EECE 7360 Project 3

Garrett Goode and Daniel Hullihen

Spring 2017

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

$$s = \sum_{i=1}^{n} a_i, for 1 \leq i \leq n.$$

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 would 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, we developed an implementation of a greedy algorithm for the subset sum problem, and ran it against a suite of instances to evaluate the performance of this algorithm.

2 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 1 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.

3 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.

4 Results

The results are presented in Figure 1 below. The vertical axis represents the input size and the horizontal axis the word length of the elements in bits.

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 2 shows the run-times for the exhaustive algorithm.

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

Looking at the results described in Figure 1, 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 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

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

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.