instance with target 0. However, this time we use a different transformation. To each input number we add the value $\Delta := (-B/2^t) \mod M$, where $t = \lceil \log_2 M/\ell \rceil$, so the modified instance is $(a'_1, \ldots, a'_n; 0, M)$, where $a'_i = a_i + \Delta$.

Our motivation for making this transformation becomes clear when we reveal our plan to make sure that any solution returned by our algorithm contains exactly 2^t elements. Since the sum of the solution of the modified instance is zero modulo M, the sum of the corresponding numbers in the original instance is B, as each number of the solution contributes an extra Δ to the sum and

$$2^t \cdot \Delta \equiv -B \mod M.$$

The fact that M is odd is required to ensure that such a Δ exists.

Now it is convenient to view elements from $\mathbb{Z}_M$ as numbers from the interval $I = \{-(M-1)/2, \ldots, (M-1)/2\}$, following the transformation

$$a \to \begin{cases} a, & \text{if } a \le (M-1)/2; \\ a - M, & \text{otherwise.} \end{cases} \tag{1}$$

Given a target-zero instance $(a'_1, \ldots, a'_n; 0, M)$ with M odd, we find a solution of cardinality 2^t as follows: we find a maximum matching among the input

procedure $\mathsf{SSPmodOdd}(a_1, \ldots, a_n; B, M, \ell)$
  $t := \lceil \log_2 M/\ell \rceil$
  $\Delta := (-B/2^t) \mod M$
  return $\mathsf{SSPmodOddRec}(a_1 + \Delta, \ldots, a_n + \Delta; M, \ell, 1)$

procedure $\mathsf{SSPmodOddRec}(a_1, \ldots, a_n; M, \ell, d)$
  /** *we view a_i's as numbers from $I = \{-(M-1)/2, \ldots, (M-1)/2\}$* **/
  $S := \{\}$
  $V := \{1, \ldots, n\}$
  $E := \{(i, j) : \exists k \in \mathbb{Z}, a_i \in [kM/2^{d\ell+1}, (k+1)M/2^{d\ell+1}],$
      $a_j \in [-(k+1)M/2^{d\ell+1}, -kM/2^{d\ell+1}]\}$
  $E' :=$ maximum matching in $G = (V, E)$
  /** *wlog.* $E' = (e_1, \ldots, e_s)$ **/
  if E' is non-empty **then**
    if $d \cdot \ell < \lceil \log_2 M \rceil$ **then**
      $\forall e_k \in E'$, $e_k = (i_k, j_k)$, **let** $a'_k := (a_{i_k} + a_{j_k})$
      $S' := \mathsf{SSPmodOddRec}(a'_1, \ldots, a'_s; M, \ell, d+1)$
      if S' is not empty **then**
        /** $S' \subseteq \{1 \ldots s\}$ **/
        $S := \bigcup_{e_k : k \in S', e_k = (i_k, j_k)} \{i_k, j_k\}$
    else
      $S := \{i_1, j_1\}$, where $e_1 \in E'$, $e_1 = (i_1, j_1)$.
  return S

Fig. 2. The proposed algorithm for solving dense SSP instances with an odd modulus

numbers, where two numbers a'_i, a'_j can be matched iff there exists an integer k so that when viewed as elements of the interval I, as in (1), $a'_i \in [kM/2^{\ell+1}, (k+1)M/2^{\ell+1}]$ and $a'_j \in [-(k+1)M/2^{\ell+1}, -kM/2^{\ell+1}]$. Again, from the matching we generate a "smaller" instance of SSP, which we solve recursively. Given a matching of size s,

$$((a'_{i_1}, a'_{j_1}), \ldots, (a'_{i_s}, a'_{j_s})),$$

we generate an instance $((a'_{i_1} + a'_{j_1}), \ldots, (a'_{i_s} + a'_{j_s}); 0, M)$. By the property of the matched numbers, the input numbers of the new instance are smaller in the sense that they are closer to 0 when viewed as elements of interval I. Figure 2 presents in pseudocode the algorithm for odd moduli.

2.3 Combined Algorithm

As mentioned above, given an instance $(a_1, \ldots, a_n; B, M)$, for any (m-bit) modulus M, we write $M = \bar{M} \cdot M'$, with $\bar{M} = 2^{\bar{m}}$ and M' odd, and apply both algorithms described above, one for $\bar{M}$ and one for M'.

First, we solve the instance $(a_1, \ldots, a_n; B, \bar{M})$ using procedure SSPmod2. As a solution, we obtain a sequence $\mathcal{S} = (S_1, \ldots, S_s)$ of disjoint subsets of $\{1, \ldots, n\}$, where for each $i = 1..(s-1)$ we have $\sum_{j \in S_i} a_j \equiv 0 \mod \bar{M}$, and $\sum_{j \in S_s} a_j \equiv B \mod \bar{M}$(i.e., the last subset is a solution for target B). From this solution we generate an instance for the second algorithm, $(a'_1, \ldots, a'_{n'}; B', M')$, where $n' = s-1$, $a'_i = (\sum_{j \in S_i} a_j) \mod M'$ for $i = 1..n'$, and $B' = B - \sum_{j \in S_s} a_j$. The second algorithm returns a solution $S' \subseteq \{1, \ldots, n'\}$, from which we derive our answer

$$S_s \cup \left(\bigcup_{j \in S'} S_j \right)$$

Figure 3 presents the combined algorithm in pseudocode.

3 Analysis

3.1 Correctness

We need to argue that any non-empty subset returned by the algorithm is a valid solution.

First consider computation modulo $\bar{M}$ which is a power of 2. At each level of recursion we match pairs of input numbers so that ℓ least significant bits are zeroed, while respecting the constraint, that the last input number is matched. Therefore, in recursive call, we have zeroed the least significant bits of the resulting numbers, so it follows by induction that all the subsets returned by SSPmod2rec sum up to $0 \mod \bar{M}$.

Moreover, we need to argue that the last subset returned by SSPmod2rec determines a solution for the given target B. Indeed, if S_s is the last subset returned by SSPmod2rec, then $(n+1) \in S_s$ and $\sum_{i \in S_s} a_i \equiv 0 \mod \bar{M}$. Since $a_{n+1} = -B$, this implies that $\sum_{i \in S \setminus \{n+1\}} a_i \equiv B \mod \bar{M}$, as desired.

procedure $\mathsf{DenseSSP}(a_1, \ldots, a_n; B, M, \ell)$
$S := \{\}$
find M' and $\bar{M} = 2^{\bar{m}}$ such that M' is odd and $M = 2^{\bar{m}} \cdot M'$
$\mathcal{S} := \mathsf{SSPmod2}(a_1, \ldots, a_n, B, \bar{m}, \ell)$ /** *here* $\mathcal{S} = (S_1, \ldots, S_s)$*, with* **/
/** $\sum_{j \in S_i} a_j \equiv 0 \mod \bar{M}$ *for* $i = 1..(s-1)$*, and* **/
/** $\sum_{j \in S_s} a_j \equiv B \mod \bar{M}$ **/
if $\mathcal{S}$ is not empty **then**
$\forall i = 1..(s-1)$ **let** $a'_i := (\sum_{j \in S_i} a_j) \mod M'$
$B' := B - \sum_{j \in S_s} a_j$
$S' := \mathsf{SSPmodOdd}(a'_1, \ldots, a'_{s-1}; B', M', \ell)$ /** *here* $S' \subseteq \{1, \ldots, n'\}$ **/
if S' is not empty **then**
$S := S_s \cup \left(\bigcup_{j \in S'} S_j\right)$
return S

Fig. 3. The combined algorithm for solving dense SSP instances

To prove the correctness of the computation modulo an odd number M', note that the transformation to a target-zero instance gives the desired result: any solution is created bottom-up, by first matching two input numbers, than matching two pairs matched previously, and so on, i.e., at each recursion level the number of the numbers in a solution is doubled, so the size of the solution subset is equal 2^t, where t is the depth of recursion, which, in turn, equals $\lceil \log_2 M'/\ell \rceil$. Therefore, any solution with target zero will have exactly $2^t \cdot \Delta \equiv -B \mod M'$ of "extra" sum, i.e., the corresponding original numbers sum up to $B \mod M'$.

Further, since the algorithm matches the numbers which have opposite signs but "close" magnitudes, at each level of recursion a portion of ℓ *most* significant bits is zeroed, while avoiding the problems of overflows and wrap-arounds when adding the numbers. Hence, by induction, the subset returned by $\mathsf{SSPmodOddRec}$ sums up to $0 \mod M'$

The correctness of the combined argument follows from the above arguments and from the Chinese Remainder Theorem, since $M = \bar{M} \cdot M'$, where $\bar{M}$ and M' are relatively prime.

3.2 Success Probability

We consider the cases with magnitudes m up to $(\log n)^2/16$ and we set $\ell = (\log n)/2$. At a given level in the recursion, in both cases (power of 2 and odd), the success of the algorithm at that level depends on the number of numbers in the recursion being "enough". And, the number of numbers in the recursion at a given level is equal to the number of edges in the matching at the previous level. We will argue by induction. Let t_k denote the number of numbers available at the beginning of level k of the recursion, and let s_k denote the number of edges in the matching at level k.

Lemma 1. *For s_k and t_k defined as above, Let $\mathcal{A}_k$ denote the event that $s_k \geq t_k/4$. Then*

$$\Pr[\mathcal{A}_k \mid \mathcal{A}_1, \ldots, \mathcal{A}_{k-1}] \leq \exp\left(-n^{3/4}/32\right).$$

Proof If $\mathcal{A}_1, \ldots, \mathcal{A}_{k-1}$ occur (meaning, in every previous level of the recursion, we have managed to keep at least $1/4$ of the numbers), then we begin level k with at least $n(1/4)^{(\log n)/8} = n^{3/4}$ numbers (since there are at most $m/\ell \leq (\log n)/8$ levels of recursion total).

Lemma 1 is easier to argue when the modulus is a power of 2. Then the subinstances are formed by zeroing least significant bits, and so the reduced numbers are independent and uniformly random. When the modulus is an odd, the reduced numbers are independent but not uniformly random. Fortunately, they are distributed symmetrically, in the sense that $\Pr[a'_i = a] = \Pr[a'_i = -a]$. We argue this by induction: Suppose $\Pr[a_i = a] = \Pr[a_i = -a]$ for all i. Then, since each edge $(i, j) \in E$ yields an $a'_k = a_i + a_j$, we have

$$\begin{aligned} \Pr[a'_k = a] &= \sum_b \Pr[a_i = b] \Pr[a_j = a - b] \\ &= \sum_b \Pr[a_i = -b] \Pr[a_j = -(a - b)] \\ &= \Pr[a'_k = -a]. \end{aligned}$$

This symmetry property is all that we need to show s_k is very likely to exceed $t_k/4$. We can pretend the t_k input numbers are generated by a two-step process: first, we pick the absolute value of the numbers constituting the instance, and then we pick the sign of each number. Since the distribution is symmetric, in the second step each number in the instance becomes negative with probability 1/2.

Let T_i denote the number of numbers picked in the first step with absolute value in the interval $[(i-1)M/L^d, iM/L^d]$, where $L = 2^\ell$ and $i = 1 \ldots L$. Then the number of *negative* numbers in interval i is a random variable $X_i \sim \mathrm{Bi}(T_i, 1/2)$, and we can match all but $Y_i := |X_i - (T_i - X_i)|$ numbers in interval i. Further,

$$\mathrm{E}[Y_i] = \sum_{k=1}^{T_i} \Pr[Y_i \geq k] = \sum_{k=1}^{T_i} \Pr[|X_i - T_i/2| \geq k/2],$$

and by Azuma's inequality, this is at most

$$\sum_{k=1}^{T_i} 2e^{-k^2/(2T_i)} \leq \int_{x=0}^{\infty} 2e^{-x^2} \sqrt{2T_i}dx = \sqrt{2\pi T_i}.$$

Let Y denote the total discrepancy of all the bins,

$$Y = \sum_{i=1}^{L} Y_i.$$

By linearity of expectation, we have that we expect to match all but

$$\mathrm{E}[Y] = \mathcal{O}(\sqrt{T_1} + \cdots + \sqrt{T_L})$$

numbers. This sum is maximized when $T_1 = \cdots = T_L$, and minimized when $T_i = t$ for some i (and all other T_j's are zero), hence

$$\mathcal{O}(\sqrt{t}) \leq \mathrm{E}[Y] \leq \mathcal{O}(\sqrt{tL}) \ . \tag{2}$$

Changing a single number in the instance can change the discrepancy by at most 2, so we use Azuma's inequality in a convenient formulation given by McDiarmiad [McD89] (see also Bollobás [Bol88]) and the fact that $L = 2^\ell = \sqrt{n}$ and $t \geq n^{3/4}$.

$$\begin{aligned}
\Pr[s \leq t/4] &= \Pr[Y \geq t/2] \\
&\leq \Pr[Y \geq \mathrm{E}[Y] + t/4] \\
&\leq e^{-t/32} \\
&\leq \exp\left(-n^{3/4}/32\right) .
\end{aligned}$$

□

Then, in the case of an odd modulus, the failure probability is bounded by

$$\Pr[\text{failure}] \leq \sum_{k=1}^{m/\ell} \Pr[\mathcal{A}_k \mid \mathcal{A}_1, \ldots, \mathcal{A}_k] \leq (\log n) \exp -n^{3/4}/32 = \mathcal{O}(e^{-\sqrt{n}}).$$

If the modulus is a power of 2, we must also account for the possibility of failure due to not matching the special number a_{n+1} at some stage. Let $\mathcal{B}_k$ denote the event that the special number is not matched at stage k. This type of failure only occurs if all $t_k - 1$ other numbers are different from the special number. Since the mod 2 reductions keep the numbers at stage k uniformly distributed among $m - k\ell$ possibilities, the probability of $\mathcal{B}_k$ given t_k is $\left(1 - \frac{1}{m-k\ell}\right)^{t_k - 1}$ and if $\mathcal{A}_1, \ldots, \mathcal{A}_{k-1}$ hold, this is at most $\exp\left(-(\log n)^{-2} n^{3/4}\right)$. So again, the probability of failure is $\mathcal{O}(e^{-\sqrt{n}})$.

3.3 Running Time

The running time of the algorithm above is dominated by the time required to solve all the subinstances, which is bounded by $(n-1)/(\ell-1) \cdot \mathcal{O}(2^\ell) = \mathcal{O}(n^{3/2})$.

In the case of failure, we can solve the instance by dynamic programming in time $O(2^{(\log n)^2})$. Since (when n is sufficiently large) the failure probability is much less than $2^{-(\log n)^2}$, combining the algorithm above with a dynamic programming backup for failures yields a complete algorithm that runs in expected polynomial time.

3.4 Choice of Parameters

The parameters above are not optimized, but there is a curious feature in the proof of Lemma 1 that puts a restriction on the range of coefficients c that would work for $m = c(\log n)^2$. Similarly, the range of constants c' that would work for $\ell = c' \log n$ is restricted in a way that does not seem natural. For $\ell = (\log n)/2$ and $m = (\log n)^2/16$, the number of stages of recursion is small enough that each stage has sufficiently many numbers to succeed with high probability. But for $\ell = (\log n)/2$ and $m = (\log n)^2/8$, McDiarmid's version of Azuma's inequality will not work in the way we have used it.

4 Conclusions and Open Problems

We presented an expected polynomial time algorithm for solving uniformly random subset sum problems of medium density over $\mathbb{Z}_M$, with m bounded by $\mathcal{O}((\log n)^2)$, where n is the number of the input numbers. As far as we are aware, this is the first algorithm for dense instances that works efficiently beyond the magnitude bound of $\mathcal{O}(\log n)$, thus narrowing the interval with hard-to-solve SSP instances. A natural open question is whether the bound on the magnitude can be further extended, e.g. up to $(\log n)^z$ for some $z > 2$.

Finally, recall that DenseSSP is a deterministic algorithm which can fail with non-zero probability. Since this probability is very low, upon failure we can run a dynamic programming algorithm and still obtain expected polynomial time in total. A different way of handling failures might be to run DenseSSP again on randomly permuted input. Note however that such multiple trials are not fully independent, thus complicating the analysis. It is an interesting problem to compare this alternative approach with the one we have analyzed.

5 Acknowledgement

We would like to thank Krzysztof Pietrzak for useful discussions.

References

[BKW03] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *J. ACM*, 50(4):506–519, 2003.

[Bol88] B. Bollobás. Martingales, isoperimetric inequalities and random graphs. In A. Hajnal, L. Lovász, and V. T. Sós, editors, *Combinatorics*, number 52 in Colloq. Math. Soc. János Bolyai, pages 113–139. North Holland, 1988.

[BV03] René Beier and Berthold Vöcking. Random knapsack in expected polynomial time. In *Proc. 35th ACM STOC*, pages 232–241, 2003.

[CFG89] M. Chaimovich, G. Freiman, and Z. Galil. Solving dense subset-sum problems by using analytical number theory. *J. Complex.*, 5(3):271–282, 1989.

[Chv80] V. Chvatal. Hard knapsack problems. *Operations Research*, 28:1402–1411, 1980.

[CJL+92] Matthijs J. Coster, Antoine Joux, Brian A. LaMacchia, Andrew M. Odlyzko, Claus-Peter Schnorr, and Jacques Stern. Improved low-density subset sum algorithms. *Comput. Complex.*, 2(2):111–128, 1992.

[Fri86] Alan Frieze. On the Lagarias-Odlyzko algorithm for the subset sum problem. *SIAM J. Comput.*, 15(2):536–539, 1986.

[GM91] Zvi Galil and Oded Margalit. An almost linear-time algorithm for the dense subset-sum problem. *SIAM J. Comput.*, 20(6):1157–1189, 1991.

[IN96] Russell Impagliazzo and Moni Naor. Efficient cryptographic schemes provably as secure as subset sum. *Journal of Cryptology*, 9(4):199–216, Fall 1996.

[LO85] J. C. Lagarias and A. M. Odlyzko. Solving low-density subset sum problems. *J. ACM*, 32(1):229–246, 1985.

[McD89] Colin McDiarmid. On the method of bounded differences. In *London Mathematical Society Lecture Note Series*, volume 141, pages 148–188. Cambridge University Press, 1989.