The Subset-Sum Problem: Revisited with an Improved Approximated Solution

Hashem A. Isa
Faculty of Information Technology, Al-Balqa Applied University, Jordan

Saleh Oqeili
Faculty of Information Technology, Al-Balqa Applied University, Jordan

Sulieman Bani-Ahmad
Faculty of Information Technology, Al-Balqa Applied University, Jordan

ABSTRACT
The Subset-sum Problem is one of the easiest to describe and understand NP-complete problems. Available algorithms that solve this problem exactly need an exponential time, thus finding a solution to this problem is not currently feasible. The current paper revisits the subset-sum problem and suggests a new approach to find an approximate solution to this problem. The proposed algorithm gives a reasonable solution with a polynomial time-complexity.

Keywords: NP-complete problem, the subset-sum problem.

1. INTRODUCTION
The Subset-Sum Problem (SSP) is defined as follows: given a set of positive integers S, e.g., {s1, s2, s3, s4, s5, s6}, and a positive integer C. This problem is to find one/all subsets of S that sum as close as possible to, but do not exceed, C [1, 2]. For an example, consider the set S={1, 2, 3, 4, 5} and let the target sum C be 10. The total number of subsets of S in this case is 25. Some of the valid solutions to this problem are the sets {1, 2, 3, 4}, {1, 4, 5}, and {2, 3, 5}.

In general, we notice that the total number of subsets taken from a set of n elements is 2n [7, 8]. An algorithm that tests all of these possible solution subsets needs an exponential time. Let the number of inputs, that is the size of set S, be n. Using a computer that can generate and test one subset in one microsecond we need 0.001 sec to solve an SSP problem of input size 10. However, as this number of inputs grows to 100, we need 4.1016 years on that same machine! This was the main motive to work on this problem.

Approximation Algorithms try to attack NP-complete problems [1, 5, 6]. Since it's unlikely that there can be efficient algorithms that solve such problems, one settles for non-optimal solutions in order to have approximate solution be found in a polynomial time [14, 16]. Unlike heuristics, which just usually find good solutions reasonably fast, one may need provable solution quality and provable run time bounds both achieved together.

The current paper introduces a new approach that promises finding some of the possible solutions to the SSP problems based on a set of current user's constraints. These constraints mainly decide upon the maximum number of subsets required. Based on the provided constraints, and using a polynomial number of iterations the proposed algorithm can successfully produce some of approximated and valid solutions to the problem. The proposed algorithm can successfully produce all possible solutions.

The rest of this paper is organized as follows: in section 2 we quickly survey the already proposed approximated solutions. We explain the proposed approximation algorithm in section 3. A set of illustrative examples to clarify the major steps of our proposed solution are presented in section 4. The proposed solution is analyzed in terms of time and space complexities in sections 5 and 6. Finally, section 7 is a conclusion.

2. APPROXIMATION SCHEMES
As an example for an approximation algorithm, consider a greedy algorithm for solving subset-sum problem that starts with an empty solution subset and examines the input numbers in decreasing order of their values [16]. Each considered number is inserted into the current solution if and only if it is smaller than the difference between the target sum and the sum of the current solution [16].

The time complexity of such approach is dominated by time required by the sorting algorithm used to sorting the numbers in hand [16, 7]. Notice that the greedy algorithm itself needs linear time which is as small as O(n).

We can use a randomized algorithm that does not require the numbers in hand to be sorted as a priory step. This should reduce its execution time and the resulting randomized algorithm gives often good results if we perform a few independent trials on the given data and finally return the best solution.

3. THE PROPOSED SOLUTION
The time requirement of the proposed algorithm described below can be controlled by predetermining the number iterations based on the current user input. Given some predetermined time-frame, the algorithm may or may not find a solution for the problem using the specified number of iterations. When the provided number of iterations is exhausted, the algorithm stops and displays the valid solutions(s) that it has achieved within given time-frame. Sometimes, the number of iterations may be small and in this case the algorithm will stop before finding all solutions.

Algorithm Subset-sum Approximation

Inputs*:*

Integer N, number of elements

Elements, a set of N distinct positive integers

Integer C, the target sum

Output:

All subsets of the set Elements that have a sum equal to C.

Important declarations:

SSP(int N, int Elements[N], int C)

int Bitmap[N];

int K;

int PSum;

int Iteration;

```
int Count;
int I, J;
boolean Found;
```

Where

- Bitmap is an array of size N and is used to store ones and zeros; 1 at position x indicated that the xth element is part of the identified solution. N is the size of the set of integers in hand.
- K is the index of the last element in the sorted array that is smaller than the required sum C.
- PSum holds the current partial sum.
- Iteration is the number of already elapsed iterations (required to control the number of iterations covered).
- Count holds the total number of already identified valid solutions.

Step 1:

```
// sort Elements in an  ascending order
//(using any sorting technique)
SortElements(N, Elements[N]);
// set Count and Iteration
Iteration = 1;
Count = 0;
```

Note: SortElements() is a function called to sort the set of elements in ascending order. At this point, the variable Iteration is set to 1 indicating the beginning of the first iteration.

Step 2:

```
// remove from Elements all numbers greater than C
while Elements(N) > C
  N = N – 1
// set current number of elements
K = N;
```

Step 3:

```
// fill Bitmap with zeros starting from
// position K down to 1
for I = K downto 1
Bitmap[I] = 0;
```

Step 4:

```
// set partial sum to zero
PSum = 0;
```

Step 5:

```
// find the first partial subset and its sum
for I = K downto 1
if PSum + Elements[I] <= C
Bitmap[I] = 1;
PSum += Elements[I];
```

Step 6:

```
// print the partial subset if its sum equals C
// and add 1 to Count
if C = PSum
for I = 1 to N
if Bitmap[I] = 1
print Elements[I];
Count++;
```

Step 7:

```
// try to find the first “01”
// starting from the beginning of Bitmap
J = 1;
Found = False;
while J <= N and Not Found
if Bitmap[J] = 0 and Bitmap[J + 1] = 1
Found = True
else
J++;
```

Step 8:

```
// if we can not find “01” then there are no
// more solutions
if not Found
        go to Step 11;
else
// swap the “01” and fill the left side
// of Bitmap with zeros starting from J - 2
Bitmap[J] = 0;
Bitmap[J – 1] = 1;
for I = J – 2 downto 1
Bitmap[I] =  0;
```

Step 9:

```
// recalculate PSum for numbers from N downto J - 1
PSum = 0;
for I = N downto J – 1
if Bitmap[I] = 1
PSum += Elements[I];
```

Step 10:

```
// find the next partial subset and it sum
K = J – 2;
// count the number of iterations
Iteration = Iteration + 1
go to Step 5;
```

Step 11:

// End – no more solutions

print Iteration, Count;

The algorithm finds all subsets that have a sum that is equal to C. The user does not specify the maximum number of iterations. Instead, the algorithm stops when there are no more solutions available. At that point of time, it prints the number of iterations needed and the number of subsets found.

To allow the user to specify the maximum number of iterations (to control the execution time of the algorithm), Step 10 can be modified as follows:

Step 10:
// find the next partial subset and its sum
K = J – 2;
// count the number of iterations
Iteration=Iteration++
if Iteration <= MaxIterations
go to Step 5;

4. AN ILLUSTRATIVE EXAMPLE

The input in this example is the following set: {3, 5, 6, 8, 10, 11, 12, 14, 15, 17}. Their size of this set is 10, and the target sum is 25. Our goal is to find all subsets that sum to the target sum. We use the following table to trace the algorithm.

The numbers										C
3	5	6	8	10	11	12	14	15	17	25

Start from the right-hand side of the numbers (the largest number after sorting the set in ascending order) and try to find a partial set of numbers with sum less than or equal to 25. Put a "1" under the number you include in this partial set. In this example, the numbers 17 and 8 have a sum that is equal to 25, therefore the first subset in the solution is {8, 17} as shown next.

The numbers										C
3	5	6	8	10	11	12	14	15	17	25
			1						1	

Now, try to find the next solution. Start from the left side and find the first "1" after a set of zeros. This "1" is below the number 8. Move this "1" to the left one step and try to find another solution with sum less than or equal to 25.

The numbers										C
3	5	6	8	10	11	12	14	15	17	25
		1							1	

The sum of the numbers that have a "1" under them is 17 + 6 = 23, so we continue with the rest of the numbers to the left of 6. We find that those numbers when added to the previous sum 23 will give a new sum that is greater than 25, so we move the leftmost "1" to the left to be under the number 5. Now the partial sum is 17 + 5 = 22. We continue with the number 3 to find the second solution {3, 5, 17}.

The numbers										C
3	5	6	8	10	11	12	14	15	17	25
1	1								1	

Now we repeat the same steps, starting from the left and looking for another "1" that comes after a group of zeros. We find the number 17, so we move the "1" below it to the left. This means that there are no other solutions that contain the number 17. The same steps are followed to find the next solution that is {10, 15}.

The numbers										C
3	5	6	8	10	11	12	14	15	17	25
				1				1		

The same steps are repeated again and again. During that we will find other solutions that are {11, 14}, {3, 8, 14}, {5, 6, 14}, {3, 10, 12}, {5, 8, 12}, {6, 8, 11}, {3, 5, 6, 11}. Note that each time we move a "1" to the left, we empty all the ones to the left of it. The final configurations of numbers is:

The numbers										C
3	5	6	8	10	11	12	14	15	17	25
1	1	1			1					

This is the last step. We try to find the first "1" after a group of zeros and we find the 1 under the number 11, so we move it to the left under the number 10. We try to find a sum that is equal to 25 from the numbers 10, 8, 6, 5 and 3, but we could not, so we again move the 1 under 10 to the left.

The numbers										C
3	5	6	8	10	11	12	14	15	17	25
			1							

Since the sum of all numbers starting from 8 is less than 25, the algorithm will stop and this means that there are no more solutions.

5. ALGORITHM COMPLEXITY

Let T represent the number of iterations done by the algorithm to find all subsets that sum up to a given target sum C based on n input numbers. We can estimate the worst case complexity as follows:

Removing numbers that are larger than C need n steps. This step is optional since the algorithm will not consider all the numbers that are greater than C from the first iteration.

Initializing the array Bitmap: n steps.

T iterations, each contains the following steps:

Finding a partial sum: n steps.

Print a solution: n steps.

Finding the first "01" in the array Bitmap: n steps.

If we add the number of steps needed to sort the array of numbers, then the worst time complexity of the algorithm will be $O(n \log n + 3n \cdot T)$.

6. EXPERIMENTAL RESULTS AND ANALYSIS

Table 1 shows some experimental results of the proposed algorithm. The algorithm was tested using the set {1, 2, …, 100} by taking 10 more numbers each time. The table shows the number of iterations and the number of subsets found.

Figure 1 shows the relations between number of values and the number of iterations done by the algorithm. Figure 2 shows the relationship between number of iterations and number of subsets found. Note that the relation is linear with a slope of

0.77. This means that the algorithm finds 77 solutions in each set of 100 iterations.

7. CONCLUSION

The proposed algorithm presented above is a good approximation algorithm for the subset-sum problem. It finds all solutions for a given input or based on a maximum number of iterations, which prevents it from going very long search paths.

The experimental results show that the proposed algorithm's performance is 77%. This means that the proposed algorithm finds an average of 77 subsets in each bulk of 100 iterations which is a high percent.

Table 1: Different problem sizes vs number of iterations and the number of subsets found.

Numbers	C	#Iterations	#Subsets
1 – 10	10	11	10
1 – 20	20	78	64
1 – 30	30	370	296
1 – 40	40	1,416	1,113
1 – 50	50	4,708	3,658
1 – 60	60	14,130	10,880
1 – 70	70	39,168	29,927
1 – 80	80	101,820	77,312
1 – 90	90	251,028	189,586
1 – 100	100	591,724	444,793

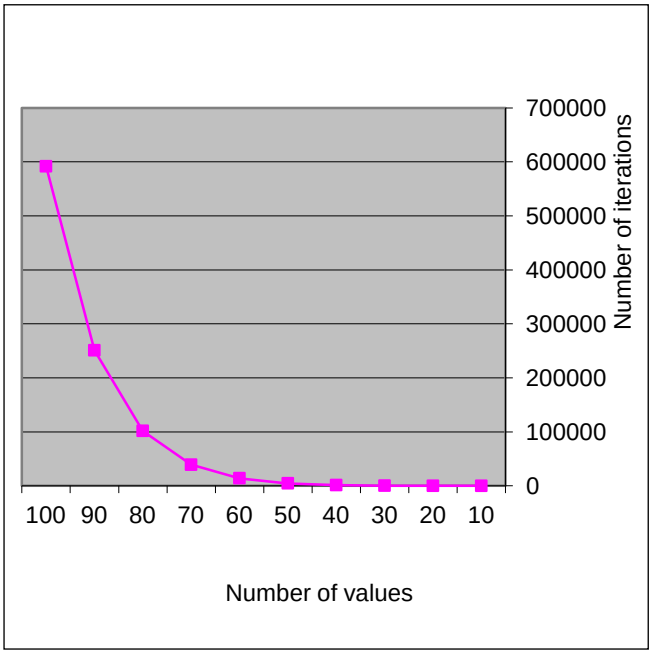

Figure 1: Time complexity of the proposed algorithm

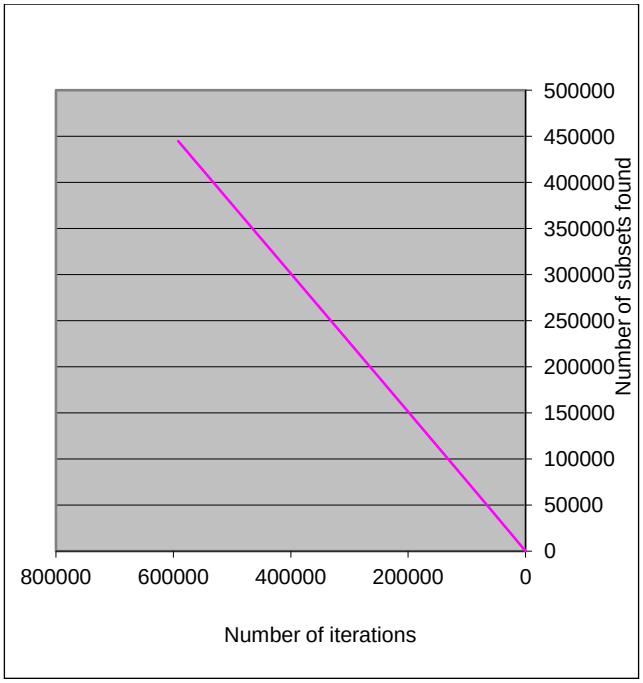

Figure 2: The relationship between number of iterations and number of subsets found using the proposed algorithm.

8. REFERENCES

[1] Baase, S. and Gelder, A. V. (2000). *Computer Algorithms.* Addison Wesley Longman.

[2] Bazgan, C., Santha, M., and Tuza, Z. (1998). *Efficient approximation algorithms for the subset-sum equality problem.*

[3] Bentley, J. (1986). *Programming Pearls*, Addison-Wesley Reading.

[4] Blair, C. (1994). *Notes on Cryptography*. Business Administration Dept., University of Illinois, http://www.math.sunysb.edu/~scott/blair/Blair_s_Cryptography_Notes.html

[5] Borwein, J. and Bailey, D. (2003) *Mathematics by Experiment: Plausible Reasoning in the 21st Century*, Natick, MA: A. K. Peters.

[6] Cook, S. A. (1971). *The complexity of theorem proving procedures.* Third Annual ACM Symposium on the Theory of Computing, ACM, New York.

[7] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms, 2nd Edition*. MIT Press and McGraw-Hill, 2001.

[8] Coster, M. J.; Joux, A.; LaMacchia, B. A.; Odlyzko, A. M.; Schnorr, C. P.; and Stern, J. (1992). *Improved Low-Density Subset-sum Algorithms*, Computing Complex. 2.

[9] Ferguson, H. R. P. and Bailey, D. H. (1992). *A Polynomial Time, Numerically Stable Integer Relation Algorithm*, RNR Technical Report RNR-91-032.

[10] Garey, M. and Johnson, D. (1979). *Computers and Intractability; A Guide to the Theory of NP-Completeness.*

[11] Garey, M., Johnson, D., and Stockmeyer, L. (1974). *Some simplified NP-complete problems*, Proceedings of the sixth annual ACM symposium on Theory of computing..

[12] Hodges, A. (1970). *Alan Turing: The Enigma*, Simon and Schuster, New York.

[13] Impagliazzo R. and Naor M., (1996). *Efficient cryptographic schemes provably as secure as subset-sum*, Department of Computer Science, University of California at San Diego, 1996.

[14] Lagarias, L. C. and Odlyzko, A. M. (1985) "Solving Low-Density Subset-sum Problems." Journal of ACM 32.

[15] Karp, R. (1972). *Reducibility Among Combinatorial Problems*. Proceedings of a Symposium on the Complexity of Computer Computations.

[16] Martello, S. and Toth, P. (1984). *Worst case analysis of greedy algorithms for the subset-sum problem.* Mathematical Programming.

[17] Papadimitriou, C.H. (1994). *Computational Complexity*. Addison-Wesley.