A medley for computational complexity: With applications of information theory, learning theory, and Ketan Mulmuley's parametric complexity technique
Loff, B.F.L.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 3

Hardness results for Knapsack problems

In this chapter we show various hardness results for knapsack and related problems; in particular we will show that unless the Exponential-Time Hypothesis is false, subset-sum cannot be approximated any better than with an FPTAS. We also provide new unconditional lower bounds for approximating knapsack in Ketan Mulmuley’s parallel PRAM model. Furthermore, we give a simple new algorithm for approximating knapsack and subset-sum, that can be adapted to work for small space, or in small parallel time.

The results in this chapter are based on the paper:


3.1 Introduction

The Knapsack problem is a natural example of an NP-complete optimization problem which nevertheless has a fully-polynomial time approximation scheme (FPTAS). However, there is no a priori reason to think that FPTAS would be the best one could hope for. In fact, in order to prove NP-hardness of knapsack we require the problem to be solved exactly, so in principle there could exist polynomial-time approximation algorithms for knapsack with an approximation ratio strictly closer to 1 than inverse-polynomial.

We begin this work by giving evidence that inverse-polynomial is as good an approximation as we are likely to get. For example, we obtain the following:

3.1.1. Proposition. If there is a polynomial-time algorithm to approximate knapsack with inverse-super-polynomial error ratio, then we can decide the satisfiability of $n$-variable NC circuits of size $\omega(n)$ in time $2^{o(n)}$.

In a way, this is a refinement of the NP-hardness of the knapsack problem. The reduction actually offers a robust trade-off, in that algorithms with a successively better approximation ratio can be used to simulate successively larger non-deterministic NC-computations.

A reduction from 3Sat instead of NC circuit satisfiability was independently discovered by Cygan et al. [45]. Their proof is more contrived, as it is
the result of several intermediate reductions, and their result is slightly weaker, though some of the reduction gadgets they use have a similar flavor to our own. Our proof, however, is simple, short and direct.

From the proposition above, and the sparsification lemma of [57], we get, in particular, a $2^{o(n)}$-time algorithm for 3Sat from the assumption that such a good approximation can be obtained in polynomial time. This would contradict the Exponential–Time Hypothesis [55]. This is the first hardness-of-approximation result of its kind, since hardness of approximation has been shown for every other approximation ratio (cf. Section 3.4). This also follows as a consequence of [45], although in that paper the result was not clearly formulated.

The techniques in these proofs will allow us to prove a new unconditional lower bound for solving knapsack in parallel, using Mulmuley’s parametric complexity technique [77]. Our reduction techniques can be used to show that the knapsack problem has high parametric complexity, even when the bit-lengths of the inputs are small. It will follow that it is impossible to approximate knapsack within a factor of $1 + \varepsilon$ in time $o((\log \frac{1}{\varepsilon})^\frac{1}{4})$, and using $2^{o((\log \frac{1}{\varepsilon})^\frac{1}{4})}$ processors, in Mulmuley’s parallel PRAM model without bit operations.

To complement these hardness results, we provide a simple, and to our knowledge new, algorithm for approximating knapsack and subset-sum in parallel, obtained through a mix of two standard results: the meet-in-the-middle algorithm for knapsack [53], and the PSPACE completeness of alternating time [93]. We are able to approximate subset-sum up to error ratio $1 + \varepsilon$ in space $\log \frac{1}{\varepsilon} \cdot \log n$. A linear arithmetic PRAM [see 77] can compute it in time $\log n$ using $(\frac{1}{\varepsilon})^{\log n}$ processors. Alternatively, it may be implemented by an $O(\log n)$-depth $AC$ circuit of size $(\frac{1}{\varepsilon})^{\log n}$.

After some preliminaries in Section 3.2, we study in Section 3.3 both old and new approximation algorithms for knapsack and related problems. In Section 3.4 we show how to reduce satisfiability problems to subset-sum with small bit-weights, and then proceed with the lower-bounds for Mulmuley’s model in Section 3.5.

### 3.2 Preliminaries

For a given natural number $n$, $[n]$ denotes the set $\{1, \ldots, n\}$, and $S_n$ denotes the group of permutations of $n$ letters. For $\sigma_1, \sigma_2 \in S_n$, the permutation $\sigma_1 \sigma_2$ is defined by $(\sigma_1 \sigma_2)(i) = \sigma_2(\sigma_1(i))$ for all $i \in [n]$. For a given predicate $P$, we will use the notation $[P?]$ to denote $0$ if $P$ is false, and $1$ if $P$ is true.

### 3.2.1 Circuit classes and bounded non-determinism

A width-5 permutation branching program $P$ over $k$ input bits $y_1, \ldots, y_k$ is a set (of so-called instructions) $\{(j, z_j, \alpha_j, \beta_j)\}_{j=1}^S$, with $z_j \in [k]$, $\alpha_j, \beta_j \in S_5$. For a given input $\bar{y} \in \{0, 1\}^k$, the evaluation of the program $P(\bar{y}) \in S_5$ is equal to $\gamma_1 \gamma_2 \ldots \gamma_S$, where $\gamma_j$ equals $\alpha_j$ if $y_{z_j} = 0$ and equals $\beta_j$ if $y_{z_j} = 1$. We usually
write $P(\bar{y}) = 1$ when $P(\bar{y}) = e$ (the identity) and write $P(\bar{y}) = 0$ to mean $P(\bar{y}) \neq e$.

We will let $\text{NC-Sat}(S, k)$ denote the set of circuits $C$ composed of $k$ boolean input gates and $S$ fan-in-2 NAND gates, such that $C(\bar{y}) = 1$ for some choice of $\bar{y} \in \{0, 1\}^k$. We use $\exists \text{NC}_1(k)$ to denote the class of sets accepted by log-depth fan-in-2 uniformly generated circuits of polynomial size which make use of $k$ non-deterministic bits — i.e., a set $A$ is in $\exists \text{NC}_1(k)$ if there is a uniform family of polysize boolean formulae $F_n$ such that $x \in A$ iff $F_{|x|}(x, y) = 1$ for some $y \in \{0, 1\}^k$.

Finally, we let $\text{NC}_1\text{-Sat}(S, k)$ denote the set of satisfiable size-$S$ formulae with $k$ boolean variables. Notice that $\text{NC}_1\text{-Sat}(S, k)$ denotes a set and $\exists \text{NC}_1(k)$ denotes a class of sets for which $\text{NC}_1\text{-Sat}(\text{poly}(n), k)$ is a complete problem.

### 3.2.2 Mulmuley’s parallel model of computation

Mulmuley’s model is a semi-algebraic model of parallel computation. The inputs are usually thought of as having a binary part, giving the combinatorial structure (for instance, the adjacency matrix of a graph), and an integer part, giving numerical data about this structure (such as the weights of the edges of said graph). The model treats these two types of data differently with respect to pointer jumping, which makes its full description more involved than it needs to be for our purpose. In fact, the knapsack problem has no combinatorial structure, and so we can look at a simpler form of Mulmuley’s model, and leave the full details to [77].

A (numerical) problem in this setting is a family $A_n$ of subsets of $\mathbb{Z}^n$. The model of computation is the arithmetic PRAM, a device composed of a certain number of registers and processors. At the beginning of the computation, a tuple $\bar{a} = (a_1, \ldots, a_n)$ is given as input by placing each $a_i$ in register $i$. Each processor is given a numbered sequence of instructions, each of which is one of the following:

- $w = u \circ v$, where $w$ is a register and $u$ and $v$ are either registers or constants, and $\circ$ is one of $+, -, \times$;
- If register $i$ is greater than zero, then go to instruction $\ell$, or else go to instruction $\ell'$.

At every time-step, each processor executes its current instruction simultaneously; we assume that concurrent reads and writes are OK, and are handled, for instance, by ordering the processors according to some priority. All instructions have unit cost, which actually means that the model can handle very large numbers. The machine eventually halts, for instance by letting processor 1 execute a special instruction, and we say that $\bar{a}$ was accepted if register 1 holds a zero (meaning $\bar{a} \in A$), and was rejected otherwise.

It should be noted that division is not part of the basic instructions. As a result of this, it can be proven that the model cannot know the least significant bit of an $n$-bit register in $o(\sqrt{n})$ steps and using $2^{o(\sqrt{n})}$ processors; because of

\footnote{We may assume that such formula have logarithmic-depth [38, 39].}
Chapter 3. Hardness results for Knapsack problems

this, the model is sometimes called **PRAM without bit operations**. This is an algebraic model, in that the contents of the registers at any given time-step are essentially polynomials in the numerical parameters given as the input.\(^2\)

Now we may define *Ketan’s class* \(\text{KC}(P, T)\) as the class of problems \(A\) which can be decided by such a device using \(P\) processors and \(T\) time. An algebraic algorithm will have \(P\) and \(T\) depend only on \(n\), whereas a semi-algebraic algorithm will have them depend on the bit-length of the integers \(a_i\). This model is quite powerful, and in fact it is capable of implementing every parallel algorithm that we know of. Nevertheless, Mulmuley [77] shows that the decision version of maximum flow is not in \(\text{KC}(2^{\sqrt{n/a}}, \sqrt{n/a})\), for some constant \(a\), even when the bit-lengths of the flow capacities are at most \(O(n^2)\). The result extends to any numerical problem, such as the traveling salesman problem, to which max-flow reduces by a parallel algebraic reduction. But subset-sum is not known to be such a problem.\(^3\) In the paper [90], it was shown that subset-sum is not in \(\text{KC}(2^{\sqrt{n/a}}, \sqrt{n/a})\), but using a different technique than Mulmuley’s, which gives no limit on the bit-length of the inputs among which a hard instance will be found (we could say that the lower bound was proven for algebraic algorithms, and not for semi-algebraic algorithms). In Section 3.5 we prove that subset-sum is not in \(\text{KC}(2^{1.1/a n^{1/4}}, 1/a n^{1/4})\), even for bit-length of the inputs bounded by \(O(n^2)\).

### 3.2.3 Knapsack and generalizations

As part of our study we will include a variant of the knapsack problem which we call the *symmetric knapsack problem*, and which is obtained by adding additional constraints to the knapsack instance, in the form of equations over the symmetric group. It will be seen that the symmetric knapsack problem is significantly harder to solve than the knapsack problem. Our reason to study this problem here is two-fold. First of all, not many hardness results are known for problems in non-commutative algebra [cf. 81], and symmetric subset-sum is a natural problem in that setting (see Observation 3.2.4 below). Secondly, our results show that the addition of even the simplest combinatorial structure to the knapsack problem already makes it significantly harder.

In the definition below, it helps to think of \(\{1, \ldots, n\}\) as a set of items, the \(v(i)\) as values, \(w(i)\) as weights, and \(\sigma(i)\) as patterns of these items, and that our task is to fit a set of items of maximum total value into a knapsack that

---

\(^2\)More precisely, the contents of a register at any given time-step is a polynomial in these parameters, but precisely which polynomial depends on the outcomes of previous conditionals (greater-than-zero instructions).

\(^3\)While the max-flow problem, like any problem in P, reduces to the knapsack problem, it does not seem to be possible to have such a reduction in a way that preserves the bit-length of the numerical parameters (the flow capacities). So the lower bound for max-flow does not imply a lower bound for the knapsack problem.

Intuitively, there is good reason to believe that such a reduction does not exist. The subset-sum problem is a “purely numerical” problem, in the sense that it has no combinatorial structure whatsoever; hence the combinatorial structure of a max-flow problem (the graph itself) will need to be somehow encoded in the numbers of the subset-sum problem we would wish to reduce to; but this graph can be much larger than the bit-lengths of the edge weights.
3.2. Preliminaries

3.2.1. Definition. The 0–1 symmetric knapsack problem, which we denote with \( \text{SymK}(n,a,b) \) for given integer parameters \( n,a,b \geq 0 \), is defined as the following optimization problem: We are given \( v : [n] \rightarrow [2^a] \), \( w : [n] \rightarrow [2^b] \), \( \sigma : [n] \rightarrow S_5 \), as well as \( W \in [2^b], \Sigma \in S_5 \), and then, going over all subsets \( I \subseteq [n] \), we wish to maximize

\[
\sum_{i \in I} v(i)
\]

subject to

\[
\sum_{i \in I} w(i) \leq W
\]

\[
\bigodot_{i \in I} \sigma(i) = \Sigma
\]

The notation \( \bigodot_{i \in I} \sigma(i) \) means the product of the elements \( \sigma(i) \) for \( i \in I \), in ascending order of \( i \), and for completion we let \( \bigodot_{i \in \emptyset} \sigma(i) \) denote \( e \), the identity permutation. Note that the order is important since \( S_5 \) is not commutative.

Throughout, we will assume that every \( w(i) \leq W \), since of course larger items do not fit into the knapsack, and can be ignored.

The symmetric subset-sum problem, \( \text{SYMSS}(n,b) \), is the problem of deciding, when given an instance \( X = (v,w,\sigma,W,\Sigma) \) of \( \text{SymK}(n,b,b) \) with \( v = w \), if there exists a feasible solution matching the bound \( W \):

3.2.2. Definition. The symmetric subset-sum problem \( \text{SYMSS}(n,b) \) is defined as the following decision problem: We are given \( w : [n] \rightarrow [2^b] \), \( \sigma : [n] \rightarrow S_5 \), \( W \in [2^b] \), and \( \Sigma \in S_5 \), and we wish to decide if there exists an \( I \subseteq [n] \) such that \( \sum_{i \in I} w(i) = W \) and \( \bigodot_{i \in I} \sigma(i) = \Sigma \).

Again it will be convenient to think of \([n]\) as a set of items, \( w(i) \) as weights, and \( \sigma(i) \) as patterns. Intuitively, our goal is now to find a choice of items matching a specific weight with a specific pattern. Note that subset-sum is defined as a decision problem whereas knapsack is defined as an optimization problem. If we restrict \( \sigma_1 \) to be the identity of \( S_5 \), we have the original knapsack and subset-sum problems:

3.2.3. Definition. The 0–1 knapsack problem \( K(n,a,b) \) is \( \text{SymK}(n,a,b) \) restricted to the case when \( \sigma_1, \Sigma \) are all \( e \). The 0–1 subset-sum problem \( \text{SS}(n,b) \) is defined in the same way.

3.2.4. Observation. The symmetric subset-sum problem, as defined here, is a natural problem in the setting of non-commutative algebra, because it is a special case of the subset-sum problem over general groups (studied in [81]). Indeed, to an instance \((w,\sigma,W,\Sigma)\) of \( \text{SYMSS}(n,b) \) corresponds an instance of the subset-sum problem over the symmetric group, as follows. Let \( M = p_1 \ldots p_m \) be the smallest product of the first \( m \) primes such that \( M > n2^b \), and let \( \phi(w,\sigma) = (w \mod p_1, \ldots, w \mod p_m, \sigma) \) be the canonical (additive) group isomorphism from \( \mathbb{Z}_M \times S_5 \) to \( \mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_m} \times S_5 \), itself a subgroup of \( \mathbb{S}_B \), with \( B = p_1 + \ldots + p_m + 5 \).

Then there exists an \( I \subseteq [n] \) such that \( \sum_{i \in I} w(i) = W \) and \( \bigodot_{i \in I} \sigma(i) = \Sigma \), if and only if there exists an \( I \subseteq [n] \) such that \( \bigodot_{i \in I} \phi(w(i),\sigma(i)) = \phi(W,\Sigma) \), and this latter problem is an instance of subset-sum over \( \mathbb{S}_B \).

From \( m \leq \log n \) and [20, §2.7], we get \( B \leq (\log n)^2 \cdot \log(\log n) + 5 \).
3.3 Approximation algorithms: Old and new

Given an instance $X = (v, w, \sigma, W, \Sigma)$ of symmetric knapsack, it will have a unique optimum value $m^*$, corresponding to one (among possibly many) optimal solution $I^*$.

The goal of an approximation algorithm is to estimate the (unknown) value of $m^*$. The algorithm is said to achieve approximation ratio $\alpha$ if it always outputs a value $m$ such that $m^* \leq \alpha m$.

3.3.1 Equivalence of approximation and exact solution for small weights

A classical observation in the study of knapsack problems is that, modulo a multiplicative factor of $n$, solving an $n$-item knapsack approximately to ratio $1 + \varepsilon$ is equivalent to solving instances of knapsack having integer values bounded by $\frac{1}{\varepsilon}$. This holds the exact same way for the symmetric variant.

3.3.1. Theorem. Let $\varepsilon > 0$; then each problem in the following list reduces to the one below:

1. Solving $\text{SymK}(n, a, b)$ with approximation ratio $1 + \varepsilon$;
2. Solving $\text{SymK}(n, \log n + \log(1 + \frac{1}{\varepsilon}), b)$ exactly;
3. Solving $\text{SymK}(n, a, b)$ with approximation ratio $1 + \frac{\varepsilon}{n^2}$.

This theorem will allow us to prove both upper and lower bounds for approximation algorithms by ignoring the approximation ratio altogether, and working with instances having small values $v(i)$. The proof is standard [95], and we write it here for completeness.

Proof. (1 reduces to 2) Let $X = (v, w, \sigma, W, \Sigma)$ be an instance of $\text{SymK}(n, a, b)$, and let $V = \max_i v(i)$. Set $t = \log(\frac{\varepsilon}{1 + \varepsilon} \cdot \frac{V}{n})$, and make $v'(i) = [v(i)/2^t]$. Define $X'$ to be $X$ with $v$ replaced by $v'$. Our approximate solution $I$ will be an optimal solution of $X'$. Note that $X'$ is an instance of $\text{SymK}(n, a', b)$ with $a' = \log n + \log(1 + \frac{1}{\varepsilon})$, and that furthermore since $w, \sigma, W, \Sigma$ remain unchanged, then any $I$ feasible for $X'$ will be feasible for $X$ also (and vice-versa).

To conclude, we prove an upper bound on the approximation ratio. So let $m = \sum_{i \in I} v(i)$ be the value of $I$ (i.e., $m'$ is the $X$-value of the $X'$-optimal solution), and $m^*$ the value of an optimal solution $I^*$ of $X$. Since $X'$ is $X$ with the $t$ least significant bits truncated, it is easy to see that $m^* - m \leq n 2^t$. Since, furthermore, $V \leq m^*$, we conclude that

$$\frac{m^* - m}{m} \leq \frac{n 2^t}{V}$$

which implies, by simple algebraic manipulation and our choice of $t$, that

$$m^* \leq \left(\frac{V}{V - n 2^t}\right) m \leq (1 + \varepsilon) m.$$ 

Item 2 reduces to 3 via the following more general claim:
3.3. Approximation algorithms: Old and new

3.3.2. Claim. Solving SymK(n, a, b) exactly reduces to solving SymK(n, a, b) with approximation ratio $1 + \frac{1}{n^{2a}}$.

This is almost trivial to see. Note that the optimum of any instance of SymK(n, a, b) is bounded by $m^* \leq n^{2a}$. Hence any value $m$ such that $m^* - m \leq m \left(1 - \frac{1}{1 + \frac{1}{n^{2a}}} \right) < 1$, and so $m^* = m$ (as they are both integers).

3.3.3. Observation. When the inputs are given in binary, the above reductions can be computed by an NC$^0$ circuit. When the inputs are given as numerical parameters to Mulmuley's PRAM model, the reductions can be computed in time $O(\log n + \log(1 + \frac{1}{\varepsilon}))$ using $O(n/\varepsilon)$ processors.

3.3.2 Dynamic programming algorithm and a FPTAS

We now show that the original dynamic programming algorithm for classical 0–1 knapsack [26], together with its derived FPTAS, will also work for the symmetric knapsack problem. Again the proof is standard.

3.3.4. Theorem. There is a dynamic programming algorithm for finding the optimum of SymK(n, a, b), in time $O(n^{2a}B)$, where $B$ is the time required to sum and compare $(b + \log n)$-bit numbers.

Proof. Let $X = (v, w, \sigma, W, \Sigma)$ be an instance of SymK(n, a, b), and let $V = \max_i v(i)$. Throughout the algorithm, $I = I(k, \tilde{v}, \tilde{\sigma})$, for $k \in [n], \tilde{v} \in [nV], \tilde{\sigma} \in \mathbb{M}$, when defined, will denote a solution which:

1. is a subset of $[k]$, meaning $i \notin I$ for $i > k$;
2. has value $\sum_{i \in I} v(i) = \tilde{v}$,
3. pattern $\odot_{i \in I} \sigma(i) = \tilde{\sigma}$, and
4. is weight-optimal, in the sense that $\sum_{i \in I} w(i)$ is minimum among the weights of all solutions obeying (1-3).

Now $I$ is computed iteratively: for $k = 1$, we set $I(1, 0, e) = \emptyset$ and $I(1, v(1), \sigma(1)) = \{1\}$, and let $I(1, \tilde{v}, \tilde{\sigma})$ remain undefined for all other pairs of $\tilde{v}$ and $\tilde{\sigma}$ — i.e., with only one item we can either put it in $I$ with value $v(1)$ and pattern $\sigma(1)$, or not put it in $I$, which results in value 0 and pattern $e$.

Suppose we have defined $I$ up to $k - 1$. Then let $I' = I(k - 1, \tilde{v}, \tilde{\sigma})$, and $I'' = I(k - 1, \tilde{v} - v(k), \tilde{\sigma}\sigma^{-1}_k) \cup \{k\}$. Then we define $I(k, \tilde{v}, \tilde{\sigma})$ to be equal to the $I$ among $I'$ and $I''$ that has minimum weight.\footnote{And undefined if $I'$ and $I''$ are both undefined.} Again the same reasoning applies (we can either put $k$ into $I$ or not), and in this way we prove that the requirement of minimum weight is inductively maintained.

If we keep $I$ and $W$ as arrays in a random access memory, then each iteration can be computed in time $O(nVB) = O(n^{2a}B)$. 

Finally, the optimum solution for $X$ is the highest value $\tilde{v}$ for which $I = I(n, \tilde{v}, \Sigma)$ is defined, and has weight no greater than $W$. ■

As in the original 0–1 knapsack problem, this algorithm can be used to obtain an FPTAS.

3.3.5. Corollary. Solutions of $\text{SymK}(n, a, b)$ achieving approximation ratio $1 + \varepsilon$ can be obtained in time $O((1 + \frac{1}{\varepsilon})n^3B)$.

3.3.3 A new, simple algorithm for small space and small parallel time

We present a very simple algorithm, to our knowledge new, that solves subset-sum in $O(W \log n)$ time using $O(\log W \cdot \log n)$ space. Notice that if $W$ is at least $2^{\Omega(n/\log n)}$, then we can instead run the trivial $O(n)$-space algorithm that tries every possible setting of the items, for an improved $2^n$ time bound. So this algorithm is only interesting for smaller values of $W$; but because approximation algorithms are equivalent to algorithms for small weights (Theorem 3.3.1) this makes our algorithm particularly suitable for approximation.

Our algorithm can be made to work for both the subset-sum and knapsack problems. It is a mix of ideas from two classical papers from the 70s: the meet-in-the-middle algorithm [53], and the PSPACE hardness of alternating time [93]. It can appropriately be called recursive meet-in-the-middle algorithm.

3.3.6. Theorem (Recursive meet-in-the-middle). There is an algorithm for $\text{SS}(n, b)$ that works in time $O(W \log n)$ and space $O(\log W \cdot \log n)$.

3.3.7. Observation. It is interesting to compare the time and space parameters of our algorithm with those of [71]. In that paper, the authors present a clever algorithm for subset-sum based on the use of an adequate Fourier transform, which runs in time $\tilde{O}(n^2W \log W)$ and space $\tilde{O}(n^2)$. Our algorithm is slower, but when applying it to solve subset-sum approximately, our algorithm uses less space, because usually $W \ll 2^{n/\log n}$ in this scenario.

We will present the algorithm for the subset-sum problem first, and then explain how it generalizes to the symmetric version and the knapsack problem.

Proof of Theorem 3.3.6. We show a recursive procedure for the following problem: we are given an instance $X = (w, W) \in \text{SS}(n, b)$, and a value $m \in [W]$; we will output whether there exists a subset of items $I \subseteq [n]$ such that $\sum_{i \in I} w(i) = m$. The procedure works as follows:

1. (Base case) If $n = 1$, we return true iff $w(1) = m$, else we return false.
2. We go through each possible value $m' \leq m$;
3. then we consider the partial problems $X_1$ having the first half of the items, and $X_2$, having the second half;
4. we recursively call our procedure on inputs $X_1, m'$ and $X_2, m - m'$: if no solution was found for either of the sub-problems, we move on to the next choice of $m'$. 

3.4. Hardness of approximation

(5) if solutions were found, return true.

(6) If we have gone through every value, we return false.

To see that we return true in case a solution exists, note that if \( I \) is a solution of \( X \) having value \( m \), then \( I_1 = I \cap \{1, \ldots, \lfloor \frac{n}{2} \rfloor \} \) and \( I_2 = I \setminus I_1 \) will have some values, respectively \( m_1 \) and \( m_2 \). Since \( I \) has value \( m \), it must be that \( m_2 = m - m_1 \); hence the algorithm will find some solution for \( X_1 \) and \( X_2 \) when calling (4) with \( m' = m_1 \) (here we assume inductively that solutions will be found for smaller \( n \)).

Finally, an algorithm for \( SS(n, b) \) will call this procedure with \( m = W \). To bound the time and space used, we notice that the recursion has depth \( \log n \), and each call in the stack uses \( O(\log W) \) space to store \( m \) and \( m' \).

By analyzing the algorithm above, we can see that it is given by an alternation of ORs and ANDs. Let \( SS(X, m) \) mean that there exists a solution for \( X \) of value \( m \). Then it is clear that

\[
SS(X, m) \iff \bigvee_{m' \in [m]} [SS(X_1, m') \land SS(X_2, m - m')].
\]

Expanding this out in a circuit gives us \( O(\log n) \) layers. The bottom layer will simply contain equality tests. Hence we conclude that:

**3.3.8. Corollary.** A parallel version of the algorithm can be implemented with \( AC \) circuits of depth \( O(\log n) \) and size \( O(W^{\log n}b) \), or in an arithmetic PRAM with \( O(W^{\log n}) \) processors running for \( O(\log n) \) time.

**3.3.9. Observation.** For the knapsack problem \( K[n, a, b] \), where \( w \) and \( v \) may be distinct, each recursive call will look for solutions whose weight is at most \( m \), rather than exactly \( m \), and maintain in memory the maximum value among the admissible solutions found so far (rather than simply whether there exists a solution or not); i.e., the recursion is now:

\[
K(X, m) = \max\{K(X_1, m') + K(X_2, m - m') | m' \in [m]\},
\]

where \( K(X, m) \) is the maximum value attainable for the instance \( X \) with weight up to \( m \). The basic case is \( v(1) \) if \( w(1) \leq m \), and 0 otherwise. This is computable serially in \( O(W^{\log n}) \) time and \( (\log W + a) \log n \) space, or by means of an \( AC \) circuit of depth \( O(\log n) \) and size \( O(W^{\log n}(a^2W^2 + b)) \).

\[\text{To see this, notice that the maximum of } W\text{-many } a\text{-bit numbers can be computed by } AC \text{ circuits of size } O(W^2a^2), \text{ and the basic case can be computed by a circuit of size } O(b).\]

3.4 Hardness of approximation

In the last few decades, theoretical computer science has classified most known NP-hard optimization problems according to how close to the optimum solution a polynomial time algorithm is likely to get. For instance, if \( P \neq NP \), then it holds that:
• For some constants $c > c'$, SETCOVER can be approximated to error ratio $c \log n$, but not $c' \log n$ [88].

• For some constants $c > c'$, MAX2SAT can be approximated to error ratio $c$ [68], but not $c'$ [50].

• The problem of finding an optimal multi-processor scheduling with speed factors\(^6\) has a PTAS, but no FPTAS [44, 52].

It is now natural to ask: how about problems that do have a FPTAS, can we prove that this kind of approximability is optimal, under a hardness assumption of some kind? We now show that the answer is yes, and that in this context the (much stronger) exponential-time hypothesis is the adequate choice.

### 3.4.1 Hardness of approximation for knapsack

Our strategy is the usual: we reduce a hard problem to the task of approximating subset-sum with ratio better than $1 + \frac{1}{\text{poly}}$.

#### 3.4.1. Theorem. NC-Sat$(S, k) \leq_{AC_0}^{AC_0} \text{SS}(O(S + k), O(S + k))$.

**Proof.** We will show that there is a uniform $AC_0$ circuit which, given as input a size-$S$ circuit $C(\bar{y})$ over $k$ variables $\bar{y} = y_1, \ldots, y_k$, will output an instance $X = (w, W)$ of

$$\text{SS}(2k + 3S, 2k + 4S + 1),$$

that admits a subset-sum of weight $W$ if and only if $C(\bar{y}) = 1$ for some choice of $\bar{y}$.

Suppose $C$ is given as a sequence $(G_1, \ldots, G_S)$ of binary NAND gates, where each gate $G_j$ is defined by the two gates feeding into it (these could be input gates or gates $G_{j'}$ for $j' < j$). Then $X$ will have two items $Y_i^{(0)}$ and $Y_i^{(1)}$ for each $i \in [k]$, and three items $G_j^{(00)}, G_j^{(01)}, G_j^{(10)}$ for each NAND gate $j \in [S]$. We will construct our knapsack instance in such a way that any optimal solution must choose exactly one of the $Y_i^{(0)}, Y_i^{(1)}$ items for each $i \in [k]$, and exactly one of the $G_j^{ab}$ items for each $j \in [S]$. For any optimal solution, a choice of the $Y$ items will force a choice of gate items corresponding to an evaluation of the circuit.

It will help comprehension if we present the weights of the items in our instance both algebraically and in table form. For $i \in [k], j \in [S]$, let $c(i, j)$ be 2 if input $i$ is the first input of gate $j$, 1 if it is the second input, and 0 otherwise. Similarly with gates, for $j, j' \in [S]$, let $d(j, j')$ be 2 if gate $j$ is the first input of gate $j'$, 1 if it is the second input, and 0 otherwise. We will set

---

\(^6\)This is the problem of scheduling tasks to processors of different speeds, while attempting to minimize execution time. Hence each task $t$ in a set of tasks $T$ has a length $l(t)$, and each processor $p$ in a set of processors $P$ has a speed factor $s(p)$. We wish to find an assignment of the tasks $f : T \rightarrow P$ minimizing completion time: $\max_{p \in P} \sum_{t : f(t) = p} l(t)s(p)$. 

the weights to:

\[ w(Y_i^{(0)}) = 2^{2(k-i)} + 4S + 1 \]

\[ w(Y_i^{(1)}) = 2^{2(k-i)} + 4S + 1 + \sum_{j=1}^{S} c(i,j)2^{4(S-j)+1} \]

\[ w(G_j^{(00)}) = (2 + 1) \times 2^{4(S-j)+1} + \sum_{j'=j+1}^{S} d(j,j')2^{4(S-j') +1} + [j = S?] \]

\[ w(G_j^{(01)}) = (2 + 0) \times 2^{4(S-j)+1} + \sum_{j'=j+1}^{S} d(j,j')2^{4(S-j') +1} + [j = S?] \]

\[ w(G_j^{(10)}) = (0 + 1) \times 2^{4(S-j)+1} + \sum_{j'=j+1}^{S} d(j,j')2^{4(S-j') +1} + [j = S?] \]

(Recall that \([j = S?]\) denotes 1 when \(j = S\) and 0 otherwise.) We set the maximum weight to:

\[ W = \sum_{i=1}^{k} 2^{2(k-i)} + 4S + 1 + \sum_{j=1}^{S} 3 \times 2^{4(S-j)+1} + 1 \]

The DLOGTIME uniformity of the reduction follows from the simplicity of the above expressions, which make the circuit computing the bits of the instance very easy to describe. Some global lookup is required to compute the functions \(c\) and \(d\), which makes the reduction \(AC_0\), rather than simply \(NC_0\).\(^7\)

Let us see what these numbers look like when written in binary. Since each number is less than \(2^{2k+4S+1}\), we will write them in a table with \(k + S + 1\) blocks. The first \(k\) blocks have two bits each, the middle blocks have four bits each, and the last block has one bit.

<table>
<thead>
<tr>
<th>(w)</th>
<th>(1)</th>
<th>(i)</th>
<th>(k)</th>
<th>(G_1)</th>
<th>(G_j)</th>
<th>(G_{j'})</th>
<th>(G_S)</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Y_i^{(0)})</td>
<td>00</td>
<td>...</td>
<td>01</td>
<td>...</td>
<td>00</td>
<td>0000</td>
<td>...</td>
<td>0000</td>
</tr>
<tr>
<td>(Y_i^{(1)})</td>
<td>00</td>
<td>...</td>
<td>01</td>
<td>...</td>
<td>00</td>
<td>(c(i,1))</td>
<td>...</td>
<td>(c(i,j))</td>
</tr>
<tr>
<td>(G_j^{(00)})</td>
<td>00</td>
<td>...</td>
<td>00</td>
<td>...</td>
<td>00</td>
<td>0000</td>
<td>0</td>
<td>0011</td>
</tr>
<tr>
<td>(G_j^{(01)})</td>
<td>00</td>
<td>...</td>
<td>00</td>
<td>...</td>
<td>00</td>
<td>0000</td>
<td>0</td>
<td>0010</td>
</tr>
<tr>
<td>(G_j^{(10)})</td>
<td>00</td>
<td>...</td>
<td>00</td>
<td>...</td>
<td>00</td>
<td>0000</td>
<td>0</td>
<td>0001</td>
</tr>
</tbody>
</table>

| \(W\) | 01 | ... | 01 | ... | 01 | 0011 | ... | 0011 | ... | 0011 | ... | 0011 | 1 |

It is vital to notice that in each column \(G_j\) there will be exactly five non-zero entries: two correspond to the inputs, having binary form 10 and 01, and three correspond to the gate \(G_j\), and have binary form 11, 10 and 01. These are chosen so that for any setting \(ab \in \{0,1\}^2\) of the items corresponding to the two inputs, then if we wish to achieve the maximum weight \(W\), we will be forced to pick the gate item \(G_j^{(ab)}\). Formally, we wish to prove the following claim, from which the theorem follows immediately.

\(^7\)The encoding of the circuit could be changed to make this lookup easier.
3.4.2. Claim. The solutions \( I \) of \( X \) having weight exactly \( W \), are in 1–1 correspondence with the assignments of \( \bar{y} \) causing \( C(\bar{y}) = 1 \).

Given a choice of \( \bar{y} \), we can define the solution \( I(\bar{y}) \) inductively as follows:

- \( \{Y_i^{(y_i)}\} \subseteq I; \)

- For each gate \( G_j \), we let
  - \( a_j = 1 \) if input \( i \) is the first input to gate \( j \) and \( Y_i^{(1)} \in I \), or if gate \( \ell \) is the first input to gate \( j \) and \( G_{\ell}^{(a_\ell b_\ell)} \in I \); and \( a_j = 0 \) otherwise;
  - similarly for \( b_j \) and the second input to gate \( j \).

- Then \( G_j^{(a_j b_j)} \in I \) if \( a_j \) and \( b_j \) are not both 1.

This solution will have weight exactly \( W \), if \( C(\bar{y}) = 1 \), and weight \( W - 1 \), if \( C(\bar{y}) = 0 \). This can be seen easily by inspection of the table above: for instance, if \( G_j = \text{NAND}(y_i, y_i') \), \( y_i = 1 \) and \( y_i' = 0 \), then \( Y_i^{(1)} \) contributes (binary) weight 10 to column \( G_j \), \( Y_i^{(0)} \) contributes weight 0, and \( G_j^{(0)} \) contributes weight 01, for a total of 11, which is exactly \( W \) for that column. The final column will be 1 iff the output gate \( G_S \) evaluates to 1.

To complete the proof of the claim, and hence of the theorem, it suffices to show that any solution achieving weight \( W \) must be of the form \( I(\bar{y}) \) for some unique choice of \( \bar{y} \). Let \( I \) be any solution with total weight \( W \). Again by inspection of the table above, we will find that, for each \( i \), exactly one of \( Y_i^{(0)} \) or \( Y_i^{(1)} \) must be in \( I \). This follows by letting \( i \) be the hypothetical first coordinate where this fails to hold: if both \( Y_i^{(0)} \) and \( Y_i^{(1)} \) are in \( I \), then the total weight will surpass \( W \), and if both \( Y_i^{(0)} \) and \( Y_i^{(1)} \) are missed, the total weight must be less than \( W \) because of the low weight of the items which are yet to be fixed.\(^8\) So let \( Y_i^{(y_i)} \), for \( i \in [k] \), give us the set of chosen \( Y \) items. Then again in an inductive fashion we establish that for each gate \( G_j \), we must pick at most one item among \( G_j^{(a b)} \), exactly as defined in \( I(\bar{y}) \): again it holds that this is the unique choice which fits the weight \( W \) exactly right. \( \blacksquare \)

This allows us to obtain a hardness-of-approximation result for the knapsack problem:

3.4.3. Corollary. For every \( f(n) = \omega(1) \), there exists \( g(n) = \omega(n) \), such that if subset-sum can be approximated to ratio \( 1 + n^{-f(n)} \) in polynomial time, then \( \text{NC-SAT}(g(n), g(n)) \subseteq \text{DTIME}(2^{o(n)}) \) and the Exponential–Time Hypothesis is false.

Proof. If subset-sum can be approximated thus, then by Claim 3.3.2 (more precisely, by the analogous claim for subset-sum), we can solve \( \text{SS}(n, h(n) \log n) \) in time \( n^c \), for some non-decreasing super-constant \( h(n) \). By an appropriate

\(^8\)Here it is worth pointing out: this is the reason we need to “pad” each block of the weights with additional bits. This bears some resemblance to the superincreasing sequences used in the Merkle-Hellman knapsack cryptosystem.
consulted in the
choice will force the correct choice of either

\[ 3.4.4. \text{Theorem.} \text{ NC formula size.} \]

result where the bit-length of the weights only depends logarithmically on the
amount of combinatorial structure to the problem will allow for a hardness
original subset-sum problem as a special case. However, the addition of a small
SymSS
It is immediate to see that

\[ 3.4.2 \text{ Hardness of approximation for symmetric knapsack} \]

\[ 51 \]

substitution (say \( n \leftarrow 2^{n/\sqrt{h(n)}} \)), this implies we can solve SS(\( 2^n, \omega(n) \)) in time
\( 2^{|n|} \). By the previous theorem we can now conclude that NC-SAT(\( \omega(n), \omega(n) \))
can be solved in time \( 2^{|n|} \).

In particular this implies that the satisfiability of 3CNF formulas with \( \omega(n) \)-
many clauses can be decided in time \( 2^{|n|} \) But the sparsification lemma of \([57]\)
will then imply that the satisfiability of 3CNF formulas of any size can also
be decided in time \( 2^{|n|} \), i.e., that the Exponential–Time Hypothesis is false.

More precisely, the sparsification lemma states that there is a function \( f \)
such that, for any \( \varepsilon > 0 \), any 3CNF formula is equivalent to the OR of \( O(2^{|n|}) \)
3CNF formulas in which the number of clauses is \( f(\varepsilon)n \), and these formulas
can be produced in time \( poly(n)2^{|n|} \). Hence we let \( \varepsilon = \varepsilon(n) = o(1) \) decrease
slowly enough, causing \( f(\varepsilon(n)) = \omega(1) \) to be sufficiently small, so that we can
solve 3CNF formulas with \( f(\varepsilon(n))n \) clauses in time \( 2^{|n|} \). Then producing and
evaluating the big OR of 3CNFs would take time \( poly(n)2^{2|n|+\omega(n)} = 2^{|n|} \) in
total.

Notice that this result is optimal, since for any constant \( c \) it is possible to
approximate knapsack to ratio \( 1 + \frac{1}{n^c} \) in polynomial time.

\[ 3.4.2 \text{ Hardness of approximation for symmetric knapsack} \]

It is immediate to see that SymSS(\( n, n \)) is NP-hard, since it includes the
original subset-sum problem as a special case. However, the addition of a small
amount of combinatorial structure to the problem will allow for a hardness
result where the bit-length of the weights only depends logarithmically on the
formula size.

To show that any \( \mathrm{AC}_0 \) circuit of size \( S \) can be evaluated by such a width-5 branching program.

So let \( P = \{ (j, z_j, \alpha_j, \beta_j) \}_{j=1}^S \), with \( z_j \in [k] \), \( \alpha_j, \beta_j \in \mathbb{S}_5 \). We will always
use \( i \) to index the \( y \)'s, and \( j \) to index the instructions of \( P \).

In our symmetric subset-sum problem, we will have two items for each
possible \( y \), denoted by \( Y^{(1)}_i \) and \( Y^{(0)}_i \); these items represent the possible as-
signments of the respective variable. We will also have two items for each
instruction, denoted by \( A_j \) and \( B_j \), which represent the choice of either \( \alpha_j \) or
\( \beta_j \).

We will construct our knapsack instance in such a way that the optimal
solution will choose exactly one of the \( Y^{(0)}_i, Y^{(1)}_i \) items for each \( i \), and this
choice will force the correct choice of either \( A_j \) or \( B_j \), whenever \( y_i \) is the variable
consulted in the \( j \)-th instruction of \( P \) (i.e., whenever \( z_j = i \)). If we choose to
Chapter 3. Hardness results for Knapsack problems

take item $Y_i^{(0)}$, for instance, then, in order to achieve weight $W$, we will be forced to always pick every $A_j$ whenever $z_j = i$, and won’t be able to pick any of the $B_j$.

Let $N(i) \leq S$ denote the number of times that $y_i$ is consulted by $P$, i.e., the number of $j$ such that $z_j = i$, and make $\ell = \lceil \log S + 1 \rceil$, so that each $N(i) < 2^\ell - 1$.

As before, we present the weights of the items in our instance both algebraically and in table form.

We will set the weights to:

\[
\begin{align*}
w(Y_i^{(0)}) & = 2^{2(k-i)} + 2^{2k} + N(i)2^{2(k-i)\ell} \\
w(Y_i^{(1)}) & = 2^{2(k-i)} + 2^{2k} + N(i)2^{2\ell+2(k-i)\ell} \\
\text{if } z_j = i, \text{ then } w(A_j) & = 2^{\ell+2(k-i)\ell} \\
\text{and } w(B_j) & = 2^{2(k-i)\ell}
\end{align*}
\]

We set the maximum weight to:

\[
W = \sum_{i=1}^{k} 2^{2(k-i)} + 2^{2k} + N(i) \left( 2^{2(k-i)\ell} + 2^{\ell+2(k-i)\ell} \right).
\]

The DLOGTIME uniformity of the reduction follows from the simplicity of the above expressions, which make the circuit computing the bits of the instance very easy to describe. Sum is required to compute $N(i)$, which makes the reduction $AC_0$, rather than simply $NC_0$.

Let us again see what these numbers look like in table form. Since each number is less than $2^{2k} + 2^{2\ell}$, we will write them in a table with $3k$ blocks. The first $k$ blocks have two bits each, and the last $2k$ blocks have $\ell$ bits each:

<table>
<thead>
<tr>
<th>$w$</th>
<th>1</th>
<th>$i$</th>
<th>$k$</th>
<th>$A^{(1)}$</th>
<th>$B^{(1)}$</th>
<th>$A^{(i)}$</th>
<th>$B^{(i)}$</th>
<th>$A^{(k)}$</th>
<th>$B^{(k)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Y_i^{(0)}$</td>
<td>00</td>
<td>01</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$N(i)$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$Y_i^{(1)}$</td>
<td>00</td>
<td>01</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$N(i)$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$A_j \mid z_j = i$</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$B_j \mid z_j = i$</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

| $W$ | 01 | 01 | 01 | $N(1)$ | $N(1)$ | $N(i)$ | $N(i)$ | $N(k)$ | $N(k)$ |

Finally, we set $\sigma(Y_i^{(0)}) = \sigma(Y_i^{(1)}) = e$, $\sigma(A_j) = \alpha_j$ and $\sigma(B_j) = \beta_j$. Furthermore, in our sorting of the items (which is relevant for the outcome of $\odot_i \sigma(i)$), we ensure that the items $A_j, B_j$ appear in growing order of $j$.

Here, our result follows from the following claim by setting $\Sigma = e$:

3.4.5. Claim. The solutions $I$ of $X$ having weight exactly $W$ and pattern $\Sigma$, are in 1-1 correspondence with the assignments of $\bar{y}$ causing $P(\bar{y}) = \Sigma$.

To see this, notice that given a choice of $\bar{y}$, we define the solution $I(\bar{y}) = \{Y_i^{(y_i)}\} \cup \{A_j \mid z_j = i \land y_i = 0\} \cup \{B_j \mid z_j = i \land y_i = 1\}$, and this solution will have
weight exactly $W$, and pattern $P(\bar{y})$. This can be seen easily by inspection of the table above: for instance, if $y_i = 0$, the choice of $Y_i^{(0)}$ contributes weight 1 to the $i$-th column of the $i$-th block, and weight $N(i)$ to the column $B(i)$; then the choice of $N(i)$-many $A_j$ variables will contribute with a total weight of $N(i)$ to the column $A(i)$. Now the pattern of $I(\bar{y})$ will simply be $P(\bar{y})$, by construction.

To complete the proof of the claim, and hence of the theorem, it suffices to show that any solution achieving weight $W$ must be of the form $I(\bar{y})$ for some unique choice of $\bar{y}$. Let $I$ be any solution with total weight $W$. Again by inspection of the table above, we will find that, for each $i$, exactly one of $Y_i^{(0)}$ or $Y_i^{(1)}$ must be in $I$. This follows by letting $i$ be the hypothetical first coordinate where this fails to hold: if both $Y_i^{(0)}$ and $Y_i^{(1)}$ are missed, the total weight will be less than $W$, and if both $Y_i^{(0)}$ and $Y_i^{(1)}$ are in $I$, then the total weight will surpass $W$. So let $Y_i^{(y_i)}$, for $i \in [k]$, give us the set of chosen $Y$ items. In the same way as before, it is easy to see that, for each $i$, every $A_j$ with $z_j = i$ must be picked, if $y_i = 0$, or otherwise every corresponding $B_j$ must be picked. Hence $I = I(\bar{y})$, as intended.

Let us instantiate $k$, in order to see what kind of parameters show up:

3.4.6. Corollary. Symmetric knapsack cannot be approximated to ratio $1 + 2^{-(\log n)^3}$ in polynomial time, unless $\exists \text{NC}_1((\log n)^2) \subseteq \text{P}$.

3.5 Lower bounds for parallel time

We will now prove that the knapsack problem cannot be efficiently parallelized in Mulmuley’s model. The hardness of knapsack has already been proven in [90] in the fully-algebraic setting, and now we present a proof for the optimization version of subset-sum in the semi-algebraic setting. We do this using Mulmuley’s parametric complexity technique. We will stick to the least general definition that applies to the subset-sum case.

The optimization version of subset-sum is simply the knapsack problem restricted to the case where the values of the items equal their weights; i.e., we are given $(\bar{w}, W)$ and we wish to compute:

$$S(\bar{w}, W) = \max\{\bar{a} \cdot \bar{w}|\bar{a} \in \{0, 1\}^n \text{ and } \bar{a} \cdot \bar{w} \leq W\}$$

We will now refer to the problem of computing $S$ as simply the knapsack problem. A linear parametrization for the knapsack problem is a function $P_n : \mathbb{R} \rightarrow \mathbb{R}^{n+1}$ mapping some interval $[A, B]$ into instances $(\bar{w}, W)$, where $W$ is fixed and each $w_i$ is given by a linear function of a single parameter $\lambda$. $P_n$ is said to have bit-length $\beta = \beta(n)$ if all the coefficients of these linear functions are integers of bit-length $\beta$. For each $\lambda \in [A, B]$, let $m(\lambda) = S(P_n(\lambda))$. Then it is easy to see that $m(\lambda)$ is a piecewise-linear and convex function of $\lambda$. The

---

*Here we extend the knapsack problem to the reals. This can easily be done due to it being a homogeneous optimization problem, i.e. $S(\alpha \bar{w}, \alpha W) = \alpha S(\bar{w}, W)$ for any constant $\alpha$. We could avoid this entirely, except that it makes the definitions easier to visualize.*
complexity $\rho(n)$ of $P_n$ is then the number of different slopes of $m(\lambda)$ as $\lambda$ goes from $A$ to $B$.

The parametric complexity of the knapsack problem for bit-length $\beta(n)$, $\phi(n, \beta(n))$, is equal to the maximum complexity $\rho(n)$, over all parametrizations $P_n$ of bit-length $\beta(n)$. Now let us define the following decision version of the knapsack problem: $A_n = \{(\bar{w}, W, z) | S(\bar{w}, W) \leq z\}$. Then the following is proven in [77]:

3.5.1. Theorem (Theorem 3.3 of [77]). There exist large enough constants $a, b$ such that $A_n$ cannot be solved in $KC(2\sqrt{\log \phi(n, \beta(n))}/b, \sqrt{\log \phi(n, \beta(n))}/b)$; this is so even if we restrict every numeric parameter in the input to be an integer with bit-length at most $a\beta(n)$.

Below we will prove exponential lower bounds on $\phi(O(n^2), O(n^2))$, which imply that $A_n$ is not in $KC(2^{n^{1/2}}/b, n^{1/2}/b)$, even for instances where each weight has at most $an$ bits. This is a stronger setting than the results in [90], which prove that subset-sum is not in $KC(2^{\sqrt{n}}/b, \sqrt{n}/b)$, but without any restriction on the bit-length at which the hard instances will be found. Unfortunately we were unable to show a parallel algebraic reduction from $A_n$ to subset-sum, and hence our results hold only for the knapsack problem, rather than subset-sum.

3.5.1 High parametric complexity for knapsack

3.5.2. Theorem. $\phi(O(n^2), O(n^2)) \geq 2^n$. Hence $A_n$ is not in $KC(2^{n^{1/2}}/b, n^{1/2}/b)$, even for bit-lengths restricted to $O(n)$.

The proofs require a careful blending of the techniques of Theorem 3.4.1 and of the paper [77].

Proof. We will construct a linear parametrization $P(\lambda) : [0, 2^k] \rightarrow SS(O(k + K), O(k + K))$ such that the weight of each item in $P(\lambda)$ is a linear function of $\lambda \in [0, 2^k]$, and such that the graph of the optimum value $m(\lambda)$ of $P(\lambda)$ will be a piecewise linear convex graph with $2^k$-many different slopes. The hardness then follows from Theorem 3.5.1.

Let $\bar{y} = y_1, \ldots, y_{k+K}$ denote binary vectors of length $k + K = k + \frac{k(k-1)}{2}$. For $1 \leq i_1 < i_2 \leq k$, we let $(i_1, i_2)$ stand for some bijection with $k + 1, \ldots, k + K$, and we make use of the notation $y_{(i_1, i_2)}$ for convenience.

Let $C(\bar{y})$ be the formula:

$$\bigwedge_{1 \leq i_1 < i_2 \leq k} \left( y_{(i_1, i_2)} = y_{i_1} \land \neg y_{i_2} \right), \quad (3.1)$$

and let $X = X_C$ be the corresponding subset-sum instance given by Theorem 3.4.1. Since $C$ can be implemented by an NC circuit with $10K$ NAND gates, then $X$ will have $2k + 30K$ items and the weights have bit-lengths at most $2k + 40K + 1$. 
Our $P(\lambda)$ will have the same items as $X$, namely for each variable $y_i$ there are two items designated $Y_j^{(0)}$ and $Y_j^{(1)}$, and we will use the generic designation $C_j$, for $j = 1, \ldots, 30K$, to denote the remaining items. However, the weights of $P(\lambda)$ will be different: although the most significant bits of $P(\lambda)$ will be set in the same way as in $X$ (and thus do not depend on $\lambda$), the least significant bits will be set so that as $\lambda$ goes from 0 to $2^k$, the optimum value of $P(\lambda)$ will have $2^k$-many different slopes.

Again we will present $P(\lambda)$ both numerically and in table form. Let $w_X$ and $W_X$ denote the weights of $X$. We will set the weights of $P(\lambda)$ as follows:

$$
w(Y_i^{(0)}) = 2^{k(k+5)} w_X(Y_i^{(0)}) + f_i^{(0)}(\lambda)
$$

$$
w(Y_i^{(1)}) = 2^{k(k+5)} w_X(Y_i^{(1)}) + f_i^{(1)}(\lambda)
$$

$$
w(C_j) = 2^{k(k+5)} w_X(C_j)
$$

$$
W = 2^{k(k+5)} W_X + 2^{k(k+5)} - 1
$$

We define $f$ as follows:

$$
\text{for } 1 \leq i \leq k, f_i^{(0)}(\lambda) = 2^{(k-i)(k+5)}(2^{k-i+1} - \lambda)
$$

$$
\text{for } 1 \leq i_1 < i_2 \leq k, f_{(i_1,i_2)}^{(0)}(\lambda) = 0
$$

$$
\text{for } 1 \leq i_1 < i_2 \leq k, f_{(i_1,i_2)}^{(1)}(\lambda) = 2^{(k-i_2)(k+5)} 2^{k-i_1}
$$

Notice that the weights $w$ are a left shift of $w_X$ by $k(k+5)$ bits, except for the items $Y_j^{(0)}$ and $Y_j^{(1)}$, who additionally get a least-significant value parametrized by $\lambda$. See also that the values of $f$, written in binary, can be partitioned into $k$ blocks of $k+5$ bits each.

Let us illustrate this with the following table:

| $Y_i^{(0)}$ | $Y_i^{(1)}$ | $Y_{(i_1,i_2)}^{(0)} |_{i_1 < i}$ | $Y_{(i_1,i_2)}^{(1)} |_{i_1 < i}$ | $W$ |
|-------------|-------------|-------------------|-------------------|-----|
|                        | Thm. 3.4.1  | Thm. 3.4.1        | Thm. 3.4.1        | Thm. 3.4.1 |
| $(2k + 40K + 1 \text{ bits})$ | $1 \ldots i \ldots k$ | $0 \ldots 2^{k-i+1} - \lambda \ldots 0$ | $0 \ldots 2^{k-i} \ldots 0$ | $2^{k+5} - 1 \ldots 2^{k+5} - 1 \ldots 2^{k+5} - 1$ |

The similarity with Theorem 3.4.1 ensures that we must pick exactly one of $Y_i^{(0)}$ and $Y_i^{(1)}$, and that $C(\bar{y}) = 1$ for the vector $\bar{y}$ corresponding to this choice, i.e.:

\footnote{We will also use the notation $Y_{(i_1,i_2)}^{(0)}, Y_{(i_1,i_2)}^{(1)}$ to denote the items corresponding to the variable $y_{(i_1,i_2)}$.}

\footnote{By this we mean that each $C_j$, for $j = 1, \ldots, 30K$, is one of the three items $G_j^{(00)}, G_j^{(01)}, G_j^{(10)}$ for some $j'$, that appear in the proof of Theorem 3.4.1. We group them together because we will treat them in the exact same way.}
3.5.3. Claim. Each optimal solution of $P(\lambda)$ corresponds to a unique choice of $\hat{y}$ obeying equation (3.1).

The proof of this claim is exactly as before. The correspondence is no longer bijective, since there might be values of $\hat{y}$ obeying 3.1 which have suboptimal value due to $f$. However, any solution which is optimal up to the first $2k + 40K + 1$ bits must already satisfy the formula $C$, and in this case these bits must be set to $W_X$ as in the proof of Theorem 3.4.1. We take this one step further, by noticing that (3.1) fully determines $\hat{y}$ from the first $k$ bits $y_1, \ldots, y_k$.

So let $\hat{y}$ denote $y_1, \ldots, y_k$, and $I(\hat{y})$ be the solution containing the items $Y_i^{(y_i)}$, $Y_{(i_1, i_2)}^{(y_1 \land \neg y_2)}$, and the corresponding $C_j$'s that simulate the computation of the circuit. Notice that $I(\hat{y})$ maximizes the first $2k + 10K + 1$ bits of the value to be exactly $W_X$: so we define $g_\lambda : \{0, 1\}^k \rightarrow [2^{k(k+5)} - 1]$, to be the optimum of $I(\hat{y})$ minus $2^{k(k+5)}W_X$. Now the following claim holds:

3.5.4. Claim. $\hat{y} \mapsto I(\hat{y})$ is a bijective correspondence between $\{0, 1\}^k$ and solutions of $P(\lambda)$ having value at least $2^{k(k+5)}W_X$. Furthermore, the optimum solutions of $P(\lambda)$ are those $I(\hat{y})$ which maximize $g_\lambda(\hat{y})$.

From this we may define $m^*(\lambda) = \max_{\hat{y} \in \{0, 1\}^k} g_\lambda(\hat{y})$, and this will be the value of the optimum solution of $P(\lambda)$, minus $2^{k(k+5)}W_X$. The instance $P(\lambda)$ was constructed so that the optimum is given exactly by $\lambda$.

3.5.5. Claim. For a given $\lambda \in [0, 2^k)$ of the form $\lambda = [\lambda] + \Delta$, $\Delta \in (\frac{1}{4}, \frac{3}{4})$, the maximum value of $g_\lambda(\hat{y})$ is attained when $\hat{y}$ is the binary expansion of $[\lambda]$.

We now prove this claim. Fix some choice $\hat{y} = y_1, \ldots, y_k$. Then, by summing over the columns of the above table, and by equation (3.1), one sees that the following formula holds:

$$g_\lambda(\hat{y}) = \sum_{i=1}^{k} (y_i 2^{-i} + (1 - y_i) \left(2^{-i+1} - \lambda \sum_{i_1 < i} y_{i_1} 2^{-i_1}\right)) 2^{(k-i)(k+5)}$$

(3.2)

Take the binary expansion of $[\lambda] = \lambda_1 2^{-1} + \ldots + \lambda_k 2^0$. We prove that the maximum of $g_\lambda$ is achieved when $\hat{y} = \lambda_1 \ldots \lambda_k$. For any $\hat{y} \neq \lambda_1 \ldots \lambda_k$, let $d$ be the first bit for which $\lambda_d \neq y_d$. Then we let $\tilde{y} = \lambda_1 \ldots \lambda_d y_{d+1} \ldots y_k$, and show that $g_\lambda(\tilde{y}) \geq g_\lambda(\hat{y})$.

Let $g^{(i)}_\lambda(\hat{y}) 2^{(k-i)(k+5)}$ be the $i$-th term in the summand of equation (3.2), and $G_i = g^{(i)}_\lambda(\hat{y}) - g^{(i)}_\lambda(\tilde{y})$. Then $G_i = 0$ for $i < d$, and so

$$g_\lambda(\hat{y}) - g_\lambda(\tilde{y}) = G_d 2^{(k-d)(k+5)} + \sum_{i > d} G_i 2^{(k-i)(k+5)}.$$
We may lower bound \( G_d \) as follows:

\[
G_d = (\lambda_d - y_d)2^{k-d} + (y_d - \lambda_d) \left( 2^{k-d+1} - \lambda + \sum_{i_1 < d} \lambda_{i_1} 2^{k-i_1} \right)
\]

\[
= (\lambda_d - y_d) \left( \lambda_d 2^{k-d} - 2^{k-d} + \Delta + \sum_{i > d} \lambda_i 2^{k-i} \right)
\]

\[
\geq \min(\Delta, 1 - \Delta) \geq \frac{1}{4}
\]

Since every \( |G_i| \) is upper bounded by \( 2^{k+3} \), it now follows that

\[
g_\lambda(\bar{y}) - g_\lambda(\hat{y}) \geq G_d 2^{(k-d)(k+5)} - \sum_{i > d} |G_i| 2^{(k-i)(k+5)} > 0,
\]

and the claim is proven.

Finally, we establish that the parametric complexity of knapsack is \( \Omega(2^k) \):

3.5.6. **Claim.** The slope of \( m^*(\lambda) \) is unique for each interval

\[
([\lambda] + \frac{1}{4}, [\lambda] + \frac{3}{4}) \subset [0, 2^k].
\]

Looking at equation 3.2, from the previous claim it follows that \( m^*(\lambda) = g_\lambda([\lambda]) \), which simplifies to the linear form:

\[
m^*(\lambda) = g_\lambda(\lambda_1, \ldots, \lambda_k) = \sum_{i : \lambda_i = 1} 2^{(k-i)(k+5)} \lambda + A_{[\lambda]}
\]

for some constant \( A_{[\lambda]} \). And so each binary expansion of \( [\lambda] \) gives rise to a unique slope.

We now state the consequences for parallel algorithms attempting to approximate the knapsack problem.

3.5.7. **Corollary.** Knapsack instances with \( n \) items cannot be approximated with error ratio \( 1 + \varepsilon \) in time \( (\log \frac{1}{\varepsilon})^{\frac{1}{2}} / a \), and using \( 2^{(\log \frac{1}{\varepsilon})^{\frac{1}{2}}} / a \) processors, for some positive constant \( a \), in Mulmuley’s parallel PRAM model without bit operations.

**Proof.** By the same proof as in Claim 3.3.2 we can show that optimally solving a knapsack instance (having \( v = w \)) with \( n \) items and bit-length \( n \) reduces to solving the same knapsack instance with error ratio \( 1 + \frac{1}{n^2 \varepsilon} \). So setting \( \frac{1}{\varepsilon} = n2^n \), by Theorem 3.5.2 this cannot be done in KC(2^{n^{\frac{1}{2}}} / b, n^{\frac{1}{4}} / b) for some \( b \), and hence neither in KC(2^{(\log \frac{1}{\varepsilon})^{\frac{1}{2}} / a}, (\log \frac{1}{\varepsilon})^{\frac{1}{2}} / a) for some larger \( a \).

From our construction we also derive a new lower-bound for solving knapsack exactly, which matches the lower bound Mulmuley [77] proved for max-flow.

3.5.8. **Corollary.** Knapsack instances with \( n \) items and bit-length \( O(n^2) \) cannot be solved in time \( \sqrt{n} / a \), using \( 2^{\sqrt{n} / a} \) processors, for some positive constant \( a \), in Mulmuley’s parallel PRAM model without bit operations.
3.6 Open Problems

We consider the following two open problems worthy of further research. First, we have proven a lower bound for the optimization version of the subset-sum problem (the problem of computing $S(w, W)$ exactly), even in the case of small input bit-lengths. Can we prove a lower bound for the decision version of subset-sum? Surprisingly, there is a reduction from the optimization version to the decision version [cf. 82], but this reduction is not algebraic, or even semi-algebraic, and so the question remains open.

Second, we have proven that we can approximate knapsack up to error $1 + \varepsilon$ in $\log n$ time with $(\frac{1}{\varepsilon})^{\log n}$ processors, and that we cannot do it in time $(\log \frac{1}{\varepsilon})^{\frac{1}{2}}/a$, and using $2(\log \frac{1}{\varepsilon})^{\frac{1}{2}}/a$ processors. Can this gap be closed?