2022

LASSO: Listing All Subset Sums Obediently for Evaluating Unbounded Subset Sums

Christopher N. Burgoyne
University of Montana

Travis J. Wheeler
University of Montana

LASSO: LISTING ALL SUBSET SUMS OBEDIENTLY FOR EVALUATING UNBOUNDED SUBSET SUMS

By

Christopher Nicholas Burgoyne Ph.D.

PhD, University of Johannesburg, Johannesburg, South Africa, 2019

Thesis

presented in partial fulfillment of the requirements
for the degree of

Master of Science
in Computer Science

University of Montana
Missoula, MT

Spring 2022

Approved by:

Ashby Kinch Ph.D., Dean
Graduate School

Travis Wheeler Ph.D., Chair
Computer Science

Jesse Johnson Ph.D.
Computer Science

Cory Palmer Ph.D.
Mathematics

Burgoyne, Christopher Nicholas (BSocSci, BAHons, MA Cum Laude, MSc, PhD), May 2022

LASSO: Listing All Subset Sums Obediently for Evaluating Unbounded Subset Sums

Chairperson: Travis Wheeler Ph.D.

In this study we present a novel algorithm, LASSO, for solving the unbounded and bounded subset sum problem. The LASSO algorithm was designed to solve the unbounded SSP quickly and to return all subsets summing to a target sum. As speed was the highest priority, we benchmarked the run time performance of LASSO against implementations of some common approaches to the bounded SSP, as well as the only comparable implementation for solving the unbounded SSP that we could find. In solving the bounded SSP, our algorithm had a significantly faster run time than the competing algorithms when the target sum returned at least one subset. When the target returned no subsets, LASSO had a poorer run time growth rate than the competing algorithms solving bounded subset sum. For solving the USSP LASSO was significantly faster than the only comparable algorithm for this problem, both in run time and run time growth rate.

ACKNOWLEDGMENTS

To my long-suffering wife: may this lead us to the life we want.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION

Subset Sum is a well-studied task that was among Karp's famed 21 NP-complete problems [2]. The canonical version of the Subset Sum Problem (SSP) is: given a set S of n positive integers {s_1, ..., s_n} and a positive integer t, does there exist a subset S' of S such that the sum of the elements in S' equals t? Despite the likely worst-case intractability of SSP, Bellman's seminal dynamic programming approach [3] produces a solution in pseudo-polynomial time. In this approach, let S_j = {s_1,...,s_j} (where $1 \leq j \leq n$), and T[j,k] = 1 if there is a subset of S_j that sums to k, and 0 otherwise. With the recurrence relation T[j,k] = { T[j-1,k-s_j] — T[j-1,k] }, the solution to SSP is the value of T[n,t]. The time to fill this dynamic programming matrix is O(nt). When $t < 2^n$, this approach improves on the O(2^n) run time for a naïve algorithm.

Since the introduction of Bellman's algorithm in 1957, a number of solutions have improved the theoretical run time of solving SSP. In 1999, Pisinger [8] published an algorithm for SSP with time O(n max(S)) and O(t) space. In 2003, Pisinger achieved a more broadly recognized improvement using a bit-packing technique that represents sub-solutions as bits in an integer [9] and applies the Bellman recurrence for SSP, achieving a solution in O(nt/logt) time and O(t/logt) space. This represented the first major run time improvement with the introduction of a log factor. In 2017, Bringmann presented a randomized heuristic solution [5] to SSP that runs in $\tilde{O}$(n+t) time and O(nt) space where $\tilde{O}$ hides polylog factors, as well as a solution that runs in $\tilde{O}$(nt) time with space $\tilde{O}$(nlogt), assuming the Extended Riemann Hypothesis. Another recent achievement was made by Koiliaris and Xu with their algorithm which achieves a run time of O(nt) in space O(t) and can be used to produce all subset sums up to the integer t in time O(minn t, t5/4, σ), where σ is the sum of values in the input set6.

1.1 Variants of the Subset Sum Problem

A number of variants of the SSP have received attention in the literature. One of these is the modular SSP: given a target integer t, a modulus m, and a multiset S of n positive integers with value less than m, is there a subset of S with elements summing to t (mod m) [10,11]. Another variant is the generalized SSP, in which the cardinality of the solution subset must be less than a pre-specified integer value [12]. In the equal-sums subset sum problem involves finding two subsets for which the components of each subset add up to the same value [13]. The Unbounded Subset Sum Problem (USSP) is similar to classical SSP, but allows each element of S to be used more than once; as a result the target t in a USSP can be unbounded, while the SSP target is bounded by the sum of the values in S [15,24]. We concern ourselves here with USSP.

1.2 The Unbounded Subset Sum Problem

The unbounded subset sum problem was posed by Alfonsin (1996) as the Subset Sum with Repetitions Problem and it was shown to be NP-complete even where elements of the input set are not repeated [16]. The same year, Hansen and Ryan solved the problem in time O($p_1{}^2 + n$) where the input set is composed of $p_1,\ldots,p_n$ coprime integers [17]. Since then, there has been a growing body of literature that refers to this problem by the title Unbounded Subset Sum. Muntean and Oltean (2009) presented special hardware that uses light beams to determine if there is a solution to the problem [18]. The device contains a graph-like structure which allows light to travel in specified paths with the length of the paths being defined by the values in the input set. With light flowing through the device it will exit the paths in varying amounts of time. By measuring the time taken for light to flow through the paths, a boolean solution can be determined.

A significant contribution by Bringmann (2017) led to a solution obtainable in time O(t log t) [5]. Wojtczak posed the problem as being strongly NP-complete with rational numbers, meaning that there is no pseudo-polynomial algorithm to solve this problem unless P=NP [19]. Jansen and Rohwedder (2019) produced a logarithmic time solution of O($s_n \log s_n$) [20], with the parameter s_n being the largest value of the input set S, and providing a stronger bound on run time than

Bringmann's use of the target value t. There are other solutions that solve USSP in polynomial time but these solutions typically place special constraints on the inputs, such as with the algorithm by Salimi and Mala (2021) [21].

1.3 Limitations of Current Approaches

A similarity among most of these existing solutions to both the bounded and unbounded variants of the subset sum problem is that they rely on the input set being a set of integer values. However in real-world applications of this problem, it is likely that floating point inputs could be required. A possible solution to this is to multiply the floating point input values by a fixed number, thus producing an integer that represents a floating point number with a fixed degree of precision.

One foreseeable problem with this approach is that the run times of these existing algorithms typically depend on the magnitude of the input values. This means that the run times will increase with increasing floating point precision of the input values. Moreover, where run times do not depend on the magnitude of the input values, there are typically special conditions on the input values that limit their ability to produce universal solutions [21,23]. In any case, a solution with a run time that is not dependent on the magnitude of the input values could have a broader range of possible applications.

Another important similarity among existing solutions to these problems is that they produce a binary solution: they answer the yes/no question about the existence of a subset summing to the target value. What they typically cannot do is enumerate all subsets of S that lead to an affirmative solution. This is harder than NP-hard due to the possibility of multiple exponential values affecting run time and is thus a very challenging problem to solve. Other than the fact that the canonical problem poses a yes/no question, one of the reasons for this is the popular use of the Fast Fourier Transform which reduces polynomial multiplication from polynomial time to logarithmic time, a significant reduction in computational cost [6,15]. In using this technique, the composition of summed values relative to their inputs is lost and recovery can be expensive or impossible. As such, they lack the ability to provide the entire set of combinations of input set

elements that sum to the target value.

While there are many applications where solving the yes/no question of the unbounded subset sum would be useful, there are a number of applications where this solution is insufficient. Moreover, it is often useful to produce a catalog of all possible subsets that match the target. For example in metabolomics, the first step in identifying a metabolite usually involves calculating possible elemental compositions of a molecule [25]. One approach to this would involve applying the unbounded subset sum problem to an input set that represents the masses of possible elements in the metabolite. While this does not necessarily produce an ordered molecular composition, it is a reasonable start to the search for one [25]. Other practical applications of the unbounded subset sum problem are in supply chain logistics and demand planning, which involve calculating optimal combinations from a set of available items which can be included in those combinations more than once [26,27].

1.4 Overview

In this paper we present an algorithm that solves the Unbounded Subset Sum Problem and produces all combinations of an input set S that sum to the target value t. The algorithm accepts floating point inputs, can optionally provide all or a single target-matching combination, and can also trivially solve the yes/no Unbounded Subset Sum Problem. The algorithm runs in two parts. The first part sums and stores certain combinations of input set values in a basic data structure. The second part is a branch and bound approach that takes advantage of calculations performed in the first part to prune areas of the numeric search space that cannot produce combinations summing to the target. In this paper we describe the algorithm and present a worked example to aid understanding.

In demonstrating the efficacy of this algorithm, we ran a series of run time tests to compare our algorithm against a benchmark of other publicly available algorithms that could be adapted to perform similar tasks. Empirically, our algorithm demonstrates a significantly faster run time than competing algorithms when asked to retrieve all combinations summing to the target. It has an

observed run time that is as low as quadratic for certain cases of solving the yes/no question and we demonstrate that it is bounded in the worst case by the number of combinations summing to the target value. The theoretical run time of our algorithm depends on the size of the input set and the magnitude of the target value in relation to that input set, but due to the extensive pruning that occurs in the second part of the algorithm the empirical run time is typically much faster than this. Due to the input-dependent nature of the algorithm's pruning strategy, we have not been able to determine a closed form for the expected run time but we do demonstrate instances of the theoretical worst case.

In the following sections, we discuss the design of the algorithm, implementation details, constraints, and practical run time results. The algorithm is open-source and is hosted at https://github.com/koos-burgoyne/LASSO.

CHAPTER 2 METHODS

The LASSO algorithm works by performing a preprocessing step to compute and store sums of subsets up to some predetermined size m, then using these precomputed values as the basis of a fast exploration of sums of subsets with larger size. The precomputed subset sum values are stored in memory, in a data structure that we refer to as a Zeroboard. The main routine leveraging the Zeroboard makes use of a branch and bound strategy to prune parts of the search space that cannot contain a subset summing to the target sum. Below we describe the concept of a Zeroboard, and describe how it is filled, then used in the final stage.

2.1 Lookup Tables for Subset Sum – a prelude

Use of the Zeroboard in the LASSO algorithm is analogous to the use of a hash-table to achieve expected O(n) run time for the 2-SUM problem. To set the stage for a complete explanation of the Zeroboard, we begin by discussing conventional approaches to solving the 2-SUM and 3-SUM problems and how a Zeroboard can be applied to these problems.

In the 2-SUM problem, an input set of n integers is queried to determine if two of the values sum to a target value. The naive approach is to produce all subsets of size two and check if their sum equals the target, with a run time of O(n^2). An improved approach is to first employ the preprocessing step of entering the n values into a hash-table, then iterate over elements in the set, for each element querying the hash-table for the difference between the current element and the target sum. The preprocessing step requires an expected O(n) run time (n values inserted into a hash table with expected constant run time), and the search stage also requires O(n) time (n elements, each requiring a single lookup with expected constant time). Thus, the run time of the

algorithm is O(n).

To solve the 3-SUM problem in less than O(n^3) time, a similar technique can be used. One can store the values of the input set in a hash-table (linear time and space) and then iterate over subsets of size 2 (quadratic time) to query the hash-table for the difference between the current subset sum and the target sum. Alternatively, the preprocessing step can iterate over all subsets of size two, storing sums of those size-2 subsets in a hash-table (quadratic time and space). Subsequently, during a linear pass over the input set, one can query the hash-table for the difference between the current 2-set sum and the target sum. Both approaches require quadratic time, but the first requires space that is only linear in the size of S.

2.2 Defining and using the Zeroboard Data Structure

2.2.1 The Zeroboard Data-Structure

In the hash-table approach to solving 2- and 3-SUM, the hash-table is typically used to store the sum of a subset. The Zeroboard concept reframes this idea of querying for sums in constant time. Instead of using the sum of a subset of a particular size as the key in a hash-table, we define a Zeroboard key as the difference between the sum of the largest subset stored in the Zeroboard and the sum of the subset being stored with that key. All values stored in a Zeroboard are thus relative to the largest subset sum stored in it. This numeric structure is what gives the Zeroboard its name: the smallest key stored in the Zeroboard is guaranteed to be zero and only the largest subset will be stored with this key. Figure 2.1 presents an example of the representation of subset and Zeroboard values. All examples depend on the following nomenclature to define the subset space S and the Zeroboard space Z in which truncated subset sums are stored.

$$S = \{S_1, S_2, S_3, S_4\} = \{2, 4, 8, 11\}$$

Let $\mathbb{S}_{(d_1,d_2,\ldots,d_k)} = S_{d_1} + S_{d_2} + \ldots + S_{d_k}$

e.g. $\mathbb{S}_{(1,1,2,3)} = S_1 + S_1 + S_2 + S_3 = 2 + 2 + 4 + 8 = 16$

Let $\mathbb{Z}_{(d_1,d_2,\ldots,d_k)} = \mathbb{S}_{(n,n,\ldots,n)} - \mathbb{S}_{(d_1,d_2,\ldots,d_k)}$

e.g. $\mathbb{Z}_{(1,1,2,3)} = \mathbb{Z}_{(2,2,2,2)} = 28$

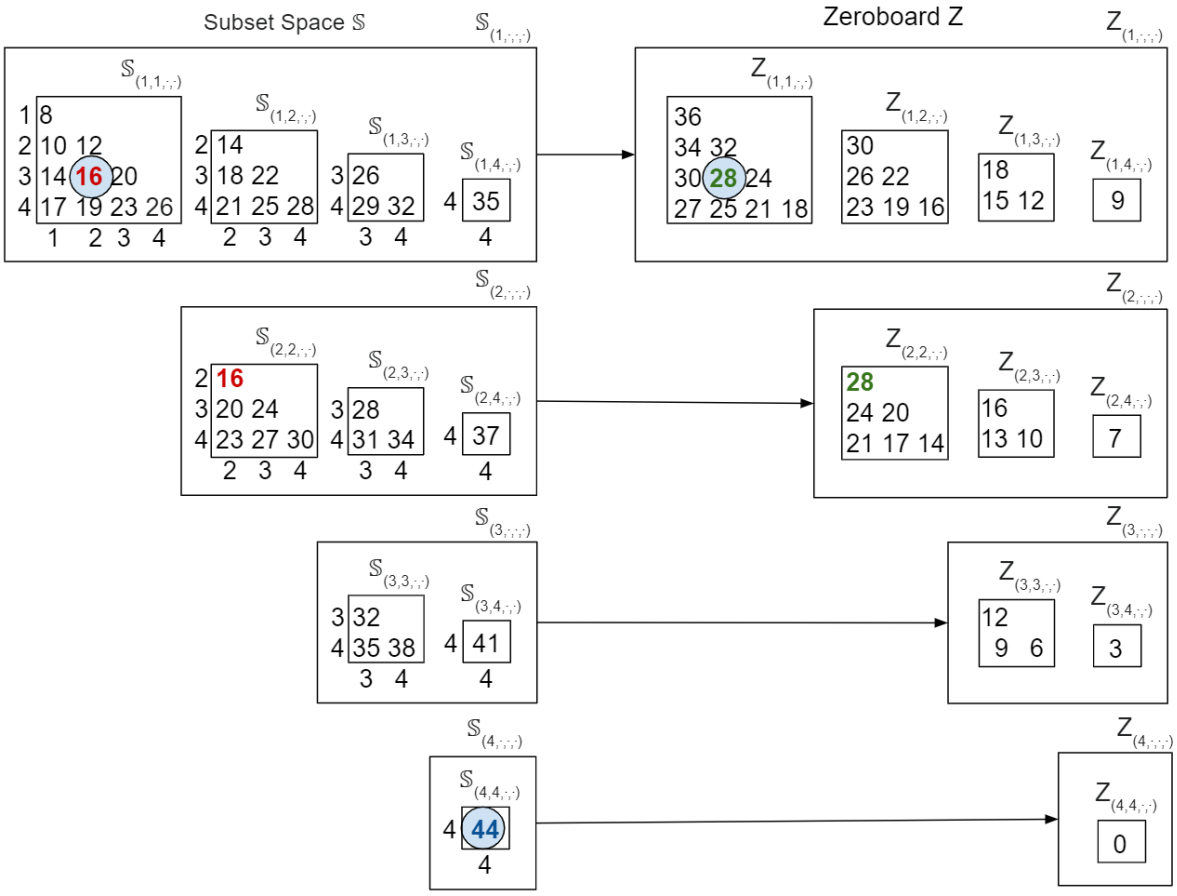

Figure 2.1: An example of the subset space S and an equivalent Zeroboard, both containing subsets of size four. Note the definitions of nomenclature that describe individual subsets and groups of subsets found in the subset space S and the Zeroboard space Z.

2.2.2 Solving Subset Sum of Size 1 using a Zeroboard

We begin by exploring the (degenerate) task of searching a set S for a subset of size one that matches a target t. Consider an input set S = {2, 4, 8, 11}, with size n=4. To find subsets of

size one that sum to the target value, the only dimension possible for a Zeroboard is 1. In order to construct this Zeroboard, we perform a linear pass over the input set S. For the i^{th} value, the Zeroboard key will be calculated as $Z_{(i)} = S_{(n)} - S_{(i)}$ as shown in Figure 2.1. After a linear pass over the input set, Figure 2 shows that the Zeroboard would contain the keys {9, 7, 3, 0}. A single query of the Zeroboard is required to find any subsets of size 1 that "sum" to the target value. The query key is calculated as $Z_{(k)} = S_{(n)} - t$ and a single search of the Zeroboard will find a match if one exists. In total, this requires linear pre-processing time and space, followed by constant time lookup.

For example, if t=8, the Zeroboard search value is 11-8=3, which matches 3. In contrast, if t=7, then the Zeroboard query key is 11-7=4, which is not found in the Zeroboard.

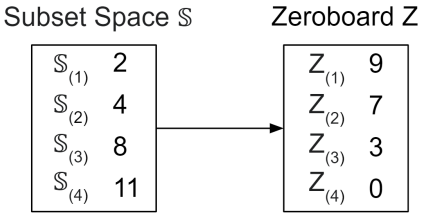

Figure 2.2: A depiction of the subset space that contains subsets of size 1 from the input set {2,4,8,11} with the corresponding 1-dimensional Zeroboard.

2.2.3 Solving Subset Sum of Size 2 using a Zeroboard

Consider the same input set S from above. To find size-2 subsets of S that sum to a target value, it is possible to use the 1-dimensional Zeroboard from Figure 2.1. For the second (search) phase, we iterate over all subsets of the subset size one – we refer to the remaining number elements in the target set size as the residual subset size. For each element in the subset of size one, we search the Zeroboard for a matching value. This solution is achieved in O(n) time and space.

As an alternative approach, the Zeroboard can be written in 2 dimensions, as depicted in Figure 2.3. Writing such a Zeroboard requires iterating over all subsets of S of size 2 and storing their Zeroboard values. The work required for this pre-processing step is quadratic. A single query of

the Zeroboard will determine the existence of a match, as in the previous section. This is a poorer approach as the overall run time and space is O(n^2).

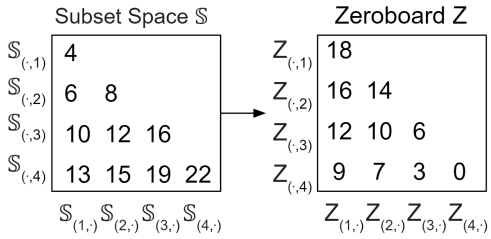

Figure 2.3: A depiction of a 2-dimensional subset sum space from set {2,4,8,11} that contains subsets of size 2, with the corresponding 2-dimensional Zeroboard.

2.2.4 Solving Subset Sum of Size 3 using a Zeroboard

Using the input set S from the previous sections, consider the problem of finding all subsets of S with size 3. To solve this problem there are three possible values for Zeroboard dimensionality m. In the first case, the Zeroboard can be 1-dimensional. During the first subroutine, a Zeroboard is written as in Figure 4. The Zeroboard keys are calculated using subsets of size one, so the search routine will iterate over all subsets of size 2, seeking a matching size-1 addition to reach the target t. This approach requires quadratic time in that the search subroutine to iterate over all subsets of size 2, but the space required is linear for the values in the Zeroboard.

In the second case, we produce a 2-dimensional Zeroboard. During pre-processing, the Zeroboard is written by iterating over all subsets of size 2 and storing the key associated with the respective subset sum. This requires quadratic time and space. In the search subroutine, each subset space S(i,,) can be queried for valid subsets using the query key equation from Figure 2.1 during a linear pass over the input set.

In the third case, the Zeroboard can be 3-dimensional. In this case, the Zeroboard is written by iterating over all subsets of size 3 and storing the key associated with each subset. This requires cubic time and space. The search routine performs a single search against the Zeroboard. This approach is inferior to the other two because of the greater run time and space requirements.

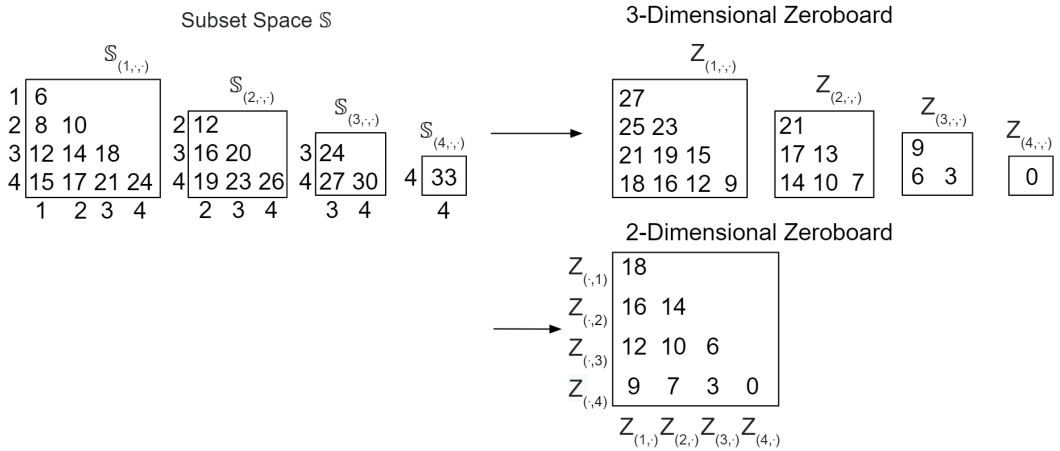

Figure 2.4: A depiction of a 3-dimensional subset space from a set {2,4,8,11} and two Zeroboards of different dimensions that can be derived from the subset space.

2.2.5 Solving Subset Sum of Size k using a Zeroboard

We can also use a Zeroboard to solve the SSP where the required subset size is any integer k. The range of possible values for the Zeroboard dimensionality (m) is from 1 to k. The run time and space for Zeroboard creation is Omnm, because there are nm combinations, each of size m with an input set of size n. If all residual (r) subsets are considered in the search stage, its run time is expected O(nm), because there are nm combinations, and each combination induces a single hash table (Zeroboard) lookup.

2.2.6 Solving Subset Sum using a Zeroboard in LASSO

In the cases described above we were solving specific k-sized subset sum problems. However, in its typical form the SSP does not specify the size of the subset that should be found. In the general case of the SSP, an algorithm is tasked with finding subsets of any size that sum to the target sum. In order to achieve this, our algorithm uses an approach that replicates the size-k solution described above across all possible sizes of subset. The range of subset sizes k that are possible is calculated as

$$k_{\max} = floor(\frac{t}{S_n})$$

$$k_{\max} = ceiling(\frac{t}{S_0})$$

where k_{min} and k_{max} are the minimum and maximum subset sizes possible for the given target sum and input set, n is the size of the input set, and t is the target value. By creating a Zeroboard that stores subsets of size k_{min} we can use it to solve every k-sized SSP from k_{min} to k_{max} and thus solve the SSP.

2.3 Algorithmic Details

2.3.1 Pre-processing: Writing the zeroboard to memory

The values of k_{min} and k_{max} are computed, in order to determine the value for m, the subset size stored in the Zeroboard. Though the ideal value of m is expected to be k_{min}, it's selected value is constrained by the space requirements of storing $\binom{n}{m}$ combinations. Supposing LASSO is allowed to use b bytes of memory, a proper choice is to let m=floor($\log_n b$,k_{min}).

In order to write a Zeroboard storing subsets of size m, we iterate over all nm subsets of size m and calculate the key for each subset using the equation described in Figure 2.1. At this point, the values in each subset can be entered into a bag of subsets stored under that key allowing for retrieval of those subsets later. In a naive approach to calculating the Zeroboard keys, each subset sum would need to be calculated by summing all the m values in each subset. In order to optimize this process, it is useful to note that the subset sums in a Zeroboard are related to each other by the differences between respective values in the subsets. For example in the 3-dimension Zeroboard of Figure 2.4, the difference between $S_{(1,4,4)} = 24$ and $S_{(2,2,2)} = 12$ is equal to (S_1 - S_2) + (S_4 - S_2) + (S_4 - S_2) = -2 + 7 + 7 = 12. In order to calculate the next Zeroboard key, we simply use the previous key and add the differences between the values that make up each subset. In cases where some of the indexes are the same across subsets, they need not be computed, thereby reducing the work done to calculate subset sums.

2.3.2 The Search Stage: Querying the zeroboard

The second part of the algorithm queries the Zeroboard to find subsets summing to the target sum. In the naive approach, we iterate over all possible subsets of input set values that the Zeroboard can be queried with.

To understand this better, let us consider a SSP where k = 4, and a Zeroboard that stores subsets of size m = 3. Each Zeroboard key for the subsets of size 3 is calculated as $Z_{(i,j,k)} = S_{(n,n,n)} - S_{(i,j,k)}$. The residual subset size not calculated in the pre-processing step (which wrote the Zeroboard) is r = k - m = 4 - 3 = 1, which is simply all subsets of size one. Iterating over all subsets of size 1 simply requires one linear pass over the input set values, with one Zeroboard query (hash table lookup) per input set value s_i. For each query we calculate the Zeroboard query key as $Z_{(i,\cdot,\cdot)} = S_{(i,n,n,n)} - t = (s_i + s_n * m) - t$.

As another example, consider a k-sized SSP where k = 5 and m = 3. In the naive approach we query the Zeroboard with every possible subset of size r = k - m = 2. In this case, for each pair (i,j), the key for a query of the Zeroboard is calculated as $Z_{(\cdot,\cdot,\cdot)} = S_{(i,j,n,n,n)} - t. = s_i + s_j + s_n * m) - t$.

2.3.3 Pruning the Search Space

An important optimization in the LASSO algorithm is that it prunes parts of the search using a simple assumption about where the target sum can be found in the subset space. Note that the subset space can be portioned into sections with the same dimension as an m-dimension Zeroboard. Each m-dimension subset space section can be queried to find valid subsets and in the naive approach, each of these sections will be queried. However, the target sum will not occur within each m-dimension subset space section. The target sum can only occur within sections where it is between the minimum and maximum values for that section. The rest of the subset sections where the target value cannot occur can be pruned.

As an example from the subset space of Figure 1, a query value of 16 can only exist in $S_{(1,\cdot,\cdot,\cdot)}$ and $S_{(2,\cdot,\cdot,\cdot)}$. If the Zeroboard dimension m is 4, then a single query of the Zeroboard covers the

entire subset space. If m = 3 then the search-stage of LASSO involves iterating over the residual combination size r = k - m = 4 - 3 = 1. This simply entails iterating over the values in the input set and using each one to query the 3-dimension Zeroboard for valid subsets.

In order to prune the subset space of Figure 1 when LASSO is using a 3-dimension Zeroboard, LASSO begins by checking if $S_{(1,n,n,n)}$ is larger than the target sum. That value is 35 and it is larger than 16, so the Zeroboard must be queried using $Z_{(\cdot,\cdot,\cdot)} = (s_1 + s_n{*}m) - t = 35 - 16 = 19$. This value of 19 is found in the Zeroboard so a valid subset is found. Given that the subset sums in $S_{(i+1,n,n,n)}$ are guaranteed to be larger than the respective sums in $S_{(i,n,n,n)}$, LASSO now moves to checking $S_{(i,i,i,i)}$ for each subsequent $S_{(i,\cdot,\cdot,\cdot)}$. So for the next subset space section $S_{(2,\cdot,\cdot,\cdot)}$ the algorithm checks if $S_{(2,2,2,2)}$ is larger than the target sum. This value is 16 which is equal to the target sum 16. Because the value $S_{(i,i,i,i)}$ can only occur once with $S_{(i,\cdot,\cdot,\cdot)}$, the subset can be recorded without querying the Zeroboard. Note that due to sorted ordering, all further $S_{(i,i,i,i)}$ values will be greater than the target sum. This means that the target sum cannot be found in subsequent parts of the subset space, and the search-phase of LASSO comes to an end.

2.4 Implementation Details

2.4.1 The Zeroboard Data-Structure

In its simplest form, the Zeroboard data-structure is essentially a hash-table where the key is a hash-value calculated as in Figure 1 and the associated bin can contain a list of subsets that are associated with that key. During experimentation, it was found that the fastest implementation of this hash-table was achieved using the unordered_map data-structure from the Boost C++ library[1], with the key as a double type and the bin containing a pointer to the head of a linked list of the subsets associated with that key. The hash-table is implemented as an object of the Boost::unordered_map class and additional functions were written for inserting subsets associated with a key, as well as retrieving those subsets.

[1] https://www.boost.org/doc/libs/1_79_0/*libs/unordered/doc/html/unordered.html*

2.4.2 Storing Subsets and their Sums

In order to insert a subset sum into the Zeroboard, the key must be calculated. The function will then access the Zeroboard entry associated with that key and add the subset values to the head of a linked list that is stored in that bin. A result of this approach is that the subsets are stored in the list in a decreasing order of the values in each subset. This means that when looking at the values in the ith subset, the value at any index in that subset is guaranteed to be less than the value at that index in any subsequent subset in the list.

2.4.3 Performing Lookups by Subset Sum

One important caveat is that when querying a Zeroboard, not all subsets should be included in the bag of subsets returned. If we return all subsets in the bag with every query there will be repetition of certain subsets.

To understand why, we find it useful to consider a practical example. In Figure 4, there is a 3-dimension subset space (left) and a 2-dimension Zeroboard (bottom right). Consider a target t=10 (which occurs once at $S_{(1,2,2)}$). Using the 2-dimension Zeroboard, the query step first searches the Zeroboard based on the value s1, so that it queries $S_{(1,\cdot,\cdot)}$. The corresponding query key is $Z_{(2,2)}$ = (s_1 + s_n*2) - t = 24 - 10 = 14. Because this key (14) is found in the 2-dimension Zeroboard, LASSO returns a valid subset by concatenating the subset stored in the Zeroboard {2,2} to the residual subset {1}. This results in the subset {1,2,2}.

Subsequently when the algorithm queries $S_{(2,\cdot,\cdot)}$ the Zeroboard query key will be calculated as $_{(\cdot,\cdot)}$ = (s_2 + s_n*2) - t which results in 26 - 10 = 16 = $Z_{(1,2)}$. This key is found in the Zeroboard and the concatenated subset returned is {2,1,2}. This subset is the same as the subset returned from the previous query (because order is not important). This repetition arises because the query returned the subset $S_{(2,1,2)}$ which does not exist in the subset space.

LASSO avoids this problem by performing a simple test. If the last index in the residual subset is greater than the first index of the subset stored in the Zeroboard then the subset is invalid. This test basically checks for a sorted subset. If the subset returned from a query is not sorted, then the

algorithm is adding subset sums to the subset space where they do not exist.

2.4.4 Reduced Precision Keys

The LASSO algorithm was written specifically to be able to deal with floating point number inputs. Floating point arithmetic is prone to rounding error on computers, due to their binary floating point representation. LASSO addresses this error by establishing an epsilon value (a user-definable power of ten), and rounding all Zeroboard keys to the corresponding level of precision. Allowing for this epsilon variance means using a specialized data-structure for the Zeroboard. We calculate the Zeroboard key with reduced precision as follows:

$$\mathbb{Z}(...) = ceil[(\mathbb{S}(..., n) - \mathbb{S}(...)) * dp]/dp$$

where dp is the exponent of epsilon's power of 10. As a result, each bin will now contain a list of subsets with the same low-precision key, but potentially different full-precision sums. In order to retrieve the maximal precision using this approach and ensure that subsets grouped together sum to the same value, we can modify the bag associated with a key by making it a bag of bags where each bag contains subsets that sum to the same value at full precision. The number of bags in this approach remains relatively small so experimentation showed that a doubly linked list yielded better performance than a resizing vector. The bin contains a reference to the head and the tail of the list as a small optimization: when accessing a bin that contains subsets summing to a smaller value the search can start at the tail of the list and stop when it gets to a bag that contains subsets summing to a value less than the query value minus epsilon.

One application-specific benefit of reduced precision Zeroboard keys is to streamline identification of possible subsets of amino acids in peptides that are fragmented and measured using a mass spectrometer [25]. In addition to the set of amino acids involved in producing some weight measured by the mass spectrometer, there can also be modifications (e.g. methylation) to the molecules that slightly alter their function and weight. In order to find modified molecules, LASSO can be used to explore the range of potential modified molecule masses by retrieving all subset

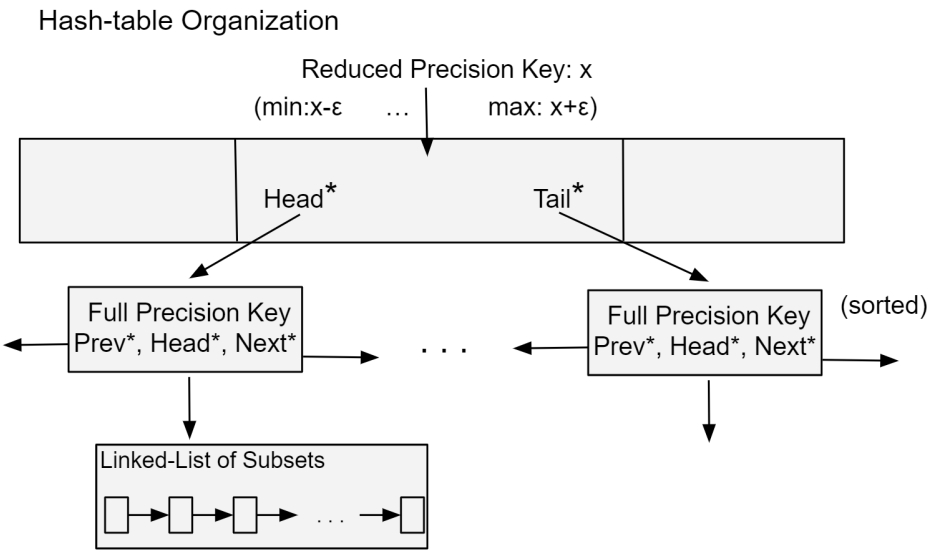

Figure 2.5: A Zeroboard data-structure that allows for epsilon variance in the input values. The table at the top is a Boost unordered_map, an implementation of a hash-table. The two blocks below the map are structs that hold the full precision key, with pointers to neighboring structs in the same bin and a pointer to the head of a linked-list of subsets that sum to the same value. Note that the range of full precision keys stored in a Zeroboard bin are within epsilon range of the reduced precision key for that bin.

sums within epsilon range of the target sum.

CHAPTER 3 RESULTS

3.1 Comparative Run Time Testing

3.1.1 Benchmark Algorithms

Since an implementation of a method is the only way an algorithm can be practically used to solve a problem with specific inputs, we have sought to explore the efficacy of a few of the implementations that we were able to identify that represent some common approaches to solving the SSP. Like Pisinger [34], we designed a variety of test cases that would explicitly test the ability of our algorithm and competing SSP algorithms to handle a range of input values that would change run time performance. We present the performance of seven competing algorithms (see Table 3.1) on these test cases. These implementations are all publicly available and they cover a range of approaches. We found only one implementation of an algorithm that solves the multiple-solution unbounded SSP problem.

Identifier	Method	Language	Returns	Reference
KoiliarisXu	Dynamic Programming	Java	All subset-sum values	28
Std DP	Dynamic Programming	Python	T/F	29
Opt DP	Dynamic Programming	Python	All valid subsets	29
Exact PyTools	Exact	Python	One valid subset	30
Memoized	Exact, Memoized	Python	One valid subset	31
Randomized	Randomized, Heuristic	Python	T/F	32
Hetland	Approximate	Python	One valid subset	33

Table 3.1: A summary of competing algorithms for benchmarking run time of the LASSO algorithm.

3.1.2 Testing Platform

The testing platform was written as a series of Python scripts that each ran one of the tests listed above. The tests were run on a Dell Inspiron 15 5575 Laptop with a 7th generation Intel Core i7 processor, 32GB of RAM, and a 500GB solid state storage drive. The operating system was Microsoft Windows 10 and was up to date with all OS updates in May 2022. To run a test, the laptop was restarted with no ancillary applications running at startup. The Python scripts that contain the tests were then run on the machine sequentially and the run times of the algorithms for each test set of inputs were recorded. The testing platform is freely available for reproducing results and is hosted at https://github.com/koos-burgoyne/LASSO_testing.

3.2 Bounded Subset Sum

The LASSO algorithm was designed to solve the unbounded SSP producing multiple solutions, but we find it informative to start with the more constrained solution of the bounded SSP because that problem is more widely addressed in both the literature and in implementations. In order to benchmark LASSO against other algorithms that solve bounded subset sum, we created a modified version of LASSO that returns the first valid subset found. This modification makes LASSO comparable to most of the algorithms listed in Table 3.1.

3.2.1 Variable Input Set Size

We benchmarked the modified single-solution LASSO implementation against the competing algorithms that solve the bounded SSP. The test used a growing size of input set to evaluate performance with variable n, with input sets ranging in size from n=5 to n=30, with values 10, 12, 14, ..., 10+2(n-1). The target was set as half the sum of the values in the input set, rounded up to the nearest even integer.

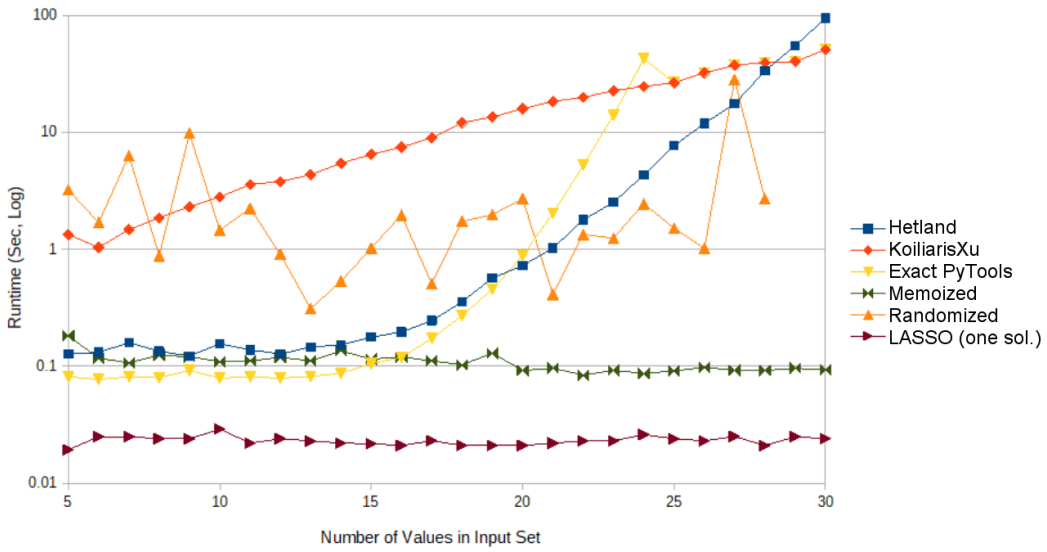

Figure 3.1: A plot of the run times showing the log-scaled run time of bounded subset sum algorithms in seconds by increasing size of the input set. Note that LASSO is modified to return a single result. The input sets used comprised $\{2x10^6 : x \in \mathbb{Z}, 5 \leq x \leq 4+n\}$ where n ranged from 5 to 30 as seen on the x-axis. The target value was calculated as the sum of the inputs, divided by 2 and rounded up to the nearest subset sum guaranteed to return at least one combination.

In Figure 3.1 the run time for LASSO appears asymptotically constant for solving the bounded SSP with a boolean response. We explicitly state that while it appears to be a constant run time, the complexity of generating the Zeroboard is $\binom{n}{m}$ in the modified LASSO implementation. We test LASSO further in Figure 3.3. For the other algorithms that performed this task, two implementations were increasing exponentially when the input set size increased beyond roughly 15. The only algorithms other than LASSO that presented a constant run time were a randomized algorithm that returns a heuristic solution and a memoized implementation.

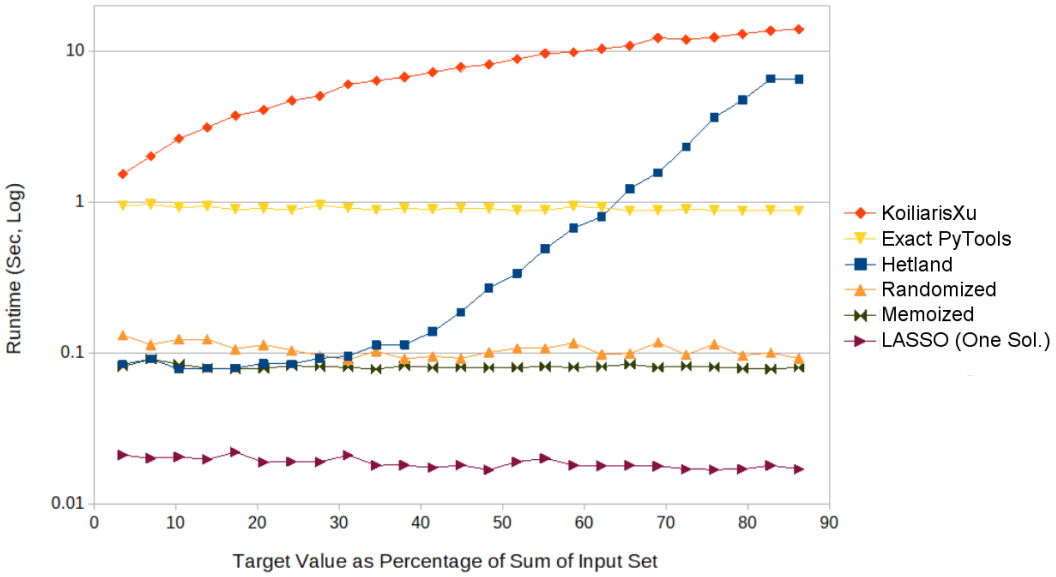

Figure 3.2: A plot of the run times of bounded subset sum algorithms showing log-scaled run time in seconds by increasing size of target value as a percentage of the input set sum. The inputs set remained static over all targets. All target values are guaranteed to return at least one subset.

3.2.2 Variable Target Sum

In another test, we compared the ability of our algorithm to deal with a static input set and a growing target value for the bounded SSP. This is useful as most SSP algorithms are based on the dynamic programming approach and so have a run time that depends on the target value. We used target values for this test that were defined as $\{2\text{x}10^6 : \text{x} \in \mathbb{Z},\ 5 \leq \text{x} \leq 25\}$. Similar to the previous tests, the input set remained static as $\{2\text{x}10^6 : \text{x} \in \mathbb{Z},\ 15 \leq \text{x} \leq 24\}$. These input values were used to determine the impact of increasing the target value over a static input set for the bounded SSP, with target values being guaranteed to return at least one combination.

The results of Figure 3.2 show a run time for LASSO that is constant with increasing target values over a static input set, and also faster than all competing algorithms that solve the bounded SSP. In this result, the run time will remain constant because the complexity of LASSO does not depend on the magnitude of the target value.

3.2.3 Modified LASSO with Larger Input Set Size

To test the idea that the performance of LASSO will grow quadratically as the input set size increases, we tested the modified implementation to measure run time as n grew beyond the range of sizes seen in Figure 3.1. We used input sets defined as for Figure 3.1 but in this case the value of n grew to 300 instead of 30. In addition, note that the target values are guaranteed to produce at least one valid subset of the input set. As seen in Figure 3.3, the quadratic run time $\binom{n}{2}$ that we predicted for this implementation is convincingly demonstrated.

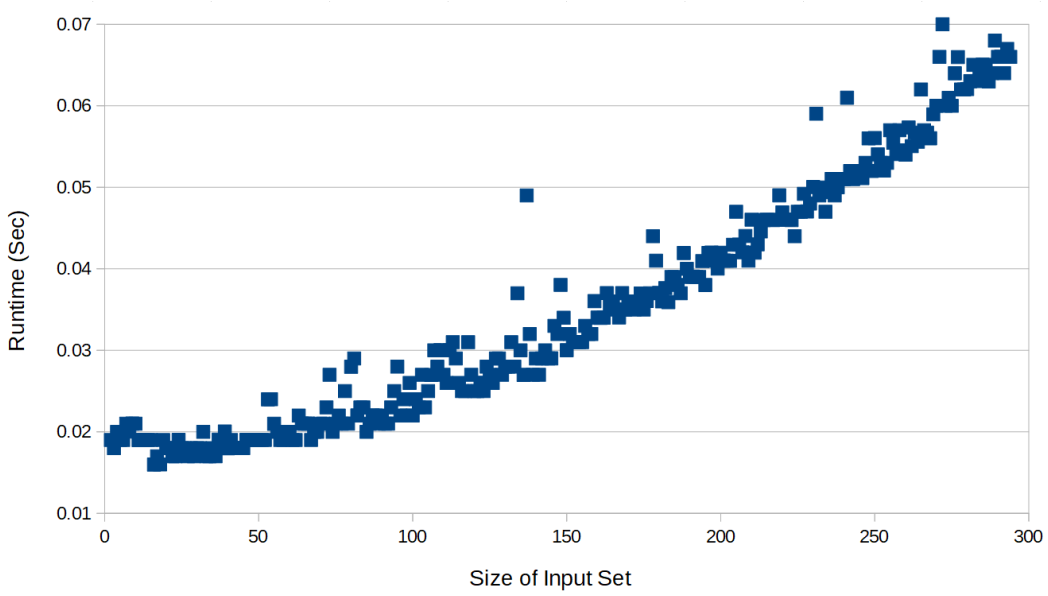

Figure 3.3: The run time of LASSO modified to return the first valid subset with input sets defined by $\{2x10^6 : x \in \mathbb{Z},\ 5 \leq x \leq 4+n\}$ where n ranged from 1 to 290 as seen on the x axis. Target sums were defined as the sum of the input set divided by 2 and rounded to the nearest value guaranteed to return at least one valid subset. The run time is quadratic because of the time taken to produce the Zeroboard, in this implementation $\binom{n}{m}$.

3.3 Unbounded Subset Sum

LASSO was designed specifically to solve the unbounded SSP quickly, so we benchmarked it against the only other implementation we found that solves this problem. To determine if the algorithm provides any significant improvement over the competing approach, we ran the algorithms

on a range of inputs that had an increasing size of input set, n.

3.3.1 Variable Size of Input Set

In Figure 3.4, we see that the run times of the two algorithms grow at an exponential rate, but LASSO performs faster than the optimized dynamic programming algorithm that also returns all valid subsets. Notably, the difference between the run times of these two algorithms does not remain consistent. The run time difference begins at roughly 2 orders of magnitude and then expands to 4 orders when n=18.

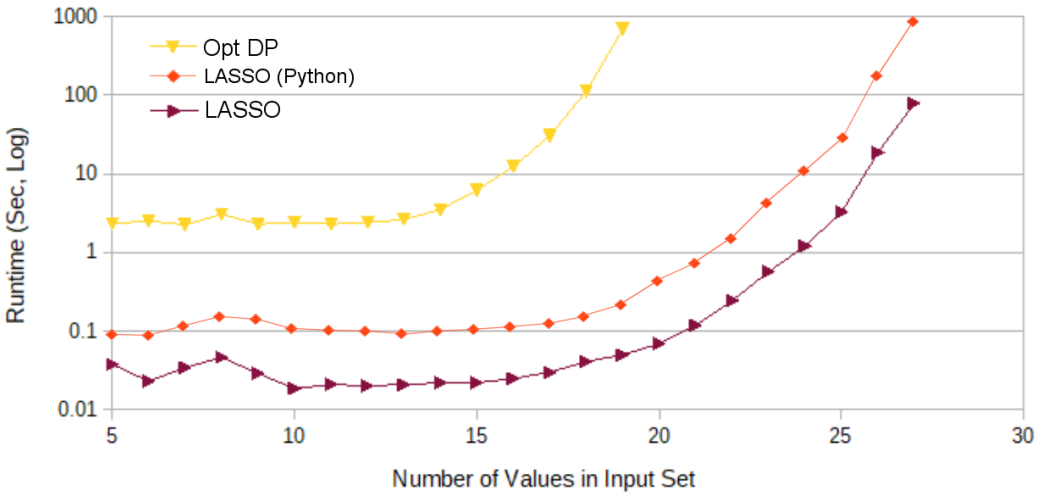

Figure 3.4: The run time of unbounded subset sum algorithms in seconds by increasing size of the input set. The input sets used comprised $\{2x10^6 : x \in \mathbb{Z}, 5 \leq x \leq 4+n\}$ where n ranged from 5 to 30 as seen on the x-axis. The target value was calculated as the sum of the inputs, divided by 2 and rounded up to the nearest subset sum guaranteed to return at least one combination.

Note that these two algorithms are written in different languages, the Optimized DP algorithm is implemented in Python and the reference implementation of LASSO is implemented in C++. To understand the role that implementation language plays in the performance gap, we evaluated an equivalent Python implementation of LASSO. The Python implementation is approximately four times slower than the C++ implementation, and still much faster than the Optimized DP algorithm, indicating that the LASSO algorithm offers significant improvements that are language agnostic.

3.3.2 Variable Target Sum

We also evaluated performance by varying the size of the target value instead of the size of the input set. This is a useful test as many SSP algorithms have a run time that relies on the target value [38], so the run time of an approach like dynamic programming would have degrading run time performance with growing target value and static input set.

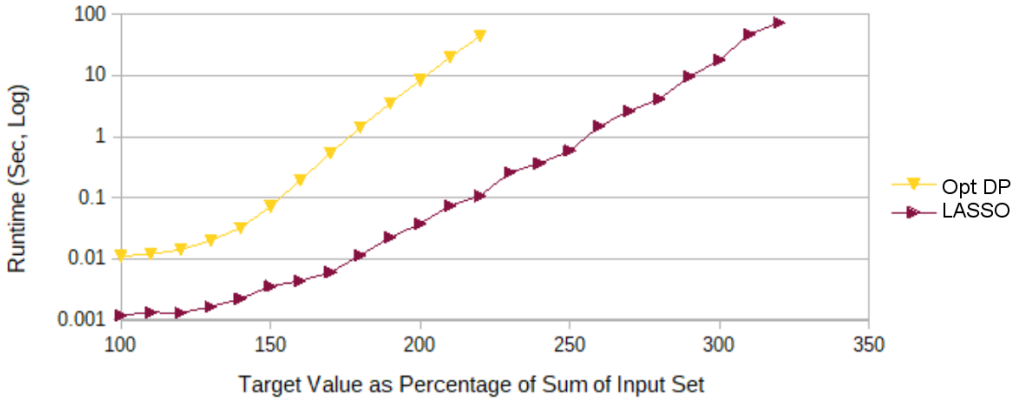

Figure 3.5: A plot of the run times of unbounded subset sum algorithms, the y-axis showing log-scaled run time in seconds and the x-axis showing increasing size of target value as a percentage of the input set sum. The input set was static, comprising $\{2x10^6 : x \in \mathbb{Z}, 30 \leq x \leq 48\}$. Target values were $\{10x10^7 : x \in \mathbb{Z}, 40 \leq x \leq 300\}$, each one guaranteed to return at least one combination.

In a run time benchmark with a static input set and increasing target values, LASSO was not only faster than an optimized dynamic programming approach, but the run time growth was also slower. This is demonstrated in Figure 3.5, which shows the run time of an optimized dynamic programming approach and LASSO. Note that the x-axis shows the target value as a percentage of the sum of the input set, a value that is regarded as an upper bound for the canonical SSP. In this test, the algorithms were tasked with querying for target values that are greater than this upper bound.

3.4 Run Time Growth Bounded by Number of Results

We expect an upper bound on the amount of work done in finding all valid subsets to be asymptotically proportional to the number of valid subsets (subsets that sum to the target value).

We ran a test that measured the run time of the LASSO algorithm as well as the number of valid subsets found with increasing target values over a static input set

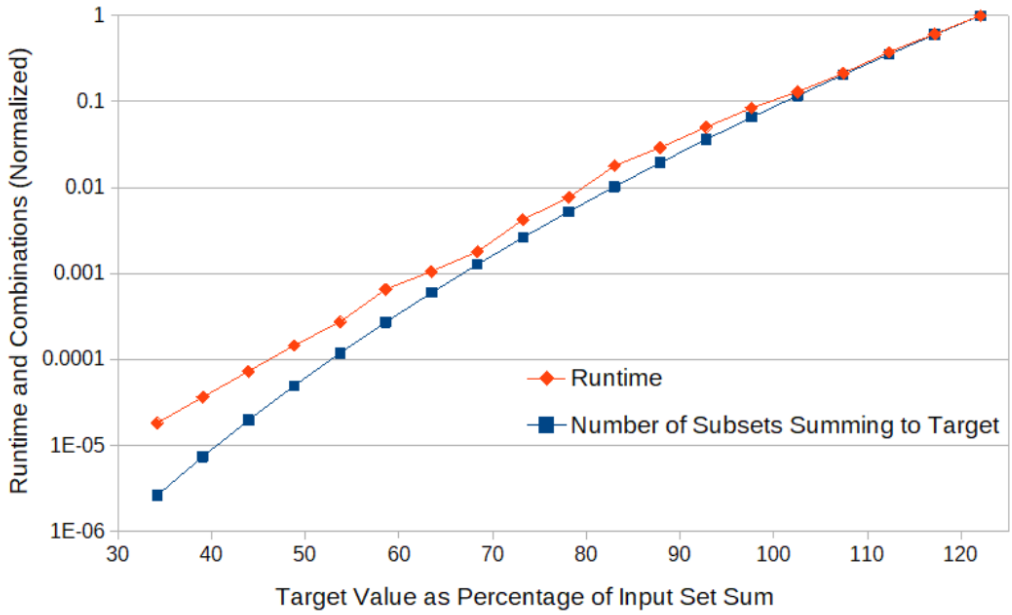

Figure 3.6: A plot with normalized run time and normalized number of subsets summing to the target value (both log-scaled), with the x-axis measuring the target value as a percentage of the sum of the input set. Note that these target values extend into unbounded space beyond the sum of the input set.

Figure 3.6 shows the run time performance of LASSO against the number of valid subsets found by the algorithm. Note that the run time growth rate converges to the growth rate of the number of subsets summing to the target value. This shows that LASSO performs an amount of work that is proportional to the minimum of what is required to list all subsets summing to the target value.

For smaller target values, the normalized run times are larger than the normalized number of subsets returned. This is to be expected with such short run times where initializing the algorithm ends up being a significant portion of the run time. As the target value grows, this initialization time becomes less significant and the normalized run time converges to the normalized number of subsets returned by the algorithm.

3.5 Number of Zeroboard Hash-Table Calls

As another test to measure the amount of work being done by the algorithm, we measured the number of Zeroboard hash-table calls during the second subroutine which searches for valid subsets using the Zeroboard data-structure. In this test we used increasing target values selected to represent every percentile over the possible range in the bounded SSP.

Figure 3.7 confirms that the algorithm performs a minute number of hash-table calls relative to a naive implementation with no bound pruning. Interestingly, the rate of growth undergoes a series of repeated, decreasing steps in the number of calls made. Each of these steps occur as a result of the Zeroboard dimensionality growing, allowing the search-phase to make fewer calls to the Zeroboard due to the larger residual subset size of k - m. Despite this exponential increase, the rate of growth does appear to stabilize unlike the growth observed in Figure 3.4.

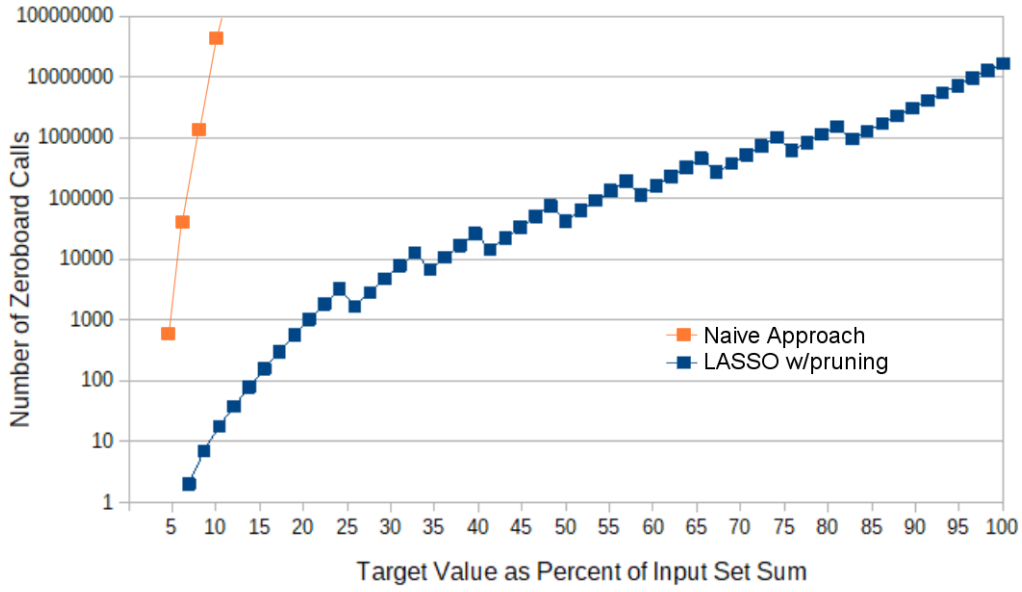

Figure 3.7: A plot of the log-scaled number of Zeroboard hash-table calls by target value as a percentage of the input set sum. Note the comparison between the number of calls made by the naive approach and the pruning approach that LASSO performs. The input values were the set $\{2\text{x}10^6 : \text{x} \in \mathbb{Z},\ 5 \leq \text{x} \leq 24\}$, and the target values were $\{2\text{x}10^7 : \text{x} \in \mathbb{Z},\ 1 \leq \text{x} \leq 29\}$.

3.6 Bounds on run time of the True/False Solution

Having observed our algorithm solving the unbounded SSP in run times that grew exponentially with increasing n, and solving the bounded SSP in run times that were asymptotically constant, we created an additional test to determine if this performance would hold across a range of target values. We tested our algorithm on increasing target values that represent every percentile within the range of target values in the bounded SSP. The input set was static and constructed as described for the last test. This test was designed to convincingly demonstrate our new algorithm's performance with increasing target values over a static input set, where target values have no guarantee on the number of subsets returned by the algorithm.

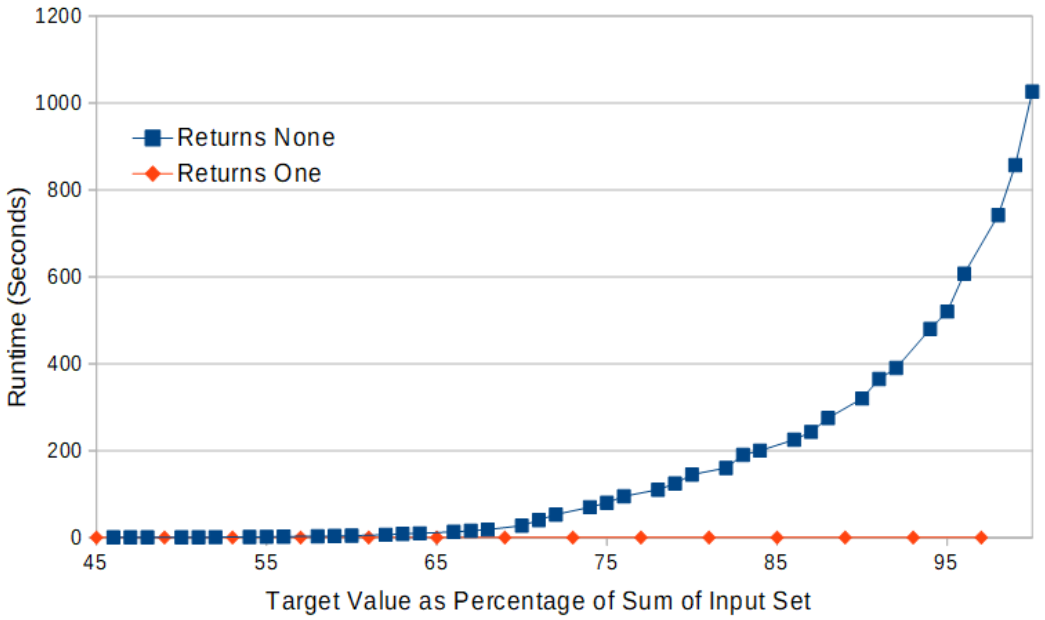

Figure 3.8: A plot of the normalized run time of the modified algorithm that returns only a single valid subset if it exists, showing normalized run time measured over target values generated for every percentile of the input set sum. The input values were the set $\{2x10 : x \in \mathbb{Z}, 1 \leq x \leq 19\}$, and the target values started at 200, incrementing by 55 up to the sum of the input set.

In Figure 3.8 we see our modified algorithm's run time showing two distinct paths that diverge. On the lower bound there is an asymptotically constant run time, while on the upper bound we see what appears to be an exponential growth rate over increasing target values with a static input set.

The upper bound run time path is much like the run time growth rate demonstrated in Figure 3.4. The difference in this case is that these run times resulted from target values that did not return any subsets. For the lower bound, the target values returned a single valid subset. In a case where the input values are randomly selected floating point numbers, the run time would be between these bounds.

3.7 Increasing Order of Magnitude of Input Values

As a final test on variable input values, we measured the run time performance of all competing algorithms over input values that had an increasing order of magnitude. Our algorithm was explicitly designed to be able to handle floating point numbers without any impact on the run time. The main reasons for this were that in real-world applications there are various applications that would require the ability to work with floating point numbers, and in many proposed solutions to the SSP, the inputs are required to be integers. This is largely because of many solutions using a dynamic programming approach where the run time relies on the target value, and we would thus assume that a changing magnitude of the input values would have an impact on run time performance.

As the SSP typically requires integers as inputs, we wanted to determine if converting floating point numbers to integers would be a viable alternative to directly processing floating point values. This test was used to determine the ability of the competing algorithms to solve the SSP over floating point inputs converted to integers at increasing degrees of precision.

Figure 3.9 shows that two of the dynamic programming algorithms show exponentially increasing run time with growing magnitude of input values. Koiliaris and Xu [28] introduced an algorithm that is an optimized approach to dynamic programming and as such their algorithm shows a corresponding run time increase with increasing input value magnitude. Interestingly, the optimized dynamic programming solution does not show a change in run time. The implementation fills a sparse dynamic programming matrix with the same number of non-zero cells, regardless of numeric scale, so that increasing set value magnitude has no impact on run time. The other competing algorithms did not show a run time increase over this test set, demonstrating that in some cases,

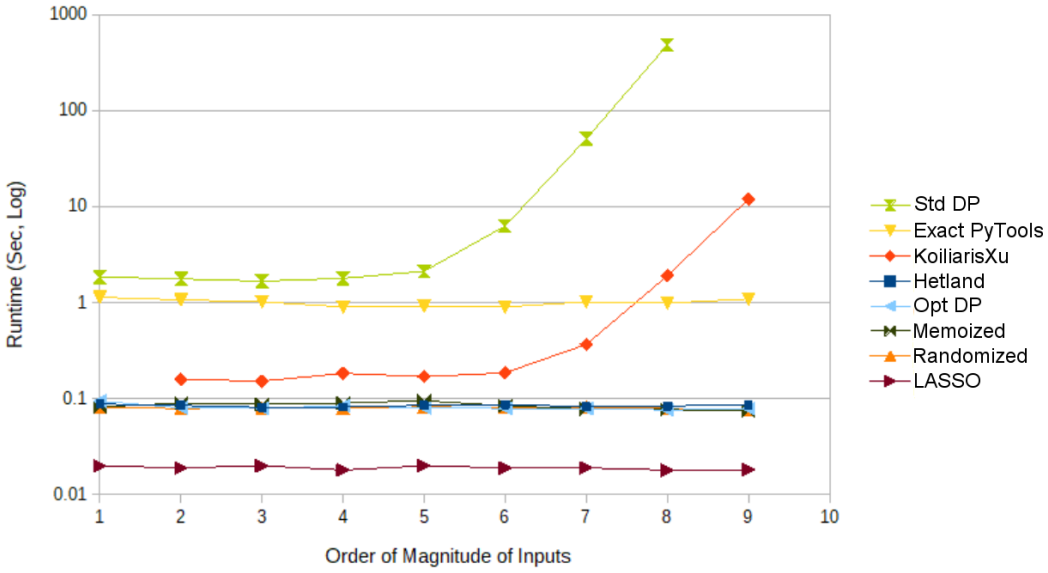

Figure 3.9: A plot of the run times showing log-scaled run time in seconds by increasing order of magnitude of the input values. The input set was defined by $\{2x10p : x \in \mathbb{Z}, 5 \leq x \leq 24\}$, the target value was 3010p, and p comprised integers -1 to 6.

solving the bounded and unbounded SSP over floating point inputs is possible if we convert the inputs to integers.

3.8 Bounded Subset Sum, Variable Target, No Solution

To further explore LASSO's constant run time in solving bounded subset sum, we modified the test of the bounded subset sum illustrated in Figure 3.2. We created target values that were guaranteed to return no subsets by adding 1 to each target from that previous test.

Figure 3.10 shows that LASSO's run time grows exponentially when the target values return no combination. The rate of growth appears to be relatively stable. The reason for this is that it must iterate through all of the valid, unpruned branches of the subset space. That number of branches grows as $\binom{n}{r}$, a function of y the size of the residual subset. In this test, LASSO outperformed the optimized dynamic programming implementation, but it did not perform as well as the other algorithms which mostly had little change in their run time performance. This highlights the fact that LASSO's utility may be greatest in cases where a subset matching the target sum is expected

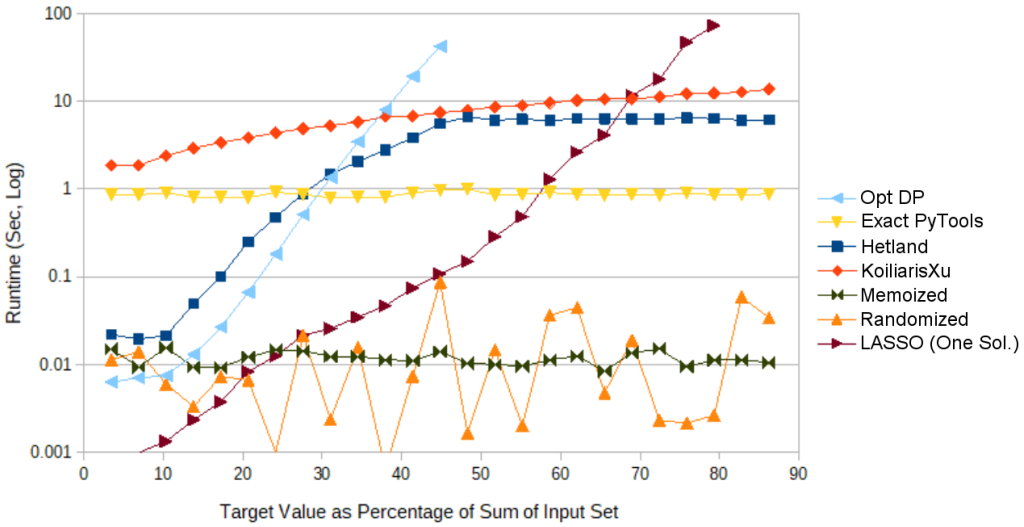

Figure 3.10: A plot of the run times of all competing algorithms showing the log-scaled run time in seconds by increasing size of target value as percentage of input set sum. The input set was static, comprising $\{2x10^6 : x \in \mathbb{Z}, 5 \leq x \leq 24\}$. Target values were $\{(2x10^7)+1 : x \in \mathbb{Z}, 2 \leq x \leq 50\}$, each one guaranteed to return no combination.

to exist, and the true challenge is identifying that subset.

CHAPTER 4 DISCUSSION

In this study we introduced a novel algorithm, LASSO, for solving the unbounded and bounded subset sum problem. The LASSO algorithm was designed to solve the unbounded SSP quickly and to return all subsets summing to a target sum. As speed was the highest priority, we benchmarked the run time performance of LASSO against implementations of some common approaches to the bounded SSP, as well as the only comparable implementation for solving the unbounded SSP that we could find. In solving the bounded SSP, our algorithm had a significantly faster run time than the competing algorithms when the target sum returned at least one subset. When the target returned no subsets, LASSO had a poorer run time growth rate than the competing algorithms solving bounded subset sum. However, for solving the USSP LASSO was still faster than the only comparable algorithm for this problem. We posit that this is due to the novel branch and bound approach that LASSO takes during the search phase.

In our benchmark tests, we demonstrated that the worst-case run time of LASSO is bounded by the number of subsets that sum to the target and is unaffected by the magnitude of the input values and target sum. The other algorithms have run times that depend on the size of the input set and target sum, whereas LASSO depends on the size of the largest possible subset and the size of the subset stored in the Zeroboard. This is relatively unique among algorithms that are accessible for solving this problem. Moreover LASSO caters for floating point input values despite the problem being defined for integers, and it provides the user with the ability to find subset sums that are within epsilon range of the target sum. These unique characteristics set LASSO apart from the competing approaches, particularly where other published algorithms do not have publicly available implementations.

As a drawback, when the target sum returns no subsets LASSO does not perform as well on bounded subset sum as other algorithms designed to solve that problem. However, its performance for solving the unbounded problem for which it was designed offers a significant improvement over an optimized dynamic programming approach. Not only was the run time faster but the growth rate of run time with increasing n was also significantly better. Future work on LASSO can improve performance on its solution to the bounded SSP, reducing the worst-case run time and making it a more adaptable algorithm for a wider range of problems.

BIBLIOGRAPHY

[1] M. Hetland, "Hard problems and (limited) sloppiness," in *Python Algorithms: Mastering Basic Algorithms in the Python Language*. Princeton, United States, 2010, pp. 241–269.

[2] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, R. Miller, J. Thatcher, and J. Bohlinger, Eds. New York, USA: Plenum, 1972, pp. 85–103.

[3] R. E. Bellman, *Dynamic programming*. Princeton, USA: Princeton University Press, 1957.

[4] L. Liu, L. Wang, Z. Cao, and X. Wang, "Algorithms for subset sum problem," *International Journal of Electronics and Information Engineering*, vol. 9, no. 2, pp. 106–114, 2018.

[5] K. Bringmann, "A near-linear pseudopolynomial time algorithm for subset sum," in *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, Barcelona, Spain, January 12-19, 2017*. Philadelphia, USA: Society for Industrial and Applied Mathematics, 2017, pp. 1073–1084.

[6] K. Koiliaris and C. Xu, "Faster pseudopolynomial time algorithms for subset sum," *ACM-Trans. Alg.*, vol. 15, no. 3, pp. 1–40, 2019.

[7] U. Pferschy, "Dynamic programming revisited: Improving knapsack algorithms," *Computing*, vol. 63, no. 4, pp. 419–430, 1999.

[8] D. Pisinger, "Linear time algorithms for knapsack problems with bounded weights," *Journal of Algorithms*, vol. 33, no. 1, pp. 1–14, 1999.

[9] ——, "Dynamic programming on the word ram," *Algorithmica*, vol. 35, no. 2, pp. 128–145, 2003.

[10] J. Cardinal and J. Iacono, "Modular subset sum, dynamic strings, and zero-sum sets," in *Symposium on Simplicity in Algorithms (SOSA)*. Society for Industrial and Applied Mathematics, 2021, pp. 45–46.

[11] K. Axiotis, A. Backurs, K. Bringmann, C. Jin, V. Nakos, C. Tzamos, and H. Wu, "Fast and simple modular subset sum," in *Symposium on Simplicity in Algorithms, Virtual, United States, January 11-12, 2021*. Philadelphia, USA: Society for Industrial and Applied Mathematics, 2021, pp. 57–67.

[12] M. Hasan, S. Hossain, M. Rahman, and M. Rahman, "An optical solution for the subset sum problem," in *Natural Computing*. Tokyo: Springer, 2010, pp. 165–173.

[13] A. Antonopoulos, A. Pagourtzis, S. Petsalakis, and M. Vasilakis, "Faster algorithms for k-subset sum and variations," 2021. [Online]. Available: arXiv preprint arXiv:2112.04244

[14] J. Allcock, Y. Hamoudi, A. Joux, F. Klingelhöfer, and M. Santha, "Classical and quantum dynamic programming for subset-sum and variants," 2021. [Online]. Available: arXiv preprint arXiv:2111.07059

[15] T. Chan and Q. He, "More on change-making and related problems," *Journal of Computer and System Sciences*, vol. 124, pp. 159–169, 2022.

[16] J. Alfonsín, "On variations of the subset sum problem," *Discrete Applied Mathematics*, vol. 81, no. 1-3, pp. 1–7, 1998.

[17] P. Hansen and J. Ryan, "Testing integer knapsacks for feasibility," *European journal of operational research*, no. 3, pp. 578–582, 1996.

[18] O. Muntean and M. Oltean, "Optical solutions for the unbounded subset-sum problem," *Int J Innov Comput Inf Control*, vol. 5, no. 8, pp. 2159–2167, 2009.

[19] D. Wojtczak, "On strong np-completeness of rational problems," in *International Computer Science Symposium in Russia*. Berlin: Springer, 2018, pp. 308–320.

[20] K. Jansen and L. Rohwedder, "On integer programming and convolution," in *10th Innovations in Theoretical Computer Science Conference (ITCS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[21] M. Salimi and H. Mala, "A polynomial-time algorithm for special cases of the unbounded subset-sum problem," 2021. [Online]. Available: arXiv preprint arXiv:2103.09080

[22] K. Klein, "On the fine-grained complexity of the unbounded subsetsum and the frobenius problem," pp. 3567–3582, 2022.

[23] P. Dutta and M. Rajasree, "Efficient reductions and algorithms for variants of subset sum," 2021. [Online]. Available: arXiv preprint arXiv:2112.11020

[24] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*. Berlin: Springer, 2004.

[25] E. Kenar, "Design and implementation of efficient workflows for computational metabolomics," Ph.D. dissertation, Tübingen, Germany, 2015. [Online]. Available: https://core.ac.uk/download/pdf/158280742.pdf

[26] R. Wang and O. Sahin, "The impact of consumer search cost on assortment planning and pricing," *Management Science*, vol. 64, no. 8, pp. 3649–3666, 2018.

[27] K. Jansen, F. Land, and K. Land, "Bounding the running time of algorithms for scheduling and packing problems," *SIAM Journal on Discrete Mathematics*, vol. 30, no. 1, pp. 343–366, 2016.

[28] K. Koiliaris and C. Xu, "Faster pseudopolynomial time algorithms for subset sum," *ACM Transactions on Algorithms (TALG)*, vol. 15, no. 3, pp. 1–20, 2019. [Online]. Available: https://github.com/shtratos/subsetsum

[29] O. Serang, "Junior guide to combinatorial chemistry," 2020. [Online]. Available: https://alpinealgorithmics.bitbucket.io/junior-guide-to-combinatorial-chemistry/index.html

[30] "Boolean solution to subset sum using python powerset. 2011," 2011. [Online]. Available: https://github.com/saltycrane/subset-sum/blob/master/subsetsum/bruteforce.py

[31] "Memoized exact recursive solution to subset sum," 2011. [Online]. Available: https://github.com/saltycrane/subset-sum/blob/master/subsetsum/stackoverflow.py

[32] "Pseudo-polynomial time heuristic solution to subset sum," 2011. [Online]. Available: https://github.com/saltycrane/subset-sum/blob/master/subsetsum/wikipedia.py

[33] M. Hetland, "Python algorithms: Mastering basic algorithms in the python language," pp. 241–269, 2010. [Online]. Available: https://github.com/saltycrane/subset-sum/blob/master/subsetsum/hetland.py.

[34] D. Pisinger, "Where are the hard knapsack problems?" *Computers Operations Research*, vol. 32, no. 9, pp. 2271–2284, 2005.