A Visualization Program for Subset Sum Instances

Thomas E. O'Neil and Abhilasha Bhatia
Computer Science Department
University of North Dakota
Grand Forks, ND 58202
oneil@cs.und.edu
abhilasha.bhatia@my.und.edu

Abstract

This paper describes a program called *SumFinder* that is designed to support experimentation with instances of the Subset Sum problem. Subset Sum is perhaps the simplest of the NP-complete problems. It can be defined as follows: given a set of positive integers S and a target sum t, is there a subset of S whose sum is t? The *SumFinder* program provides the ability to generate random sets of integers as problem instances given a set size n and a maximum value m. Editing operations are also provided to allow the user to create any specific set. The program automatically generates all sums for subsets of the input set S, and for each sum t in the sum set, the program will display all subsets that have sum t. In addition to the processing of individual input sets, the program provides the ability to test all sets with given values of n and m to determine whether there are ranges of sums that are always generated.

SumFinder has value for both instruction and research. It can be used as an instructional tool for introducing a simple NP-Complete problem. Experimentation with *SumFinder* makes the combinatorial nature of the Subset Sum problem immediately evident. It also provides a thinking exercise that requires multiple levels of abstraction, including numbers, sets of numbers, sets of subsets of a set of numbers, and the set of sums of subsets of a set of numbers. As a research tool, *SumFinder* is being used in an on-going study of the conjecture that Subset Sum has a specific density-based decision threshold. It appears that if the density of an input set is high enough, a subset can be found for any non-peripheral target sum. Specifically, if $n \geq m/2 + 2$, there is a subset for any sum t such that $m < t < sum(S) - t$. There is substantial empirical evidence in support of this conjecture, but the proof remains open. The *SumFinder* program is being used to search for additional properties of the sum sets that would lead to a proof of the conjecture.

1 Introduction

Subset Sum is one of the simplest NP-complete problems. It can be defined as follows: given a set of positive integers and a target value, determine whether some subset has a sum equal to the target. *SumFinder* is a Java program that solves single instances of the Subset Sum problem. An instance is just a set of positive integers, which can be entered directly by the user or produced by a random instance generator. The program is designed to illustrate the entire range of subsets and subset sums. Given an instance of the problem, it shows which sums can and cannot be found. And for each sum that can be produced, the program shows the list of all subsets that produce it.

As an instructional tool, *SumFinder* is valuable for illustration of a combinatorial problem. Small problem instances are seen to be easily manageable, but the number of subsets associated with each sum escalates rapidly as the problem instances grow larger. The program requires the user to conceptualize both sets of numbers and sets of sets of numbers. The initial problem instance is just a set of numbers. The program generates the sum set of the input set – the set of all sums of subsets of the input set. While the logic of the program is based on computation of all possible sums rather than all possible subsets, it effectively computes the power set of the input. Each sum is associated with a subset of the power set. Using the program becomes an exercise in thinking at multiple levels of abstraction.

As a research tool, *SumFinder* can be used to search for and test properties of the sum sets of Subset Sum instances. The space of problem instances can be parametrized using the size of the input set n and the maximum number in the input set m. The density of an instance is the ratio n/m. The time complexity of Subset Sum is very sensitive to the density of the input set. The critical region of the problem space, where instances have about a 50% chance of being solvable, is found at the very low density of about $n/2^n$ [6]. So the instances that are most difficult to solve are relatively small sets of very large numbers. While all known exact algorithms require exponential time for such low-density instances, a few algorithms have been defined that operate in expected polynomial time for medium and high density instances [3, 4]. For very dense instances, Subset Sum decision can be trivial. There is a density threshold beyond which the decision is always true for central target sums. An approximation of this threshold is given in [1], and an exact threshold is specified as a conjecture in [7]. Proof of the exact threshold apparently remains an open problem, and the *SumFinder* program provides a tool for discovery of properties of sum sets that might lead to a proof of the threshold conjecture.

This paper gives a description of the *SumFinder* program and discusses how it has been used to test properties of sum sets. Section 2 below describes the software and Section 3 illustrates its use as a research tool.

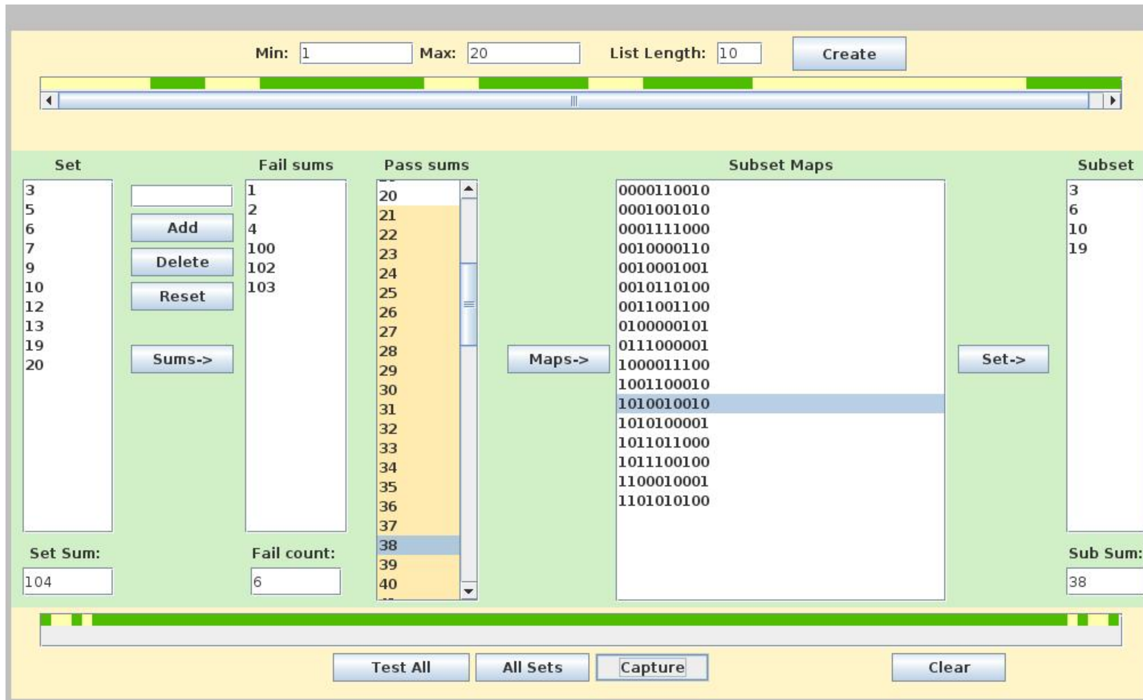

Figure 1: An experiment with 10 numbers between 1 and 20.

2 The *SumFinder* Program

Figure 1 shows the *SumFinder* output for a randomly generated set of 10 numbers between 1 and 20. The text boxes and *Create* button across the top of the application frame allow the user to create a random input set. Boxes are provided for the minimum value in the set (*Min*), the maximum value (*Max*), and the size of the set (*List Length*). The *Create* button generates a random set using the values of these parameters as entered by the user. Thereafter it initiates the *checkSums()* method, which is applied on the generated list to find the sums of all possible subsets.

The program uses Java's *Random* class to generate the set of numbers. Code for the input set generator is shown in Figure 2. The logic simulates selection without replacement by skipping duplicates. If the set size is more than half of the maximum value, a set is created by randomly generating its complement. This ensures that the random generator never needs to produce more than half the numbers in the range $\{1, \ldots, m\}$, thus preventing delays that might result from recurring duplications while generating very dense sets.

Once an input set has been generated, it is displayed in the leftmost list box. The sum of the entire set is shown in a text box just below the list box of the application frame. The utility buttons *Add* and *Delete* allow users to add elements to the list and remove elements from the list respectively. The addition and removal operations also update the *Max* and *List Length* fields if necessary. The *Reset* button generates a list that consists of all elements between the *Min* and *Max* values. These features allow the user to manually create

```
public OrderedIntSet makeIntSet(int size, int max)
{
  int count = 0;
  boolean [] map = new boolean [max]; //false by default
  boolean complement = false;
  Random randgen = new Random();
  if (max < size)
    return null;
  else if (size > max/2)
  {
    size = max - size;
    complement = true;
  }
  while (count < size)
  {
    int next = randgen.nextInt(max);
    if (map[next] == false)
    {
      map[next] = true;
      count++;
    }
  }
  OrderedIntSet setlist = new OrderedIntSet();
  for (int i=0; i<max; i++)
    if (complement && !map[i] || !complement && map[i])
      setlist.add(i+1);
  return setlist;
}
```

Figure 2: A random integer set generator

or modify an input set. A visual representation of a bit map of the input set is displayed in the bar across the top panel. Dark shaded areas (green) in the bar represent numbers that are present, while light shaded areas (yellow) indicate missing numbers.

For an input set that has been manually entered or modified, its sum set is displayed when the *Sums* button is pushed. Two list boxes are used to display the sums. If some subset of the input set will produce a sum, the sum is displayed in the right box. If no subset can produce a sum, it is listed in the left box. Shading is used in the sum boxes to distinguish peripheral sums from central sums. For an input set S with maximum m, the central sums are those greater than m and less than the sum of S minus m. Central sums are important for the decision threshold conjecture, which is discussed further in the next section. A visual bit map for the sum set is displayed across the bottom panel of the application frame. As with the input set, missing sums are shaded lighter. It is immediately apparent from the

visual bit map in Figure 1 that the sum set is symmetric. This illustrates a general property of sum sets for all problem instances: if the input set S has a subset X with sum t, then there is also a subset $S - X$ whose sum is the sum of S minus t.

The logic for computing all subsets for each possible sum is an example of the dynamic programming technique. The code for the computation of the sum set is shown in Figure 3. There may be multiple distinct subsets (solutions) that add up to each sum, so the sum set is an array (called *sollist[]*) of references to lists of solutions, where each solution is represented as a bit map of n bits. Initially, a bit map representing the empty set is appended to *sollist[0]*, and the solution lists for every other sum are left empty. The numbers from the input set are processed one at a time from low to high. For each input number *next*, *sollist[]* is traversed from high index to low. If *sollist[i]* is not empty, then for each solution on the *sollist[i]* list, a new solution is created by adding *next* and the new solution is appended to the *sollist[i+next]* list.

```
public SolutionSet [] findSums()
{
  sollist[0] = new SolutionSet(0, numcount);
  sollist[0].add(zeroString(numcount));
  for (int i=1; i<=setsum; i++)
    sollist[i] = new SolutionSet(i, numcount);
  int topone = 0;
  for (int j=0; j<numcount; j++)
  {
    int next = intset[j];
    for (int i=topone;i>=0; i--)
    {
      if (sollist[i].setlist.size() > 0)
      {
        for (String smap: sollist[i].setlist)
        {
          StringBuilder newmap = new StringBuilder(smap);
          newmap.setCharAt(j, '1');
          sollist[i+next].add(newmap.toString());
        }
      }
    }
    topone = topone + next;
  }
  return sollist;
}
```

Figure 3: Computing the subsets for all sums via dynamic programming

The user can select any sum on the sum list and press the *Maps* button to see the bit maps of the subsets that add up to that sum. Each bit map is a binary bit string representation where positions of the 1-bits represent the elements of the subset. If position i in the bit string is 1, then element i from the ordered list of set elements is in the subset. The user can select any subset map and press the *Set* button to display the subset as a list of values in the rightmost list box of the application frame. The sum of the subset is displayed below the rightmost list box.

```
private String successor(String w, int onecount)
{  // Returns the successor of string w with the same
   // length and the same number of ones.  The onecount
   // parameter is assumed to be the number of ones in w.
   // Returns null if there is no such successor
   // of the same length.

   int n = w.length();
   String suffix;
   if (onecount == n || onecount == 0)
      return null;
   if (w.charAt(0) == '0')
   {
      suffix = successor(w.substring(1), onecount);
      if (suffix != null)
         return "0" + suffix;
      else
         return "1"+ zeroString(n-onecount) +
          oneString(onecount-1);
   }
   else // w.charAt(0) == '1'
   {
      suffix = successor(w.substring(1), onecount-1);
      if (suffix != null)
         return "1" + suffix;
      else
         return null;
   }
}
```

Figure 4: Successor computation for bit strings with $\binom{m}{n}$ ones

In addition to processing one problem instance at a time, *SumFinder* can perform an exhaustive enumeration of all instances with the specified values of n and m while checking for some property of the sum set. The enumeration and testing continues until a problem instance fails the property test, and the failed instance is displayed in the list boxes for

closer examination by the user. The bottom panel has two buttons that trigger an exhaustive enumeration: the *All sets* button tests the sum set for missing central sums, and the *Test all* button provides an additional enumeration that tests some property of interest to the user. The property to be tested is specified by modifying the methods of the *PropertyTester* class. To test a new property, it is therefore necessary to modify and recompile the code.

The property tests are discussed in more detail in the next section. The enumeration of problem instances uses logic that generates bit strings. A string of m bits that has n ones provides a bit map for a problem instance. Enumeration of all instances reduces to the problem of systematically generating all $\binom{m}{n}$ strings of m bits with n ones. The logic employed by *SumFinder* generates these strings in lexicographic order (as defined and illustrated in [5], p. 17). The code for a recursive *successor* function for this ordering can be found in Figure 4.

Finally, the bottom panel provides a *Capture* button takes the snapshot of the current state of the interface. The snapshot is stored as an image file called *jframe.jpg* in the user's current directory. This makes it convenient to transfer the results of interesting experiments to written reports.

3 Experimenting with *SumFinder*

The *SumFinder* program computes the sums of all subsets of its input set. All subsets are stored in lists for potential display. As a result, the program has exponential time and space requirements. On a desktop with two 3.0 GHz Pentium 4 processors and a 2.5 GiB main memory, sets of more than 22 integers can cause Java *OutOfMemory* errors.

The experiment illustrated in Figure 1 creates a set of $n = 10$ numbers between 1 and $m = 20$. The sum of the entire set is 104, and only 6 of the 104 possible subset sums are missing. A sum t is defined to be central with respect to input set S if $m < t < \Sigma S - m$, where ΣS represents the sum of all elements in S. It is interesting to note that there are no missing central sums in the experiment of Figure 1. It is conjectured in [7] that for any set with $n > m/2 + 1$, there will be no missing central sums. The *All sets* button can be used to test this conjecture for small values of n and m, and no counter-examples will emerge. The proof of this conjecture, however, is apparently an open problem. A volunteer computing project was undertaken at the University of North Dakota to extend the search for a counter-example [2], and to date, all sets with $n < 49$ have been tested, and no counter-example has been found.

While empirical evidence in favor of the conjecture is strong, the proof has been elusive. The *SumFinder* program is intended to reveal properties of sum sets that will lead to a proof of the conjecture. The research literature contains a theorem that estimates the probability of finding the innermost sums in the central region based on the density of the input set [1], but the theorem is not strong enough to cover the entire central region. The width of the

Maximum value m	Instance size n	Instances with missing central sums
8	5	3
9	5	21
10	6	2
11	6	29
12	7	2
13	7	28
14	8	2
15	8	35
16	9	2
17	9	43
18	10	2
19	10	57
20	11	2
21	11	72
22	12	2
23	13	94

Figure 5: Counting sets with missing central sums

central cluster of sums from the theorem is only $2m \log m$, whereas the width of the entire central region in the threshold conjecture is $\Theta(m^2)$.

It is informative to examine sets with density just below the threshold that have missing central sums. If we try all sets with $n = 11$ and $m = 20$, for example, we find that there are 2 sets with missing central sums. The number of sets with missing central sums for $8 \leq m \leq 22$ and $n = m/2+1$ is shown in the table of Figure 5. It is interesting to note that if m is even, the number of sets that fail the test is constant at 2, while the number of sets that fail for odd values of m is apparently an increasing function of n or m (or both). It is also interesting that for even values of m, the sets that fail are always $\{1, n, n+1, \ldots, m\}$ and $\{1, 2, n+1, n+2, \ldots, m\}$. We also see that the target sums for which these sets fail are very close to the edges of the central region.

The threshold conjecture implies that for any set with density higher than the threshold, we can swap a number in the set with a number not in the set (provided both are within $\{1, \ldots, m\}$) and still find a subset for every central sum. This raises the question of redundancy in the solution sets for a given central sum t. Is it possible that every subset with sum t must contain some value x, or will we always find at least one subset with and one without x? Let $\Gamma_S(t)$ denote the *solution set* for a given target sum. $\Gamma_S(t)$ is defined to be the collection of subsets of S that have sum t. We define the *depth* of $\Gamma_S(t)$ to be the size of the smallest set cover for the collection. $\Gamma_S(t)$ is *redundant* if it has depth greater than 1. We can use the testing capability of *SumFinder* to test for redundancy. The *Test all* button on the bottom panel of the application frame will trigger an exhaustive enumeration of sets with the specified size and maximum value. For each such problem instance, the program will test for some property of the sum set (other than missing central sums), as coded by

the user. Figure 6 shows the results of a test for redundancy on a set of 10 numbers with maximum value 16. This set exceeds the density threshold, so there is a non-empty solution set for every central sum. The redundancy test shows, however, that the solution sets are not all redundant. The solution sets $\Gamma_S(t)$ for nine values of t between 87 and 98 failed the test. Examination of the solution set for $t = 89$ reveals that every subset with sum 89 contains the number 13, constituting a set cover of size 1. Removal of 13 from the input set S would obviously induce missing central sums for a set S' just below the density threshold.

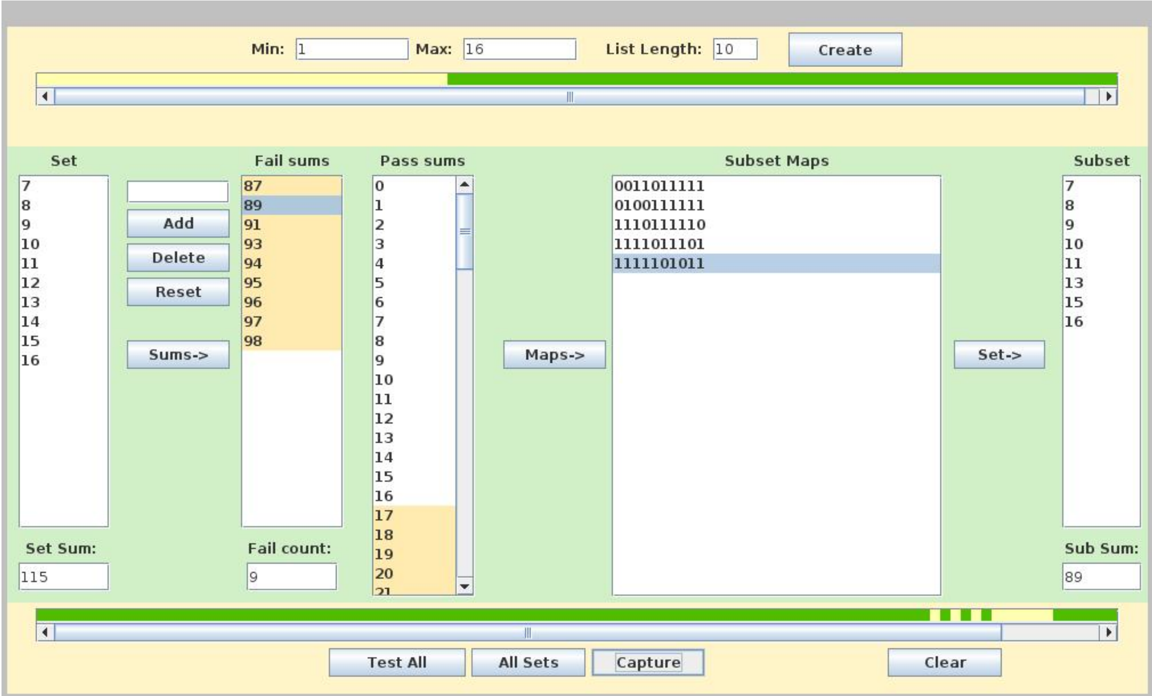

Figure 6: Testing sum sets for redundancy

While the existence of non-redundant solution sets for instances on the density threshold was predictable, the *SumFinder* program also shows that the non-redundant sets are all close to the end of the central region. This raises the possibility that the depth of $\Gamma_S(t)$ is related to the distance of t from the edge of the central region. We could define the rank of a sum to be a measure of its distance from the edges of the central region and devise additional property tests for *SumFinder* to test a new and stronger conjecture about the decision threshold. The redundancy test illustrates the value of *SumFinder* as a research tool, and it is clear that there are many more experiments to be designed in exploration of the Subset Sum problem.

4 Conclusion

The *SumFinder* program provides a useful tool for both instruction and research. As an instructional tool, it can be used to introduce a simple NP-complete problem, giving students

a concrete experience in dealing with combinatorial explosion. It can also be used to make students aware that computer science is indeed a science. We still know very little about the properties of computational problems, even for well-known classic problems in computing such as Subset Sum. We can discover new phenomena in the computational universe by conducting experiments in virtual laboratories. And we gain highly relevant and valuable software development skills in building the research tools, such as *SumFinder*, that provide those virtual laboratories.

References

[1] M. Chaimovich, G. Freiman, and Z. Galil, Solving Dense Subset-Sum Problems by Using Analytical Number Theory, *Journal of Complexity* **5** (Academic Press, 1989), pp. 271-282.

[2] T. Desell and T. O'Neil, SubsetSum@Home Project, URL *volunteer.cs.und.edu/subset_sum*, University of North Dakota (2012).

[3] A. Flaxman and B. Przydatek, Solving Medium-Density Subset Sum Problems in Expected Polynomial Time, *Lecture Notes in Computer Science* **3404**, (2005) pp. 305-314.

[4] Z. Galil and O. Margalit, An Almost linear-time Algorithm for the Dense Subset-Sum Problem, *SIAM Journal on Computing* **20:6** (1991), pp. 1157-1189.

[5] D. Knuth, *The Art of Computer Programming*, volume 4, fascicle 3, (Addison-Wesley, 2005).

[6] S. Mertens, The Easiest Hard Problem: Number Partitioning, Inst. f. Theor. Physik, University of Magdeburg, Magdeburg, Germany (2003).

[7] T. E. O'Neil, On Clustering in the Subset Sum Problem, *Proceedings of the 44th Midwest Instruction and Computing Symposium* (Duluth, MN, 2011).