EECE 7360 Project 5
Subset Sum

Garrett Goode and Daniel Hullihen

April 25, 2017

Chapter 1

Introduction

The subset sum problem (also referred to as the "exact knapsack problem") is defined below.

Let $A = \{a_1, ..., a_n\}$ *represent some set of integers. Given a sum* s*, find a subset* $A' \subset A$ *such that*

$$s = \sum_{i=1}^{m} a'_i, for 1 \leq i \leq m.$$

Where m *is the size of* A'*.*

In other words, if we are given a list of numbers and some target sum, we want to find the numbers in the list that add up to the target sum. Put as a decision problem, the question would be "Is there a subset A' of A where the sum of the elements of A' is s?"

In this project, several approaches for solving instances of the subset sum problem were investigated and compared against each other. Benchmarks were developed based on prior work, and background research was done to highlight subproblems of subset sum. Exhaustive and greedy solvers were implemented and evaluated. Linear programming (LP) and integer-linear programming (ILP) models were developed and evaluated as well. Lastly, local search algorithms were implemented, evaluated and compared to all other previous implementations.

Chapter 2

Benchmarking

Before any benchmark or instances can be generated, it is necessary to understand the prior work done to evaluate implementations of algorithms that solve the subset sum problem. The most popular metric used for determining the difficulty of an instance of the subset sum problem is the "density" of the set in question. The following equation describes the density of a set.

$$d = n/log_2 max(a_i)$$

where n is the size of the set in quation and $max(a_i)$ is the largest element in the set.

Previous work has focused on being able to solve a group of instances that had a some density [4]. For example, Radziszowski and Kreher explored efficiently solving low density subset sum problems ($d < 0.7$) [7]. Schnorr and Euchner pointed out "the hardest subset sum problems turn out to be those that have a density that is slightly larger than 1, i.e. a density about $1 + log_2(n/2))/n$" [8]. On top of that, LaMacchia points out that subset sum problems with a density $d < 0.6463...$ could be solved in polynomial time by a "lattice oracle" [5], showing that there are existing algorithms designed to work efficiently in certain areas of the instance space (in this case, with a sufficiently low density).

What this boils down to is: much research has focused on subset sum problems with densities of and around 1.0. More recently, when evaluating a GPU implementation of the subset-sum problem, Wan et al. crossed vector sizes of 36 to 54 containing random values in the range $[1, 10^8]$, covering a large range of possible densities [2]. Thus, in order to provide a fair comparison with prior work, we need to create instances with densities on and around 1.0. For breadth, we should focus on sufficiently small and large densities as well, and cover different combinations of set length and maximum value that may yield the same density in order to see if those values may have a disproprtionate impact on performance.

2.0.1 Instance Generation

Given the prior work, when it comes to generating benchmarks for evaluating different implementations of the subset sum problem, it is necessary to focus on the density of the instances generated, and the range of densities covered by the overall suite. 152 instances were generated, with an emphasis on instances with densities around 1.0 in order to have a finer granularity of data. To generate each instance, we determined the number of elements needed in the set, as well as the largest value that will exist in the set (determined by the number of bits it represents). For each generated number in the set, b bits of random value 0 or 1 were generated. These bits were then concatenated together to form an interger. Once the list was complete, we randomly chose exactly half of the elements in the list and added them up. This sum is the solution for the instance.

The following table shows the different groups of instances that were generated and some properties that were derived from them. In each row, a range of n and b values were generated. These lists of values were crossed to generate a group.

n (Start:End:Stride)	b (Start:End:Stride)	Count	Min d	Max d	Avg. d
2:10:2	20:26:2	20	0.076	0.5	0.263
16:30:2	15:30:2	64	0.55	2.0	1.09
2:10:2	2:10:2	25	0.2	5.0	1.37
50:100:10	15:30:5	24	1.66	6.66	3.5
90:100:2	2:10:3	18	11.25	50	26.125

Note it is possible to achieve the same density with different values of n and b. In general, we aimed to cover a cross of large and small n with large and small b (e.g. large set with small numbers, small set with large numbers, etc.). Together, the above groups in the table span densities from as low as 0.076 to as high as 50, covering a wider range than what was described in the literature. Given the emphasis on cases around a density of 1.0, we argue this is representative of the instance space and in line with prior work.

Chapter 3

Exhaustive Algorithm

For the first attempt at solving the Subset Sum problem instances we had generated, we chose an exhaustive "brute force" approach. Rather than attempt to target specific subsets of our input set, we cycled iteratively through every possible solution.

Algorithm 1 Exhaustive Algorithm to Solve Subset Sum

```
for subset s in set S do
    sum ← 0
    for element i in s do
        sum+ = element
    end for
    if sum == targetSum||timeOutReached then
        break
    end if
end for
```

Since an element in the input set can be "in" or "out" of the solution set, there are 2^n possible subsets. Therefore, the possible subsets can all be represented by an n-digit binary number. In our implementation of a data representation of a subset sum instance, we represented a solution as an array with the same number of elements as the input set, with each entry set to "INCLUDED" or "EXCLUDED". This representation allowed us to emulate adding 1 to a binary number, starting with every entry set to EXCLUDED and ending when every entry was set to INCLUDED.

Given the above, it is easy to characterize the run time of our exhaustive algorithm. There is an initiation step, and the 2^n possible solutions are looped through in the worst case. Additionally, generating the current sum of the solution set also requires cycling through all n of its elements. This yields the expression below.

$$O(n + n * (2^n)) = O(2^n)$$

As expected, the worst cases of Subset Sum cannot be solved exhaustively in poynomial time.

3.1 Results

The results are presented in the figure below. The vertical axis represents the input size and the horizontal axis the word length of the elements in bits.

Notice that the majority of the instances were solved very quickly. And there were more still that could be solved in less than 1 minute, namely up to an instance size of 30, depending on the max bit-width of the elements in the set. The case with a bit-width of 27, for example, took longer than a minute. The largest instance set that could be solved in 1 minute or less was 94, while the largest size that could be solved in less than 10 minutes was 100. Even then, there were instances between these two cases that could not be solved in 10 minutes.

	2	4	5	6	8	10	15	17	19	20	21	22	23	24	25	26	27	29	30
2	0	0		0	0	0				0		0		0		0			
4	0	0		0	0	0				0		0		0		0			
6	0	0		0	0	0				0		0		0		0			
8	0	0		0	0	0				0		0		0		0			
10	0	0		0	0	0				0		0		0		0			
16							0	0	0		0		0		0		0	0	
18							0	0	0		0		0		0		0	0	
20							0	0	0		0		0		0		0	0	
22							0	0	0		0		0		0		0	0	
24							0	0	1		0		2		0		1	0	
26							0	0	0		1		3		2		6	4	
28							0	0	0		3		6		1		20	27	
30							0	0	1		0		13		34		107	55	
50							402			5					424				
60							600			600					600				
70							600			600					600				
80							600			600					600				
90	600		600		600		600			600					600				29
92	600		600		600														375
94	600		600		600														26
96	600		600		600														600
98	600		600		600														600
100	600		600		600		600			600					600				210

Figure 3.1: Run time for various input size and word length combinations

As expected, the exhaustive "brute force" approach largely failed to solve instances where $n > 50$. This is not universally true however, as the algorithm did manage to solve a few large instances of the largest word size we tested. From this we can conclude that word size does not have a large impact on the run time of the algorithm and that brute force can be viable for even some large instances. However speaking more generally an exhaustive approach is really only viable for smaller instances of subset sum, per both the complexity analysis in the preceeding section and the data presented in the last figure.

3.2 Conclusion

The exhaustive algorithm is a simple algorithn to implement, but can quickly approach its timeout. By definition every single instance will be considered unless a timeout mechanism is used to end the solver early. The worst-case time complexity is not realized in our experiments until the instance size was greater than 60 elements. Even at 50 elements the algorithm began to show signs of struggle. In general, the density of the instance did not impact the run-time of this algorithm; instead, the algorithm was largely impacted by the instance size.

Chapter 4

Survey of Complexity Landscape

In this chapter, we explore the complexity landscape around the subset sum problem, including subproblems to subset sum, as well as problems of which which subset sum is itself a subproblem. Identifying possible subproblems can be helpful when it comes to deciding which kind of algorithm is best used to solve it, as it could possibly exploit some unique property to that subproblem.

4.1 Natural Subproblems of Subset Sum

The subset sum problem can be solved in pseudo-polynomial time, which means there are subproblems to subset sum that can be solved in polynomial time, but the "hardest" subproblem in subset sum is nevertheless NP-Complete. Due to this property, it is possible to detect some subproblems to subset sum and apply a known algorithm that solves the problem in polynomial time. Some algorithms used for solving subset sum use a divide-and-conquer approach and identify these subproblems. Subproblems to subset sum, for example, can have all numbers that have some special property, or the set of numbers itself has some property that can be exploited. This section explores some of the natural subproblems to subset sum.

4.1.1 Sets with a low density

One metric for describing an instance of subset sum is the density of the set. The following equation describes the density of a set.

$$d = n/log_2 max(a_i)$$

where n is the size of the set in quation and $max(a_i)$ is the largest element in the set. Lagarias and Odlyzko found that, for sets that have a density less than 0.645, their "Algorithm SV" can solve the instance in polynomial time [4]. One of the steps in their algorithm involves taking set and perform a transformation in order to apply an algorithm developed by Lenstra, Lenstra, and Lovasz, which has a time complexity of $O(n^{12} + n^9(log|f|^3))$ [6]. The transformed problem is one where a short vector e must be found within an integer lattice L = L(a,M)

4.1.2 Sets where all the numbers are coprime to m

This approach (as well as the prior approach) focus on sets that are finite cyclic groups, defined as $\mathbb{Z}_m = 0, 1, ..., m-1$ of order m. Koiliaris and Xu argue that, given a set $S \subseteq U(\mathbb{Z})_m$, which describes a set of integers that are coprime to some integer m, finding the set of all subset sums can be donyge with a time complexity of $O(min(\sqrt{n}m, m^{5/4})logmlogn)$ [3]. Note that an integer x is coprime to m if the greatest common denomintor of the two values is 1. As Koiliaris explains, the method behind the speed-up with this approach is the ability to partition the set into different subsets such that "every such subset is contained in an arithmetic progression of the form x, 2x, ..., lx". With this, one can calculate the subset sums very quickly by scaling l accordingly. These sums can then be added together. This is only doable if m is a prime number, or if all the numbers are relative prime to m.

4.1.3 Set is a subset of Z_m

Koiliaris and Xu are able to take the previous case and take it a step further, developing an algorithm that finds all of the subsets of a set S in with a time complexity of $O(min(\sqrt{n}m, m^{5/4})log^2m)$ [3]. This is for sets $S \subseteq \mathbb{Z}_m$, which is slightly different from the previous case in that the elements of the set no longer have to be coprime to some integer m. Their algorithm involves recursively computing a partial subset sum as they work across subsets of the original set.

4.1.4 Set size $m \geq l^{1/\alpha}$ and elements are distinct

Chaimovich, Freiman and Galil found that, if they took the subset sum problem and restricted it such that $maxa_i|a\epsilon A \leq l \leq m^\alpha)$ (where l is some bound and m is the number of variables in the set), then the instance could be solved with a time complexity of $O(lm^2)$ [1]. One unique feature of their algorithm is that supports large sets (previous algorithms could only handle moderately large sets). The main approach in their algorithm is, assuming A^* is defined as $S_b|B \subseteq A$, and S_b is defined as $\sum_{a_i \epsilon B} a_i$, characterize A^* as "a small collection of arithmetic progressions."

4.2 Problems of which Subset Sum is a Natural Subproblem

We were unable to identify any problems of which Subset sum is a natural subproblem. However rather than ignore this area of investigation entirely, we chose to explore similar problems that are close neighbors via transformation in the landscape of common NP-complete problems.

4.2.1 Satisfiability

While Satisfiability (SAT) is a few transformations away from Subset Sum on the complexity landscape we have presented, establishing this link between subset sum and SAT is crucial as almost all NP completeness proofs are derived from Cook's original proof for SAT and some transformations. The problem is defined below.

Given a set of U variables, collection of C clauses over U, is there a satisfying truth assignment for C?

As proven by Cook in the aforementioned paper, SAT is NP-complete.

4.2.2 3-Satisfiability

Transformed from SAT, 3-Satisfiability (3SAT) is a special case of SAT where each logical clause can have only exactly three variables. The problem is formally defined as follows.

Given a set of U variables, collection of C clauses over U such that $|c| = 3$*, is there a satisfying truth assignment for C?*

Despite this restriction on the domain of problem instances when compared to SAT, 3SAT remains NP-complete.

4.2.3 3-Dimensional Matching

Transformed from 3SAT, the 3-dimensional matching problem deals with finding a matching in a three-dimensional graph. The formal definition is given below.

Given a set $M \subset WxXxY$*, where W, X, and Y are disjoint sets with the same number of elements q, does M contain a matching* $M' \subset M$ *such that* $|M'| = q$ *and no two elements are the same?*

Like the previous problems discussed in this section, 3DM remaings NP-complete in all but a few ideal scenarios. Interestingly enough, 2-Dimensional Matching (2DM) which is not discussed at length in this seciton, is tractable.

4.2.4 Partition

The Partition problem is defined below.

Let $A = \{a_1, ..., a_n\}$ represent some set of positive integers. Does there exist a subset $A' \subset A$ such that

$$\sum_{a \epsilon A'} s(a) = \sum_{a \epsilon A - A'} s(a)?$$

In other words, is there a subset of the given set with sum equal to the sum of the members of the original set not included in the subset?

Once again, the partition problem is known to be NP-Complete. Of all of the problems discussed in this section, the partition problem is likely the closest to the Subset Sum problem. However it is not a natural subproblem of Subset Sum as no target is given as input. Instead, the target is implied as the half of the sum of the input set.

One can see clearly how simple a transformation from an instance of Partition to and instance of Subset Sum would be however, simply by summing the input set and setting the target total to half of the calculated sum.

4.2.5 Knapsack

The Knapsack problem is also a transformation from the Partition problem (like Subset Sum). As one might expect, the problems are therefore very similar in nature. The Knapsack problem is defined as follows.

Given a finite set U, with each $u \epsilon U$ having an associated size s(u) and value v(u), is there a subset $U' \subset U$ such that the sum of the s(u') for all $u' \epsilon U'$ is less than a given integer K while the sum of v(u') for all $u' \epsilon U'$ is greater than a given integer H?

Like all of the other problems in this section, knapsack is NP-complete. There are variations of the the Knapsack problem that are tractable, but they are not discussed in this paper as they are less related to the Subset Sum problem in question. Subset Sum itself is sometimes refered to as the exact knapsack problem.

4.3 Summary

The chart in Figure 1 on the following page maps out all problems discussed in the preceding sections. Note that all subproblems are tractable, while all problems related by transformation are believed to be intractable.

4.4 Conclusion

Overall the investigation into the subproblems and closely related NPC problems was very helpful in getting a clearer vision of the nature of the Subset Sum problem. There was some some difficulty identifying problems that subset sum was a natural subproblem of, but in the end that seemed to be apporpriate as the subset sum problem itself is very general. It is quite simple to see how when bounds and restrictions are introduced to the origional subset sum problem statement we arrive at various other important or interesting subproblems, but it is difficult to relax the input domain of a problem that is already so broad and nonspecific in nature.

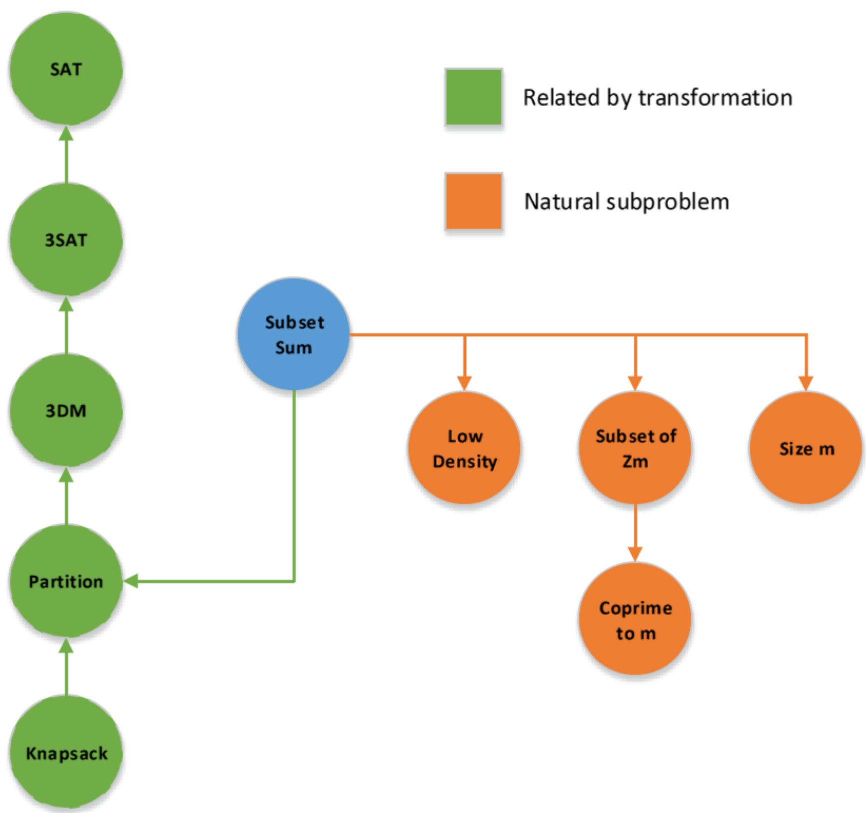

Figure 4.1: Complexity landscape for Subset Sum

Chapter 5

Greedy Algorithm

In this chapter, we review an implementation of a greedy algorithm for the subset sum problem. This implementation was run against a suite of instances, and the performance of this algorithm is described.

5.1 Greedy Algorithm Implementation

The implementation of the greedy algorithm that was used for this project is relatively simple. Below is the psuedocode for the algorithm.

Algorithm 2 Greedy Algorithm to Solve Subset Sum

```
runningSum ← 0
subset ← [] {Initialize subset to an empty list}
for integer i in set S do
  if runningSum + i ≤ targetSum then
    runningSum ← runningSum + i
    append i to subset
  end if
end for
if runningSum == targetSum then
  return subset
else
  return NULL
end if
```

We walk through each element of the provided instance, adding numbers so long as the running sum has not exceeded the target sum. At the end, we check to see if the running sum matches the target sum; if it does, we have successfully solved the instance and return the subset of integers. Otherwise we have not solved the instance and return NULL.

One characteristic of a greedy algorithm is that, once we decide to include an element to the sum, we never undo the decision. In the case of the subset sum problem, this can easily cause the algorithm to not find the correct list of elements to use for the sum since not all combinations of integers in a list are necessarily going to add up to the target sum. Because of this, a margin of error is introduced where the algorithm may fall short of the target sum. The upside of this algorithm is the time complexity improvement: the algorithm visits each element of the set only once, giving the algorithm a tightly bound time complxity of O(n). This algorithm may be useful for applications that only need a subset that is within some percent of the target sum, especially given the improved time complexity over the brute-force solution.

5.2 Trivial Case Where the Greedy Algorithm Fails

Due to the nature of the greedy algorithm explained in the previous section, we can identify a trivial case where the algorithm will fail. Take for example the set of integers S below where the target sum is 150.

$$S = \{49, 100, 50\}$$

The greedy algorithm will start from the first element and work its way through the list, adding numbers so long as the running sum does not go beyond the target sum of 150. In this case, the algorithm will accept the integers 49 and 100, resulting in a sum of 149. When it encounters 50, it decides not to add this number since it would bring the running sum over the target sum. The algorithm has then completed processing the set, but it has failed to find a solution depite the fact one actually exists.

5.3 Results

The results are presented in Figure 6.4 below. The vertical axis represents the input size and the horizontal axis the word length of the elements in bits.

number of elements \ bit width of largest value in set	2	4	5	6	8	10	15	17	19	20	21	22	23	24	25	26	27	29	30
2	1.000	0.167		1.000	1.000	1.000				1.000		1.000		1.000		1.000			
4	1.000	0.941		1.000	1.000	0.649				0.708		0.561		1.000		0.998			
6	1.000	0.750		0.443	0.906	0.855				0.851		0.828		0.895		0.868			
8	1.000	0.917		0.809	0.908	0.744				0.984		0.962		0.947		0.802			
10	1.000	1.000		0.973	0.992	0.994				0.956		0.969		0.962		0.988			
16							0.979	0.995	0.983		0.985		0.950		0.993		0.989	0.933	
18							0.971	0.992	0.991		0.992		0.970		0.993		0.989	0.991	
20							0.996	0.977	1.000		0.969		0.995		0.988		0.987	1.000	
22							0.999	1.000	0.998		0.995		0.971		0.989		0.978	0.997	
24							0.979	0.984	0.992		0.999		0.989		0.990		0.991	0.997	
26							0.979	0.998	0.969		0.994		0.989		0.997		0.988	0.992	
28							0.986	0.997	1.000		0.986		0.996		0.997		0.993	0.991	
30							0.996	0.985	0.995		0.982		0.999		0.998		0.975	0.994	
50							0.998			1.000					0.999				0.997
60							0.994			0.997					1.000				0.603
70							0.999			1.000					0.998				0.997
80							0.998			0.999					0.999				0.984
90	1.000		0.995		0.999		1.000			0.998					0.997				0.995
92	1.000		1.000		0.999														
94	1.000		1.000		1.000														
96	1.000		1.000		1.000														
98	1.000		1.000		0.999														
100	1.000		1.000		0.999		0.997			0.999					1.000				0.980

Figure 5.1: Margin of error for various input size and word length combinations for the greedy algorithm

Each cell in the above figure is color-coded based on the margin of error the greedy algorithm had for the given instance. A value of 1 means the algorithm successfully found a subset whose sum of elements matched the target sum. Varying yellow, orange and red elements are cases where the algorithm was more progressively off from the target sum, with red being the most off/greatest error.

Of the 151 instances that were tested, the greedy algorithm was able to correctly solve 28 of them, yielding an 18.5% success rate. Compared to the exhaustive algorithm, which had a 76% success rate, the greedy algorithm is much less successful despite it's lower time complexity. As a reminder, Figure 6.5 shows the run-times for the exhaustive algorithm.

Looking at the results described in Figure 6.4, the greedy algorithm was successful when there were sufficiently few numbers in the set, as well large sets with relatively small numbers. For example, the majority of the instances with 90+ elements amd a max bit width ¡ 10 passed. This may be due to the fact that, given enough sufficiently small numbers, the algorithm can work at a small enough granularity and have a better chance to come across a total set of numbers that can add up to the target. However, as the size of the max value increases, the margin begins to decrease; the algorithm is less likely to hit the target sum.

The instance where the greedy algorithm had the worst margin was the one with 2 elements and a max bit width of 4. This is most likely due to chance. With fewer numbers, there are fewer chances to find other

	2	4	5	6	8	10	15	17	19	20	21	22	23	24	25	26	27	29	30
2	0	0		0	0	0				0		0		0		0			
4	0	0		0	0	0				0		0		0		0			
6	0	0		0	0	0				0		0		0		0			
8	0	0		0	0	0				0		0		0		0			
10	0	0		0	0	0				0		0		0		0			
16							0	0	0		0		0		0		0	0	
18							0	0	0		0		0		0		0	0	
20							0	0	0		0		0		0		0	0	
22							0	0	0		0		0		0		0	0	
24							0	0	1		0		2		0		1	0	
26							0	0	0		1		3		2		6	4	
28							0	0	0		3		6		1		20	27	
30							0	0	1		0		13		34		107	55	
50							402			5					424				
60							600			600					600				
70							600			600					600				
80							600			600					600				
90	600		600		600		600			600					600				29
92	600		600		600														375
94	600		600		600														26
96	600		600		600														600
98	600		600		600														600
100	600		600		600		600			600					600				210

Figure 5.2: Run time for various input size and word length combinations for the exhaustive algorithm

numbers that can add up to the target sum. It is also a function of the distribution of numbers in the set. If there is a large distribution of integers and the target sum actually needs the larger value, but the larger value is at the end of the set, then the algorithm is more likely to not include it and end up with a large margin of error. One way to possibly correct for this case is by sorting the integers from largest to smallest before processing them.

There appears to be a sufficiently large area where the margin of error is within a few percent of the target sum, but the algorithm nevertheless fails. This appears to happen with medium-sized sets (i.e. sets with more than just a couple elements, but do not have a large number either). This is namely from set sizes of about 6 to 50. The greedy algorithm is unable to correctly solve the majority of these instances. This may be due to the fact that there are enough integers where the algorithm can accidentally add one that will actually never allow the running sum to equal the target sum. Had there been more integers, there would be more opportunities for the algorithm to find some other integer that could still allow the algorithm to achieve the target sum.

In general, the amount of margin exhibited by the greedy algorithm seems to come down to how many subsets actually do exist within the set that satisfy the target sum, which depends on factors such as the distribution of integers values in the set and whether a value repeats itself. For example, for sets with several large and small numbers that are similar to each other, there may be more subsets that satisfy a given target sum compared to a set with a more even distribution of integers. Granted, this also depends on the target sum itself. But if there are many subsets that satisfy the target sum, then the greedy algorithm has a better chance of solving a given instance.

5.4 Conclusion

The greedy algorithm appears to be generally effective on instances with a relatively low density. However, it is also at the mercy of not including some element in its set that will prevent the solver from ever being able to reach the target sum, leaving smaller instances still vulnerable to some margin of error, regardless of the instance size itself. The margin of error decreases with suitably larger instances since there are more opportunities to choose to include a correct element in the final set that will bring the running total close to (if not exactly on) the target sum.

Chapter 6

ILP and LP Modeling

In this chapter, we review LP and ILP techniques. An ILP model was developed. The results of running our ILP model against are suite of instances is described.

6.1 ILP Formulation

The implementation of the ILP formulation we utilized is given by the following AMPL model pseudo-code.

Algorithm 3 AMPL Model for Subset Sum

$values \leftarrow$ {input set}
$X \leftarrow$ [] {empty binary array}

Ensure:
values[i] * X[i] is maximal

Require:
sum(values[i] * X[i]) = target

First parameters are created to store the input data for the problem instance, the set of integers and the target. A second array of binary values is created to track which members of the input set will form the subset that represents the solution.

We elected to maximize the sum of the subset as our objective function, so that even in a situation with no solution we would still be able to achieve the best possible value. This was also the way our greedy algorithm behaved, so it would make it easier to compare the results.

Lastly, our sole condition guaranteed that the sum of the subset would have to be equal to the target. This way, optimal solutions would always stop at the target, despite trying to "maximize" the sum.

6.2 LP Lower Bound

Unsurprisingly, the LP lower bounds we calculated essentially match the ILP results discussed later in the Results portion of the report. This holds with our expectation, as since subset sum seeks an exact value. One notable difference was the LP models ran much faster than their ILP counterparts, as evidenced by the figures below.

6.3 Results

The results are presented in Figure 6.3 below. The vertical axis represents the input size and the horizontal axis the word length of the elements in bits.

	Bit width																		
Input size	2	4	5	6	8	10	15	17	19	20	21	22	23	24	25	26	27	29	30
2	0.003733	0.004053		0	0.003733	0.003309				0.003551		0.003526		0.00353		0.003753			
4	0.003564	0.003769		0.00361	0.00348	0.003825				0.003753		0.003391		0.003739		0.003628			
6	0.003623	0.003849		0.003805	0.00362	0.003805				0.003529		0.003689		0.003812		0.003554			
8	0.003557	0.003534		0.003704	0.003514	0.004159				0.003676		0.003446		0.003697		0.003459			
10	0.00355	0.003653		0.003658	0.003729	0.003533				0.003616		0.003371		0.003875		0.00382			
16							0.004458	0.003615	0.003728		0.003557		0.004179		0.004216		0.004369	0.003858	
18							0.003701	0.003731	0.003865		0.004061		0.003833		0.00389		0.003782	0.003697	
20							0.003509	0.003844	0.004084		0.004032		0.004056		0.003536		0.00369	0.003629	
22							0.003764	0.003628	0.003923		0.003876		0.003494		0.003605		0.003575	0.00353	
24							0.00363	0.004002	0.003727		0.003491		0.003839		0.003619		0.003323	0.003892	
26							0.003624	0.004089	0.003584		0.003546		0.003628		0.004343		0.003866	0.003517	
28							0.004157	0.003854	0.003678		0.003664		0.004009		0.003784		0.003897	0.00469	
30							0.00385	0.003837	0.003755		0.003627		0.003675		0.003707		0.00366	0.003505	
50							0.003758			0.003782					0.003616				0.003795
60							0.00382			0.003702					0.003567				0.003888
70							0.003873			0.004088					0.003538				0.003664
80							0.003887			0.003989					0.003814				0.003781
90	0.003442		0.003795		0.003616		0.003744			0.003724					0.003901				0.003966
92	0.003598		0.003867		0.004026														
94	0.003465		0.003811		0.003808														
96	0.00393		0.003564		0.003666														
98	0.003614		0.004063		0.003872														
100	0.003707		0.003952		0.00365		0.003749			0.003737					0.003584				0.003859

Figure 6.1: Runtimes for LP subset sum models

	Bit width																		
Input size	2	4	5	6	8	10	15	17	19	20	21	22	23	24	25	26	27	29	30
2	0	0		0	0	0				0		0		0		0			
4	0.122001	0		0.511278	0.426933	0.436103				0.39752		0.378656		0.156528		0			
6	0.573227	0.344486		0.344632	0.262518	0.347527				0.369383		0.292613		0.046806		0.01049			
8	0.698173	0.787634		0.71189	0.850388	0.40748				0.63521		1.02536		0.138974		0.017803			
10	0.784521	0.582892		0.845672	0.70142	0.670944				45.1428		0.441186		4.62149		0.010194			
16							58.8177	21.4316	57.7484		22.722		46.6254		55.9903		61.7264	0.03662	
18							40.8419	52.244	67.5898		47.3129		54.499		41.1921		52.4214	45.8529	
20							17.8074	44.5947	50.2626		61.144		0.699012		51.3472		51.4473	64.3695	
22							0.72373	59.4118	47.5246		44.5033		49.9935		133.112		42.8144	94.0554	
24							58.782	52.9151	98.5954		49.6893		155.24		96.7399		146.057	57.1084	
26							46.4253	48.3465	96.8108		136.58		140.388		153.763		99.861	211.802	
28							0.941136	44.3117	49.0089		193.401		50.2341		216.754		185.89	198.987	
30							48.739	17.1562	103.806		145.232		125.426		120.239		132.77	117.555	
50							44.3881			96.4059					114.873				80.5586
60							39.9474			0.84924					265.748				150.352
70							1.00856			96.1769					39.7205				253.682
80							0.747665			157.096					225.306				143.687
90	0.976784		0.734031		0.684358		0.745812			105.324					38.6097				31.3719
92	0.670504		0.557017		0.826344														
94	0.850475		0.860876		0.809628														
96	0.729519		0.682609		0.676373														
98	0.341049		0.722678		0.924143														
100	0.780185		0.529281		0.654773		14.0667			124.45					117.322				232.672

Figure 6.2: Runtimes for ILP subset sum models

Each cell in the above figure is color-coded based on the margin of error the ILP model had in its final solution. This figure is presented mostly to contrast with the other figures in the section, as it is clear the ILP model was able to produce an accurate solution in every instance. Comparing this result with the same figure for the greedy algorithm below, it is abundantly clear that the ILP approach is vastly superior in terms of accuracy. Note that the table in Figure 6.4 uses an accuracy rating that is the inverse of the ILP table - a rating of 1 is a perfect solution and 0 is instead the worst.

Of the 151 instances that were tested, the greedy algorithm was able to correctly solve 28 of them, yielding an 18.5% success rate. Compared to the ILP formulation, which had a 100% success rate, the greedy algorithm is much less successful despite it's lower time complexity. As a reminder, Figure 6.5 shows the run-times for the exhaustive algorithm.

In general, the amount of margin exhibited by the greedy algorithm seems to come down to how many subsets actually do exist within the set that satisfy the target sum, which depends on factors such as the distribution of integers values in the set and whether a value repeats itself. For example, for sets with several large and small numbers that are similar to each other, there may be more subsets that satisfy a given target sum compared to a set with a more even distribution of integers. Granted, this also depends on the target sum itself. But if there are many subsets that satisfy the target sum, then the greedy algorithm has a better chance of solving a given instance.

When compared to the exhaustive algorithm, the accuracy advantage of the ILP approach is similar to that of the greedy algorithm. However, when the timing results of the ILP algorithm in Figure 6.2 are compared to that of the exhaustive algorithm, it is actually the much more primitive exhaustive algorithm that has the advantage in solving smaller or less complex instances. This result was surprising, as all other findings seemed to indicate that ILP was more or less a 'magic bullet' for solving our subset sum instances.

Bit width

Input size	2	4	5	6	8	10	15	17	19	20	21	22	23	24	25	26	27	29	30
2	0	0		0	0	0				0		0		0		0			
4	0	0		0	0	0				0		0		0		0			
6	0	0		0	0	0				0		0		0		0			
8	0	0		0	0	0				0		0		0		0			
10	0	0		0	0	0				0		0		0		0			
16							0	0	0		0		0		0		0	0	
18							0	0	0		0		0		0		0	0	
20							0	0	0		0		0		0		0	0	
22							0	0	0		0		0		0		0	0	
24							0	0	0		0		0		0		0	0	
26							0	0	0		0		0		0		0	0	
28							0	0	0		0		0		0		0	0	
30							0	0	0		0		0		0		0	0	
50							0			0					0				0
60							0			0					0				0
70							0			0					0				0
80							0			0					0				0
90	0		0		0		0			0					0				0
92	0		0		0														
94	0		0		0														
96	0		0		0														
98	0		0		0														
100	0		0		0		0			0					0				0

Figure 6.3: Margin of error for various input size and word length combinations for ILP

bit width of largest value in set

number of elements	2	4	5	6	8	10	15	17	19	20	21	22	23	24	25	26	27	29	30
2	1.000	0.167		1.000	1.000	1.000				1.000		1.000		1.000		1.000			
4	1.000	0.941		1.000	1.000	0.649				0.708		0.561		1.000		0.998			
6	1.000	0.750		0.443	0.906	0.855				0.851		0.828		0.895		0.868			
8	1.000	0.917		0.809	0.908	0.744				0.984		0.962		0.947		0.802			
10	1.000	1.000		0.973	0.992	0.994				0.956		0.969		0.962		0.988			
16							0.979	0.995	0.983		0.985		0.950		0.993		0.989	0.933	
18							0.971	0.992	0.991		0.992		0.970		0.993		0.989	0.991	
20							0.996	0.977	1.000		0.969		0.995		0.988		0.987	1.000	
22							0.999	1.000	0.998		0.995		0.971		0.989		0.978	0.997	
24							0.979	0.984	0.992		0.999		0.989		0.990		0.991	0.997	
26							0.979	0.998	0.969		0.994		0.989		0.997		0.988	0.992	
28							0.986	0.997	1.000		0.986		0.996		0.997		0.993	0.991	
30							0.996	0.985	0.995		0.982		0.999		0.998		0.975	0.994	
50							0.998			1.000					0.999				0.997
60							0.994			0.997					1.000				0.603
70							0.999			1.000					0.998				0.997
80							0.998			0.999					0.999				0.984
90	1.000		0.995		0.999		1.000			0.998					0.997				0.995
92	1.000		1.000		0.999														
94	1.000		1.000		1.000														
96	1.000		1.000		1.000														
98	1.000		1.000		0.999														
100	1.000		1.000		0.999		0.997			0.999					1.000				0.980

Figure 6.4: Margin of error for various input size and word length combinations for the greedy algorithm

6.4 Conclusion

Overall it seems that ILP is a very strong for solving subset sum instances. More specifically, ILP excels at solving more complex instances that greedy or exhaustive approaches cannot solve in a reasonable amount of time. However, due to its sophisticated approach to generating a solution, it seems to be excessively costly for solving simpler instances. In these instances, one can reach an optimal solution much faster by using a simpler approach. This illustrates that not only is there no perfect method to solve any optimization problem, but it seems that often there is no perfect method for solving every instance of a single optimization problem.

	2	4	5	6	8	10	15	17	19	20	21	22	23	24	25	26	27	29	30
2	0	0		0	0	0				0		0		0		0			
4	0	0		0	0	0				0		0		0		0			
6	0	0		0	0	0				0		0		0		0			
8	0	0		0	0	0				0		0		0		0			
10	0	0		0	0	0				0		0		0		0			
16							0	0	0		0		0		0		0	0	
18							0	0	0		0		0		0		0	0	
20							0	0	0		0		0		0		0	0	
22							0	0	0		0		0		0		0	0	
24							0	0	1		0		2		0		1	0	
26							0	0	0		1		3		2		6	4	
28							0	0	0		3		6		1		20	27	
30							0	0	1		0		13		34		107	55	
50							402			5					424				
60							600			600					600				
70							600			600					600				
80							600			600					600				
90	600		600		600		600			600					600				29
92	600		600		600														375
94	600		600		600														26
96	600		600		600														600
98	600		600		600														600
100	600		600		600		600			600					600				210

Figure 6.5: Run time for various input size and word length combinations for the exhaustive algorithm

Chapter 7

Local Search Algorithms

In this chapter, we review some of the local search algorithm implementations. One benefit of local search is that we can choose a neighborhood of possible solutions to explore, and continue exploring until we either find the best solution in the neighborhood, or we run out of time.

7.1 Finding an Initial Solution

One requirement for local search algorithms is that they need an initial solution to start from, which can be generated by any mean. For this project, we chose to generate initial solutions through the (1) greedy and (2) random algorithm implementations.

7.2 Searching a Neighborhood

The local search algorithms require a neighborhood to be defined as well. We decided to define a neighborhood via 1-OPT. In this implementation, given an initial solution, we will choose to remove one element in the set and add another element. In the case of following a steepest descent behavior, we repeat this until we either ran out of time, or until we reached a case where swapping out any element in the current solution would only increase the delta between the current sum and the target sum (the goal in this case is to always try to get closer to the target sum). In that case we have found a locally optimal solution and stop. Thus, starting with the initial solution found via the greedy and random algorithms, we take 1-OPT and continue to drill down to the nearest locally optimal solution.

As a point of comparison, we also implemented the Tabu search algorithm. In this case, we maintain a history of previous solutions considered, avoiding those solutions when looking for new ones. The general benefit of this approach is, given a sufficiently large window of history, we can escape "valleys" where locally optimal solutions exist, and possibly find an even better solution. In this case, we kept a history of the past 100 solutions that were considered.

7.3 Results

In general, across all of the algorithms, all of the instances were solved well below the provided 1-minute and 5-minute time limit, so there is no plot that shows the run-time; the runtime of all of the instances would be reported as 0. This is most likely due to the size of the neighborhood provided by 1-OPT; it is small/steep enough to quickly drill down and find a locally optimal solution. Note that this means that all iterations were worked on in the neighborhood for all algorithms.

7.3.1 Margin of Error

Figures 7.1, 7.2, and 7.3 show the margin of error to the target sum each algorithm had for the solution it came up with. On average, starting with the solution generated by the greedy algorithm led to a result that was 0.994 of the target sum; for the random algorithm it was 0.965. So on average, albeit by not very much, using an initial solution from the greedy algorithm got us into a neighborhood with a better locally optimal solution.

	Max bit-width																		
Number of elements	2	4	5	6	8	10	15	17	19	20	21	22	23	24	25	26	27	29	30
2	1.000	1.000		1.000	1.000	1.000				1.000		1.000		1.000		1.000			
4	1.000	1.000		1.000	1.000	1.000				0.708		1.000		1.000		0.998			
6	1.000	1.000		1.000	1.000	1.000				0.934		0.987		0.898		0.868			
8	1.000	0.958		0.936	1.000	0.952				0.987		1.000		1.000		0.986			
10	1.000	1.000		0.988	0.999	0.994				0.993		0.969		0.989		0.999			
16							0.995	0.998	0.999		0.998		0.988		0.997		0.994	1.000	
18							0.995	1.000	0.993		1.000		1.000		0.999		0.995	0.998	
20							1.000	1.000	1.000		0.999		0.995		1.000		0.999	1.000	
22							1.000	1.000	0.998		0.999		0.999		0.999		0.992	0.999	
24							1.000	0.998	0.999		1.000		0.999		0.999		0.996	1.000	
26							1.000	0.999	1.000		0.999		1.000		0.999		0.999	0.999	
28							1.000	0.999	1.000		1.000		1.000		1.000		1.000	1.000	
30							1.000	0.999	0.998		1.000		1.000		1.000		1.000	0.998	
50							1.000			1.000					1.000				1.000
60							1.000			1.000					1.000				0.999
70							1.000			1.000					1.000				1.000
80							1.000			1.000					1.000				1.000
90	1.000		1.000		1.000		1.000			1.000					1.000				0.996
92	1.000		1.000		1.000														
94	1.000		1.000		1.000														
96	1.000		1.000		1.000														
98	1.000		1.000		1.000														
100	1.000		1.000		1.000		1.000			1.000					1.000				1.000

Figure 7.1: Margin of error of 1-OPT with Greedy algorithm initial solution

	Max bit-width																		
Number of elements	2	4	5	6	8	10	15	17	19	20	21	22	23	24	25	26	27	29	30
2	1.000	0.000		0.000	0.000	1.000				1.000		1.000		1.000		1.000			
4	1.000	1.000		1.000	0.603	1.000				0.508		1.000		1.000		0.998			
6	1.000	0.917		0.957	0.771	1.000				0.934		1.000		0.898		0.868			
8	1.000	0.958		0.981	0.944	0.986				1.000		1.000		1.000		1.000			
10	1.000	0.976		0.702	0.875	0.986				0.990		0.995		0.981		0.999			
16							0.993	0.996	1.000		0.992		0.998		0.988		0.998	0.998	
18							0.997	0.997	0.996		0.996		1.000		0.996		0.994	1.000	
20							0.999	1.000	1.000		0.997		0.995		0.998		0.996	0.996	
22							0.997	0.997	0.994		1.000		0.999		0.999		0.999	0.997	
24							0.999	0.999	0.999		1.000		0.999		1.000		0.999	0.999	
26							0.999	0.999	1.000		1.000		1.000		1.000		0.997	1.000	
28							1.000	1.000	1.000		1.000		0.999		1.000		0.999	1.000	
30							0.999	0.999	0.999		1.000		1.000		1.000		1.000	0.998	
50							1.000			1.000					1.000				1.000
60							1.000			1.000					1.000				1.000
70							1.000			1.000					1.000				1.000
80							1.000			1.000					1.000				1.000
90	1.000		1.000		1.000		1.000			1.000					1.000				0.998
92	1.000		1.000		1.000														
94	1.000		1.000		1.000														
96	1.000		1.000		1.000														
98	1.000		1.000		1.000														
100	1.000		1.000		1.000		1.000			1.000					1.000				1.000

Figure 7.2: Margin of error of 1-OPT with Random algorithm initial solution

For Tabu, we came within 0.994 of the target sum, which matches the result of the greedy algorithm. This may be because not a short enough history window was used.

7.3.2 Improvement Over Initial Solution

Given the different approaches to the initial solution, it is possible that this impacted the locally optimal solution for the instance. This can be expressed as a ratio of the sum of the locally optimal solution divided by the initial sum was found. Figures 7.4 an 7.5 show the amount of benefit the 1-OPT algorithm was able to provide over the initial solutions that were found. In general, neither algorithm provided a significant advantage over the other. That said, the instances that show the largest benefit were typically low density

Number of elements	Max bit-width: 2	4	5	6	8	10	15	17	19	20	21	22	23	24	25	26	27	29	30
2	1.000	1.000		1.000	1.000	1.000				1.000		1.000		1.000		1.000			
4	1.000	1.000		1.000	1.000	1.000				0.708		1.000		1.000		0.998			
6	1.000	1.000		1.000	1.000	1.000				0.934		0.987		0.898		0.868			
8	1.000	0.958		0.936	1.000	0.952				0.987		1.000		1.000		0.986			
10	1.000	1.000		0.988	0.999	0.994				0.993		0.969		0.989		0.999			
16							0.995	0.998	0.999		0.998		0.988		0.997		0.994	1.000	
18							0.995	1.000	0.993		1.000		1.000		0.999		0.995	0.998	
20							1.000	1.000	1.000		0.999		0.995		1.000		0.999	1.000	
22							1.000	1.000	0.998		0.999		0.999		0.999		0.992	0.999	
24							1.000	0.998	0.999		1.000		0.999		0.999		0.996	1.000	
26							1.000	0.999	1.000		0.999		1.000		0.999		0.999	0.999	
28							1.000	0.999	1.000		1.000		1.000		1.000		1.000	1.000	
30							1.000	0.999	0.998		1.000		1.000		1.000		1.000	0.998	
50							1.000			1.000					1.000				1.000
60							1.000			1.000					1.000				0.999
70							1.000			1.000					1.000				1.000
80							1.000			1.000					1.000				1.000
90	1.000		1.000		1.000		1.000			1.000					1.000				0.996
92	1.000		1.000		1.000														
94	1.000		1.000		1.000														
96	1.000		1.000		1.000														
98	1.000		1.000		1.000														
100	1.000		1.000		1.000		1.000			1.000					1.000				1.000

Figure 7.3: Margin of error of 1-OPT with Tabu search algorithm initial solution

(i.e. smaller values and a small number of elements in the set). This is most likely due to the fact that, given fewer numbers in a set and small numbers in general, adding and removing values is going to have a disproportionate impact on the current sum. Several instances show a value of 1.00, which means that the initial solution was actually sufficient. Ignoring the top-left region and the instances that showed no benefit, the remaining instances generally show only minor improvements.

It appears that the random algorithm did help some of the smaller density instances. Interestingly enough, given the random algorithm, it actually led to some cases where the instance could not be solved. This would mean that there there no solution that contained n elements, as selected in the initial solution. In that case the "landscape" was flat, with no locally optimal solution in the neighborhood.

On average, starting with the initial solution found via the greedy algorithm, 1-OPT was able to generate a solution with a sum that was 1.073 times larger on average. When the random algorithm was used instead, 1-OPT was able to come up with a solution that was 1.126 times larger than the initial solution on average. Note that we will not exceed the target sum. So, on average, starting with the solution led to the largest improvement in the sum

Number of elements	Max bit-width: 2	4	5	6	8	10	15	17	19	20	21	22	23	24	25	26	27	29	30
2	1.000	6.000		1.000	1.000	1.000				1.000		1.000		1.000		1.000			
4	1.000	1.063		1.000	1.000	1.542				1.000		1.784		1.000		1.000			
6	1.000	1.333		2.258	1.103	1.170				1.097		1.192		1.004		1.000			
8	1.000	1.045		1.157	1.101	1.279				1.003		1.039		1.056		1.229			
10	1.000	1.000		1.016	1.007	1.000				1.039		1.000		1.028		1.011			
16							1.016	1.003	1.017		1.013		1.040		1.004		1.005	1.071	
18							1.025	1.008	1.002		1.008		1.030		1.007		1.006	1.007	
20							1.003	1.023	1.000		1.031		1.000		1.012		1.012	1.000	
22							1.001	1.000	1.000		1.004		1.029		1.009		1.013	1.003	
24							1.021	1.014	1.007		1.001		1.010		1.010		1.005	1.003	
26							1.022	1.001	1.032		1.005		1.011		1.002		1.012	1.007	
28							1.014	1.002	1.000		1.015		1.003		1.003		1.007	1.009	
30							1.004	1.015	1.003		1.018		1.000		1.002		1.025	1.004	
50							1.001			1.000					1.001				1.003
60							1.006			1.003					1.000				1.656
70							1.001			1.000					1.002				1.003
80							1.002			1.001					1.001				1.016
90	1.000		1.005		1.001		1.000			1.002					1.003				1.002
92	1.000		1.000		1.001														
94	1.000		1.000		1.000														
96	1.000		1.000		1.000														
98	1.000		1.000		1.001														
100	1.000		1.000		1.001		1.003			1.001					1.000				1.020

Figure 7.4: Improvement of final sum over intial sum found via Greedy algorithm

	Max bit-width																		
Number of elements	2	4	5	6	8	10	15	17	19	20	21	22	23	24	25	26	27	29	30
2	1.000	0.000		0.000	0.000	1.000				1.000		1.000		1.000		1.000			
4	1.000	5.667		1.385	1.188	1.542				1.000		1.524		1.000		1.000			
6	1.000	3.667		1.196	1.199	1.000				1.000		1.198		1.004		1.000			
8	1.000	1.353		1.730	2.702	2.111				1.194		1.039		1.056		1.319			
10	1.000	2.733		1.366	1.012	1.022				1.011		1.000		1.084		1.056			
16							1.037	1.350	1.031		1.247		1.039		1.005		1.007	1.028	
18							1.033	1.018	1.025		1.240		1.025		1.031		1.000	1.025	
20							1.019	1.045	1.320		1.121		1.018		1.018		1.016	1.000	
22							1.094	1.040	1.055		1.065		1.016		1.043		1.003	1.000	
24							1.010	1.005	1.067		1.091		1.026		1.000		1.060	1.001	
26							1.082	1.036	1.036		1.007		1.008		1.014		1.000	1.022	
28							1.029	1.033	1.028		1.049		1.025		1.003		1.005	1.007	
30							1.055	1.015	1.051		1.117		1.021		1.004		1.011	1.003	
50							1.153			1.002					1.003				1.024
60							1.233			1.104					1.016				1.329
70							1.084			1.005					1.015				1.004
80							1.254			1.198					1.008				1.000
90	1.000		1.073		1.005		1.331			1.144					1.006				1.008
92	1.000		1.005		1.003														
94	1.000		1.046		1.037														
96	1.000		1.101		1.001														
98	1.013		1.103		1.035														
100	1.000		1.018		1.070		1.141			1.212					1.000				1.003

Figure 7.5: Improvement of final sum over intial sum found via Random algorithm

Figure 7.6 shows the benefit provided by the tabu algorithm. Note that, compared to the random algorithm, a locally optimal solution was found for each instance. On average, the final sum found in the Tabu search algorithm was 1.073 times larger than the initial solution. This is actually similar to the greedy algorithm, and technically worse on average than the random algorithm. This may be due to not using a large enough window, leading us to slip back down into the original valley and settle on the locally optimal solution there. This would imply relatively steep valleys, which is interesting given the improvements shown in the random algorithm imply large "flat" areas where there is no locally optimal solution (or, technically where all solutions are locally optimal). In terms of the shape of the landscape in general, it would appear that, depending on the density of the instance, the can be very flat areas where local search fails, and very steep areas where a significantly large window is needed by Tabu in order to find another valley (and thus another possible locally optimal solution).

	Max bit-width																		
Number of elements	2	4	5	6	8	10	15	17	19	20	21	22	23	24	25	26	27	29	30
2	1.000	6.000		1.000	1.000	1.000				1.000		1.000		1.000		1.000			
4	1.000	1.063		1.000	1.000	1.542				1.000		1.784		1.000		1.000			
6	1.000	1.333		2.258	1.103	1.170				1.097		1.192		1.004		1.000			
8	1.000	1.045		1.157	1.101	1.279				1.003		1.039		1.056		1.229			
10	1.000	1.000		1.016	1.007	1.000				1.039		1.000		1.028		1.011			
16							1.016	1.003	1.017		1.013		1.040		1.004		1.005	1.071	
18							1.025	1.008	1.002		1.008		1.030		1.007		1.006	1.007	
20							1.003	1.023	1.000		1.031		1.000		1.012		1.012	1.000	
22							1.001	1.000	1.000		1.004		1.029		1.009		1.013	1.003	
24							1.021	1.014	1.007		1.001		1.010		1.010		1.005	1.003	
26							1.022	1.001	1.032		1.005		1.011		1.002		1.012	1.007	
28							1.014	1.002	1.000		1.015		1.003		1.003		1.007	1.009	
30							1.004	1.015	1.003		1.018		1.000		1.002		1.025	1.004	
50							1.001			1.000					1.001				1.003
60							1.006			1.003					1.000				1.656
70							1.001			1.000					1.002				1.003
80							1.002			1.001					1.001				1.016
90	1.000		1.005		1.001		1.000			1.002					1.003				1.002
92	1.000		1.000		1.001														
94	1.000		1.000		1.000														
96	1.000		1.000		1.000														
98	1.000		1.000		1.001														
100	1.000		1.000		1.001		1.003			1.001					1.000				1.020

Figure 7.6: Improvement of final sum over intial sum using Tabu

It is interesting to see that, while the margin of error from the greedy algorithm implementation was smaller than that of the random algorithm (and matching that of Tabu), the actual amount of improvement over the inital sum was largest for the random algorithm. This highlights the benefit of local searches and de-emphasizes the importantace of how exactly the initial solution is found in this case.

7.4 Comparison of Algorithms

Given the multitude of algorithms that have been implemented and evaluated individually, a comparison between the algorithms is warranted. Figure 7.7 shows the run-time of each algorithm as a function of the instance density. Instance density was chosen since it was the original metric used to come up with the original distribution of instances.

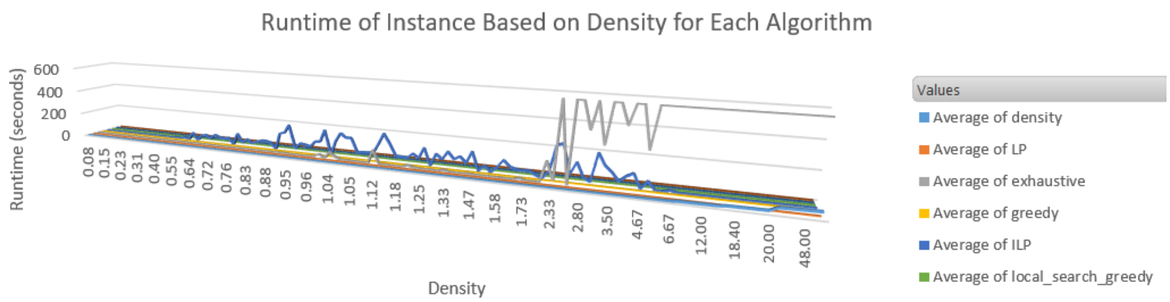

Figure 7.7: Comparison of runtimes based on instance density for each algorithm

Based on this plot, we see generally low variability for lower densities. However, for densities around 2 to 4, there this is some variation, namely between the exhaustive search and ILP implementations. In fact, these two algorithms are the source most of the variation across the densities. We can see that the ILP model does take some time to solve each instance, but is in a way consistent across the different densities. The exhaustive search algorithm, on the other hand, starts to hit the time limit, and is unable to solve instances of higher densities.

Figure 7.8 shows a comparison of the the margin between the solution found and the target sum for each algorithm, as a function of instance density. For this plot, a higher value is better (a value of 1.0 means the solution and target sum match and the instance was solved). For lower densities we see some variation across all the algorithms, with the greedy/1-OPT algorithm showing the most variation. As mentioned earlier in this chapter, this is most likely due to the instances with smaller sets and numbers (which would be implied by the lower density) causing each number swap to disproportionately impact the final sum. Beyond that, the ratios are relatively consistent until densities of about 2 and above, where the exhaustive search algorithm starts to fail, as was seen in Figure 7.7.

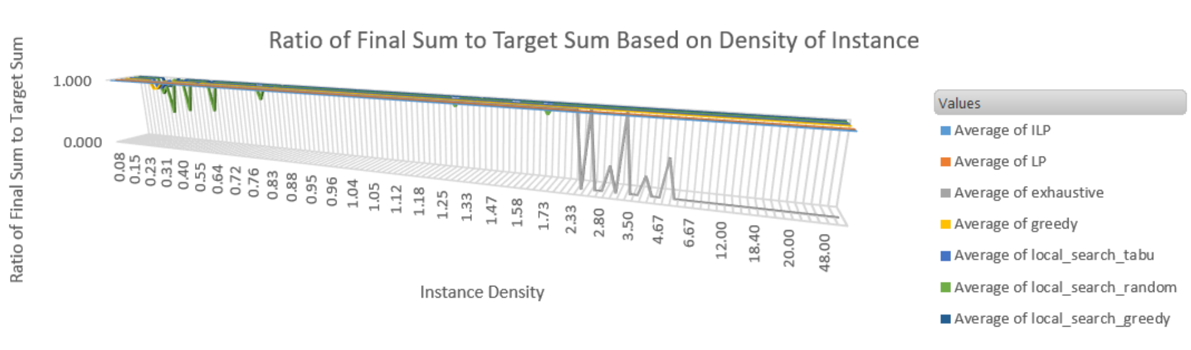

Figure 7.8: Comparison of final sum/target sum margin based on instance density for each algorithm

Focusing on a higher-level view across the different algorithms, Figure 7.9 shows the average ratio of the final sum and target sum across all instances for each algorithm. The range of the plot was cut to give greater emphasis to the variation between the non-exhaustive search algorithms. The exhaustive search algorithm, given its inability to solve higher-density instances, is naturally going to have a lower ratio on average. The local search algorithms provide decent margin, with the greedy and Tabu implementations in particularly proving to be better than random. That said, the ILP and LP models prove to be the best in the group. However, there is going to be the trade-off in run-time, as can be seen in Figure 7.7. Remember that there was significant variation across the instances for the ILP algorithm, but not for LP. Depending on how much margin one would be comfortable with having, the Tabu algorithm may be a better trade-off.

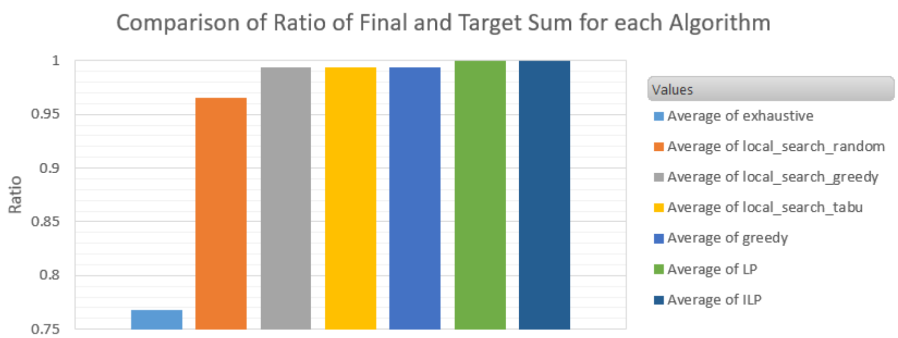

Figure 7.9: Comparison of overall average ratio of final sum/target sum for each algorithm

Chapter 8

Conclusion

Through this project we have done a deep exploration of the subset sum problem. We developed a suite of benchmark tests to evaluate a variety of algorithms that can be used to solve instances of this problem. These instances vary by their density, or the ratio of the number of elements in the set and the maximum bit width in the set. Each algorithm showed different results when run on the suite of instances, highlighting strengths and weaknesses. A literature review was also done to highlight subproblems of subset sum that have unique properties that could possibly be exploited by some solvers (if they were to identify these properties before-hand, and apply a given solver accordingly).

The exhaustive search algorithm's mileage varies significantly, but nevertheless is able to work on instances with a small enough number of elements in the initial set. This algorithm is actually a bit more agnostic to the density metric, being able to solve high density instances so long as there are sufficiently few enough elements in the original set. However, once the sets become large enough, this algorithm very quickly starts to take much too long to solve anything.

The greedy algorithm is relatively quick to come up with some solution, but it is inherently prone to making a "wrong decision" that it cannot undo, ending up on a path that will never lead it to the correct solution. This is particularly apparent for instances with relatively small sets and large numbers; in that case the solver may incorrectly chose some number to include in its set, but the inclusion of that number actually may not be allowed in the final set. This introduces a margin of error that can vary significantly, but on the other hand leads to a low computation time.

The LP solver proved to be the fastest solver, but at the same time did not produce a valid solution because fractions of values in the original set are used. The ILP solver addresses this constraint, but is directly trying to solve the true NP-complete problem. Nevertheless, the ILP solver was able to solve all of the instances, albeit with significant variation in run time. While the LP solver can provide a bound for the ILP solver, the bound is technically already known in the sense that we know what the target sum, so really the LP solver itself does not provide much in this context. Linear programming in general has proved to be a very powerful tool in solving all of these instances.

The last set of the algorithms that were evaluated were the local search algorithms. Initial instances were created via the greedy and random algorithms, using 1-OPT to define and explore the neighborhood. Starting with an instance from the greedy algorithm proved to yield slightly better results, granted the solution random/1-OPT algorithm was significantly improved. Yet interestingly some instances could not even be solved with with random/1-OPT, highlighting the fact that, in the landscape of possible solutions, there can be flat areas where all local solutions are equally optimal. Lastly, a Tabu search algorithm was implemented, proving to be comparable to the the greedy/1-OPT algorithm. The window size of 100 that was used in Tabu may have been too small. The fact that this window size proved inadequate suggests that the locally optimal solutions are in very steep "valleys." Coupled with the results seen from the random/1-OPT algorithm, it is interesting to see that the landscape of subset sum solutions can vary wildly from extremely flat (all solutions are equally good) to extremely steep (a significantly large Tabu window is needed in order to find signifcant locally optimal solution). This becomes a trade-off of memory consumption of the window and how much better a locally optimal solution we can hope to find. We also found that the 1-OPT neighborhood was relatively small, with each of our algorithms able to iterate through all possible neighbors.

Bibliography

[1] Mark Chaimovich and Gregory Freiman. Solving dense subset-sum problems by using analytical number theory. *Journal of Complexity*, 1989.

[2] Lanjun Wan et al. Gpu implementation of a parallel two-list algorithm for the subset-sum problem. *Concurrency and Computation: Practice and Experience*, 27.1:119–145, 2015.

[3] Konstantinos Koiliaris and Chao Xu. A faster pseudopolynomial time algorithm for subset sum. *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2016.

[4] Jeffrey C. Lagarias and Andrew M. Odlyzko. Solving low-density subset sum problems. *Journal of the ACM (JACM)*, 32(1):229–246, 1985.

[5] Brian A. LaMacchia. Basis reduction algorithms and subset sum problems. Master's thesis, Massachusetts Institute of Technology, 1991.

[6] Hendrik Willem Lenstra Lenstra, Arjen Klaas and Laszlo Lovasz. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.

[7] Stanislaw Radziszowski and Donald Kreher. Solving subset sum problems with the l3 algorithm. *The Charles Babbage Research Centre: The Journal of Combinatorial Mathematics and Combinatorial Computing*, 3, 1988.

[8] C. P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical programming*, 66.1-3:181–199, 1994.