On Clustering in the Subset Sum Problem

Thomas E. O'Neil
Computer Science Department
University of North Dakota
Grand Forks, ND 58202-9015
oneil@cs.und.edu

Abstract

The research literature on NP-complete problems includes some attempts to identify easier and harder problems within the class, and the results with respect to Subset Sum are not particularly conclusive. If the complexity parameter is the cardinality of the set *S*, the problem appears to be strongly exponential (as hard as any problems in the class). If the classical complexity measure of bit length of the problem input is used, however, it can be shown that Subset Sum has a sub-exponential-time algorithm. This would make it an easier problem than those that remain strongly exponential when input length is the complexity parameter.

This paper is intended to contribute evidence that Subset Sum is really an easier problem – that its apparent sub-exponential complexity is genuine and not just an artifact of an inefficient representation. We proceed by empirically examining the clustering phenomenon in the list of sums of subsets of *S*. When the set of integers *S* is dense (when the cardinality of the set is close to the maximum integer within it), we discover that almost all possible sums are covered by some subset. There appears to be a density threshold beyond which a subset with target sum *t* always exists for targets greater than the maximum value in *S* and less than the sum of all elements in *S* minus the maximum. The proof of such a result would provide the basis for an algorithm whose running time would remain sub-exponential even for a short, complement-based representation of a dense set. This, in turn, would solidify the argument that Subset Sum is truly an easier hard problem.

1 Introduction

The Subset Sum problem is described as follows: given a set of positive integers S and a target sum t, is there a subset of S whose sum is t? It is one of the NP-complete problems that is known to have a pseudo-polynomial-time solution [2]. It can be solved by a backtracking algorithm in time $O(2^n)$, where n is the size of set S, or it can be solved by dynamic programming in time $p(t)$, where $p(t)$ is a polynomial function of the target value t. Of course, the magnitude of the target value t may be an exponential function of the number of integers in the set S, and in that case the apparently polynomial-time dynamic programming solution is really exponential in n.

The current upper bound for Subset Sum is apparently $2^{O(n/2)}$ when size of the input set (denoted n) is used as the complexity parameter [8]. When the maximum value in the set (denoted m) is used as the complexity parameter, dynamic programming can be used to solve the problem in $O(m^3)$ time. While most research literature of Subset Sum employs one of these semantic parameters, there is a third choice. In classical complexity theory, the complexity measure is the bit-length of the input string. This parameter is formally determined, simply by counting the bits in the string. The advantage of using the formal measure is that it requires no semantic interpretation of the input string, and problems with vastly different semantics can by grouped together in formal complexity classes. The formal measure is usually not employed for analysis of algorithms, probably because analysis based on semantic parameters is simpler, and because the complexity classification is the same under both measures. This is true for the strong NP-complete problems such as Satisfiability or the Independent Set problem in graphs. The natural complexity parameters for Boolean expressions are the number of variables and the number of clauses. The natural complexity parameters for graphs are the number of nodes and the number of edges. For both these problems, the two optional parameters are mutually dependent. For Boolean expressions, the number of distinct clauses is bounded as a function of the number of variables. And with graphs, the number of edges in a meaningful problem instance is $O(n^2)$, where n is the number of vertices. With these problems, we can choose either of the semantic parameters and obtain an analysis similar to what we would get using input length as the parameter.

With the Subset Sum problem, however, we do not find a mutual dependence between the number of objects in the set and the maximum value. We cannot use the list length n to bound the maximum value m in meaningful problem instances. And favoring one parameter over the other gives drastically different results. The formal complexity parameter actually incorporates both the semantic parameters. The bit length of a problem instance is $O(n \cdot \log m)$, and analysis based on this measure actually yields a result that is distinct from $2^{O(n/2)}$ and $O(m^3)$. Stearns and Hunt [7] used input length x to demonstrate that an algorithm for the Partition problem (a special case of Subset Sum) exhibits sub-exponential time: $2^{O(\sqrt{x})}$. The significance of this result was probably obscured by the claim in the same paper that the Clique problem is also sub-exponential, while its dual problem Independent Set remains strongly exponential. This apparent anomaly is a representation-dependent distinction, and it disappears when a symmetric representation for the problem instance is used [4]. Sub-exponential time for Partition,

however, appears to have stronger credibility. This result was replicated explicitly for Subset Sum (using a different algorithm) in [6], and it seems unlikely that symmetric representation will make it disappear. This sets the stage for the current study, in which we gather empirical evidence that instances of Subset Sum where the input set is dense (n is $\Theta(m)$) are very easy to solve. The ultimate goal, beyond the scope of this paper, is to develop an algorithm for Subset Sum that remains sub-exponential even under the test of symmetric representation for the input set. This would solidify the argument that Subset Sum is truly an easier hard problem.

In the sections that follow we explore the relationship between the density of the input set and clustering of subset sums. As the input density increases, a threshold is encountered beyond which clustering dominates the output set, and almost all possible sums are covered by some subset. The empirical evidence suggests that the threshold occurs at density just above 50%. The experiments are conducted with the aid of the AlgoLab software [5], a virtual laboratory for empirical study of algorithms.

2 The Density Threshold

Density in the Subset Sum problem has been known to affect the expected running time in studies that date back to the early 1990's (e.g. [1]). More recently, the complexity of a specialized version of the problem with applications in cryptography has also been shown to be highly dependent on density [3]. The classification of density in those studies uses the threshold of $m = 2^n$ to distinguish high ($m < 2^n$) from low ($m > 2^n$). Dynamic programming works best for high density instances, while backtracking is better for low density instances. Here we are interested in very high density instances where n is $\Theta(m)$, since these are the only instances for which symmetric representation might affect an algorithm's time analysis.

For a set of n positive integers S with maximum value m, all possible subset sums fall within the range $\{1, ..., m(m+1)/2\}$. We will use ΣS to denote the sum of the entire set, and we observe that if S has a subset R that has sum t, it also has a subset $(S - R)$ that has sum $\Sigma S - t$. So we expect that both subset sums and missing subset sums will occur in pairs.

Our empirical study starts with a program (called *AllSums*) that allows the user to randomly generate a set S with a specified minimum, maximum, and size. The program will create the list of all sums of subsets of S. Since the focus is on dense sets where almost all target sums will be achievable, the program actually displays the complement of the output list -- the list of missing sums -- to the user. The number of missing sums closely approximates the number of clusters in the list of output sums. Figure 1 shows the results of using *AllSums* to generate a set of 6 numbers between 1 and 16. The user enters these parameters in the top panel and presses the *Create* button. A randomly generated set consistent with the parameters is displayed in the list box on the left, and a list of all missing subset sums is displayed in the list box on the right. A scrollable banded bar is displayed in the top panel representing the input set, and another bar is

displayed in the bottom panel to represent the list of sums. The light segments on the bar correspond to missing values, while the dark segments correspond to values that are present. This allows the user to visualize the degree of fragmentation or clustering in both sets. In the experiment of Figure 1, the input set contains only about one-third of the possible values, and the output list remains somewhat fragmented. Clusters of six or seven sums have formed, separated by small clusters of missing sums.

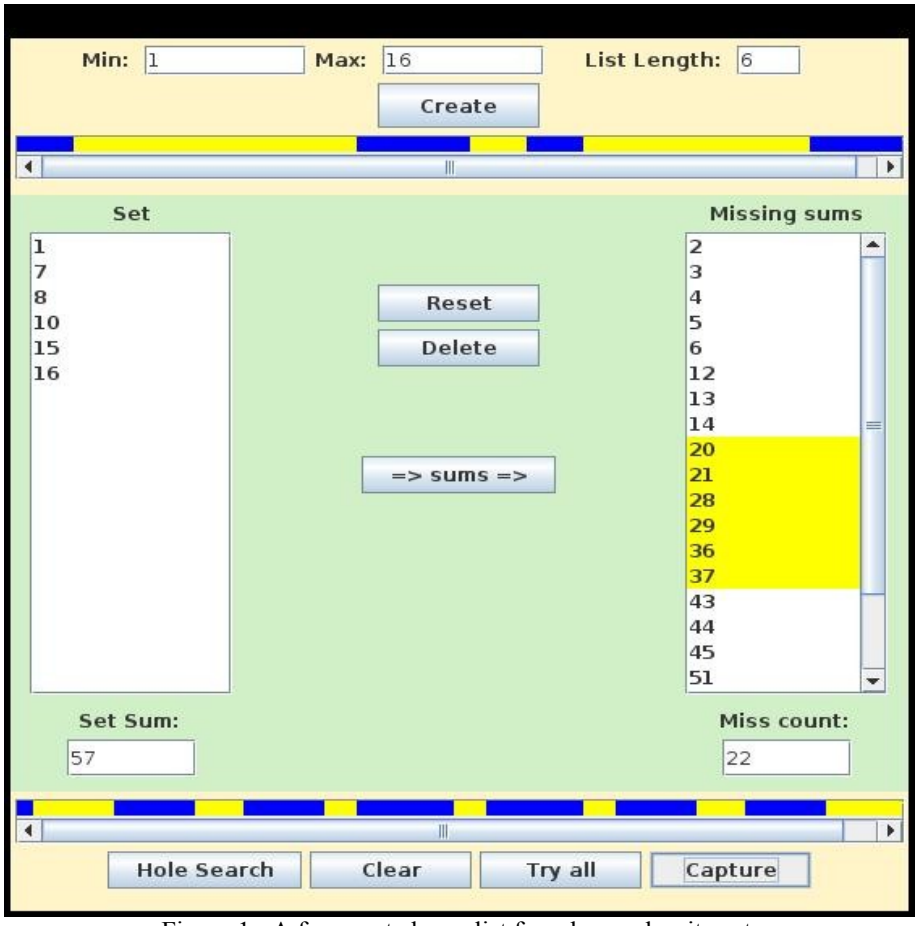

Figure 1: A fragmented sum list for a lower-density set.

Some of the sums in the missing sum list are highlighted. This distinguishes the missing sums that are greater than or equal to the maximum value m in the input set from the others. We define a sum t to be *central* if $m < t < \Sigma S - t$, and *peripheral* otherwise. Missing peripheral sums should be easy to find – they correspond to missing values in the input set. Missing central sums, on the other hand, can be hard to find. It would be a significant and useful discovery to find a density threshold beyond which no central sums are missing.

Figure 2 shows a typical result for a randomly generated set with about half the possible values in the input set. The sum list contains a large cluster that contains all central sums. The only missing sums are peripheral. Almost all sets of 8 values with a maximum of 16 show this pattern. Before we jump to the conclusion that density 0.5 is sufficient to guarantee clustering, however, we should the consider the special case of an input set that

has only even numbers. The *Allsums* program provides limited editing capabilities that allow the user to build such a set. The *Reset* button in the center panel creates a set with all values between the specified minimum and maximum, and the *Delete* button can be used to delete selected items from the input set. Figure 3 shows the missing sums for the set of all even numbers between 1 and 16. All odd sums are missing, so the output set is highly fragmented. It is interesting to note that adding a single odd number to this set

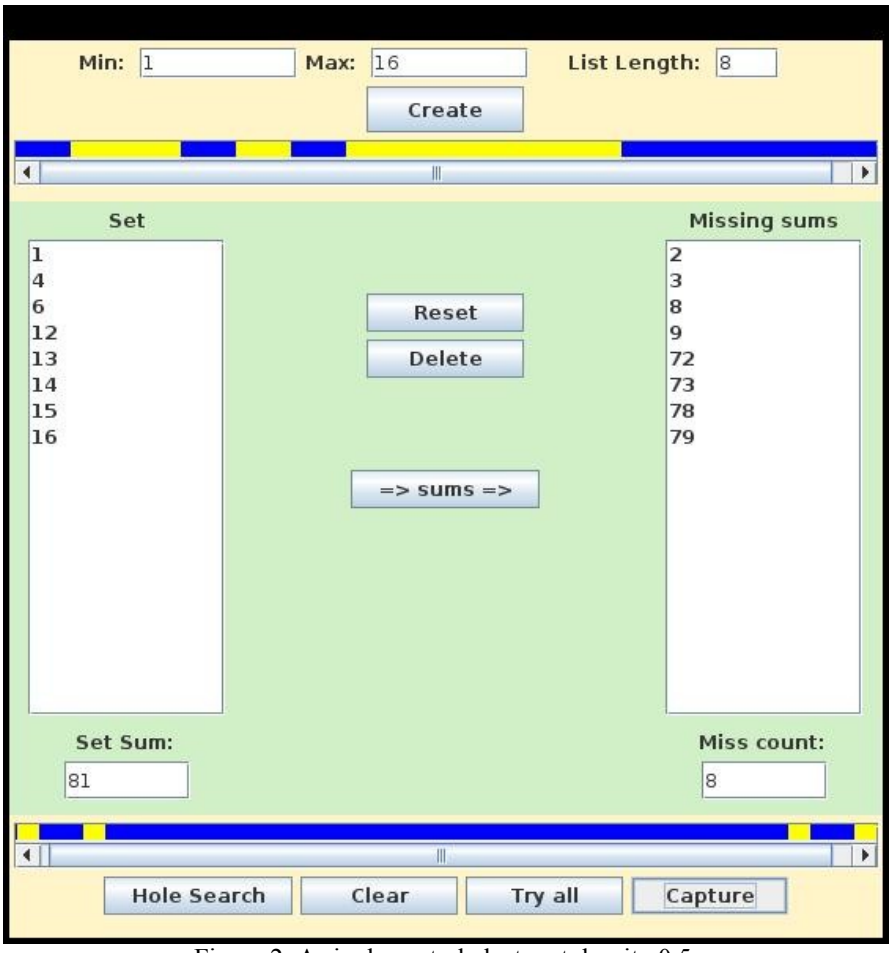

Figure 2: A single central cluster at density 0.5.

(any odd between 1 and the maximum) will fill in all central sums, creating the usual central cluster. This example heightens the expectation that clustering is a threshold phenomenon – that the degree of clustering increases rapidly over a relatively small increase in density.

The *AllSums* program has two options that support a search for an exceptional higher density set that might have missing central sums. The *Hole Search* button in the bottom panel will generate up to 10,000 problem instances until one is found that has missing central sums. The search terminates with the display of such a set, if it is found, or with the display of the 10,000th random instance that has only peripheral sums. If the *Hole Search* does not turn up any missing central sums, the user can resort to the *Try all* button. The program will then calculate all subset sums of all sets with the specified

minimum, maximum, and size. This, of course, can take a long time. The *Try all* button is recommended only for relatively small set sizes.

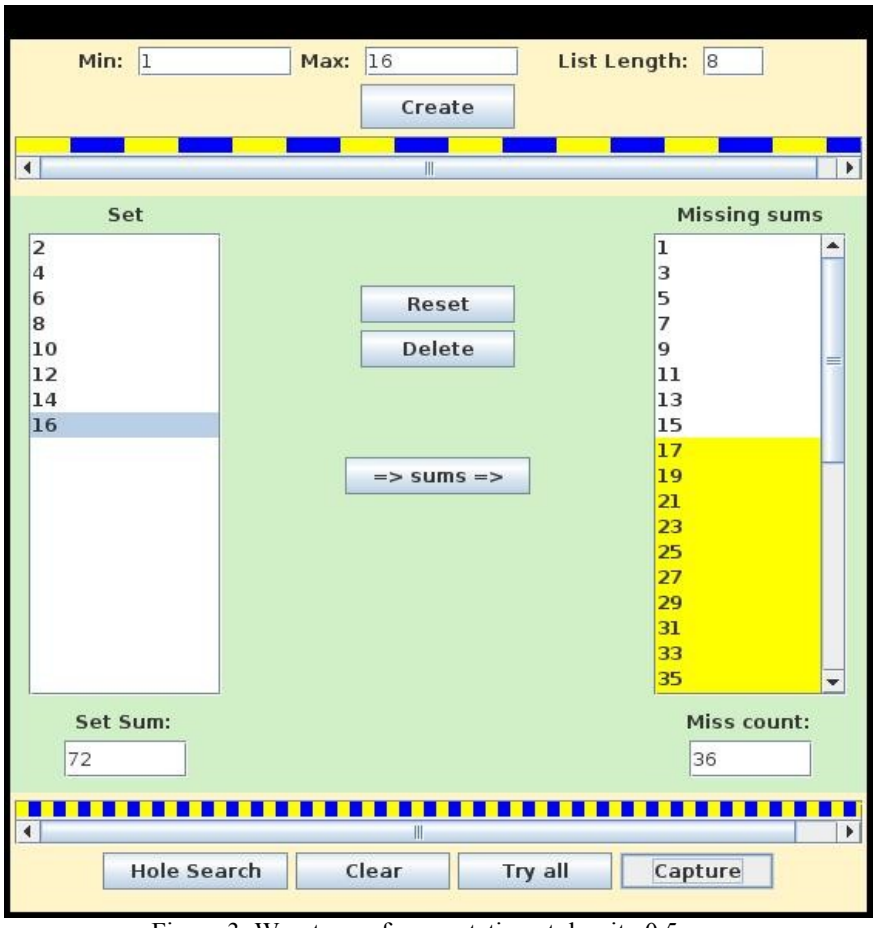

Figure 3: Worst-case fragmentation at density 0.5.

Figure 4 shows a special case that is discovered using either the *Hole Search* or *Try all* button for a set of 9 numbers between 1 and 16. If all numbers from 2 to 8 are missing from the input, then no subset adds up to 18 or 83 -- there is a pair of missing central sums. Again, adding one more number will eliminate the missing central sums. So we press the *Try all* button for set size 10, maximum 16, and discover that no instance has missing central sums. We repeat these experiments for other maximum values and find that the central cluster is complete whenever the input set size is two more than half of the maximum, and we propose the following hypothesis:

Hypothesis 1. A set of positive integers with maximum m and size > (m/2+1) has no missing central sums.

The implication of the hypothesis is that we can design a very efficient decision algorithm for instances of subset sum where the set size exceeds $m/2 + 1$. If the target sum t is a central sum, we simply answer *yes*. If the target is peripheral, we can conduct a search based only on the values missing from the input set. We expect that the complexity of

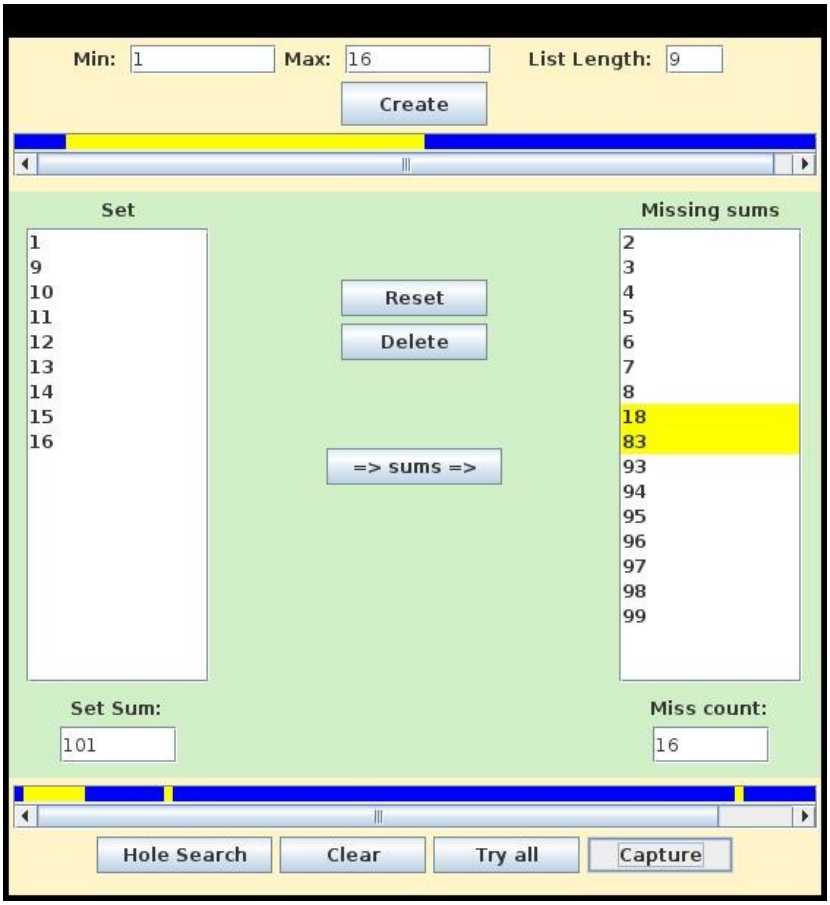

Figure 4: Missing central sums at density 9/16.

such an algorithm would remain sub-exponential, even under symmetric representation for the input set.

3 The Cluster-count Peak

The experiments in the previous section show results for individual problem instances. It is also informative to run batches of random instances and compute and display average results. The AlgoLab program [5] can be employed for this purpose. For the sake of comparison with the individual instance results described in the previous section, we begin by running batches of instances where the maximum value is 16. Figures 5 through 7 show the results of an experiment in which sets of size 1 through 16 with a maximum value of 16 (700 replications of each set size) were generated and tested. Three algorithms were run on each instance: *SumFinder*, *MissFinder*, and *BlockFinder*. These are actually three versions of the same algorithm, which creates a list containing the sums of all subsets of its input set. The result returned by *SumFinder* is the count of distinct subset sums; the result returned by *MissFinder* is the count of all missing central subset sums; and the result returned by *BlockFinder* is the count of clusters of consecutive sums on the sum list. The lists are implement using the *IntBlockList* data structure, a linked list of blocks of consecutive integers.

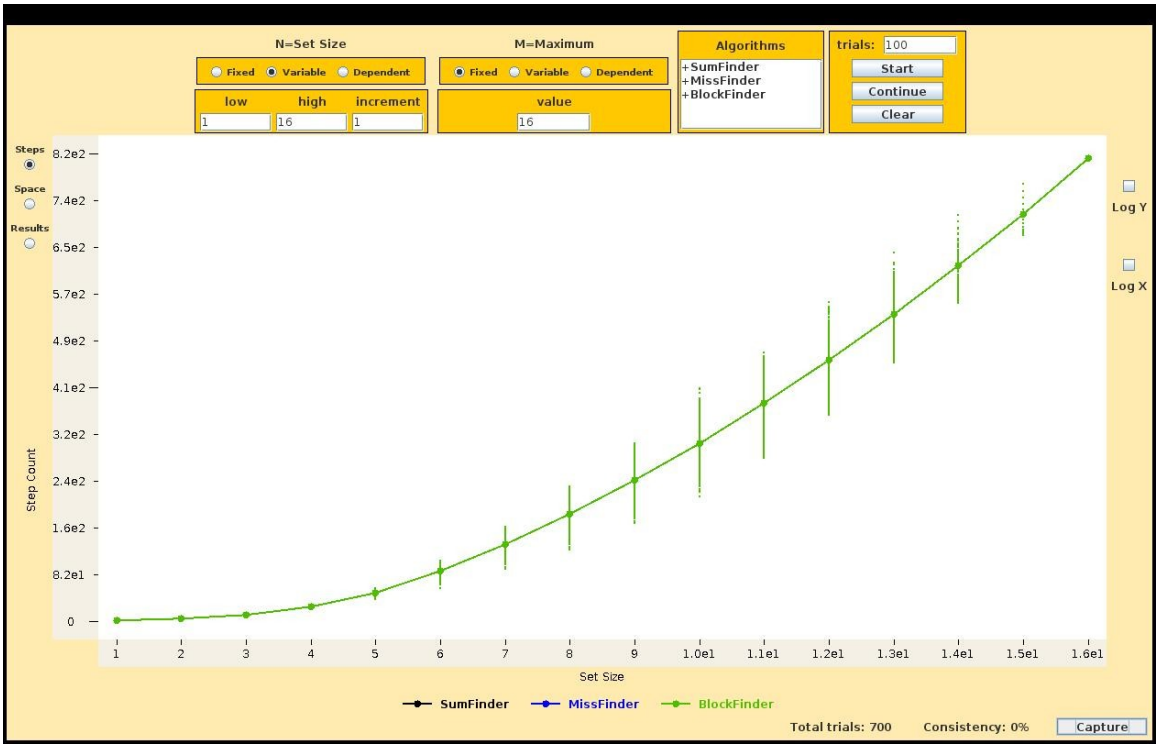

Figure 5: Step counts for calculating all subset sums of sets with 1 to 16 elements.

For each size of input set, the algorithms report a step count and a space requirement as well as a result value. The step counts and space metrics are identical for the three algorithms. The step count is the total number of iterations of the innermost loop as the

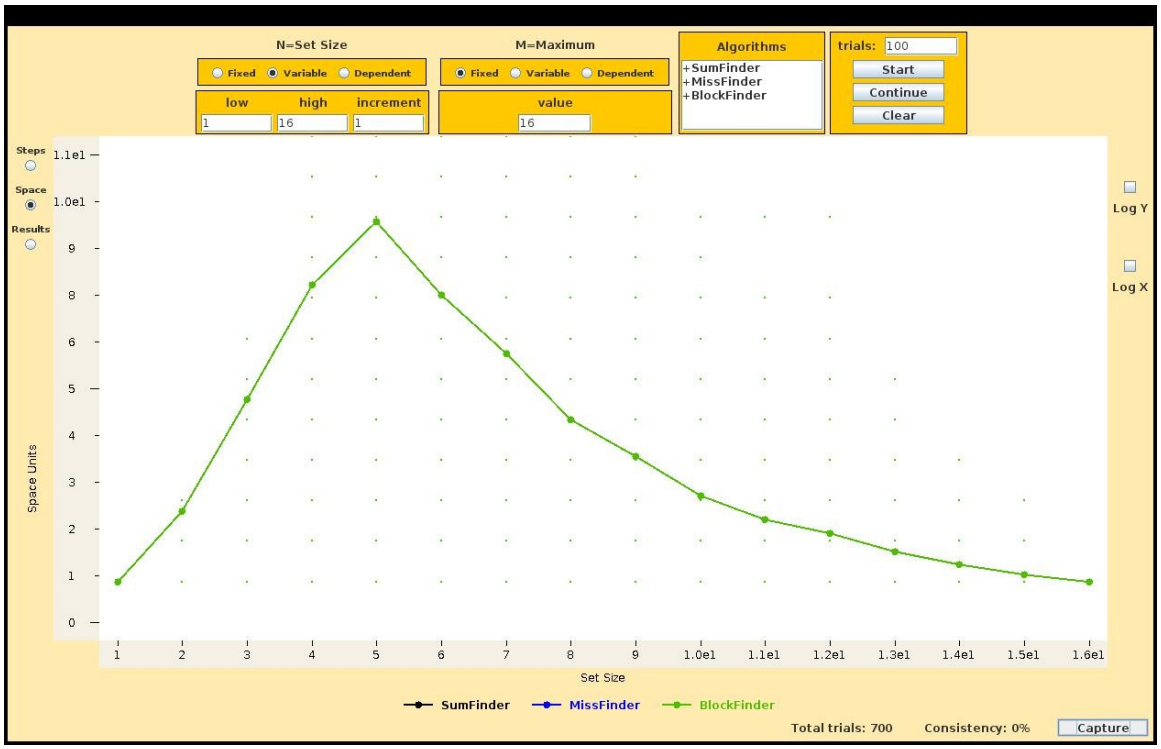

Figure 6: Space used for calculating all subset sums of sets with 1 to 16 elements.

list of all sums is generated. The space metric is the maximum size reached by the block list as the sums are computed. Figures 5 and 6 show the step counts and space metrics, respectively. The curve in Figure 5 indicates moderate polynomial growth in step counts as the input sets become larger. The curve in Figure 6 is actually very close to the curve for the BlockFinder result curve in Figure 7. This indicates that the maximum number of clusters in the sum list during execution is usually equal to the final number of clusters in the list. There is an obvious peak in the cluster count when the set size is 5.

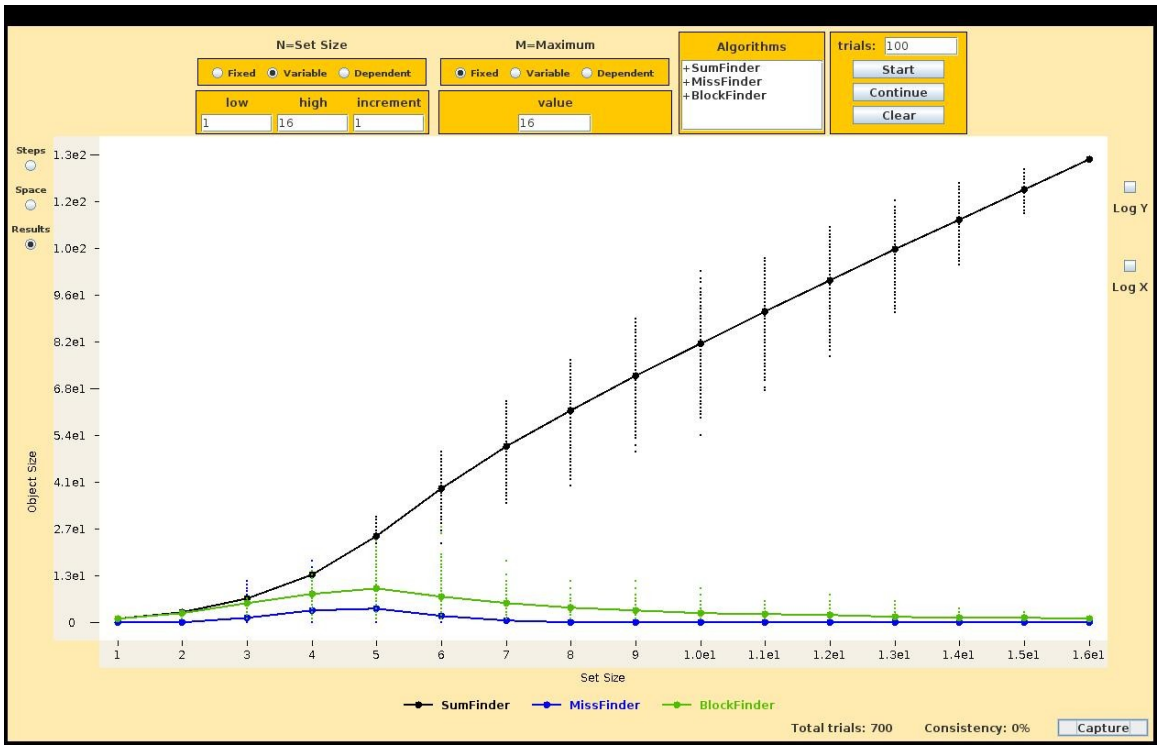

Figure 7: Comparative average sum /missing sum counts for sets with 1 to 16 elements.

Figure 7 shows the average sum counts (top curve), missing central sum counts (bottom curve), and number of sum clusters (middle curve) for each set size. The sum count curve is somewhat 'S'-shaped, culminating at 136 (the sum of all numbers from 1 to 16). The missing central sum curve is consistent with Hypothesis 1 – there are no missing sums for sets of 10 or more numbers. There are also no missing sums for sets of size 8 and 9, indicating that in 700 random instances, the special cases of Figures 3 and 4 did not occur. The final cluster count follows the missing central sum curve closely, but is slightly higher. It reflects all missing sums (including peripheral sums) as gaps between the sum clusters. Some peripheral sums will be missing unless all possible values are in the input set (for which case the cluster count is 1).

Additional *AlgoLab* experiments have confirmed that the phenomena described above persist for larger values of n and m. Figures 8 and 9 show the results of an experiment with $m = 100$ and n ranging from 1 to 25. As before, the curve for total subset sums in Figure 8 is slightly 'S'-shaped, and it appears to make linear progress as n increases toward the sum of all numbers from 1 to 100. The curves for missing central sums and

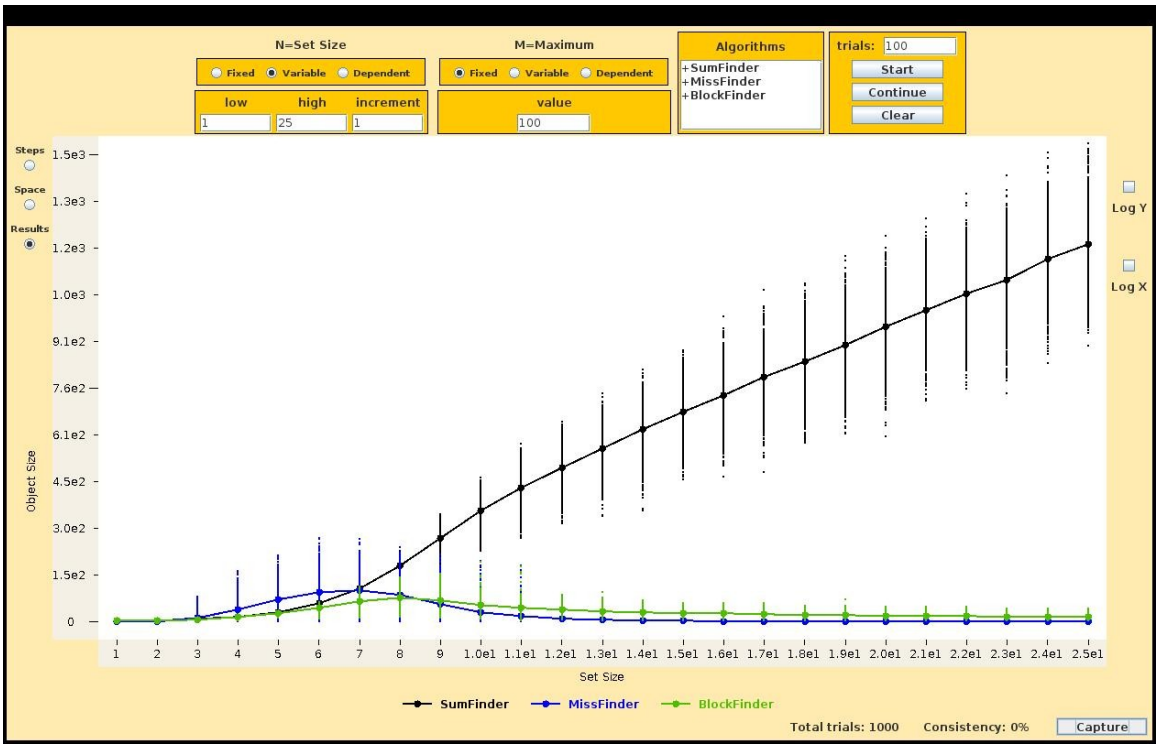

Figure 8: Comparative average sum /missing sum counts for sets with 1 to 100 elements.

number of sum clusters again are closely related, with the missing central sum count declining to zero at about $n = 16$. Both these curves reach a maximum for a small value of n, but in contrast to the previous experiment, the peaks are distinct. The missing

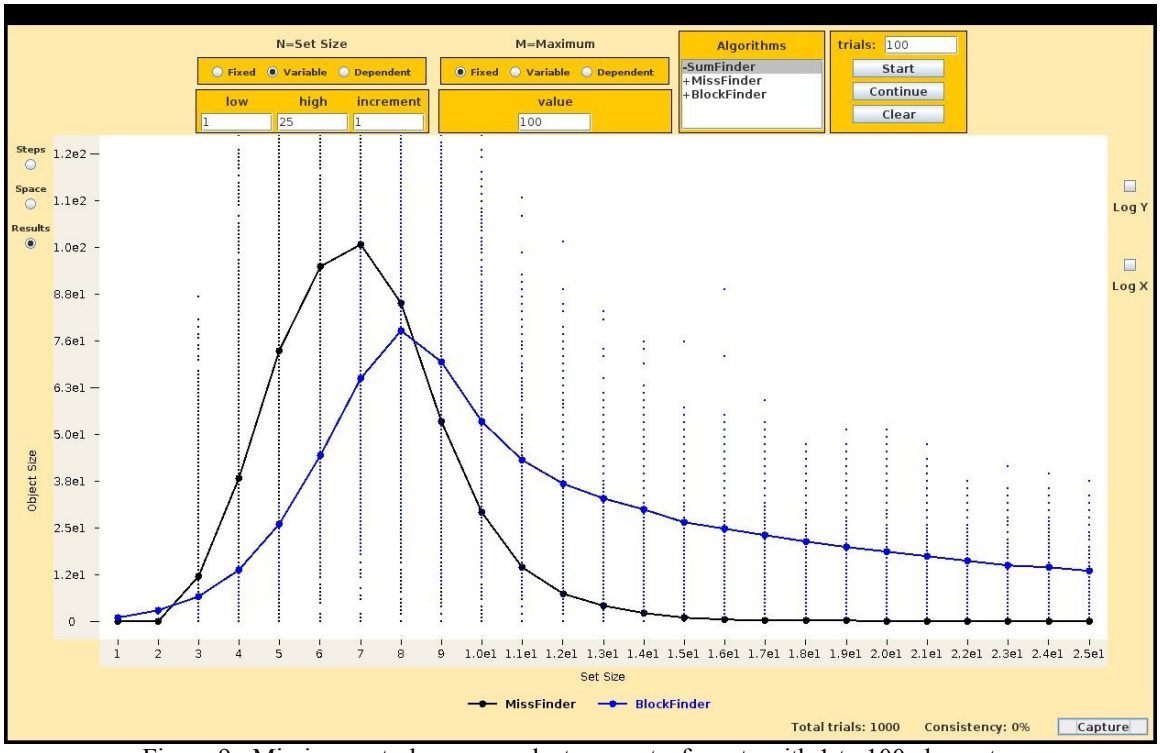

Figure 9: Missing central sums vs. cluster counts for sets with 1 to 100 elements.

central sum curve grows faster and peaks sooner than the cluster count, but it also declines faster to undercut the cluster count just beyond the cluster count peak. Figure 9 shows a closer view of this phenomenon. The missing central sum curve peaks at $n = 7$ about 25% higher than the cluster count, which peaks at $n = 8$. By $n = 9$, the missing central sum count has fallen below the cluster count. This indicates large blocks of missing sums at lower values of n.

The position of the cluster count peak (at $n = 8$) has not moved much beyond its position in the previous experiment (at $n = 5$), in spite of the fact that the maximum value m is more than 6 times larger. This indicates that the expected position of the cluster count peak is a sub-linear function of the maximum value. The results of additional experiments are summarized in the table of Figure 10. These results suggest that the expected position of the peaks is a logarithmic function of m. In fact, the observed positions of the missing central sum peak exactly match $n = \lg m$ for $m > 16$. The position of the cluster count peak moves ahead at a slightly higher rate. The logarithmic relationship between n and m for these peaks is consistent with the known critical region for the Subset Sum problem. The peaks in cluster count and missing centrals sums coincide with peaks in step counts when using the best algorithms for the problem.

Maximum set value (m)	**Value of n at cluster count peak**	**Value of n at missing central sum peak**
8	4	4
16	5	5
32	6	5
64	7	6
128	8	7
256	10	8
512	11	9
1024	12	10
2048	13	11

Figure 10: Observed peaks in cluster count and missing central sums.

4 Conclusion

The Subset Sum problem is an NP-complete problem that has a pseudo-polynomial time algorithm. It can be argued that such problems are easier than the other NP-complete problems, based on algorithms that solve the problem in sub-exponential time. This argument would be strengthened if dense instances of the problem could be solved in time that is a sub-exponential function of the size of the complement of the input set. Empirical evidence strongly suggests that this can be achieved. Even moderate-density input sets are likely to have a large central cluster of subsets sums. The cluster is likely to be present for any sets whose size exceeds the log of the maximum value m in the set.

Further, it appears that a complete central cluster is guaranteed when the set size exceeds $m/2+1$.

The empirical results set the stage for further work. It would be interesting to find or derive a mathematical model that relates the density of the input set to the probability of missing central sums. It would also be significant to prove (or find such a proof in research literature) that there are no missing central sums for sets with more than $m/2+1$ elements. And finally, an algorithm for dense sets that identifies missing sums in time that is sub-exponential in the number of missing input values would solidify the argument that Subset Sum is truly easier than other NP-complete problems.

References

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

[2] M. Garey and D. Johnson, C*omputers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman Press, San Francisco, CA (1979).

[3] V. Lyubashevsky, "On Random High Density Subset Sums," *Electronic Colloquium on Computational Complexity*, no. 7 (2005).

[4] T. E. O'Neil, "The Importance of Symmetric Representation," *Proceedings of the 2009 International Conference on Foundations of Computer Science (FCS 2009)*, pp. 115-119, CSREA Press (2009).

[5] T. E. O'Neil and S. Kerlin, "A Virtual Laboratory for Study of Algorithms," *Proceedings of the 42nd Midwest Instruction and Computing Symposium* (*MICS* '09), 7 pages, Rapid City, SD, (2009).

[6] T. E. O'Neil and S. Kerlin, "A Simple $2^{O(\sqrt{x})}$ Algorithm for Partition and Subset Sum," *Proceedings of the 2010 International Conference on Foundations of Computer Science (FCS 2010)*, pp. 55-58, CSREA Press (2010).

[7] R. Stearns and H. Hunt, "Power Indices and Easier Hard Problems", *Mathematical Systems Theory* 23, pp. 209-225 (1990).

[8] G. J. Woeginger, "Exact Algorithms for NP-Hard Problems: A Survey," *Lecture Notes in Computer Science* 2570, pp. 185-207, Springer-Verlaug, Berlin (2003).